Theses and Dissertations   1. Thesis and Dissertation Collection, all items

1991-12

Hypercube solutions for conjugate directions.

Hartman, Jonathan Edward

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/26581

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

HYPERCUBE SOLUTIONS
FOR
CONJUGATE DIRECTIONS

by

Jonathan Edward Hartman

December, 1991

Thesis Co-Advisor: William B. Gragg
Thesis Co-Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		Approved for public release; distribution is unlimited.		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Computer Science Department Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS		

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

HYPERCUBE SOLUTIONS FOR CONJUGATE DIRECTIONS

12. PERSONAL AUTHOR(S)

Hartman, Jonathan Edward

13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM 06/89 TO 12/91	14. DATE OF REPORT (Year, Month, Day) December 1991	15. PAGE COUNT 380

16. SUPPLEMENTARY NOTATION   The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB–GROUP	Conjugate Gradients, Gaussian Elimination, Gauss Factorization, Hyper-
			cube, iPSC/2, Matrices, Multiprocessors, Transputers, Communicating
			Sequential Processes, Message Passing, Distributed Memory, MIMD.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

As computing machines advance, new fields are explored and old ones are expanded. This thesis considers parallel solutions to several well–known problems from numerical linear algebra, including Gauss Factorization and the method of Conjugate Gradients. The Gauss algorithm was implemented on two parallel machines: an Intel iPSC/2, and a network of INMOS T–800 transputers. Interprocessor communication—in both cases—was borne by a hypercube interconnection topology.

The results reveal general findings from parallel computing and more specific data and information concerning the systems and algorithms that were employed. Communication is timed and the results are analyzed, showing typical features of a message passing system. System performance is illustrated by results from the Gauss codes. The use of two different pivoting strategies shows the potential and the limitations of a parallel machine. The iPSC/2 and transputer systems both show excellent parallel performance when solving large, dense, unstructured systems. Differences, advantages, and disadvantages of these two systems are examined and expectations for current and future machines are discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Uno R. Kodres	22b. TELEPHONE (Include Area Code) (408) 646-2197	22c. OFFICE SYMBOL CSKr

DD FORM 1473, 84 MAR
83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

HYPERCUBE SOLUTIONS
for
CONJUGATE DIRECTIONS

by

Jonathan Edward Hartman
Captain, United States Marine Corps
B.S., United States Naval Academy, 1984

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE
MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL

December, 1991

ABSTRACT

As computing machines advance, new fields are explored and old ones are expanded. This thesis considers parallel solutions to several well–known problems from numerical linear algebra, including Gauss Factorization and the method of Conjugate Gradients. The Gauss algorithm was implemented on two parallel machines: an Intel iPSC/2, and a network of INMOS T–800 transputers. Interprocessor communication—in both cases—was borne by a hypercube interconnection topology.

The results reveal general findings from parallel computing and more specific data and information concerning the systems and algorithms that were employed. Communication is timed and the results are analyzed, showing typical features of a message passing system. System performance is illustrated by results from the Gauss codes. The use of two different pivoting strategies shows the potential and the limitations of a parallel machine. The iPSC/2 and transputer systems both show excellent parallel performance when solving large, dense, unstructured systems. Differences, advantages, and disadvantages of these two systems are examined and expectations for current and future machines are discussed.

THESIS DISCLAIMER

The computer programs developed in this research have not been exercised for all cases of interest. Every reasonable effort has been made to eliminate computational and logical errors, but the programs should not be considered fully verified. Any application of these programs without additional verification is at the user's risk. A reasonable effort has been put forth to make the code efficient. Optimization has been suppressed, however, in areas where it would jeopardize the simplicity and clarity of the algorithm without great reward in terms of performance.

IMS, **inmos**, and **occam** are trademarks of INMOS Limited, a member of the SGS–THOMSON Microelectronics Group. INTEL, int_el, and iPSC are trademarks of Intel Corporation. IBM, PC AT, and PC XT are registered trademarks of International Business Machines Corporation. CIO, LD–ONE, LD–NET, TASM, TCX, TIO, TLIB, and TLNK are trademarks of Logical Systems. MS–DOS is a trademark of Microsoft Corporation. MATLAB is a trademark of The MathWorks, Inc. All other brand and product names are trademarks or registered trademarks of their respective companies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

TABLE OF SYMBOLS

The Greek Alphabet			
Lower Case		**Upper**	
Normal	Variant	Case	Name
α		A	Alpha
β		B	Beta
γ		Γ	Gamma
δ		Δ	Delta
ϵ	ε	E	Epsilon
ζ		Z	Zeta
η		H	Eta
θ	ϑ	Θ	Theta
ι		I	Iota
κ		K	Kappa
λ		Λ	Lambda
μ		M	Mu
ν		N	Nu
ξ		Ξ	Xi
o		O	Omicron
π	ϖ	Π	Pi
ρ	ϱ	P	Rho
σ	ς	Σ	Sigma
τ		T	Tau
υ		Υ	Upsilon
ϕ	φ	Φ	Phi
χ		X	Chi
ψ		Ψ	Psi
ω		Ω	Omega

ACKNOWLEDGMENTS

I. PREFACE

The need for *speed accompanied by reliability* has driven many advances in machine design. The history of computing is replete with examples—many from scientific fields—where necessity became the impetus for faster, more reliable machinery. Without exception, history and past designs have played key roles in the invention of new equipment. The maturity of mechanical calculator design was foundational in the construction of electronic computers. Today's multiprocessor computers are extensions of uniprocessor machines and include technology developed by our telephone industry. Many well–worn tools and lessons from the past can be applied. Many new ideas must be put to the test. This thesis is about applying old principles and evaluating new tools and equipment.

A. A SURVEY OF COMPUTING MACHINERY

> *Nothing is more important than to see the sources of invention, which are,*
> *in my opinion, more interesting than the inventions themselves.*
>
> — *GOTTFRIED WILHELM LEIBNIZ (1646-1716)*

1. Beginnings

The history of mathematics and computing is as old as civilization. Tools like the abacus have been used to simplify arithmetic problems. Wilhelm Schickhard (1592-1635), Blaise Pascal (1623-1662), and Gottfried Wilhelm Leibniz designed and built *mechanical, gear–driven* calculators. The latest of these was essentially a four-function calculator. By the mid–1800s, Charles Babbage had designed his *Difference Engine* and proceeded to the more advanced *Analytical Engine*. These machines were

1

never completed (at least not to the grand scale that Babbage planned), but the basic design of the Analytical Engine lies at the heart of any modern computer. Consider his motivation.

> *The following example was frequently cited by Charles Babbage (1792–1871) to justify the construction of his first computing machine, the Difference Engine [Ref. 1]. In 1794 a project was begun by the French government under the direction of Baron Gaspard de Prony (1755–1839) to compute entirely by hand an enormous set of mathematical tables. Among the tables constructed were the logarithms of the natural numbers from 1 to 200,000 calculated to 19 decimal places. Comparable tables were constructed for the natural sines and tangents, their logarithms, and the logarithms of the ratios of the sines and tangents to their arcs. The entire project took about 2 years to complete and employed from 70 to 100 people. The mathematical abilities of most of the people involved were limited to addition and subtraction. A small group of skilled mathematicians provided them with their instructions. To minimize errors, each number was calculated twice by two independent human calculators and the results were compared. The final set of tables occupied 17 large folio volumes (which were never published, however). The table of logarithms of the natural numbers alone was estimated to contain about 8 million digits.*

This quote, from Hayes [Ref. 2: p. 1], helps to explain why computers exist and shows some of the incentive for making them better. Computing machinery is designed for **speed** and **reliability**. A computer's "performance" should be measured against *both* of these components. Speed normally receives the most attention. Reliability, by whatever label you choose to give it, rarely receives due (and/or timely) attention. Too often errors and issues of correctness receive careful consideration in reactive—not proactive—situations. Kahan says, "The Fast drives out the Slow even if the Fast is wrong" [Ref. 3: p. 596].

The correctness side of performance is a much tougher game; and reliability can be a fairly subjective matter. Often we pursue solutions that are "good enough" (and this cannot always be defined). Time, on the other hand, has well–defined units and the standards for measuring time enjoy a history as old as the first sunrise. The ease with which the programmer can access the machine's clock makes measurements of this side of performance somewhat easier.
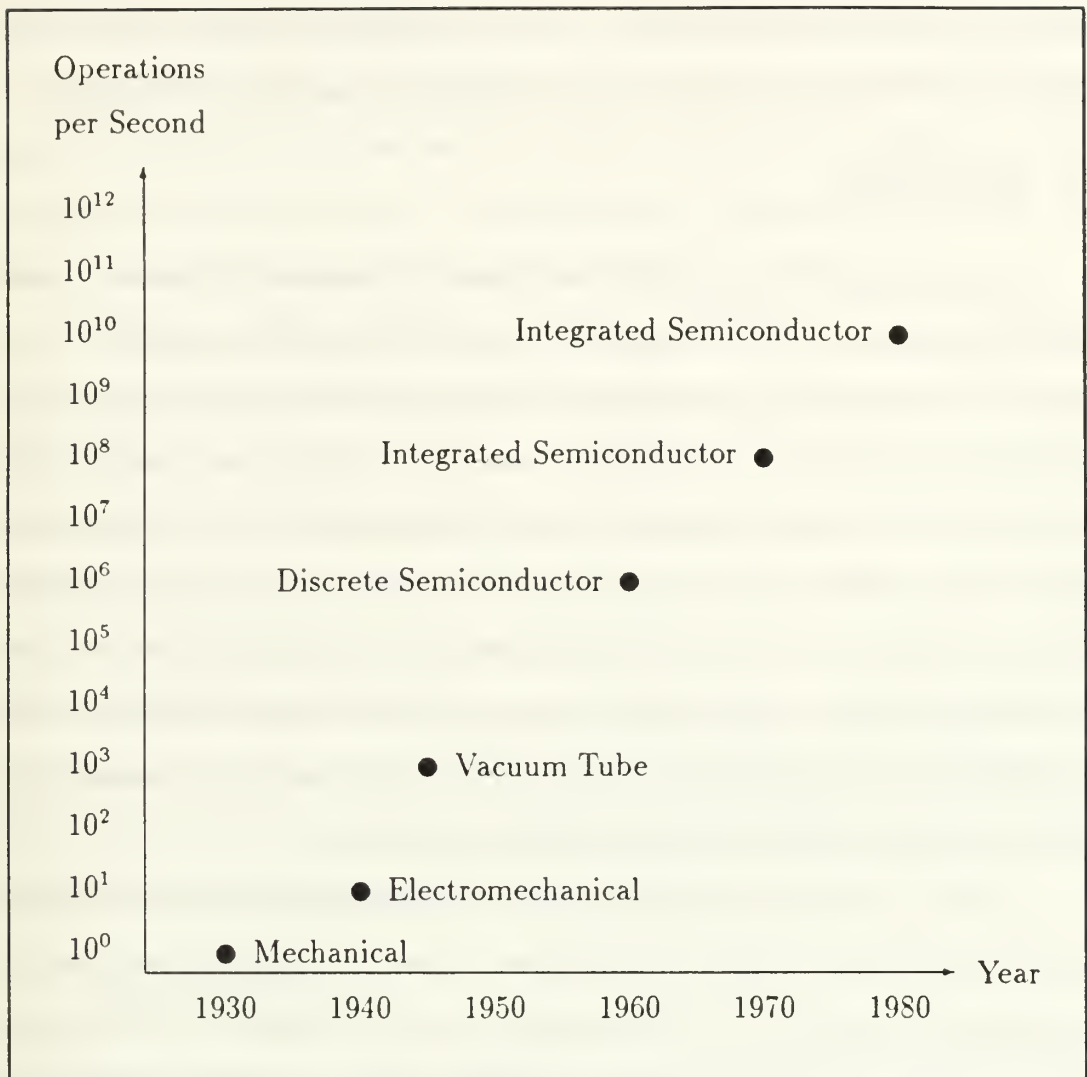
Figure 1.1: Technologies and Computing Speed

Industry demands fast machines because "time is money" and speed alone can make difficult, time–consuming problems tolerable. Without doubt, the speed of a processor and execution time *are* important performance considerations. But speed is partly dependent upon technology. Babbage's designs represented quite an advance, but they could not be realized in his day. Technology can determine which designs succeed, and to what extent. Figure 1.1 compares several recent technologies using speed (measured in operations per second) as the yardstick. The data for this

illustration was taken from Hayes [Ref. 2: p. 9]. As the figure indicates, it was nearly a century after Babbage's work when major technological advances came about.

2. Electricity

Significant gains in speed were made possible when electricity could be used in computer engineering. The United States census of 1890 employed punched cards that were read using electricity and light. Herman Hollerith (1860–1929), the designer of these cards, formed a company that would later join others and (in 1924) take on the name International Business Machines Corporation. Punched paper tape was later used by IBM in the Harvard Mark I, a general–purpose electromechanical computer designed by Howard Aiken (1900–1973). In the late 1930s, at Iowa State University, John V. Atanasoff was creating a special–purpose machine to solve systems of linear equations. He is credited with "the first attempt to construct an electronic computer using vacuum tubes" [Ref. 2: p. 16].

In 1943, J. Presper Eckert and John W. Mauchly began work—at the University of Pennsylvania—to direct the creation of "the first widely known general–purpose electronic computer". The Electronic Numerical Integrator and Calculator (ENIAC) project was funded by the U. S. Army Ordnance Department. The 30–ton machine was completed in 1946. It held more than 18,000 vacuum tubes. It could perform a ten–digit multiplication in three milliseconds, three orders of magnitude faster than the Harvard Mark I. [Ref. 2: pp. 17–18]

3. First Generation Computers

From Babbage's Analytical Engine to ENIAC, computer architectures held data and programs in separate memories. In 1945, John von Neumann (1903–1957) proposed the *stored–program concept* (i.e., programs and data could be stored in the same memory unit). The Hungarian–born mathematician's involvement in the

ENIAC project is not remembered by many, but the "von Neumann architecture" has become commonplace. In fact, it "has become synonymous with any computer of conventional design independent of its date of introduction" [Ref. 2 : p. 31]. Hennessy and Patterson [Ref. 3 : pp. 23–24] object to the widespread use of this term, claiming that Eckert and Mauchly deserved more of the credit.

In 1946, von Neumann (and others) began to design such an architecture at the Institute for Advanced Studies (IAS), Princeton. This machine, now called the IAS computer, is representative of so–called *first–generation computers* (as Hayes points out: "a somewhat short–sighted view of computer history"). The IAS machine was roughly ten times faster than ENIAC [Ref. 3 : p. 24]. During the 1946–1948 timeframe, A. W. Burks, H. H. Goldstine, and John von Neumann wrote a series of reports describing the IAS design and programming. The advances and refinements in computer design that came out of this period were important and lasting. By 1950, von Neumann and his colleagues had formed a foundation of theory and design worthy of advanced technology. [Ref. 2: pp. 19–20]

4. Transistors

The change from vacuum tube to transistor technology marked the beginning of the "second–generation" of computers (approximately 1955–1964). Transistor technology provided faster switching elements, but this was not the only change of the decade. Many of the plans of the late forties and early fifties involved memory, so it was fitting that ferrite cores and magnetic drums be used for faster main memories. Changes such as these led Hennessy and Patterson to conclude that "cheaper computers" were the principal new product of the early 1960s [Ref. 3: p. 26].

Additionally, machines began to become more sophisticated. The space and tasks of the central processing unit (CPU) and main memories were decentralized with the advent of special-purpose processors to augment the CPU and special–

purpose memories (e.g., registers) to augment the main memory. Finally, system software was becoming a greater issue. Programming continued moving upward, away from the machine level, and the processing of batch jobs was becoming more automated. [Ref. 2: pp. 31–32]

5. Integrated Circuits

The first integrated circuit (IC) was introduced in 1961 [Ref. 4: p. 1], and the use of ICs would be among the most significant advances evident in third-generation computers (starting about 1965). Integrated circuits brought major changes in cost, maintenance, reliability, and the amount of real estate required. Other than these hardware improvements (circuits and memory), third-generation computing was not easy to distinguish from that of the second generation. There was some migration from hardware to software (e.g., microprogramming), more specialized and compartmentalized CPUs (e.g., pipelining), and system software continued to advance (e.g., operating systems that could support multiprogramming through "time-slicing"). [Ref. 2: p. 40]

6. Instruction Set Trade–Offs

A large part of designing computer hardware and software involves analysis of cost–performance ratios. Other than genuine advances in design or technology, almost every aspect of computer architecture involves trade–offs. There is usually a spectrum of options from which the computer architect chooses, and the "best" solutions are not always found near the ends of the spectrum. Performance can rarely be optimized with respect to both *space* and *time*, so a balance must be sought. This space–time conflict and others appear when a designer must select a sophisticated instruction set, or a very simple one, or one of the many options along the spectrum between these options.

In the late 1970s and early 1980s both hardware and software became progressively more sophisticated. Instructions became longer and more complex. The Complex Instruction Set Computer (CISC) was popular. This design has the advantage of powerful instructions, but the machine must decode each instruction (it is a binary code). The decoding process favors brevity because longer instructions require more levels of decoding circuitry. Nonetheless, if the longer instructions could carry enough meaning, the decoding endeavor would be justified.

IBM researchers uncovered a provocative statistic—20% of the instruction set was carrying 80% of the burden [Ref. 5: p. 5]. The instruction set had become too complex. With some help from several researchers and IBM, the Reduced Instruction Set Computer (RISC) architecture became popular. RISC machines admit a smaller vocabulary, but claim quicker comprehension. In fact, the goal of the RISC architectures is one–cycle execution of the instructions [Ref. 5: pp. 6–7]. Hennessy and Patterson, both key contributors to the RISC movement, give an indication of the current broad acceptance of the RISC architecture [Ref. 3: p. 190]:

> *Prior to the RISC architecture movement, the major trend had been highly microcoded architectures aimed at reducing the semantic gap. DEC, with the VAX, and Intel, with the iAPX 432, were among the leaders in this approach. In 1989, DEC and Intel both announced RISC products—the DECstation 3100 (based on the MIPS Computer Systems R2000) and the Intel i860, a new RISC microprocessor. With these announcements, RISC technology has achieved very broad acceptance. In 1990 it is hard to find a computer company without a RISC product either shipping or in active development.*

Three major research projects were central to early RISC developments. The first—the IBM 801—began in the late 1970s, under the direction of John Cocke. In 1980, David Patterson and his colleagues at the University of California at Berkeley began the RISC–I and RISC–II projects for which the architecture is named. Finally, John Hennessy and others at Stanford University "published a description of the MIPS machine" in 1981. [Ref. 3: p. 189]

7. Multiprocessors and Multicomputers

The most recent advances in the design of computing machinery include parallel and concurrent architectures. The terminology associated with these machines has been developing for about twenty-five years, but it is still immature. The terms "multiprocessor" and "multicomputer", for instance, are sometimes used with additional meaning. C. Gordon Bell proposes that an *MIMD machine with message passing* and *no shared memory* be called a **multicomputer**. He calls a *shared-memory MIMD machine* a **multiprocessor** [Ref. 6: p. 1092]. This terminology seems to be on the way to acceptance, and it seems useful in giving a general characterization to many systems, but it lacks the sort of precision that may be necessary.

First, the word "computer" usually carries many expectations with it. From a computer, we expect things like input and output facilities, peripheral devices, and so on. These are things that a node on a typical "multicomputer" does not always possess. A "processor" is just the opposite. It might be just about any sort of processor and we are cautious about attaching any expectations to the term. Many processors are special-purpose machines, but (more substantial) central processing units and arithmetic logic units are also numbered among processors. The terms "computer" and "processor" are not precise.

Secondly, by automatically associating Flynn's taxonomy, memory models (e.g., shared, distributed), and other things with a terminology, we reduce their importance and hide them behind the term. By using the term "multicomputer", without careful definition up front, we run the risk of forgetting that we are talking about an MIMD machine that uses message passing and has no shared memory. Additionally, this terminology—packed with expectations—ignores an entire spectrum of very real possibilities. Are we saying that a machine cannot employ a *combination*

of shared and distributed memory? Using this terminology, how would we say that the memory available to each node of a given system was 30 percent shared and 70 percent local (distributed)?

Nevertheless, the terms have some use, provided we don't expect too much of them. After all, we distinguish cars from trucks in everyday conversation with reasonably little confusion. But—in the same way that it is not prudent to assume that "car" implies a vehicle equipped with a V–8 engine and four doors—we should be careful to guard against packing too many specifics and expectations into the terms "multiprocessor" and "multicomputer." For this reason, the terms multiprocessor and multicomputer are used almost interchangeably in this work. A conscious effort is made to support them with a clear description of the memory paradigm, communications facilities, and so on.

Bell's terminology identifies the systems used in this work (iPSC/2 and transputer networks) as multicomputers. Nevertheless, I often use the term "multiprocessor" to identify a system with more than one processor (such as the ones described in Chapter V and Appendix B). That is, multiprocessor means nothing more than the expected combination of "multi" with "processor." To forestall confusion, the rest of the thesis pertains to distributed memory machines that use message passing to communicate instructions and data between nodes.

8. Uniprocessors and Multiprocessors

At the chip level, multiprocessor systems resemble their single–processor predecessors. Experience (e.g., telephone industry, electronic technology) and a foundation of theory and design (e.g., von Neumann's work, network theory) are distinct benefits in the development of equipment and techniques for distributed and parallel computing. From a system perspective, though, the *concurrent* use of more than one processor creates a fundamentally different environment.

Uniprocessor systems differ substantially from multiprocessors and multi-computers in their ability to access data without competition. In the presence of more than one processor—regardless of memory model—there is a need to coordinate requests for data. This means that the multicomputer must accommodate interprocessor communications. The nodes of a multiprocessor system must work together efficiently to justify the cost of the resulting system. Some parts of the solution are relatively mature, but a vast territory—algorithms, electronic components, media for communication, and software engineering techniques—begs further exploration.

B. CURRENT APPROACHES

1. Machines

To compare the capabilities of different machines, some method of *benchmarking* is typically used. By timing the execution of a certain program(s) on a given machine we can determine its performance for the given problem. By comparing the execution times for the same problem(s) on different machines, we arrive at a notion of their relative power. A popular method for sizing up the computing power of a machine is the LINPACK benchmarking program [Ref. 7]. This is essentially a program involving the solution of a dense system of linear equations.

Currently, under this LINPACK test, the fastest machines in the world have surpassed the gigaflop mark (a billion floating-point operations per second). Table 1.1, adapted from Dongarra's report [Ref. 8: p. 21], shows performance data. The leftmost column of this table gives the name of the system and the cycle time (in parentheses). The next column contains p, the number of processors used to obtain the data that is shown in the four remaining columns. For most systems (e.g., the Intel iPSC/860) the size of the system (number of processors used for a given run) can be scaled, so data was reported for several different system sizes.

TABLE 1.1: WORLD'S FASTEST COMPUTERS

Computer (Clock Rate)	p	r_{max}	n_{max}	$n_{1/2}$	r_{peak}
Intel Delta (40 MHz)	512	11.9	25000	7000	20
Thinking Machines CM–200 (10 MHz)	2048	9.0	28672	11264	20
Intel Delta (40 MHz)	256	5.9	18000	5000	10
Thinking Machines CM–2 (7 MHz)	2048	5.2	26624	11000	14
Intel Delta (40 MHz)	192	4.0	12000	4000	7.7
Intel Delta (40 MHz)	128	3.0	12500	3500	5
Intel iPSC/860 (40 MHz)	128	1.9	8600	3000	5
nCUBE 2 (20 MHz)	1024	1.9	21376	3193	2.4
Intel Delta (40 MHz)	64	1.5	8000	3000	2.6
nCUBE 2 (20 MHz)	512	.958	15200	2240	1.2
Intel iPSC/860 (40 MHz)	64	.928	5750	2500	2.6
Fujitsu AP1000	512	2.251	25600	2500	2.8
Intel iPSC/860 (40 MHz)	32	.486	4000	1500	1.3
nCUBE 2 (20 MHz)	256	.482	10784	1504	.64
MasPar MP–1 (80 ns)	16384	.44	5504	1180	.58
Fujitsu AP1000	256	1.162	18000	1600	1.4
Intel iPSC/860 (40 MHz)	16	.258	3000	1000	.64
nCUBE 2 (20 MHz)	128	.242	7776	1050	.32
Fujitsu AP1000	128	.566	12800	1100	.71
Intel iPSC/860 (40 MHz)	8	.132	2000	600	.32
nCUBE 2 (20 MHz)	64	.121	5472	701	.15
Fujitsu AP1000	64	.291	10000	648	.36
Intel iPSC/860 (40 MHz)	4	.061	1000	400	.16
nCUBE 2 (20 MHz)	32	.0611	3888	486	.075
Intel iPSC/860 (40 MHz)	2	.044	1000	400	.08
nCUBE 2 (20 MHz)	16	.0320	5580	342	.038
Intel iPSC/860 (40 MHz)	1	.024	750		.04
nCUBE 2 (20 MHz)	8	.0161	3960	241	.019
nCUBE 2 (20 MHz)	4	.0080	2760	143	.0094
nCUBE 2 (20 MHz)	8	.0040	1280	94	.0047
nCUBE 2 (20 MHz)	8	.0020	1280	51	.0024

The column labeled r_{max} gives the performance (in gigaflops) for the largest problem run on the machine. The size of that largest problem is indicated by n_{max}, where n is the dimension of the matrix of coefficients, $A \in \Re^{n \times n}$. The $n_{1/2}$ column gives the problem size that yielded a rate of execution that was half of r_{max}. Finally, r_{peak} denotes the theoretical peak performance (in gigaflops) for the machine.

This data indicates that Intel is the current leader—among companies in the United States—of the teraflop race, so we shall take a closer look at their products. The Intel i860 microprocessor, together with 8 megabytes of memory, forms one of 128 nodes in the hypercube-connected iPSC/860. This machine achieves performances of nearly two gigaflops with LINPACK. iPSC stands for int$_e$l Personal SuperComputer, so this entry would not appear to target high–end markets. The most significant project in supercomputing at Intel today is the Touchstone project.

George E. Brown, chairman of the U. S. House Committee on Science, Space, and Technology, cut the ribbon around the Intel Touchstone Delta at the California Institute of Technology on May 31, 1991 [Ref. 9 : p. 96]. The Delta is a mesh of 528 nodes. Each node holds an i860 processor and 16 megabytes of memory. This machine has reached the 11.9 gigaflop mark with the LINPACK benchmark. The closest competitor in the world would appear to be the CM–200 from Thinking Machines, Inc. This 2,048-node machine benchmarks at 9 gigaflops [Ref. 8: p. 21]. The Touchstone program is not over. Intel plans to follow the Delta with the Touchstone Sigma. Sigma will have at least 2,048 nodes, each consisting of the i860 XP processor (about twice as powerful as the i860). [Ref. 9: p. 96]

The European high–performance computing market favors the transputer, a microprocessor made by INMOS. The *New York Times* of May 31, 1991 lists one German company, Parsytec, and seven American companies—Bolt, Beranek, and Newman (BBN), Cray Research, IBM, Intel, NCube, Thinking Machines, and Tera Computer—that have entered the teraflop race [Ref. 10]. Parsytec expects their GC

to provide "the necessary 2 to 3 orders of magnitude increase in performance above existing supercomputers to give scientists the tool to attack their *Grand Challenges.*" [Ref. 10: p. 1]

Parsytec envisions a system of up to 16,384 processing elements based upon the INMOS T9000 transputer (see Chapter VII). This would give the Parsytec machine 25–megaflop nodes capable of communications bandwidths near 100 megabytes per second. The Parsytec design begins with a *cluster* of seventeen T9000 processors (sixteen primary processors and the seventeenth for backup) and four C104 wormhole routing chips. From four clusters, the company will craft a *GigaCube* (or simply Cube) of 64 processors (not counting redundant elements in the design). The GC–1 would represent a one gigaflop system and this would be the building block for greater systems (lesser systems can initially be equipped with 16, 32, or 48 nodes). The processors in a single *(Giga)Cube* are arranged in a three–dimensional ($4 \times 4 \times 4$) grid. [Ref. 10]

2. Programming Practice

Software engineering for multiprocessor systems is similar to contemporary practices for sequential machines. The programming languages used in this work provide normal C libraries with additional functions to accommodate interprocessor communications. The systems typically provide a loader designed to load executable code onto the (host and) nodes according to the programmer's instructions. Some loaders require that the same code be loaded onto each of the nodes. Other, more flexible, loaders allow the user to specify which program should be loaded onto each node. The Logical Systems C network loader, LD–NET is such a program. It takes a *Network Information File* (NIF), describing the network's interconnections and loading instructions, as input and performs the loading process.

C. THE FUTURE

1. Crossroads

Parallel and distributed computing is in the early years of a very promising lifetime. We should give careful consideration to the direction that the field should assume. Lacking years of experience, I will lean on the writings and advice of others while trying to peer a little ways into the future of parallel computing. A regrettable side effect of this decision is that this section seems to consist primarily of the observations and opinions of others. Notwithstanding the many quotations, I believe that several important ideas are exposed.

This business is filled with a combination of old, established ideas and proven techniques. It also holds new questions and opportunities. Hamming's advice [Ref. 11: p. 14] seems most fitting in this situation:

> *Now I see constantly attempts to force new ideas to old molds. That is frequently sensible: How can I make sense of what I'm seeing compared to what I did before? But also one must ask, "Am I seeing something fundamentally new?" That part many people will not try. You cannot afford to make everything brand new and not connect anything together with existing ideas, nor can you try to make everything fit into preconceived categories. Some combination of the two is necessary.*
>
> *We limped through the transistor revolution and the computer revolution, which are connected with the bandwidth revolution; they are all connected together... You have to abandon old ideas when you get an order of magnitude of change. . . .*
>
> *— RICHARD W. HAMMING*

Developments in scientific computing today make Dr. Hamming's thoughts especially timely. The field needs to establish a strategy; a direction that will lead from its present immaturity to a place of fulfilling its potential. Kenneth Wilson proposes *Grand Challenges* for computational science that may help to establish this strategy [Ref. 12].

2. Grand Challenges

Wilson identifies three modes of scientific activity: theoretical, experimental, and computational. He defines these areas, claiming that—with today's supercomputers—the most recent science (computational) is becoming more significant. So significant, in fact, that "long experience or professional training is required to be successful in computational science at the supercomputer level, making it appropriate to think of computational science as both a separate mode of scientific endeavor and new discipline." [Ref. 12: p. 172]

Wilson is careful to distinguish computational science from computer science. He defines *computer science* as the business of addressing "generic intellectual challenges of the computer itself" and characterizes *computational science* as being tailored to specific applications areas (with serious training in the application discipline) [Ref. 12: p. 172]. To advance computational science, Wilson recommends a quantitative approach with clear strategies [Ref. 12: p. 173]:

> *The major future opportunities for benefits of supercomputers to basic research should be identified without the existing compromises, but presented as challenges to be overcome with the many obstacles to success clearly explained. The compromises and inadequacies of current computations need to be described and the level of advances required to overcome these inadequacies discussed. Furthermore, a few key areas with both extreme difficulties and extraordinary rewards for success should be labelled as the "Grand Challenges of Computational Science". Two examples are electronic structure and turbulence. No easy promises of success in Grand Challenges should be offered. Instead, computational scientists should be building plans to assault the Grand Challenges, pushing for the major advances in algorithms, software, and technology that will be required for true progress to be achieved in these areas. The Grand Challenges should define opportunities to open up vast new domains of scientific research, domains that are inaccessible to traditional experimental or theoretical modes of investigation.*

Wilson describes a few examples that demonstrate the limitations of experimental instrumentation and the potential of supercomputers. Weather prediction, astronomy, materials science, molecular biology, aerodynamics, and quantum field

theory are the six areas that Wilson chooses to make his point. He describes these areas in reasonable detail and briefly mentions other topics. [Ref. 12: pp. 175–179]

a. Mathematical Background

Wilson stresses the need for sound design practices and good algorithms. (To see why, consider Table A.1). Additionally, he warns that we should spend less time in awe of today's supercomputing power and admit that it is terribly inadequate. Modeling methods and sound mathematical background also appear in the "needs improvement" category. Wilson [Ref. 12: p. 180] believes that

> *Mathematical developments that relate to numerical computation are highly important. Theorems about numerical errors or sources of error, exact solutions and expansions, existence and uniqueness proofs and the like, can make a major difference in establishing the credibility of a numerical computation. All too frequently there is too little mathematical understanding backing up numerical simulation.*

b. Issues of Quality

Wilson does not consider these to be the only problems facing computational scientists. He believes that *quality* is endangered, primarily from two directions [Ref. 12: pp. 180–181]:

- A tendency to stay on the safe, easy side; not wandering far from the position: "our calculation agrees with experiment."

- The quality of computational programs, measured against practical criteria, is lacking. The standards include rounding errors (e.g., catastrophic cancellation), overflows, and stability (with respect to input parameters).

c. *Languages*

Wilson cites a number of reasons for revolutions in computer languages. In particular, he believes that "Fortran is in the long–term the most fundamental barrier to progress" [Ref. 12: p. 182]. His approach is realistic enough to recognize the vast investments of scientific communities in Fortran. The language cannot and should not be eliminated in a day. Nevertheless, it has very serious shortcomings. Some problems could be overcome by a Fortran preprocessor (the same idea as the C preprocessor). Other problems, like lack of support for abstraction and the unnatural exclusion of basic mathematical symbols in the language, are not solved as easily. [Ref. 12: p. 182]

Wilson *does not* recommend a simple change of language as the solution, but searches for deeper problems. He believes that the entire way that computational scientists and programmers think about and plan programs must change as well. After reading Wilson's analysis of language problems, the basic impression that prevails is that we have an urgent need for general–purpose practices to replace patchwork, hit–or–miss, case–by–case solutions.

3. Generality

David Harel is also an advocate of the need for general purpose techniques. In the preface to his book [Ref. 13: p. viii] he warns:

> *Curiously, there appears to be very little written material devoted to the science of computing and aimed at the technically oriented general reader as well as the professional. This fact is doubly curious in view of the abundance of precisely this kind of literature in most other scientific areas, such as physics, biology, chemistry and mathematics, not to mention humanities and the arts. There appears to be an acute need for a technically detailed, expository account of the fundamentals of computer science; one that suffers as little as possible from the bit/byte or semicolon syndromes and their derivatives, one that transcends the technological and linguistic whirlpool of specifics, and one that is useful both to a sophisticated*

17

layperson and to a computer expert. It seems that we have all been too busy with the revolution to be bothered with satisfying such a need.

This idea is not unique. One of the other major proponents of general–purpose parallel computing is David May of INMOS. In an invited lecture at the the Transputing '91 conference [Ref. 14], he highlighted features that general–purpose parallel hardware should deliver. Among the important components of a general approach, May included the following:

- **Scaling.** Performance must scale with number of processors. Efficiency is partly dependent on problem size, but—with adequate problem size—systems of a thousand processors should be within technological reach. Each processor is expected to achieve 10^8–10^9 flops.

- **Portability.** This is almost synonymous with "general purpose." May emphasizes *algorithms* based upon features common to many machines, and which remain valid as technology evolves. He stresses that this general purpose parallel architecture will benefit both the computer designer and the programmer. The designer will gain since the market will be somewhat predictable. The programmer's code will work on several machines and hold a strong hope for working into future years.

To achieve these goals, May proposes several guidelines. First, for a message passing system using p processors, the nodes must be capable of concurrent computing and communication. The interconnection topology must provide scalable throughput (linear in p) and bounded delay, probably $\log(p)$. Programs, May believes, should be written at as high a level as possible and make use of many processes. *The algorithm should express the maximum possible parallelism.* Much of May's theory is based upon the structure of a hypercube interconnection topology (or virtual hypercube).

4. Projections

Kenneth Wilson makes a credible claim that says parallel computing is here to stay. His reasoning is based upon the fact that *mass production* and *heavy competition* are proven ingredients in keeping the cost of chips low. Rather than summarize, I will quote his conclusion [Ref. 12: p. 185]:

> *Today a single processing unit costing millions of dollars can still be cost-effective but I don't think this can last very long, over a period of time (I cannot estimate how many years) it seems likely that the maximum price of a cost–effective processor will plunge to one hundred thousand dollars, to ten thousand dollars, to ???. I cannot estimate the ultimate equilibrium price at which this plunge will stop.*
>
> *Meanwhile I can find no prospects that single supercomputer processors speeds will advance at anything like the pace at which processor costs are being reduced, even using Gallium Arsenide or superconducting Josephson junctions.*
>
> *The result of this is inevitable—overall advances at the supercomputer level have to come through parallelism, namely, big increases in speed have to come from the simultaneous use of many processors in parallel.*

David May agrees with Wilson, who states that increasingly complex components and faster clock speeds are not likely avenues of advancement. This makes parallel processing "technically attractive." He also agrees that mass production will make the most effective use of design and production facilities. His conclusion: "A general purpose parallel architecture would allow cheap, standard multiprocessors to become pervasive." [Ref. 14]

May's prediction for 1995 includes processors capable of 100 megaflops. INMOS believes strongly in the idea of *balancing computation and communication*, and May projects that node throughputs will have reached 500 megabytes per second. In 1995's multiprocessor systems, he envisions teraflop performance. By 2000, May projects "scalable general purpose parallel computers will cover the performance range up to 10^{11} flops. Specialised parallel computers will extend this to 10^{13} flops."

19

D. OVERVIEW

This chapter has surveyed the (relatively recent) history of computing, considered the state–of–the–art, and made a few guesses as to the future. Additionally, it has introduced numerical and parallel computing. This serves as a backdrop for the remainder of the thesis. Chapter II expands the background on parallel processing and numerical methods. The latter provides a lead–in to the specific algorithms and theory that appear in Chapter III. Chapter IV introduces the parallel design and methods used in the work. A description of the environment, tools, and equipment appears in Chapter V. Results and conclusions appear in Chapters VI and VII.

Appendices are provided to keep the chapters concise and focused. The appendix material operates on both sides of that focus. Some of the material is designed to give sufficient background and the rest—code mostly—is provided for more in–depth study. The background material may be obvious to some readers and new to others. I have assumed that the reader has some knowledge of the background material. I *do not* presume that the reader will be familiar with the code.

To simplify the discussion we must speak the same language. Appendix A gives the basic terms and notation used in the rest of the thesis. Next, we discuss the machines used to perform the work. While this is the subject of Chapter V, a more detailed account is reserved for Appendix B. Appendix C provides a general background on interconnection topologies. Emphasis is placed upon the hypercube connection scheme. Appendix D describes the process whereby a real–world problem is translated into matrix notation. Appendix E gives some information and results for communications performance in a hypercube. Finally, Appendix F provides listings for most of the code used in the research.

II. BACKGROUND

Mathematics is the door and key to the sciences.

— *ROGER BACON*

Chapter I provided a backdrop, showing the state of scientific computing, especially parallel and distributed forms, today. In the present chapter, the scope is limited to material and equipment pertaining to this research. The thesis work deals with methods of *conjugate directions* implemented upon two contemporary MIMD machines. The goal is to *introduce* the theory, machines, methods, and a few peripheral issues that will be helpful as background information.

A. COMPUTING WITH REAL NUMBERS

As illustrated in Figure 1.1, the speed of computing machinery has risen swiftly since the 1940s. This has often been encouraged by substantial advances in technology. Today's multiprocessor machines seem to be maintaining the fast–paced growth. Additionally—although precision is a less glamorous business than speed— the accuracy of machine solutions has become more standard. This section considers some of the principal issues of computing with finite approximations of real numbers.

We have observed that the history of computing shows close ties to science and mathematics. As the design and construction of computers becomes a more specialized business—mostly performed by electrical and computer engineers—we still find that many of the fundamental requirements are related to scientific problems. These problems typically involve mathematics and a significant amount of scientific computing applies numerical methods that involve real numbers. The trend in com-

puter (hardware and software) design is toward abstraction, but from time to time we absolutely must understand and work with the underlying, concrete principles.

1. Finite–Precision

New problems are generated as the *speed* of computing machinery improves with each generation of machines. One question to be considered is, how *reliable* are the machines and the software that runs on them? This is a constant concern in computing. Many scientific problems involve continuous phenomena in the real world. Accordingly, we like to be able to represent the real numbers, $\Re$, within the machine. But, lacking infinite storage, this is impossible. There have been several more–or–less reasonable ideas and implementations of approximations to the real numbers within the limits of computer storage. Of these, the *floating–point* concept of storage and arithmetic enjoys the most widespread use.

The Institute of Electrical and Electronics Engineers (IEEE) has established the principal standards for floating–point representations and arithmetic. These standards make machine arithmetic more predictable. Surprisingly, while they exist in much of today's computing hardware, the standards are not widely understood by practitioners. Then, software and applications are sometimes formed in ignorance. The title of David Goldberg's paper [Ref. 15] speaks volumes: "What Every Computer Scientist Should Know About Floating–Point Arithmetic." Goldberg is also responsible for several other contributions describing floating–point arithmetic and the IEEE standards. Appendix A of Hennessy and Patterson's book on architecture [Ref. 3] is such a contribution. He gives a very useful description of the IEEE standards and instruction on how to perform arithmetic operations on machines that adhere to the IEEE standards.

2. IEEE 754

Of the four precisions specified by the IEEE 754–1985 standard, this thesis uses the *double precision* format most often (to approximate real numbers) so it will receive the most attention. In the C programming language, these numbers correspond to the type **double**. They are floating–point values stored in eight bytes (64 bits). The storage representation is illustrated as three components: one sign bit, s; an 11–bit exponent, e; and a 52–bit *fraction*, f. Figure 2.1 shows an example. We say that e is a *biased* exponent. Both negative and positive exponents are stored using a range of positive binary numbers biased about (nearly) the middle. *Significand* or *mantissa* is the name given to the number $(1.f)$. The fraction is a *packed* form of the significand. This means that the leading one of the significand is implicit. This is called a *normalized* number. [Ref. 16]

All IEEE floating–point numbers are normalized except for the special representations when $e = 00000000000 = 0$ or $e = 11111111111 = 2047$. These are called *denormalized* (or *subnormalized*) numbers. Only the fraction, f, of a normalized number is stored [Ref. 3: p. A–14]. Figure 2.1 shows a representation of the floating–point number, $x = 7.0$. First, x is shown as it would be defined in a C program. The C *address of* operator, &, is used to indicate the address of x in memory. That is, somewhere (namely &x) in memory, there are eight contiguous bytes that hold a floating–point representation of x and (for illustration purposes) we can imagine the IEEE 754 double-precision representation of x as Figure 2.1 indicates.

A standard, such as IEEE 754 (and the lesser–known IEEE 854), is not a panacea for the finite–precision problem but it lends tremendous support to those who would scientifically deal with the problems of finite–precision arithmetic. Programs given in the files **num_sys.h** and **num_sys.c** (in Appendix F) are of interest to those who would explore further. The programs can demonstrate that the actual

```
double x = 7.0;
```

&x

| 0 | 10000000001 | 1100000000000000000000000000000000000000000000000000 |

s $e = 1025$ $f = .11_2$

$$
\begin{aligned}
\text{Interpretation:} \qquad x &= (-1^s) \times 1.f_2 \times 2^{e-1023} \\
&= (-1^0) \times 1.11_2 \times 2^{1025-1023} \\
&= 1.11_2 \times 4 \\
&= 111_2 \\
&= 7
\end{aligned}
$$

Figure 2.1: IEEE 754 Representation: Double Precision

order and location of bits in memory may not match the representation of Figure 2.1. This reflects practicalities concerning storage and transmission of bytes at a very low level in the machine. It is perfectly reasonable (and easier) to use the common abstraction of Figure 2.1 regardless of machine implementation.

B. NUMERICAL ISSUES

1. The Need

Consider the problem of determining the area under a bounded function $f(x)$ over a closed interval $[a, b]$. Numerical *quadrature* (integration) rules such as the Trapezoidal Rule or Simpson's Rule are used to arrive at an approximating (or *Riemann*) sum of many smaller areas within the region. Numerical methods are

often used to approximate the solution to a problem. This is no trivial problem. To solve it (numerically) by anything other than accident, one must first understand the theory and analytical approach. Next, the problem can be translated into an *algorithm* (a plan—usually mathematical in nature—for solving the problem step-by-step) which can, in turn, be translated into the sort of language that a machine understands.

This is a relatively simple approximation problem compared to the problem of finding the solution to a system of 500 equations in 500 unknowns. Consider the (perhaps more realistic) problem of using numerical linear algebra to solve an elliptic partial differential equation like the one presented in Appendix D. Numerical concerns abound in problems such as these. Additionally, many problems in numerical linear algebra have time complexities of $\Theta(n^2)$ or $\Theta(n^3)$ and storage requirements of $\Theta(n^2)$ so speed is essential. (Appendix A reviews the complexity notation such as big–Oh and big–Theta).

2. Errors and Blunders

A clear understanding of the differences between *errors* and *blunders* is important since recognition of the source of error is prerequisite to eliminating or reducing them. The terms are introduced in [Ref. 17: p. 1]:

> *Blunders result from fallibility, errors from finitude. Blunders will not be considered here to any extent. There are fairly obvious ways to guard against them, and their effect, when they occur, can be gross, insignificant, or anywhere in between. Generally the sources of error other than blunders will leave a limited range of uncertainty, and generally this can be reduced, if necessary, by additional labor. It is important to be able to estimate the extent of the range of uncertainty.*
>
> — *ALSTON S. HOUSEHOLDER*

3. The Issues

To anticipate—or even troubleshoot—error we must know from whence it comes. In [Ref. 17: p. 2], Alston Householder lists the *four sources of error* that were set forth by John von Neumann and Herman Goldstine:

- Mathematical formulations are seldom exactly descriptive of any real situation, but only of more or less idealized models. Perfect gases and material points do not exist.

- Most mathematical formulations contain parameters, such as lengths, times, masses, temperatures, etc., whose values can be had only from measurement. Such measurements may be accurate to within 1, 0.1, or 0.01 percent, or better, but however small the limit of error, it is not zero.

- Many mathematical equations have solutions that can be constructed only in the sense that an infinite process can be described whose limit is the solution in question. By definition the infinite process cannot be completed. So one must stop with some term in the sequence, accepting this as the adequate approximation to the required solution. This results in a type of error called the truncation error.

- The decimal representation of a number is made by writing a sequence of digits to the left, and one to the right, of an origin which is marked by a decimal point. The digits to the left of the decimal point are finite in number and are understood to represent coefficients of decreasing powers of 10. In digital computation only a finite number of these digits can be taken account of. The error due to dropping the others is called the round–off error. . . .

26

C. MACHINE METHODS

We would like to somehow characterize the techniques that make a problem-solving method "good". The abilities of machines and people are distinct enough that we should not always expect an algorithm for machine solution to mirror the pencil-and-paper method of an individual. Hestenes and Stiefel make this distinction, defining a *hand method* as "one in which a desk calculator may be used" and a *machine method* as "one in which sequence-controlled machines are used." [Ref. 18: p. 409] Further, in the same reference, they list the following characteristics that a good machine method exhibits:

(1) The method should be simple, composed of a repetition of elementary routines requiring a minimum of storage space.

(2) The method should insure rapid convergence if the number of steps required for the solution is infinite. A method which—if no rounding-off errors occur—will yield the solution in a finite number of steps is to be preferred.

(3) The procedure should be stable with respect to rounding-off errors. If needed, a subroutine should be available to insure this stability. It should be possible to diminish rounding-off errors by a repetition of the same routine, starting with the previous result as the new estimate of the solution.

(4) Each step should give information about the solution and should yield a new and better estimate than the previous one.

(5) As many of the original data as possible should be used during each step of the routine. Special properties of the given linear system—such as having many vanishing coefficients—should be preserved. (For example, in the Gauss elimination special properties of this type may be destroyed.)

D. CONJUGATE DIRECTIONS

Hestenes and Stiefel describe the *method of conjugate directions* (CD). This is a general approach to solving systems of linear equations that uses *direction vectors*, $p_0, p_1, \ldots$, to determine how the search for a solution should proceed from step-to-step. When the method for determining these vectors is defined, CD becomes a

specific method. There are at least two of these specific methods within CD that are especially suited to computer implementation: *Gauss factorization* (GF) and the method of *conjugate gradients* (CG). [Ref. 18: p. 412]

The term *conjugate* is clearly an important one for these methods. Given a matrix $A \in \Re^{n \times n}$ that is *symmetric*, we say that two vectors x and y are *conjugate* if

$$x^T A y = (Ax)^T y = 0 . \tag{2.1}$$

There is an alternative term that emphasizes the role of A in this definition. We also say that x and y are *A-orthogonal*. [Ref. 18: p. 410]

The method of conjugate gradients chooses its direction vectors, p_i, to be mutually conjugate ($p_i^T A p_j = 0$ whenever $i \neq j$) *and* in such a manner that p_{i+1} depends upon p_i. (A specific formula is given near the end of Chapter III). The Gauss factorization chooses $p_i = e_i$, the ith axis vector. [Ref. 18: pp. 412,425–427]

In this research, the Gauss method gets almost all of the attention, but the method of conjugate gradients receives a short overview near the end of Chapter III. The theory of conjugate directions is not at all trivial, and the ties of Gauss and conjugate gradients to conjugate directions are fairly deep. These issues are covered in the work of Hestenes and Stiefel [Ref. 18]. This thesis develops the Gauss method from an implementation standpoint.

E. PARALLEL PROCESSING

The field of parallel and distributed computing is a relatively new one. In one sense, it is quite natural. We perform work in parallel every day. In fact, a manager–worker notion is a very useful means to understand the issues of this field. The programs developed in this research involve a host or manager and nodes or workers. This is often called the *workfarm* approach.

The principal "problem" in parallel computing is *communication*. Appendix C relates some of the considerations. Of course, there are other concerns as well: load balancing, problem size (granularity), and so on. These issues, as they apply to the this research, are discussed in Chapter IV.

The bottom line—after all of the design and implementation work—is performance. With *multicomputers*, as in a workfarm, we are after efficiency so that more computing can be done in a shorter time and for less money. Bell is even more specific. He believes the multicomputer must offer two key facilities to become established [Ref. 6: p. 1097]:

- Power that is not otherwise available.

- Performance for a price that is "at least an order of magnitude cheaper than traditional supercomputers."

In Chapter VI, we consider results obtained upon two contemporary parallel machines. This information helps us to evaluate the potential of MIMD architectures in terms of Bell's criteria.

F. SPEEDUP

The terms *speedup* and *efficiency*, defined in Appendix A, capture most of the interest when we talk about the potential of parallel computing. The principal reason for choosing a multicomputer over a single computer is speed. Therefore, we are most interested in knowing what kind of *speed* we can obtain from a multiprocessor system. Bell's comments on price are germane as well.

Speedup and efficiency are both machine dependent and problem dependent. Some problems should not be executed on a parallel machine! Suppose, for instance, that part of a problem must be performed sequentially. *Amdahl's law* is a well–known attempt to characterize this problem. Amdahl stated that speedup on P processors, S, is limited in the following manner:

$$S \leq \frac{1}{f + (1 - f)/P} \tag{2.2}$$

where f is "the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$" [Ref. 19 : p. 19]. With speedup, S, defined as in (2.2) we see that

$$\lim_{P \to \infty} S = \frac{1}{f}. \tag{2.3}$$

Figure 2.2 shows how this limit begins to take effect as the number of processors, P, is increased from zero to 500. The figure is based on Amdahl's law (2.2) with sequential percentages, f, of 5%, 10%, and 25%.

We can see that Amdahl's law has some very discouraging news for so–called *massively* parallel computing. The *massive* part of the term is loosely defined, apparently meaning "many" processors. But Amdahl's law may be based upon a faulty assumption [Ref. 20]. Consider the following reasoning. Let P be the number of processors and consider the following *arguments concerning time.* Let s be the time required to execute the serial portions of a program on a *serial processor* and let p be the amount of time required to complete the parallel work on the same *serial processor.* Using this notation, and normalizing $(s + p = 1)$, Amdahl's law can be restated

$$S = \frac{s + p}{s + (p/P)} = \frac{1}{s + (p/P)}. \tag{2.4}$$

Then, if we consider the case $P = 1,024$ with $s \leq 10\%$, we see in Figure 2.3, that speedup is severely restricted.
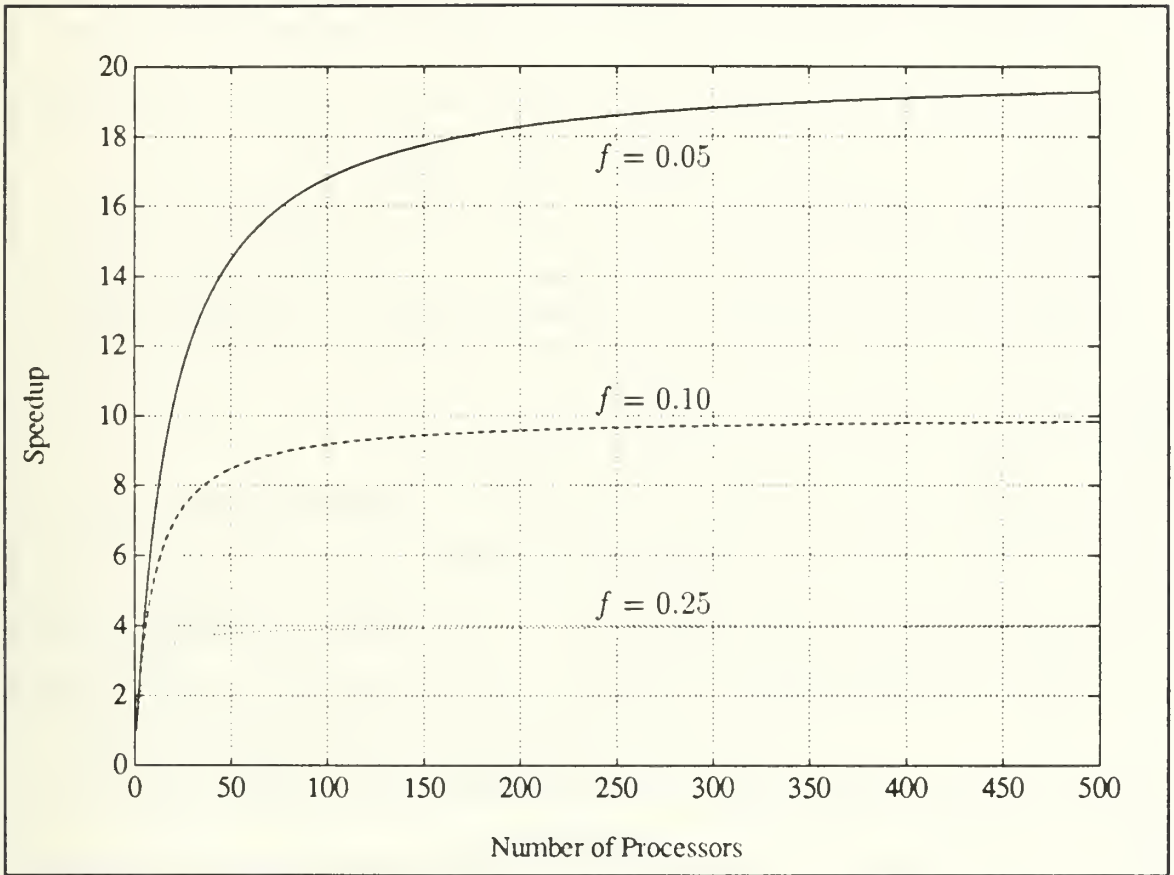
Figure 2.2: Amdahl's Law ($1 \leq P \leq 500$)
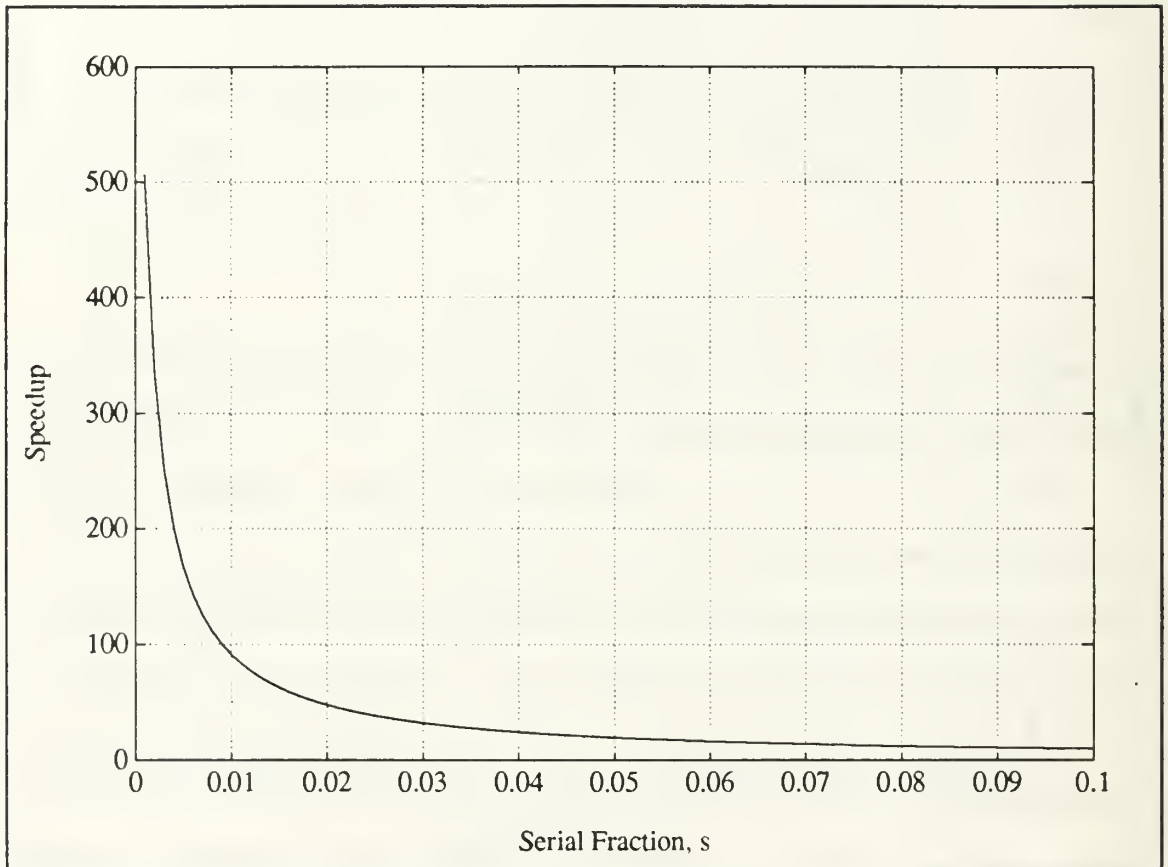
Figure 2.3: Amdahl's Law ($P = 1024$)

G. SCALED SPEEDUP

These problems with the usual notion of speedup led Gustafson, Montry, and Benner to question the validity of Amdahl's assumptions [Ref. 20: p. 3]:

> *The expression and graph are based on the implicit assumption that p is independent of P. However, one does not generally take a fixed size problem and run it on various numbers of processors; in practice, a scientific computing problem scales with the available processing power. The fixed quantity is not the problem size but rather the amount of time a user is willing to wait for an answer; when given more computing power, the user expands the problem (more spatial variables, for example) to use the available hardware resources.*
>
> *As a first approximation, we have found that it is the parallel part of a program that scales with the problem size. Times for program loading, serial bottlenecks, and I/O that make up the s component of the application do not scale with the problem size. When we double the number of processors, we double the number of spatial variables in a physical simulation. As a first approximation, the amount of work that can be done in parallel varies linearly with the number of processors . . . .*

Based upon this analysis, they present the notion of *scaled speedup*. They let s' and p' represent the serial and parallel time spent on a *parallel* system (inverse of Amdahl's method). So that $s' + p' = 1$ and a uniprocessor requires time $s' + p'P$ to perform the task. With these definitions, they define scaled speedup, S', to be

$$S' = \frac{s' + p'P}{s' + p'} = P + (1 - P)s'. \tag{2.5}$$

If we consider the same range of serial fractions as we did in Figure 2.3, we see that scaled speedup is much better than the usual speedup. Figure 2.4 shows the plot of scaled speedup.

H. SUMMARY

This chapter considers the background necessary to develop the algorithms (Chapters III and IV) and implement them (Chapter V). Algorithms are described as sequential plans first (Chapter III). The Gauss factorization algorithm is given
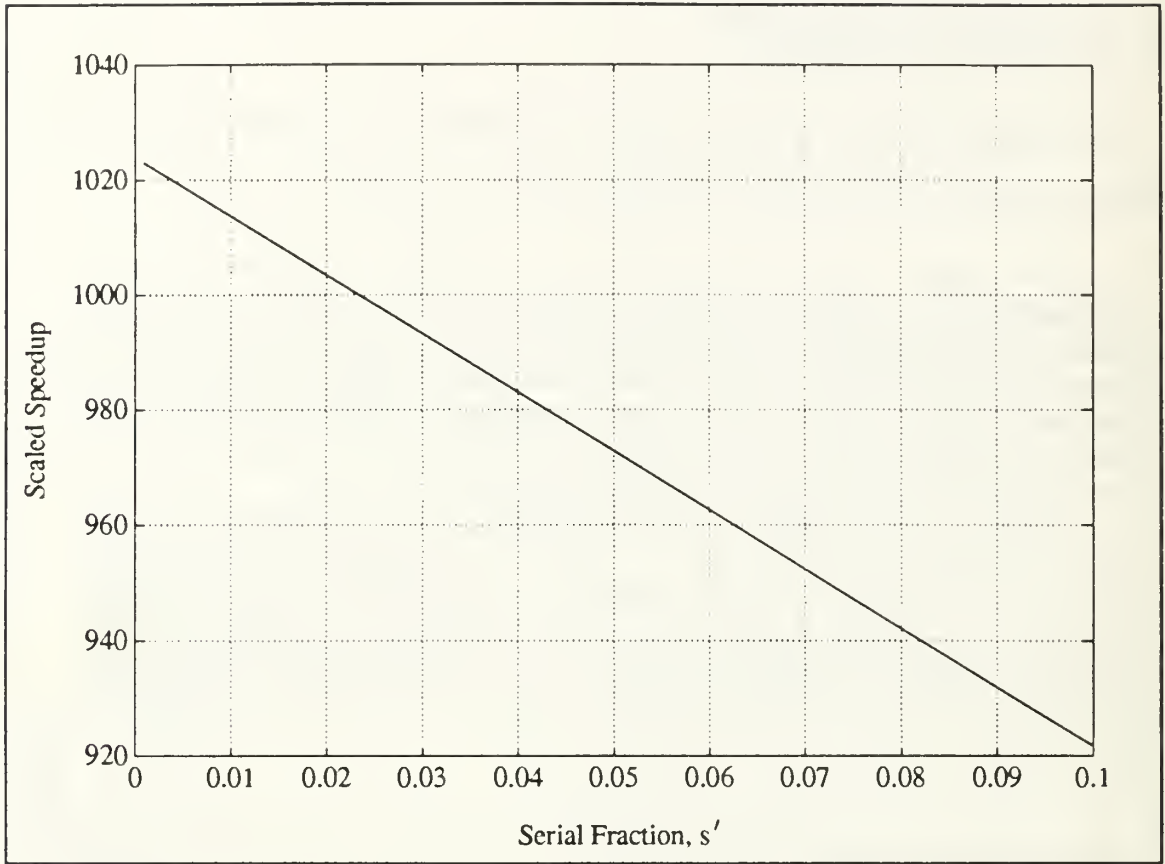
Figure 2.4: Scaled Speedup

in detail (Chapter III), including a discussion on the significance of pivoting. The method of conjugate gradients receives less attention, but a brief introduction is given near the end of Chapter III. The parallel considerations surveyed quickly in this chapter receive more attention in Chapter IV.

III. THEORY

No human investigation can be called real science if it cannot be demonstrated mathematically.

— *LEONARDO DA VINCI (1452–1519)*

A. SCOPE

The goal of this research is to demonstrate a parallel method for solving a system of linear equations. The implementation targets two contemporary MIMD architectures: the Intel iPSC/2 and networks of INMOS transputers. There are many methods for solving linear systems. This work concentrates primarily upon Gauss factorization (GF), but the method of conjugate gradients (CG) is also introduced. Regrettably, CG is not developed due to time constraints (the derivation is not trivial). This does not imply that Gauss factorization is superior, nor that it possesses greater potential for parallel solution. Indeed, Hestenes and Stiefel preferred CG to GF for a number of very good reasons [Ref. 18: p. 409].

As we shall see, the utility of either method is quite dependent upon the nature of the particular problem. Consider the system of linear equations represented by

$$Au = b. \tag{3.1}$$

Much of the subsequent discussion applies to general, rectangular systems where $A \in \Re^{m \times n}$. For the examples, however, *square* systems ($A \in \Re^{n \times n}$) are used. This restriction greatly simplifies the discussion without losing much of the concept as it applies to general systems. The Gauss *process*, i.e., the main part of the work, excluding the stopping criteria and interpretation of the result, is the same in all three cases ($m < n$, $m = n$, and $m > n$).

To be sure, the three cases ($m < n$, $m = n$, and $m > n$) correspond to funda-mentally different real–world systems, but the algorithms for each case are almost identical. The restriction to a square system will greatly simplify the discussion without blinding us to the general, rectangular case. The extensions to the general case are well known. Golub and Van Loan [Ref. 21: p. 102] give more detail, but the square case is most expedient for now. Square systems also simplify the experimental procedure, data collection and analysis.

The Gauss method follows naturally from a hand method and it holds strong appeal to intuition. Without a *pivoting strategy*, however, Gauss can attempt division by zero. There is also a more subtle issue of rounding errors within the limits of finite–precision arithmetic. To forestall errors of both kinds, partial and complete pivoting strategies are used. This chapter develops the (sequential) algorithms and explains the concept of pivoting. This is a sensible starting point for Chapter IV, where *parallel* versions of the algorithms are given.

B. APPROACH

There are many methods that may be applied to determine the solution of a system of linear equations. The methods were designed for different reasons and with different problems in mind, so each exhibits a unique behavior. One method is often preferred over another for a given problem. Ultimately, the criterion is performance, both in reliability and speed. The approach described here and in the remaining chapters seeks to "maximize performance" while retaining a reasonable balance of both efficiency and quality. Speed and numerical accuracy tend to oppose one another so we are left to choose from several options.

A hand method introduces each algorithm. The example is small and concrete. Solving a small problem gives useful insights into the algorithms. Once the hand method is established, it is expressed in an equivalent matrix notation. A high–level

36

sequential algorithm is built upon this foundation. This algorithm shows how a machine, using a sequence of instructions, solves the problem. It also gives good estimates for the problem's time and storage complexities. The sequential–to–parallel transition involves enough issues to warrant separate coverage. These considerations appear in Chapter IV.

In the sections that follow, Gaussian elimination is presented first. It reveals the background (sort of a first pass) for Gauss factorization. Once the reduction process is understood, we proceed to factorization. A description of the method of conjugate gradients is given at the end of the chapter. This method, due to Hestenes and Stiefel, is based upon relatively deep theory. Thus the derivations and background are not included. Nevertheless, a synopsis of the method is given.

C. APPLYING THE METHODS

A particular method is often tailored to a specific type of system. The method of conjugate gradients, for instance, is usually used when the matrix of coefficients, A, is symmetric and positive definite [Ref. 18: p. 411]. The Gauss factorization algorithm is equally important, but it takes quite another approach to solving this system. Both CG and GF lie within the broad category of methods of conjugate directions (Chapter II). Indeed both work in just about any case. But, the better results are obtained by using the tool that fits the task at hand.

A very rough characterization of the problem can simplify algorithm selection. We will look for two qualities: **structure** and **density**. CG, for instance, performs best when applied to highly structured, sparse matrices (i.e., matrices with many zero entries). Systems like the sparse, symmetric, highly–structured result of Appendix D deserve careful solutions that do not destroy the existing zeros. Zeros are not always easy to come by. Gaussian elimination must expend $2n^3/3$ flops to create them.

Selecting the wrong algorithm can lead to slower execution. More importantly, poor algorithm choice is a *blunder* (Chaper II). It can produce results that are accidentally perfect, grossly incorrect, or anywhere between. Therefore, no less than three tasks confront us:

- Characterize the problem. In systems like (3.1), attributes of the matrix of coefficients, A, may provide a wealth of information.

- Understand the algorithm(s). Know the types of problem(s) it is designed for (and, more importantly, know why).

- Create or select an algorithm that suits the problem.

The sparse, highly–structured problems are not rare! Anyone who has observed nature knows that many natural phenomena exhibit incredible structure and simplicity. Strategies for solving the corresponding system should always seek to exploit these characteristics. Both sparseness and structure can reduce storage requirements and the number of flops required. If we know the structure in advance, there may be a smart way to avoid some calculations entirely or minimize the work involved. (Recall Hestenes and Stiefel's characterization of a "good" machine method from Chapter II). Other problems, when translated into the form (3.1), exhibit a dense matrix, A, with little or no apparent structure.

These two types of problems should not be handled with the same tools. As with many computational problems, the reasons involve the use of time and space. We shall see that the Gauss algorithm has time complexity $\Theta(n^3)$ and storage requirements $\Theta(n^2)$. (Complexity notation appears in Appendix A). Numbers like these grow rapidly with n and, regardless of how much memory is available, the problem can quickly overpower the computer. A naïve approach to problems of these kinds can be expensive in terms of both storage and time. This is usually

adequate incentive to take advantage of sparseness and structure whenever possible. When it is not possible, Gauss is a good choice.

D. GAUSSIAN ELIMINATION

Suppose that we want to solve a system of linear equations using a systematic, step–by–step method. We assume that the system of linear equations is given, and that the method must preserve the original properties of the system. That is, the method must be restricted to certain operations; namely:

- Multiply an equation by a nonzero constant.

- Interchange equations.

- Add a multiple of one equation to another.

The fact that the first two operations do not change the system's properties is evident. The third operation is legitimate also—maybe not quite so obviously—and computationally, the most significant. Now let us apply some of these operations to a system of four equations in the four unknowns, v_1, v_2, v_3, and v_4.

$$\begin{array}{rcr} 2v_1 + 3v_2 + 4v_3 + 5v_4 &=& 0 \\ 4v_1 + 6v_2 + 8v_3 + 5v_4 &=& -5 \\ 2v_1 + 4v_2 + 7v_3 + 9v_4 &=& 13 \\ 6v_1 + 8v_2 + 8v_3 + 9v_4 &=& -17 \end{array} \qquad (3.2)$$

Let $m \ (= 4)$ be the number of equations, and let $n \ (= 4)$ be the number of unknowns in each equation. Additionally, let i be an equation (or row) index $(1 \le i \le m)$ and let j indicate a subscript of v (column index) so that $1 \le j \le n$. Finally, let α_{ij} be the coefficient of v_j in equation i (e.g., $\alpha_{12} = 3$). Suppose that the last equation contains only one nonzero coefficient (say α_{44}) and the third equation has only two nonzero coefficients (α_{33} and α_{34}) and so on. This defines a *triangular* system (Appendix A). The triangular system is our goal because it is easier to solve (by *back substitution*) than the current (square, dense) system.

Next, observe that a triangular system would result if we could *eliminate* every coefficient, α_{i1}, of v_1 in all equations but the first ($i > 1$), coefficients, α_{i2}, of v_2 in the last two equations ($i > 2$), and the coefficient, α_{43}, of v_3 in the final equation. To do this, we work by stages. At stage k, the coefficient, α_{kk}, of v_k in the kth equation is called the *pivot*. This term has little significance now but is clarified later (and it plays a very important role in the examples presented. In a particular stage, k, the goal is to operate upon all equations i where $i \in \{(k+1), (k+2), \ldots, m\}$ and *eliminate* all coefficients, α_{ik}, of v_k.

1. A Hand Method

Before attempting to describe an algorithm for a machine solution, we consider an application of Gaussian elimination (GE) by hand. Initially, let $k = 1$. In the example system (3.2), the first ($k = 1$) pivot is the coefficient, $\alpha_{11} = 2$, of v_1 in the first equation. Notice that by subtracting twice the first equation from the second, a zero is produced under the pivot (eliminating α_{21}). Similarly, by subtracting the first equation from the third, a zero appears as the leading coefficient in the third equation (eliminating α_{31}). Finally, three times the first equation subtracted from the fourth equation eliminates the coefficient α_{41}. Following these steps the altered system is:

$$
\begin{array}{rcr}
2v_1 + 3v_2 + 4v_3 + 5v_4 & = & 0 \\
-5v_4 & = & -5 \\
v_2 + 3v_3 + 4v_4 & = & 13 \\
-v_2 - 4v_3 - 6v_4 & = & -17
\end{array}
\tag{3.3}
$$

This is called the *natural reduction process* [Ref. 22: p. 72]. In the particular case, there are no changes on the right–hand side because the first equation's right–hand side is zero. This makes for trivial arithmetic on the right–hand side, but we should remember to perform the arithmetic upon *whole equations* (including the right–hand side) in general. The elimination is even more successful than planned.

The second equation already has zeros where we ultimately wanted them in the *fourth* equation. That is, the system (3.3) would be closer to upper triangular if we were to alter it by interchanging equations 2 and 4.

$$
\begin{array}{rcr}
2v_1 + 3v_2 + 4v_3 + 5v_4 &=& 0 \\
-v_2 - 4v_3 - 6v_4 &=& -17 \\
v_2 + 3v_3 + 4v_4 &=& 13 \\
-5v_4 &=& -5
\end{array}
\tag{3.4}
$$

The system (3.4) is called a *row permutation* of (3.3). The ability to recognize patterns is a great advantage that human problem solvers enjoy. Therefore, taking advantage of our capabilities we use a rather subjective "human" pivoting strategy. But it is not fitting to assume that an efficient algorithm for a machine would involve the same sort of pattern recognition.

The system (3.4) is nearly triangular. The pivot moves to the second equation ($k = 2$), and we focus on the coefficient, $a_{22} = -1$, of $v_k = v_2$. By adding the second equation to the third, the only nonzero coefficient remaining in the lower triangle (a_{32}) is eliminated. The resulting system becomes

$$
\begin{array}{rcr}
2v_1 + 3v_2 + 4v_3 + 5v_4 &=& 0 \\
-v_2 - 4v_3 - 6v_4 &=& -17 \\
-v_3 - 2v_4 &=& -4 \\
-5v_4 &=& -5
\end{array}
\tag{3.5}
$$

The system is triangular, and it is easy to solve for the unknown values, v_i, by back substitution. By inspection, $v_4 = 1$. Substituting this value into the third equation, we find that $v_3 = 2$. Substituting both values (v_4 and v_3) into the second equation yields $v_2 = 3$. Finally, by substituting the values v_4, v_3, and v_2 into the first equation gives $v_1 = -11$. The solution to the system is then

$$
u = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} -11 \\ 3 \\ 2 \\ 1 \end{bmatrix}.
\tag{3.6}
$$

2. A Machine Method

The foregoing example illustrated the GE process as done on paper. The system was intentionally created for easy solution by hand calculation. I.e., it uses integers and elimination occurs faster than the usual case. Even this simple example requires a few minutes to determine u from the system (3.2) by hand. In Chapter VI, we see that a machine can perform this task in (much) less than a second. For this reason, it is worth examining an equivalent process to solve for such a system by machine.

We reenact the solution from the beginning, this time in a fashion that a sequence–controlled machine could perform. Until now, we have used the term "pivot" but have found no practical use for pivots. In this example, we begin to realize the utility of a pivoting strategy. We start with "no pivoting" and shift to the "partial pivoting" strategy. Additionally, we begin to use a more compact matrix notation. Appendix A describes the notation followed.

By the method described in Appendix A, we give the linear system (3.2) matrix representation that corresponds to (3.1):

$$Au = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 4 & 6 & 8 & 5 \\ 2 & 4 & 7 & 9 \\ 6 & 8 & 8 & 9 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 0 \\ -5 \\ 13 \\ -17 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{bmatrix} = b. \tag{3.7}$$

First, we initialize a stage counter, k, so that $k = 1$. The *pivot* in stage k is α_{kk}, on the diagonal of A ($\alpha_{11} = 2$). The immediate goal is to produce zeros beneath the pivot, in $A(2\!:\!4, 1)$. A three–step process eliminates these coefficients in row order:

- **Divide.** Divide every element beneath the pivot by the pivot value.

- **Update.** Perform arithmetic in the Gauss transform area.

- **Eliminate.** Set the elements beneath the pivot equal to zero.

The first step is a division. The denominator (pivot) is $\alpha_{kk} = \alpha_{11} = 2$ so α_{21} becomes the *multiplier* $(\alpha_{21}/2) = 2$. Similarly, let $\alpha_{31} = 1$ and let $\alpha_{41} = 3$. Now

$$A = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 2 & 6 & 8 & 5 \\ 1 & 4 & 7 & 9 \\ 3 & 8 & 8 & 9 \end{bmatrix}. \tag{3.8}$$

Next, consider everything below and to the right of the pivot. This is the *Gauss transform area*, $G = A((k+1):m, (k+1):n) = A(2:4, 2:4)$. For each element in G, replace the current value, α_{ij}, with $\alpha_{ij} - (\alpha_{ik})(\alpha_{kj})$. Do the same thing in the corresponding rows $(i > k)$ of b, replacing β_i with $\beta_i - (\alpha_{ik})(\beta_k)$. We will call this the process of performing arithmetic in (or updating) the Gauss transform area, G.

Finally, when the values beneath the pivot are no longer needed, eliminate them (set them equal to zero). The result is equivalent to the system (3.3):

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & -5 \\ 0 & 1 & 3 & 4 \\ 0 & -1 & -4 & -6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 0 \\ -5 \\ 13 \\ -17 \end{bmatrix}. \tag{3.9}$$

We have finished one stage of GE. We move into the next stage, $k = 2$. This time, when we try to update G we run into a very serious problem. The first step is to divide everything underneath the pivot by the pivot value $\alpha_{kk} = \alpha_{22} = 0$. This is the divide–by–zero problem of a "no pivoting" strategy.

During the execution of the hand example we simply moved the row to the bottom of the system to avoid this problem. Now, we could instruct the machine to test every element in $A(k:m, k:n)$ and interchange rows so that those with the most leading zeros were placed at the bottom. This is problematic for several reasons. First, it is not dependable (testing for equality of floating–point numbers begs disaster). Secondly—even if we could identify zeros with confidence—it would add a sorting problem to GE! We are not looking for extra work. The solution is *partial pivoting*.

3. Partial Pivoting

Partial pivoting is an application of row interchanges to eliminate (primarily) the divide–by–zero problem. Consider the system of equations (3.1) with the nonsingular matrix of coefficients, $A \in \Re^{m \times n}$ (i.e., $m = n$ and the system has exactly one solution). Suppose further that storage and arithmetic is performed in *infinite precision*. (These assumptions—infinite precision and A nonsingular—are essential).

Even in this ideal situation Gauss without pivoting is dangerous because, as we have just seen, it may attempt to divide by zero. Proper *row permutations completely eliminate* this problem. *Partial pivoting* will **guarantee** the existence of n nonzero pivots for A nonsingular. In fact, if we encounter a zero pivot with partial pivoting, it means that A is singular [Ref. 23]. The remainder of this section describes the partial pivoting strategy.

Consider stage k of the GE process with $A \in \Re^{m \times n}$. The goal is to pick the "best" row remaining (i.e., at or below the current pivot) and install it as row k, the pivot row. For reasons that are explained later, "best" shall mean the row whose k^{th} (pivot column) element is largest. Let s be the row index for the best pivot candidate. Initially, let $s = k$ (i.e., α_{kk} is the first candidate). Next, we move down the pivot column, considering all α_{ik} where $i \geq k$.

To eliminate unnecessary assignments, we replace the current candidate with another only if $|\alpha_{ik}| > |\alpha_{sk}|$. When this occurs, we make sure that s is updated by setting it equal to i. After considering all elements, α_{ik}, for $k \leq i \leq m$, s is the index of "best possible" pivot row. To accomplish our goal, we must perform a row interchange. This is easy after the new pivot row has been determined. We simply swap rows k and s (if $k \neq s$). Within the assumptions above, we have completely eliminated the potential for division by zero. Now let us return to the problem at hand.

4. A Machine Method (Resumed)

Applying partial pivoting to the system (3.9), we find that the next pivot is located at $A(3, 2)$ so we must interchange rows (equations) two and three. Before performing this step, however, let us create a vector to keep track of the row permutations. Let $q \in \Re^m$ be the row permutation vector. We initialize q so that $\psi_i = i$:

$$
q = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}
\tag{3.10}
$$

and perform row interchanges in q corresponding to those in A so that ψ_i is always the original equation number for current equation number i. Thus, after performing the row interchange, we have

$$
\begin{bmatrix} 2 & 3 & 4 & 5 \\ 0 & 1 & 3 & 4 \\ 0 & 0 & 0 & -5 \\ 0 & -1 & -4 & -6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 13 \\ -5 \\ -17 \end{bmatrix} \qquad q = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix}
\tag{3.11}
$$

Notice that $\psi_3 = 2$ indicates that the third equation in (3.11) was the second equation in the original system (3.7). Now, since $\alpha_{32} = 0$, no arithmetic is required in the third row. In row four, the arithmetic will be equivalent to the notion of adding (the current) equation two to equation four. The result is

$$
\begin{bmatrix} 2 & 3 & 4 & 5 \\ 0 & 1 & 3 & 4 \\ 0 & 0 & 0 & -5 \\ 0 & 0 & -1 & -2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 13 \\ -5 \\ -4 \end{bmatrix}.
\tag{3.12}
$$

When we move the pivot index to the third equation ($k = 3$), we notice that $\alpha_{33} = 0$. The divide–by–zero problem has resurfaced. Once again, we pivot, swapping rows three and four. After this, we have

$$
\begin{bmatrix} 2 & 3 & 4 & 5 \\ 0 & 1 & 3 & 4 \\ 0 & 0 & -1 & -2 \\ 0 & 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 13 \\ -4 \\ -5 \end{bmatrix} \qquad q = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 2 \end{bmatrix}
\tag{3.13}
$$

The zero beneath the final pivot obviates the need for further arithmetic. The triangular system (3.13), found by our machine method, does not look like the system (3.5) from the hand method because we did not perform the same row interchanges. If we had maintained a row permutation vector, $\tilde{q}$, for the hand method we would have noticed that

$$
q = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 2 \end{bmatrix} \neq \begin{bmatrix} 1 \\ 4 \\ 3 \\ 2 \end{bmatrix} = \tilde{q}. \tag{3.14}
$$

Of course, back substitution for the final (triangular) machine system (3.13) yields the same solution

$$
u = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} -11 \\ 3 \\ 2 \\ 1 \end{bmatrix} \tag{3.15}
$$

as that of the hand method. Thus, even though we used different permutation schemes, the "pivots" in both cases were always nonzero and the solutions were the same. This is not surprising, since A is nonsingular and row permutation is merely the practice of interchanging equations.

Let us review first the *process* and then the *theory* of Gaussian elimination. The GE process performs a systematic elimination of the lower (in our example) triangle of a matrix of coefficients, A. Arithmetic operations are performed upon entire equations at the same time (including the right–hand side, b). In other words, during stage k of the process, arithmetic operations are performed upon (portions of) all rows i ($i > k$) of A and upon all elements (rows) β_i (for $i > k$) of the right–hand sides, b. The process *depends upon both A and b* and both of them can be changed substantially.

The idea behind Gaussian elimination is that general square systems are difficult to solve, but triangular systems are easy. The goal is to transform a general matrix A into triangular form, performing legitimate arithmetic upon entire equa-

tions (including the right–hand sides). Reduction to triangular form costs $2n^3/3$ flops. Once A is reduced to triangular form, back substitution yields a solution for the unknown, u, in n^2 flops. Thus GE solves a general, dense, square system of n equations in n unknowns by the application of $2n^3/3 + n^2$ flops. [Ref. 21: pp. 88, 97]

E. GAUSS FACTORIZATION

Gauss *factorization* (GF) is a well–known method for solving linear systems like (3.1) that (simultaneously) *factors A*. GF has strong ties to the GE process. Those ties will become evident as we develop the same example over again, this time using the GF bookkeeping and method. GF holds several major advantages over GE. Among these: A is recoverable (the process does not destroy it) and the process is *independent of the right–hand side, b.* In fact, b is not used in the factoring process.

1. Complete Pivoting

The *complete pivoting* strategy will be applied in this example. There is no special significance behind the introduction of complete pivoting with the GF process. Either strategy—the choice of a "no pivoting" strategy is also available, but not generally acceptable for serious problems—can be used with GE or GF. The complete strategy is a straightforward extension of the partial strategy, so introducing partial pivoting first was practical.

With complete pivoting, row interchanges are still allowed, but so are column interchanges. We will continue to use $q \in \Re^m$ for row interchange bookkeeping. The vector $p \in \Re^n$, similarly, will maintain the column permutation information. We search not just the pivot column, but the entire Gauss transform area, for the next pivot. This takes longer but generally produces better solutions. The numerical differences between partial and complete pivoting involve some difficult error analysis. These issues will be addressed briefly after we complete the examples.

2. Example

Now the GF process is demonstrated. We start with the same system of four equations in four unknowns:

$$\begin{aligned}
2v_1 + 3v_2 + 4v_3 + 5v_4 &= 0 \\
4v_1 + 6v_2 + 8v_3 + 5v_4 &= -5 \\
2v_1 + 4v_2 + 7v_3 + 9v_4 &= 13 \\
6v_1 + 8v_2 + 8v_3 + 9v_4 &= -17
\end{aligned}$$

(3.16)

and proceed immediately to the matrix of coefficients (the factoring part of GF concerns itself with A only).

$$A = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 4 & 6 & 8 & 5 \\ 2 & 4 & 7 & 9 \\ 6 & 8 & 8 & 9 \end{bmatrix}$$

(3.17)

a. Stage Zero

For the initial stage, $k = 0$, let the Gauss transform area be $G = A$. Also initialize pivot indices $s = t = 1$. The sole purpose of stage zero is to find the first pivot. Initially, we guess that the pivot is α_{11}, located at $A(1,1)$, the upper left–hand corner of G. (This is the position where the new pivot will be installed). Accordingly, we set row and column indices, $s = 1$ and $t = 1$ to keep track of the best pivot candidate.

Indices s and t are changed only when we find a superior candidate for the pivot. To begin the column–by–column search for the pivot we move down the columns in order from left to right and through each column in a top–to–bottom manner. When we have considered every element in G, we know that the next pivot is currently situated at $A(s,t)$.

For the current example, as we move down the first column of G, the values of s and t are adjusted twice. A better pivot candidate is found, first at $A(2,1)$, and next at $A(4,1)$. The indices are adjusted again in the last row of column two,

48

where the value, 8, is larger than the value of the current candidate, 6. Column three has no candidates larger than 8, so we do not adjust the indices again until we find the 9 at $A(3,4)$. Thus $s = 3$ and $t = 4$ have located the next pivot according to a complete pivoting strategy. This accomplishes the goal of stage zero. Now we specify the process for each of the remaining stages.

b. Outline of the GF Process

For each stage, k, of GF, we shall perform the following steps:

- Locate the pivot according to a pivoting strategy (none, partial, or complete). If complete pivoting is used, search all of G for the next pivot.

- Increment the pivot index, k.

- Perform any row and/or column permutations that are required to move the pivot into the position $A(k,k)$. Update p and q accordingly.

- Divide every element beneath the pivot by the pivot value.

- Redefine the Gauss transform area so that $G = A((k+1):m, (k+1):n)$.

- Perform the appropriate arithmetic in G.

Let us return to the example and exercise the process.

c. Stage One

Since stage zero has already located the first pivot, the first step of section *b* is not necessary in this stage. We increment k (to $k = 1$) and install the pivot $A(3,4)$ at $A(k,k) = A(1,1)$. This means that rows 1 and 3 must be swapped. Columns 1 and 4 must be swapped in addition. The permutation vectors, p and q, record the interchanges.

49

After interchanging rows and columns, we have

$$A = \begin{bmatrix} 9 & 4 & 7 & 2 \\ 5 & 6 & 8 & 4 \\ 5 & 3 & 4 & 2 \\ 9 & 8 & 8 & 6 \end{bmatrix} \qquad p = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 1 \end{bmatrix} \qquad q = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix}. \tag{3.18}$$

Now we perform the division beneath the pivot, producing the multipliers in the lower three rows in the leftmost column of A. When this is done, we perform the arithmetic in $G = A((k+1):m, (k+1):n) = A(2:4, 2:4)$. For GF, we do not replace the multipliers with zeros. We shall find that the multipliers are very useful in the end. The result is

$$A = \begin{bmatrix} 9 & 4 & 7 & 2 \\ 5/9 & 34/9 & 37/9 & 26/9 \\ 5/9 & 7/9 & 1/9 & 8/9 \\ 1 & 4 & 1 & 4 \end{bmatrix}. \tag{3.19}$$

Next, with G being the lower right (3×3) block of A, we search G for the next pivot and find that $A(s, t) = A(2, 3)$ holds $(37/9)$, the largest second pivot candidate.

d. Stage Two

We increment the stage counter $(k = 2)$, so that it points to the new pivot location, $A(2, 2)$. Since $s = k$, we know that no row interchange is necessary and q will not change. We must, however, swap columns $k = 2$ and $t = 3$. The result is:

$$A = \begin{bmatrix} 9 & 7 & 4 & 2 \\ 5/9 & 37/9 & 34/9 & 26/9 \\ 5/9 & 1/9 & 7/9 & 8/9 \\ 1 & 1 & 4 & 4 \end{bmatrix} \qquad p = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix} \qquad q = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix} \tag{3.20}$$

Once again, we divide everything under the pivot by the value of the pivot and update G. This yields

$$A = \begin{bmatrix} 9 & 7 & 4 & 2 \\ 5/9 & 37/9 & 34/9 & 26/9 \\ 5/9 & 1/37 & 25/37 & 30/37 \\ 1 & 9/37 & 114/37 & 122/37 \end{bmatrix}. \tag{3.21}$$

50

e. Stage Three

Now G becomes the (2×2) lower right block of A and the next pivot $(122/37)$ is found at $A(s,t) = A(4,4)$. Since $k = 3$ we must interchange rows 3 and 4 as well as columns 3 and 4. The result of the permutation is

$$
A = \begin{bmatrix} 9 & 7 & 2 & 4 \\ 5/9 & 37/9 & 26/9 & 34/9 \\ 1 & 9/37 & 122/37 & 114/37 \\ 5/9 & 1/37 & 30/37 & 25/37 \end{bmatrix} \qquad p = \begin{bmatrix} 4 \\ 3 \\ 1 \\ 2 \end{bmatrix} \qquad q = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 1 \end{bmatrix}. \qquad (3.22)
$$

Then, dividing at the bottom of the pivot column and updating G, we have

$$
A = \begin{bmatrix} 9 & 7 & 2 & 4 \\ 5/9 & 37/9 & 26/9 & 34/9 \\ 1 & 9/37 & 122/37 & 114/37 \\ 5/9 & 1/37 & 15/61 & -15/183 \end{bmatrix} \qquad (3.23)
$$

f. Stage Four

The final stage, where $k = 4 = \min(m,n)$, is always trivial. We need only to verify that α_{44} is nonzero. This tells us that, indeed, A is nonsingular. There is no arithmetic to perform, so (3.23) is the final, factored, copy of A.

g. Summary

Using the Gauss factorization process we have systematically transformed the matrix $A \in \Re^{4 \times 4}$ into a form that factors the original version of A. At this point the factorization itself has not been discussed, only the process whereby we claim to have factored A. Before we explore the resulting factorization, let us consider—in a general way—what happens in any stage, k, of GF.

3. One Stage of Gauss Factorization

The most important part of GF is the **factorization** that it produces. The GF process is reversible (pivots and other key information become part of the

factorization). This section—using block matrix notation and induction on the stage number—illustrates the effect of one stage of GF. The proof shows that we can perform an n–step Gauss factorization $A = LR$, with L unit lower triangular and R right (upper) triangular with nonzero diagonal elements. Before the proof, however, let us consider a concrete illustration where $n = 15$.

Let $\otimes$ denote those elements that Gauss has *fixed* in both value and position. The $\times$ symbol marks elements that are subject to permutations but not changes in value. Those elements that are subject to *both* permutation and changes in value are indicated by the $\odot$ symbol. Elements in the pivot row are marked with the $\ominus$ symbol and the symbol $\oslash$ denotes elements beneath the pivot. White space indicates zeros, α is the pivot, and any ρ_i was a former pivot (in stage i). Let $k = 7$. Then the leftmost 7 columns of R_7 are already fixed in upper triangular form and L_7 is unit lower triangular with the special form described above. Upon entering stage $(k+1) = 8$ of the Gauss factorization process, the matrices L_7 and R_7 would appear as shown below:

$$L_7 = \begin{bmatrix}
1 & & & & & & & & & & & & & & \\
\otimes & 1 & & & & & & & & & & & & & \\
\otimes & \otimes & 1 & & & & & & & & & & & & \\
\otimes & \otimes & \otimes & 1 & & & & & & & & & & & \\
\otimes & \otimes & \otimes & \otimes & 1 & & & & & & & & & & \\
\otimes & \otimes & \otimes & \otimes & \otimes & 1 & & & & & & & & & \\
\otimes & \otimes & \otimes & \otimes & \otimes & \otimes & 1 & & & & & & & & \\
\ominus & \ominus & \ominus & \ominus & \ominus & \ominus & \ominus & 1 & & & & & & & \\
\times & \times & \times & \times & \times & \times & \times & & 1 & & & & & & \\
\times & \times & \times & \times & \times & \times & \times & & & 1 & & & & & \\
\times & \times & \times & \times & \times & \times & \times & & & & 1 & & & & \\
\times & \times & \times & \times & \times & \times & \times & & & & & 1 & & & \\
\times & \times & \times & \times & \times & \times & \times & & & & & & 1 & & \\
\times & \times & \times & \times & \times & \times & \times & & & & & & & 1 & \\
\times & \times & \times & \times & \times & \times & \times & & & & & & & & 1
\end{bmatrix} \tag{3.24}$$

$$
R_7 = \begin{bmatrix}
\rho_1 & \otimes & \otimes & \otimes & \otimes & \otimes & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & \rho_2 & \otimes & \otimes & \otimes & \otimes & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & \rho_3 & \otimes & \otimes & \otimes & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & & \rho_4 & \otimes & \otimes & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & & & \rho_5 & \otimes & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & \rho_6 & \otimes & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & \rho_7 & \otimes & \times & \times & \times & \times & \times & \times & \times \\
 & & & & & & & \alpha & \ominus & \ominus & \ominus & \ominus & \ominus & \ominus & \ominus \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot \\
 & & & & & & & \oslash & \odot & \odot & \odot & \odot & \odot & \odot & \odot
\end{bmatrix}
\tag{3.25}
$$

With this illustration in mind, let us prove the effect of GF.

Proposition: Given $A \in \Re^{n \times n}$. Let $L_i \in \Re^{n \times n}$ be the unit lower triangular matrix with I_{n-i}—the $(n-i) \times (n-i)$ identity—as its lower, right–hand block. Let $R_i \in \Re^{n \times n}$ be the matrix that is upper right triangular in its leftmost i columns. Initially, let $A = L_0 R_0$ with $L_0 = I$ and $R_0 = A$. Let $P(k)$ be the proposition: "Stage k of the Gauss factorization process yields the factorization, $A = L_k R_k$."

To Show: $P(k) \Rightarrow P(k+1)$ for $0 \le k \le (n-1)$.

Assumptions: Pivoting, according to any valid strategy, is performed outside of this factorization procedure and the pivoting strategy yields pivots, $\alpha \ne 0$.

Notation: We can partition A so that

$$A = \begin{bmatrix} \alpha & y^T \\ x & G \end{bmatrix} \tag{3.26}$$

where $\alpha \in \Re$ is the initial pivot, $x \in \Re^{n-1}$ holds the values beneath the pivot, $y \in \Re^{n-1}$ holds the values of the elements in the pivot row to the right of the pivot, and $G \in \Re^{(n-1) \times (n-1)}$ is the Gauss transform area.

Basis for Induction: We must show that $P(0) \Rightarrow P(1)$. $P(0)$ means that $L_0 = I_n$ and $R_0 = A$. That is, R_0 has no special structure except (by assumption) we are guaranteed a nonzero pivot α. Consider stage $k = 1$ of Gauss factorization. Let us partition A as above and factor

$$A = \begin{bmatrix} \alpha & y^T \\ x & G \end{bmatrix} = \begin{bmatrix} 1 & 0^T \\ \ell & I \end{bmatrix} \begin{bmatrix} \rho & r^T \\ 0 & B \end{bmatrix} = L_1 R_1 \tag{3.27}$$

where B, ℓ, r, and ρ (with the obvious sizes) are defined as

$$\rho = \alpha \tag{3.28}$$

$$r = y \tag{3.29}$$

$$\ell = \left(\frac{1}{\rho} \right) x \tag{3.30}$$

$$B = G - \ell r^T \tag{3.31}$$

Thus, given $A = L_0 R_0$, Gauss factors $A = L_1 R_1$ and $P(0) \Rightarrow P(1)$.

Inductive Step: Consider the matrices L_k and R_k that are submitted to stage $(k+1)$ of a Gauss factorization procedure. We make the inductive step to show that $P(k) \Rightarrow P(k+1)$. For $0 \le k \le n$, $A = L_k R_k$ may be partitioned so that

$$A = \begin{bmatrix} L & 0 & 0 \\ m^T & 1 & 0 \\ N & 0 & I \end{bmatrix} \begin{bmatrix} R & s & T \\ 0^T & \alpha & y^T \\ 0 & x & G \end{bmatrix} = L_k R_k \tag{3.32}$$

54

where $L \in \Re^{k \times k}$ is a unit lower triangular matrix and $R \in \Re^{k \times k}$ is a right (upper) triangular matrix with nonzero diagonal elements.

The Gauss process forms ρ as in (3.28), r as in (3.29), multipliers, ℓ as in (3.30), and B as in (3.31). Then, for $0 \leq k \leq (n-1)$, GF forms

$$A = \begin{bmatrix} L & 0 & 0 \\ m^T & 1 & 0 \\ N & \ell & I \end{bmatrix} \begin{bmatrix} R & s & T \\ 0^T & \rho & r^T \\ 0 & 0 & G \end{bmatrix} = L_{k+1} R_{k+1}. \tag{3.33}$$

Thus, for $0 \leq k \leq n$, $P(k) \Rightarrow P(k+1)$. [Ref. 24]

■

Conclusion: The nonsingular matrix $A \in \Re^{n \times n}$ can be factored, in n steps of the Gauss factorization process, so that $A = LR$ with L being unit lower triangular and R being upper triangular with nonzero diagonal elements.

The proof has demonstrated the effect of GF. For simplicity, it excluded the pivoting strategy (simply assuming that, at every stage, a pivot $\alpha \neq 0$ would be available). It also held A square. In this sense the proof is somewhat specific. There is a more general conclusion to be made. This conclusion holds for GF with pivoting and $0 \neq A \in \Re^{m \times n}$ and it is absolutely essential to understanding the factorization.

4. The LR Theorem

With the GF process complete, and the vast majority of the work done, we show how to form a solution from our factorization. Various methods of pivoting (resulting in permutation vectors) and the method whereby A is factored have been discussed. To solve the system, we must put all of this information together. The key is the **LR Theorem** [Ref. 24]:

Theorem 3.1 (LR Theorem) *Let $0 \neq A \in \Re^{m \times n}$. Then there are permutation matrices $P \in \Re^{n \times n}$ and $Q \in \Re^{m \times m}$, an integer $r \geq 1$, a lower trapezoidal matrix $L \in \Re^{m \times r}$ and an upper (right) trapezoidal matrix $R \in \Re^{r \times n}$ so that $Q^T A P = LR$. The diagonal elements of L satisfy $\lambda_{i,i} = 1$ with $i = 1, 2, \ldots, r$ and the diagonal elements of R satisfy $\rho_{i,i} \neq 0$ for $i = 1, 2, \ldots, r$.*

5. Filling in the Blanks

a. The Main Factors

GF used the space of A to hold the two principal matrices, L and R, in the factorization of A. To see them, we will extract the lower triangular matrix, L, and upper (right) triangular matrix, R, from the final copy of A (3.23). Initially, let $L = R = 0$. We form L by placing ones on its diagonal and filling the elements below the diagonal from the corresponding locations in A.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5/9 & 1 & 0 & 0 \\ 1 & 9/37 & 1 & 0 \\ 5/9 & 1/37 & 15/61 & 1 \end{bmatrix} \tag{3.34}$$

R is formed with the diagonal elements (i.e., pivots) and upper triangle of A.

$$R = \begin{bmatrix} 9 & 7 & 2 & 4 \\ 0 & 37/9 & 26/9 & 34/9 \\ 0 & 0 & 122/37 & 114/37 \\ 0 & 0 & 0 & -15/183 \end{bmatrix} \tag{3.35}$$

b. Permutation Matrices

The bookkeeping allows us to construct P and Q very quickly. To form $P \in \Re^{n \times n}$, we set every column, j, in P equal to the axis vector implied by π_j, the jth element of p. This yields the permutation matrix, P, that will satisfy the LR Theorem, namely

$$p = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 1 \\ 2 \end{bmatrix} \implies P = \begin{bmatrix} e_4 & e_3 & e_1 & e_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{3.36}$$

Similarly, every column, j, in $Q \in \Re^{m \times m}$ is set equal to the axis vector implied by ψ_j, the jth element of q. For our example, we have

$$q = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 1 \end{bmatrix} \implies Q = \begin{bmatrix} e_3 & e_2 & e_4 & e_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{3.37}$$

c. Check

Now we check to make sure that our solution satisfies the LR Theorem.
First, consider the product LR:

$$LR = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5/9 & 1 & 0 & 0 \\ 1 & 9/37 & 1 & 0 \\ 5/9 & 1/37 & 15/61 & 1 \end{bmatrix} \begin{bmatrix} 9 & 7 & 2 & 4 \\ 0 & 37/9 & 26/9 & 34/9 \\ 0 & 0 & 122/37 & 114/37 \\ 0 & 0 & 0 & -15/183 \end{bmatrix} \tag{3.38}$$

$$= \begin{bmatrix} 9 & 7 & 2 & 4 \\ 5 & 8 & 4 & 6 \\ 9 & 8 & 6 & 8 \\ 5 & 4 & 2 & 3 \end{bmatrix} \tag{3.39}$$

And

$$Q^T A P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 & 5 \\ 4 & 6 & 8 & 5 \\ 2 & 4 & 7 & 9 \\ 6 & 8 & 8 & 9 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{3.40}$$

$$= (Q^T A) P = \begin{bmatrix} 2 & 4 & 7 & 9 \\ 4 & 6 & 8 & 5 \\ 6 & 8 & 8 & 9 \\ 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{3.41}$$

$$= \begin{bmatrix} 9 & 7 & 2 & 4 \\ 5 & 8 & 4 & 6 \\ 9 & 8 & 6 & 8 \\ 5 & 4 & 2 & 3 \end{bmatrix} \tag{3.42}$$

Our factorization satisfies $Q^T A P = LR$.

d. Solution

Now we solve the system. Recall that Gaussian elimination operated on the matrix, A, *and* the right–hand side, b, at the same time. The end result of GE is that A is reduced to upper triangular form by successive elimination of the lower triangle so that we could solve for u with a relatively easy back substitution.

The strategy of Gauss factorization is different. First, b is not part of the factorization process. Secondly, even though we are changing A, we know that

we can get it back at the end (if we want to), so there is no need to save the original A. Now, using the LR Theorem, we complete the solution. Recall that the original system was

$$Au = b. \tag{3.43}$$

The factorization process constructs permutation matrices P and Q and transforms the original matrix A into a combined version of L and R. Further (by the LR Theorem) we know that these matrices satisfy

$$Q^T A P = LR. \tag{3.44}$$

Now, by multiplying (3.44) through by Q from the left and P^T on the right, we see that

$$QQ^T A P P^T = QLRP^T. \tag{3.45}$$

Performing the cancellations on the left–hand side, we have

$$A = QLRP^T. \tag{3.46}$$

This is the factorization of A. Substituting this into (3.43) yields

$$QLRP^T u = b \tag{3.47}$$

or

$$LRP^T u = Q^T b. \tag{3.48}$$

Now let $\tilde{b} = Q^T b$ and let $\tilde{u} = P^T u$. Then

$$LR\tilde{u} = \tilde{b}. \tag{3.49}$$

Further, let $R\tilde{u} = c$ for some unknown vector, c. We have

$$Lc = \tilde{b}. \tag{3.50}$$

58

Since we know L and b, we may solve for c by a simple forward substitution. Then, using c and knowing that $R\tilde{u} = c$, we perform a simple back substitution and determine $\tilde{u}$. Finally, by definition, $\tilde{u} = P^T u$ (i.e., $\tilde{u}$ is a mere permutation of u) so we can swap elements in $\tilde{u}$ to arrive at u using $P\tilde{u} = u$.

Let us summarize this lengthy process into the main steps. The GF process factors $A = QLRP^T$, changing the general matrix into a product where the most significant factors are both triangular. This reduces the hard problem to two easy ones. It is designed so that we can solve for u in two steps:

- Solve, by forward substitution, the system $Lc = \tilde{b}$ for a vector, c, of unknowns.

- Solve, by back substitution, the system $R\tilde{u} = c$ for (a permutation of) the original unknowns, u.

So, for our example, the first step is to solve

$$Lc = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5/9 & 1 & 0 & 0 \\ 1 & 9/37 & 1 & 0 \\ 5/9 & 1/37 & 15/61 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = Q^T b = \begin{bmatrix} 13 \\ -5 \\ -17 \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{\beta}_1 \\ \tilde{\beta}_2 \\ \tilde{\beta}_3 \\ \tilde{\beta}_4 \end{bmatrix} = \tilde{b} \quad (3.51)$$

Forward substitution, applied to this system, yields

$$c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 13 \\ -110/9 \\ -1000/37 \\ -15/61 \end{bmatrix} \quad (3.52)$$

59

Now we know c, so we can solve the second triangular system, $R\tilde{u} = c$ for $\tilde{u}$ by back substitution

$$R\tilde{u} = \begin{bmatrix} 9 & 7 & 2 & 4 \\ 0 & 37/9 & 26/9 & 34/9 \\ 0 & 0 & 122/37 & 114/37 \\ 0 & 0 & 0 & -15/183 \end{bmatrix} \begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{bmatrix} = \begin{bmatrix} 13 \\ -110/9 \\ -1000/37 \\ -15/61 \end{bmatrix} = c \qquad (3.53)$$

which yields

$$\tilde{u} = \begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ -11 \\ 3 \end{bmatrix} \qquad (3.54)$$

Now it is easy to recover u. Since we have defined $\tilde{u} = P^T u$, we know that $P\tilde{u} = u$ (a simple rearrangement of the elements that we have already found). We apply P to $\tilde{u}$ and find that

$$P\tilde{u} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \tilde{v}_1 \\ \tilde{v}_2 \\ \tilde{v}_3 \\ \tilde{v}_4 \end{bmatrix} = \begin{bmatrix} \tilde{v}_3 \\ \tilde{v}_4 \\ \tilde{v}_2 \\ \tilde{v}_1 \end{bmatrix} = \begin{bmatrix} -11 \\ 3 \\ 2 \\ 1 \end{bmatrix} \qquad (3.55)$$

Comparing this to earlier solutions, we find that GF has arrived at the same solution.

In these examples, the notion of elimination was developed first. The GE process performs successive eliminations beneath its pivots and reduces A to triangular form, and then the solution is available in only n^2 flops. GF spends an almost identical amount of work in the reduction process, but the result is a factorization with L and R being the significant factors. (They are the only ones that are more than a permutation of the identity). In the examples, we used pivoting because it was practical. Now let us take a closer look at the justifications for pivoting.

F. PIVOTING FOR SIZE

The issue of pivoting is a very interesting and important one. We concluded that we must pivot or face the possibility of attempting to divide by zero, an unacceptable option. To solve this problem, we may pick *any* nonzero element in $A(k:m, k:n)$ and perform the column and row interchanges required to install it as the new pivot (k is the pivot index). There are many strategies that we could adopt.

The logical question would be something like: "Given that we must pivot, what is the best means available?" But the answer is not so easy, and there are many trade–offs to be considered. We are faced with choosing along a spectrum, where *speed* lies at one end and *accuracy* lies at the other. For instance, we could begin a search and pick the *first* nonzero element in this area. Or, we could search for the row with the *most* nonzero elements (that had a nonzero element in the kth column).

The two most common strategies for pivoting are the *partial* and *complete* methods, which we have discussed. We determined that partial pivoting would work *perfectly* (with no error) if A was nonsingular and the storage and arithmetic could be handled with infinite precision. If infinite precision were available, we could stop right here. There would be no need to try to refine the method. In a finite–precision machine, however, we must deal with the issue of errors.

To deal with errors, the problem must be stated more precisely. The errors that concern us would arise due to *growth* of the elements of L and/or R as we step through the stages of Gauss. In the end, partial pivoting guarantees that all of the elements of L will be, at most, unity. This is easy to see. The pivoting strategy chooses each pivot to be the largest element (in absolute value) in column k at or below row k. This value is installed at $A(k, k)$ and everything below the pivot is divided by the pivot.

Unfortunately, partial pivoting cannot make the same guarantee for the elements of R. It helps: the *multipliers* are less than or equal to one in absolute value. The elements of R are bounded by $2^{n-1}a$, where a is the largest absolute value of the elements in A. This bound is not normally attained "in practice". [Ref. 23]

Growth is an indicator of trouble in this process. If we cannot control it completely, we should, at a minimum, *monitor* it. The *growth factor*, $g(n)$, of a Gauss factorization process for $A \in \Re^{n \times n}$ is defined as follows. Let a be the largest absolute value in the original matrix, A. Let b be the largest absolute value that occurs in any Gauss transform, G, including the first one, $G = A$. Then $g(n) = b/a$ gives a growth factor normalized by a (i.e., $g(n) \geq 1$).

A great deal of analysis has been done on this subject. Wilkinson showed that, with *complete pivoting* and *real matrices*, $g(n)$ grows much more slowly than 2^n. He conjectured that $g(n) \leq n$. The latter has recently been disproved, with a counterexample by Nicholas Young. [Ref. 23]

As a practical matter, when one seeks to monitor growth one uses complete pivoting. To consider performance, one uses the partial pivoting strategy. The growth factor, $g(n)$, is easy to monitor with a complete pivoting strategy since we are moving through the entire Gauss transform area at each stage anyway. For clarity, the pivoting algorithms and the **Update** algorithm are listed separately in this chapter. In real code (e.g., Appendix F), however, the pivot for stage $(k+1)$ should be located during the update of G in stage k (to avoid unnecessary passes through the matrix). This would mean extra work in the partial pivoting algorithm. Since the primary reason for using partial pivoting is performance, it is counterproductive to monitor $g(n)$ while using partial pivoting. A description of both pivoting policies, in algorithm form, follows.

Algorithm 3.1 (Partial Column Pivoting for Size) *Given the matrix of coefficients, $A \in \Re^{m \times n}$; a permutation vector, $q \in \Re^m$; and an index, k, indicating the pivot column, this algorithm performs partial pivoting. First, the pivot element is located at $A(s, k)$ with $s \geq k$. Once the pivot has been located, rows s and k are swapped to install the new pivot. Additionally, elements in q, indexed by s and k, are swapped to record the row interchanges.*

begin PP

    $s = k$;

    for $i = (k + 1) : m$

        if $(|A(i, k)| > |A(s, k)|)$

            $s = i$;

        end if

    end for

    if $(s \neq k)$

        for $j = 1 : n$

            $x = A(k, j)$;

            $A(k, j) = A(s, j)$;

            $A(s, j) = x$;

        end for

        $i = q(k)$;

        $q(k) = q(s)$;

        $q(s) = i$;

    end if

end PP

Algorithm 3.2 (Complete Pivoting for Size) *Given the matrix of coefficients,* $A \in \Re^{m \times n}$; *permutation vectors,* $p \in \Re^n$ *and* $q \in \Re^m$; *and an index,* k, *indicating the pivot row and column, this algorithm performs complete pivoting. First, the pivot element is located at* $A(s,t)$. *Once the pivot has been located, rows* s *and* k *and columns* t *and* k *are swapped to install the new pivot. The permutation vectors are updated accordingly.*

begin PC

    $s = k$;

    $t = k$;

    for $i = k : m$ (locate the pivot)

        for $j = k : n$

            if $(|A(i,j)| > |A(s,t)|)$

                $s = i$;

                $t = j$;

            end if

        end for

    end for

    if $(s \neq k)$ (row interchanges)

        for $j = 1 : n$

            $x = A(k,j)$; $A(k,j) = A(s,j)$; $A(s,j) = x$;

        end for

        $i = q(k)$: $q(k) = q(s)$; $q(s) = i$;

    end if

    if $(t \neq k)$ (column interchanges)

        for $i = 1 : m$

            $x = A(i,k)$; $A(i,k) = A(i,t)$; $A(i,t) = x$;

        end for

        $i = p(k)$; $p(k) = p(t)$; $p(t) = i$;

    end if

end PC

G. SEQUENTIAL ALGORITHMS

The examples considered have described the Gauss process. We first considered elimination (GE) and then a factorization method (GF). Both methods require work of the same order, so the latter, yielding a *factorization* of A is much preferred. Algorithms for the GF process are described below. The arithmetic in the Gauss transform area, G, is performed the same (regardless of pivoting strategy) so a separate algorithm is given for updating G. The algorithms GFPP (pivoting, partial) and GFPC (pivoting, complete) are given following the updating algorithm. These algorithms are adapted from Gragg [Ref. 23].

Algorithm 3.3 (Update Gauss Transform Area) *Given the matrix of coefficients, $A \in \Re^{m \times n}$; and k, the pivot column, this algorithm performs the appropriate arithmetic throughout the pivot column and Gauss transform area, G, of A.*

begin Update

    $x = A(k, k);$ (x is the pivot value)

    for $i = (k + 1) : m$ (pivot column division)

        $A(i, k) = A(i, k)/x;$

    end for

    for $i = (k + 1) : m$ (arithmetic in G)

        $x = A(i, k);$ (now x is the multiplier)

        for $j = 1 : n$

            $A(i, j) = A(i, j) - x \times A(k, j);$

        end for

    end for

end Update

Algorithm 3.4 (Gauss Factorization with Partial Pivoting) *Given the matrix of coefficients, $A \in \Re^{n \times n}$, this algorithm modifies (overwrites) A with a unit lower triangular matrix (with an implicit diagonal), $L \in \Re^{n \times n}$, and an upper (right) triangular matrix, $R \in \Re^{n \times n}$ having nonzero diagonal elements (the pivots). The process also forms the row permutation vector, q, and the corresponding permutation matrix, $Q \in \Re^{n \times n}$, that results from partial column pivoting for size. The algorithm gives the factorization: $Q^T A = LR$.*

begin GFPP

    $n = \text{order}(A)$

    $Q = \text{zeros}(n, n)$

    for $j = 1 : n$

        $q(j) = j;$ (initialize q)

    end for

    for $r = 1 : n$ (the Gauss process)

        PP(A, q, k) (pivoting)

        if $(A(k, k) = 0)$

            print "A is singular!"

            exit

        end if

        Update(A, k) (Update G)

    end for

    for $j = 1 : n$

        $Q(q(j), j) = 1.0;$

    end for

end GFPP

Algorithm 3.5 (Gauss Factorization with Complete Pivoting) *Given a matrix of coefficients, $A \in \Re^{m \times n}$, the following algorithm modifies (overwrites) A with a unit lower trapezoidal matrix (with implicit diagonal), $L \in \Re^{m \times n}$, and an upper (right) trapezoidal matrix, $R \in \Re^{m \times n}$. The diagonal elements of R are nonzero (pivots). The process forms permutation matrices, $P \in \Re^{n \times n}$ and $Q \in \Re^{m \times m}$, to reflect the complete pivoting for size. These matrices are formed to satisfy the LR Theorem: $Q^T A P = LR$.*

begin GFPC

    $m = \text{rows}(A);$ $n = \text{cols}(A);$ (initialization)

    $P = \text{zeros}(n, n);$ $Q = \text{zeros}(m, m);$

    for $j = 1 : n$

        $p(j) = j;$

    end for

    for $i = 1 : m$

        $q(i) = i;$

    end for

    for $r = 1 : n$ (the Gauss process)

        PC(A, q, k) (pivoting)

        if $(A(k, k) = 0)$

            print "*A* is singular!"

            exit

        end if

        Update(A, k) (Update G)

    end for

    for $j = 1 : n$

        $P(p(j), j) = 1.0;$ (form P)

    end for

    for $j = 1 : m$

        $Q(q(j), j) = 1.0;$ (form Q)

    end for

end GFPC

H. CONJUGATE GRADIENTS

Time permits only a brief synopsis of the method of conjugate gradients (CG). This method was described by Magnus R. Hestenes and Eduard Stiefel [Ref. 18]. CG possesses some very nice characteristics and it is quite different from the Gauss method. Once again, we begin with a system of linear equations

$$Au = b \qquad (3.56)$$

The algorithm given by Hestenes and Stiefel is designed for $A \in \Re^{n \times n}$ *symmetric* and *positive definite* (Appendix A). Let $s \in \Re^n$ be the vector that would solve (3.56) exactly, so that $As = b$. Let $u_i \in \Re^n$ be the estimate of the solution, s, produced in the ith iteration. The original estimate, u_0, is merely a guess (it may be a good guess). For instance, in the absence of better information, we could choose u_0 to be the vector of all zeros or all ones.

The CG process takes our initial guess and develops a (guaranteed) better estimate for the next stage. To measure the progress, we could use the *residual* vector

$$r_i = b - Au_i \qquad (3.57)$$

but Hestenes and Stiefel warn that its Euclidean norm, $\| r_i \|_2$, may actually *increase* in every step but the last! A more reliable measure, called the *error vector*

$$e_i = s - u_i \qquad (3.58)$$

has monotonically decreasing length. After n iterations of the CG process, we are guaranteed to have a very good estimate u_n of s. In fact, if no rounding errors occur, we have $u_n = s$. In practice, CG can find a very good estimate, u_m, of s in m iterations, with $m \ll n$. The process "terminates in at most n steps if no rounding–off errors are encountered." [Ref. 18: p. 410]

The algorithm below is adopted from Hestenes and Stiefel [Ref. 18]. Before considering the algorithm, however, we should define the key term, *conjugate*. For A symmetric, two vectors $x \in \Re^n$ and $y \in \Re^n$ are said to be *A–orthogonal* (or *conjugate*) if the relation $x^T A y = (Ax)^T y = 0$ holds [Ref. 18 : p. 410]. This is an extension of vector orthogonality, $x^T y = 0$. The algorithm given below is very simple. The iteration blindly proceeds from $i = 0$ to $i = n$. A more sophisticated (finite precision) scheme would set a tolerance (notion of "good enough") and stop (exit the loop) when this criterion was satisfied.

Algorithm 3.6 (The Method of Conjugate Gradients) *Given the symmetric, positive definite matrix of coefficients, $A \in \Re^{n \times n}$; and an initial guess, u_0; for the solution, s; of the system $Au = b$, this algorithm (in the absence of rounding-off errors) finds $u_i = s$ in i iterations $(i \leq n)$. The algorithm keeps track of a residual vector, r_i, and direction vectors, p_i. The residuals, r_i, are mutually orthogonal and the direction vectors, p_i are mutually conjugate (A–orthogonal).*

begin CG

    $u_0 = \text{zeros}(n)$ (arbitrary initial guess)

    $p_0 = r_0 = b - Au_0$

    for $i = 0 : n$

        $\delta = p_i^T A p_i$ (denominator used below)

        $\alpha_i = (p_i^T r_i)/\delta$ (scalar multiplier used below)

        $u_{i+1} = u_i + \alpha_i p_i$ (estimate of solution)

        $r_{i+1} = r_i - \alpha_i A p_i$ (residual vector)

        $\beta_i = (r_{i+1}^T r_i)/\delta$

        $p_{i+1} = r_{i+1} + \beta_i p_i$ (direction vector)

    end for

end CG

I. SUMMARY

This chapter develops the Gaussian elimination process, the Gauss factorization process, pivoting strategies, and (briefly) the method of conjugate gradients. Each of the corresponding algorithms possesses potential for parallel solution. A parallel implementation of GF appears in the following chapter. Both partial and complete pivoting are pursued, with further discussion on their implications in a parallel environment.

IV. PARALLEL DESIGN

Nature is pleased with simplicity, and affects not the pomp of superfluous causes.

— *SIR ISAAC NEWTON (1642-1727)*

Sequential algorithms for Gauss factorization (GF) and the method of conjugate gradients (CG) are established in Chapter III. The goal of this chapter is to show *parallel* algorithms for Gauss factorization. The C programs that implement these algorithms are discussed in Chapter V and listed in Appendix F.

Parallel algorithm design is a process that includes many considerations. The question of how to achieve parallelism is largely an art and is not discussed here. The method used in this research is often called a *workfarm* approach because the algorithm farms out work to processors. Equivalently, it may be called a *manager-worker* model. When we distribute the problem across many processors in a workfarm style, there are quite a number of issues that warrant careful consideration. The concerns associated with programming a parallel machine—even with a relatively simple model such as this—could occupy volumes.

Communications, load balancing, granularity, and other considerations abound. Metrics like speedup and efficiency should be used to lend credibility to the parallel nature of the algorithm. Additionally, we should consider the usual issues of maintainability, readability, portability, and other traits commonly associated with good (sequential) programming practice. Parallel codes must be clear combinations of sequential codes that are joined together in a logical manner. Simplicity should hold a place of great esteem in a parallel algorithm. The rest of this chapter *introduces* the issues of parallel design, particularly as they pertain to Gauss factorization.

A. INTERPROCESSOR COMMUNICATIONS

Interprocessor communication is one of the most fundamental issues in parallel processing and, quite possibly, the most involved. Without a means of communicating (in a message–passing environment), the multiprocessor system is meaningless. The implications of any communications scheme are many and the interactions can be quite complex. Exhaustive coverage of this issue is out of the question, so we will consider a few of the most essential ideas.

1. The Network

A network is the part of a multiprocessor system's hardware that bears the interprocessor communications burden. It is a combination of *nodes* and *links* that connect those nodes, and it is the foundation upon which all communications must build. We will also refer to the nodes of a multiprocessor—using somewhat loose terminology—as processors. The term *node* is a more general term. Nodes are typically more sophisticated than a simple central processing unit (CPU) or, for that matter, any other sort of processor. The *link* is a wire that connects two nodes. An *interconnection topology* describes the pattern of links used to connect the nodes of a network. The network can be drawn or illustrated so that we can see how its nodes are connected. Appendix C discusses interconnection topologies and it gives a description (and illustrations) of the particular scheme used in this research: the *hypercube*.

Intel combines an 80386 CPU with an 80387 math coprocessor and communications facilities to form a "CX" node for the iPSC/2 that was used in this research. INMOS provides the same general capabilities but packages it all on a (very sophisticated) single chip, called a *transputer*. Figure 4.1, from INMOS' T9000 Transputer Products Overview Manual [Ref. 25: p. 31], shows a high–level block diagram of the

components of a T9000 transputer. Thus, any node of a message–passing multiprocessor system can be thought of as a combination of computing and communications facilities. It may possess other capabilities as well.

2. Message Routing

The machines used in this research exhibit different message transmission schemes. The transputer system employs high–speed (20 megabits per second) point–to–point serial communications and *store–and–forward* message passing. That is, for multi–hop communications, each node along the way must receive the message, store it in local memory temporarily, and then pass it to the next node in the route.

The Intel iPSC/2 uses another technique, called *circuit switching* or *direct–connect* communications. This approach is much like our telephone system. First, the originator of the message sends a small message containing information about the message (e.g., destination node number, length of message) to the destination via the nodes in–between. As this small header packet makes its way to the destination the nodes along the way flip switches, closing a circuit from the sender to the receiver. Once this circuit is established, the message proceeds from the sender to the destination without interruption.

Each method has its advantages and disadvantages. The circuit switching approach allows for fewer interruptions along the way, but it ties up the entire path for the duration of the communication. The store–and–forward method imposes delays for storing the message into, and then retrieving it from, the memory of every node along the way. (A more complete description of these two techniques, together with experimental results, is given in Appendix B). For the algorithms employed in this research, almost all communications were "nearest neighbor" in the hypercube. In this case, the two approaches to message routing are insignificant and the nearest neighbor performance becomes more important.
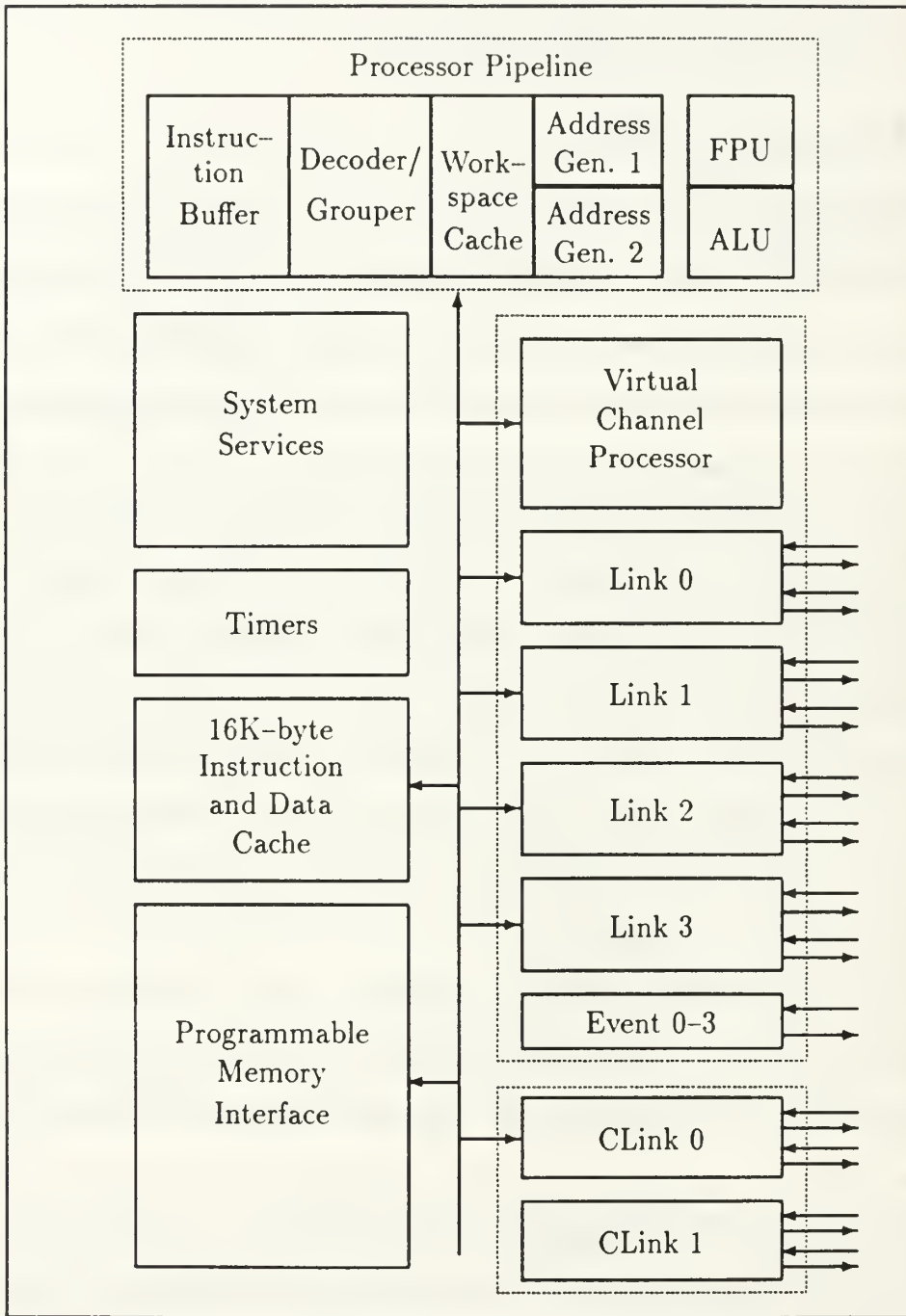
Figure 4.1: IMS T9000 Block Diagram

74

3. Concurrent Computing and Communicating

The nodes of a multiprocessor machine should be able to both compute and communicate *efficiently* and *concurrently*. This is no small undertaking. The computing side must access memory to accomplish its mission, but the message–passing begins by drawing data out of memory and ends by storing data into memory. Therefore, at a minimum, we have competition related to memory accesses. Furthermore, the computing and communication must be synchronized to some extent. The algorithms used in this research used *blocking communications*—described in Appendix E—which enforces synchronization.

There are overheads associated with communications and this synchronization problem. Bryant showed how transputers perform under various communication loads [Ref. 26] and this is mentioned in Appendix E. The issue of overheads is one that Charles Seitz considered for the "Cosmic Cube." Much, but not all, of the overhead is communication–related. Seitz listed three of the major problems [Ref. 27: p. 28]:

> *(1) the idle time that results from imperfect load balancing, (2) the waiting time caused by communications latencies in the channels and in the message forwarding, and (3) the processor time dedicated to processing and forwarding messages, a consideration that can be effectively eliminated by architectural improvements in the nodes.*

Included in these costs, we should also recognize that some amount of time is required for the processor to perform "context switching" (changing jobs) and/or coordination with a special–purpose processor that we might call the *communications manager*.

Although the issue of concurrent communication and computing is a very complex one, we may consider significant issues that are related to the efficiency of communications and the effect upon the processor. Geoffrey Fox presents the notion of comparing communications ability to processing ability [Ref. 28: pp. 50–51]. Let t_{calc} be "the typical time required to perform a generic calculation. For scientific

problems, this can be taken as a floating–point calculation $a = b \times c$ or $a = b + c$."
Furthermore, let t_{comm} be "the typical time taken to communicate a single word between two nodes connected in the hardware topology." Then the ratio

$$\frac{t_{comm}}{t_{calc}}$$

is a general *characteristic of a particular system* that can be quite useful in comparing machines. Fox uses this ratio in much of the rest of his work.

A parallel machine must necessarily possess a capable communications subsystem, but this is not enough. The *program* should also make prudent use of the communications facilities. This means that the programmer and/or compiler must exhibit a good understanding the machine's communications abilities and weaknesses. Some characteristics are nearly universal. Most machines, for instance, reward the use of *long messages* because there is an overhead—nearly independent of message length in many cases—to sending any message. Other characteristics are very much machine–dependent. This means that the programmer should be relatively familiar with the communications abilities and characteristics of the target machine.

4. Accessing the Clock

The ability to accurately measure the *time* required by communications and computations, preferably at the host and every node in the system, is absolutely essential in a multiprocessor environment. *Profiling*, in a sequential program, allows us to compare the time required by various parts of a program. Timing in a parallel environment allows us profile the code. Thus we can determine the time required for instructions, loops, functions, or communications.

Profiling is an even more important practice for parallel coding than it is in the sequential case. The *only way* for a parallel program to be useful is if it can be

can be implemented efficiently upon an acceptable number of processors. That is, in general, the *only* object in choosing a multiprocessor system over a sequential machine is the *speed* with which computation can be performed. One of the best tools available to the parallel programmer is the ability to see where and how much time is being spent.

At a minimum, we need the ability to sample a clock with reasonable precision. Both machines and compilers used in this research provide this capability (see **timing.h** in Appendix F for details). The transputers offer a choice of frequencies: the clock associated with low priority processes has a period of 64 microseconds and the high priority clock offers one microsecond *ticks*. The iPSC/2 **mclock()** function gives time in milliseconds.

B. METRICS FOR PARALLEL COMPUTING

1. Complexity

Perhaps the most obvious measures for a parallel algorithm are simply those that we use for sequential algorithms. We want to keep time and storage requirements to a minimum. Perhaps the major difference in complexity analysis for a parallel algorithm is that we are primarily interested in a per–processor notion of complexity. If the problem has been farmed out in a *fair* manner, complexity analysis for the parallel case is merely an extension of the sequential case.

Consider the matrix $A \in \Re^{n \times n}$. Suppose that its elements are 8–byte, double–precision, floating–point values (type **double** in C). Let M_p denote the total memory (in bytes) required to store A on p processors and let T_p denote the time required for p processors to solve the system characterized by A. Then $M_1 = 8n^2$ bytes of storage, but (ideally) $M_8 = n^2$. When the problem is distributed across p processors simultaneously, the processors can share the storage burden.

77

Exceptions abound. For certain problems, it may actually be convenient (faster or more reliable) to store the entire matrix at each processor. Nevertheless, in most cases we would like to minimize local memory requirements. The Gauss factorization algorithm considered near the end of this chapter is no exception. Indeed, the transputers used in this work had only 32 kilobytes of storage each and the results of Chapter VI for transputers show how this can dictate the size of the problem that can be executed. The concepts of *time* and *storage* complexity have been developed in detail for sequential algorithms and they seem to hold a place in parallel algorithm assessment as well. We consider other measures that have been developed for parallel computing in the following section.

2. Contemporary Measures

The concepts of speedup and efficiency (Appendix A) are two of the most common performance measures currently associated with parallel computing, with the ideal case (100% efficiency) yielding $t_P = t_1/P$ on a P–processor system. Selim Akl proposes the following criteria for analyzing algorithms [Ref. 29: pp. 21–28]:

- **Running Time**: Running time $t(n)$ is the time required to execute an algorithm for a problem of input size n. Akl lists three ways to express this notion. First, we may count the steps in an algorithm. Akl distinguishes between *computational* steps (i.e., something like flops) and *routing* steps that are associated with interprocessor communication. Second, we have lower and upper bounds (e.g., the complexity notation presented in Appendix A). Finally, we have *speedup*. Akl gives the usual definition of speedup but clarifies it somewhat (details below).

- **Number of Processors**: Second in importance, Akl considers the number of processors required by an algorithm. He uses $p(n)$ to denote the number of processors required for a problem of size n.

- **Cost**: Akl defines the *cost*, $c(n)$ for a parallel algorithm as the product of the first two factors. That is, $c(n) = t(n) \times p(n)$.

- **Other Measures**: In this category, we have no less than three other qualities of a parallel system that deserve consideration. The *area* (i.e., chip real estate) required by the processors is significant. The *length* of the links, as well as any patterns figures in (regularity and modularity). And finally, the *period* between processing different elements of an input is important.

Apparently metrics for parallel computing are still developing. There are several very useful concepts such as speedup and efficiency. The definition of speedup, at a first glance, is rather standard. It doesn't take much probing, however, to find that different authors make different assumptions. Akl defines speedup S in the usual manner.

$$S = \frac{t_1}{t_P} \tag{4.1}$$

except that he is somewhat more specific about the times. He defines t_1 as the "worst–case running time of fastest known sequential algorithm for problem" and t_P as "worst–case running time of parallel algorithm." [Ref. 29 : p. 24] He has been more specific than most authors, but it seems likely that the algorithms, method of obtaining times t_1 and t_P, and systems should also be specified. Speedup is defined loosely in most cases. A parameterization to accompany speedup would be tedious, but useful. Until speedup becomes a standard term with accepted meaning, we shall have to specify exactly what it means. We should be more careful with this term.

3. Other Ideas

Akl has appropriately distinguished between *computational* steps and *routing* steps. The term *floating–point operations* (flops) has become quite popular (along with *benchmarks*) and this is a useful means of expressing the computational ability of a machine (for floating–point applications). The notion of *routing*, however, is somewhat vague. Nevertheless, this idea must be addressed. It should probably become more specific as we talk about similar machines.

The machines used for this research were MIMD message–passing systems. We can get much more specific about "routing steps" for such a machine. First, using the clock as a stopwatch, we can profile any segment of code (including calculations and/or communications). An implementation specific version of Fox's t_{comm}/t_{calc} ratio can be instructive. It is important to apply this ratio to the hardware as Fox defines it, but it is equally important to recognize the role of the software (algorithm). That is, for some specific implementation, we should be interested in finding some measure of how much time is spent communicating and how much time is spent computing. More specifically, a careful profile could be made of a program in the following manner.

The ratio of cumulative (i.e., over the execution of the entire program) time spent communicating to time spent computing should be considered as a first cut, especially if performance (efficiency) is weak. Algorithms such as Gauss factorization are executed in stages, within a loop of some sort. In this case, the t_{comm}/t_{calc} ratio *per iteration* is an interesting figure (and—if the loop represents most of the program's execution time—this should be approximately equal to the cumulative figure).

When possible, the analysis of communications complexities should be analyzed carefully. For instance, in the Gauss factorization code that is presented in

Appendix F, a C *structure* is used to relay the owner (node id) of a pivot and the pivot's row, column, and value. This structure is 20-bytes of data and we know the pattern with which these structures are moved about during the course of the program. It is important to *quantify* communication like this when possible. The vague notation should lose significance in the presence of such concrete information.

There are other important and related ideas. The frequency and volume of communications traffic is easy to determine with a high degree of accuracy for algorithms such as Gauss factorization. Once again, in the presence of this kind of information, we should dispense with vague concepts. It is useful to consider something like a pie chart showing the various amounts of time spent on each portion of the major loop in a program. Indeed, this was a part of the development of the Gauss code given in this thesis. Tools such as these are important in refining parallel algorithms and streamlining code.

The parallel program designer must consider many other issues regarding communications. Graph theory notation is a natural tool. A link–by–link analysis of the communications over the course of a program is not out of the question (especially if the communication is merely a repetition of very simple messages). Efficient use of the topology is important. We should consider the percentage of links used, balancing of the communications load, frequency of traffic for each link (often the communication comes in bursts and often between iterations of the basic algorithm), flow rate (in bytes per second) for each link during the bursts or over longer periods of time, timelines showing dependencies, and other specific characteristics of communications. Analysis should be done on a per–stage basis for algorithms that exhibit iteration (loops).

Perhaps most importantly, a plan for interprocessor communication should begin well in advance, before the code is ever written. A reactive approach is necessary, like debugging code. But a proactive, strong design effort can simplify matters.

81

The notion of *communicating sequential processes* (CSP) deserves attention. This model is due to C. A. R. Hoare [Ref. 30], and it is never far away in the world of transputers. There is a very close relationship between transputers, **occam** (their native language), and CSP. CSP is a useful paradigm for this sort of (message–passing) machine. When possible, a problem should be logically separated into processes. The division of the problem should be natural, so that every process represents a logical group of tasks. The processes are allowed channels to communicate, and these channels are implemented as either links in hardware or buffers in memory if, for instance, two processes on the same processor wanted to communicate.

If a problem is designed correctly, we should have substantial amounts of work within a process and minimal interprocess communication. If the processes and channels are represented as the nodes and edges of a directed graph, we can make use of some nice tools and theorems from graph theory. For instance, we should like to maximize computation and minimize communications. One natural method is to begin with atomic processes and start to build.

Suppose that we have *many* such processes (at least as many as processors) and we represent them as the nodes of a directed graph. We can assign the processes (nodes) a weight that reflects some form of *computational* difficulty. This should be a fairly concrete number, assuming that the task (process) is well–defined. It might be the number of flops per iteration, for example. Next, the channels should be clearly indicated as weighted, directed edges. The weight should usually be a very concrete number as well, like the number of bytes that passes along that channel between each stage of a computation.

This model gives the problem the sort of order that is necessary to keep the parallel design simple, logical, and formal (i.e., friendly for proof of program correctness). Once the problem has been expressed in such a manner, there are many options. For example, we could consider minimum cuts of the flow rates to

82

decide how to efficiently apportion processes to processors. This mapping alone could greatly enhance the performance of code.

It seems that much of the work in this area is rather imprecise and generally unacceptable. Granted, parallel design methodology is a relatively recent problem but it can be improved substantially. Good parallel designs that consider these kinds of issues and express them clearly will likely be in high demand as parallel computing machinery develops.

C. PARALLEL METHODS

The wide–ranging capabilities of contemporary computing machinery are evident. An exhaustive list would demand pages, but most readers could readily name several applications that bear little resemblance to each other. For a single, very specific machine there is almost no limit to the combinations of sequential instructions that it may carry out. Put another way, a particular machine can be designed and built in a few months or years depending upon the level of sophistication involved. But the different types and purposes of software that may be created to run on that single machine are nearly limitless. Consider Householder's comments on the art of computation [Ref. 17: p. 1]:

> *If a computation requires more than a very few operations, there are usually many different possible routines for achieving the same end result. Even so simple a computation as ab/c can be done (ab)/c, (a/c)b, or a(b/c), not to mention the possibility of reversing the order of the factors in the multiplication. Mathematically these are all equivalent; computationally they are not (cf. §1.2 and §1.4). Various, and sometimes conflicting, criteria must be applied in the final selection of a particular routine. If the routine must be given to someone else, or to a computing machine, it is desirable to have a routine in which the steps are easily laid out, and this is a serious and important consideration in the use of sequenced computing machines. Naturally one would like the routine to be as short as possible, to be self-checking as far as possible, to give results that are at least as accurate as may be required. And with reference to the last point, one would like the routine to be such that it is possible to assert with confidence (better yet, with certainty) and in advance that the results will be as accurate as may be desired, or if an advance*

*assessment is out of the question, as it often is, one would hope that it can be made
at least upon completion of the computation.*

<div align="right">

— ALSTON S. HOUSEHOLDER

</div>

Parallel algorithms are combinations of sequential ones, so their complexity
can grow quickly. In general, the hardware issues surrounding parallel problems
are mature and straightforward. Software, on the other hand, is developing and
generally difficult to use.

In addition to the familiar design considerations for a straightforward sequential
algorithm, the design of a parallel solution must specify:

- An awareness of the *interaction* between processing and communication. Frequency and duration (message length) of communications should be known, if possible. Additionally, we should know how this compares to the frequency and duration (flops) of computing work.

- A plan for interprocessor communication; including hardware and software.

- A scheme for memory usage.

- The *granularity* of the problem (i.e., should the processors be given larger or smaller "chunks" of work at a time).

- Load balancing among several processors.

- A method for accessing input/output resources.

This is a very high level look at the problem. The issue of communications alone,
can be more than half of the problem. The simplicity of this short list does not do
the problem justice. Correct execution, as in the sequential case, is very important.
But parallel algorithms are subject to the added scrutiny of performance data (e.g.,
efficiency).

The methodology for constructing parallel algorithms is a very creative process, and there are many questions that can be asked. Is a highly efficient parallel solution possible, or is the problem bound by dependencies and sequential work? What is the ratio of time spent communicating to time spent computing? How nearly does a given algorithm approach the optimal solution? What would happen on some other number of processors? Are there any bottlenecks that can be eliminated? Nevertheless, the current performance of parallel machines and the promise of future architectures is more than adequate motivation to continue developing these products.

D. ALGORITHMS

With the preceding concerns in mind, let us consider the algorithm for Gauss factorization that was used in this work. The algorithm is given at a very high level because detail can be gleaned from Chapter V and from the actual code in Appendix F. The first consideration for GF was "How should the work be distributed?" There are many options. The matrix could be distributed by rows, or columns, or blocks. The method chosen in this case was a distribution of the columns of A across the nodes of the machine. The columns were distributed so that column j went to processor number $j \pmod{P}$ in a P–processor network.

Such a distribution scheme seems natural for several reasons. First, the work associated with the Gauss process moves toward the lower right–hand corner of the matrix $A \in \Re^{n \times n}$. By using a modulus assignment, and assuming that $n \gg P$, we have a situation where the load on the processors is nearly balanced for most of the process. Second, a column–oriented assignment places the pivot column on a single node at each stage. This makes division by the pivot value a simple task. It is interesting to note that a similar distribution of A by *rows* would have merit as well.

Once the matrix has been distributed, the code simply moves, in a synchronized fashion, from stage to stage of Gauss. At each stage, we must pivot according to some strategy. The complete pivoting showed especially poor performance since it involved a great deal of communication and synchronization between stages. The partial pivoting method allows us to determine which node will have the pivot and much less communication is required when this node simply broadcasts the pivot and pivot column. After the pivot node divides every element under the pivot by the pivot value, it broadcasts the entire pivot column to every other processor. When the processors obtain the pivot column, they use the multipliers to perform arithmetic in the Gauss transform area, and then proceed to the next stage.

The following algorithms give an overview of the programs that appear in Appendix F.

Algorithm 4.1 (Parallel GF: Host) *At this level, the host code is essentially the same for both partial pivoting and complete pivoting. The program is very simple: distribute the columns, and then accept them back one-by-one. Let $A \in \Re^{m \times n}$ be the matrix of coefficients, and let P be the number of processors. This algorithm forms the modified copy of A by overwriting the original copy. After the nth column is returned from the nodes, we have the factored version of A that can be separated into L and R in the usual manner.*

begin GF (Host)

    for $j = 0 : (n - 1)$

        send $A(:, j)$ to node $(j \bmod P)$

    end for

    for $r = 0 : (n - 1)$

        receive $A(:, r)$ from node $(r \bmod P)$

    end for

end GF (Host)

Algorithm 4.2 (Parallel GFPP: Nodes) *Let $A \in \Re^{m \times n}$ be the entire matrix (held at the host). This algorithm is executed on each node in a P-processor network. Let the node number be N and let $A_N \in \Re^{m_N \times n}$ be the local copy of select columns of the matrix A (where $m_N \approx m/P$ is the number of columns held locally). Let G_N be that part of the Gauss transform area, G, that is held locally. This node receives every column, j, of A where $(j \bmod P) = N$.*

begin GFPP (Nodes)

    for $j = 0 : (m_N - 1)$

        receive column and place in $A_N(:, j)$

    end for

    for $r = 0 : (n - 1)$

        if $(r \bmod P) = N$ (pivot is held locally)

            perform partial pivoting

            broadcast pivot row index, s, to all nodes

            perform pivot column arithmetic

            broadcast pivot column to all nodes

        else

            receive pivot row index, s, and perform row interchanges

            receive broadcast of pivot column

        end if

        if $N = 0$

            send pivot column to host

        end if

        perform arithmetic in G_N

    end for

end GFPP (Nodes)

Algorithm 4.3 (Parallel GFPC: Nodes) *Let $A \in \Re^{m \times n}$ be the entire matrix (held at the host). This algorithm is executed on each node in a P-processor network. Let the node number be N and let $A_N \in \Re^{m_N \times n}$ be the local copy of select columns of the matrix A (where $m_N \approx m/P$ is the number of columns held locally). Let G_N be that part of the Gauss transform area, G, that is held locally. This node receives every column, j, of A where $(j \bmod P) = N$.*

begin GFPC (Nodes)

    for $j = 0 : (m_N - 1)$

        receive column and place in $A_N(:, j)$

    end for

    for $r = 0 : (n - 1)$

        locate best (local) pivot candidate

        elect pivot (let node N_P hold the winner of the pivot election)

        if $(N_P = N)$

            broadcast pivot indexes, (s, t), to all nodes

            perform pivot column arithmetic

            broadcast pivot column to all nodes

        else

            receive pivot indexes, (s, t)

            perform permutations

            receive broadcast of pivot column

        end if

        if $N = 0$

            send pivot column to host

        end if

        perform arithmetic in G_N

    end for

end GFPC (Nodes)

V. IMPLEMENTATION

A. ENVIRONMENT

Chapter IV introduces parallel algorithms for Gauss factorization (GF). The GF algorithms are produced for partial and complete pivoting strategies. All of the programs associated with this research are written in parallel versions of the C language and executed on two types of machines at the U. S. Naval Postgraduate School. The Math Department's iPSC/2 afforded eight of Intel's CX type processors arranged in a hypercube topology. The Parallel Command and Decision Systems (PARCDS) Laboratory in the Computer Science Department has more than seventy transputers available for the experiments. The discussion below gives a more exact description of the material and equipment used in the work.

1. Hardware

This section describes the machines upon which the work was carried out. A general knowledge is assumed, including familiarity with the Intel 80386 microprocessor, 80387 math coprocessor, and INMOS transputers. Some of this information is provided in Appendix B.

The hardware used in this research represents the state–of–the–art for the mid–to–late 1980s. These machines are quickly becoming outdated—fitting the history of computing—but both INMOS and Intel have more recent, competitive products in today's market and fine prospects for future machines. So, while they are a bit dated, the products used in this research represent important contemporary parallel architectures.
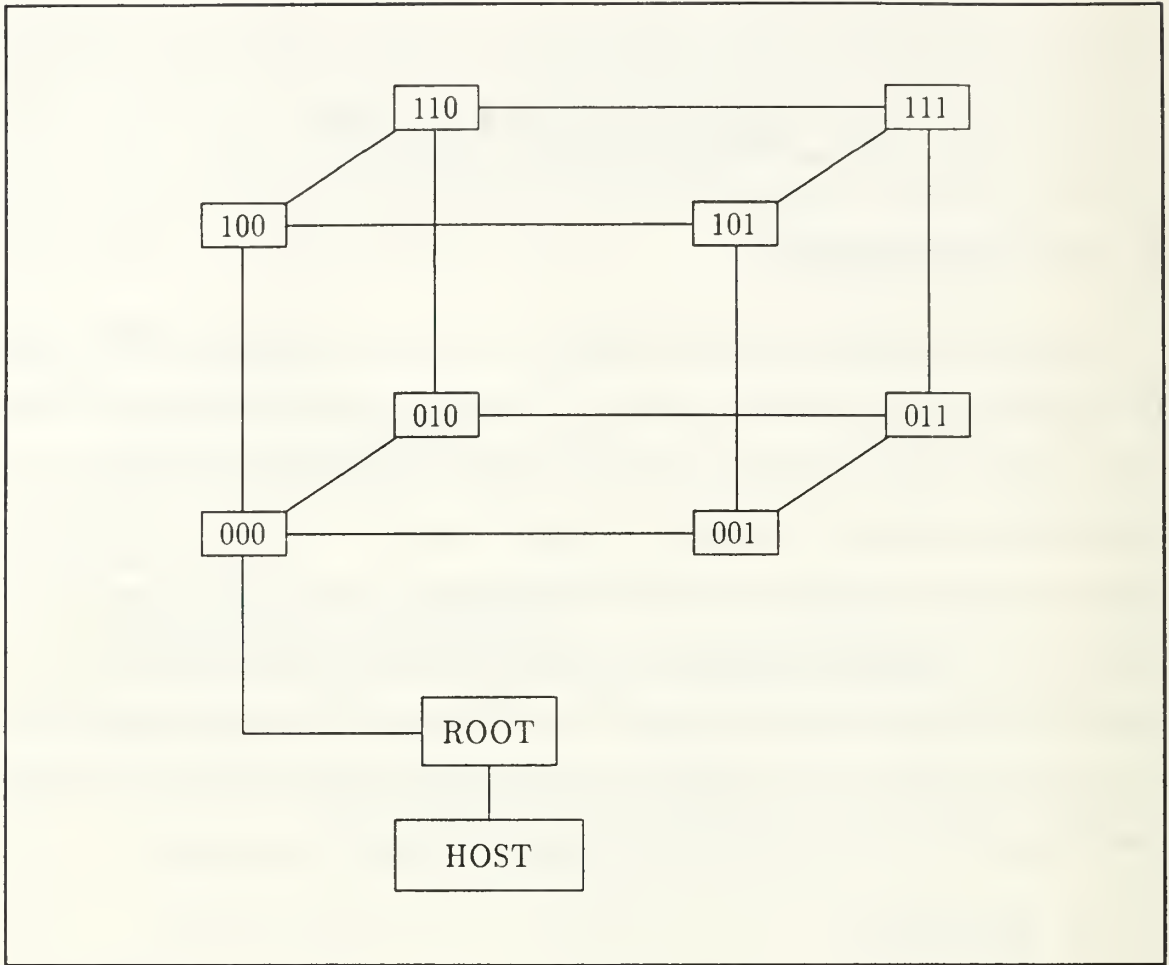
Figure 5.1: Hypercube Interconnection Topology: Order $n \leq 3$

a. Networks of Transputers

The majority of the research was performed upon hypercubes of order $n \in \{0, 1, 2, 3\}$. These are the usual hypercubes (see Appendix C) and each is imbedded in the 3–cube. Figure 5.1 shows this topology. Some of the transputer work for this thesis was performed by a network of sixteen IMS T800–20 transputers connected in nearly hypercube fashion (Figure 5.2). This is not identical to the 4–cube, so it will be called the *hybrid* cube (it is used as a root with two subtrees that happen to be 3–cubes). The subtrees of the hybrid cube can be distinguished by the first bit. One of the 3-cubes has labels like $0xxx$; the other is labeled $1xxx$.
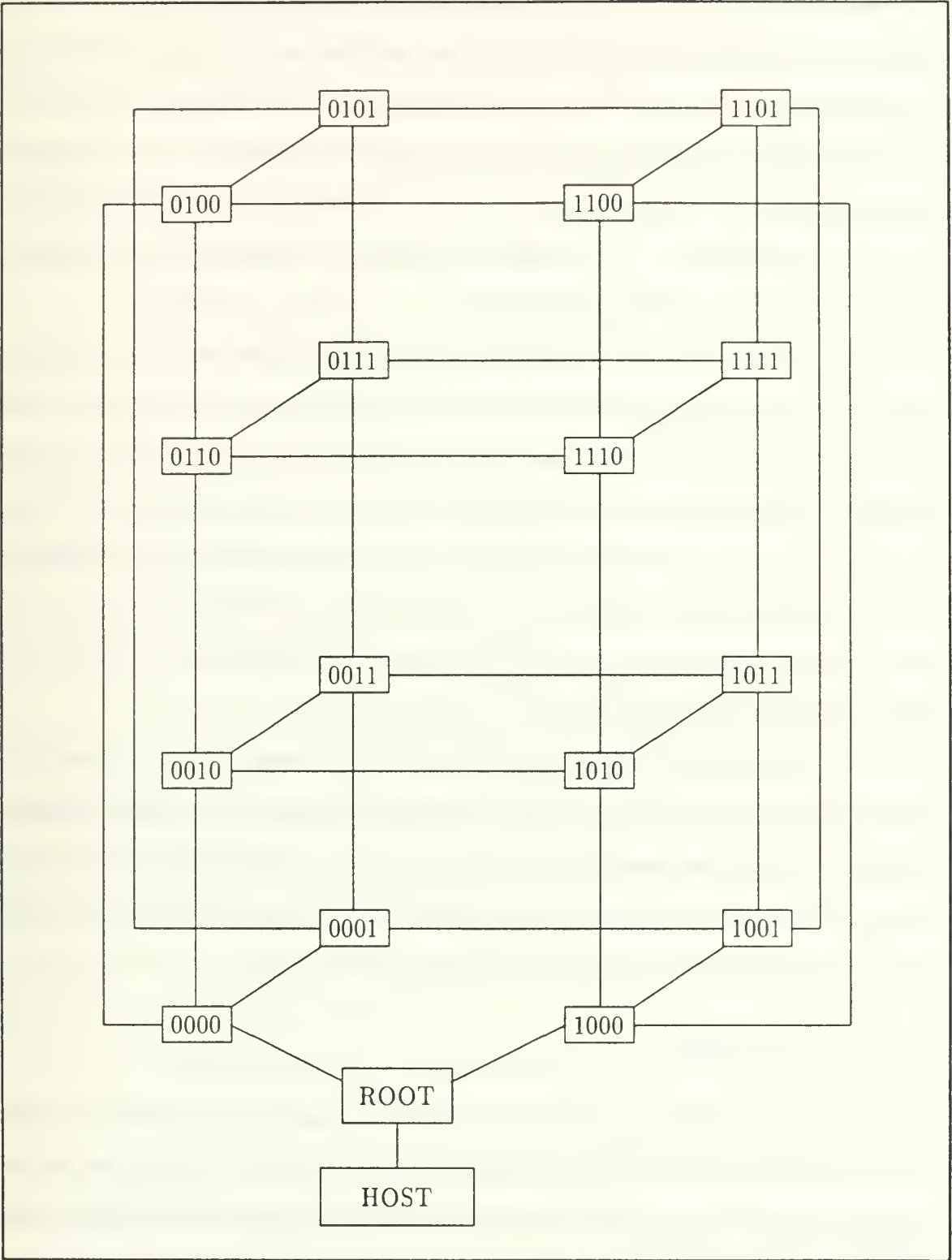
Figure 5.2: Hybrid Hypercube Interconnection Topology

The rationale behind building the hybrid cube is purely *practical*. The transputers have only four links. Assuming that we define nodes of the hypercube to be a single transputer, a pure hypercube of order four would be a closed interconnection scheme with no opportunity for input or output to or from the system. Here, the root node has been inserted between nodes zero (0000) and eight (1000). While this deals a horrible blow to the elegance of hypercube algorithms—particularly communications—it can be used effectively.

The hardware for the hybrid hypercube is configured with code by Mike Esposito [Ref. 31]. This gives us sort of an unlabeled version of the structure that appears in Figure 5.2. To make use of this configuration, the nodes must be labeled in a logical fashion. The Gray code (Appendix C) is a reasonable choice for labeling the nodes. The actual labeling is accomplished by a Network Information File (NIF) when the transputers are loaded by the Logical Systems C Network Loader, LD–NET. A more detailed description of this process is contained in the file named hyprcube.nif in Appendix F.

Networks of transputers use point–to–point communications across bidirectional *links*. The links for this work operate at 20 megabits per second (bidirectionally). That is, ten megabits per second is a peak unidirectional transmission rate. Current transputer implementations employ a store–and–forward approach to message passing (see Appendix B) for multi–hop transmissions.

b. Intel iPSC/2

The iPSC/2 used for this research contained eight processors of the "CX" type (80386/80387 combination). The host is an 80386–based IBM–compatible personal computer running AT&T UNIX System V (version 3.2). The nodes run a local subset of UNIX called NX. The host is capable of supporting many users at once, but each node only supports a single–user.

Users can request p nodes, where $p = 2^n$ for $n \in \{0, 1, 2, 3\}$. If another user does not already have the requested portion of the cube, the request is granted. As long as nodes remain, another user can access them. For instance, one user could be working on two nodes and—at the same time—another user could access up to four others. While the first two users still possessed these six nodes, a third user could get one or both of the remaining two nodes.

Unlike the transputers, Intel uses a direct–connect circuit switching (see Appendix B) approach to multi–hop communications. There is an overhead associated with setting up the path for communication, but this cost is nearly the same regardless of how many hops the message cross. Once the circuit is established, the message can proceed directly from the origin to the destination with negligible interference from intermediate nodes.

c. *Host and Root*

The notion of host is similar on both machines, but there is a slight difference. The Intel hypercube is directly connected to the host. The transputer network, however, uses a substantially different protocol than the typical personal computer. Transputers employ point–to–point serial communications, using an 11–bit link protocol with byte–by–byte acknowledgment. The acknowledge is a two–bit packet with dual meaning. The receiving transputer has begun to receive the byte and it has storage space for another.

In the transputer case, *host* means the PC. We use the term *root* transputer to identify the transputer within the host PC that acts something like a host to the attached network of transputers. Figure 5.1 illustrates this configuration. An IMS B004 extension board in the host PC holds a T414 root transputer. The B004 is plugged into the PC's bus and a parallel-serial converter lies between the PC and the T414. In Figure 5.1 the "host" is a PC and the "root" transputer is the T414.

93

The iPSC/2 host is simplified, and could almost be thought of as a combination of the host and root for the transputer case. Since the entire thesis uses the same programs for both machines, the *root* and *host* terminology can become confusing. As it is not always convenient to express this difference in painstaking detail, I will use the terms somewhat loosely. An understanding of the differences between the machines should serve to eliminate confusion in every case. When only one of the terms (*host* or *root*) is needed, I have used the correct term. When both of the terms apply, I have used them almost interchangeably and they should be interpreted according to the machine under consideration.

2. Software

The software for this research was written in the C language. The Logical Systems C product (version 89.1 of 15 January 1990) was used for the transputer implementation. For the iPSC/2 work, the C compiler supplied by Intel was used.

B. COMMUNICATIONS FUNCTIONS

Prior to implementing the Gauss algorithms, a substantial communications package was constructed. Most of the code for communications appears in the files **comm.h** and **comm.c** (see Appendix F). As expected, the header file provides definitions for manifest constants and specifications (declarations) for the functions. An overview of the functions provided in this file is is useful before we discuss the Gauss code that called these functions.

The **cubecast()** function supports broadcasts from the host to all the nodes of a hypercube. Given a hypercube of order $n \in \{0, 1, 2, 3\}$ with $p = 2^n$ processors, this communication is completed in n, or $\log_2(p)$, stages. This has some utility in a 3–cube, but imagine the impact in a 10–cube. All 1,024 processors in the hypercube would have the message after 10 stages of communication. This function

is especially useful at the beginning of a problem, when data must be shipped to each of the workers in the network.

Often we need to gather information in the reverse direction, from the workers back to the root. The **coalesce()** function is one way to accomplish this task. If no modification was necessary at intermediate nodes, this operation could be completed without interference. In the algorithms that I used, however, there was occasion to modify the information along the way back to the root. For this reason, the gathering is accomplished using two function calls. First, information is coalesced to a given node. Upon return from **coalesce()**, the data exists locally and may be operated upon. When the data is ready for submission, the **submit()** function is used to pass it one step closer to the root.

A modification of the **cubecast()** function that was useful for the Gauss problem was **cubecast_from()**. This function does not assume that the host is the originator of the broadcast. Instead, the source is specified as the first argument to this function. The function still performs the broadcast in $\log_2(p)$ stages, but it uses the concept of a *direction* to accomplish this.

The concept of directions in the hypercube turns out to be a fairly useful one. For concreteness, consider the 3–cube shown in Figure C.2. Starting at any given node, we can specify a *direction* using one of the three combinations $d \in \{001, 010, 100\}$. Suppose that the node's label is ℓ and let $\oplus$ denote the exclusive OR operation. Then for some direction, d, the number $(\ell \oplus d)$ is the label of the node in the direction d from the node ℓ.

This concept can be applied in general in a hypercube of order n using n-bit labels for the nodes and some direction d. The possible directions are all the n combinations of $(n-1)$ zeros and a single one in an n-bit number. Accordingly, the code uses directions $d \in \{1, 2, 4, \ldots 2^{n-1}\}$. In most cases, when a direction–by–direction approach is desired for all possible directions, we start with one and use

the C left shift operator ($<<$) to produce the other directions incrementally.

These functions and several others are described in detail in the code of Appendix F, but these basic ideas give us a reasonably good introduction at a level that is adequate for understanding the algorithms.

C. CODE DESCRIPTIONS

A detailed description of the source code used to implement the algorithms of Chapter IV is given in the header file **gf.h**. This header file, located in Appendix F, is used by both the partial pivoting and complete pivoting codes. The code for GF with partial pivoting can be found in **gfpphost.c**, the host program, and **gfppnode.c**, the node program. The code for the complete pivoting algorithm is similar except for the election of pivots, so most of it has been omitted in the interest of saving space. Only the **elect_next_pivot**() function remains because it is the significant difference between the partial and complete pivoting codes. This function appears in **gfpcnode.c**.

VI. RESULTS

A. GAUSS WITH COMPLETE PIVOTING

The host code, **gfpchost.c**, and the node program, **gfpcnode.c**, are written to provide a parallel implementation of Gauss Factorization with complete pivoting. Since the columns of A are distributed among the nodes of the multiprocessor system, the selection of each pivot requires communication. The selection process, in this case, begins with each node selecting its own best *candidate* for pivot. Once each of the nodes has made this choice, an *election* is held to select the best candidate among all of the nodes.

Implementation details for the election process are described in the source code, so a detailed description is not given here. Nevertheless, these results show how communication—like the election process—can withstand efficient parallel programming. This program shows how parallel performance can suffer from the effects of communications. (Recall Fox's t_{comm}/t_{calc} and Seitz's three components of overhead from Chapter IV).

The complete pivoting strategy inserts inefficient communications between each stage of the process. The communications themselves are bound to be inefficient since the election process finds all nodes of an n–cube participating in an n–stage exchange of a 20-byte structure (pivot candidates). In addition to the use of small messages, the election imposes an added measure of synchronization upon the problem. This allows the processors less independence and forces them to transition between "useful" program execution and communication more frequently. This transition can become burdensome and the processor can eventually find little time to perform calculations.

In addition to the election process, there is a one-to-all broadcast from the node holding the pivot to inform the others of the pivot column values. With an $m \times m$ matrix A, this message is essentially a column of m double precision floating-point values. Doubles for this implementation were eight bytes each, so this is a unidirectional broadcast of $8m$ bytes with exponential fanout.

The election process—as simple as it appears—will prove to be an obstacle that opposes efficiency. Both the iPSC/2 and transputer systems reward, in terms of transmission rates, the sender of *long messages*. Short messages are essentially penalized by the overhead involved in setting up the transmission line and manager. Let us consider the results of this complete pivoting strategy. The results from the iPSC/2 appear first followed by the transputer results. The largest dimension, n, that is recorded is $n = 176$. The iPSC/2 machine would handle larger problems, but this seemed pointless since the performance appears to approach maximum efficiency early.

1. Data for the iPSC/2 System

Table 6.1 shows the timing data for execution of Gauss Factorization with complete pivoting on the Intel iPSC/2 system.

TABLE 6.1: EXECUTION TIMES FOR GF(PC) ON THE iPSC/2

Dimension	Time (seconds) on a Hypercube of Order			
(n)	0	1	2	3
8	0.126	0.097	0.092	0.155
16	0.716	0.674	0.608	0.744
24	2.208	1.751	1.616	1.568
32	4.627	3.705	3.239	3.149
40	9.246	6.888	5.895	5.250
48	14.888	11.479	9.770	9.109
56	23.686	17.883	15.206	13.796
64	36.123	26.424	22.326	19.957
72	49.227	38.178	31.421	28.460
80	70.546	50.754	42.087	37.810
88	89.210	69.257	56.803	51.148
96	115.473	86.760	72.346	63.954
104	150.915	110.247	91.966	82.680
112	182.475	138.880	114.486	102.266
120	224.458	168.056	139.587	123.683
128	282.491	206.222	170.650	153.379
136	339.076	248.422	208.745	186.205
144	385.623	295.217	241.564	217.099
152	468.763	345.049	281.972	254.538
160	527.953	404.235	331.653	292.352
168	636.004	457.089	381.597	338.464
176	723.596	532.597	449.745	395.008

TABLE 6.2: SPEEDUPS FOR GF(PC) ON THE iPSC/2

Dimension	Speedup on a Hypercube of Order		
(n)	1	2	3
8	1.299	1.373	0.813
16	1.063	1.178	0.962
24	1.261	1.367	1.408
32	1.249	1.429	1.470
40	1.342	1.569	1.761
48	1.297	1.524	1.635
56	1.324	1.558	1.717
64	1.367	1.618	1.810
72	1.289	1.567	1.730
80	1.390	1.676	1.866
88	1.288	1.571	1.744
96	1.331	1.596	1.806
104	1.369	1.641	1.825
112	1.314	1.594	1.784
120	1.336	1.608	1.815
128	1.370	1.655	1.842
136	1.365	1.624	1.821
144	1.306	1.596	1.776
152	1.359	1.662	1.842
160	1.306	1.592	1.806
168	1.391	1.667	1.879
176	1.359	1.609	1.832

The speedup data that is shown in Table 6.2 is derived from these execution times. Speedup was calculated using the usual formula (see Appendix A for details)

$$S_p = \frac{T_1}{T_p}$$

for speedup on p processors.

TABLE 6.3: EFFICIENCIES FOR GF(PC) ON THE iPSC/2

Dimension	Efficiency (percent) on a Hypercube of Order		
(n)	1	2	3
8	64.948	34.332	10.161
16	53.155	29.441	12.024
24	63.068	34.169	17.603
32	62.451	35.716	18.370
40	67.122	39.215	22.015
48	64.852	38.098	20.431
56	66.225	38.943	21.462
64	68.354	40.450	22.625
72	64.470	39.168	21.621
80	69.498	41.905	23.323
88	64.405	39.263	21.802
96	66.548	39.903	22.570
104	68.444	41.025	22.816
112	65.695	39.847	22.304
120	66.781	40.200	22.685
128	68.492	41.385	23.022
136	68.246	40.609	22.762
144	65.312	39.909	22.203
152	67.927	41.561	23.020
160	65.303	39.797	22.574
168	69.571	41.667	23.489
176	67.931	40.223	22.898

Given the execution times and speedups presented in Tables 6.1 and 6.2, and using the formula

$$E_p = \frac{S_p}{p}$$

(as defined in Appendix A), we can determine the efficiency of p processors applied to the Gauss problem. This efficiency data is shown in Table 6.3.
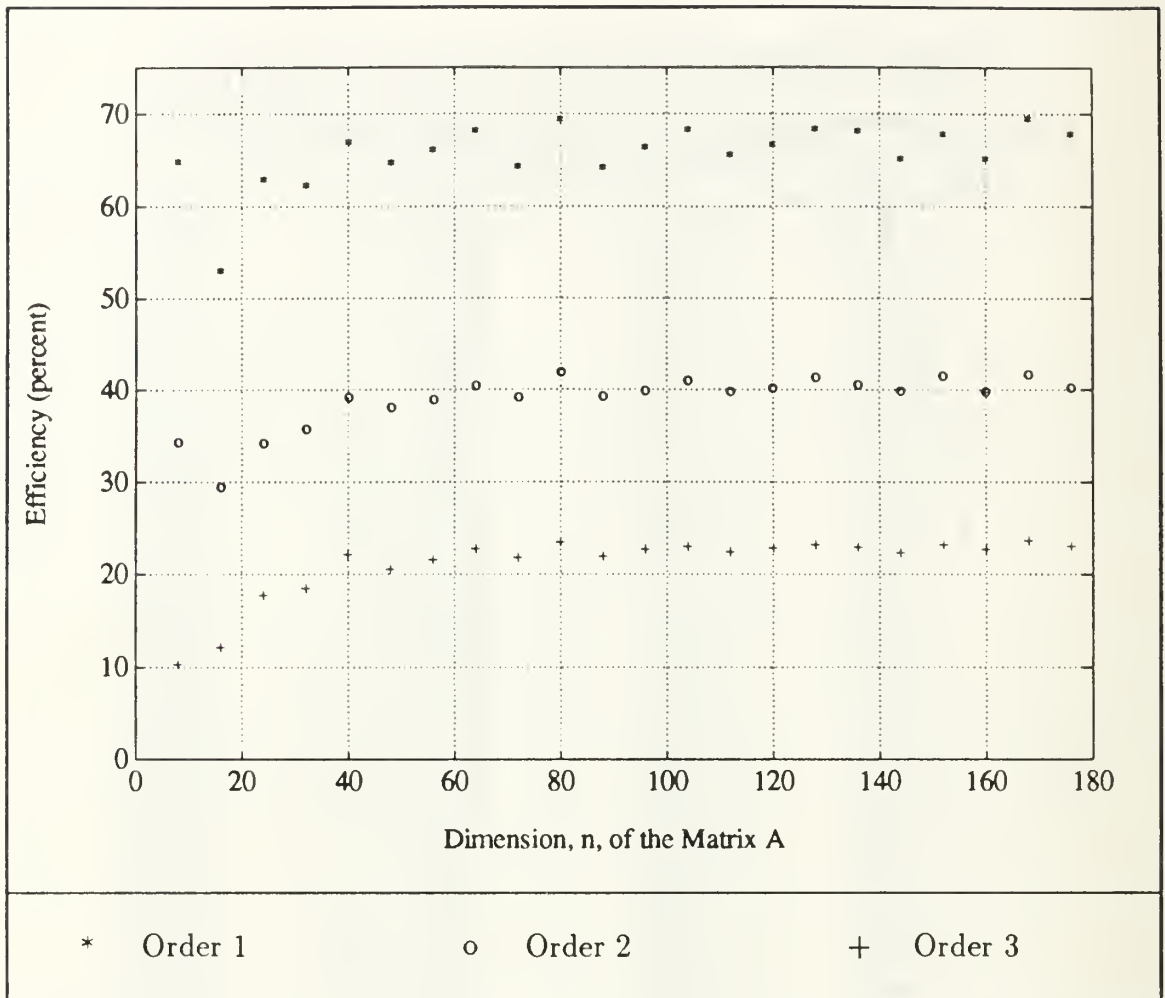
Figure 6.1: Efficiencies for GF (PC) on the iPSC/2

Many different graphical displays of this data would be interesting, but the efficiency data may be the most interesting since it sort of captures the success or failure of a parallel program (i.e., poor efficiencies should lead us to question the parallel nature of the algorithm). Figure 6.1 shows a scatterplot of the data from Table 6.3.

102

TABLE 6.4: EXECUTION TIMES FOR GF(PC) ON THE TRANSPUTERS

Dimension	Time (seconds) on a Hypercube of Order				
(n)	0	1	2	3	4
8	0.0083	0.0075	0.0077	0.0088	0.0925
16	0.0481	0.0392	0.0373	0.0372	0.1236
24	0.1494	0.1173	0.1063	0.1001	0.1855
32	0.3417	0.2580	0.2220	0.2132	0.2947
40	0.6538	0.4922	0.4135	0.3798	0.4587
48	1.1158	0.8202	0.6934	0.6397	0.7041
56		1.2950	1.0716	0.9696	1.0239
64		1.8940	1.5688	1.4046	1.4407
72			2.2116	1.9817	1.9808
80			2.9560	2.6529	2.6248
88			3.9127	3.4812	3.4090
96				4.4808	4.3812
104				5.6442	5.4519
112				7.0388	6.7087
120				8.5430	8.1252
128				10.3300	9.7532
136					11.6930
144					13.6538
152					16.1029
160					18.5476
168					21.4437
176					24.4684
n_{max}	48	67	92	128	176

2. Data for the Transputer System

Using the same methods, the timing (Table 6.4), speedup (Table 6.5), and efficiency (Table 6.6) data for the transputer system is determined. Unfortunately, the memory limitations of the transputers used for this work prevented comparisons for large problem size. Empty portions of Table 6.4 signify inavailability of data (i.e., execution failure due to inappropriate or excessive problem size). The maximum problem size that executed successfully for each configuration is listed on the last line of the Table. Figure 6.2 shows a scatterplot of the data from Table 6.6.

103

TABLE 6.5: SPEEDUPS FOR GF(PC) ON THE TRANSPUTERS

Dimension	Speedup on a Hypercube of Order			
(n)	1	2	3	4
8	1.111	1.074	0.942	0.090
16	1.227	1.288	1.290	0.389
24	1.274	1.405	1.493	0.805
32	1.324	1.539	1.602	1.159
40	1.328	1.581	1.721	1.425
48	1.360	1.609	1.744	1.585
56	1.363	1.648	1.821	1.724
64	1.389	1.677	1.872	1.826
72		1.691	1.887	1.888
80		1.734	1.932	1.953
88		1.743	1.959	2.001
96			1.975	2.020
104			1.993	2.064
112			1.996	2.094
120			2.022	2.126
128			2.030	2.150
136				2.150
144				2.186
152				2.180
160				2.207
168				2.210
176				2.227

TABLE 6.6: EFFICIENCIES FOR GF(PC) ON THE TRANSPUTERS

Dimension (n)	Efficiency (percent) on a Hypercube of Order			
	1	2	3	4
8	55.556	26.860	11.775	1.125
16	61.356	32.204	16.130	2.431
24	63.693	35.133	18.662	5.034
32	66.224	38.477	20.029	7.246
40	66.409	39.526	21.514	8.908
48	68.017	40.230	21.803	9.905
56	68.167	41.190	22.760	10.776
64	69.431	41.913	23.406	11.410
72		42.279	23.592	11.801
80		43.358	24.155	12.207
88		43.575	24.488	12.504
96			24.691	12.626
104			24.916	12.897
112			24.948	13.088
120			25.279	13.289
128			25.369	13.435
136				13.440
144				13.662
152				13.623
160				13.795
168				13.812
176				13.917
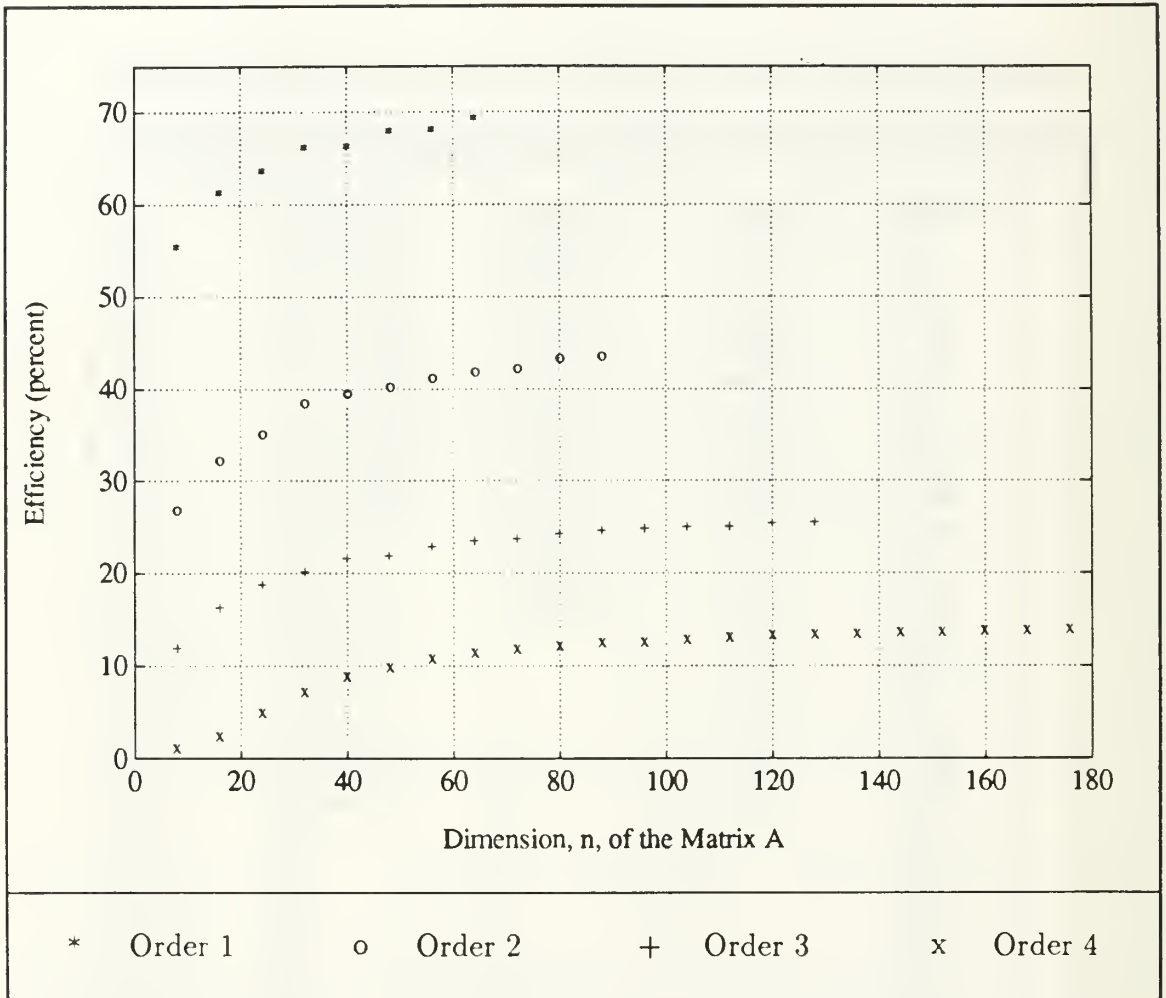
Figure 6.2: Efficiencies for GF (PC) on Transputers

B. GAUSS WITH PARTIAL PIVOTING

1. Data for the iPSC/2 System

Table 6.7 shows the timing data for execution of the Gauss Factorization (partial pivoting) codes (**gfpphost.c** and **gfppnode.c**) on the Intel iPSC/2 system. The speedup data that is shown in Table 6.8 is derived from these execution times. Speedup was calculated using the usual formula (see Appendix A for details)

$$S_p = \frac{T_1}{T_p}$$

for speedup on p processors. Given the execution times and speedups presented in Tables 6.7 and 6.8, and using the formula

$$E_p = \frac{S_p}{p}$$

(as defined in Appendix A), we can determine the effectiveness (efficiency) of p processors applied to the Gauss problem. This efficiency data is shown in Table 6.9.

TABLE 6.7: EXECUTION TIMES FOR GF(PP) ON THE iPSC/2

Dimension	Time (seconds) on a Hypercube of Order			
(n)	0	1	2	3
8	0.109	0.130	0.127	0.155
16	0.371	0.359	0.394	0.493
24	0.508	0.489	0.519	0.624
32	0.752	0.673	0.675	0.782
40	1.055	0.880	0.834	0.911
48	1.499	1.144	1.024	1.067
56	2.019	1.473	1.248	1.228
64	2.733	1.878	1.491	1.402
72	3.646	2.412	1.872	1.721
80	4.743	3.040	2.256	1.989
88	6.053	3.719	2.644	2.237
96	7.567	4.547	3.125	2.560
104	9.431	5.477	3.698	2.912
112	11.468	6.561	4.252	3.237
120	13.847	7.859	4.933	3.646
128	16.552	9.211	5.661	4.070
136	19.619	10.873	6.590	4.633
144	23.071	12.632	7.532	5.170
152	26.982	14.681	8.940	5.866
160	31.204	16.869	9.866	6.539
168	35.865	19.318	11.143	7.284
176	41.064	21.990	12.605	8.084
200	59.453	31.437	17.598	10.910
225	83.962	44.076	24.329	14.701
250	114.319	59.515	32.410	19.118
275	151.443	78.652	42.336	24.512
300	195.822	102.589	54.138	30.927
325	248.153	127.840	68.082	38.418
350	309.241	158.859	84.072	46.978
375	379.538	194.599	101.984	56.280
400	459.740	235.259	122.946	67.366
425	550.536	281.312	147.058	80.439
450	653.070	333.180	173.748	94.656
475	767.616	391.136	203.513	110.243
500	894.705	455.308	236.483	127.631

TABLE 6.8: SPEEDUPS FOR GF(PP) ON THE iPSC/2

Dimension	Speedup on a Hypercube of Order		
(n)	1	2	3
8	0.842	0.860	0.704
16	1.035	0.941	0.753
24	1.039	0.979	0.814
32	1.118	1.114	0.961
40	1.199	1.265	1.158
48	1.311	1.465	1.405
56	1.371	1.618	1.645
64	1.455	1.833	1.949
72	1.512	1.948	2.119
80	1.560	2.102	2.384
88	1.628	2.289	2.706
96	1.664	2.422	2.956
104	1.722	2.550	3.239
112	1.748	2.697	3.543
120	1.762	2.807	3.798
128	1.797	2.924	4.067
136	1.804	2.977	4.235
144	1.826	3.063	4.462
152	1.838	3.018	4.600
160	1.850	3.163	4.772
168	1.857	3.219	4.924
176	1.867	3.258	5.080
200	1.891	3.378	5.449
225	1.905	3.451	5.711
250	1.921	3.527	5.980
275	1.925	3.577	6.178
300	1.909	3.617	6.332
325	1.941	3.645	6.459
350	1.947	3.678	6.583
375	1.950	3.722	6.744
400	1.954	3.739	6.825
425	1.957	3.744	6.844
450	1.960	3.759	6.899
475	1.963	3.772	6.963
500	1.965	3.783	7.010

TABLE 6.9: EFFICIENCIES FOR GF(PP) ON THE iPSC/2

Dimension	Efficiency (percent) on a Hypercube of Order		
(n)	1	2	3
8	42.085	21.499	8.803
16	51.743	23.526	9.416
24	51.943	24.470	10.174
32	55.911	27.842	12.019
40	59.943	31.615	14.472
48	65.544	36.615	17.563
56	68.557	40.453	20.560
64	72.764	45.825	24.365
72	75.580	48.698	26.482
80	78.023	52.554	29.804
88	81.390	57.228	33.821
96	83.218	60.541	36.955
104	86.104	63.762	40.482
112	87.402	67.427	44.287
120	88.096	70.175	47.475
128	89.849	73.097	50.832
136	90.219	74.430	52.934
144	91.323	76.577	55.781
152	91.897	75.451	57.497
160	92.492	79.072	59.651
168	92.830	80.469	61.544
176	93.372	81.442	63.498
200	94.559	84.462	68.115
225	95.247	86.278	71.393
250	96.042	88.181	74.744
275	96.274	89.430	77.230
300	95.440	90.427	79.147
325	97.056	91.123	80.742
350	97.332	91.958	82.283
375	97.518	93.039	84.297
400	97.709	93.484	85.307
425	97.851	93.591	85.552
450	98.006	93.968	86.243
475	98.127	94.296	87.037
500	98.253	94.584	87.626
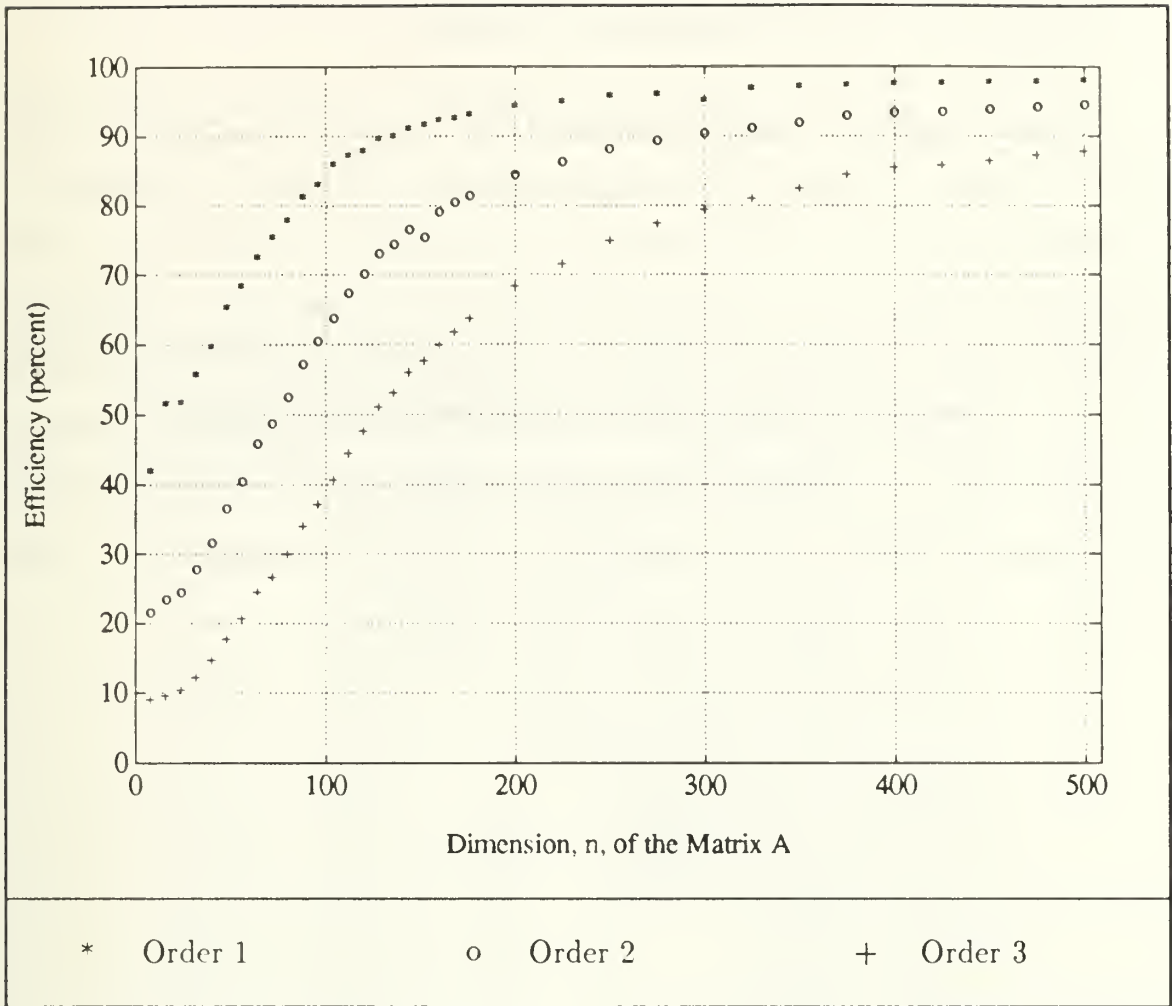
Figure 6.3: Efficiencies for GF (PP) on the iPSC/2

Here, again, only the efficiency is plotted. Figure 6.3 shows a scatterplot of the data from Table 6.9.

2. Data for the Transputer System

Using the same methods; the timing (Table 6.10), speedup (Table 6.11), and efficiency (Table 6.12) data for the transputer system is determined. Unfortunately, the memory limitations of the transputers (32 kilobytes per node) used for this work prevented comparisons for large (interesting) problem size. Empty portions of Table 6.10 signify inavailability of data (i.e., execution failure due to inappropriate or excessive problem size). The maximum problem size that executed successfully for each configuration is listed on the last line of Table 6.10. The minimum problem size for the hybrid cube on 16 processors was one where the dimension of A was $n = 16$.

TABLE 6.10: EXECUTION TIMES FOR GF(PP) ON THE TRANSPUTERS

Dimension	Time (seconds) on a Hypercube of Order				
(n)	0	1	2	3	4
8	0.0906	0.0904	0.0906	0.0909	
16	0.1126	0.1101	0.1102	0.1107	0.1092
24	0.1582	0.1480	0.1462	0.1461	0.1439
32	0.2312	0.2038	0.1965	0.1952	0.1889
40	0.3360	0.2765	0.2568	0.2520	0.2446
48		0.3782	0.3402	0.3297	0.3149
56		0.5124	0.4463	0.4258	0.4064
64		0.6911	0.5863	0.5505	0.5196
72			0.7277	0.6715	0.6308
80			0.8976	0.8147	0.7560
88			1.0675	0.9482	0.8732
96				1.1584	1.0581
104				1.3657	1.2430
112				1.6129	1.4551
120				1.8388	1.6490
128					1.8585
136					2.1306
144					2.3606
152					2.6717
160					2.9846
168					3.2910
176					3.6606
n_{max}	47	66	92	127	176

TABLE 6.11: SPEEDUPS FOR GF(PP) ON THE TRANSPUTERS

Dimension	Speedup on a Hypercube of Order			
(n)	1	2	3	4
8	1.002	1.000	0.997	
16	1.023	1.022	1.017	1.031
24	1.069	1.082	1.083	1.099
32	1.134	1.177	1.184	1.224
40	1.215	1.308	1.333	1.374
48	1.302	1.447	1.493	1.563
56	1.387	1.592	1.669	1.748
64	1.448	1.707	1.818	1.926
72		1.888	2.046	2.178
80		2.049	2.258	2.433
88		2.256	2.539	2.758
96			2.667	2.920
104			2.853	3.134
112			2.998	3.323
120			3.219	3.590
128				3.852
136				4.019
144				4.296
152				4.456
160				4.646
168				4.871
176				5.031

TABLE 6.12: EFFICIENCIES FOR GF(PP) ON THE TRANSPUTERS

Dimension	Efficiency (percent) on a Hypercube of Order			
(n)	1	2	3	4
8	50.111	25.000	12.459	
16	51.135	25.544	12.715	6.445
24	53.446	27.052	13.535	6.871
32	56.722	29.415	14.805	7.650
40	60.759	32.710	16.667	8.585
48	65.090	36.180	18.666	9.772
56	69.334	39.801	20.859	10.927
64	72.412	42.678	22.727	12.039
72		47.193	25.571	13.611
80		51.228	28.220	15.206
88		56.392	31.744	17.235
96			33.343	18.252
104			35.657	19.589
112			37.475	20.770
120			40.241	22.436
128				24.073
136				25.116
144				26.849
152				27.850
160				29.036
168				30.447
176				31.441
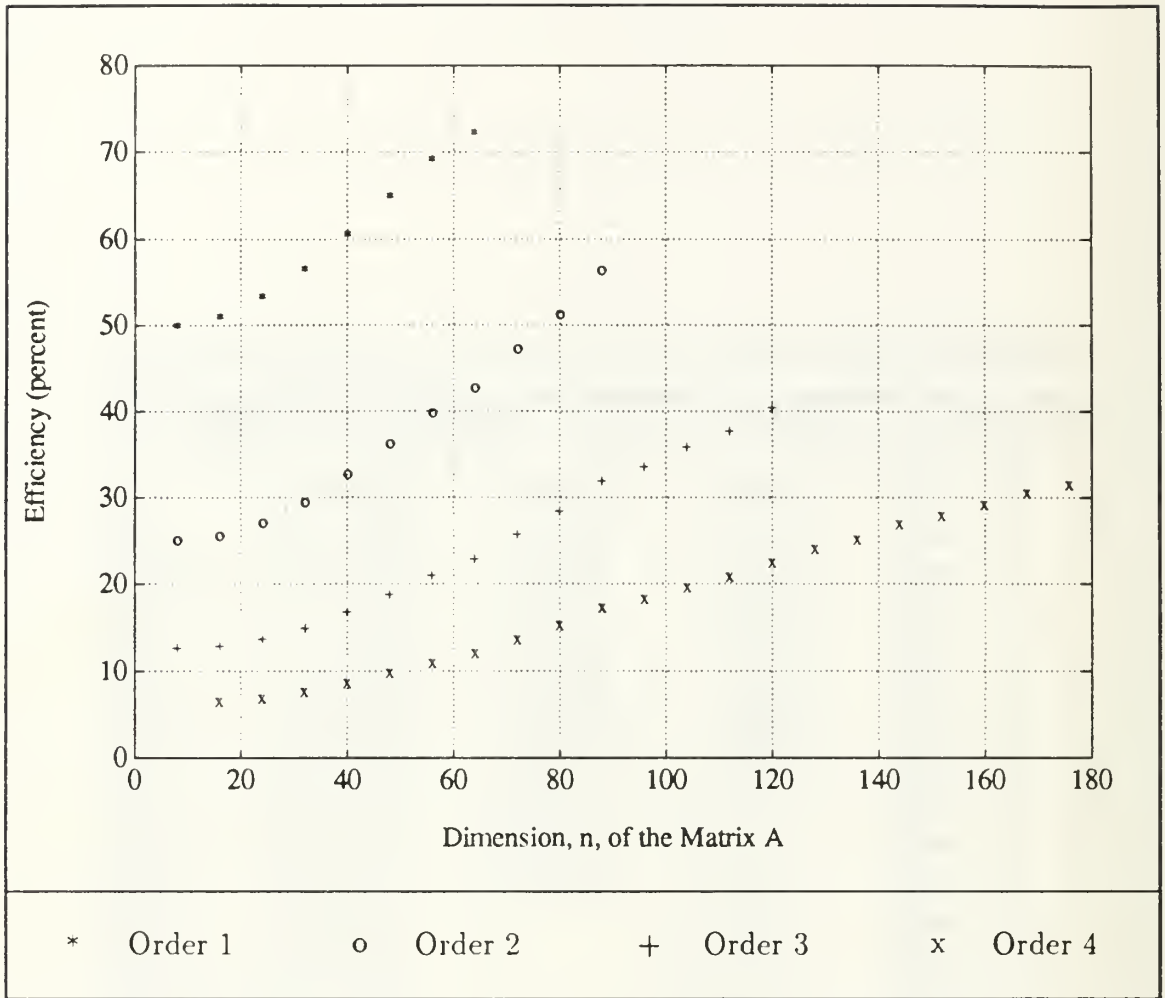
Figure 6.4: Efficiencies for GF (PP) on Transputers

Figure 6.4 shows a scatterplot of the data from Table 6.12.

VII. CONCLUSIONS

I value the discovery of a single even insignificant truth more highly than all the argumentation on the highest questions which fails to reach a truth.

— *GALILEO (1564–1642)*

A. SIGNIFICANCE OF THE RESULTS

1. Communications and Computation

Perhaps one of the most obvious effects that can be noticed in the results of Chapter VI is the abysmal performance of the complete pivoting code when compared to the partial pivoting implementation. The relatively small amount of extra communications required for the complete pivoting algorithm seems to force synchronization delays, thus reducing the system's performance. This demonstrates the criticality of balancing communications with calculation in parallel processing. The conclusion, for this problem, is that parallel designs must minimize the frequency of synchronizing events and minimize the communications volume on occasions when communication is necessary. The greater the amount of uninterrupted work that a processor can accomplish, the better. While control, i.e., blocking communications, synchronization, loop–by–loop data distribution, is necessary it will have adverse impacts on performance. The individual processors of a multiprocessor system should be granted the maximum degree of independence that the mission will allow.

While there is undoubtedly some room for improvement in the complete pivoting code, it would appear that maximum efficiencies of approximately 22%, 40%, and 70% for hypercubes of order three, two, and one, respectively, are likely on the iPSC/2. The same code seems to be headed for somewhat better performance

117

on the transputers, but with the shortage of memory, it is difficult to extrapolate and determine the direction of the plots. The higher order cubes appear to flatten at about the same efficiency that the iPSC/2 showed as a terminal efficiency.

The partial pivoting code, on the other hand, exhibits the kind of characteristics that we like to see in parallel code. Both systems show efficiencies rising sharply (again, the size limit for the transputers is unfortunate) and the iPSC/2 shows some very nice results as the dimension of the matrix exceeds about 250.

B. THE TERAFLOP RACE

One of the biggest challenges to parallel computing today can be found in the "teraflop race". There are at least three competitors with teraflop initiatives: the United States, Europe, and Japan. The United States effort centers around Intel with projects like Touchstone (Chapter I). The European effort relies on the T9000 transputer. Considering the three to five year old technology used for this research, together with the numbers that the various parallel computer designers boast today, it seems that we might see teraflop performance by the mid–1990s. C. Gordon Bell claims that the teraflop is conceivable [Ref. 6: p. 1099]

> *Two relatively simple and sure paths exist for building a system that could deliver on the order of 1 teraflop by 1995. They are: (1) A 4K node multicomputer with 800 gigaflops peak or a 32K node multicomputer with 1.5 teraflops. (2) A Connection Machine with more than one teraflop and several million processing elements.*

Current products suggest that INMOS and Intel will be among the most likely competitors. Table 7.1, adapted from Jack Dongarra's report [Ref. 8: p. 20], shows how transputer–based systems compare to Intel products. This Table summarizes a test involving the solution for a 1000×1000 system of linear equations. The processors used for my thesis show floating–point capabilities of 0.37 Mflops (T800–20) and 0.16 Mflops (Compaq 386/20 with 80387) in Dongarra's report [Ref. 8: pp. 14, 16].

118

TABLE 7.1: PARALLEL MACHINE COMPARISON

Computer	t_1	p	t_p	Speedup	Efficiency
Parsytec FT–400	1075	400	4.90	219.0	.55
Parsytec FT–400	1075	256	6.59	163.0	.64
Parsytec FT–400	1075	100	13.20	81.4	.81
Parsytec FT–400	1075	64	19.10	56.3	.88
Parsytec FT–400	1075	16	69.20	15.5	.97
Intel iPSC/860	59	32	5.30	11.0	.34
Intel iPSC/860	59	16	6.80	8.7	.54
Intel iPSC/860	59	8	10.60	5.6	.70

The iPSC/860 illustrates the most recent technology and shows excellent uniprocessor performance (6.5 Mflops) [Ref. 8: p. 9]. The T800 transputer that Parsytec used is somewhat dated and will soon be replaced by the T9000. Nevertheless, the transputer–based system shows good parallel performance. The *times* of execution in the experiments of this thesis also indicate that the T800 is faster for floating–point calculations than the 386/387 combination in the iPSC/2.

C. FURTHER WORK

My research suggests many areas for further investigation. The method of conjugate gradients shows a great deal of promise as a candidate for parallelization. Indeed, it was the original aim of this thesis, but the development of other portions of the code required a great deal of time. The parallel CG algorithm should be relatively simple to code and holds great potential with respect to performance. Additionally, it possesses a nontrivial derivation and the theory behind the algorithm would be interesting to develop.

There are many other variations on Gauss factorization that could be coded and tested. While the programs presented in this thesis are designed in an effort to produce efficient performance, there is undoubtedly much that might be done to enhance this code. Among the options: at a very basic level, we could begin with

119

other distributions of the matrix A. A block method or row method may actually yield better performance. As the LINPACK benchmarks seem to use blocks, this is probably worth pursuing.

General purpose parallel computing, the ability to rely on parallel architectures for general purpose computation without a need for investigation to be more concerned with the architecture than the problem being computed, still requires much work. The ability to use parallel architectures as a computational tool to solve problems will mark an increasing maturity in this field.

Applying object–oriented design and programming paradigms to the parallel world may hold a great deal of promise. In particular, the C++ language seems to be a prudent choice for parallel programming.

In addition to the more practical options, the study of parallel theory and algorithms seems interesting and shows a great need for development. In particular, this field seems to need a more–or–less general (at least for MIMD machines) approach to classifying parallel algorithms and specifying their performance. As noted in Chapter IV, a mixture of this field with graph theory may hold a great deal of promise.

On an initial glance, the use of the Ada programming language with its inbuilt tasking constructs might seem optimum for the type of computing investigated in this thesis. Ada, in this regard, however, is optimized for use with shared memory multiprocessors. The use of Ada on transputers still requires much experimentation and better tools. Presently only one, rather expensive, Ada compiler is available for transputer use. Its required use of *occam* harnesses makes using Ada on transputers awkward at best. Further research is needed to create a better environment for Ada programming on transputers. Given the significance of Ada to the DoD establishment, this should become a priority. The inclusion of a standard math package and the advent of Ada 9X may hold some promise in this regard.

APPENDIX A
NOTATION AND TERMINOLOGY

This appendix explains the shorthand used in the rest of the thesis. *Conventions*, by definition, are generally accepted rules of the business. This would seem to obviate the need for further discussion of conventions, but there are several good reasons for discussing notation and terminology. First, the notation may not be conventional. In the absence of convention (or when the foundation that it provides is inadequate) a more substantial agreement is required. Second, even for conventional notation, the audience may be diverse enough to warrant familiarization. The following discussion provides this familiarity and gives the terms of an agreement to establish the meaning of the words and symbols used in the rest of the work. On occasion, neither convention nor this agreement will suffice. These situations will be handled case–by–case with the philosophy that clarity should never be sacrificed for brevity.

A. BASICS

Most of the work deals with the integers, $\mathbb{Z}$ (from the German word for numbers, Zahlen), the set of real numbers, $\mathbb{R}$, and the complex numbers, $\mathbb{C}$. Often, the German $\Re$ is used to represent the reals. A complex number is a number, $x + iy = z \in \mathbb{C}$, that has a *real part* ($x \in \Re$) and an *imaginary part* ($y \in \Re$), with the *complex unit* $i = \sqrt{-1}$. Sometimes the real part is denoted $\text{Re}(z)$ and $\text{Im}(z)$ is used to represent the imaginary part.

A *scalar* is simply a real number, and is usually denoted by a lower–case Greek letter.[1] A *vector* is an ordered set of scalars. Lower–case Latin letters like b, x, and y are used to denote vectors. Sometimes an arrow is placed above the name of a vector—like $\vec{x}$—to emphasize the fact that it is a vector.

---

[1] The Greek alphabet is shown in the **Table of Symbols**.

Matrices are two dimensional and usually contain real or complex elements. Capital letters (Greek or Latin) are used to represent matrices. Common examples include A, P, Q, R, Λ, and Σ.

The number systems introduced above cannot be represented in a finite space. There are two basic problems. First, we should consider the size (or *cardinality*) of the sets. The integers are countable or *denumerable* since there exists a one–to–one mapping between $\mathbb{Z}$ and the natural numbers, $\mathbb{N}$. This is an advantage in finite storage since it means that we can choose a finite *range* of the integers and be quite certain that *every integer* in that range is represented (exactly). Even though $\mathbb{Z}$ is denumerable, it is a set with *infinite* cardinality.

The real numbers present a more difficult situation for finite storage. The real number line is dense in comparison to the integers. $\Re$ is not only an infinite set, it is not countable (i.e., $\Re$ is *uncountable*). It is said to have the *power of the continuum*. To represent a real number, x, we use the floating–point approximation, fl(x), to x. This is a number that may be described by three parts: the sign s, the exponent e, and the mantissa d. An illustration of such a number is provided in Chapter II.

B. COMPLEX NUMBERS

1. Notation

The previous section introduced one notation for complex numbers; namely, $z = x + iy$. There are several other representations, each of which makes its own contribution in practical use. Electrical engineers usually replace the i with j since i is used to represent electrical current. Since the complex number can be represented by an ordered pair of real numbers, the graphical notation of Figure A.1 is natural. In this plane, the real and imaginary axes are used to represent the components of a complex number.
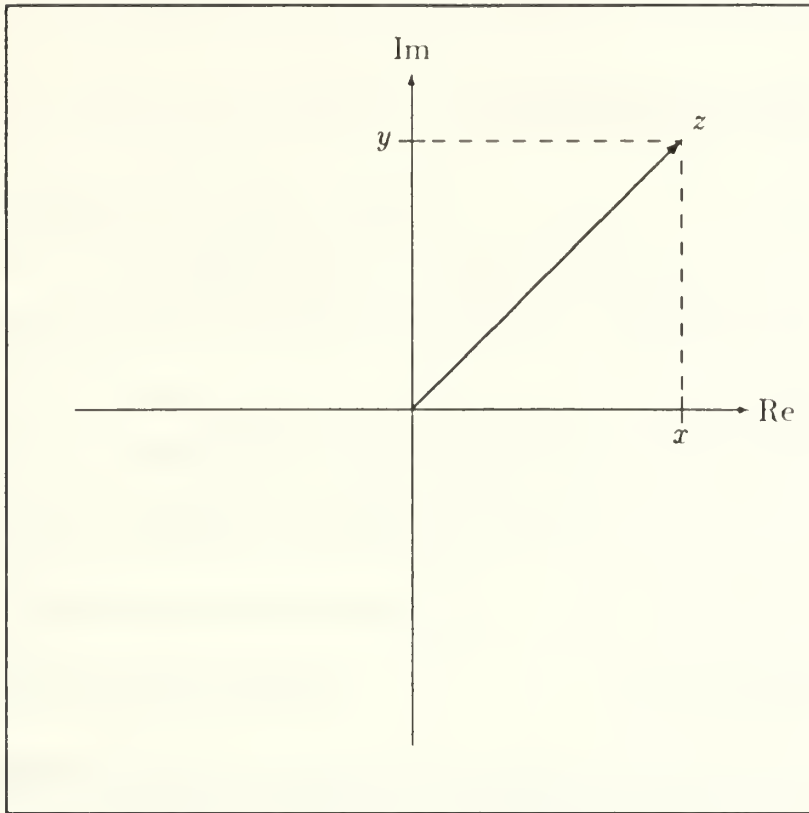
Figure A.1: The Complex Plane

The vector sum of these two parts, $\vec{z} = \vec{x} + \vec{y}$, is an equivalent and useful way to model complex numbers. There is yet another way to describe z. Let r be the magnitude of the vector z and let θ be the angle measured from the positive real axis counter-clockwise to z. Using this notation, we could use trigonometry to describe the complex number as $z = r(\cos\theta + i\sin\theta)$. The Euler formula [Ref. 32: p. 74],

$$e^z = e^{x+iy} = e^x e^{iy} = e^x(\cos y + i\sin y), \tag{A.1}$$

can be used to convert a complex number to yet another form: $z = re^{i\theta}$.

2. Operations

a. Addition and Subtraction

Addition and subtraction of complex numbers is performed in the same manner that vectors are added or subtracted. For instance, let $z_1 = a + ib$ and let $z_2 = c - id$. Then the sum, $z_1 + z_2$, is the same as the sum of the corresponding vectors:

$$z_1 + z_2 = \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} c \\ -d \end{bmatrix} = \begin{bmatrix} a + c \\ b - d \end{bmatrix} \tag{A.2}$$

so the sum is $z_1 + z_2 = (a + c) + i(b - d)$. Differences are handled in the obvious way, as vector differences.

b. Multiplication

Multiplication is performed by applying high school algebra. For the same complex numbers z_1 and z_2:

$$z_1 \times z_2 = (a + ib)(c - id) = ac - (a)(id) + (ib)(c) - (ib)(id) \tag{A.3}$$

and using the definition of the complex unit, $i = \sqrt{-1}$, we may combine the middle terms and move the $i^2 = -1$ outside the last term to find the (complex) product:

$$z_1 \times z_2 = ac - i(ad - bc) + bd = (ac + bd) - i(ad - bc) \tag{A.4}$$

c. Conjugation

The *complex conjugate* of a complex number $z = x + iy$ is defined as $\bar{z} = x - iy$. This simple operation finds practical application in complex **division**.

d. Division

Consider the quotient (z_1/z_2) of the same complex numbers that were used in equations A.2, A.3, and A.4. If we multiply both the numerator and the

124

denominator by the complex conjugate of the denominator, $\bar{z}_2$, we have:

$$\frac{z_1}{z_2} = \frac{a + ib}{c - id} = \frac{(a + ib)(c + id)}{(c - id)(c + id)} = \frac{ac + i(ad) + i(bc) + i^2(bd)}{c^2 - i^2 d^2} \tag{A.5}$$

and then, by applying $i^2 = -1$, we conclude:

$$\frac{z_1}{z_2} = \frac{ac - bd + i(bc + ad)}{c^2 + d^2} = \frac{(ac - bd)}{(c^2 + d^2)} + i\frac{(bc + ad)}{(c^2 + d^2)} \tag{A.6}$$

As a practical matter, this is not the way we would *compute* a complex quotient. The code given in Appendix F (function **cdiv()** in **complex.h**) provides a method that is better suited to the finite precision environment.

C. VECTORS AND MATRICES

1. Columns and Rows

Vectors are ordered collections of scalars represented as columns. Let $\alpha, \beta, \gamma \in \mathbb{C}$ with $\alpha = 1.0 + i4.0$, $\beta = 2.0 - i5.0$, and $\gamma = 3.0 + i6.0$. Then:

$$x = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} 1.0 + i4.0 \\ 2.0 - i5.0 \\ 3.0 + i6.0 \end{bmatrix}$$

If row–orientation is intended the *transpose* is used:

$$x^T = [\, \alpha \ \ \beta \ \ \gamma \,] = [\, (1.0 + i4.0) \ \ (2.0 - i5.0) \ \ (3.0 + i6.0) \,]$$

Matrices may be formed as ordered combinations of elements, vectors, or blocks. Suppose that $\mu = 3.0$ and $\nu = 7.0$. Then, with x as given above, the following matrices are equivalent:

$$A = \begin{bmatrix} x & \mu x & \nu x \end{bmatrix} = \begin{bmatrix} 1.0 + i4.0 & 3.0 + i12.0 & 7.0 + i28.0 \\ 2.0 - i5.0 & 6.0 - i15.0 & 14.0 - i35.0 \\ 3.0 + i6.0 & 9.0 + i18.0 & 21.0 + i42.0 \end{bmatrix} \tag{A.7}$$

An element within a matrix is usually denoted $A(i, j)$, where i is the row index and j is the column index. For instance, $A(1, 3) = 7.0 + i28.0$ in (A.7).

125

A *block* of the matrix A is a rectangular matrix B within A. MATLAB notation is useful. For instance, $B = A(i : j, k : l)$ means that B is the block of A's rows i through j and columns k through l. The row or column ':' means all rows or all columns. For instance:

$$B = A(:, 1 : 2) = \begin{bmatrix} 1.0 + i4.0 & 3.0 + i12.0 \\ 2.0 - i5.0 & 6.0 - i15.0 \\ 3.0 + i6.0 & 9.0 + i18.0 \end{bmatrix} \tag{A.8}$$

As a sidenote, a number with a decimal point should usually be taken as a real number. Mathematically speaking, $1 = 1.0$. But many compilers treat 1 as an integer and use the decimal point to recognize 1.0 as a floating–point value. Therefore, all of the code associated with this work and most of the examples use the decimal point as a clue that the number is a real number or its floating–point approximation.

2. Conjugation and Transposition

The conjugate of a vector or matrix is simply a vector or matrix whose entries are the conjugates of the original entries. A superscript C is used to denote the conjugate of a vector or matrix. For instance, with A as given A.7,

$$A^C = \begin{bmatrix} 1.0 - i4.0 & 3.0 - i12.0 & 7.0 - i28.0 \\ 2.0 + i5.0 & 6.0 + i15.0 & 14.0 + i35.0 \\ 3.0 - i6.0 & 9.0 - i18.0 & 21.0 - i42.0 \end{bmatrix} \tag{A.9}$$

The transpose of a vector or matrix, denoted with a superscript T, refers to a transposition of its rows and columns. With $A \in \mathbb{C}^{m \times n}$, the effect of transposition is that $A(i, j) = A^T(j, i)$ for all i such that $1 \leq i \leq m$, and all j so that $1 \leq j \leq n$. For example, consider the transposition of the matrix A that is found in equation A.7.

$$A^T = \begin{bmatrix} x^T \\ \mu x^T \\ \nu x^T \end{bmatrix} = \begin{bmatrix} 1.0 + i4.0 & 2.0 - i5.0 & 3.0 + i6.0 \\ 3.0 + i12.0 & 6.0 - i15.0 & 9.0 + i18.0 \\ 7.0 + i28.0 & 14.0 - i35.0 & 21.0 + i42.0 \end{bmatrix} \tag{A.10}$$

In this example we see that the columns of a matrix become the rows of its transpose. This example also demonstrates that when we first transpose, and then stack the columns of a matrix, we arrive at the transpose of the matrix. In the event that $A = A^T$, we say that A is symmetric.

The conjugate (or *Hermitian*) transpose of A is A^H. This matrix is the result of combining the conjugation and transposition operations on A. The following example shows the Hermitian transpose of A:

$$A^H = \begin{bmatrix} 1.0 - i4.0 & 2.0 + i5.0 & 3.0 - i6.0 \\ 3.0 - i12.0 & 6.0 + i15.0 & 9.0 - i18.0 \\ 7.0 - i28.0 & 14.0 + i35.0 & 21.0 - i42.0 \end{bmatrix} \tag{A.11}$$

If $A = A^H$, we say that "A is Hermitian." We should never confuse "A is Hermitian" with "A Hermitian" (the conjugate transpose, A^H, of A). [Ref. 33: p. 294]

3. Zeros

It could be argued that zero is the most important number. In addition to its use as a number, zero is also used to represent a vector or matrix in which every element is equal to zero. In the (extremely rare) event that the context does not clearly indicate the size of a "0–vector" or "0–matrix", its size will be given explicitly. In the absence of implied or specified size, 0 should be interpreted as the number zero. Additionally, blank space within a matrix usually means that all elements in that region are zero.

4. Special Forms

a. Axis Vectors

An *axis vector*, e_i, is simply the ith column (or row) of the identity matrix.

b. Lower Triangular

A *lower triangular matrix*, usually denoted L, has the form

$$L = \begin{bmatrix} \times & & \\ \times & \times & \\ \times & \times & \times \end{bmatrix} \qquad \text{(A.12)}$$

If L has ones on the diagonal, it is called *unit* lower triangular. Similarly, the **upper triangular** matrix U has the form

$$U = \begin{bmatrix} \times & \times & \times \\ & \times & \times \\ & & \times \end{bmatrix} \qquad \text{(A.13)}$$

U is called *unit* upper triangular if the diagonal elements are all ones. Sometimes (e.g., Chapter III) such a matrix is called *right* triangular and denoted R. When the matrix is not square, the lower and upper triangular ideas are translated to lower and upper *trapezoidal*, with the *unit* trapezoidal matrices having ones on the diagonal. The following matrices illustrate the different kinds of trapezoidal matrices. The matrices may be tall and skinny as

$$U = \begin{bmatrix} \times & \times & \times \\ & \times & \times \\ & & \times \\ & & \\ & & \end{bmatrix} \qquad L = \begin{bmatrix} \times & & \\ \times & \times & \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}, \qquad \text{(A.14)}$$

or short and fat

$$U = \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \end{bmatrix} \qquad L = \begin{bmatrix} \times & & & \\ \times & \times & & \\ \times & \times & \times & \end{bmatrix}. \qquad \text{(A.15)}$$

D. NORMS

The information below was taken from [Ref. 21: pp. 53–60], so it seems fitting to begin with a few of Golub and Van Loan's comments on norms.

Norms serve the same purpose on vector spaces that absolute value does on the real line: they furnish a measure of distance. More precisely, $\Re^n$ together with a norm on $\Re^n$ defines a metric space. Therefore, we have the familiar notions of neighborhood, open sets, convergence, and continuity when working with vectors and vector-valued functions.

1. Vector Norms

a. Definition

A *vector norm* on $\Re^n$ is a function $f : \Re^n \to \Re$ that satisfies the following properties [Ref. 21: p. 53]:

$$f(x) \geq 0 \quad x \in \Re^n, \quad (f(x) = 0 \; iff \; x = 0) \tag{A.16}$$

$$f(x + y) \leq f(x) + f(y) \quad x, y \in \Re^n \tag{A.17}$$

$$f(\alpha x) = |\alpha| f(x) \quad \alpha \in \Re, x \in \Re^n \tag{A.18}$$

We denote such a function with a double bar notation: $f(x) = \| x \|$.

b. The p–Norm

Subscripts on the double bar are used to distinguish between various norms. The most popular example of this is the p–norm, $\| \cdot \|_p$. This norm is defined by [Ref. 21: p. 53]

$$\| x \|_p = (| x_1 |^p + \cdots + | x_n |^p)^{\frac{1}{p}} \quad p \geq 1. \tag{A.19}$$

The 2–norm is the one used most frequently in this work, but the 1– and ∞–norms find frequent application in other work. A natural representation of the 2–norm is the square root of an inner product

$$\| x \|_2 = (| x_1 |^2 + \cdots + | x_n |^2)^{\frac{1}{2}} = \sqrt{x^T x} \tag{A.20}$$

The 2–norm of x is the *Euclidean length* of the vector x.

2. Matrix Norms

a. *Definition*

A *matrix norm* on $\Re^{m \times n}$ is a function $f : \Re^{m \times n} \to \Re$ that satisfies properties similar to those presented in the vector case [Ref. 21: p. 56]:

$$f(A) \geq 0 \qquad A \in \Re^{m \times n}, \qquad (f(A) = 0 \; iff \; A = 0) \quad \text{(A.21)}$$

$$f(A + B) \leq f(A) + f(B) \qquad A, B \in \Re^{m \times n} \tag{A.22}$$

$$f(\alpha A) = | \alpha | f(A) \quad \alpha \in \Re, A \in \Re^{m \times n} \tag{A.23}$$

Matrix norms also use the double bar notation: $f(A) = \| A \|$. The Frobenius norm and the p-norm are the most common matrix norms

b. *Frobenius*

The Frobenius norm is defined as

$$\| A \|_F = \sqrt{ \sum_{i=1}^{m} \sum_{j=1}^{n} | a_{ij} |^2 }. \tag{A.24}$$

c. *p-Norms*

The p-norm of a matrix, A, is defined by

$$\| A \|_p = \sup_{x \neq 0} \frac{\| Ax \|_p}{\| x \|_p}. \tag{A.25}$$

E. LINEAR SYSTEMS

One of the fundamental tasks of linear algebra is to form a matrix representation of a system of linear equations. Consider the system of linear equations:

$$\begin{array}{rrrrrrr}
2u_1 & + & 3u_2 & - & 4u_3 & = & 7 \\
3u_1 & - & 5u_2 & + & 7u_3 & = & 3 \\
4u_1 & + & 6u_2 & - & 2u_3 & = & 1
\end{array} \tag{A.26}$$

This system of equations can be expressed using the matrix notation $Au = b$

$$Au = \begin{bmatrix} 2 & 3 & -4 \\ 3 & -5 & 7 \\ 4 & 6 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 3 \\ 1 \end{bmatrix} = b \tag{A.27}$$

F. MEASURES OF COMPLEXITY

The first, and most rudimentary requirement for an algorithm is that it produce the correct answer. This seems utterly obvious, but it must never be lost in the algorithm designer's pursuit of the next most important elements—efficiency in using time and space. For the moment, we shall assume that the algorithm arrives at an acceptable answer. Then the algorithm's use of time and space becomes a very serious subject. Knuth provides the notation in [Ref. 34].

The *time complexity* of an algorithm, also known as *running time*, describes how the program works under a stopwatch. *Space complexity* is the amount of temporary storage required to carry out the algorithm. For example, suppose a person stood at a chalkboard, ready to solve a problem. We would not regard the input or output storage space, but only the required space on the chalkboard, in the space complexity of the problem. Usually we like to link the idea of complexity to the input size of the problem, n. The following discussion of time complexity outlines a few tools that are standard in the study of algorithms. The same tools and ideas apply for space complexity analysis. [Ref. 35: pp. 42–43]

The most common method for describing the time complexity of an algorithm is the "big–Oh" notation [Ref. 35: p. 39].[2] A function $g(n)$ is $O(f(n))$ if there exist constants c and N so that, for all $n \geq N$, $g(n) \leq cf(n)$.

$$g(n) = O(f(n)) \Longleftrightarrow g(n) \leq cf(n), \quad n \geq N \tag{A.28}$$

---

[2] $O(f(n))$ is read "order $f(n)$."

This means that for a large enough problem size n, the time to execute $g(n)$ is a constant multiple of some function, $f(n)$. Big–Oh notation *does not mean a least upper bound*, only an upper bound for n sufficiently large. Practically, $O(f(n))$ must be augmented so that we may determine how tightly $cf(n)$ bounds $g(n)$.

By adding a *lower bound* to big–Oh, we may arrive at a more informative statement concerning an algorithm's complexity. This is achieved through the use of "big Omega". $T(n) = \Omega(g(n))$ means that there exist constants c and N such that, for all $n \geq N$, the number of steps $T(n)$ required to solve the problem for input size n is at least $cg(n)$.

$$T(n) = \Omega(g(n)) \iff T(n) \geq cg(n), \quad n \geq N \qquad \text{(A.29)}$$

This is essentially a lower bound on time complexity. If a function, $f(n)$ satisfies both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$—not necessarily using the same constants c and N for both O and Ω—then we say that $f(n) = \Theta(g(n))$. [Ref. 35: p. 41]

$$f(n) = O(g(n)) = \Omega(g(n)) \iff f(n) = \Theta(g(n)), \quad n \geq N \qquad \text{(A.30)}$$

Now and then, notation similar to O and Ω is required except that a *strict* inequality is desired. In this case, we use "little oh" and "little omega". The definitions are:

$$f(n) = o(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \iff g(n) = \omega(f(n)) \qquad \text{(A.31)}$$

We have seen that O, Ω, Θ, o, and ω are roughly equivalent to the inequalities $\leq$, $\geq$, $=$, $<$, and $>$, respectively. Is this notation meaningful? Does it have utility in problem solving? The answer is a guarded "yes." We must understand the purpose of the notation. It *cannot substitute for timing data* taken from the actual execution of an algorithm. It is intended as a good first estimate. There are too many variables involved in modern tools and machinery to expect accurate analysis from other than actual execution.

TABLE A.1: ALGORITHM COMPLEXITY AND MACHINE SPEED

Algorithm Complexity	Execution Time (in Seconds) for Machine Speed			
	1000 steps/sec	2000 steps/sec	4000 steps/sec	8000 steps/sec
$\log_2 n$	0.01	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Nevertheless, a rough estimate of how a problem grows is important to the problem solving process. Indeed, experimental results and complexity analysis should not usually be considered independently, but compared and used as complementary instruments. The time complexity of an algorithm is, in a sense, more important than the speed of the machine upon which it is executed. Consider the data in Table A.1 (adapted from [Ref. 35: p. 41]). This is based upon a problem of size $n = 1000$ and demonstrates the ability of an algorithm to dominate a machine. For this reason, and with these conditions clearly established, we will find many occasions to use time– and space–complexity notation.

Finally, the two most common performance measures for *parallel* computing are *speedup* and *efficiency*. Suppose that T_n is the time of execution for a particular algorithm, A, on n processors. Consider the best uniprocessor time T_1 for a sequential version of A compared to the execution of an equivalent (not necessarily the same) parallel program on P processors that executes in time T_P. Then speedup, S_P, is defined as

$$S_P = \frac{T_1}{T_P}$$

and the efficiency, E_P, is defined to be

$$E_P = \frac{S_P}{P}.$$

APPENDIX B
EQUIPMENT

A transputer is a microcomputer with its own local memory and with links for connecting one transputer to another transputer.

*The transputer architecture defines a family of programmable VLSI components. The definition of the architecture falls naturally into the **logical** aspects which define how a system of interconnected transputers is designed and programmed, and the **physical** aspects which define how transputers, as VLSI components, are interconnected and controlled.*

A typical member of the transputer product family is a single chip containing processor, memory, and communication links which provide point to point connection between transputers. In addition, each transputer product contains special circuitry and interfaces adapting it to a particular use. For example, a peripheral control transputer, such as a graphics or disk controller, has interfaces tailored to the requirements of a specific device.

A transputer can be used in a single processor system or in networks to build high performance concurrent systems. A network of transputers and peripheral controllers is easily constructed using point-to-point communication.

— INMOS

This introduction is provided by the transputer's maker in [Ref. 36: p. 7].

A. TRANSPUTER MODULES

INMOS makes a wide variety of microprocessors to suit differing needs. To provide a simple, modular interface they have developed the notion of a transputer module (TRAM). The TRAM is a small board containing the microprocessor, RAM, other circuitry, and a standard sixteen signal interface.

B. THE IMS B012

Most of the later experiments were carried out on an IMS B012 board. This board accommodates sixteen transputers; each of which is installed on its own IMS

B401 TRAM. In our case the TRAM holds 32 kilobytes of memory (in addition to the four kilobytes onboard the T800-20 transputer).

d. INMOS Transputers

The INMOS transputer gives the system designer a tremendous amount of latitude. With these processors—perhaps more than with any other parallel architecture—one should give careful thought to the size, component processors, and interconnection topology as the *first elements in designing a solution to a problem.* This cannot be overemphasized. When the hardware is not "general purpose" in nature, it must receive thoughtful consideration along the path to solving the problem. Some of the largest applications for parallel machines—especially for transputers—are embedded systems.

An embedded computer system is defined as "one that forms a part of a larger system whose purpose is not primarily computational." [Ref. 37: pp. 15–16] To automatically accept or assume a particular machine configuration is to relinquish control of one of the tools available in system design.

Transputer is the name given to the members of a family of microprocessors. While INMOS is the largest producer of these processors, they have not chosen to protect the name *transputer* with any sort of trademark. The name comes from a combination of "**trans**istor com**puter**" and each transputer is essentially a computer on a chip. The chip possesses an arithmetic logic unit (ALU), memory, and a communication system that supports bidirectional serial communication links. Most of the transputers used for this research also include a 64–bit (IEEE 754 standard) floating-point unit (FPU).

The transputer module (TRAM) is the most common package for transputers. The capabilities of these modules are quite diverse, but they hold to a standard interface design. This makes the TRAM easy to use. Systems designed

around TRAMS enjoy simple replacement of components, ease of modification, and great scalability. Indeed, the laboratory environment in which these TRAMs were exercised is a very dynamic one.

The PARCDS laboratory has six 80286–based IBM–compatible personal computers, each of which contains a transputer interface board. Five hold IMS B004 boards and one holds a Transtech TMB08 board. The B004 boards each have two megabytes of memory and an IMS T414 transputer in addition to the requisite serial–to–parallel converter and interface circuits. The TMB08 holds four megabytes of memory and an IMS T800–20 transputer. These "host" machines can each be connected to an arbitrarily large network of transputers.

For this purpose, we have two INMOS Transputer Evaluation Module (ITEM) boxes. These boxes can hold at least ten boards of the Double Eurocard size (approximately 22 cm × 23.5 cm). Of primary interest for this thesis was the IMS B012 board; a motherboard capable of supporting sixteen TRAMs. For this research, all sixteen slots were filled with a TRAM that held an IMS T800–20 transputer and 32 kilobytes of TRAM memory (in addition to the transputer's four kilobytes). The shortage of memory is probably the greatest deficiency and indicator of the outdated nature of these processors. TRAMs with four and eight megabytes of memory and IMS T805–25 transputers are currently available for less than $900.00 and $1,300.00 respectively.

e. Intel iPSC/2

The iPSC/2 used for this research contained eight node processors of the "CX" type (80386/80387 combination). Like the transputers, this machine is somewhat dated. Today's i860 chips have exceedingly more capacity.

C. SWITCHING METHODS

The iPSC/2 and transputer hardware use of different switching methods. Intel uses a *circuit switching* approach, whereas the INMOS approach is *store-and-forward* switching. Each approach has advantages and disadvantages. The circuit switching approach is "almost universally used for telephone networks." [Ref. 38: p. 12] The idea is to first define a path (close a circuit) from the source to the destination and then use it as a dedicated line.

This requires a start-up overhead that depends entirely upon the current load being handled by the system. If any part of the medium (links or switches) between the source and destination is busy, the message will wait at the source until the entire path is clear. The path is determined (in the iPSC/2 case) in a deterministic fashion, so that a message from node i to node j will always insist on a particular path, even if some other communication is blocking that path. As the path becomes clear, switches between the source and destination are set so that a dedicated line will exist from source to destination.

After the overhead of establishing (closing) the circuit has been paid, communication proceeds at a rapid rate. The intermediate nodes along the path do not store the message. Instead, their switches have been set so that the message flows through. Intuitively, this approach should be quite effective in a network with a very structured interconnection topology and a relatively small number of nodes. The hypercube gives us this structure. Hypercubes of order three or four are probably small enough to avoid difficulties that might arise as many nodes contend for the same medium.

The store-and-forward approach does not require the availability of the entire path between source and destination nodes. Instead, each node along the path accepts the entire message in turn and then forwards it to the next node in the path.

This requires the use of no more than one link at a time. For a many–node environment (particularly if there is little structure or the potential of dynamic routing), this approach would seem to offer some advantages over the circuit switching approach.

The routing criteria is separate from the type of switching used. Either of the two general approaches described above can support many forms of routing. Deterministic approaches alone include many methods. For the hypercube topology with Gray–coded node labels, it is probably useful to combine the Gray code with the notion of Hamming distance to arrive at a shortest path route. Even with this approach, there are as many optimum paths between two nodes i and j as the Hamming distance, $H(i,j)$, between them. [Ref. 39: p. 7]. If a dynamic scheme is used to determine the path, there are even more combinations of potential paths from i to j. Usually a dynamic approach considers media utilization, "hot spot" avoidance, and so on.

APPENDIX C

INTERCONNECTION TOPOLOGIES

Multiprocessor computing brings with it a fundamental concern: interprocessor communication. Communication is—to any designer of computing machinery or software—a burden and hindrance. An *interconnection topology* describes the network that handles this load. The *hypercube* is one of the many topologies used in multiprocessor computing. It has been the subject of both hype and criticism. Nevertheless, this particular scheme possesses the qualities that quickly draw the attention of mathematicians and parallel programmers. The hypercube's structure and simplicity make it dependable and predictable. The same properties that enable the hypercube to endure the rigor of mathematical proof lead to practical solutions in parallel programming. This discussion describes the hypercube topology and explores some of the the qualities that make it a practical choice for multiprocessor computing.

A. A FAMILIAR SETTING

Organizing processors into a suitable topology is analogous to the familiar problem of organizing personnel into groups. An independent worker has limited capacity, so we often set more hands (or machinery) to the task for productivity's sake. Groups of people are often less efficient. *Efficiency* is a ratio of time spent doing useful work to the total time spent. Other metrics might work, but *time* is universally recognized as the standard against which productivity is measured. **Dependence upon others requires communication and consumes time**. The loss may be minimized, but not avoided. Any group working toward a common goal must deal with this problem. To be efficient, an organization must possess structure and media for communication.

People spend time on meetings, paperwork, and peripheral pursuits—all for the sake of an organization that hopes to outperform the individual. Organizations typically perform tasks that are simply impossible for an individual. To be sure, an

individual often possesses the independence and efficiency that makes him the proper choice. There are tasks that seem to fit one or the other and—while there is some crossover in ability—we aren't likely to get rid of either organizations or individual workers soon! This is worth considerable attention. Individuals and organizations are chosen for *different* tasks.

These ideas apply in the world of parallel processing. First, there are many tasks. Some fit nicely onto a single processor. Others beg a parallel solution. Finally, some have natural solutions by either method. Even when one of these options is selected, there are many ways to solve the problem. If a multiprocessor is used to solve the problem, the issue of communications will be unavoidable.

An interconnection topology must carry the burden of interprocessor communications. There are many schemes for handling this mission. This discussion focuses on one design that fulfills that mission: the hypercube. To forestall confusion: the subject is an interconnection topology, not a particular vendor's product.

B. APPEAL TO INTUITION

Productivity can suffer when the members of an organization communicate excessively. A lack of communication can also reduce efficiency. In a network of processors, lines of communication (links) are literal. The system will not be flexible if there is a shortage of links, but with too many links a message could get delayed or lost in the confusion. The hypercube attempts to strike a balance.

Hypercubes come in different sizes. In fact, scalability is a key characteristic of the hypercube. It allows the designer to tailor a network to a problem. There are several ways to express the cube's size: order is one measure. The term "hypercube of order n" (usually called an n-cube) is filled with meaning. A more detailed description is given later, but pictures provide the most direct introduction. Figure C.1 shows hypercubes of order n where $n \in \{0, 1, 2, 3\}$.
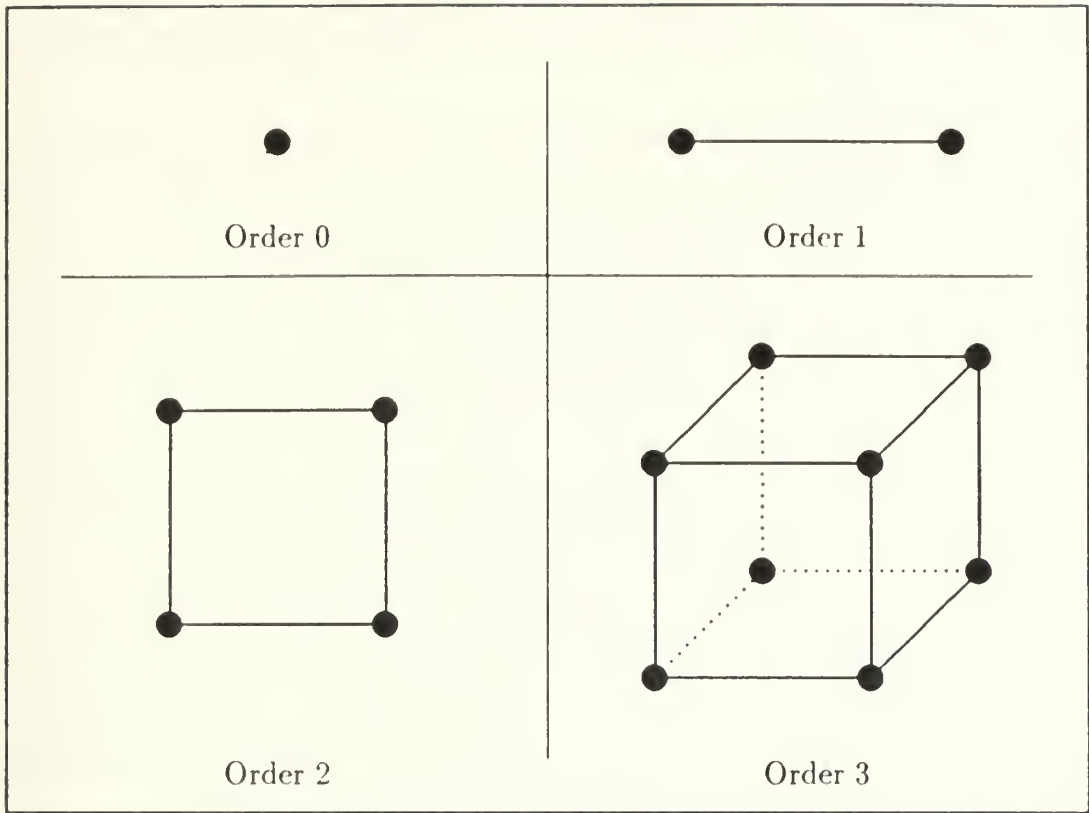
Figure C.1: The Four Smallest Hypercubes

This illustration is important. The hypercube shows geometry, structure, and symmetry. A few observations nearly jump out of the pictures. One can see several terms of a geometric series developing. There is also a recurrence relation at work in the building of hypercubes. Intuition suggests the use of well–oiled mathematical tools to analyze the hypercube.

C. TOOLS

Many benefits may be derived from a few definitions, conventions, and tools (that suit the hypercube's structure). Figure C.2 demonstrates the utility of Cartesian coordinates in n–dimensional space.

The picture is deceptively simple, but worth careful study. Figure C.2 shows a unit cube in three dimensions. The vertex labels express (xyz) position in the coor-
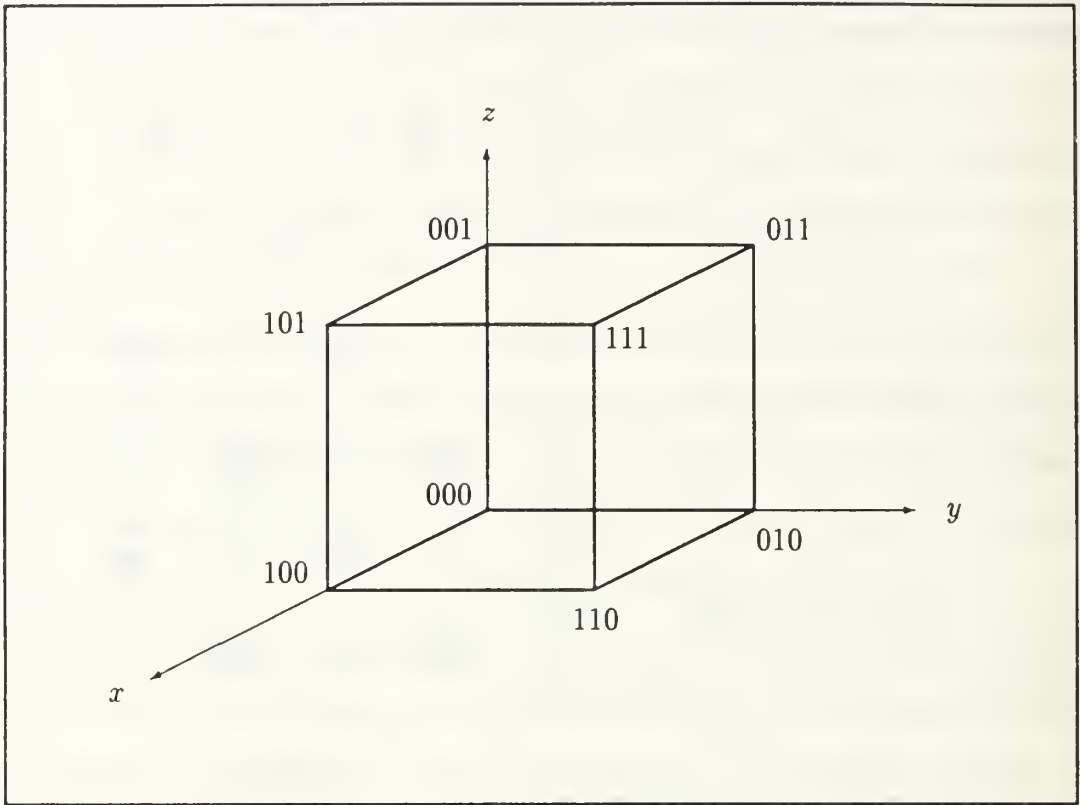
Figure C.2: Cartesian Coordinates for a 3–Cube

dinate system. The labels also form a binary (*Gray*) code that is somehow equivalent to coordinate labeling of a cube in n–dimensional space. The issue of communications invoked this discussion, so distance must be addressed. A comparison of the binary labels of any two nodes reveals that the distance between the nodes is equal to the number of bits that differ in the labels. This measure, called *Hamming distance*, and the Gray code are presented in more detail later.

This brief introduction is just enough to embark upon a more precise description of the hypercube. The ideas of a coordinate system, node labeling, and distance are fundamental. Graph theory also finds application in topology design. In the hypercube these four tools complement each other nicely. Despite their simplicity they can be explored in almost endless detail, even within the constraints of hypercube structure.

142

D. DESCRIBING THE HYPERCUBE

The hypercube interconnection topology cannot be captured in a one–sentence definition. A definition is often inappropriate for material objects. A description given from several perspectives may be more useful. This is the case with topologies. Each tool introduced above has its own utility. In a sense, each takes up a particular perspective. A meaningful characterization of the hypercube can be achieved by combining these perspectives.

The geometric view is most useful for visualizing the cubes. Despite its tendency to break down (with three–dimensional limitations), geometry's intuitive appeal is indispensable. Geometry and pictures lay the foundation for the setting of an undirected graph. Figures C.1 and C.2 take advantage of geometry, but three–dimensional sketches begin to lose their appeal as order increases. Nevertheless, geometry and visual models hold an important place in describing the hypercube. They furnish us with (a) examples for comparison, and (b) expectations that are useful in the transition to a more general description of the topology.

A hypercube of order n may be described as a set of 2^n points (vertices, nodes, or processors) connected by a set of edges. The points are each given an n–bit binary label, $b_n \ldots b_3 b_2 b_1$. Thus the hypercube's node labels exhaust all possible n–bit binary combinations. Furthermore, the labeling convention used in Figure C.2 describes the point's n–dimensional Cartesian coordinates.

The hypercube edge set (communication links) includes an edge between every pair of points p_i and p_j whose binary labels differ in exactly one bit position, say b_k. That is, adjacent nodes have a Hamming distance of one. This measure of distance proves especially convenient in the hypercube, and it can be thought of in several equivalent ways. A first definition of *Hamming distance* is the number of bits that differ in the two labels. Equivalently, it is the number of 1's in a bitwise exclusive

or (**XOR**) of the numbers. Figure C.2 contains an example. Let p_i be the point labeled **100** and p_j be **110**. The binary labels differ in exactly one bit position, namely b_2 (the second bit). The points are neighbors (one hop from each other in communications terms). [Ref. 40]

Despite the appeal of the geometric approach, it holds limited value in a general n–dimensional space. Consider $n = 4$ in three dimensions. Typical illustrations show the sixteen–node cube as a cube inside a cube with connections between corresponding nodes of the inner and outer cubes. An equivalent diagram would display two 3–cubes side–by–side with connections to corresponding nodes. Nevertheless, it seems that an n–dimensional coordinate system is the most convenient environment for sketching the hypercube of order n.

E. GREATER DIMENSIONS

Three–dimensional sketches become difficult to manage. The time comes for a change of method. Some of the finest tools available for spanning such a gap are recurrence relations and the principle of mathematical induction. The approach is not extremely formal, but those so inclined will not find it hard to add the formalities.

Induction can be used to generate a Gray code suitable for labeling the nodes of a hypercube. This code and the Hamming distance can be used to determine the cube. The first topic is a procedural description of how to build hypercubes. A Gray code construction procedure will follow. If the two topics appear similar, it is because they are completely equivalent (assuming that the Gray code is combined with the concept of Hamming distance).

Constructing a hypercube of order zero is trivial. This is not important except that it leads to greater things (i.e., it is the basis for induction). Second, suppose that this hypothesis for induction is true: "we know how to construct any hypercube of order k where $0 \le k < n$". Induction forms a hypercube of order n using this

base case and hypothesis. This can be done in three steps:

- Replicate the Hypercube of Order $(n - 1)$ so that there are two identical copies. For concreteness, one will be copy number **0** and the other will be copy number **1**. The hypercubes have $2^{(n-1)}$ nodes each.

- Prepend the copy number to the existing node labels. That is, place a leading **0** in front of the labels for each node of copy **0** and place a **1** in front of every node label in copy **1**. Now every node in one copy has a corresponding node in the other copy. These corresponding nodes are separated by a Hamming distance of one. That is, the last $(n - 1)$ bits are the same for corresponding nodes and they differ only in the prepended copy number.

- Connect all nodes whose labels differ only in the prepended copy number. This adds $2^{(n-1)}$ edges between the two copies.

F. GRAY CODE GENERATION

The procedure above generates hypercubes. By focusing on the vertex labels, Gray code generation can be discussed. A Gray code is a cyclic list of all of the n-bit numbers which changes in only one bit from one number to the next [Ref. 40]. Since the code is binary, there are 2^n numbers in the list. The starting point is arbitrary (it is cyclic) but I have started with zero. Perhaps the best explanation of Gray codes comes in the construction of one. As in the construction of hypercubes, a base case is required to begin generation.

- Start with 0. This is a one–bit number $(n = 1)$ so the one–bit Gray code must have a total of $2^1 = 2$ numbers. The other is 1. Next, the hypercube building steps established above are applied with slight modification.

- Given the one–bit case, it is easy to generate the $n = 2$ code. Write down the previous code and draw a line below it. Next, form a copy by *reflecting* the code

145

TABLE C.1: GRAY CODE GENERATION

<u>0</u>	00	000	0000
1	<u>01</u>	001	0001
	11	011	0011
	10	<u>010</u>	0010
		110	0110
		111	0111
		101	0101
		100	<u>0100</u>
			1100
			1101
			1111
			1110
			1010
			1011
			1001
			1000

downward across the line. Place a zero in front of each number in the previous code (above the line), and a one in front of each number in the new copy (below the line).

- This is a Gray code for $n = 2$. Table C.1 extends the idea. The list is cyclic, each number consists of n bits, and the list contains all 2^n possible numbers. To construct the code for larger n, the process may be applied repetitively. Copy by reflecting the $(n - 1)$-bit code downward across a line, prepend a zero to everything above the (most recent) line, and prepend a one to those below that line.

The Gray code is probably the most useful node labeling to attach to the hypercube. This code often appears in implementation. The program listing that begins on page 152 shows one way to generate the code. It can be used, for instance, as the

146

backbone of a routing function in a network. Labels with a Hamming distance of one mark neighbors in the hypercube. What about the labels of two nodes that differ in exactly k bits (i.e., have a Hamming distance of k)? It turns out that k is the distance (number of edges) between these nodes. For all communications between these nodes, the shortest path will involve k hops.

This also indicates that, for an n–cube, there is no pair of nodes that have a Hamming distance of more than n (e.g., communication between nodes 0000010 and 1111101 in a 7–cube can be achieved in seven hops). The greatest distance across the n–cube is n hops. In fact, for each node in a hypercube, there is a unique corresponding node at a Hamming distance of n. Also, there are n nodes at a Hamming distance of one from each of the hypercube's nodes.

Two approaches have been considered so far: sketching cubes in n–dimensional Cartesian coordinates and studying the labels associated with the cubes. Though the approaches are fundamentally different, they arrived at many of the same conclusions. Careful application of the Gray code and Hamming distance could produce a nearly endless string of results, but it is more convenient to introduce some material from the study of graphs at this point. Graph theory combines the two approaches: it looks at the pictures and studies the numbers as well. The small hypercubes described with earlier methods are given graph representation in the illustration of Figure C.3.

G. GRAPHS OF HYPERCUBES

Graph theory is, of course, much more sophisticated than the small subset used here. Buckley and Harary provide a valuable source [Ref. 41]. This discussion exposes a few salient features of the hypercube from the perspective of graphs.

A graph, H, consists of a vertex set, $V(H)$, and an edge set, $E(H)$. The vertices, or nodes, in the multiprocessor network model are the processors. The edges are the
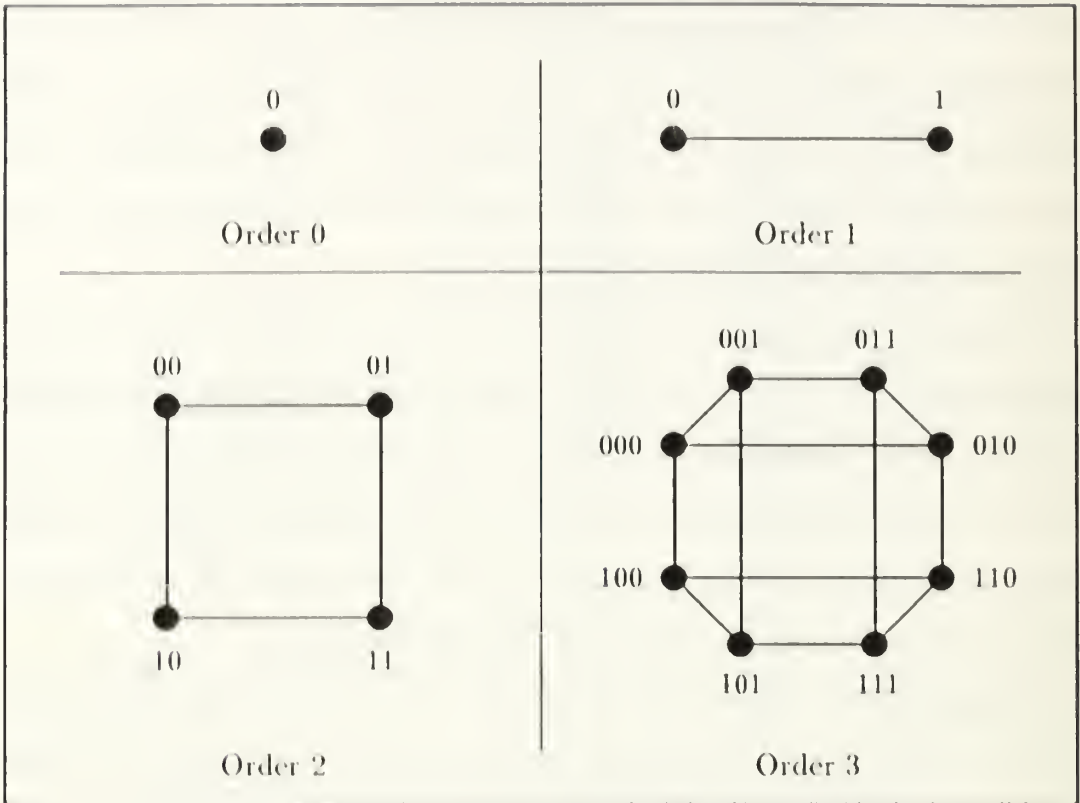
Figure C.3: Hypercube Graphs

links that connect the processors. I will avoid using the term *order* in its graph theory sense (i.e., number of nodes) so that it cannot be confused with the order of the hypercube. Consider the graph, H_n, of a hypercube of order n. The graph has these characteristics:

- There are 2^n nodes. This means that the number of nodes (i.e., processors) grows very quickly with order.

- Every vertex, v, in H_n has eccentricity $e(v) = n$. Eccentricity is the distance to a node farthest from v. Additionally, each node in a hypercube has **exactly** one eccentric (farthest) node. This property means that hypercubes are unique eccentric node (u.e.n.) graphs.

- The radius of a graph is the minimum eccentricity of the nodes and diameter is the maximum eccentricity. The hypercube is self-centered, meaning its radius and diameter are the same: $r(H_n) = d(H_n) = n$. This is significant because it says that worst-case communications distances only grow like the order of the hypercube.

- Connectivity is a measure of reliability or fault tolerance in multiprocessor networks. The connectivity of a hypercube is equal to the order of the cube, n. The edge connectivity is also n (each node has n incident edges).

Counting the number of nodes in a hypercube is easy. The hypercube construction process also points to a recurrence relation that reveals the number of edges in a hypercube. The initial case, of course, is the hypercube of order zero with no edges. After this, the number of edges can be expressed in terms of the size of the previous cube. Suppose a hypercube of order n has q edges. Then the hypercube of order $(n + 1)$ will have $2q + 2^n$ edges. This is because the construction procedure calls for two copies and 2^n edges between them.

Figure C.4 provides an example. This is the graph, H_4, of the hypercube of order four. All of the characteristics given above are evident. Additionally, a Gray code labeling of the nodes is given. The recurrence relation above is useful, but it retains a dependence upon q. A more convenient formula would depend on n alone.

In fact, there is a simple formula for the number of edges in the graph of a hypercube, but it requires a closer look at the recurrence relation. In more formal terms: let $q(n)$ represent the number of edges in a hypercube of order n. Then:

$$q(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2q(n-1) + 2^{(n-1)} & \text{if } n \geq 1 \end{cases}.$$

This can be expanded and shown equivalent to: $q(n) = n(2^{(n-1)})$. Table C.2 provides an example.

TABLE C.2: NODES AND EDGES FOR A HYPERCUBE

Order	Number of Nodes	Number of Edges
0	1	0
1	$2^1 = 2$	$2(0) + 2^0 = 1$
2	$2^2 = 4$	$2(1) + 2^1 = 4$
3	$2^3 = 8$	$2(4) + 2^2 = 12$
4	$2^4 = 16$	$2(12) + 2^3 = 32$
5	$2^5 = 32$	$2(32) + 2^4 = 80$
6	$2^6 = 64$	$2(80) + 2^5 = 192$
7	$2^7 = 128$	$2(192) + 2^6 = 448$
$\vdots$	$\vdots$	$\vdots$
$(n-1)$	$2^{(n-1)}$	q
n	2^n	$2q + 2^{(n-1)}$
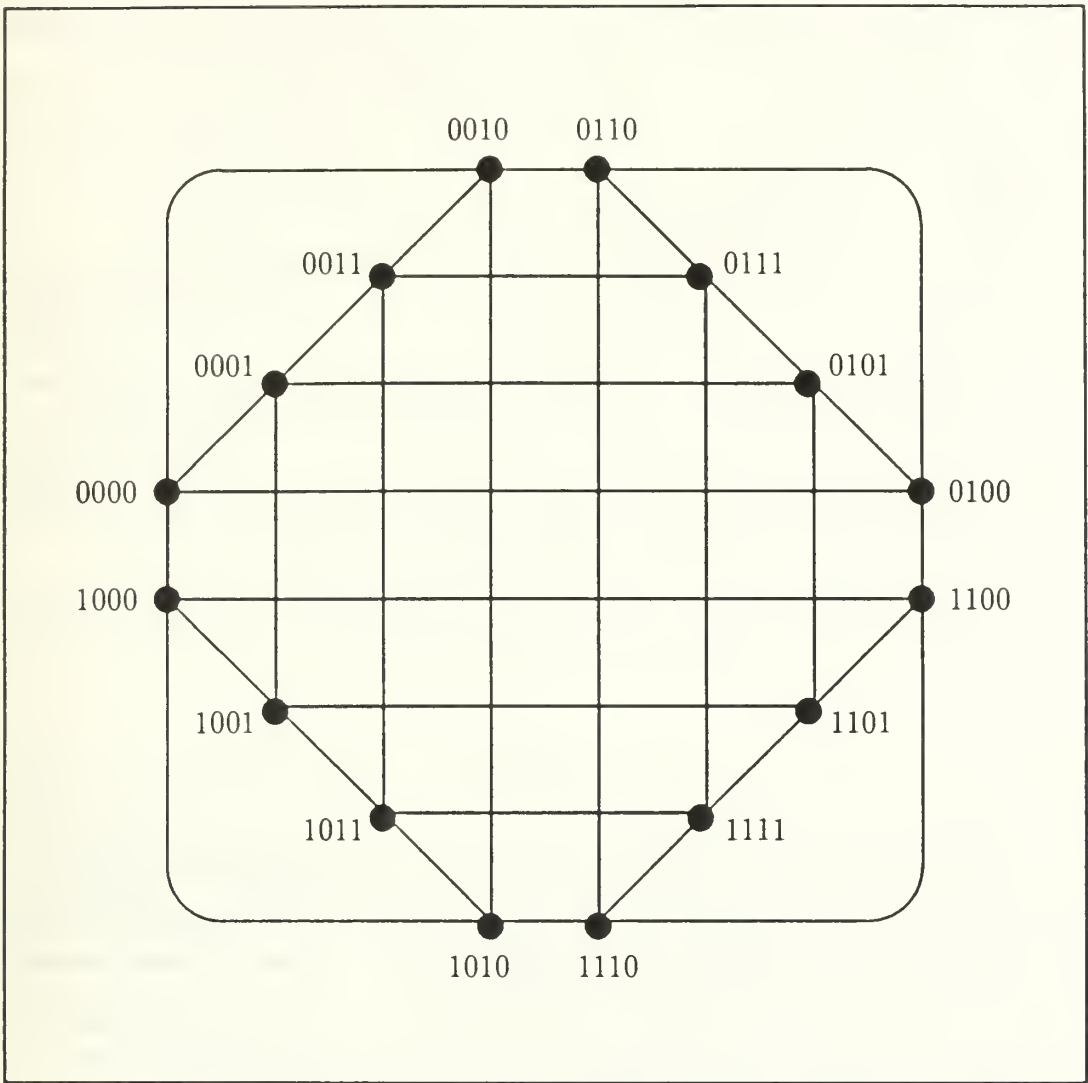
Figure C.4: Graph of a 4–Cube

H. SOURCE CODE LISTINGS

A listing of the Gray code generation program **gray.c** follows.

```
1  /* ----------------=========        PROGRAM INFORMATION     =========-----------------
2  *
3  *  SOURCE       :   gray.c
4  *  VERSION      :   1.2
5  *  DATE         :   01 August 1991
6  *  AUTHOR       :   Jon Hartman, U. S. Naval Postgraduate School
7  *  USAGE        :   gray
8  *  REFERENCES   :
9  *
10 *     [1] Hamming, Richard W.  "Coding and Information Theory", 2nd edition,
11 *         edition, Englewood Cliffs, N.J.: Prentice-Hall, 1986, pp. 97-99.
12 *
13 *
14 *  -------------===============       DESCRIPTION     ===============-------------
15 *
16 *     This program generates and displays the Gray code described in [1].
17 *
18 *  ----------------================================================================-------------
19 */
20
21
22 /* ----------------=================        ALGORITHM      ===============-------------
23 *
24 *  Consider a b-bit Gray code beginning at zero.  Let j be an integral index
25 *  such that 0 <= j < b.  Consider two b-vectors, mod_counter[] and bin[].
26 *  Each element, mod_counter[j], holds a count mod (2^(j+1)).  Initially we
27 *  shall set mod_counter[j] = (2^j).  Furthermore, let the elements of bin[]
28 *  represent a binary number in the natural way.     That is, each element,
29 *  bin[j] will be either 0 or 1,  and bin[] will be formed so that the sum,
30 *  ( 2^0 * bin[0] + 2^1 * bin[1] + 2^2 * bin[2] + ... ), represents the
31 *  'value' of bin[].  We have elected to start the code at zero, so let
32 *  bin[] be set to zeros initially.  Next perform this algorithm:
33 *
34 *  for (i = 0; i < (2^b); i++) {
35 *
36 *      Print the "binary number" represented by bin[].
37 *
38 *      for (j = 0; j < b; j++) {
39 *
40 *          Let mod_counter[j] = (mod_counter[j] + 1) mod (2^(j+1))
41 *
42 *          If mod_counter[j] == 0, then toggle the bit in bin[j]
43 *          (i.e., bin[j] = (bin[j] XOR 1) ).
44 *
45 *      } end for(j)
46 *
47 *  } end for(i)
48 *
49 *  ----------------================================================================-------------
50 */
```

152

```
51
52
53 #include <stdio.h>
54
55
56
57 #ifndef EXIT_FAILURE
58 #define EXIT_FAILURE    1
59 #endif
60
61
62 #ifndef SUCCESS
63 #define SUCCESS         0
64 #endif
65
66
67 #define POW2(n)         ((1) << (n))
68
69
70
71
72
73 main() {
74
75     int  patience = 5;      /*  there's a limit to my patience!          */
76
77     long b = 0,             /*  as in b-bit Gray code                    */
78          *bin,              /*  as described above                       */
79          i,                 /*  generic integral values                 */
80          j,
81          l,                 /*  length of Gray code (2^b)                */
82          *mod_counter;      /*  as described above                      */
83
84
85     printf("\n\n\n\n\n\n----====   ");
86     printf("This program generates the binary numbers of a Gray code.   ");
87     printf("====----\n\n\n");
88
89     printf("     Successive numbers in a Gray code differ in exactly ");
90     printf("one bit position.\n");
91
92     printf("     The list generated by this program will be complete.   ");
93     printf("That is, if you\n");
94
95     printf("     request the code of numbers that are b-bits long, ");
96     printf("you will get a list\n");
97
98     printf("     of (2^b) binary numbers, starting with zero.\n\n\n");
99
100
```

153

```
101     /*  The sole purpose of this while() loop is to get the value of b  */
102     while (b <= 0) {
103
104         printf("     Please enter desired length (binary digits): ");
105         scanf("%d", &b);
106         fflush(stdin);
107         printf("\n\n");
108
109         if (b > 0) {      /*  else ask again (patience permitting)  */
110
111             l = POW2(b);
112
113             if (l <= 0) { /*  guard against too many left shifts!  */
114
115                 printf("     The acceptable range is ");
116                 printf("1..%d.  ", (sizeof(long)*8-2));
117                 printf("Please try again.\n\n\n");
118
119                 b = -1;
120             }
121         }
122
123         if (--patience <= 0) {
124
125             printf("     Ran out of patience!\n");
126             exit(EXIT_FAILURE);
127         }
128     } /* end while (b <= 0) */
129
130
131     /*  Allocate storage for the arrays, test to see if it worked  */
132     bin         = (long*) calloc (b, sizeof(long));
133     mod_counter = (long*) calloc (b, sizeof(long));
134
135     if ((!bin) || (!mod_counter)) {
136
137         printf("main(): Allocation failure bin[] or mod_counter[].\n");
138         exit(EXIT_FAILURE);
139     }
140
141
142     /*  Initialize mod_counter[]  */
143     for (i = 0; i < b; i++)  mod_counter[i] = POW2(i);
144
145     printf("     Gray code for %ld bits will generate ", b);
146     printf("%ld numbers.\n\n\n", l);
147     printf("     Press RETURN to continue....");
148     fflush(stdin);
149     i = getc(stdin);
150     printf("\n\n\n");
```

154

```
151      /*  Do the for() loop spoken of in the "ALGORITHM" section above   */
152
153      for (i = 0;  i < 1;  i++) {
154
155          /*  Print the binary representation held in bin[]   */
156          printf("\t");
157
158          for (j = (b-1);  j >= 0;  j--) { printf("%ld", bin[j]); }
159
160          printf("\n");
161
162
163          /*   Adjust the counters using addition mod (2^(j+1)) and toggle the
164           *   corresponding bit in bin[] whenever an element of mod_counter[]
165           *   reaches zero.
166           */
167          for (j = 0;  j < b;  j++) {
168
169              mod_counter[j]++;
170
171              if ((mod_counter[j] %= POW2(j+1)) == 0)  bin[j] ^= 1;
172          }
173      } /* end for(i) */
174
175      free(bin);
176      free(mod_counter);
177
178      return(SUCCESS);
179 }
180 /* --------------------============    EOF  gray.c   ============------------- */
```

APPENDIX D

A SPARSE MATRIX

Partial differential equations can be used to characterize many physical problems. Explicit solutions to these problems are often quite complicated, so alternative approaches warrant our attention. Simple matrices exist as legitimate representatives of complex problems. A system of linear equations can be constructed to give a discrete approximation to the problem. The structure of the physical setting guarantees that the corresponding matrix of coefficients will be sparse and symmetric. Why does this happen? When do we have the right to expect such a simple matrix? Where does the matrix come from and what does it mean?

This discussion explains how to construct the matrix of coefficients and vectors that describe the numerical approximation to an elliptic partial differential equation. Poisson's equation in two dimensions is used to demonstrate the process. The first step uses a finite difference approximation to produce a system of equations. The system is fine-tuned and the matrix of coefficients is extracted. The process reveals the origins of **structure** and shows why the matrix is sparse and symmetric.

A. LAPLACE AND POISSON

To most engineers, mathematicians, and scientists, Laplace and Poisson are familiar French names. Pierre–Simon de Laplace (1749–1827) and Siméon Denis Poisson (1781–1840) made sizeable contributions to several fields. In a moment, the discussion turns to partial differential equations named in honor of these gentlemen.

If the material seems a bit difficult, the following quote from [Ref. 42: p. 10] may provide some encouragement. The ideas are not so obvious to everyone as they may have been to Laplace.

> *Nathaniel Bowditch (1773–1838), an American astronomer and mathematician, while translating Laplace's Mécanique céleste in the early 1800s, stated, "I never come across one of Laplace's 'Thus it plainly appears' without feeling sure that I have hours of hard work before me to fill up the chasm and find out and show how it plainly appears."*

The next several pages are dedicated to showing *how* the matrix representation of a partial differential equation *plainly appears!* The objective is to describe a particular physical problem, then convert it to the equivalent matrix representation using a deliberate, step–by–step approach.

B. EQUATIONS

Laplace and Poisson worked with partial differential equations that can be observed in nature. What kinds of natural phenomena can be described with partial differential equations? This section gives a brief answer to this question. The discussion includes the natural setting, the equations, and a quick look at the variables and constants involved. The link between the equations and their physical meaning is critical, so this aspect must be developed. The *heat equation* has one of the most intuitive physical interpretations available, so it is used as a starting point. After developing a general perspective, the field can be narrowed to a particular example— Poisson's equation. Such a limited survey of partial differential equations can only hope to succeed by appealling to the reader's experience and intuition.

1. Heat

Before looking at a partial differential equation, let us recall some plane geometry. The intersection of a plane and a cone(s) provides many interesting shapes and equations. Consider the equation that describes all points equidistant from a point (focus) and a line (directrix):

$$y = \left(\frac{1}{4c}\right)x^2 + k. \tag{D.1}$$

This is a parabola whose focus and vertex both lie on the y–axis (the axis of the parabola is the y–axis). The focal length is c and the vertex is located at $(0, k)$.

Partial differential equations are classified using conic sections much like equations in the xy-plane. Introductions to partial differential equations often begin with the **heat equation**:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} + Q(x,t). \tag{D.2}$$

This is an example of a *parabolic* partial differential equation. Note the similarity of equations (D.1) and (D.2).

a. Definitions and Notation

The heat equation describes the temperature, $u(x,t)$, in a "thin rod" (the single dimension x appears in the equation). The presence of t indicates dependence upon time. If there is a heat source (or sink) present, it is represented by Q. We can see that Q may be a function of x or t or both. When mass density (ρ), specific heat (s), and thermal conductivity (K) are known; the thermal diffusivity, κ, can be determined using the following relation:

$$\kappa = \frac{K}{s\rho} \tag{D.3}$$

b. Houses and Heat

From our youth, we have observed several important properties of heat flow. The lessons are simple, few in number, and can be observed from the comfort of our home. First, heat energy only flows when there is a difference in temperature. If the temperature outside is the same as the indoor temperature, no heat energy will cross the threshhold (even with the door open). A temperature difference represents an instability and heat will flow to counter this situation.

When heat does flow, it goes from hotter to colder regions. The loss of heat energy from the warmer region reduces the temperature there, and the temperature in the colder region rises as it gains heat energy. The transfer of heat

has a stabilizing effect (the environment will not be at rest as long as temperature differences exist). We do not find the changes in temperature surprising, but our conversation indicates confusion concerning the *direction* of the flow. Most of us have heard someone say: "Close the door, you're letting cold air in!". We understand that this statement is not correct, but it seems to persist from one generation to the next.

In addition to the idea that heat flows in the presence of temperature differences (gradients), we clearly understand that *larger* differences are related to *greater heat flow*. On a very cold Winter day, the parent notices more quickly that the child left the door open (and displays more urgency in shutting it). In other words, the effect of heat flow is to balance differences in temperature and it somehow "works harder" when there is a greater difference to balance. In mathematical terms, we would suspect (correctly) that heat flow is *proportional* to temperature difference.

Finally, we recognize an ability to restrict heat's ever–present balancing efforts. Sometimes we *want an imbalance* in temperature, and we often use insulation to maintain this imbalance. When we shut the door, we expect that it will slow the transfer of thermal energy through the doorway and enable us to maintain an acceptable imbalance in temperature. For the same reason we use special materials in the construction of refrigerators to keep heat out, and in ovens to keep heat energy inside. This means that the effectiveness of heat transfer is subject to properties of the medium (air, glass windows, fiberglass insulation, wood doors, steel, styrofoam, and so on) through which it flows.

c. Heat Flux

The right–hand side of the heat equation looks a bit complex, but it merely captures this idea of heat flow. Before tackling the *second* partial derivative of u with respect to x, think about the first partial derivative. The first partial derivative of u with respect to x (scaled by the thermal conductivity, K) describes

159

movement of thermal energy. This flow of heat is usually called heat *flux*, denoted ϕ, and can be calculated using **Fourier's law of heat conduction**:

$$\phi = -K\frac{\partial u}{\partial x} \tag{D.4}$$

Heat flux is a measure of how much thermal energy per unit time is moving to the right per unit surface area (by convention, flow to the left is assigned a negative value and flow to the right is positive) [Ref. 43: p. 3]. The second partial derivative measures changes in flux with respect to position. In other words, it represents increasing or decreasing flux.

d. Heat Equation Summary

Let us carefully reassemble the pieces of the heat equation (D.2) to see if the theory agrees with experience. Temperature has spatial and temporal dependencies. The left–hand side describes changes in temperature over time. Changes in heat flux are captured in the second partial of u that appears on the right–hand side. Flux, heat energy in motion, acts to equalize temperature. The thermal diffusivity, κ, measures the material's resistance to heat flux. That is, a temperature difference activates the flow of heat but the speed and effectiveness of this flow is moderated by material properties. Considering everything, then, the heat equation can be stated in one (long) sentence: Changes in temperature over time are caused by (equal to, due to, related to) changes in heat flow (moderated or accelerated by properties of the material) and thermal source(s).

2. Notation

With two or more dimensions, the same equations that looked simple in one dimension can begin to look complex. The linear operator, Δ, is used to simplify the notation. For example, Δu, substituted into the right–hand side of (D.2), gives

the heat equation a new look:

$$\frac{\partial u}{\partial t} = \kappa \Delta u + Q(x, t) \tag{D.5}$$

This is a more general equation since the linear operator Δu can be applied in any number of dimensions. For instance (in three dimensions),

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \tag{D.6}$$

Sometimes this operator is called the *Laplacian* of u and some authors use the *del* operator, ∇, in these equations ($\nabla^2 u \equiv \Delta u$).

3. Diffusion

The behavior of thermal energy is actually a special instance of *diffusion*, so (D.5) is often referred to as the **diffusion equation**. With an appropriate substitution for κ, the equation might describe the spreading of dye through ocean water. In an agricultural application, it could characterize water or chemical penetration in soil. We shall continue to use the term "heat equation", though, for the sake of consistent terminology and notation.

4. Laplace's Equation

Consider the effect of a few restrictions on the heat equation. Suppose that there is no source of thermal energy ($Q = 0$) and the physical properties of the material do not vary (κ is constant). Finally, what happens if the time–dependency is removed?

The left–hand side of the equation goes away. This is not so unrealistic. Systems may reach a steady (equilibrium) state after a time (especially in the absence of sources). We can divide through by κ (assuming $\kappa \neq 0$) and the equation becomes:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{D.7}$$

This is **Laplace's equation** in the two dimensions x and y. Sometimes it is called the **potential equation** since it also describes the cases in which u stands for gravity or voltage. It can also describe "steady–state heat flow... hydrodynamics, gravitational attraction, elasticity, and certain motions of incompressible fluids". [Ref. 44: pp. 660-661]

5. Ellipses

Although Laplace's equation seems like a steady–state heat equation, it is fundamentally different. It falls in the *elliptic* class of partial differential equations. Consider an ellipse centered at the origin with foci (on the x–axis at a distance of c from the origin) located at $(-c, 0)$ and $(c, 0)$. Suppose that the foci are labeled F_1 and F_2. The major axis passes through the center and through the foci, connecting two vertices positioned at $(-a, 0)$ and $(a, 0)$. The minor axis passes through the center perpendicular to the major axis and connects the vertices at $(0, -b)$ and $(0, b)$. The major axis deserves its name since $a > b$ (in the case of equality the ellipse degenerates and we get a special case—the circle).

For any arbitrary point, p, let the distance d_1 be the distance from p to F_1 and let d_2 be the distance from p to F_2. Furthermore, let $d = d_1 + d_2$. The ellipse is described by all points satisfying $d = 2a$, where a is the constant length of the ellipse's semi–major axis as described above. The standard form for the equation of this ellipse is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{D.8}$$

Using the distances from this ellipse, a right triangle can be formed with sides of length b and c and hypotenuse of length a. This means a, b, and c are related by the Pythagorean Theorem.
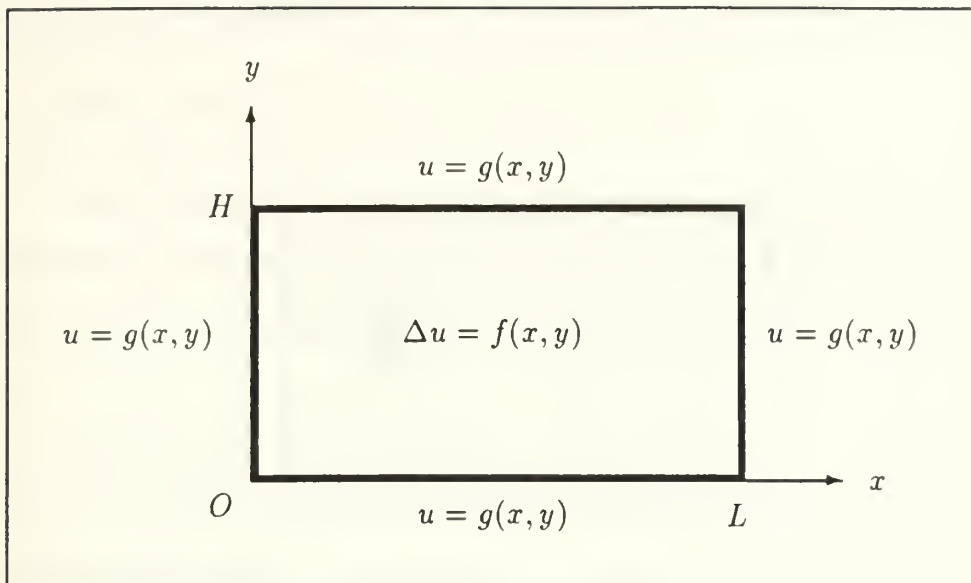
Figure D.1: The Region

6. Poisson's Equation

We have discussed several partial differential equations and observed the impact of changing a few parameters. Laplace's equation showed what happens in the steady–state case when sources are removed and the thermal diffusivity is non–zero. Now we return to the more general problem that can be represented in the presence of a source, sometimes called a *driving* (or *forcing*) function, say $f(x, y)$.

The result is **Poisson's equation** (shown here in two dimensions):

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \qquad (D.9)$$

Again, $u(x, y)$ typically represents temperature or voltage. Laplace's equation (D.7) is just the special case of Poisson's equation (D.9) where $f(x, y) = 0$. The rest of the discussion will focus on Poisson's equation within the rectangular region (shown in Figure D.1): $0 \leq x \leq L, 0 \leq y \leq H$.
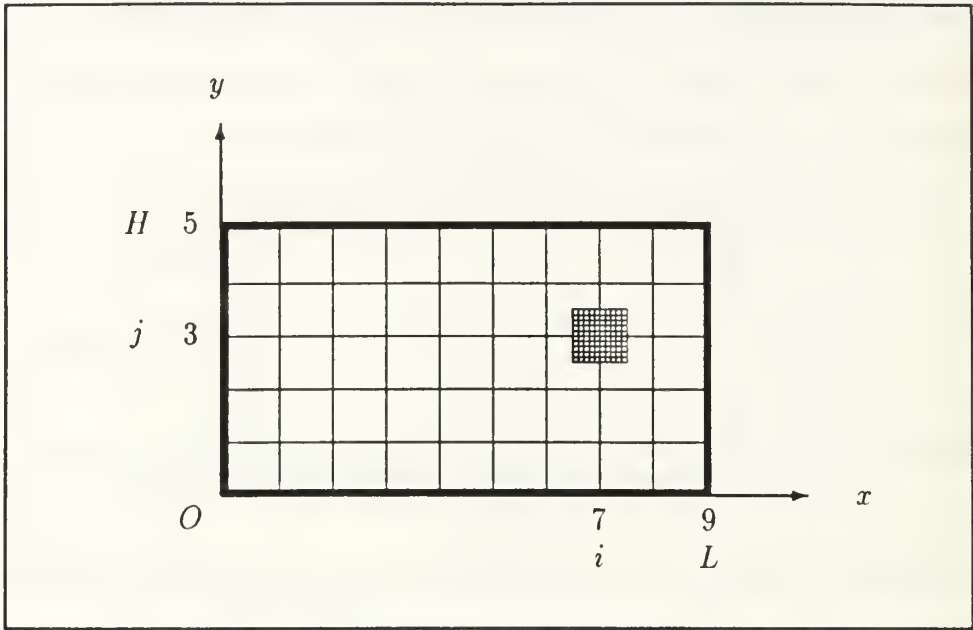
Figure D.2: Subdividing the Rectangle

7. Final Assumptions

We shall assume that the conditions along the boundaries are known and are given by $u = g(x, y)$. The problem is solved in the presence of a forcing function f. The goal is to produce something that a computing machine can "solve". To reach this position, several steps are required. First, the domain is divided into many smaller regions. Using this subdivision scheme, a system of equations is developed. The information that is known (f and g) can be moved to the right–hand side of the system. The system can then be represented in typical $Ax = b$ fashion.

C. DISCRETIZATION

Before attempting a numerical solution, the domain must be subdivided into a finite (but probably large) number of elements. Figure D.2 provides an illustration of what this mesh looks like. We should not forget that actual applications may involve 100 (or more) divisions in each direction. Nevertheless, (artificially) small

examples are quite sufficient for conveying notation and measures within the region.

1. Notation

A clear understanding of the problem domain, conventions, and notation is prerequisite to developing the system of equations. Consider Figure D.2. This domain will serve as a reference for the upcoming discussion on conventions and notation.

The rectangular region has length $L = 9$ and height $H = 5$. It has been subdivided into 45 smaller elements by a mesh made of four horizontal lines and eight vertical lines. The integers m and n are used to keep track of how many horizontal and vertical dividing lines are used (here $m = 4$ and $n = 8$). Each element has length h (in the x–direction) and height k (in the y–direction). In this particular example, the elements are (conveniently) square with $h = k = 1$. In general, the individual elements within the region are rectangular (it is not necessarily true that $h = k$).

The elements within the region are uniformly spaced (each has the same size). L, H, h, and k do not need to be integers—they can be any convenient units. To guarantee uniform spacing, of course, L and H must be integer multiples of h and k, respectively. That is:

$$L = (n + 1)h, \quad n \in \{0, 1, 2, 3, \dots\}$$
$$H = (m + 1)k, \quad m \in \{0, 1, 2, 3, \dots\}$$

2. Internal Mesh Points

Our goal is a system of equations, and ultimately a problem stated in terms of a matrix and vectors. We will eventually see that there are mn equations in mn unknowns, one for each *internal mesh point* (where the lines cross). Imagine elements of size $h \times k$ (as before) that are centered on these points, such as the cross–hatched

165

element at $(7, 3)$. Each equation in the system will correspond to one of these line-crossings and represent one of these elements. It is useful to label the lines for reference purposes. To accomplish this, we use the (integer) counters i and j.

These counters are used to reference particular vertical and horizontal dividing lines. The i counter refers to a vertical line $(1 \le i \le n)$ and the horizontal lines are indexed by j $(1 \le j \le m)$. Figure D.2 may be deceptively simple due to the element dimensions $h = k = 1$. Because of this, $i = 7$ indicates an x–coordinate of 7 and $j = 3$ means $y = 3$. But the counters i and j are not generally equivalent to x– and y–position in the coordinate system. Given h, k, i, and j the corresponding coordinates are $(x, y) = (ih, jk)$.

D. A SYSTEM OF EQUATIONS

The next step is to build a system of mn equations that describes the problem. First, we need to agree upon a referencing scheme for the internal mesh points. The numbering will be based upon i and j as defined above. This numbering scheme begins at the bottom left (i.e., $i = j = 1$), proceeds up the first column and then moves, column–by–column, to the right. Specifically, the points will be assigned a *label*

$$\ell = m(i - 1) + j \qquad (D.10)$$

Given the values i and j for any internal point, now we can assign it a label $(1 \le \ell \le mn)$. Figure D.3 shows values of i along the x–axis, values of j along the y–axis, and labeling of internal mesh points according to (D.10).

1. Finite Differences

The approach calls for analyzing each internal mesh point. Figure D.4 shows the point referenced by i and j and its neighbors to the North, South, East,

166
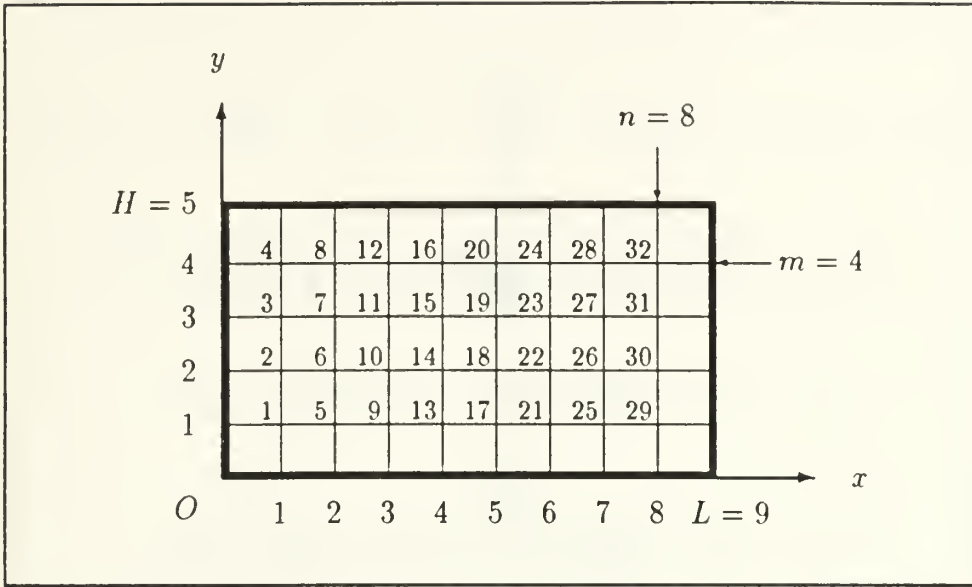
Figure D.3: Numbering the Equations

and West. We use a centered finite difference method to approximate the partial derivatives in (D.9) and arrive at the equations for these points. The finite difference approximations for the partial derivatives are:

$$\frac{\partial^2 u}{\partial x^2}_{(i,j)} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} \tag{D.11}$$

$$\frac{\partial^2 u}{\partial y^2}_{(i,j)} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{k^2} \tag{D.12}$$

The approximation for the partial derivative in the x–direction (D.11) considers the neighbor to the West, the point itself, and the neighbor to the East. Similarly, the approximation in the y–direction (D.12) recognizes neighbors to the South and North in addition to the point. Both finite difference approximations favor the center point (i, j), giving it twice the weight of its neighbors.

Substituting these into Poisson's equation (D.9) yields:

$$-\left(\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}\right) - \left(\frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{k^2}\right) \approx -\Delta u_{i,j} = -f_{i,j} \tag{D.13}$$

167
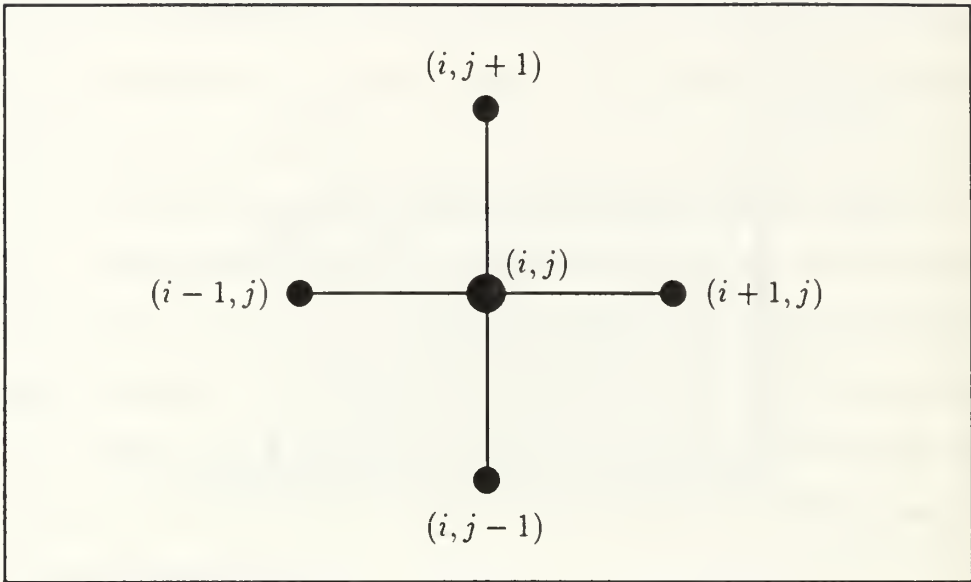
Figure D.4: Neighbors to the North, South, East, and West

The forcing function, $f_{i,j}$, is known so (D.13) begins to look like one of many equations in a linear system. There is such an equation for every internal mesh point. To make sure that we consider all of the internal mesh points in an orderly fashion, we may number them as in Figure D.3 and consider them one at a time.

2. More Equations

At this point, we know the general form (D.13) for each of the equations that must be considered. The matrix of coefficients may not be completely clear yet, so let us consider each of the equations in the order of their labels. For now, we will leave the i, j subscripts on everything:

$$-\left(\frac{u_{0,1} - 2u_{1,1} + u_{2,1}}{h^2}\right) - \left(\frac{u_{1,0} - 2u_{1,1} + u_{1,2}}{k^2}\right) \approx -f_{1,1}$$

$$-\left(\frac{u_{0,2} - 2u_{1,2} + u_{2,2}}{h^2}\right) - \left(\frac{u_{1,1} - 2u_{1,2} + u_{1,3}}{k^2}\right) \approx -f_{1,2}$$

$$\vdots$$

168

$$-\left(\frac{u_{0,m-1} - 2u_{1,m-1} + u_{2,m-1}}{h^2}\right) - \left(\frac{u_{1,m-2} - 2u_{1,m-1} + u_{1,m}}{k^2}\right) \approx -f_{1,m-1}$$

$$-\left(\frac{u_{0,m} - 2u_{1,m} + u_{2,m}}{h^2}\right) - \left(\frac{u_{1,m-1} - 2u_{1,m} + u_{1,m+1}}{k^2}\right) \approx -f_{1,m}$$

$$-\left(\frac{u_{1,1} - 2u_{2,1} + u_{3,1}}{h^2}\right) - \left(\frac{u_{2,0} - 2u_{2,1} + u_{2,2}}{k^2}\right) \approx -f_{2,1}$$

$$-\left(\frac{u_{1,2} - 2u_{2,2} + u_{3,2}}{h^2}\right) - \left(\frac{u_{2,1} - 2u_{2,2} + u_{2,3}}{k^2}\right) \approx -f_{2,2}$$

$$\vdots$$

$$-\left(\frac{u_{1,m-1} - 2u_{2,m-1} + u_{3,m-1}}{h^2}\right) - \left(\frac{u_{2,m-2} - 2u_{2,m-1} + u_{2,m}}{k^2}\right) \approx -f_{2,m-1}$$

$$-\left(\frac{u_{1,m} - 2u_{2,m} + u_{3,m}}{h^2}\right) - \left(\frac{u_{2,m-1} - 2u_{2,m} + u_{2,m+1}}{k^2}\right) \approx -f_{2,m}$$

$$\vdots$$

$$-\left(\frac{u_{n-2,1} - 2u_{n-1,1} + u_{n,1}}{h^2}\right) - \left(\frac{u_{n-1,0} - 2u_{n-1,1} + u_{n-1,2}}{k^2}\right) \approx -f_{n-1,1}$$

$$-\left(\frac{u_{n-2,2} - 2u_{n-1,2} + u_{n,2}}{h^2}\right) - \left(\frac{u_{n-1,1} - 2u_{n-1,2} + u_{n-1,3}}{k^2}\right) \approx -f_{n-1,2}$$

$$\vdots$$

$$-\left(\frac{u_{n-2,m-1} - 2u_{n-1,m-1} + u_{n,m-1}}{h^2}\right) - \left(\frac{u_{n-1,m-2} - 2u_{n-1,m-1} + u_{n-1,m}}{k^2}\right) \approx -f_{n-1,m-1}$$

$$-\left(\frac{u_{n-2,m} - 2u_{n-1,m} + u_{n,m}}{h^2}\right) - \left(\frac{u_{n-1,m-1} - 2u_{n-1,m} + u_{n-1,m+1}}{k^2}\right) \approx -f_{n-1,m}$$

$$-\left(\frac{u_{n-1,1} - 2u_{n,1} + u_{n+1,1}}{h^2}\right) - \left(\frac{u_{n,0} - 2u_{n,1} + u_{n,2}}{k^2}\right) \approx -f_{n,1}$$

$$-\left(\frac{u_{n-1,2} - 2u_{n,2} + u_{n+1,2}}{h^2}\right) - \left(\frac{u_{n,1} - 2u_{n,2} + u_{n,3}}{k^2}\right) \approx -f_{n,2}$$

$$\vdots$$

$$-\left(\frac{u_{n-1,m-1} - 2u_{n,m-1} + u_{n+1,m-1}}{h^2}\right) - \left(\frac{u_{n,m-2} - 2u_{n,m-1} + u_{n,m}}{k^2}\right) \approx -f_{n,m-1}$$

$$-\left(\frac{u_{n-1,m} - 2u_{n,m} + u_{n+1,m}}{h^2}\right) - \left(\frac{u_{n,m-1} - 2u_{n,m} + u_{n,m+1}}{k^2}\right) \approx -f_{n,m}$$

3. Modification

The goal is to determine $u_{i,j}$ for all internal points (i, j). Having completed several foundational steps, we can see a developing system of mn equations. Let's clean it up a bit. To do this, we need to make better use of one more piece of the given information—the boundary values. For those points just inside the boundaries (a horizontal distance of h from the sides and/or a vertical distance of k from the top or bottom) we already know part of the left side of (D.13). In particular, any subscript $i = 0$, $j = 0$, $i = n + 1$, and/or $j = m + 1$ signifies a (known) boundary point.

Multiplying through by $(hk)^2$ and moving the known information to the right–hand side of the equations, we again start with the left–most column $(i = 1)$ and work in the order of the labels. Now the system of equations looks like this:

$$k^2(2u_{1,1} - u_{2,1}) + h^2(2u_{1,1} - u_{1,2}) \approx -(hk)^2 f_{1,1} + k^2 u_{0,1} + h^2 u_{1,0}$$

$$k^2(2u_{1,2} - u_{2,2}) + h^2(-u_{1,1} + 2u_{1,2} - u_{1,3}) \approx -(hk)^2 f_{1,2} + k^2 u_{0,2}$$

$$\vdots$$

$$k^2(2u_{1,m-1} - u_{2,m-1}) + h^2(-u_{1,m-2} + 2u_{1,m-1} - u_{1,m}) \approx -(hk)^2 f_{1,m-1} + k^2 u_{0,m-1}$$

$$k^2(2u_{1,m} - u_{2,m}) + h^2(-u_{1,m-1} + 2u_{1,m}) \approx -(hk)^2 f_{1,m} + k^2 u_{0,m} + h^2 u_{1,m+1}$$

$$k^2(-u_{1,1} + 2u_{2,1} - u_{3,1}) + h^2(2u_{2,1} - u_{2,2}) \approx -(hk)^2 f_{2,1} + h^2 u_{2,0}$$

$$k^2(-u_{1,2} + 2u_{2,2} - u_{3,2}) + h^2(-u_{2,1} + 2u_{2,2} - u_{2,3}) \approx -(hk)^2 f_{2,2}$$

$$\vdots$$

$$k^2(-u_{1,m-1} + 2u_{2,m-1} - u_{3,m-1}) + h^2(-u_{2,m-2} + 2u_{2,m-1} - u_{2,m}) \approx -(hk)^2 f_{2,m-1}$$

$$k^2(-u_{1,m} + 2u_{2,m} - u_{3,m}) + h^2(-u_{2,m-1} + 2u_{2,m}) \approx -(hk)^2 f_{2,m} + h^2 u_{2,m+1}$$

$$\vdots$$

$$k^2(-u_{n-2,1} + 2u_{n-1,1} - u_{n,1}) + h^2(2u_{n-1,1} - u_{n-1,2}) \approx -(hk)^2 f_{n-1,1} + h^2 u_{n-1,0}$$

$$k^2(-u_{n-2,2} + 2u_{n-1,2} - u_{n,2}) + h^2(-u_{n-1,1} + 2u_{n-1,2} - u_{n-1,3}) \approx -(hk)^2 f_{n-1,2}$$

$$\vdots$$

$$k^2(-u_{n-2,m-1}+2u_{n-1,m-1}-u_{n,m-1})+h^2(-u_{n-1,m-2}+2u_{n-1,m-1}-u_{n-1,m}) \approx -(hk)^2 f_{n-1,m-1}$$

$$k^2(-u_{n-2,m}+2u_{n-1,m}-u_{n,m})+h^2(-u_{n-1,m-1}+2u_{n-1,m}) \approx -(hk)^2 f_{n-1,m}+h^2 u_{n-1,m+1}$$

$$k^2(-u_{n-1,1}+2u_{n,1})+h^2(2u_{n,1}-u_{n,2}) \approx -(hk)^2 f_{n,1}+k^2 u_{n+1,1}+h^2 u_{n,0}$$

$$k^2(-u_{n-1,2}+2u_{n,2})+h^2(-u_{n,1}+2u_{n,2}-u_{n,3}) \approx -(hk)^2 f_{n,2}+k^2 u_{n+1,2}$$

$$\vdots$$

$$k^2(-u_{n-1,m-1}+2u_{n,m-1})+h^2(-u_{n,m-2}+2u_{n,m-1}-u_{n,m}) \approx -(hk)^2 f_{n,m-1}+k^2 u_{n+1,m-1}$$

$$k^2(-u_{n-1,m}+2u_{n,m})+h^2(-u_{n,m-1}+2u_{n,m}) \approx -(hk)^2 f_{n,m}+k^2 u_{n+1,m}+h^2 u_{n,m+1}$$

Now the equations are very close to what we want. There are some unfortunate side effects to such a deliberate approach. The list of equations is tedious, the subscripts are a bit involved, and it takes some concentration to match things up. There are some benefits, though, for those who can endure! It will take very little effort to see how the coefficients are collected.

E. MATRIX REPRESENTATION

It is not hard to translate the preceding equations into the familiar representation $Ax = b$. Notation is quite important. We will start with the obvious, exchanging u for x so that (eventually) the system will look like $Au = b$. Dimensions are important too. The goal is a large, sparse, symmetrix matrix $A \in \Re^{mn \times mn}$. The vectors u and b have the obvious dimensions and are assumed to contain real numbers as well.

1. Unknowns

Since there is a great deal of structure in this problem, it is useful to partition the vector of unknowns, u. Let $u_{i,j}$ have the same meaning as it did in equation (D.13) and consider the m-vector:

$$u_i = \begin{bmatrix} u_{i,1} \\ u_{i,2} \\ \vdots \\ u_{i,m-1} \\ u_{i,m} \end{bmatrix}$$

This vector captures all of the unknowns for a given column, i, of the original region. Now we can stack the columns, n in number, forming the entire vector u of unknowns:

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix}$$

This process has clearly formed $u \in \Re^{mn}$. Now we turn to the matrix of coefficients.

2. Coefficients

The matrix A is formed by combining two smaller matrices, T and D. First we shall consider the tridiagonal matrix $T \in \Re^{m \times m}$. For aesthetic purposes only, let the diagonal elements of T be $d = 2(h^2 + k^2)$.

$$T = \begin{bmatrix} d & -h^2 & & & & & \\ -h^2 & d & -h^2 & & & & \\ & -h^2 & d & -h^2 & & & \\ & & & \ddots & & & \\ & & & -h^2 & d & -h^2 & \\ & & & & -h^2 & d & -h^2 \\ & & & & & -h^2 & d \end{bmatrix}$$

Next, consider the diagonal matrix $D \in \Re^{m \times m}$:

$$D = \begin{bmatrix} -k^2 & & & & \\ & -k^2 & & & \\ & & \ddots & & \\ & & & -k^2 & \\ & & & & -k^2 \end{bmatrix}$$

Forming the matrix A requires n identical copies of T and $2(n-1)$ identical copies of D. The matrices in A below are assigned subscripts for counting purposes. The matrix subscripts, by the way, denote a value of i corresponding to the partition u_i which the matrix will multiply. A is the block–tridiagonal matrix

$$A = \begin{bmatrix} T_1 & D_2 & & & & & \\ D_1 & T_2 & D_3 & & & & \\ & D_2 & T_3 & D_4 & & & \\ & & & \ddots & & & \\ & & & D_{n-3} & T_{n-2} & D_{n-1} & \\ & & & & D_{n-2} & T_{n-1} & D_n \\ & & & & & D_{n-1} & T_n \end{bmatrix}$$

3. Knowns

We could proceed immediately to the solution vector, $b \in \Re^{mn}$, using the equations provided in the previous section. Again, though, the result can be cleaned up a bit if we form b as the sum of three vectors f, v, w.

The vector $f \in \Re^{mn}$ represents the forcing function. The equations clearly indicate where the scalar multiplier comes from.

$$f = -(hk)^2 \begin{bmatrix} f_{1,1} \\ f_{1,2} \\ \vdots \\ f_{1,m-1} \\ f_{1,m} \\ f_{2,1} \\ f_{2,2} \\ \vdots \\ f_{n,m-1} \\ f_{n,m} \end{bmatrix}$$

Next, the vector $v \in \Re^{mn}$ is used to represent the information that is known due to the boundary values on the East and West sides of the region.

$$
v = k^2 \begin{bmatrix}
u_{0,1} \\
u_{0,2} \\
\vdots \\
u_{0,m-1} \\
u_{0,m} \\
0 \\
\vdots \\
0 \\
u_{n+1,1} \\
u_{n+1,2} \\
\vdots \\
u_{n+1,m-1} \\
u_{n+1,m}
\end{bmatrix}
$$

Finally, the vector $w \in \Re^{mn}$ is used to represent the information that is known due to the boundary values on the North and South sides of the region.

$$
w = h^2 \begin{bmatrix}
u_{1,0} \\
0 \\
\vdots \\
0 \\
u_{1,m+1} \\
u_{2,0} \\
0 \\
\vdots \\
0 \\
u_{2,m+1} \\
u_{3,0} \\
\vdots \\
u_{n,m+1}
\end{bmatrix}
$$

Now b is a simple sum of these vectors: $b = f + v + w$.

F. CONCLUSION

This process has shown a few examples of partial differential equations that appear frequently in nature. Poisson's equation in two dimensions was selected as an example. After the finite difference approximation is selected, determining the system of equations is a tedious (but not too complicated) process. Once the system of equations is written down, the matrix representation is easy to come by.

APPENDIX E

HYPERCUBE COMMUNICATIONS

This report displays the results of point–to–point communications tests that were performed on the Intel iPSC/2 hypercube. The emphasis of the experiment was to evaluate several aspects of communications time. The exercise showed that communication on this machine is virtually independent of the Hamming distance between communicating nodes. There is clear evidence that transmission rates are related to message length (the transmission system favors longer messages) due—at least in part—to an overhead charged to begin the communication. Communications between the *host* and a node never achieve the rate that can be realized with node–to–node transmissions.

The communications test code described in this appendix was only executed on the iPSC/2. Time did not permit modification of the code and testing on the transputer networks. A thorough test of communications and computational abilities of the T414 and T800 transputers has already been performed by Gregory Bryant. His masters thesis [Ref. 26] contains the documentation of this work. A short summary of Bryant's findings is included in the conclusions to this appendix.

A. SOURCE CODE OVERVIEW

The host program (**commtst.c**) and a node program (**commtstn.c**) contain most of the code for this experiment. There is also a header file, **commtst.h**, shared by these codes,. Finally (but perhaps most important for any high–level survey of the code), the makefile **commtst.mak** shows dependencies and compilation procedures.

In the discussion that follows, bold–faced type is used to indicate function and object names that actually appear in the code.

B. STRATEGY

The program must define the *valid arguments*. The function **interpret_args()** takes care of checking for occurrences of these arguments in the command line. When the arguments have been interpreted, we know how to set variables like **reps** (repetitions), **bytes** (length of the message to be passed), and **verbose** (to control how much data is spewed out). Once these values are known, the host instructs each node to either RECEIVE or SEND. A special **Tasking** packet (structure) carries instructions to each node independently. Only one node is designated to SEND at any one time; the rest RECEIVE. Receivers simply **crecv()** the given number of bytes and return the message to the originator by calling **csend()**. Since this involves a round–trip, the issue of timing requires attention.

We can divide the time measurement by two (to account for the round–trip), provided we aren't deceived by the outcome. That is, passing two b–byte messages is *not the same as* passing a single message of length $2b$ bytes. To make the timing data credible, however, the round–trip method is essential. The precision of the **mclock()** function is an additional issue. At best, **mclock()** is accurate to the millisecond (and ten milliseconds may be a more reasonable expectation). Very short messages can produce questionable results in terms of the precision of the timing data.

For this reason, tests of short messages should be repeated a number of times within the block surrounded by time checks. This, of course, revives the same issue (multiple repetitions of a message are not equivalent to a single, longer message). We may proceed, however, provided we establish a common understanding of the problem domain and terminology. I have used the term *effective time* to capture this subtlety.

Wherever this term appears, it should be interpreted according to the following definition:

$$t_e = \frac{t}{2\rho}$$

where t_e is the effective time, t is the actual time measurement for the message, and ρ is the number of repetitions. The factor of two is included to account for the round–trip. For instance, suppose that the user asks for three repetitions of a message. The implementation carries this out in a **for** loop. Time is sampled before and after the loop. The inside of the loop is the simple **csend()** and **crecv()** sequence described earlier. The effective time in this example would be $t_e = t/6$.

In summary, there is no convenient (and credible) method for timing one–way communications. If we time one–way communications, the results could be misleading in that we could not be certain that the clock was starting just before the beginning of the **csend()** and stopped immediately after the receiving node accumulated the final byte of the message. We must also consider the issue of *blocking communication*.[1] Thus, the (round–trip) method is not so easily misled by the fact that **csend()** is not actually *blocking*. The transmission duties are quickly handed over to a communication manager and processing continues directly. The **crecv()** enforces blocking communications and execution stops at this function until the last byte has been acquired. Thus the round–trip method seems to be quite reliable, particularly in the case of node–to–node communications (if the host is involved, the results are less consistent).

Since receiver nodes have nothing else to do but receive and retransmit the message, the performance loss due to the round–trip method should be (almost entirely) accounted for by two factors (loosely) placed into "software" and "hardware"

---

[1]By definition, *blocking* means that the invoking process (send or receive) causes execution of the program to stop (be blocked from the CPU) until the communications requirement has been satisfied.

179

categories:

- *Software overheads* like establishing and freeing the activation stack for functions (e.g., the **csend()** and **crecv()** functions).

- *Hardware overheads* associated with establishing the communication path and performing switching. The take–down time for this task is probably negligible.

Hence, if this method of analyzing communications performance errs, it does so on the *conservative* side. That is, the timing used in this method is liberal (if anything), so that communication rates will be estimated conservatively.

C. RESULTS

Considering the nature of the implementation, communications will be considered bidirectional. In particular, the term "host–to–node" communications does not imply that the host is the originator of directed communication, but that a bidirectional exchange takes place between some node and the host. The host *does* send directed, one–way instructions to the nodes, but all *timed* communication originates at a node and returns to that node (even if it goes to the host). There are essentially three groups of results; each of which captures data for *node–to–node* communications and *host–to–node* communications.

1. Small Messages Repeated Ten Times

The first test involved messages of length $\ell \leq 1,024$ bytes. Since the shortest of these would not generate trustworthy timing data, the repetition count, ρ, was set at ten. This gave $t_e = t/20$. Table E.1 shows the results.

TABLE E.1: SHORT MESSAGES WITH TEN REPETITIONS

Message Length (Bytes)	Node-to-Node			Host-to-Node		
	t (msec)	t_e (msec)	Rate (kbytes/sec)	t (msec)	t_e (msec)	Rate (kbytes/sec)
1	7.10	0.36	2.75	71.40	3.57	0.27
2	7.00	0.35	5.58	79.40	3.97	0.49
4	7.00	0.35	11.16	78.90	3.95	0.99
8	7.00	0.35	22.32	75.80	3.79	2.06
16	7.20	0.36	43.40	78.10	3.91	4.00
32	7.30	0.37	85.62	79.40	3.97	7.87
64	7.70	0.39	162.34	87.10	4.36	14.35
128	13.90	0.70	179.86	132.10	6.61	18.93
192	14.30	0.72	262.24	134.60	6.73	27.86
256	14.70	0.74	340.14	137.50	6.88	36.36
320	15.30	0.77	408.50	139.60	6.98	44.77
384	15.80	0.79	474.68	142.40	7.12	52.67
448	16.20	0.81	540.12	147.40	7.37	59.36
512	16.70	0.84	598.80	180.30	9.02	55.46
576	17.10	0.86	657.89	201.50	10.08	55.83
640	17.60	0.88	710.23	207.00	10.35	60.39
704	18.10	0.91	759.67	208.80	10.44	65.85
768	18.50	0.93	810.81	204.50	10.23	73.35
832	19.00	0.95	855.26	180.00	9.00	90.28
896	19.40	0.97	902.06	152.30	7.62	114.90
960	19.90	0.99	942.21	147.80	7.39	126.86
1024	20.40	1.02	980.39	148.90	7.45	134.32
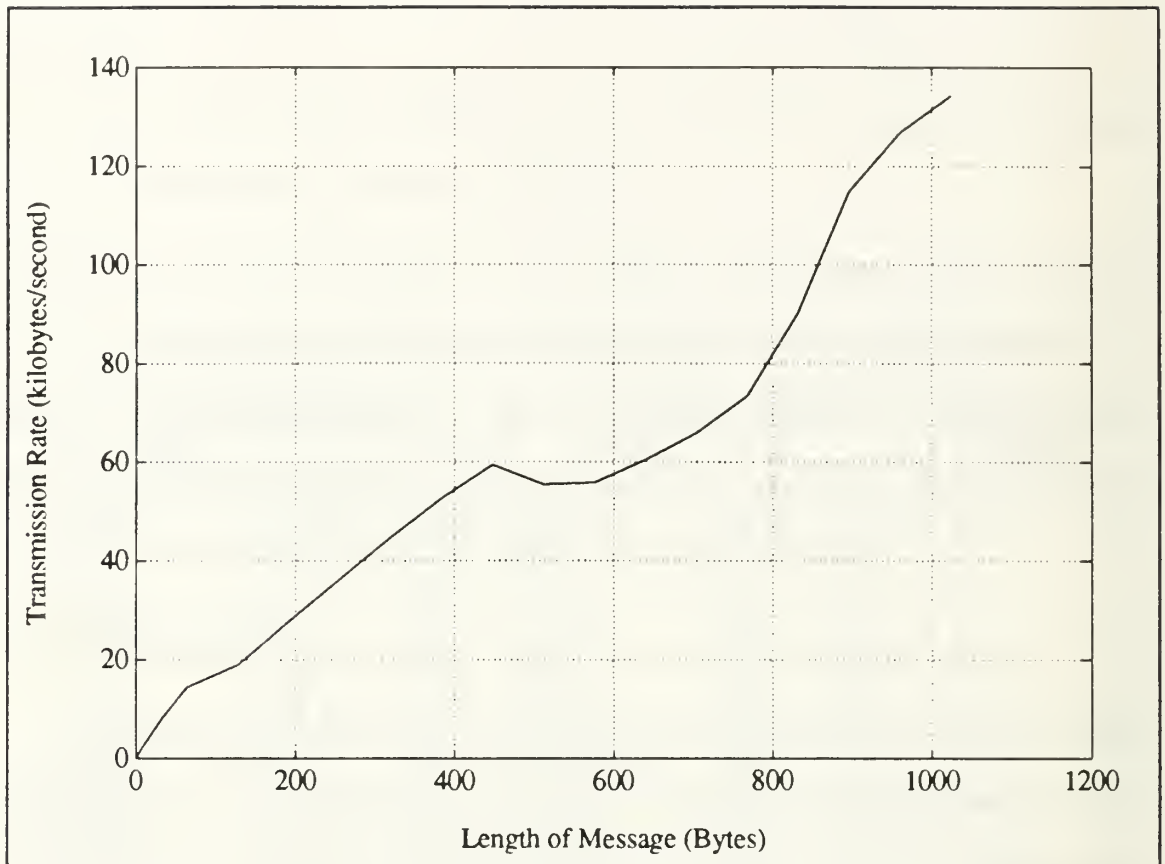
Figure E.1: Speed of Small Host–Node Messages (Ten Repetitions)

a. *Host–to–Node Performance*

The communication rates for small host–node messages with a repetition count of ten are illustrated in Figure E.1. Communications involving the host produce very irregular results (in the sense that the relationship between length and performance is not straightforward). The experiment was executed when only one user was logged in at the host and the results followed the same general pattern on repeated tests.
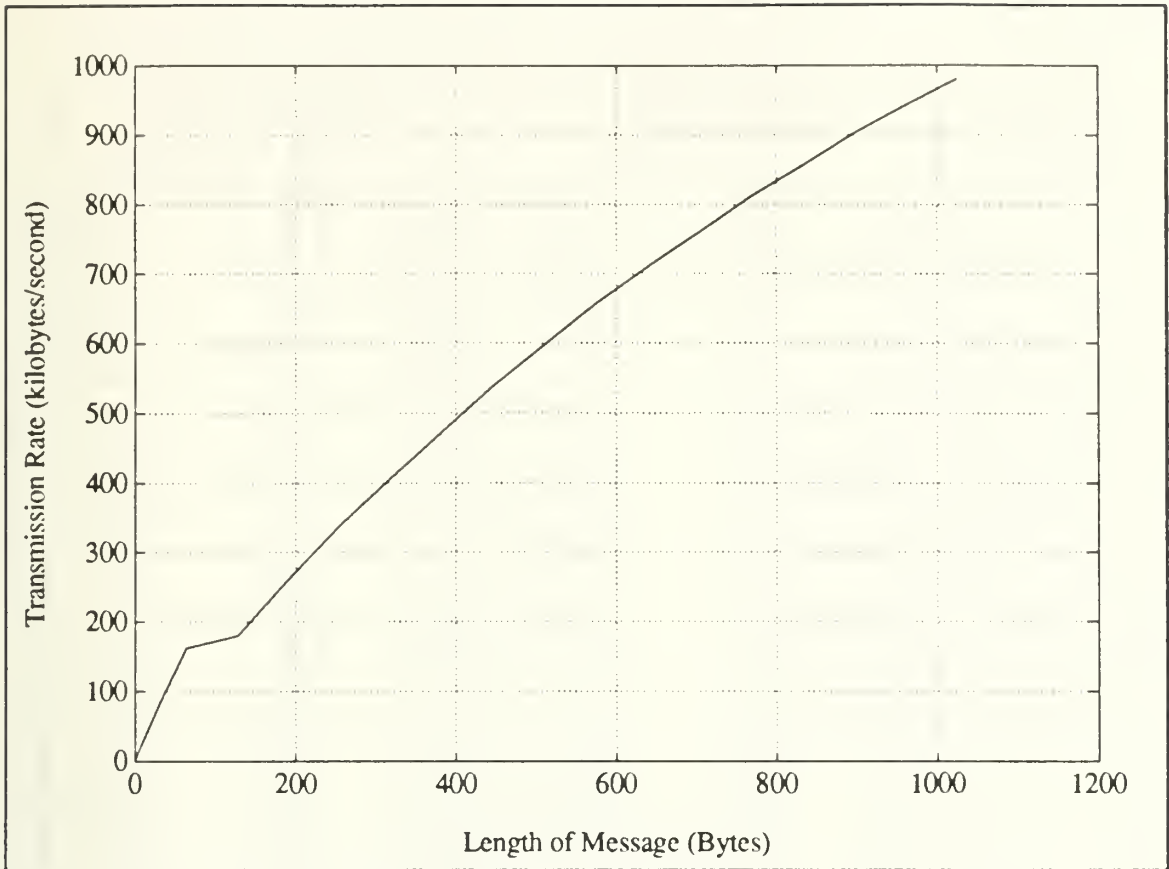
Figure E.2: Speed of Small Messages Between Nodes (Ten Repetitions)

b. Node–to–Node Performance

In the absence of contention for the communication medium, node–to–node communications within the cube are quite predictable. Figure E.2 shows transmission rates for small messages (up to one kilobyte) repeated ten times.

TABLE E.2: SHORT MESSAGES WITH ONE HUNDRED REPETITIONS

Message Length (Bytes)	Node–to–Node			Host–to–Node		
	t (msec)	t_e (msec)	Rate (kbytes/sec)	t (msec)	t_e (msec)	Rate (kbytes/sec)
1	68.60	0.34	2.85	837.40	4.19	0.23
2	68.60	0.34	5.69	818.30	4.09	0.48
4	68.70	0.34	11.37	795.00	3.98	0.98
8	69.40	0.35	22.51	774.50	3.87	2.02
16	70.30	0.35	44.45	758.30	3.79	4.12
32	71.70	0.36	87.17	737.10	3.69	8.48
64	75.30	0.38	166.00	721.30	3.61	17.33
128	137.60	0.69	181.69	1020.10	5.10	24.51
192	142.30	0.71	263.53	1007.10	5.04	37.24
256	146.80	0.73	340.60	1007.00	5.04	49.65
320	152.00	0.76	411.18	1004.50	5.02	62.22
384	156.20	0.78	480.15	1013.40	5.07	74.01
448	161.00	0.81	543.48	1043.80	5.22	83.83
512	165.30	0.83	604.96	1152.90	5.76	86.74
576	169.80	0.85	662.54	1335.40	6.68	84.24
640	174.50	0.87	716.33	1419.50	7.10	88.06
704	179.30	0.90	766.87	1688.50	8.44	81.43
768	183.20	0.92	818.78	1869.90	9.35	80.22
832	188.20	0.94	863.44	1520.00	7.60	106.91
896	192.90	0.96	907.21	1070.30	5.35	163.51
960	197.70	0.99	948.41	1061.60	5.31	176.62
1024	202.40	1.01	988.14	1048.80	5.24	190.69

2. Small Messages Repeated One Hundred Times

For the next experiment data was collected from runs using the same message lengths, but the repetition count, ρ, was raised to one hundred. This gives $t_e = t/200$, as shown in Table E.2.

a. *Host–to–Node Performance*

Figure E.3 gives the transmission rates corresponding to this data.
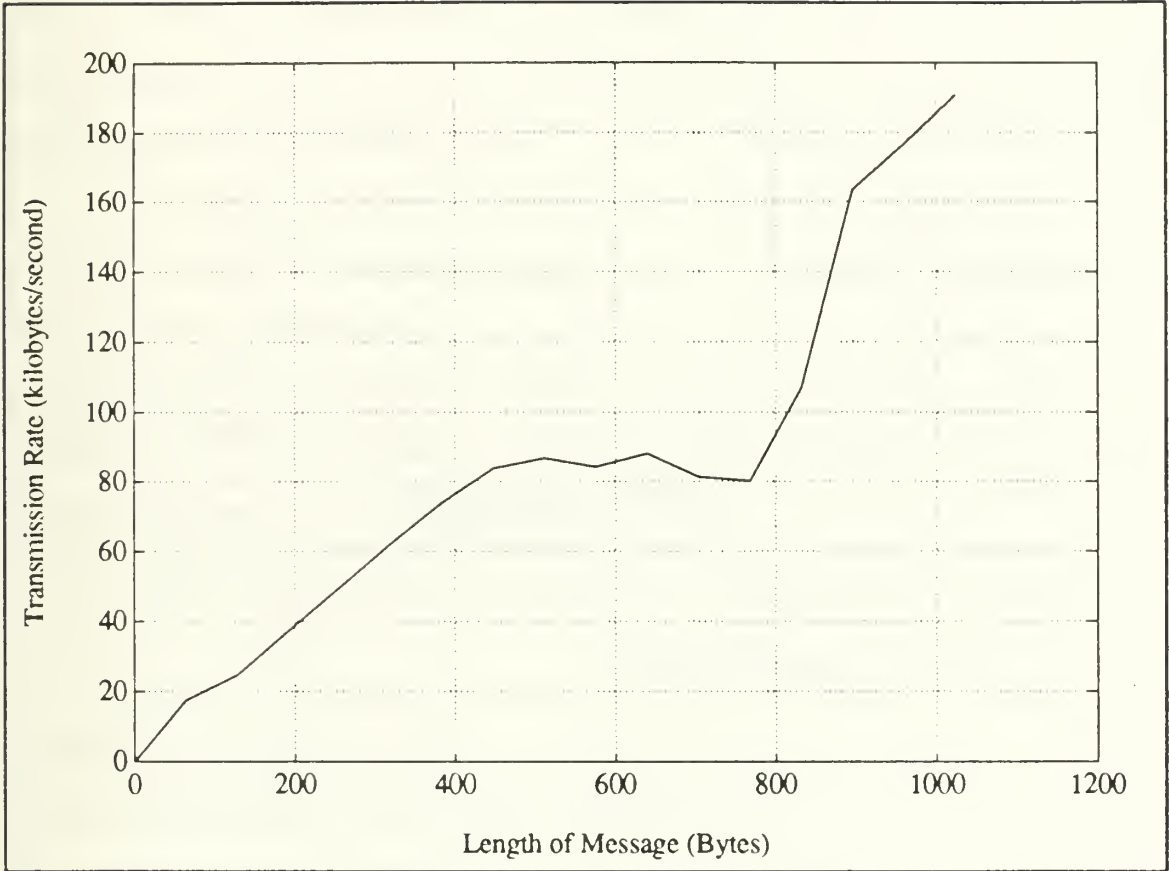
Figure E.3: Speed of Small Host–Node Messages (One Hundred Repetitions)
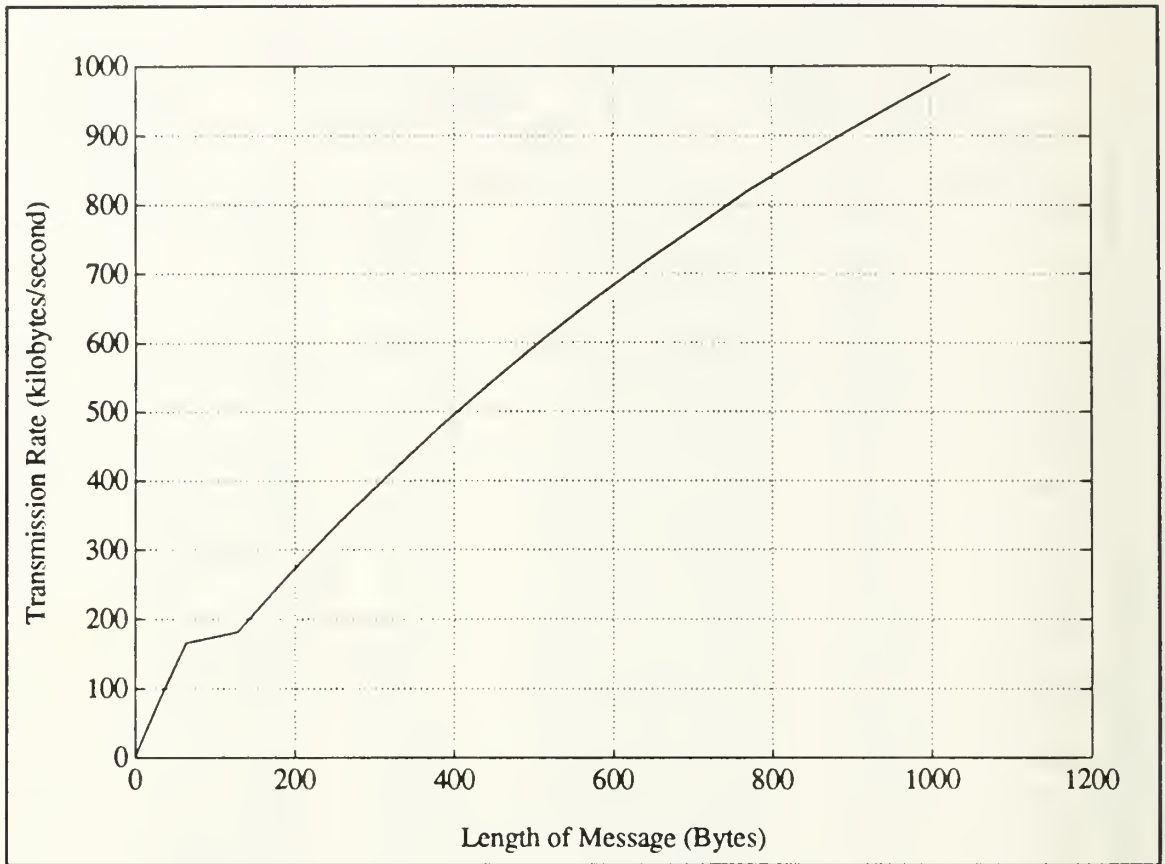
Figure E.4: Speed of Small Messages Between Nodes (One Hundred Repetitions)

b. *Node–to–Node Performance*

Figure E.4 shows the transmission rates for the node–to–node messages. This data may have important implications. Consider the transmission of a matrix row–by–row within a loop (where one row is transmitted each time through the loop). The expected communications performance is related to the number of bytes in a single row of the matrix, not the size of the entire matrix.

3. Larger Messages

The final test considered longer messages ($1,024 \leq \ell \leq 262,144$) that were not repeated. This gives $t_e = t/2$. Since the experiment was performed over a rather large set of message lengths, the data is divided at an arbitrary point. Messages of 64K bytes and less are designated "medium" length messages and placed into Table E.3. Messages of length 128K bytes and greater are designated "long" messages and placed into Table E.4. There is no hidden significance to this separation, it just made for tables of reasonable length.

The figures that follow are based upon the combined data of *both* of these Tables. The host terminates execution at the **crecv()** if we ask for more than 262,144 bytes in a single message. Chapter 2—iPSC/2 C Library Calls—of [Ref. 45: pp. 2–16, 2–19] explain: "messages to or from a host process are limited to a maximum of 256K bytes. There is no limit on message length between nodes." This explains why the data stops at that message size.

TABLE E.3: MESSAGES OF MEDIUM LENGTH

Message Length (Bytes)	Node-to-Node			Host-to-Node		
	t (msec)	t_e (msec)	Rate (kbytes/sec)	t (msec)	t_e (msec)	Rate (kbytes/sec)
1024	2.20	1.10	909.09	9.00	4.50	222.22
2048	2.80	1.40	1428.57	10.40	5.20	384.62
3072	3.70	1.85	1621.62	11.90	5.95	504.20
4096	4.40	2.20	1818.18	13.40	6.70	597.01
5120	5.10	2.55	1960.78	14.50	7.25	689.66
6144	5.80	2.90	2068.97	14.50	7.25	827.59
7168	6.50	3.25	2153.85	15.50	7.75	903.23
8192	7.40	3.70	2162.16	16.50	8.25	969.70
9216	8.10	4.05	2222.22	19.50	9.75	923.08
10240	8.80	4.40	2272.73	18.00	9.00	1111.11
11264	9.50	4.75	2315.79	18.90	9.45	1164.02
12288	10.30	5.15	2330.10	19.00	9.50	1263.16
13312	10.90	5.45	2385.32	19.60	9.80	1326.53
14336	11.80	5.90	2372.88	20.30	10.15	1379.31
15360	12.50	6.25	2400.00	21.90	10.95	1369.86
16384	13.20	6.60	2424.24	22.40	11.20	1428.57
17408	13.90	6.95	2446.04	23.30	11.65	1459.23
18432	14.60	7.30	2465.75	24.90	12.45	1445.78
19456	15.40	7.70	2467.53	24.30	12.15	1563.79
20480	16.10	8.05	2484.47	27.30	13.65	1465.20
21504	16.80	8.40	2500.00	27.10	13.55	1549.82
22528	17.60	8.80	2500.00	27.00	13.50	1629.63
23552	18.40	9.20	2500.00	27.80	13.90	1654.68
24576	19.10	9.55	2513.09	29.30	14.65	1638.23
25600	19.80	9.90	2525.25	29.40	14.70	1700.68
26624	20.50	10.25	2536.59	30.60	15.30	1699.35
27648	21.30	10.65	2535.21	30.90	15.45	1747.57
28672	22.10	11.05	2533.94	33.50	16.75	1671.64
29696	22.70	11.35	2555.07	38.50	19.25	1506.49
30720	23.50	11.75	2553.19	37.90	18.95	1583.11
31744	24.20	12.10	2561.98	37.90	18.95	1635.88
32768	24.90	12.45	2570.28	38.10	19.05	1679.79
65536	48.50	24.25	2639.18	59.90	29.95	2136.89

TABLE E.4: LONG MESSAGES

Message Length (Bytes)	Node–to–Node			Host–to–Node		
	t (msec)	t_e (msec)	Rate (kbytes/sec)	t (msec)	t_e (msec)	Rate (kbytes/sec)
131072	95.60	47.80	2677.82	109.40	54.70	2340.04
150528	109.60	54.80	2682.48	123.60	61.80	2378.64
161792	117.70	58.85	2684.79	131.60	65.80	2401.22
162816	118.40	59.20	2685.81	132.90	66.45	2392.78
163840	119.10	59.55	2686.82	133.60	66.80	2395.21
164864	119.90	59.95	2685.57	135.00	67.50	2385.19
165888	120.60	60.30	2686.57	136.30	68.15	2377.11
172032	125.00	62.50	2688.00	140.80	70.40	2386.36
182272	132.40	66.20	2688.82	148.10	74.05	2403.78
192512	139.70	69.85	2691.48	155.60	77.80	2416.45
202752	147.10	73.55	2692.05	164.60	82.30	2405.83
223232	161.80	80.90	2694.68	181.10	90.55	2407.51
243712	176.50	88.25	2696.88	194.80	97.40	2443.53
253952	183.80	91.90	2698.59	202.80	101.40	2445.76
259072	187.60	93.80	2697.23	205.50	102.75	2462.29
262144	189.70	94.85	2699.00	210.50	105.25	2432.30
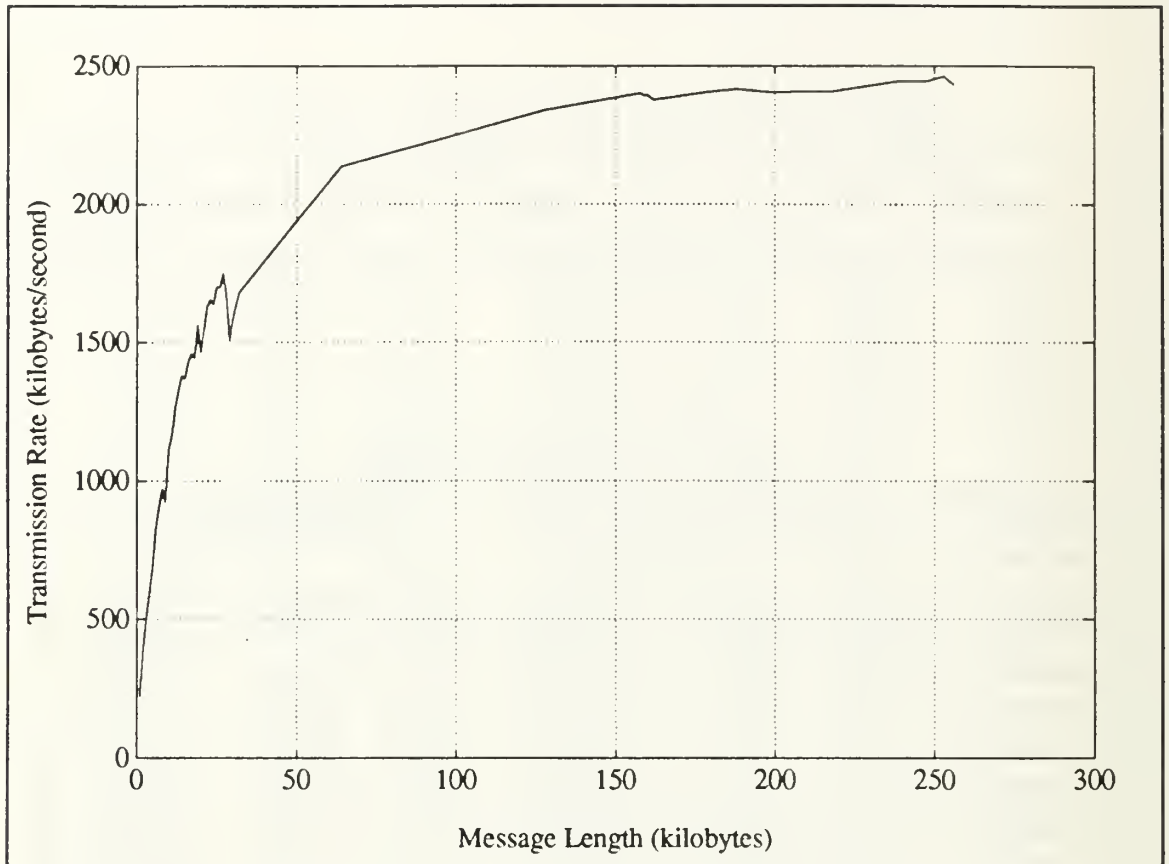
Figure E.5: Speed of Large Host–Node Messages

a. Host–to–Node Performance

The host–to–node communication rates (for large messages) are illustrated in Figure E.5.

190
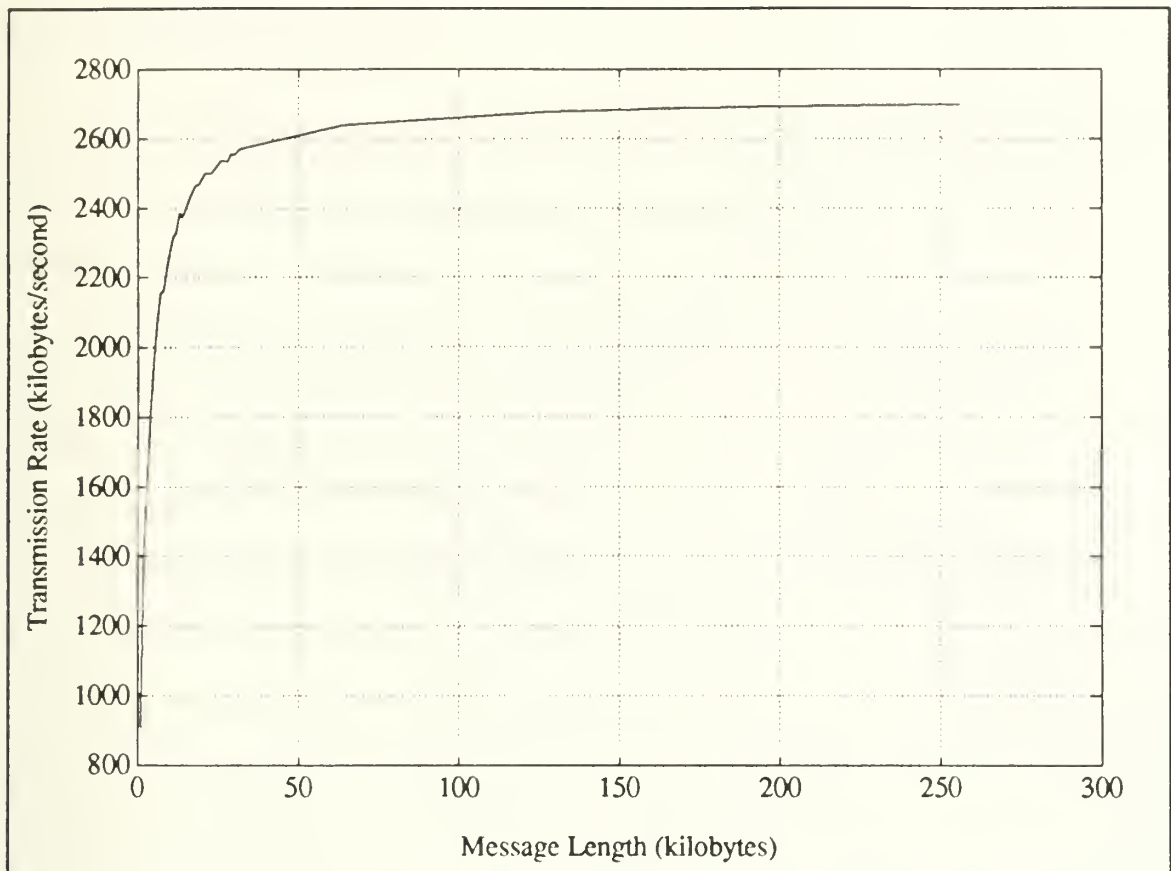
Figure E.6: Speed of Large Messages Between Nodes

b. Node–to–Node Performance

Figure E.6 shows the transmission rates for the same long messages when passed among nodes of the hypercube. To move the plot of Figure E.6 out into the open, a plot of transmission rate versus $\log_{10} \ell$ is shown in Figure E.7.
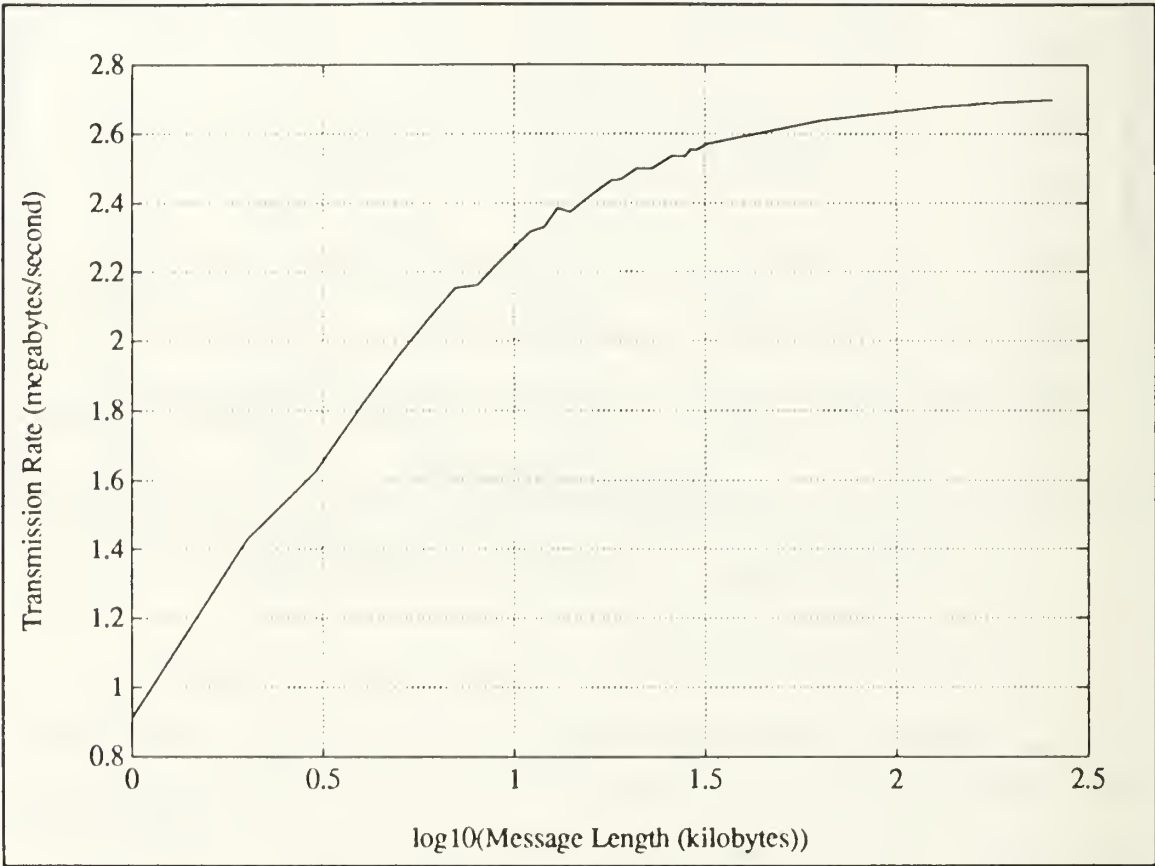
Figure E.7: Node–to–Node Transmission Rates for Large Messages

D. CONCLUSIONS

One of the obstacles that this experiment carefully avoided was competition for the links. Contention for communications resources may be inherent in certain parallel programs. Potential causes and effects of contention should always be given due consideration in the crafting of a parallel application. All of the algorithms that were tested in this research work involved very structured, regular communications schemes. An application with very random communication patterns should be expected to behave very differently. Additionally, the communication scheme for every program in this work was designed to use the shortest possible path.

The circuit switching approach has the disadvantage that a single message must control the entire path from origin to destination. Under a less controlled, random pattern of communications the performance of the communications subsystem might reasonably be expected to exhibit degraded performance. Other portions of this thesis show that a communication–bound algorithm can experience severe performance degradation as well. There is no specific claim that the results obtained in this experiment represent an *upper bound* for node–to–node communications within the hypercube, but they are probably good *estimates* for an upper bound.

Host–node communication is slower than node–to–node communication. This is not surprising (consider the physical distances and materials). In the absence of competition for the links, node–to–node transmission rates are essentially predictable for a given message length. There is a tremendous rise in transmission rate as message length goes from one byte to the vicinity of twenty kilobytes. Thereafter, smaller (apparently asymptotic) performance gains are achieved by increasing the message size. A similar phenomenon occurs with host–node communications but it takes much longer messages to break, say the two megabytes–per–second transmission rate.

These performance measures are quite appealing for long messages, but consider transmissions of shorter (and possibly repetitious) messages. The data shows that short messages are penalized, even if they are part of a loop that involves a good deal of communication. Each instance of **csend()** or **crecv()** is distinct and incurs its own start–up cost. This is an important note for anyone considering transmission of the rows (or columns) of a matrix within a loop structure. The potential of (pre-transmission) storage of matrices (two–dimensional arrays) into one–dimensional arrays *might* be investigated as a means of increasing the communications rate (provided the cost of copying the array is not prohibitive).

Communications in a transputer network was not developed in this work, but Bryant [Ref. 26] gives a very thorough analysis of communications and calculations in a network of transputers. On pages 31–34, Bryant gives a good summary of unidirectional and bidirectional data transfer rates. He discusses link interaction (i.e., how communications performance varies as one, two, or all four of the transputer's links are engaged in communication) on pages 34–38 and concludes that the effects of link interaction are minimal.

Bryant also discusses the effects of varied communication loads on processor performance. On pages 38–44, he finds that bombarding a transputer with many small messages while it is trying to perform calculations can severely degrade the processor's performance. His Figures 3.8 and 3.9 show that—with only one link active—messages of size 100 bytes and larger cause negligible performance degradation. With all four links active, messages of size greater than one kilobyte should be used to free the processor from most of the communications overhead.

Pages 36 and 37 of Bryant's thesis show the effects of message length on the communication rate. Bryant's Figures 3.4 and 3.5 are quite similar to Figure E.6 above, but the transputers are much more responsive (i.e., there seems to be less overhead involved, so the peak communications rate is achieved much earlier). In

194

fact, the transputers are near their peak transmission rate with messages of 100 bytes and messages of one kilobyte and greater always travel at peak rates.

Comparing a transputer system to an iPSC/2 system—in terms of communications performance—is essentially a lesson in the differences between store-and-forward switching versus circuit switching for multi-hop communications. Bryant shows [Ref. 26 : pp. 83–85] that the store-and-forward transmission rates suffer as the number of hops grows. The direct-connect (circuit switching) approach recovers its overhead on multi-hop communications, but it ties up the entire path to do so (making it unavailable to other potential users). The key difference is that communications performance with the direct-connect method is very nearly independent of the number of hops.

The transputer system seems to enforce true blocking communications on both the sending and receiving ends (byte-by-byte acknowledgment is part of the protocol). The iPSC/2 **csend()** is not blocking, but the **crecv()** function is blocking. Proper handling of these issues can become important when implementing an algorithm. Each method has advantages and disadvantages, but—at least for the current systems—transputers seem better suited for applications involving short messages over short distance and the iPSC/2 seems to handle long messages over long distances better.

E. SOURCE CODE LISTINGS

The source code listings for the programs used for these tests are supplied on the pages that follow. The makefile **commtst.mak** appears first and describes the dependencies among the files and compilation procedures. Next, **commtst.h** is the header file associated with these programs. Finally, the actual code is given in a host program called **commtst.c** and the node program **commtstn.c**.

```
 1 #   Author:   Jonathan E. Hartman, U. S. Naval Postgraduate School
 2 #   Purpose: Makefile for Hypercube Communications Test Programs
 3 #   Date:     07 August 1991
 4
 5 all: hostcode nodecode
 6
 7 help:
 8 chelp
 9
10
11 # -------------------------------------------------------------------
12 hostcode: commtst.o clargs.o
13 cc clargs.o commtst.o -host -o commtst
14
15 clargs.o:   clargs.h  clargs.c
16 commtst.o:  commtst.h commtst.c
17
18
19 # -------------------------------------------------------------------
20 nodecode: commtstn.o
21 cc commtstn.o -node -o commtstn
22
23 commtstn.o: commtstn.c commtst.h
24
25
26 # Execute it! ------------------------------------------------------
27 run: all
28 commtst -d 3 -b 1024 -r 2
29
30
31 # Delete object files, executables ---------------------------------
32 clean:
33 rm *.o
34 rm commtst
35 rm commtstn
36
37 # EOF commtst.mak --------------------------------------------------
```

196

```
 1 /* --------------========= PROGRAM INFORMATION =========--------------
 2  *
 3  * SOURCE      :   commtst.h
 4  * VERSION     :   1.2
 5  * DATE        :   07 August 1991
 6  * AUTHOR      :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  *
 9  * --------------============= DESCRIPTION =============--------------
10  *
11  * This header file gives common information for use across the host program
12  * commtst.c and the node program commtstn.c.  A more complete description
13  * can be found in commtst.c.
14  *
15  * --------------===================================================----------
16  */
17
18 #ifndef EXIT_FAILURE
19 #define EXIT_FAILURE    -1
20 #endif
21
22 #define MAX_CUBESIZE    16
23
24 #define ROOT            -1
25
26 #define RECEIVE         0
27 #define SEND            1
28
29 #define FALSE           0
30 #define TRUE            1
31
32
33 /* --------------============= TYPE DEFINITION =============--------------
34  *
35  *    The following structure is the framework that the root processor (host)
36  *    uses to pass instructions to the worker nodes in the cube.
37  */
38
39 typedef struct {
40
41    int  task;                    /*  choose RECEIVE or SEND as above    */
42    long bytes;                   /*  length of message                  */
43    long reps;                    /*  number of repetitions              */
44    int  destination[MAX_CUBESIZE]; /*  for senders: identifies addressees */
45
46 } Tasking;
47
48
49 /* --------------============= EOF commtst.h =============--------------- */
```

197

```
1  /* -------------------========        PROGRAM INFORMATION        ==========--------------
2   *
3   *    SOURCE      :    commtst.c
4   *    VERSION     :    1.2
5   *    DATE        :    07 August 1991
6   *    AUTHOR      :    Jonathan E. Hartman, U. S. Naval Postgraduate School
7   *
8   *    USAGE       :    commtst [-d dimension] [-b bytes] [-r repetitions] [-v]
9   *
10  *    EXAMPLE     :    If you type 'commtst -d 3 -v -b 1024 -r 10', it means to
11  *                     run the program on a dimension 3 hypercube in the verbose
12  *                     mode, with messages of length 1024 bytes, and 10 repeti-
13  *                     tions for each message.
14  *
15  *    REFERENCES  :    [1] iPSC/2 Programmer's Reference Manual
16  *
17  *
18  * -------------------================        DESCRIPTION        ==============--------------
19  *
20  *
21  *    This program runs on the host.  It orchestrates various point-to-point
22  *    communication tasks between nodes of a hypercube.  The time of round-trip
23  *    communications is gathered and printed out.  The output includes the time
24  *    required and rate of communication (taking into account repetitions and
25  *    round-trips).  The 'verbose' mode gives a more detailed node-by-node
26  *    accounting of the run.
27  *
28  * ----------------================================================================--------------
29  */
30
31  char *version = "Hypercube Communications Test, Version 1.2";
32
33
34  /* -------------------================        ALGORITHM        ==============--------------
35  *
36  *    The root (host) processor determines who will communicate with whom, and
37  *    when.  No node operates independently.  The host identifies a sender and
38  *    receiver(s).  The host also gives the length of the message that should
39  *    be passed and the number of times that the message is to be repeated
40  *    (multiple repetitions may be required when the message is short since
41  *    mclock() returns milliseconds).  The 'Tasking' structure holds instruc-
42  *    from the manager (i.e., SEND or RECEIVE, the length of the message, num-
43  *    ber of repetitions, and addressees).  When this structure is received at
44  *    a node, it performs the task and awaits further instructions from the
45  *    manager processor.  If the processor is a sender, it returns timing data
46  *    to the host upon completion.
47  *
48  * ----------------================================================================--------------
49  */
50
```

198

```
51 #include <stdio.h>
52 #include "commtst.h"
53 #include "ipsc.h"
54 #include "macros.h"
55 #include "clargs.h"
56
57 #define ASCII_CONVERSION 48   /*  for char -> int conversion of 0...3   */
58 #define CT_SIZE          4    /*  for cubetype[] size                   */
59
60 #define NUM_ARGS         4    /*  -d -b -r -v                           */
61 #define DIM              0    /*  index values into optv[]              */
62 #define BYTES            1
63 #define REPS             2
64 #define VERBOSE          3
65
66
67 /* ----------=============    FUNCTION DEFINITION    =============--------- */
68
69 #ifdef PROTOTYPE
70
71     void init(int argc, char **argv,  char cubetype[CT_SIZE],
72              int *dim, long *bytes, long *reps, int *verbose)
73
74 #else
75
76     void init(argc, argv, cubetype, dim, bytes, reps, verbose)
77
78         int  argc;
79         char **argv,
80              cubetype[CT_SIZE];
81         int  *dim;
82         long *bytes,
83              *reps;
84         int  *verbose;
85
86 #endif
87 {
88     int count = 1,
89         valid = FALSE;
90
91     Opt_Struct *optv[NUM_ARGS];
92
93
94     /*  The first step is to make a table of all of the valid arguments.  The
95      *  structure is defined more carefully in clargs.h, but the basic idea is
96      *  that we have an array of pointers to type Opt_Struct (option structure)
97      *  ...in this case, there are NUM_ARGS valid arguments and the next few
98      *  steps take care of allocation and definition of them.  When this is
99      *  done, it is time to call interpret_args() to see what the user entered.
100     */
```

199

```
101
102     optv[DIM]       = (Opt_Struct *) calloc( 1, sizeof(Opt_Struct) );
103     optv[BYTES]     = (Opt_Struct *) calloc( 1, sizeof(Opt_Struct) );
104     optv[REPS]      = (Opt_Struct *) calloc( 1, sizeof(Opt_Struct) );
105     optv[VERBOSE]   = (Opt_Struct *) calloc( 1, sizeof(Opt_Struct) );
106     optv[DIM]->lanswer  = (long *) calloc( 1, sizeof(long) );
107     optv[BYTES]->lanswer = (long *) calloc( 1, sizeof(long) );
108     optv[REPS]->lanswer  = (long *) calloc( 1, sizeof(long) );
109
110     /*  The intel compiler didn't like ...->argname = "-d"; etc.  */
111     optv[DIM]->argname[0]      = '-';
112     optv[DIM]->argname[1]      = 'd';
113     optv[DIM]->subargc         = 1;
114     optv[DIM]->subargi         = NEXT_LONG;
115
116     optv[BYTES]->argname[0]    = '-';
117     optv[BYTES]->argname[1]    = 'b';
118     optv[BYTES]->subargc       = 1;
119     optv[BYTES]->subargi       = NEXT_LONG;
120
121     optv[REPS]->argname[0]     = '-';
122     optv[REPS]->argname[1]     = 'r';
123     optv[REPS]->subargc        = 1;
124     optv[REPS]->subargi        = NEXT_LONG;
125
126     optv[VERBOSE]->argname[0]  = '-';
127     optv[VERBOSE]->argname[1]  = 'v';
128     optv[VERBOSE]->subargc     = 0;
129
130     *dim = -1;
131
132     interpret_args(argc, argv, NUM_ARGS, optv);
133
134     if (optv[DIM]->found)   *dim = (int) optv[DIM]->lanswer[0];
135
136     switch (*dim) {
137
138         case 0 :   case 1 :   case 2 :   case 3 : break;
139
140         default:
141         while (!valid) {
142
143             printf("Enter desired cube dimension (in {0, 1, 2, 3}): ");
144             scanf("%d", dim);
145             fflush(stdin);
146             switch(*dim){
147                 case 0 : case 1 : case 2 : case 3 : valid = TRUE; break;
148             }
149         }
150     } /* end switch() */
```

```
151      if (optv[BYTES]->found)    *bytes = optv[BYTES]->lanswer[0];
152
153      valid = FALSE;
154
155      if (*bytes < 1) {
156          while (!valid) {
157              printf("Enter message length (bytes): ");
158              scanf("%ld", bytes);
159              fflush(stdin);
160              if (*bytes > 0){ valid = TRUE; }
161              else { printf("Message length must be positive.\n"); }
162          }
163      }
164
165      if (optv[REPS]->found) { *reps = optv[REPS]->lanswer[0]; }
166      else {
167
168          printf("Non-existing (or invalid) repetition count, ");
169          printf("using one repetition.\n\n");
170          *reps = 1;
171      }
172
173      (optv[VERBOSE]->found) ? *verbose = TRUE : *verbose = FALSE;
174
175      cubetype[0] = 'd';                       /* for dimension (to follow)    */
176      cubetype[1] = (char)(*dim + ASCII_CONVERSION);
177      cubetype[2] = 'f';                       /* means nodes are 386/387 combo */
178      cubetype[3] = 0;
179
180      printf("Initialization complete...Cube Dimension: %d\n", *dim);
181      printf("                          Message Length: %ld\n", *bytes);
182      printf("                          Repetitions:    %ld\n\n", *reps);
183      if (*verbose) printf("                              Verbose Mode:    ON");
184 }
185 /* End init() ------------------------------------------------------------- */
186
187
188
189 #ifdef PROTOTYPE
190
191      main(int argc, char *argv[])
192
193 #else
194
195      main(argc, argv)
196
197          int   argc;
198          char *argv[];
199
200 #endif
```

201

```
201  { /* begin main() */
202
203      char *cubename          = "Hypercube",
204           cubetype[CT_SIZE],
205           *msg,
206           *nodecode          = "commtstn";
207
208      float avg,
209            avg_hostrate,
210            avg_hosttime,
211            avg_rate,
212            avg_time,
213            bytes,
214            reps;
215
216      int  cubesize,
217           dim,
218           i,
219           j,
220           verbose;
221
222      unsigned long **timing_data;
223
224      Tasking task_packet;
225
226
227      printf("\n%s\n\n", version);
228
229      init(argc, argv, cubetype, &dim, &(task_packet.bytes),
230          &(task_packet.reps), &verbose);
231
232      bytes  = (float) task_packet.bytes;
233      reps   = (float) task_packet.reps;
234      bytes *= (2.0 * reps);   /* account for two-way communications, reps */
235
236      cubesize = POW2(dim);
237
238      timing_data = (unsigned long **) calloc(cubesize, sizeof(unsigned long*));
239
240      for (i = 0; i < cubesize; i++) {
241
242          timing_data[i]=(unsigned long*)calloc(cubesize,sizeof(unsigned long));
243      }
244
245      if (!(msg = (char *) calloc(task_packet.bytes, sizeof(char)))) {
246
247          printf("main():  Allocation failure for msg.\n");
248          exit(EXIT_FAILURE);
249      }
250
```

202

```
251     /*  Get the cube and load the node code  */
252
253     getcube(cubename, cubetype, NULL, 0);
254     attachcube(cubename);
255     setpid(0);
256     load(nodecode, ALL_NODES, NODE_PID);
257
258
259     /*  Perform the tasking, receive the message, return it, receive and print
260      *  timing data...repeat for all players.  The outer loop index, i, will
261      *  represent the sender node.  The j index runs the other (RECEIVE)
262      *  players.
263      */
264
265     for (i = 0; i < cubesize; i++) {
266
267         /*  Get the receivers ready first  */
268         task_packet.task            = RECEIVE;
269         task_packet.destination[0] = i;
270         task_packet.destination[1] = cubesize; /* impossible flags end */
271
272         for (j = 0; j < i; j++) {
273
274             csend(0, &task_packet, sizeof(Tasking), j, NODE_PID);
275         }
276
277         for (j = (i+1); j < cubesize; j++) {
278
279             csend(0, &task_packet, sizeof(Tasking), j, NODE_PID);
280         }
281
282         /*  Then prepare the sender ==> he can start  */
283         task_packet.task                  = SEND;
284         for (j = 0; j < i; j++)            task_packet.destination[j] = j;
285         task_packet.destination[i]        = ROOT;
286         for (j = (i+1); j < cubesize; j++) task_packet.destination[j] = j;
287
288         csend(0, &task_packet, sizeof(Tasking), i, NODE_PID);
289
290         /*  Receive from the sender and return his message  */
291         for (j = 0; j < task_packet.reps; j++) {
292
293             crecv(ANY_TYPE, msg, task_packet.bytes);
294             csend(0, msg, task_packet.bytes, i, NODE_PID);
295         }
296
297         /*  Receive the timing data from this run and print it  */
298         crecv(ANY_TYPE, timing_data[i], (cubesize * sizeof(unsigned long)) );
299
300     } /* end for (i) */
```

203

```
301     for (i = 0; i < cubesize; i++) {
302
303         if (verbose) {
304
305             printf("Source   Dest.   Time (msec)   Rate (kilobytes/second)\n");
306             printf("======   =====   ===========   ======================\n");
307             printf("%4d      HOST %10lu      ", i, timing_data[i][i]);
308             printf(" %10.2f\n", (bytes / ((float) timing_data[i][i])) );
309         }
310
311         avg   = 0.0;
312
313         for (j = 0; j < cubesize; j++) {
314
315             if (i != j) {
316
317                 avg += (float) timing_data[i][j];
318
319                 if (verbose) {
320
321                     printf("          %4d", j);
322                     printf("   %10lu      ", timing_data[i][j]);
323                     printf("%10.2f\n", (bytes / ((float) timing_data[i][j])) );
324                 }
325             }
326
327             if (j == (cubesize - 1)) {
328
329                 avg /= (float) cubesize - 1;
330
331                 if (verbose) {
332
333                     printf("=============================================");
334                     printf("=========\n");
335                     printf("Averages...........%9.1f msec ", avg);
336                     printf("  %7.2f", bytes/avg );
337                     printf(" kbytes/sec\n\n\n");
338                 }
339             }
340         } /* end for(j) */
341     } /* end for(i) */
342
343     for (i = 0; i < cubesize; i++) {
344
345         for (j = 0; j < cubesize; j++) {
346
347             (i == j) ? avg_hosttime += timing_data[i][j] :
348                        avg_time     += timing_data[i][j] ;
349         }
350     }
```

204

```
351     avg_hosttime /= cubesize;
352     avg_hostrate  = bytes/avg_hosttime;
353
354     avg_time       /= ((cubesize - 1) * cubesize);
355     avg_rate        = bytes/avg_time;
356
357     printf("If we average all of the times and rates....\n\n");
358     printf("    Average Time:  %9.1f  milliseconds\n", avg_time);
359     printf("    Average Rate:  %10.2f kilobytes/second\n\n\n", avg_rate);
360
361     printf("NOTE:  Average and Rate values are for the nodes ONLY.\n");
362     printf("       They do not include the host timing data.\n\n\n");
363
364     printf("The averages for the node <--> host communications were:\n\n");
365     printf("    Average Time:  %9.1f  milliseconds\n", avg_hosttime);
366     printf("    Average Rate:  %10.2f kilobytes/second\n\n\n", avg_hostrate);
367
368 }
369 /* ---------------------============  EOF commtst.c   ============------------- */
```

```
1  /*. -----------------=========     PROGRAM INFORMATION     ==========--------------
2  *
3  *  SOURCE      :   commtstn.c
4  *  VERSION     :   1.2
5  *  DATE        :   07 August 1991
6  *  AUTHOR      :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7  *
8  *
9  *  ------------------============     DESCRIPTION     ============--------------
10 *
11 *  This program is loaded by commtst.c (which runs on the host).  This code
12 *  (commtstn.c) runs on the nodes of a hypercube created by the host program.
13 *  For more information, see commtst.c.
14 *
15 *  --------------===================================================-------------
16 */
17
18
19 #include <stdio.h>
20 #include "commtst.h"
21 #include "ipsc.h"
22
23 #define SUCCESS 0
24
25
26
27 #ifdef PROTOTYPE
28
29     main(int argc, char *argv[])
30
31 #else
32
33     main(argc, argv)
34
35         int  argc;
36         char *argv[];
37 #endif
38 {
39     char *msg;
40
41     int  cubesize = numnodes(),
42          i,
43          j,
44          return_addr;
45
46     long rep;
47
48     unsigned long start, *timing_data;
49
50     Tasking task_packet;
```

206

```
51      timing_data = (unsigned long*) calloc(cubesize, sizeof(unsigned long));
52
53      for (i = 0; i < cubesize; i++) {
54
55          crecv(ANY_TYPE, &task_packet, sizeof(Tasking));
56
57          msg = (char *) calloc(task_packet.bytes, sizeof(char));
58
59          switch (task_packet.task) {
60
61              case RECEIVE :
62
63                  return_addr = task_packet.destination[0];
64
65                  for (rep = 0; rep < task_packet.reps; rep++) {
66
67                      crecv(ANY_TYPE, msg, task_packet.bytes);
68                      csend(0, msg, task_packet.bytes, return_addr, NODE_PID);
69                  }
70
71                  break;
72
73
74              case SEND :
75
76                  j = 0;
77
78                  while ((j<cubesize)&&(task_packet.destination[j]<cubesize)) {
79
80                      start = mclock();
81
82                      for (rep = 0; rep < task_packet.reps; rep++) {
83
84                          (j == mynode()) ?
85                              csend(0,msg,task_packet.bytes,myhost(),NODE_PID):
86                              csend(0, msg, task_packet.bytes, j, NODE_PID);
87
88                          crecv(ANY_TYPE, msg, task_packet.bytes);
89                      }
90
91                      timing_data[j] = mclock() - start;
92
93                      j++;
94                  }
95
96                  /*  Return the timing data  */
97                  csend(0, timing_data, (cubesize * sizeof(unsigned long)),
98                          myhost(), NODE_PID);
99
100                 break;
```

207

```
101                default :
102
103                    printf("Unrecognized task at node %ld.\n", mynode() );
104                    exit(EXIT_FAILURE);
105
106        } /* end switch() */
107
108
109        free(msg);
110
111
112    } /* end for() */
113
114    return(SUCCESS);
115
116 }
117 /* -------------=========          EOF  commtstn.c    ==========------------ */
```

APPENDIX F
MATRIX LIBRARY

This appendix contains part of the matrix library, **matlib** that is often used and referenced in other sections and code. It could be argued that "matrix library" is a misnomer since much of the code has little to do with matrices. This criticism is true, but I will defend the name since *the entire reason* for the creating such a library was to handle matrices in a more reasonable way. The last section of this appendix contains all of the source code for Gauss factorization with partial pivoting, and a short excerpt from the complete pivoting code.

The specifications and a portion of the source code for the library are given on the pages to follow. The original intent was to include the source code in its entirety, but this would require more than double the current number of pages so the source has been omitted. The files are divided into three logical groups:

1. Makefiles that simplify maintenance of the library, show dependencies among the files, and describe the compilation procedures that are used to generate the loadable (executable) code.

2. Standard files (mostly C header files) that make definitions available (for consistency) across a wide range of files. The range is implied by the content of the file. These files include *manifest constants* that are installed using the C Preprocessor **#define** directive, type definitions that are intended for use across several files, and macro definitions that are expanded by the C Preprocessor.

3. Source code files that appear in pairs, like **filename.h** and **filename.c** or (mostly) as a header file alone. The header file gives remarks, definitions of manifest con-

209

stants, type definitions, and function declarations (specifications) that pertain to the associated source code (i.e., the code within **filename.c**). Again, the latter has been omitted in most cases.

4. The Gauss factorization code. All of the source code for the partial pivoting version is given, and an excerpt of the pivot election function from the complete pivoting code is also provided.

A. MAKEFILES

logc.mak This makefile is a standard template for programs compiled with the Logical Systems C (version 89.1) product.

matlib.mak This makefile is used to translate **matlib** into a useable form. With Logical Systems C, it creates a library suitable for installation and use as any other normal C library. The portion of the makefile used on the Intel iPSC/2 simply works in the current directory to translate the source into object code so that other programs can reference it.

```
 1  # ------------------------------------------------------------------------
 2  #
 3  #    AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 4  #    PURPOSE :   Makefile for Hypercube Communications Test Programs (LogC)
 5  #    DATE     :   10 August 1991
 6  #
 7  # ------------------------------------------------------------------------
 8
 9  ROOTCODE=filename
10  NODECODE=filename
11  NIF_FILE=filename
12
13
14  # ----------------------         OPTIONS AND DEFINITIONS   ----------------------
15  #
16  #    The following section establishes various options and definitions.  We
17  #    start with PP, the Logical Systems C Preprocessor.  The '-dX' option
18  #    (with no macro_expression) is like '#define X 1'.  Next we set up the
19  #    compilation options for Logical Systems' TCX Transputer C Compiler.  The
20  #    '-c' means compress the output file.  The options beginning with '-p'
21  #    tell TCX to generate code for the appropriate processor:
22  #
23  #         -p2           T212 or T222
24  #         -p25          T225
25  #         -p4           T414
26  #         -p45          T400 or T425
27  #         -p8           T800
28  #         -p85          T801 or T805
29  #
30  #    Logical Systems' TASM Transputer Assembler is next.  The '-c' means
31  #    compress the output file (it can cut it in half)!  The '-t' is used
32  #    because the input to TASM will be from a language translator (TCX's
33  #    output) and not from assembly source code.
34  #
35  #    The final list tells TLNK which libraries to look at during linking.
36  #    It also establishes an entry point.  You should always use _main for
37  #    the root node; otherwise use _ns_main (for other nodes).
38
39  PPOPT2=-dPROTOTYPE -dTRANSPUTER -dT212
40  PPOPT4=-dPROTOTYPE -dTRANSPUTER -dT414
41  PPOPT8=-dPROTOTYPE -dTRANSPUTER -dT800
42  TCXOPT2=-cp2
43  TCXOPT4=-cp4
44  TCXOPT8=-cp8
45  TASMOPT=-ct
46  T2LIB=t2lib.tll
47  T4LIB=matlib4.tll t4cube.tll t4lib.tll
48  T8LIB=matlib8.tll t8cube.tll t8lib.tll
49  RENTRY=_main
50  NENTRY=_ns_main
```

211

```
51
52
53 # ---------------------          DEFAULT ===> MAKE ALL   -----------------------
54 #
55
56 all: $(ROOTCODE).tld $(NODECODE).tld
57
58
59
60
61
62 # ---------------------------          ROOT CODE   ---------------------------
63 #
64
65 $(ROOTCODE): $(ROOTCODE).tld
66
67 $(ROOTCODE).tld: $(ROOTCODE).trl
68      echo FLAG    c                >   $(ROOTCODE).lnk
69      echo LIST    $(ROOTCODE).map >>  $(ROOTCODE).lnk
70      echo INPUT   $(ROOTCODE).trl >>  $(ROOTCODE).lnk
71      echo ENTRY   $(RENTRY)       >>  $(ROOTCODE).lnk
72      echo LIBRARY $(T4LIB)        >>  $(ROOTCODE).lnk
73      tlnk $(ROOTCODE).lnk
74
75 $(ROOTCODE).trl: $(ROOTCODE).c
76      pp   $(ROOTCODE).c   $(PPOPT4)
77      tcx  $(ROOTCODE).pp  $(TCXOPT4)
78      tasm $(ROOTCODE).tal $(TASMOPT)
79
80
81
82
83
84 # ---------------------------          NODE CODE   ---------------------------
85 #
86
87 $(NODECODE): $(NODECODE).tld
88
89 $(NODECODE).tld: $(NODECODE).trl
90      echo FLAG    c                >   $(NODECODE).lnk
91      echo LIST    $(NODECODE).map >>  $(NODECODE).lnk
92      echo INPUT   $(NODECODE).trl >>  $(NODECODE).lnk
93      echo ENTRY   $(NENTRY)       >>  $(NODECODE).lnk
94      echo LIBRARY $(T8LIB)        >>  $(NODECODE).lnk
95      tlnk $(NODECODE).lnk
96
97 $(NODECODE).trl: $(NODECODE).c
98      pp   $(NODECODE).c   $(PPOPT8)
99      tcx  $(NODECODE).pp  $(TCXOPT8)
100     tasm $(NODECODE).tal $(TASMOPT)
```

212

```
101
102 # --------------------------   EXECUTION   ------------------------------
103 #
104
105 run: $(ROOTCODE).tld $(NODECODE).tld $(NIF_FILE).nif
106     ld-net $(NIF_FILE)
107
108
109 # ---------------------------   CLEAN  UP   ------------------------------
110 #
111
112 clean:
113     del $(ROOTCODE).lnk
114     del $(NODECODE).lnk
115     del $(ROOTCODE).map
116     del $(NODECODE).map
117     del $(ROOTCODE).tal
118     del $(NODECODE).tal
119     del $(ROOTCODE).pp
120     del $(NODECODE).pp
121     del $(ROOTCODE).trl
122     del $(NODECODE).trl
123
124
125 # EOF logc.mak -----------------------------------------------------------
```

```
 1 # ---------========== MAKEFILE FOR MATRIX LIBRARY ==========----------
 2 #
 3 #    SOURCE  :   matlib.mak
 4 #    DATE    :   17 August 1991
 5 #    AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 6 #
 7 #    PURPOSE :   Make the matrix library 'matlib'.
 8 #
 9 #    REMARKS :   This makefile works with Logical Systems C, version 89.1,
10 #                and the Intel iPSC/2 compiler. The LogC portions of this
11 #    makefile actually construct libraries of the functions available in the
12 #    source files indicated. There are two libraries generated--matlib4.tll
13 #    & matlib8.tll---since the code is compiled for T414 or T800 processors.
14 #    For the Intel compiler, I have not created a library; but have used the
15 #    object code as needed. There are a few sections that pertain to both
16 #    compilers. The sections that only pertain to a particular compiler are
17 #    clearly marked 'Intel iPSC/2' or 'Logical Systems C'.
18 #
19 # ---------=================================================================----------
20
21
22
23
24
25 # ---------========== 1.) DEFINITIONS AND OPTIONS ==========----------
26 #
27 #    The following options and definitions are required. A more thorough
28 #    explanation can be found in 'logc.mak' or in the Logical Systems C
29 #    Transputer Toolset manual.
30 #
31 # ---------=================================================================----------
32
33 THISMAKEFILE=matlib.mak
34
35
36 # ---------================== 1.1) Intel iPSC/2 ==================----------
37 #
38
39 # MATLIBDIR is the directory that contains the matlib files
40 MATLIBDIR = /usr/hartman/matlib
41 OBJECTS   = clargs.o comm.o hcube.o generate.o mat_ops.o matrixio.o memory.o math.o
sep.o timing.o vec_ops.o
42
43
44
45
46 # ---------========== 1.2) Logical Systems C ==========----------
47 #
48
49 T414LIBNAME=matlib4
```

214

```
50 T800LIBNAME=matlib8
51
52 TRL4FILES=clargs.trl4 comm.trl4 complex.trl4 generate.trl4 machine.trl4 mat_ops.trl4
math.trl4 matrixio.trl4 memory.trl4 num_sys.trl4 sep.trl4 timing.trl4 vec_ops.trl4
53 TRL8FILES=clargs.trl8 comm.trl8 complex.trl8 generate.trl8 machine.trl8 mat_ops.trl8
math.trl8 matrixio.trl8 memory.trl8 num_sys.trl8 sep.trl8 timing.trl8 vec_ops.trl8
54
55 TLIB4FILES=clargs comm complex generate machine mat_ops math matrixio memory num_sys
sep timing vec_ops
56 TLIB8FILES=clargs comm complex generate machine mat_ops math matrixio memory num_sys
sep timing vec_ops
57
58 PPOPT2=-dPROTOTYPE -dTRANSPUTER -dT212
59 PPOPT4=-dPROTOTYPE -dTRANSPUTER -dT414
60 PPOPT8=-dPROTOTYPE -dTRANSPUTER -dT800
61
62 TCXOPT2=-cp2
63 TCXOPT4=-cp4
64 TCXOPT8=-cp8
65
66 TASMOPT=-ct
67
68 T2LIB=t2lib.tll
69 T4LIB=matlib4.tll t4cube.tll t4lib.tll
70 T8LIB=matlib8.tll t8cube.tll t8lib.tll
71
72 RENTRY=_main
73 NENTRY=_ns_main
74
75
76
77
78
79 # -----------======    2.) INSTRUCTIONS FOR DEFAULT MAKE   ================----------
80 #
81 #    The following sections give the default (since they appear first in the
82 #    makefile) options for this makefile.  By commenting one or the other
83 #    out, one can get to the defaults easily.
84 #
85 # --------=====================================================================--------
86
87 ipsc:  imatlib
88 clean: iclean
89 #   tptr: tmatlib
90 #   clean: tclean
91
92
93 # ------------================    2.1) Intel iPSC/2   ================----------
94 #
95
```

215

```
 96 imatlib: $(OBJECTS)
 97
 98
 99
100 # ----------------============  2.2) Logical Systems C   =============----------
101 #
102 #    Make everything and install in the library directory designated by the
103 #    environment variable TLIB.
104
105
106 tmatlib:
107 make -f $(THISMAKEFILE) $(T414LIBNAME).tll
108 make -f $(THISMAKEFILE) install4
109 make -f $(THISMAKEFILE) tclean
110 make -f $(THISMAKEFILE) $(T800LIBNAME).tll
111 make -f $(THISMAKEFILE) install8
112 make -f $(THISMAKEFILE) tclean
113 make -f $(THISMAKEFILE) install_headers
114
115
116 # ---------------    CREATE T414 VERSION OF THE LIBRARY    ----------------
117
118
119 $(T414LIBNAME).tll : $(TRL4FILES)
120 tlib $(T414LIBNAME) -b $(TLIB4FILES)
121
122 clargs.trl4 : clargs.h clargs.c
123 pp    clargs.c      $(PPOPT4)
124 tcx   clargs.pp     $(TCXOPT4)
125 tasm clargs.tal     $(TASMOPT)
126
127 comm.trl4 : comm.h comm.c
128 pp    comm.c        $(PPOPT4)
129 tcx   comm.pp       $(TCXOPT4)
130 tasm comm.tal       $(TASMOPT)
131
132 complex.trl4 : complex.h complex.c
133 pp    complex.c     $(PPOPT4)
134 tcx   complex.pp    $(TCXOPT4)
135 tasm complex.tal    $(TASMOPT)
136
137 generate.trl4 : generate.h generate.c matrix.h memory.trl4
138 pp    generate.c    $(PPOPT4)
139 tcx   generate.pp   $(TCXOPT4)
140 tasm generate.tal   $(TASMOPT)
141
142 hcube.trl4 : hcube.h hcube.c
143 pp    hcube.c       $(PPOPT4)
144 tcx   hcube.pp      $(TCXOPT4)
145 tasm hcube.tal      $(TASMOPT)
```

216

```
146
147 machine.trl4 : machine.h machine.c
148 pp    machine.c        $(PPOPT4)
149 tcx  machine.pp        $(TCXOPT4)
150 tasm machine.tal       $(TASMOPT)
151
152 mat_ops.trl4 : mat_ops.h mat_ops.c matrix.h
153 pp    mat_ops.c        $(PPOPT4)
154 tcx  mat_ops.pp        $(TCXOPT4)
155 tasm mat_ops.tal       $(TASMOPT)
156
157 math.trl4 : math.h math.c
158 pp    math.c           $(PPOPT4)
159 tcx  math.pp           $(TCXOPT4)
160 tasm math.tal          $(TASMOPT)
161
162 matrixio.trl4 : matrixio.h matrixio.c ascii.h matrix.h memory.trl4
163 pp    matrixio.c       $(PPOPT4)
164 tcx  matrixio.pp       $(TCXOPT4)
165 tasm matrixio.tal      $(TASMOPT)
166
167 memory.trl4 : memory.h memory.c matrix.h
168 pp    memory.c         $(PPOPT4)
169 tcx  memory.pp         $(TCXOPT4)
170 tasm memory.tal        $(TASMOPT)
171
172 num_sys.trl4 : num_sys.h num_sys.c matrix.h
173 pp    num_sys.c        $(PPOPT4)
174 tcx  num_sys.pp        $(TCXOPT4)
175 tasm num_sys.tal       $(TASMOPT)
176
177 sep.trl4 : sep.h sep.c
178 pp    sep.c            $(PPOPT4)
179 tcx  sep.pp            $(TCXOPT4)
180 tasm sep.tal           $(TASMOPT)
181
182 timing.trl4 : timing.h timing.c
183 pp    timing.c         $(PPOPT4)
184 tcx  timing.pp         $(TCXOPT4)
185 tasm timing.tal        $(TASMOPT)
186
187 vec_ops.trl4 : vec_ops.h vec_ops.c
188 pp    vec_ops.c        $(PPOPT4)
189 tcx  vec_ops.pp        $(TCXOPT4)
190 tasm vec_ops.tal       $(TASMOPT)
191
192
193 # ---------------      CREATE T800 VERSION OF THE LIBRARY    -----------------
194
195
```

217

```
196 $(T800LIBNAME).tll : $(TRL8FILES)
197 tlib $(T800LIBNAME) -b $(TLIB8FILES)
198
199 clargs.trl8 : clargs.h clargs.c
200 pp    clargs.c       $(PPOPT8)
201 tcx   clargs.pp      $(TCXOPT8)
202 tasm  clargs.tal     $(TASMOPT)
203
204 comm.trl8 : comm.h comm.c
205 pp    comm.c         $(PPOPT8)
206 tcx   comm.pp        $(TCXOPT8)
207 tasm  comm.tal       $(TASMOPT)
208
209 complex.trl8 : complex.h complex.c
210 pp    complex.c      $(PPOPT8)
211 tcx   complex.pp     $(TCXOPT8)
212 tasm  complex.tal    $(TASMOPT)
213
214 generate.trl8 : generate.h generate.c matrix.h memory.trl8
215 pp    generate.c     $(PPOPT8)
216 tcx   generate.pp    $(TCXOPT8)
217 tasm  generate.tal   $(TASMOPT)
218
219 hcube.trl8 : hcube.h hcube.c
220 pp    hcube.c        $(PPOPT8)
221 tcx   hcube.pp       $(TCXOPT8)
222 tasm  hcube.tal      $(TASMOPT)
223
224 machine.trl8 : machine.h machine.c
225 pp    machine.c      $(PPOPT8)
226 tcx   machine.pp     $(TCXOPT8)
227 tasm  machine.tal    $(TASMOPT)
228
229 mat_ops.trl8 : mat_ops.h mat_ops.c matrix.h
230 pp    mat_ops.c      $(PPOPT8)
231 tcx   mat_ops.pp     $(TCXOPT8)
232 tasm  mat_ops.tal    $(TASMOPT)
233
234 math.trl8 : math.h math.c
235 pp    math.c         $(PPOPT8)
236 tcx   math.pp        $(TCXOPT8)
237 tasm  math.tal       $(TASMOPT)
238
239 matrixio.trl8 : matrixio.h matrixio.c ascii.h matrix.h memory.trl8
240 pp    matrixio.c     $(PPOPT8)
241 tcx   matrixio.pp    $(TCXOPT8)
242 tasm  matrixio.tal   $(TASMOPT)
243
244 memory.trl8 : memory.h memory.c matrix.h
245 pp    memory.c       $(PPOPT8)
```

```
246 tcx   memory.pp      $(TCXOPT8)
247 tasm memory.tal      $(TASMOPT)
248
249 num_sys.trl8 : num_sys.h num_sys.c matrix.h
250 pp    num_sys.c      $(PPOPT8)
251 tcx   num_sys.pp     $(TCXOPT8)
252 tasm num_sys.tal     $(TASMOPT)
253
254 sep.trl8 : sep.h sep.c
255 pp    sep.c          $(PPOPT8)
256 tcx   sep.pp         $(TCXOPT8)
257 tasm sep.tal         $(TASMOPT)
258
259 timing.trl8 : timing.h timing.c
260 pp    timing.c       $(PPOPT8)
261 tcx   timing.pp      $(TCXOPT8)
262 tasm timing.tal      $(TASMOPT)
263
264 vec_ops.trl8 : vec_ops.c vec_ops.h
265 pp    vec_ops.c      $(PPOPT8)
266 tcx   vec_ops.pp     $(TCXOPT8)
267 tasm vec_ops.tal     $(TASMOPT)
268
269
270 # ----------------    COPY LIBRARIES TO TLIB DIRECTORY    ------------------
271
272 install4:
273 copy $(T414LIBNAME).tll $(TLIB)
274
275 install8:
276 copy $(T800LIBNAME).tll $(TLIB)
277
278
279 # ----------    COPY HEADER FILES TO STANDARD INCLUDE DIRECTORY    ----------
280
281 install_headers:
282 copy ascii.h    $(TLIB)\..\include
283 copy macros.h   $(TLIB)\..\include
284 copy matrix.h   $(TLIB)\..\include
285 copy clargs.h   $(TLIB)\..\include
286 copy comm.h     $(TLIB)\..\include
287 copy complex.h  $(TLIB)\..\include
288 copy generate.h $(TLIB)\..\include
289 copy hcube.h    $(TLIB)\..\include
290 copy machine.h  $(TLIB)\..\include
291 copy mat_ops.h  $(TLIB)\..\include
292 copy math.h     $(TLIB)\..\include
293 copy matrixio.h $(TLIB)\..\include
294 copy memory.h   $(TLIB)\..\include
295 copy num_sys.h  $(TLIB)\..\include
```

219

```
296 copy sep.h        $(TLIB)\..\include
297 copy timing.h     $(TLIB)\..\include
298 copy vec_ops.h    $(TLIB)\..\include
299
300
301
302
303
304 # ----------======== 3.) FILE MANAGEMENT & UTILITIES  ========----------
305 #
306 #   This section makes short work of a few useful/routine tasks.
307 #
308 # --------==========================================================--------
309
310
311 # ----------=============== 3.1) Intel iPSC/2  ===============----------
312 #
313
314 iclean:
315 rm $(OBJECTS)
316
317
318
319
320
321 # ----------============ 3.2) Logical Systems C  ============----------
322 #
323
324 tclean:
325 del *.pp
326 del *.tal
327 del *.trl
328
329
330 # EOF matlib.mak ------------------------------------------------------
```

B. NETWORK INFORMATION FILES

hyprcube.nif This Network Information File gives a fairly complete description of the hardware configuration used to perform the transputer work.

```
 1 ;     -------------======== NETWORK INFORMATION FILE  ====================
 2 ;
 3 ;     SOURCE  :   hyprcube.nif
 4 ;     VERSION :   1.1
 5 ;     DATE    :   09 September 1991
 6 ;     AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7 ;     USAGE   :   ld-net hyprcube
 8 ;     EDITING :   replace 'rootcode' with the code to run on the root
 9 ;                 replace 'nodecode' with appropriate code(s) for the nodes
10 ;
11 ;
12 ;     -------------=============== REFERENCES  =============================
13 ;
14 ;     [1] Inmos. IMS B012 User Guide and Reference Manual. Inmos Limited,
15 ;             1988, Fig. 26, p. 28.
16 ;
17 ;
18 ;     -------------=============== DESCRIPTION  ============================
19 ;
20 ;     Network Information File (NIF) used by Logical Systems C (version 89.1)
21 ;     LD-NET Network Loader.  This file prescribes the loading action to take
22 ;     place when the 'ld-net' command is given as in USAGE above.
23 ;
24 ;
25 ;     -------------========== HARDWARE PREREQUISITES  =================------
26 ;
27 ;     NOTE: There are three node numbering systems: the one created by Inmos'
28 ;     CHECK program, the Gray code labeling, and the NIF labeling.  Since all
29 ;     three will be used on occasion, I will prefix node numbers with a C, G,
30 ;     or N to identify which system I am using!
31 ;
32 ;     The IMS B004 and IMS B012 must be configured correctly. The B004's T414
33 ;     has link 0 connected to the host PC via a serial-to-parallel converter,
34 ;     link 1 connected to the IMS B012 PipeHead, link 2 connected to the T212
35 ;     [communications manager (not used here)] on the B012, and link 3
36 ;     connected to the IMS B012 PipeTail (see [1]).  By the way, link 2 from
37 ;     the B004 goes to the the ConfigUp slot just under the PipeHead slot
38 ;     (this connects it to the T212).  Finally, the B004's Down link must run
39 ;     to the B012's Up link.
40 ;
41 ;
42 ;     -------------==== SETTING THE C004 CROSSBAR SWITCHES  ====------------
43 ;
44 ;     Once you have connected the hardware in the fashion mentioned above,
45 ;     the system is ready to be transformed to a hypercube.  Three codes by
46 ;     Mike Esposito are used here:  t2.nif, root.tld, and switch.tld.  I have
47 ;     a batch file called 'makecube.bat' that performs a 'ld-net t2' also.
48 ;
49 ;     Mike's code passes instructions to the T212 on the B012; which, in-turn
50 ;     tells the C004's how to connect their switches.  After the code has
```

```
51 ;    executed, the (very specific) configuration that we are looking for
52 ;    will exist.  Specifically, the following (output from CHECK /R) is what
53 ;    this process gives us:
54 ;
55 ;          check 1.21
56 ;             # Part rate Mb Bt [  Link0  Link1  Link2  Link3 ]
57 ;             0 T414b-15 0.09 0 [  HOST    1:1    2:1    3:2 ]
58 ;             1 T800c-20 0.80 1 [   4:3    0:1    5:1    6:0 ]
59 ;             2 T2    -17 0.49 1 [  C004    0:2    ...    C004 ]
60 ;             3 T800c-20 0.80 2 [   7:3    8:2    0:3    9:0 ]
61 ;             4 T800c-20 0.76 3 [   9:3   10:2   11:1    1:0 ]
62 ;             5 T800d-20 0.90 1 [   8:3    1:2   10:1   12:0 ]
63 ;             6 T800d-20 0.76 0 [   1:3   12:2    7:1   11:0 ]
64 ;             7 T800d-20 0.76 3 [  13:3    6:2   14:1    3:0 ]
65 ;             8 T800d-20 0.90 2 [  14:3   15:2    3:1    5:0 ]
66 ;             9 T800c-20 0.77 0 [   3:3   13:2   15:1    4:0 ]
67 ;            10 T800d-20 0.90 2 [  16:3    5:2    4:1   15:0 ]
68 ;            11 T800d-20 0.90 1 [   6:3    4:2   16:1   13:0 ]
69 ;            12 T800d-20 0.77 0 [   5:3   16:2    6:1   14:0 ]
70 ;            13 T800d-20 0.77 3 [  11:3   17:2    9:1    7:0 ]
71 ;            14 T800c-20 0.90 1 [  12:3    7:2   17:1    8:0 ]
72 ;            15 T800c-20 0.90 2 [  10:3    9:2    8:1   17:0 ]
73 ;            16 T800c-20 0.76 3 [  17:3   11:2   12:1   10:0 ]
74 ;            17 T800d-20 0.88 2 [  15:3   14:2   13:1   16:0 ]
75 ;
76 ;    Here node C0 is the root transputer (on the IMS B004) and node C2 is
77 ;    the T212 (on the IMS B012).  The other sixteen nodes are the T800's
78 ;    that are used for the work.  A logical interconnection topology is
79 ;    described below.
80 ;
81 ;
82 ;    ---------------==================  TOPOLOGY   ==================---------------
83 ;
84 ;    The physical interconnection scheme described above is an actual 4-cube
85 ;    with one exception.  The root node (C0) is situated BETWEEN nodes C1
86 ;    and C3 (which would be connected directly in the usual 4-cube).  This
87 ;    gives us two 3-cubes:  one whose node labeling is G0xxx and the other,
88 ;    whose node labeling is G1xxx (where the xxx represents all permutations
89 ;    of 3-bits).  These are the usual three cubes, and they will exist if we
90 ;    define the node numbering/labeling correctly.
91 ;
92 ;
93 ;    ---------------==================  STRATEGY   ==================---------------
94 ;
95 ;    The node labeling established by the NIF is available via the variable
96 ;    _node_number (see <conc.h>) in source code.  Therefore, we would like a
97 ;    smart labeling scheme in the NIF file so that programming is easier.
98 ;    This, of course, is subject to the restriction that NIF labels begin
99 ;    with N1 and so on.
100 ;
```

```
101 ;   One such method would be to define a NIF labeling so that the Gray code
102 ;   label for a node would be (_node_number - 2).  In fact, this is
103 ;   possible and the adjacencies defined below allow us to realize this
104 ;   feature.  Below, node N0 is the host PC, node N1 is the root transputer
105 ;   (T414 on the B004), N2 through N17 correspond to G0 through G15 (the
106 ;   nodes of a 4-cube), and N18 is not used (but it's the T212).
107 ;
108 ;   --------------================================================--------------
109
110
111 host_server cio.exe;     (default)
112
113 ;              TRANSPUTER     RESET      DESCRIPTION OF LINK CONNECTIONS
114 ; NODE         LOADABLE       COMES      -------------------------------
115 ;  ID          CODE (.tld)    FROM:      LINK0    LINK1    LINK2    LINK3
116 ; ====         ===========    =====      =====    =====    =====    =====
117    1,         rootcode,      r0,          0,       2,       ,       10; B004
118    2,         nodecode,      r1,          4,       1,      3,        6; B012
119    3,         nodecode,      r2,         11,       2,      5,        7;
120    4,         nodecode,      r5,         12,       5,      8,        2;
121    5,         nodecode,      r3,          9,       3,      4,       13;
122    6,         nodecode,      r7,          2,       7,     14,        8;
123    7,         nodecode,      r9,          3,       9,      6,       15;
124    8,         nodecode,      r4,          6,       4,      9,       16;
125    9,         nodecode,      r8,         17,       8,      7,        5;
126   10,         nodecode,      r11,        14,      11,      1,       12;
127   11,         nodecode,      r13,        15,      13,     10,        3;
128   12,         nodecode,      r16,        10,      16,     13,        4;
129   13,         nodecode,      r12,         5,      12,     11,       17;
130   14,         nodecode,      r6,         16,       6,     15,       10;
131   15,         nodecode,      r14,         7,      14,     17,       11;
132   16,         nodecode,      r17,         8,      17,     12,       14;
133   17,         nodecode,      r15,        13,      15,     16,        9;
134 ; 18,         switch,        s1,          ,        1,       ,        ; T212
135
136
137 ; --------------============     EOF hyprcube.nif     ============--------------
```

224

C. STANDARD FILES

macros.h This header file gives several C macros that are used in other programs.

matrix.h This header file establishes the standard definition of a matrix.

```
 1 /* --------------=========   PROGRAM INFORMATION   ==========-------------
 2  *
 3  *  SOURCE  :   macros.h
 4  *  VERSION :   1.3
 5  *  DATE    :   14 September 1991
 6  *  AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  * --------------===================================================-------------
 9  */
10
11
12 #define MAX(x,y)                    (((x) > (y)) ? (x) : (y))
13
14 #define MIN(x,y)                    (((x) > (y)) ? (y) : (x))
15
16 #define POW2(n)                     ((1) << (n))
17
18
19
20
21
22 /* --------------==============   EOF macros.h   ====================---------- */
```

226

```
 1 /* ----------------========== PROGRAM INFORMATION ==========-------------
 2  *
 3  *  SOURCE  :   matrix.h
 4  *  VERSION :   2.0
 5  *  DATE    :   02 September 1991
 6  *  AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  *
 9  * ----------------=============== DESCRIPTION ================-------------
10  *
11  *  A header file for a family of functions designed to work with matrices.
12  *
13  * ----------------===================================================-------------
14  */
15
16 #include "complex.h"    /*  for Complex_Type  */
17
18
19
20 /* ----------------========== MANIFEST CONSTANTS ================---------- */
21
22
23 #define BASE_TEN            10
24 #define CURRENT             1
25 #ifndef EXIT_FAILURE
26 #define EXIT_FAILURE        1
27 #endif
28 #ifndef EXIT_SUCCESS
29 #define EXIT_SUCCESS        0
30 #endif
31 #define FAILURE             1
32 #define FALSE               0
33 #define LINE_LENGTH         80
34 #define MAX_NAME_LENGTH     80
35 #define NO                  0
36 #define OFF                 0
37 #define ON                  1
38 #define ONE_BYTE            1
39 #define ONE_MEMBER          1
40 #define PREVIOUS            0
41 #define SUCCESS             0
42 #define TRUE                1
43 #define TYPE_CHAR           0
44 #define TYPE_DOUBLE         1
45 #define TYPE_FLOAT          2
46 #define TYPE_INT            3
47 #define YES                 1
48
49
50
```

227

```
51 /* ----------------------------========      TYPE  DEFINITIONS      =====================-------- */
52
53
54 typedef struct {
55
56     char    *name;
57     int     rows,
58             cols;
59     double  **matrix;
60
61 } Matrix_Type;                      /*  default/standard is type double  */
62
63
64
65 typedef struct {
66
67     char         *name;
68     int          rows,
69                  cols;
70     Complex_Type **matrix;
71
72 } Complex_Matrix_Type;              /*  type Complex_Type is in complex.h  */
73
74
75
76 typedef struct {
77
78     char    *name;
79     int     rows,
80             cols;
81     double  **matrix;
82
83 } Double_Matrix_Type;
84
85
86
87 typedef struct {
88
89     char    *name;
90     int     rows,
91             cols;
92     float   **matrix;
93
94 } Float_Matrix_Type;
95
96
97
98 typedef struct {
99
100     char    *name;
```

228

```
      matrix.h

101     int     rows,
102             cols;
103     int     **matrix;
104
105 } Int_Matrix_Type;
106
107
108 /* -------------============ EOF matrix.h ===========------------- */
```

D. SOURCE CODE FILES

There is one header file and one (.c) source code file for each remaining member of the library, so the filename is given without the suffix.

allocate Memory allocation and management functions.

clargs For processing command–line arguments.

comm Communications functions for the hypercubes.

complex Complex numbers and operations.

epsilon Machine precision functions.

generate Matrix generation functions.

io Input/output (IO) functions.

mathx A small extension to the C math library.

num_sys Various number systems (binary, decimal, hexadecimal).

ops Matrix and vector operations.

timing Functions for timing.

Again, however, most of the source code has been omitted and only the header files remain. The singular exception is **complex.c** because this source contains an algorithm referenced earlier in the thesis.

```
 1  /* --------------------========== PROGRAM INFORMATION  =========-------------
 2   *
 3   *   SOURCE  :   allocate.h
 4   *   VERSION :   2.0
 5   *   DATE    :   09 September 1991
 6   *   AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7   *
 8   *
 9   * --------------------=========== DESCRIPTION  =========-------------
10   *
11   *        Declarations of functions associated with memory allocation.
12   *
13   *
14   * --------------------========== LIST OF FUNCTIONS  =========-------------
15   *
16   *   cmatalloc()
17   *   intvecalloc()
18   *   matalloc()
19   *
20   * --------------------------------------------------------------------
21   */
22
23
24
25
26
27  /* --------------------========== FUNCTION DECLARATION  =========-------------
28   *
29   *   PURPOSE:      This function performs the memory allocation for a matrix
30   *                structure (of the Complex_Matrix_Type) using the C function
31   *                calloc().  Additionally, it fills the "rows" and "cols"
32   *                fields of the matrix structure returned with the parameters
33   *                passed to the function.  If a structure is returned (see
34   *                "RETURNS"), then its "rows" and "cols" fields will be
35   *                filled with the correct values.  The structure type is
36   *                defined in "matrix.h".
37   *
38   *   INCLUDE:      "allocate.h"
39   *
40   *   CALLS:        calloc()
41   *
42   *   CALLED BY:
43   *
44   *   PARAMETERS:  int rows    the number of rows in the desired matrix
45   *                int cols    the number of columns in the desired matrix
46   *
47   *   RETURNS:     A pointer to the structure if successful; NULL otherwise.
48   *                The NULL case includes non-positive rows or cols in addi-
49   *                tion to the obvious allocation failure.
50   *
```

231

```
51  *   EXAMPLE:      Complex_Matrix_Type *A;
52  *
53  *                 A = cmatalloc(7, 7);
54  *
55  * ------------==============================================------------
56  */
57
58 #ifdef PROTOTYPE
59
60    Complex_Matrix_Type *cmatalloc(int rows, int cols);
61
62 #else
63
64    Complex_Matrix_Type *cmatalloc();
65
66 #endif
67
68
69
70
71
72 /* ------------========== FUNCTION DECLARATION =========------------
73  *
74  *   PURPOSE:     This function performs the memory allocation for a vector,
75  *               v, of num_elements integer elements.
76  *
77  *   INCLUDE:     "allocate.h"
78  *
79  *   CALLS:       calloc()
80  *
81  *   CALLED BY:
82  *
83  *   PARAMETERS: See PURPOSE.
84  *
85  *   RETURNS:     A pointer to the array if successful and NULL otherwise.
86  *
87  *   EXAMPLE:     int desired_size_of_v = 7,
88  *                   *v;
89  *
90  *               v = intvecalloc(desired_size_of_v);
91  *
92  * ------------==============================================------------
93  */
94
95
96 #ifdef PROTOTYPE
97
98    int *intvecalloc(int num_elements);
99
100 #else
```

232

```
101
102    int *intvecalloc();
103
104 #endif
105
106
107
108
109
110 /* -----------=========      FUNCTION DECLARATION    =====================
111  *
112  *   PURPOSE:     This function performs the memory allocation for a matrix
113  *                structure using the C function calloc().  Additionally, it
114  *                fills the "rows" and "cols" fields of the matrix structure
115  *                returned with the parameters passed in to the function.
116  *                If a structure is returned (see "RETURNS"), then its "rows"
117  *                and "cols" fields will be filled with the correct values.
118  *                The structure type is defined in "matrix.h".
119  *
120  *   INCLUDE:     "allocate.h"
121  *
122  *   CALLS:       calloc()
123  *
124  *   CALLED BY:
125  *
126  *   PARAMETERS: int rows    the number of rows in the desired matrix
127  *               int cols    the number of columns in the desired matrix
128  *
129  *   RETURNS:    A pointer to the structure if successful; NULL otherwise.
130  *               The NULL case includes non-positive rows or cols in addi-
131  *               tion to the obvious allocation failure.
132  *
133  *   EXAMPLE:    Double_Matrix_Type *A = matalloc(7, 7);
134  *
135  * -----------=============================================================
136  */
137
138 #ifdef PROTOTYPE
139
140    Double_Matrix_Type *matalloc(int rows, int cols);
141
142 #else
143
144    Double_Matrix_Type *matalloc();
145
146 #endif
147
148
149 /* -----------=========      EOF  allocate.h   ===================== */
```

233

```
1  /* -----------------========== PROGRAM INFORMATION ==========--------------
2   *
3   * SOURCE   :   clargs.h
4   * VERSION  :   1.5
5   * DATE     :   09 September 1991
6   * AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7   *
8   *
9   * -------------=============== DESCRIPTION ===============--------------
10  *
11  * This header file gives the declarations to accompany clargs.c.  These
12  * files provide a standard (if somewhat limited) way of handling command-
13  * line arguments.  The objective is to handle:
14  *
15  *    1.) Simple boolean arguments like "if -v exists, set verbose = TRUE".
16  *        We will call such an argument a 'simple' argument type.  This
17  *        type of argument can be recognized by the fact that it has no
18  *        sub-arguments (the sub-argument count, subargc == 0).
19  *
20  *    2.) Arguments with sub-arguments to be interpreted as numbers.  We
21  *        will this a 'complex' argument type.  Suppose that we want to set
22  *        int dim = 3 when the command line arguments contain "-d 3 ".
23  *        This case implies several requirements:
24  *
25  *        a.) First, we must know in advance how many sub-arguments the
26  *            argument has--we'll call this subargc (in this case we are
27  *            expecting one sub-argument, so the caller would have set
28  *            subargc = 1).
29  *
30  *        b.) Secondly, we must know how to interpret each sub-argument
31  *            [i.e., what type is the sub-argument?  Is it a double or long
32  *            (float and int can be handled by type casting)?]
33  *
34  *        We will call this kind of argument a complex argument type.  They
35  *        can be recognized as those with subargc > 0.
36  *
37  * Here is the strategy.  The user makes a list of valid command-line
38  * arguments by creating an array of pointers to structures of type
39  * Arg_Struct.  We will call this the option list, (Arg_Struct *) optv[].
40  * The code assumes that you can do something like this at the top of your
41  * source:
42  *
43  *        #define MAX_NUMBER_OF_ARGS        3
44  *
45  *        static Arg_Struct *optv[MAX_NUMBER_OF_ARGS];
46  *
47  * Let (int) optc, be the option count (number of options).  Every element
48  * in (pointed to by) the option list is a structure of type Arg_Struct
49  * defined below.  By using the standard C argc and argv; and by creating
50  * and passing optc and optv around, we can manipulate command-line
```

234

```
51  *  arguments just about however we want.  The next step is to understand
52  *  the structure.
53  *
54  *
55  * ------------============  LIST OF FUNCTIONS  ============------------
56  *
57  *  install_complex_arg()
58  *  install_simple_arg()
59  *  interpret_args()
60  *
61  * ------------=========================================================-----------
62  */
63
64
65
66 /* ------------============  MANIFEST CONSTANTS  ============------------ */
67
68
69 #ifndef EXIT_FAILURE
70 #define EXIT_FAILURE        1
71 #endif
72 #ifndef EXIT_SUCCESS
73 #define EXIT_SUCCESS        1
74 #endif
75 #ifndef FALSE
76 #define FALSE               0
77 #endif
78 #ifndef NULL
79 #define NULL                0
80 #endif
81 #ifndef SUCCESS
82 #define SUCCESS             0
83 #endif
84 #ifndef TRUE
85 #define TRUE                1
86 #endif
87
88
89 /*
90  * The maximum number of characters in an argument name, MAX_ARGLEN is a
91  * relatively arbitrary thing....make it whatever you want.  The DOUBLE
92  * and LONG manifest constants are assumed to be used for values of
93  * subargi (see the structure below).
94  */
95
96 #define MAX_ARGLEN          7
97 #define DOUBLE              0
98 #define LONG                1
99
100
```

235

```
101 /* ----------------------============  DATA  STRUCTURES  ============--------------
102  *
103  *   argname      The (string) name of a valid argument.  For instance, if
104  *                you want the simple argument "-v", then argname[] would be
105  *                "-v".  If you have a complex argument that will appear as
106  *                "-number 3 4.5 6.7", then argname will be "-number" and you
107  *                must use the sub-argument variables below to handle the
108  *                integer and two floating-point values.
109  *
110  *   subargc      Consider the "-number" example again.  There are three sub-
111  *                arguments (3, 4.5, and 6.7) so the sub-argument count would
112  *                be 3.
113  *
114  *   subargi[]    This array tells us how to interpret the subarguments.  For
115  *                instance, again using the "-number" example above, we would
116  *                set subargi[0] = LONG; subargi[1] = DOUBLE; and
117  *                subargi[2] = DOUBLE.
118  *
119  *   found        This should is initialized to FALSE.  The function
120  *                interpret_args() will set this field TRUE if the argname[]
121  *                appears on the command-line (in *argv[]).
122  *
123  *   dsa[]        This field is an array of double sub-arguments.
124  *
125  *   lsa[]        This field is an array of long sub-arguments.
126  *
127  * Consider the "-number" example again.  After argument resolution, we
128  * would find that dsa[0] is not defined since subargi[0] == LONG.
129  * However, we can use subargi[] to verify that subargi[1] and subargi[2]
130  * are DOUBLE.  Knowing this, we can safely presume that the values with
131  * CORRESPONDING index in dsa[] should be interpreted as doubles.  That
132  * is, dsa[1] will be a double value (4.5) and dsa[2] will also be a
133  * double (6.7).  In a similar manner, lsa[0] must be a long (3) and
134  * lsa[1] and lsa[2] are not defined.
135  *
136  * ------------------============================================--------------
137  */
138
139 typedef struct {
140
141     char    argname[MAX_ARGLEN];
142
143     int     subargc,              /* how many subarguments expected   */
144             *subargi,             /* how to interpret subarguments    */
145             found;                /* set TRUE if the argument is found */
146
147     double *dsa;                  /* double-valued sub-arguments       */
148     long   *lsa;                  /* long-valued sub-argument list     */
149
150 } Arg_Struct;
```

236

```
151
152
153  /* ------------------========        FUNCTION DECLARATION     =========-------------
154   *
155   *   PURPOSE:      To install a valid complex argument in the option list,
156   *                 optv[].
157   *
158   *   INCLUDE:      "clargs.h"
159   *
160   *   CALLS:        strcpy()
161   *
162   *   CALLED BY:
163   *
164   *   PARAMETERS:   int         index;
165   *                 Arg_Struct *optv[];
166   *                 const char *argname;
167   *                 int        *interpret,
168   *                            subargc;
169   *
170   *   The first three parameters are exactly like the corresponding ones for
171   *   install_simple_arg().  Additionally, for complex arguments, we need to
172   *   pass in instructions concerning how many sub-arguments there are (i.e.,
173   *   subargc) and how to interpret each.  The array interpret[] should be
174   *   filled with subargc elements when you call this function.  The elements
175   *   should only be valid ones (e.g., DOUBLE, LONG).
176   *
177   * ------------------========================================================-------------
178   */
179
180  #ifdef PROTOTYPE
181
182      void install_complex_arg(int index, Arg_Struct *optv[],
183                               const char *argname, int *interpret,
184                               int subargc);
185  #else
186
187      void install_complex_arg();
188
189  #endif
190
191
192
193
194
195  /* ------------------========        FUNCTION DECLARATION     =========-------------
196   *
197   *   PURPOSE:      To install a valid simple argument in the option list,
198   *                 optv[].
199   *
200   *   INCLUDE:      "clargs.h"
```

237

```
201  *
202  *  CALLS:      strcpy()
203  *
204  *  CALLED BY:
205  *
206  *  PARAMETERS: int        index;
207  *              Arg_Struct *optv[];
208  *              const char *argname;
209  *
210  *  The 'index' gives the location of the option in the option list,
211  *  optv[].  The function uses this index to install the argname at the
212  *  proper location in optv[].  For instance, set this variable to zero for
213  *  the first option in the list.  Normal C indexing convention applies;
214  *  namely, 0 <= index < MAX_NUMBER_OF_ARGS.  The 'argname' is the string
215  *  that you want recognized as a valid argument.  For instance, suppose
216  *  that you want a timing argument to be recognized whenever "-t" appears
217  *  on the command line.  Then you would supply "-t" in this place.
218  *
219  *  -------------=================================================-----------
220  */
221
222 #ifdef PROTOTYPE
223
224      void install_simple_arg(int index, Arg_Struct *optv[],
225                                         const char *argname);
226 #else
227
228      void install_simple_arg();
229
230 #endif
231
232
233
234
235
236 /* -------------========== FUNCTION DECLARATION  ==========-------------
237  *
238  *  PURPOSE:    Once the user has defined an appropriate option list,
239  *              optv[], with optc options, this function parses the
240  *  command-line arguments (as given by argc and argv) and fills the
241  *  *optv[] structures appropriately.  For instance every valid (exists in
242  *  optv ==> valid) argument that appears on the command line will result
243  *  in the corresponding optv structure's 'found' field being set to TRUE.
244  *  The function also interprets sub-arguments and fills dsa[] and/or lsa[]
245  *  accordingly.  It assumes that the caller has established the desired
246  *  argname's, subargc's, and subargi's.
247  *
248  *  INCLUDE:    "clargs.h"
249  *
250  *  CALLS:      printf()
```

238

```
251  *                 strcmp()
252  *                 strtod()
253  *                 strtol()
254  *
255  *  CALLED BY:
256  *
257  *  PARAMETERS: As described in PURPOSE.
258  *
259  * --------------------------------------------------------------------
260  */
261
262
263 #ifdef PROTOTYPE
264
265    void interpret_args(int argc, char **argv, int optc, Arg_Struct **optv);
266
267 #else
268
269    void interpret_args();
270
271 #endif
272
273
274 /* ----------------------------===== EOF clargs.h  ============---------------- */
```

```
 1  /* ----------------============  PROGRAM INFORMATION  ==========-------------
 2   *
 3   *  SOURCE   :  comm.h
 4   *  VERSION  :  2.5
 5   *  DATE     :  14 September 1991
 6   *  AUTHOR   :  Jonathan E. Hartman, U. S. Naval Postgraduate School
 7   *
 8   *
 9   * ------------============  DESCRIPTION  ============-------------
10   *
11   *  This header file gives manifest constants and function specifications
12   *  for comm.c.  These files contain communication (and related) functions
13   *  for a normal hypercube topology and a hybrid topology.  Unfortunately
14   *  the code is a bit busy with #ifdef's, but the purpose of these files is
15   *  to make hypercubes a little more transparent.  This makes the comm.h
16   *  and comm.c files a bit hard to read, but you should be able to recoup
17   *  this loss when it comes time to write a particular application.
18   *
19   *
20   * ------------============  TOPOLOGIES  ============-------------
21   *
22   *  The functions specified below have been designed to work on three very
23   *  different machines.  First, the Intel iPSC/2 with a normal hypercube of
24   *  order 0, 1, 2, or 3 is handled.  A normal hypercube of transputers is
25   *  next on the list (also order 0, 1, 2, or 3).  Finally, there is a
26   *  hybrid topology of transputers that is handled.  The normal hypercubes
27   *  need almost no introduction.  We have a host or root processor/program
28   *  together with programs running on the nodes.  I will use host and root
29   *  interchangeably here, although 'host' is properly associated with the
30   *  Intel machine and 'root' is the more correct/descriptive term when the
31   *  subject is transputer networks.  The hybrid topology deserves a more
32   *  careful introduction.
33   *
34   *  The hybrid topology is a network of Inmos transputers (PC host with an
35   *  IMS B004 board and a T414 linked to sixteen T800 processors on an IMS
36   *  B012 board) arranged so that the 'root' is situated between nodes zero
37   *  and eight of a 4-cube.  This means that nodes 0 and 8 are NOT directly
38   *  connected.  The functions made for this topology compensate for this
39   *  situation.  Instead of trying to describe each function, I will simply
40   *  remark that the most natural way to treat this problem is (more-or-
41   *  less) as two 3-cubes attached to the root.  A more careful description
42   *  of how each problem is handled may be found in the code for the parti-
43   *  cular function.
44   *
45   *  In summary, the transputer portions of the code depend upon: (1) a very
46   *  specific hardware configuration, (2) the appropriate NIF file to
47   *  support the usual Gray code in a convenient way
48   *
49   *                    [ mynode() == _node_number - 2 ],
50   *
```

240

```
51  *  and (3) a particular link arrangement like that can be created by Mike
52  *  Esposito's t2.nif, root.tld, and switch.tld.
53  *
54  *  DETAILS:  Look for additional details in hyprcube.nif.
55  *
56  *
57  *  ------------=============  PREREQUISITES  ========================
58  *
59  *  Before using any of the functions involving send() or receive(), the
60  *  host (or root) program must initialize_hypercube().  For transputer
61  *  applications, EACH of the NODES must initialize_hypercube() too, and
62  *  you need to be sure that a hypercube exists in hardware and that your
63  *  NIF describes a hypercube with the usual Gray code.  You must define
64  *  the global variables {Channel *ic[], *oc[];} because the code depends
65  *  upon their existence.  Both of these vectors must be of length
66  *  (cubesize+1) as described in the preface to initialize_hypercube().
67  *
68  *  The cubesize and dimension that you use with the transputer implementa-
69  *  tion determine the cube.  Even though you actually have sixteen T800's
70  *  in the cube, the cubesize and dimension that you use will determine the
71  *  portion that actually gets used.  Note that both the usual hypercube
72  *  and the hybrid 4-cube are built upon the same hardware and link setup.
73  *  Many of the functions declared below DEPEND upon the proper call to the
74  *  initialize_hypercube() function.  To avoid difficulty, observe the
75  *  guidelines given with this function!  Additionally, in the transputer
76  *  case, you will need to make sure that you include <conc.h>.
77  *
78  *
79  *  ------------=============  LIST OF FUNCTIONS  ========================
80  *
81  *  coalesce()
82  *  cubecast()
83  *  cubecast_from()
84  *  directional_exchange()
85  *  directional_receive()
86  *  directional_send()
87  *  hamming_distance()
88  *  initialize_hypercube()
89  *  least_dimension()
90  *  link_number()
91  *  linkin()
92  *  linkout()
93  *  receive()
94  *  send()
95  *  submit()
96  *
97  *  ------------------====================================================
98  */
99
100
```

```
101  /* --------------====    MACROS  &  MANIFEST  CONSTANTS   ====------------ */
102
103  #ifdef TRANSPUTER
104
105  #define myhost()          -1
106  #define mynode()          (_node_number - 2)      /*  depends upon <conc.h> */
107
108  #else   /*  iPSC/2  */
109
110  #define ALL_NODES         -1
111  #define ALL_PIDS          -1
112  #define ANY_NODE           0      /*  for receive(from any node, ... )     */
113  #define ANY_TYPE          -1      /*  first non-force-type message         */
114  #define ARBITRARY_TYPE     0      /*  don't care                           */
115  #define KEEP_TIL_RELCUBE   1      /*  for getcube()                        */
116  #define NODE_PID           0      /*  arbitrary ... don't care             */
117  #ifndef NULL
118  #define NULL               0
119  #endif
120
121  #endif
122
123
124  #ifndef FALSE
125  #define FALSE              0
126  #endif
127
128  #ifndef TRUE
129  #define TRUE               1
130  #endif
131
132
133  /* ----------------========    FUNCTION DECLARATION    ========-------------
134   *
135   *   PURPOSE:    This function performs the first step in the opposite of
136   *               the cubecast() function.  That is, this one is used when
137   *   you want to collect information from the nodes in 'higher dimensions'
138   *   of the hypercube at the current node. You may want to perform some work
139   *   before forwarding this information down to the next lower dimension, so
140   *   the submit() function is given separately.
141   *
142   *   Like the other functions in this file, coalesce() performs a somewhat
143   *   different task when executed in the hybrid 4-cube, so first we will
144   *   discuss the usual hypercubes.  coalesce() is a null operation when
145   *   called from in the highest dimension [ if least_dimension(node) is
146   *   equal to dim ].  Otherwise it performs the communication to receive
147   *   from higher dimensions (i.e., neighbors with larger node numbers).  If
148   *   it is called from the host/root, it attempts to receive() from node
149   *   zero.
150   *
```

242

```
151  *  The coalesce() and submit() functions must be balanced properly across
152  *  the nodes.  The CALLER must take the necessary steps to be sure that
153  *  buf is large enough to hold ((dim - least_dimension(node)) * len)
154  *  bytes.  That is, there will be (dim - least_dimension(node)) copies of
155  *  the message accumulated at the calling node.
156  *
157  *  There are several exceptions in the hybrid 4-cube topology.  Since the
158  *  root is connected to nodes 0000 and 1000, it must make sure that buf
159  *  can hold 2 copies of length, len.  Then you should think of nodes 0xxx
160  *  as one 3-cube and nodes 1xxx as another (more-or-less separate) 3-cube.
161  *  That is, there will be no exchanges in the 1xxx direction between them.
162  *  To determine the size of buf at any node, use the following formulae:
163  *
164  *      (3 - least_dimension(node)) * len,          Nodes 0xxx
165  *
166  *      (3 - least_dimension(node - 8)) * len,      Nodes 1xxx
167  *
168  *  CAUTIONS:   If you fail to allocate enough space for buf, you may find
169  *              that your program doesn't work.
170  *
171  *              The transputer implementation depends upon the parameter
172  *              'type' being set equal to cubesize.
173  *
174  *  PREREQUISITE:  initialize_hypercube()
175  *
176  *  INCLUDE:    <conc.h>                    (Logical Systems C, version 89.1)
177  *              "comm.h"
178  *
179  *  CALLS:      least_dimension()
180  *              myhost()                    (macro given above)
181  *              pow2()                      "mathx.h"
182  *              receive()
183  *
184  *  CALLED BY:
185  *
186  *  EXAMPLE:    Suppose we are 'at' node 0 and we want to coalesce() copies
187  *              of some object from all of the appropriate nodes.  Let the
188  *  object be of size 'len' bytes.  For concreteness, let the topology be a
189  *  hypercube of order 3 (i.e., dim == 3). We would allocate a large enough
190  *  buf to hold (dim * len) bytes, since least_dimension(0) == 0.  That is,
191  *  node 0 will be receiving from all neighbors whose least_dimension() is
192  *  greater [in this case, that is ALL of its neighbors]; namely, 1, 2, and
193  *  4.  After the call, we would find the data from node 1 in the first len
194  *  bytes of buf; the data from 2 in the middle len bytes of buf; and the
195  *  data from 4 in the final len bytes of buf.  The function is treated as
196  *  a multiple receive(), in increasing origin order, from the appropriate
197  *  neighbors.
198  *
199  *  PARAMETERS:
200  *
```

243

```
201  *        int   node    the coalesce()ing (receiving) node
202  *        int   dim     the dimension of the hypercube
203  *        char *buf     a pointer to the beginning of the buffer where you want
204  *                      the message placed.
205  *        long len      the number of bytes to be received from EACH node in
206  *                      the next higher dimension that will be submit()ing.
207  *        long type     the type of the message (iPSC/2 applications only), or
208  *                      cubesize in the transputer case.
209  *
210  * --------------============================================----------------
211  */
212 #ifdef PROTOTYPE
213
214     void coalesce(int node, int dim, char *buf, long len, long type);
215
216 #else
217
218     void coalesce(/* int node, int dim, char *buf, long len, long type */);
219
220 #endif
221
222
223
224 /* ---------------========= FUNCTION DECLARATION  =========---------------
225  *
226  * PURPOSE:   This function is called from the root/host and all nodes to
227  *                execute a broadcast to all p nodes.  The host/root sends to
228  * node zero to start the process off.  Let lg(n) denote log_2(n).  This
229  * function performs the communication in lg(p) steps.  For instance, node
230  * zero receives from the host in what we'll call stage zero.  Then, in
231  * stage 1, node 0 passes the message to node 1.  In stage 2, node 0 sends
232  * the message to node 2 and node 1 sends it to node 3.  In stage three,
233  * nodes 0, 1, 2, and 3 each send the message to nodes 4, 5, 6, and 7
234  * (respectively).
235  *
236  * Then, in general, in stage i, the message moves into the ith dimension.
237  * If you prefer, you can think of a pointer starting (after the message
238  * arrives at node 0) at the rightmost bit (LSB) and indicating the direc-
239  * tion for the next transmission.  The pointer moves left until it
240  * reaches the MSB.  This is the final stage of the cubecast().
241  *
242  * The hybrid 4-cube is implemented by sending the message from the root
243  * to nodes 0 and 8 first.  Then node 0 performs the usual cubecast for
244  * the nodes that appear in the usual 3-cube.  Node 8 mirrors this action,
245  * filling the other three-cube with labels like 1xxx.
246  *
247  * In all cases, buf is filled with an initial receive() from the proper
248  * node, and then it is used in retransmissions to other nodes.  In any
249  * event, buf holds the message after execution.
250  *
```

```
251  *   CAUTION:     The transputer implementation depends upon the parameter
252  *                'type' being set equal to cubesize.
253  *
254  *   PREREQUISITE:  initialize_hypercube()
255  *
256  *   INCLUDE:     <conc.h>                    (Logical Systems C, version 89.1)
257  *                "comm.h"
258  *
259  *   CALLS:       least_dimension()
260  *                MIN()                       (macro from macros.h)
261  *                myhost()                    (macro from above)
262  *                pow2()                      "mathx.h"
263  *                receive()
264  *                send()
265  *
266  *   CALLED BY:
267  *
268  *   PARAMETERS:
269  *
270  *       int  node    the sending node
271  *       int  dim     the dimension of the hypercube
272  *       char *buf    a pointer to the head of the message
273  *       long len     the number of bytes to be passed
274  *       long type    the type of the message (iPSC/2 applications only), or
275  *                    cubesize in the transputer case.
276  *
277  *   -------------==============================================================
278  */
279
280
281 #ifdef PROTOTYPE
282
283     void cubecast(int node, int dim, char *buf, long len, long type);
284
285 #else
286
287     void cubecast(/* int node, int dim, char *buf, long len, long type */);
288
289 #endif
290
291
292
293
294
295 /* -------------========= FUNCTION DECLARATION =======------------------
296  *
297  *   PURPOSE:     This function is similar to cubecast() but more general.
298  *               Here we do not assume that the message starts at the host
299  *   or at node zero; it may start at any general source node, src. In fact,
300  *   it may NOT be called from the root/host (use cubecast() in that case).
```

```
301  *   If dim is the order of the hypercube, then src goes through dim stages,
302  *   passing the message to its neighbors.  The sequence is defined by an
303  *   XOR operation that starts at bit 1 of src and moves up through bit dim.
304  *   For instance, suppose src == 5 == 101b in the 3-cube (dim == 3).  Then
305  *   src will first send to (101 XOR 001) == node 4, next to (101 XOR 010)
306  *   == node 7, and finally to (101 XOR 100) == node 1.  Meanwhile, any time
307  *   that a non-source node gets the message, he begins the same process,
308  *   but only picks it up at the appropriate stage (the one after the stage
309  *   in which he received the message).
310  *
311  *   PREREQUISITE:  initialize_hypercube()
312  *
313  *   INCLUDE:     <conc.h>                  (Logical Systems C, version 89.1)
314  *                "comm.h"
315  *
316  *   CALLS:       directional_receive()
317  *               directional_send()
318  *               free()
319  *               least_dimension()
320  *               malloc()
321  *               pow2()                    "mathx.h"
322  *               receive()
323  *               send()
324  *               sizeof()
325  *
326  *   CALLED BY:
327  *
328  *   PARAMETERS:
329  *
330  *       int   src    the source
331  *       int   node   the number of the node calling this function
332  *       int   dim    the dimension of the hypercube
333  *       char *buf    a pointer to the head of the message
334  *       long  len    the number of bytes to be passed
335  *
336  * -------------================================================-------------
337  */
338
339
340 #ifdef PROTOTYPE
341
342     void cubecast_from(int src, int node, int dim, char *buf, long len);
343
344 #else
345
346     void cubecast_from();
347
348 #endif
349
350
```

246

```
351 /* -------------========    FUNCTION DECLARATION    =========----------------
352  *
353  *    PURPOSE:     To perform an exchange along a prescribed direction.  The
354  *                 direction is given as an integer in {1, 2, 4, 8,...,2^dim}.
355  *    This is because the direction is really a bit mask for the Gray-coded
356  *    node numbers.  For instance, if you perform a directional_exchange()
357  *    from node == 3 == 011 in the 3-cube along direction == 4 == 100, this
358  *    is the same as performing a coordinated send() and receive() combina-
359  *    tion with node (011 XOR 100 == 111 == 7).  Care is taken to make sure
360  *    that deadlock does not occur.
361  *
362  *    PREREQUISITE:  initialize_hypercube()
363  *
364  *    INCLUDE:     <conc.h>                (Logical Systems C, version 89.1)
365  *                 "comm.h"
366  *
367  *    CALLS:       pow2()                  "mathx.h"
368  *                 receive()
369  *                 send()
370  *
371  *    CALLED BY:
372  *
373  *    PARAMETERS:
374  *
375  *        int   node       the number of the node calling this function
376  *        int   dim        the dimension of the hypercube
377  *        int   direction  as described above (1, 2, 4, 8, etc.)
378  *        char *ibuf       a pointer to the head of the incoming message
379  *        char *obuf       a pointer to the head of the outgoing message
380  *        long  len        the number of bytes to be passed
381  *
382  * -------------=================================================----------------
383  */
384
385
386 #ifdef PROTOTYPE
387
388     void directional_exchange(int node, int dim, int direction,
389                               char *ibuf, char *obuf, long len);
390
391 #else
392
393     void directional_exchange();
394
395 #endif
396
397
398
399
400
```

247

```
401 /* -------------========          FUNCTION DECLARATION    ========---------------
402  *
403  *  PURPOSE:     To receive from a prescribed direction.  The direction is
404  *               as described in directional_exchange() above.
405  *
406  *  PREREQUISITE:  initialize_hypercube()
407  *
408  *  INCLUDE:     <conc.h>                  (Logical Systems C, version 89.1)
409  *               "comm.h"
410  *
411  *  CALLS:       pow2()                    "mathx.h"
412  *               receive()
413  *
414  *  CALLED BY:
415  *
416  *  PARAMETERS:
417  *
418  *      int  node       the number of the node calling this function
419  *      int  dim        the dimension of the hypercube
420  *      int  direction  direction to receive from
421  *      char *buf       a pointer to the head of the message
422  *      long len        the number of bytes to be passed
423  *
424  * -------------========================================================---------
425  */
426
427
428 #ifdef PROTOTYPE
429
430     void directional_receive(int node, int dim, int direction,
431                                                   char *buf, long len);
432
433 #else
434
435     void directional_receive();
436
437 #endif
438
439
440
441 /* -------------========          FUNCTION DECLARATION    ========---------------
442  *
443  *  PURPOSE:     To send in a prescribed direction.  The direction is as
444  *               described in directional_exchange() above.
445  *
446  *  PREREQUISITE:  initialize_hypercube()
447  *
448  *  INCLUDE:     <conc.h>                  (Logical Systems C, version 89.1)
449  *               "comm.h"
450  *
```

248

```
451  *  CALLS:        pow2()                    "mathx.h"
452  *                send()
453  *
454  *  CALLED BY:
455  *
456  *  PARAMETERS:
457  *
458  *      int  node       the number of the node calling this function
459  *      int  dim        the dimension of the hypercube
460  *      int  direction  direction to send to
461  *      char *buf       a pointer to the head of the message
462  *      long len        the number of bytes to be passed
463  *
464  *  -------------===============================================----------------
465  */
466
467
468  #ifdef PROTOTYPE
469
470      void directional_send(int node, int dim, int direction,
471                                              char *buf, long len);
472
473  #else
474
475      void directional_send();
476
477  #endif
478
479
480
481
482
483  /* --------------=========    FUNCTION DECLARATION    =====================
484   *
485   *  PURPOSE:     To give the Hamming distance between i and j.
486   *
487   *  INCLUDE:     "comm.h"
488   *
489   *  CALLS:       sizeof()
490   *
491   *  CALLED BY:
492   *
493   *  PARAMETERS: int  i, j   the numbers
494   *
495   *  RETURNS:     (int) the Hamming distance(i,j).  That is, the number of
496   *               ones in the binary exclusive OR (i XOR j).
497   *
498   *  -------------===============================================----------------
499   */
500
```

249

```
501 #ifdef PROTOTYPE
502
503     int hamming_distance(int i, int j);
504
505 #else
506
507     int hamming_distance(/* int i, int j */);
508
509 #endif
510
511
512 /* ------------========        FUNCTION  DECLARATION   =========------------
513  *
514  * PURPOSE:    The initialize_hypercube() function creates the hypercube
515  *             and performs the required setup for communications.  It
516  * must be completed before you expect to communicate.  On the iPSC/2,
517  * ONLY the host code should call this function.  For transputer implemen-
518  * tations every node should call it (in addition to the root node).  This
519  * is prerequisite to most of the other functions in this file.  The basic
520  * requirements for this function are so different (machine dependent)
521  * that there are two versions:  one for the transputers and one for the
522  * iPSC/2 machine.
523  *
524  * INCLUDE:    "comm.h"
525  *
526  * CALLS:      attachcube()            (Intel iPSC/2 C Library)
527  *             calloc()
528  *             free()
529  *             getcube()               (Intel iPSC/2 C Library)
530  *             linkin()
531  *             linkout()
532  *             load()                  (Intel iPSC/2 C Library)
533  *             malloc()
534  *             printf()
535  *             setpid()                (Intel iPSC/2 C Library)
536  *             sizeof()
537  *             strcpy()
538  *
539  * CALLED BY:
540  *
541  * PARAMETERS: In both cases, the desired dimension of the hypercube is
542  *             passed in as the first argument.  After this, the functions
543  *             are quite different.
544  *
545  *       (1) iPSC/2 -------------------------------------------------------
546  *
547  *           char *nodecode  A pointer to the filename of the nodecode is
548  *                           required so that the function can load the node
549  *                           program.
550  *
```

250

```
551   *        (2) transputers ----------------------------------------------
552   *
553   *            Channel *ic[(CUBESIZE + 1)]  This is the incoming channel list.
554   *            You must declare it globally.  Let CUBESIZE be the number of
555   *            transputers in the hypercube.  Then ic[] is a vector of length
556   *            (CUBESIZE + 1).  The indexing is such that (ic[n] == C), where
557   *            n is some neighbor and C is the incoming Channel* from n.  For
558   *            instance, if node k finds that ic[n] == LINK1IN then node k
559   *            knows to receive messages from node n via LINK1IN.  The element
560   *            ic[CUBESIZE] holds the channel for the root node (if any).
561   *            ic[n] == NULL means that there is no connection to node n.
562   *
563   *            Channel *oc[(CUBESIZE + 1)] is the outgoing channel list.  It
564   *            is completely analogous to ic[] except that it will hold
565   *            LINK0OUT, LINK1OUT, LINK2OUT, or LINK3OUT for the appropriate
566   *            node index.  Your only obligation is to define these lists as
567   *            globals in the manner shown.  The Channel pointer elements will
568   *            be filled in by initialize_hypercube().
569   *
570   *   RETURNS:    The iPSC/2 version of the function returns a pointer to the
571   *               name of the cube.  In the transputer environment, the cube-
572   *               name has no meaning, so a void function suffices.  For the
573   *               transputer environment, the single most important task that
574   *               initialize_hypercube() performs is the filling of ic[] and
575   *               oc[].  These vectors are used by most of the other communi-
576   *               cations functions.
577   *
578   * ------------------===============================================---------------
579   */
580
581
582 #ifdef TRANSPUTER
583
584     void initialize_hypercube(int dim);
585
586 #else
587
588     char *initialize_hypercube(/* int dim, char *nodecode */);
589
590 #endif
591
592
593
594
595 /* ------------------=========    FUNCTION DECLARATION    =========--------------
596   *
597   *   PURPOSE:    This function, called from any node in the hypercube,
598   *               returns the dimension of the smallest hypercube containing
599   *               that node.
600   *
```

```
601   *   INCLUDE:      "comm.h"
602   *
603   *   CALLS:        pow2()                        "mathx.h"
604   *
605   *   CALLED BY:
606   *
607   *   PARAMETERS: int node        the inquiring node
608   *
609   *   RETURNS:      For an n-cube containing P==2^(n) processors, this function
610   *                 is designed to work for nodes numbered 0 through (P-1).  If
611   *   the function is called from the root (host) node, there is no guarantee
612   *   as to the returned value.  If it is called by a valid node, it will
613   *   return the dimension of the smallest hypercube containing that node
614   *   number.  For instance least_dimension(0) == 0, least_dimension(1) == 1,
615   *   least_dimension(2) == 2, least_dimension(3) == 2, and least_dimension
616   *   (8) == 4.
617   *
618   *   -------------=============================================----------------
619   */
620
621
622  #ifdef PROTOTYPE
623
624      int least_dimension(int node);
625
626  #else
627
628      int least_dimension(/* int node */);
629
630  #endif
631
632
633
634
635  /* ------------------========    FUNCTION DECLARATIONS   ========--------------
636   *
637   *   PURPOSE:     The receive() and send() functions declared below provide
638   *                communication to (from) a buffer pointed to by buf.  The
639   *   volume of material to send (receive) is indicated in bytes by the len
640   *   argument.  The destination (origin) is given by the first argument,
641   *   using a valid node number.  Suppose you have an n-cube established upon
642   *   a system with p == (2^n) node processors.  Then you should refer to the
643   *   nodes of the hypercube by their node number, which is a Gray coded
644   *   value in the range [ 0, (p-1) ].  If you are at the root, of course,
645   *   you may not communicate with the root (at least not with these func-
646   *   tions); but if you are at one of the nodes of the hypercube, you may
647   *   communicate with the root by using myhost() as the origin (or destina-
648   *   tion) of your message.  The macro given above makes myhost() available
649   *   on the transputers.
650   *
```

252

```
651  *  Transputers or iPSC/2?  The type parameter is only used in the implied
652  *  sense with the iPSC/2 implementation [ it becomes type or typesel for
653  *  csend() or crecv() ].  For transputer implementations, type MUST BE set
654  *  equal to the number of nodes in the hypercube (e.g., p in the example
655  *  above).  I have called this 'cubesize' in most of my references.
656  *
657  *  PREREQUISITE:  initialize_hypercube()
658  *
659  *  INCLUDE:      <conc.h>                (Logical Systems C, version 89.1)
660  *               "comm.h"
661  *
662  *  CALLS:        ChanIn()                (Logical Systems C, version 89.1)
663  *               ChanOut()
664  *               crecv()                 (Intel iPSC/2 C Library)
665  *               csend()
666  *
667  *  CALLED BY:
668  *
669  *  ---------------------============== CAUTION  ==============----------------
670  *
671  *  Make sure type == cubesize in the transputer case (see the note above)!
672  *
673  *  ---------------============================================================
674  */
675  #ifdef PROTOTYPE
676
677      void receive(int origin,   char *buf, long len, long type);
678
679      void send(int destination, char *buf, long len, long type);
680
681  #else
682
683      void receive(/* int origin,   char *buf, long len, long type */);
684
685      void send(/* int destination, char *buf, long len, long type */);
686
687  #endif
688
689
690  /* ---------------========== FUNCTION DECLARATION  ==========---------------
691  *
692  *  PURPOSE:    This function is called from the nodes to submit a message
693  *             to the next lower dimension.  If it is called from the host
694  *  (root) it has no effect.  When it is called from node zero, the trans-
695  *  mission is directed to the root/host.  When called from any other node,
696  *  the information in buf is passed to the proper node in the next lower
697  *  dimension.  The lower dimension must have an accepting coalesce() or
698  *  other receiving function [ coalesce() and submit() are meant to be used
699  *  in a balanced fashion, where each submit() or group of submit()'s in
700  *  one dimension is matched by a coalesce() in the next lower dimension ].
```

253

```
701   *
702   *  PREREQUISITE: initialize_hypercube()
703   *
704   *  INCLUDE:      <conc.h>                    (Logical Systems C, version 89.1)
705   *               "comm.h"
706   *
707   *  CALLS:        least_dimension()
708   *               pow2()                       "mathx.h"
709   *               send()
710   *
711   *  CALLED BY:
712   *
713   *  EXCEPTIONS: Again, we have the hybrid hypercube in the transputer case
714   *              (see many comments above).  The general rule is changed in
715   *  this case since node 1 submit()s to the root and not node 0.  This is
716   *  the only change.
717   *
718   *  SPECIFICS:  If you need to determine exactly where a submit() will go,
719   *              you can figure it out in the following manner [ with the
720   *  obvious EXCEPTIONS (the previous paragraph) ]  ....
721   *
722   *  Suppose you are 'at' node i in an n-cube (p processors = 2^n).  You
723   *  must submit() information to the (unique) node, j, that satisfies two
724   *  requirements:
725   *
726   *      (1) hamming_distance(i, j) == 1
727   *
728   *      (2) least_dimension(i) == (least_dimension(j) + 1)
729   *
730   *  So, for instance, consider a 4-cube where i == 12.  It should be fairly
731   *  easy to see that j will be node 4.  This is because these two nodes are
732   *  adjacent and they are one dimension apart in the cube (i.e., node 4
733   *  first appears in a 3-cube and node 12 first appears in a 4-cube).
734   *
735   *  PARAMETERS:
736   *
737   *      int  node    the sending node
738   *      int  dim     the dimension of the hypercube
739   *      char *buf    a pointer to the head of the message
740   *      long len     the number of bytes to be passed
741   *      long type    the type of the message (iPSC/2 applications only), or
742   *                   cubesize in the transputer case.
743   *
744   *  ------------------===============================================----------------
745   */
746
747
748  #ifdef PROTOTYPE
749
750      void submit(int node, int dim, char *buf, long len, long type);
```

254

```
751
752 #else
753
754     void submit(/* int node, int dim, char *buf, long len, long type */);
755
756 #endif
757
758
759 /* ------------============== EOF comm.h ================------------ */
```

```
1  /* ---------------========== PROGRAM INFORMATION ==========--------------
2  *
3  * SOURCE  :   complex.h
4  * VERSION :   1.6
5  * DATE    :   09 September 1991
6  * AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7  *
8  * ------------=============== REFERENCES ==================-------------
9  *
10 * [1] Goldberg, David. ''What Every Computer Scientist Should Know About
11 *         Floating-Point Arithmetic''. ACM Computing Surveys, Vol. 23,
12 *         No. 1, March 1991.
13 *
14 *
15 * -------------============== DESCRIPTION ================-------------
16 *
17 * This file contains the definition of Complex_Type and declarations of
18 * functions that perform operations with complex numbers:
19 *
20 *     cadd()
21 *     cdiv()
22 *     cmul()
23 *     csub()
24 *     Im()
25 *     Re()
26 *
27 * --------------==================================================--------------
28 */
29
30
31
32
33
34 /* -------------========== TYPE DEFINITION ==========------------ */
35
36
37 typedef struct {
38
39     double  x,      /*  real part      */
40             y;      /*  imaginary part */
41
42 } Complex_Type;
43
44
45
46
47
48 /* -------------========== FUNCTION DECLARATION ==========------------
49 *
50 * PURPOSE:    To add two complex numbers, z1 and z2, and place their sum
```

```
51  *              in the Complex_Type '*sum'.
52  *
53  *  INCLUDE:     "complex.h"
54  *
55  *  PARAMETERS: The parameters give the two operands z1 and z2, and a
56  *              pointer to the result, sum.
57  *
58  *  EXAMPLE:     Complex_Type z1, z2, z3;
59  *
60  *              cadd(z1, z2, &z3);
61  *
62  * ------------------==========================================================
63  */
64
65 #ifdef PROTOTYPE
66
67    void cadd(Complex_Type z1, Complex_Type z2, Complex_Type *sum);
68
69 #else
70
71    void cadd();
72
73 #endif
74
75
76
77
78
79 /* ------------------==========   FUNCTION DECLARATION   =====================
80  *
81  *  PURPOSE:     To divide two complex numbers, (z1 / z2), and place the
82  *              result in the Complex_Type '*quotient'.
83  *
84  *  ALGORITHM:   The code uses Smith's formula (page 25 of [1]) to perform
85  *              the division.
86  *
87  *  INCLUDE:     "complex.h"
88  *
89  *  PARAMETERS: The parameters give the two operands z1 and z2, and a
90  *              pointer to the result, quotient.
91  *
92  *  EXAMPLE:     Complex_Type z1, z2, z3;
93  *
94  *              cdiv(z1, z2, &z3);
95  *
96  * ------------------==========================================================
97  */
98
99 #ifdef PROTOTYPE
100
```

257

```
101     void cdiv(Complex_Type z1, Complex_Type z2, Complex_Type *quotient);
102
103 #else
104
105     void cdiv();
106
107 #endif
108
109
110
111
112
113 /* -----------------========     FUNCTION DECLARATION     =======----------------
114  *
115  *   PURPOSE:     To multiply two complex numbers, z1 and z2, and place their
116  *               product in the Complex_Type '*product'.
117  *
118  *   INCLUDE:     "complex.h"
119  *
120  *   PARAMETERS: The parameters give the two operands z1 and z2, and a
121  *               pointer to the result, product.
122  *
123  *   EXAMPLE:     Complex_Type z1, z2, z3;
124  *
125  *               cmul(z1, z2, &z3);
126  *
127  * -----------------============================================================-----------
128  */
129
130
131 #ifdef PROTOTYPE
132
133     void cmul(Complex_Type z1, Complex_Type z2, Complex_Type *product);
134
135 #else
136
137     void cmul();
138
139 #endif
140
141
142
143
144
145 /* -----------------========     FUNCTION DECLARATION     =======----------------
146  *
147  *   PURPOSE:     To place the difference of two complex numbers, (z1 - z2),
148  *               into the Complex_Type '*difference'.
149  *
150  *   INCLUDE:     "complex.h"
```

258

```
151   *
152   *   PARAMETERS: The parameters give the two operands z1 and z2, and a
153   *               pointer to the result, difference.
154   *
155   *   EXAMPLE:    Complex_Type z1, z2, z3;
156   *
157   *               csub(z1, z2, &z3);
158   *
159   * -----------=========================================================---------
160   */
161
162
163 #ifdef PROTOTYPE
164
165    void csub(Complex_Type z1, Complex_Type z2, Complex_Type *difference);
166
167 #else
168
169    void csub();
170
171 #endif
172
173
174
175
176
177 /* ------------========== FUNCTION DECLARATION  =======================
178   *
179   *   PURPOSE:    To return the imaginary part of a complex number, z.
180   *
181   *   PARAMETERS: The complex number, z, is passed into Im().
182   *
183   *   RETURNS:    The imaginary part of z as type double; that is a real
184   *               number y so that y * sqrt(-1) [or iy] is the imaginary part
185   *               of z.
186   *
187   *   EXAMPLE:    y = Im(z);
188   *
189   * -----------=========================================================---------
190   */
191
192 #ifdef PROTOTYPE
193
194    double Im(Complex_Type z);
195
196 #else
197
198    double Im();
199
200 #endif
```

```
201
202
203
204
205
206 /* --------------========== FUNCTION DECLARATION ==================------
207  *
208  *  PURPOSE:    This function returns the real part of a complex number, z.
209  *
210  *  PARAMETERS: The complex number, z, is passed into Re().
211  *
212  *  RETURNS:    The real part of z as type double.
213  *
214  *  EXAMPLE:    x = Re(z);
215  *
216  * --------------=========================================================------
217  */
218
219
220 #ifdef PROTOTYPE
221
222    double Re(Complex_Type z);
223
224 #else
225
226    double Re();
227
228 #endif
229
230
231 /* --------------=========== EOF complex.h ==================------------- */
```

```
 1  /* ----------------========     PROGRAM  INFORMATION    ==========-------------
 2   *
 3   *  SOURCE   :   complex.c
 4   *  VERSION  :   1.6
 5   *  DATE     :   09 September 1991
 6   *  AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7   *  DETAILS  :   See "complex.h".
 8   *
 9   * -------------=========================================================------------
10   */
11
12  #include <stdio.h>
13  #include "complex.h"
14
15
16
17
18
19  /* ----------------========     FUNCTION  DEFINITION    ==========------------- */
20
21
22  #ifdef PROTOTYPE
23
24      void cadd(Complex_Type z1, Complex_Type z2, Complex_Type *sum)
25
26  #else
27
28      void cadd(z1, z2, sum)
29
30          Complex_Type z1,
31                       z2,
32                       *sum;
33
34  #endif
35  {
36
37      sum->x = z1.x + z2.x;
38      sum->y = z1.y + z2.y;
39
40  }
41  /*  End cadd() ---------------------------------------------------------- */
42
43
44
45
46
47  /* ----------------========     FUNCTION  DEFINITION    ==========------------- */
48
49
50  #ifdef PROTOTYPE
```

261

```
51
52      void cdiv(Complex_Type z1, Complex_Type z2, Complex_Type *quotient)
53
54 #else
55
56      void cdiv(z1, z2, quotient)
57
58          Complex_Type z1,
59                       z2,
60                       *quotient;
61 #endif
62 {
63
64      double d;
65
66
67      if (fabs(z2.y) < fabs(z2.x)) {
68
69          d = (z2.y / z2.x);
70
71          quotient->x = ((z1.x + z1.y * d)/(z2.x + z2.y * d));
72          quotient->y = ((z1.y - z1.x * d)/(z2.x + z2.y * d));
73      }
74      else {
75
76          d = (z2.x / z2.y);
77
78          quotient->x = (( z1.y + z1.x * d)/(z2.y + z2.x * d));
79          quotient->y = ((-z1.x + z1.y * d)/(z2.y + z2.x * d));
80      }
81 }
82 /*  End cdiv() --------------------------------------------------------- */
83
84
85
86
87
88 /* ----------------========         FUNCTION  DEFINITION   ========------------ */
89
90
91 #ifdef PROTOTYPE
92
93      void cmul(Complex_Type z1, Complex_Type z2, Complex_Type *product)
94
95 #else
96
97      void cmul(z1, z2, product)
98
99          Complex_Type z1,
100                       z2,
```

262

```
101                         *product;
102 #endif
103 {
104
105     product->x = (z1.x * z2.x - z1.y * z2.y);
106     product->y = (z1.x * z2.y + z1.y * z2.x);
107 }
108 /*  End cmul() ---------------------------------------------------- */
109
110
111
112
113
114 /* -------------========    FUNCTION DEFINITION   ===================--- */
115
116
117 #ifdef PROTOTYPE
118
119     void csub(Complex_Type z1, Complex_Type z2, Complex_Type *difference)
120
121 #else
122
123     void csub(z1, z2, difference)
124
125         Complex_Type z1,
126                       z2,
127                      *difference;
128 #endif
129 {
130
131     difference->x = z1.x - z2.x;
132     difference->y = z1.y - z2.y;
133
134 }
135 /*  End csub() ---------------------------------------------------- */
136
137
138
139
140
141 /* -------------========    FUNCTION DEFINITION   ===================--- */
142
143
144 #ifdef PROTOTYPE
145
146 double Im(Complex_Type z)
147
148 #else
149
150     double Im(z)
```

263

```
151
152          Complex_Type z;
153
154  #endif
155  {
156
157      return(z.x);
158
159  }
160  /*  End Im() --------------------------------------------------------------- */
161
162
163
164
165
166  /* --------------========          FUNCTION  DEFINITION   ========------------- */
167
168
169  #ifdef PROTOTYPE
170
171      double Re(Complex_Type z)
172
173  #else
174
175      double Re(z)
176
177          Complex_Type z;
178
179  #endif
180  {
181
182      return(z.y);
183
184  }
185  /*  End Re() --------------------------------------------------------------- */
186
187
188  /* --------------============          EOF complex.c    ============------------- */
```

```
1  /* --------------========== PROGRAM INFORMATION ======================
2   *
3   *   SOURCE   :   epsilon.h
4   *   VERSION  :   1.7
5   *   DATE     :   09 September 1991
6   *   AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7   *
8   *
9   * --------------============== REFERENCES ===================-------------
10  *
11  *   [1] Gragg, William B.  Personal conversations, course notes, and MATLAB
12  *           code, 1991.
13  *
14  *
15  * --------------============== DESCRIPTION ===================------------
16  *
17  *   This file contains declarations of functions that determine the machine
18  *   precision for a particular machine.  The definition of epsilon is given
19  *   below.
20  *
21  *
22  * --------------========== LIST OF FUNCTIONS ===================------------
23  *
24  *   epsd()
25  *   epsf()
26  *
27  * --------------------===============================================-----------
28  */
29
30
31
32
33
34  /* --------------========== FUNCTION DECLARATION ======================
35  *
36  *   PURPOSE:     To find the machine precision.  The machine precision, eps,
37  *               is defined as the largest number which satisfies:
38  *
39  *                               1.0 + eps == 1.0
40  *
41  *   This program uses the type "double" which normally means an 8-byte
42  *   (64-bit) floating-point number stored in the IEEE 754 double precision
43  *   standard representation of [ 1 sign bit ][ 11-bit exponent ][ 52-bit
44  *   mantissa/significand ].
45  *
46  *   INCLUDE:     "epsilon.h"
47  *
48  *   RETURNS:     The value of epsilon (double).
49  *
50  * --------------------===============================================-----------
```

```
51  */
52
53 double epsd();
54
55
56
57
58
59 /* -----------------========    FUNCTION DECLARATION    ========--------------
60  *
61  *  PURPOSE:     This function is identical to epsd() except that it returns
62  *               type float.  Note:  The values returned may be identical,
63  *               probably reflecting C arithmetic done in type double
64  *               regardless of the ultimate type returned.  Anyway, this
65  *               function does everything using type float.
66  *
67  *  INCLUDE:     "epsilon.h"
68  *
69  *  RETURNS:     The value of epsilon (float).
70  *
71  * ----------------==================================================--------------
72  */
73
74 float epsf();
75
76
77 /* -----------------========    EOF epsilon.h    ========------------- */
```

266

```
 1  /* -------------========== 	 PROGRAM  INFORMATION 	 =====================-----
 2   *
 3   *   SOURCE   :   generate.h
 4   *   VERSION  :   1.7
 5   *   DATE     :   09 September 1991
 6   *   AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7   *
 8   * -------------=============== 	 REFERENCES 	 ====================------------
 9   *
10   *   [1] Gragg, William B.  Personal conversations, course notes, and MATLAB
11   *           codes, 1991.
12   *
13   *
14   * -------------=============== 	 DESCRIPTION 	 ====================-----------
15   *
16   *   Declarations of matrix and vector generation/initialization functions.
17   *
18   *
19   * -------------=========== 	 LIST OF FUNCTIONS 	 ====================--------
20   *
21   *   hilbert()
22   *   identity()
23   *   initial_permutation_vector()
24   *   mxrand()
25   *   wilkinson()
26   *   zeros()
27   *
28   * ------------============================================================-------------
29   */
30
31
32  /* -------------========== 	 FUNCTION DECLARATION 	 ====================--------
33   *
34   *   PURPOSE:     This function generates a Hilbert matrix of the specified
35   *               size.  The function takes care of memory allocation, so
36   *               the caller does not need to do this.  The definition used
37   *               for a Hilbert matrix is (for rows and columns numbered from
38   *               1) that the element at the (i,j) position has the value
39   *               (1/(i + j - 1)).
40   *
41   *   INCLUDE:     "allocate.h"
42   *               "matrix.h"
43   *
44   *   CALLS:       matalloc()
45   *
46   *   CALLED BY:
47   *
48   *   PARAMETERS: The parameters tell the size of the desired matrix.
49   *
50   *   RETURNS:    On success (i.e. no allocation problems), hilbert() returns
```

267

```
51   *              the allocated matrix filled with the values as described.
52   *              A NULL return value flags an allocation failure.
53   *
54   *   EXAMPLE:    Double_Matrix_Type *A = hilbert(5, 7);
55   *
56   * --------------================================================--------------
57   */
58
59 #ifdef PROTOTYPE
60
61     Double_Matrix_Type *hilbert(int rows, int cols);
62
63 #else
64
65     Double_Matrix_Type *hilbert();
66
67 #endif
68
69
70
71
72
73 /* ---------------========== FUNCTION DECLARATION  ==========--------------
74   *
75   *   PURPOSE:    This function generates an Identity matrix of the specified
76   *               size.  The function takes care of memory allocation, so
77   *               the caller does not need to do this.
78   *
79   *   INCLUDE:    "allocate.h"
80   *               "matrix.h"
81   *
82   *   CALLS:      matalloc()
83   *
84   *   CALLED BY:
85   *
86   *   PARAMETERS: The parameters tell the size of the matrix.
87   *
88   *   RETURNS:    On success (i.e., no allocation problems), identity()
89   *               returns the allocated matrix filled with the ones on the
90   *               diagonal.  A NULL return value flags an allocation failure.
91   *
92   *   EXAMPLE:    Double_Matrix_Type *A = identity(5, 7);
93   *
94   * --------------================================================--------------
95   */
96
97
98 #ifdef PROTOTYPE
99
100    Double_Matrix_Type *identity(int rows, int cols);
```

268

```
101
102 #else
103
104    Double_Matrix_Type *identity();
105
106 #endif
107
108
109
110
111
112 /* ----------------------========  FUNCTION DECLARATION   =============-----------
113  *
114  *   PURPOSE:      To initialize a permutation vector, p[].  This function
115  *                 performs allocation for p[], assuming that it must contain
116  *                 n integer elements.  Additionally, the function assigns
117  *                 values p[j] = j for all 0 <= j < n.  If allocation fails, p
118  *                 will be NULL upon return.
119  *
120  *   INCLUDE:      "allocate.h"
121  *
122  *   CALLS:        intvecalloc()
123  *
124  *   CALLED BY:
125  *
126  *   PARAMETERS: The size of the vector, n.
127  *
128  *   RETURNS:      (A pointer to) The vector.
129  *
130  * -------------========================================================-----------
131  */
132
133 #ifdef PROTOTYPE
134
135    int *initial_permutation_vector(int n);
136
137 #else
138
139    int *initial_permutation_vector();
140
141 #endif
142
143
144
145
146
147 /* ----------------------========  FUNCTION DECLARATION   =============-----------
148  *
149  *   PURPOSE:      This function generates a matrix whose elements are pseudo-
150  *                 random numbers (generated by lcdrand() in mathx.c).
```

269

```
151  *
152  *   INCLUDE:     "allocate.h"
153  *                "mathx.h"
154  *                "matrix.h"
155  *
156  *   CALLS:       lcdrand()
157  *                matalloc()
158  *
159  *   CALLED BY:
160  *
161  *   PARAMETERS: The parameters tell the size of the matrix.
162  *
163  *   RETURNS:     On success (i.e., no allocation problems), mxrand() returns
164  *                the allocated matrix filled with the random values.  A NULL
165  *                return value flags an allocation failure.
166  *
167  *   EXAMPLE:     Double_Matrix_Type *A = mxrand(5, 7);
168  *
169  *   -------------==================================================-------------
170  */
171
172
173 #ifdef PROTOTYPE
174
175    Double_Matrix_Type *mxrand(int rows, int cols);
176
177 #else
178
179    Double_Matrix_Type *mxrand();
180
181 #endif
182
183
184
185
186
187 /* -------------========== FUNCTION DECLARATION    ====================-------------
188  *
189  *   PURPOSE:     This function generates a Wilkinson matrix of the specified
190  *                size.  The function takes care of memory allocation, so
191  *                the caller does not need to do this.  The definition used
192  *                for a wilkinson matrix is:  ones along the diagonal, ones
193  *                along the rightmost column, zeros in the upper right
194  *                triangle, and (-1)'s in the lower left triangle.
195  *
196  *                     [  1                        1 ]
197  *                     [ -1  1                     1 ]
198  *                     [ -1 -1  1                  1 ]
199  *                     [ -1 -1 -1  1               1 ]
200  *                                 .                  .
```

270

```
201  *                              .                          .
202  *                              .                          .
203  *
204  *   INCLUDE:     "allocate.h"
205  *                "matrix.h"
206  *
207  *   CALLS:       matalloc()
208  *
209  *   CALLED BY:
210  *
211  *   PARAMETERS: The parameters tell the size of the matrix.
212  *
213  *   RETURNS:     On success (i.e. no allocation problems), wilkinson()
214  *                returns the allocated matrix filled with the values as
215  *                described.  On (allocation) failure, wilkinson() returns
216  *                NULL.
217  *
218  *   EXAMPLE:     Double_Matrix_Type *A = wilkinson(5, 7);
219  *
220  *   ------------------=================================================================-------------
221  */
222
223 #ifdef PROTOTYPE
224
225    Double_Matrix_Type *wilkinson(int rows, int cols);
226
227 #else
228
229    Double_Matrix_Type *wilkinson();
230
231 #endif
232
233
234
235
236
237 /* -----------------=========   FUNCTION DECLARATION   =========------------
238  *
239  *   PURPOSE:     This function generates a matrix of the specified size,
240  *                where all of the entries are zero.
241  *
242  *   INCLUDE:     "allocate.h"
243  *                "matrix.h"
244  *
245  *   CALLS:       matalloc()
246  *
247  *   CALLED BY:
248  *
249  *   PARAMETERS: The parameters tell the size of the matrix.
250  *
```

271

```
251  *   RETURNS:     On success (i.e. no allocation problems), zeros() returns
252  *                the allocated matrix filled with zeros.  On allocation
253  *                failure, zeros() returns NULL.
254  *
255  *   EXAMPLE:     Double_Matrix_Type *A = zeros(5, 7);
256  *
257  *  ------------------=========================================================-------------
258  */
259
260 #ifdef PROTOTYPE
261
262    Double_Matrix_Type *zeros(int rows, int cols);
263
264 #else
265
266    Double_Matrix_Type *zeros();
267
268 #endif
269
270
271 /* -----------------===========     EOF  generate.h   ==========------------- */
```

```
 1 /* -----------------=========    PROGRAM INFORMATION    ===============------------
 2  *
 3  *   SOURCE  :   io.h
 4  *   VERSION :   2.2
 5  *   DATE    :   09 September 1991
 6  *   AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  *
 9  * ------------------==============    DESCRIPTION    ===============------------
10  *
11  *   This file contains declarations of functions for matrix and vector
12  *   input/output.  The matrix structures such as "Double_Matrix_Type" are
13  *   given in "matrix.h".
14  *
15  *   The following parameters are common enough to justify a one-time
16  *   explanation here (and not with each occurrence below):
17  *
18  *       width   the width in which to print a value
19  *
20  *       aft     the number of places to print after the decimal point
21  *
22  *
23  * ------------------===========    LIST OF FUNCTIONS    ===============------------
24  *
25  *   answer()
26  *   fill_matrix()
27  *   fread_matrix()
28  *   fwrite_matrix()
29  *   getint()
30  *   get_matrix_size()
31  *   pause()
32  *   printmd()
33  *   printvd()
34  *   printvi()
35  *
36  * -----------------==================================================------------
37  */
38
39
40 /* -----------------==========    MANIFEST CONSTANTS    ===============------- */
41
42
43 #define LONG_AFT            8
44 #define LONG_WIDTH          12
45 #define SHORT_AFT           2
46 #define SHORT_WIDTH         5
47 #define STD_AFT             3
48 #define STD_WIDTH           8
49
50
```

273

```
51 /* -------------========     FUNCTION DECLARATION     ======================
52  *
53  *   PURPOSE:     To get a yes or no answer from the user.
54  *
55  *   NOTE:        This function includes the prompt "(y/n)?  " so you do not
56  *                have to include this in your query.  There is no space
57  *                before, two spaces after, and no newline (i.e. as shown).
58  *
59  *   INCLUDE:     <stdio.h>
60  *                "io.h"
61  *
62  *   CALLS:       getchar()                 <stdio.h>
63  *
64  *   CALLED BY:
65  *
66  *   PARAMETERS: void.
67  *
68  *   RETURNS:     (int) YES or NO (as defined in matrix.h).
69  *
70  *   ------------================================================================
71  */
72
73
74 int answer();
75
76
77
78
79
80
81 /* -------------========     FUNCTION DECLARATION     ====================
82  *
83  *   PURPOSE:     A function which prompts the user for the pertinent data
84  *                about a matrix and fills the structure provided with the
85  *                appropriate information.  That is, this function allows the
86  *                user to input the values of the elements.
87  *
88  *   PARAMETERS: A pointer to the structure containing the matrix to be
89  *                filled.
90  *
91  *   INCLUDE:     <stdio.h>
92  *                "io.h"
93  *
94  *   CAUTION:     This function ASSUMES that the "rows" and "cols" fields
95  *                have been correctly assigned by something like matalloc()
96  *                [see "allocate.h"] and makes no effort to enter a value in
97  *                those fields of the matrix structure.
98  *
99  *   CALLS:       ()
100 *
```

274

```
101  *   CALLED BY:
102  *
103  *   PARAMETERS: The parameters tell the size of the matrix.
104  *
105  *   RETURNS:    The matrix associated with A is operated on during the
106  *               execution of the function, and the result is available
107  *               upon return.
108  *
109  *   EXAMPLE:    if (!fill_matrix(&A))....
110  *
111  * ------------------------------------------------------------------------
112  */
113
114 #ifdef PROTOTYPE
115
116    void fill_matrix(Double_Matrix_Type *A);
117
118 #else
119
120    void fill_matrix();
121
122 #endif
123
124
125
126 /* ---------------========== FUNCTION DECLARATION ===================------------
127  *
128  *   PURPOSE:    A function which reads data from a file and stores it in
129  *               the matrix of A.  This function takes care of matrix
130  *               allocation for the caller.
131  *
132  *   INCLUDE:    <stdio.h>
133  *               "io.h"
134  *
135  *   CAUTION:    This function ASSUMES the file has been stored in the
136  *               format described in "matrix.fmt".
137  *
138  *   CALLS:      fgets()
139  *               fscanf()
140  *               rewind()
141  *
142  *   CALLED BY:
143  *
144  *   PARAMETERS: The pointer to the matrix structure and the file pointer.
145  *
146  *   RETURNS:    1 on success and 0 on any sort of failure.
147  *
148  * ------------------------------------------------------------------------
149  */
150
```

275

```
151 #ifdef PROTOTYPE
152
153     int fread_matrix(Double_Matrix_Type **A, FILE *fp);
154
155 #else
156
157     int fread_matrix();
158
159 #endif
160
161
162
163
164
165 /* -----------------========     FUNCTION DECLARATION    =========-------------
166  *
167  *  PURPOSE:     A function which writes data from A->matrix[][] to a file
168  *               pointed to by fp.
169  *
170  *  INCLUDE:     <stdio.h>
171  *               "io.h"
172  *
173  *  ASSUMPTION: The caller has already performed fopen() on fp for the
174  *               "w" (write) mode.
175  *
176  *  CALLS:       fprintf()
177  *               rewind()
178  *
179  *  CALLED BY:
180  *
181  *  PARAMETERS: A is a pointer to the structure which contains the matrix.
182  *               fp is a FILE pointer.
183  *
184  *  RETURNS:     1 on success and 0 on failure.
185  *
186  *  ------------========================================================-------------
187  */
188
189
190 #ifdef PROTOTYPE
191
192     int fwrite_matrix(Double_Matrix_Type *A, FILE *fp, int width, int aft);
193
194 #else
195
196     int fwrite_matrix();
197
198 #endif
199
200
```

276

```
201  /* ----------------========    FUNCTION DECLARATION    ====================-----------
202   *
203   *   PURPOSE:     A function to get user input of a single integer.
204   *
205   *   INCLUDE:     <stdio.h>
206   *                "io.h"
207   *
208   *   CALLS:       fflush()
209   *                scanf()
210   *
211   *   CALLED BY:
212   *
213   *   RETURNS:     The user's integer.
214   *
215   * -------------=====================================================================-----
216   */
217
218  int getint();
219
220
221
222  /* ----------------========    FUNCTION DECLARATION    ====================-----------
223   *
224   *   PURPOSE:     A function to ask the user for the size of a matrix.
225   *
226   *   INCLUDE:     <stdio.h>
227   *                "io.h"
228   *
229   *   CALLS:       answer()
230   *                fflush()
231   *                scanf()
232   *
233   *   CALLED BY:
234   *
235   *   PARAMETERS: Pointers to the size of the matrix (m rows by n columns).
236   *
237   * -------------=====================================================================-----
238   */
239
240  #ifdef PROTOTYPE
241
242      void get_matrix_size(int *m, int *n);
243
244  #else
245
246      void get_matrix_size();
247
248  #endif
249
250
```

277

```
251 /* --------------========== FUNCTION DECLARATION ==========--------------
252  *
253  *  PURPOSE:    Press a key to continue!
254  *
255  *  INCLUDE:    <stdio.h>
256  *              "io.h"
257  *
258  *  CALLS:      fflush()
259  *              getchar()
260  *              printf()
261  *
262  * -------------========================================================--------------
263  */
264
265 void pause();
266
267
268 /* --------------========== FUNCTION DECLARATION ==========--------------
269  *
270  *  PURPOSE:    This function provides a printout of the information stored
271  *              in the structure A.
272  *
273  *  INCLUDE:    <stdio.h>
274  *              "io.h"
275  *
276  *  CALLS:      printf()
277  *
278  *  PARAMETERS: A is the structure that contains the matrix to be printed.
279  *              The width and aft values are described near the top of this
280  *              file.  The defaults are defined as manifest constants.
281  *
282  *  EXAMPLE:    Double_Matrix_Type *A = hilbert(7, 5);
283  *
284  *              printmd(*A, LONG_WIDTH, LONG_AFT);
285  *
286  * -------------========================================================--------------
287  */
288
289 #ifdef PROTOTYPE
290
291    void printmd(Double_Matrix_Type A, int width, int aft);
292
293 #else
294
295    void printmd();
296
297 #endif
298
299
300
```

278

```
301 /* --------------========    FUNCTION DECLARATION   ==========-------------
302  *
303  *  PURPOSE:    This function prints the vector, v, of doubles.
304  *
305  *  INCLUDE:    <stdio.h>
306  *              "io.h"
307  *
308  *  CALLS:      printf()
309  *
310  *  CALLED BY:
311  *
312  *  PARAMETERS: v is the vector.  size is the number of elements in v□.
313  *
314  * --------------===============================================-------------
315  */
316
317
318 #ifdef PROTOTYPE
319
320     void printvd(double *v, int size, int width, int aft);
321
322 #else
323
324     void printvd();
325
326 #endif
327
328
329
330
331
332 /* --------------========    FUNCTION DECLARATION   ==========-------------
333  *
334  *  PURPOSE:    This function provides a printout of the integer vector v.
335  *
336  *  INCLUDE:    <stdio.h>
337  *              "io.h"
338  *
339  *  CALLS:      printf()
340  *
341  *  CALLED BY:
342  *
343  *  PARAMETERS: v is a vector of size integers.
344  *
345  * --------------===============================================-------------
346  */
347
348 #ifdef PROTOTYPE
349
350     void printvi(int *v, int size, int width);
```

279

```
351
352 #else
353
354    void printvi();
355
356 #endif
357
358
359
360
361 /* --------------============== EOF io.h ==============------------- */
```

```
 1 /* ---------------========       PROGRAM  INFORMATION    =====================
 2  *
 3  *   SOURCE   :   mathx.h
 4  *   VERSION  :   1.2
 5  *   DATE     :   09 September 1991
 6  *   AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  *
 9  * -------------=============       REFERENCES       ===============-------------
10  *
11  *   [1] Knuth, Donald E.  The Art of Computer Programming, Volume 2: Semi-
12  *            numerical Algorithms.  Addison-Wesley Publishing Company,
13  *            Reading, MA, 1969, pp. 9-24.
14  *
15  *   [2] Sedgewick, Robert.  Algorithms, Second Edition. Addison-Wesley
16  *            Publishing Company, Reading, MA, 1988, pp. 513-514.
17  *
18  *
19  * ---------------=============       DESCRIPTION       ===============-------------
20  *
21  *   A small extension to the usual C <math.h>.
22  *
23  *
24  * -------------============       LIST OF FUNCTIONS    ===========--------------
25  *
26  *   lcdrand()
27  *   lclrand()
28  *   multmod()
29  *   pow2()
30  *
31  * ----------------==================================================--------------
32  */
33
34
35
36
37
38 /* ------------==========       MANIFEST CONSTANTS   ====================-------------- */
39
40 #ifndef EXIT_FAILURE
41 #define EXIT_FAILURE              -1
42 #endif
43
44 #define START               1234567    /*  starting value, Xo.  See [1] */
45 #define MULT                31415821    /*  multiplier, a.       See [1] */
46 #define INCR                       1    /*  increment, c.        See [1] */
47 #define SQRTM                  10000    /*  sqrt(m)                      */
48 #define MODULUS            100000000    /*  modulus, m.          See [1] */
49
50
```

281

```
51 /* -------------========          FUNCTION  DECLARATION   ========-------------
52  *
53  *   PURPOSE:     To calculate a pseudo-random number in the range [0, 1]
54  *                using the linear congruential method.  This function is a
55  *                very simple application of lclrand().  It merely divides
56  *                the value that lclrand() returns by the modulus, and
57  *                returns the resulting double value.
58  *
59  *   INCLUDE:     "mathx.h"
60  *
61  *   CALLS:       lclrand()
62  *
63  *   CALLED BY:   mxrand()                     "generate.c"
64  *
65  *   PARAMETERS:  The parameters are identical to those for lclrand().
66  *
67  *   RETURNS:     A pseudo-random double value in the range [ 0.0, 1.0 ].
68  *
69  *   EXAMPLE:     double d;
70  *
71  *                d = lcdrand(START, MULT, INCR, SQRTM, MODULUS);
72  *
73  * -------------========================================================-------------
74  */
75
76
77 #ifdef PROTOTYPE
78
79     double lcdrand(long Xn, long a, long c, long sqrtm, long m);
80
81 #else /* iPSC/2 */
82
83     double lcdrand(/* long Xn, long a, long c, long sqrtm, long m */);
84
85 #endif
86
87
88
89
90
91 /* -------------========          FUNCTION  DECLARATION   ========-------------
92  *
93  *   PURPOSE:     To calculate a pseudo-random number of type long in the
94  *                range [0, (m-1)], where m is the argument for modulus.  The
95  *                algorithm uses the linear congruential method.  This method
96  *                is given in great detail in [1].  A shorter, algorithmic
97  *                treatment is given in [2].  I have tested the function to
98  *                be sure that it produces the ten numbers listed on page 513
99  *                of [2].
100 *
```

282

```
101  *   INCLUDE:      "mathx.h"
102  *
103  *   CALLS:        multmod()
104  *
105  *   CALLED BY:    lcdrand()
106  *
107  *   PARAMETERS:   The notation comes from [1] (more-or-less).  Xn is the
108  *                 starting value.  a is the multiplier.  c is the increment.
109  *                 sqrtm is the square root of m, which is the modulus.  A
110  *                 negative value for any of the arguments is impossible and
111  *                 will invoke the defaults given among the manifest constants
112  *                 above.  The starting value, Xn, is the exception.  If you
113  *                 supply a nonnegative value, your value will be accepted as
114  *                 the starting value.  Else, the starting value BEGINS at the
115  *                 default START and is changed each time the function is
116  *                 called (as long as the starting value argument, Xn, is
117  *                 negative).  That is, Xn HAS MEMORY as long as your program
118  *                 is running.  The other parameters are determined from call-
119  *                 to-call.
120  *
121  *   RETURNS:      A pseudo-random long in the range [ 0, (m-1) ], where m is
122  *                 the modulus argument.
123  *
124  *   EXAMPLE:      This example illustrates the use of the default values:
125  *
126  *       long l;
127  *
128  *       l = lclrand(START, MULT, INCR, SQRTM, MODULUS);
129  *
130  *   -------------------===============================================---------------
131  */
132
133
134 #ifdef PROTOTYPE
135
136     long lclrand(long Xn, long a, long c, long sqrtm, long m);
137
138 #else /* iPSC/2 */
139
140     long lclrand(/* long Xn, long a, long c, long sqrtm, long m */);
141
142 #endif
143
144
145
146
147
148
149
150
```

283

```
151 /* ------------------========    FUNCTION  DECLARATION   ====================
152  *
153  *   PURPOSE:     To calculate (a * b) mod m^2, while trying to avoid over-
154  *               flow.  This function is adapted from Sedgewick's 'mult'
155  *               function on page 513 of [1].
156  *
157  *   INCLUDE:     "mathx.h"
158  *
159  *   CALLS:
160  *
161  *   CALLED BY:  lclrand()
162  *
163  *   PARAMETERS: long a, b, m.
164  *
165  *   RETURNS:     long (a * b) mod m^2.
166  *
167  * ------------------================================================--------------
168  */
169
170
171 #ifdef PROTOTYPE
172
173     long multmod(long a, long b, long m);
174
175 #else
176
177     long multmod(/* long a, long b, long m */);
178
179 #endif
180
181
182
183
184
185 /* ------------------========    FUNCTION  DECLARATION   ====================
186  *
187  *   PURPOSE:     To calculate the value of two raised to the (n) power. This
188  *               function [unlike the macro POW2() given in macros.h] will
189  *               handle the case where (n == 0).  This function uses left
190  *               shifts to achieve the result, so if you ask for too large a
191  *               value, the result is not guaranteed.  The value of n is
192  *               ASSUMED to be a POSITIVE integer.
193  *
194  *   INCLUDE:     "mathx.h"
195  *
196  *   CALLS:
197  *
198  *   CALLED BY:
199  *
200  *   PARAMETERS: The desired power of two, n.
```

284

```
201  *
202  *   RETURNS:    The function returns the value of 2^(n).
203  *
204  * -------------================================================----------
205  */
206
207
208 #ifdef PROTOTYPE
209
210     long pow2(int n);
211
212 #else
213
214     long pow2(/* int n */);
215
216 #endif
217
218
219
220
221
222 /* --------------============== EOF mathx.h ===============----------- */
```

```
 1  /* --------------========== PROGRAM INFORMATION ==========-------------
 2   *
 3   * SOURCE  :   num_sys.h
 4   * VERSION :   1.4
 5   * DATE    :   09 September 1991
 6   * AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7   *
 8   *
 9   * ------------=============== REFERENCES =============-------------
10   *
11   * [1] Goldberg, David.  ''What Every Computer Scientist Should Know About
12   *         Floating-Point Arithmetic.'' ACM Computing Surveys, Vol. 23,
13   *         No. 1, March, 1991, pp. 6-48.
14   *
15   * [2] Hayes, John P. ''Computer Architecture and Organization.'' McGraw-
16   *         Hill Book Company, New York, Second Edition, 1988, p. 196.
17   *
18   *
19   * -----------=============== DESCRIPTION ===========-------------
20   *
21   * The "num_sys" group of functions relate to number systems (e.g. binary,
22   * decimal, hexadecimal).
23   *
24   *
25   * ------------=========== LIST OF FUNCTIONS ==========-------------
26   *
27   * binrep()
28   * binvec()
29   * hexrep()
30   * ieeerep()
31   *
32   * -----------==================================================-------------
33   */
34
35
36  /* ------------========== FUNCTION DECLARATION ==========-------------
37   *
38   * PURPOSE:      To display the binary representation of a number. Given the
39   *               parameters described below, binrep() prints the binary
40   *               representation.  For numbers of type double, type float, or
41   *               type int; binrep() reverses the order of the bytes from the
42   *               machine storage.  This makes them more readily recognizable
43   *               as [ SIGN ][ EXPONENT ][ MANTISSA ] for the floating-point
44   *               types and orders the bytes in order of decreasing signifi-
45   *               cance for the integers.
46   *
47   * INCLUDE:    "num_sys.h"
48   *
49   * CALLS:
50   *
```

286

```
51  *  CALLED BY:
52  *
53  *  PARAMETERS: The function needs to know what type of number you are
54  *              sending in, so use the types given in matrix.h.  The
55  *              function understands TYPE_CHAR, TYPE_DOUBLE, TYPE_FLOAT,
56  *              and TYPE_INT).  It also needs a pointer to the_number.
57  *
58  *  EXAMPLE:    float f;
59  *
60  *              binrep(TYPE_FLOAT, &f);
61  *
62  *  ------------==========================================================--------------
63  */
64
65  #ifdef PROTOTYPE
66
67      void binrep(int number_type, void *the_number);
68
69  #else
70
71      void binrep();
72
73  #endif
74
75
76
77  /* ------------------=========   FUNCTION DECLARATION   ==========-------------
78  *
79  *  PURPOSE:    To expand the bits of the input into an array of integers.
80  *              The array only holds zeros and ones, with each element
81  *              representing a bit of the input number.
82  *
83  *  INCLUDE:    "num_sys.h"
84  *
85  *  CALLS:
86  *
87  *  CALLED BY:
88  *
89  *  CAUTION:    This function returns the bits AS THEY ARE IN THE MACHINE!
90  *              Many machines store type double, type float, and type int
91  *              so that their bytes are in an order that is the reverse of
92  *              what you might expect.  Of course, the bits within a byte
93  *              are in the expected (msb......lsb) order.
94  *
95  *  PARAMETERS: The function needs to know what type of number you are
96  *              sending in, so use the types given in matrix.h.  The
97  *              function recognizes TYPE_CHAR, TYPE_DOUBLE, TYPE_FLOAT, and
98  *              TYPE_INT.  It also asks for a pointer to the number.
99  *
100 *  RETURNS:    A pointer to int. The function will take care of allocation
```

287

```
101  *                for this pointer, and it will fill the array with the bits
102  *                of the number.  For indexing purposes, you will probably
103  *                need to know how big this vector is.  Multiply the
104  *                [sizeof(type you are sending in)] by 8 (bits/byte).  That's
105  *                how many elements will be in the returned vector of integer
106  *                (bits).  This pointer will be NULL if there was an alloca-
107  *                tion problem.
108  *
109  *   EXAMPLE:
110  *
111  *       float f;           Assume that this takes 4 bytes * 8 bits
112  *
113  *       int   *v;          To hold the bit vector of f (32 elements)
114  *
115  *       v = binvec(TYPE_FLOAT, &f);
116  *
117  * ------------===============================================---------------
118  */
119
120 #ifdef PROTOTYPE
121
122    int *binvec(int number_type, void *the_number);
123
124 #else
125
126    int *binvec();
127
128 #endif
129
130
131 /* ------------==========      FUNCTION DECLARATION     ==========------------
132  *
133  *   PURPOSE:     To display the hexadecimal representation of a number.
134  *
135  *   INCLUDE:     "num_sys.h"
136  *
137  *   CALLS:
138  *
139  *   CALLED BY:
140  *
141  *   PARAMETERS: The function needs to know what type of number you are
142  *                sending in, so use the types given in matrix.h.  The
143  *                function recognizes TYPE_CHAR, TYPE_DOUBLE, TYPE_FLOAT, and
144  *                TYPE_INT.  It also needs a pointer to the number.
145  *
146  *   EXAMPLE:     float f;
147  *
148  *                printf("The hexadecimal representation of %f is: ", f);
149  *                hexrep(TYPE_FLOAT, &f);
150  *
```

288

```
151   * ----------============================================================--------------
152   */
153
154  #ifdef PROTOTYPE
155
156      void hexrep(int number_type, void *the_number);
157
158  #else
159
160      void hexrep();
161
162  #endif
163
164
165  /* --------------==========    FUNCTION DECLARATION    ==========--------------
166   *
167   *   PURPOSE:     To display binary and IEEE representation of a number. This
168   *                is nearly a tutorial function!  It displays a binary repre-
169   *                sentation of the number, and then breaks out the sign,
170   *                exponent, and mantissa (or significand).  Some terse trans-
171   *                lation tips are also provided.
172   *
173   *   INCLUDE:     "num_sys.h"
174   *
175   *   CALLS:
176   *
177   *   CALLED BY:
178   *
179   *   PARAMETERS: The function needs to know what type of number you are
180   *                sending in, so use the types given in matrix.h.  This
181   *                function ONLY recognizes the floating-point types (i.e.,
182   *                TYPE_DOUBLE and TYPE_FLOAT).  It also needs a pointer to
183   *                the number.
184   *
185   *   EXAMPLE:     float f;
186   *
187   *                printf("The IEEE 754 representation of %f is: ", f);
188   *                ieeerep(TYPE_FLOAT, &f);
189   *
190   * ----------============================================================--------------
191   */
192
193  #ifdef PROTOTYPE
194
195      void ieeerep(int number_type, void *the_number);
196
197  #else
198
199      void ieeerep();
200
```

289

```
201 #endif
202
203
204 /* ------------------============ EOF num_sys.h ============------------- */
```

```
1  /* ----------------============   PROGRAM INFORMATION   =========--------------
2   *
3   *  SOURCE  :   ops.h
4   *  VERSION :   1.7
5   *  DATE    :   09 September 1991
6   *  AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7   *
8   *
9   * --------------=================   REFERENCES   ==========--------------------
10  *
11  *  [1] Golub, Gene H., and Charles F. VanLoan. Matrix Computations.  The
12  *          Johns Hopkins University Press, Baltimore, 1989.
13  *
14  *
15  * -------------================   DESCRIPTION   ==========---------------------
16  *
17  * The functions declared below perform matrix and vector operations.  For
18  * the sake of brevity, I will often use simple (MatLab-style) notation in
19  * comments.  For instance, x' means x transpose (i.e. a row).  Do not
20  * confuse the comment shorthand with what is really happening in the
21  * code.  My goal is to get function specifications across clearly and
22  * succinctly without excessive concern for implementation.  Here are a
23  * few notes.
24  *
25  * An operation preceded by a "." means "elementwise".  For instance,
26  * x .* y means the elementwise vector multiplication of x by y.  That is,
27  * the result would be some vector z like:
28  *
29  *          z' = [  x[1]*y[1],  x[2]*y[2],  ....,  x[n]*y[n]  ]
30  *
31  * If the operation appears without the preceding ".", it means the vector
32  * operation.
33  *
34  *
35  * ------------============   LIST OF FUNCTIONS   ==========--------------------
36  *
37  *  cols()
38  *  dot_product()
39  *  matrix_product()
40  *  max_element()
41  *  normp()
42  *  outer_product()
43  *  rows()
44  *  swap_cols()
45  *  swap_rows()
46  *  vec_init()
47  *
48  * --------------======================================================----------
49  */
50
```

```
51  /* ----------------========    FUNCTION DECLARATION    =========-------------
52   *
53   *   PURPOSE:     To return the number of columns in the matrix A.
54   *
55   *   INCLUDE:     "ops.h"
56   *
57   * -------------------------------------------------------------------------
58   */
59
60  #ifdef PROTOTYPE
61
62      int cols(Double_Matrix_Type *A);
63
64  #else
65
66      int cols(/* Double_Matrix_Type *A */);
67
68  #endif
69
70
71  /* ----------------========    FUNCTION DECLARATION    =========-------------
72   *
73   *   PURPOSE:     Computes the dot product of the input vectors x and y which
74   *               is defined in [1] (page 4).  The dot product of x and y is
75   *               x' * y.
76   *
77   *   PARAMETERS: The vectors x and y should be arrays of type double, each
78   *               having "size" elements.
79   *
80   *   INCLUDE:     "ops.h"
81   *
82   *   CALLS:       N/A
83   *
84   *   CALLED BY:  matrix_product()        [see below]
85   *
86   *   RETURNS:     A double (scalar) value equal to the dot product x' * y.
87   *
88   *   EXAMPLE:     The following example would conclude with answer == 10.0.
89   *
90   *       double       answer;
91   *
92   *       static double x[] = { 1.0, 2.0, 3.0 },
93   *                     y[] = { 3.0, 2.0, 1.0 };
94   *
95   *       int     size = 3;
96   *
97   *       answer = dot_product(x, y, size);
98   *
99   * -------------------------------------------------------------------------
100  */
```

292

```
101
102
103 #ifdef PROTOTYPE
104
105    double dot_product(double *x, double *y, int size);
106
107 #else
108
109    double dot_product(/* double *x, double *y, int size */);
110
111 #endif
112
113
114
115
116 /* ---------------=========     FUNCTION DECLARATION    =====================-------------
117  *
118  *   PURPOSE:     To multiply matrices A and B, placing the product in C.
119  *
120  *   INCLUDE:     "ops.h"
121  *
122  *   CALLS:       dot_product            [see above]
123  *
124  *   CALLED BY:
125  *
126  *   PARAMETERS: The parameters tell the size of the matrix.
127  *
128  *   RETURNS:     SUCCESS if the matrices were compatible for multiplication
129  *               and C contained enough space to contain the entire result.
130  *               FAILURE if A and B were incompatible or C was not big
131  *               enough to hold the product.  The values for SUCCESS and
132  *               FAILURE are given in 'matrix.h'.
133  *
134  *   EXAMPLE:     Double_Matrix_Type *A,
135  *                                  *B,
136  *                                  *C;
137  *
138  *               if (matrix_product(A,B,C) == FAILURE) {
139  *
140  *                   printf("matrix_product(A,B,C) failed.\n");
141  *                   exit(EXIT_FAILURE);
142  *               }
143  *               else {
144  *
145  *                   printf("C contains A * B.\n");
146  *               }
147  *
148  * -------------=================================================================---------
149  */
150
```

293

```
151 #ifdef PROTOTYPE
152
153     int matrix_product(Double_Matrix_Type *A,
154                        Double_Matrix_Type *B,
155                        Double_Matrix_Type *C);
156 #else
157
158     int matrix_product();
159
160 #endif
161
162
163
164 /* ---------------========== FUNCTION DECLARATION ==========-------------
165  *
166  *  PURPOSE:    To search the elements below and to the right of A(k,k) for
167  *              the element that is maximum in absolute value.
168  *
169  *  INCLUDE:    <math.h>                    [link using -lm if necessary]
170  *              "ops.h"
171  *
172  *  CALLS:      fabs()
173  *
174  *  CALLED BY:
175  *
176  *  PARAMETERS: A is the matrix (structure).  k is the index for a position
177  *              on the main diagonal, A(k,k).  The search will be conducted
178  *              for the area of the matrix that lies below k and to its
179  *              right:
180  *
181  *                      (k,k)-------------------->
182  *                          | This is the area that will be searched
183  *                          | for an element of maximum absolute value.
184  *                          | The search does NOT include row k nor
185  *                          | does it include column k.
186  *
187  *              Parameters must also include s, the address of an integer
188  *              that will contain the row number for the maximum element
189  *              upon return; and t, an address of an integer to store the
190  *              column number for the maximum element.
191  *
192  *  NOTE:       To search the WHOLE MATRIX, the parameter k should be (-1).
193  *              The values of k, s, and t should be interpreted as the C
194  *              versions of indexes (i.e. beginning with 0).
195  *
196  *  RETURNS:    The function returns the maximum (in absolute value)
197  *              element found in A (type double).  Additionally, the index
198  *              values for this element are placed in the variables pointed
199  *              to by s (row) and t (col).
200  *
```

294

```
201   *    EXAMPLE:
202   *
203   *        Double_Matrix_Type *A;
204   *
205   *        double  u;
206   *
207   *        int      k,
208   *                 s,
209   *                 t;
210   *
211   *        u = max_element(A, k, &s, &t);
212   *
213   * -------------========================================================-------------
214   */
215
216 #ifdef PROTOTYPE
217
218     double max_element(Double_Matrix_Type *A, int k, int *s, int *t);
219
220 #else
221
222     double max_element();
223
224 #endif
225
226
227
228 /* -----------------========          FUNCTION DECLARATION     =========-------------
229   *
230   * PURPOSE:     Computes the p-norm of the input vector x defined in [1]
231   *              (page 53).
232   *
233   * INCLUDE:     <math.h>
234   *              "ops.h"
235   *
236   * CALLS:       fabs()
237   *
238   * CALLED BY:
239   *
240   * PARAMETERS: x is the vector.  It must contain "size" elements of type
241   *             double.  The p argument is the p of p-norm.
242   *
243   * RETURNS:    A double (scalar) value equal to the p-norm of x.
244   *
245   * EXAMPLE:
246   *
247   *        static double  x[] = { 1.0, 2.0, 3.0 };
248   *
249   *        double         Euclidean_norm_of_x;
250   *
```

295

```
251  *        Euclidean_norm_of_x = normp(x, 2, 3);
252  *
253  * -------------===============================================--------------
254  */
255
256 #ifdef PROTOTYPE
257
258    double normp(double *x, int p, int size);
259
260 #else
261
262    double normp();
263
264 #endif
265
266
267 /* -------------==========    FUNCTION DECLARATION   ==========--------------
268  *
269  * PURPOSE:      To place the outer product of x and y in C.
270  *
271  * INCLUDE:      "ops.h"
272  *
273  * CALLS:        N/A
274  *
275  * CALLED BY:    N/A
276  *
277  * ASSUMPTION: The matrix associated with C is already allocated to the
278  *             proper size.
279  *
280  * PARAMETERS: Two vectors, x and y, of sizes x_size and y_size; and the
281  *             matrix associated with C to accept the outer product.
282  *
283  * RETURNS:     The matrix associated with C is filled with the proper
284  *             values.
285  *
286  * -------------===============================================--------------
287  */
288
289
290 #ifdef PROTOTYPE
291
292   void outer_product(double *x, int x_size, double *y, int y_size,
293                      double **C);
294 #else
295
296   void outer_product();
297
298 #endif
299
300
```

296

```
301 /* ----------------========== FUNCTION DECLARATION ==========-------------
302  *
303  *  PURPOSE:     To return the number of rows in the matrix A.
304  *
305  *  INCLUDE:     "ops.h"
306  *
307  * -------------===================================================----------
308  */
309
310 #ifdef PROTOTYPE
311
312    int rows(Double_Matrix_Type *A);
313
314 #else
315
316    int rows();
317
318 #endif
319
320
321
322 /* ----------------========== FUNCTION DECLARATION ==========-------------
323  *
324  *  PURPOSE:     To swap columns p and q in the matrix contained within A.
325  *
326  *  INCLUDE:     "ops.h"
327  *
328  *  CALLS:       N/A
329  *
330  *  CALLED BY:
331  *
332  *  PARAMETERS: A is the structure holding the matrix.  The integers p and
333  *              q are the column numbers to be swapped.  Indexes are
334  *              numbered according to the C convention (beginning at zero).
335  *
336  *  RETURNS:     Upon return, the columns have been swapped in A.
337  *
338  * -------------===================================================----------
339  */
340
341 #ifdef PROTOTYPE
342
343    void swap_cols(Double_Matrix_Type *A, int p, int q);
344
345 #else
346
347    void swap_cols();
348
349 #endif
350
```

297

```
351 /* ------------========== FUNCTION DECLARATION ========------------
352  *
353  * PURPOSE:     To swap rows p and q in the matrix contained within A.
354  *
355  * INCLUDE:     "ops.h"
356  *
357  * CALLS:       N/A
358  *
359  * CALLED BY:
360  *
361  * PARAMETERS: A is the structure holding the matrix.  The integers p and
362  *             q are the row numbers to be swapped.  Indexes are numbered
363  *             according to the C convention (beginning at zero).
364  *
365  * RETURNS:    Upon return, the rows have been swapped in A.
366  *
367  * ------------===================================================-----------
368  */
369
370 #ifdef PROTOTYPE
371
372    void swap_rows(Double_Matrix_Type *A, int p, int q);
373
374 #else
375
376    void swap_rows();
377
378 #endif
379
380
381
382
383 /* ------------========= FUNCTION DECLARATION ========------------
384  *
385  * PURPOSE:     To initialize the vector v of n integers with the values
386  *             1, 2, 3, ..., n.
387  *
388  * INCLUDE:     "ops.h"
389  *
390  * CALLS:
391  *
392  * CALLED BY:
393  *
394  * ASSUMPTION: The vector, v, has already been successfully allocated as
395  *             an array of n integers.
396  *
397  * PARAMETERS: The vector, v, to be initialized; and its size, n.
398  *
399  * RETURNS:    The vector's elements are set to the new values and these
400  *             values are in v[] upon return.
```

298

```
401   *
402   * ------------=================================================-------------
403   */
404
405
406 #ifdef PROTOTYPE
407
408     void vec_init(int *v, int n);
409
410 #else
411
412     void vec_init();
413
414 #endif
415
416
417 /* ------------=============   EOF ops.h   =============------------ */
```

299

```
 1 /* ----------------========== PROGRAM INFORMATION ============------------
 2 *
 3 *   SOURCE  :   timing.h
 4 *   VERSION :   1.2
 5 *   DATE    :   09 September 1991
 6 *   AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7 *
 8 * -----------============== REFERENCES ==============-------------
 9 *
10 *   REFERENCES :
11 *
12 *      [1] Inmos.  The Transputer Databook, Second Edition, 1989.
13 *
14 *      [2] Intel.  iPSC/2 Programmer's Reference Manual.
15 *
16 *
17 * ----------------========== DESCRIPTION ============--------------
18 *
19 *   This file contains definitions of manifest constants, type definitions,
20 *   and function declarations for time-related tasks on the Intel iPSC/2 or
21 *   a network of Inmos transputers.
22 *
23 *
24 * ----------------========== LIST OF FUNCTIONS ============-------------
25 *
26 *   clock()
27 *   delay()
28 *
29 * -----------================================================================-------------
30 */
31
32
33
34
35
36 /* ----------------========== MANIFEST CONSTANTS ============------------------ */
37
38 #ifdef TRANSPUTER
39
40 #define LO_PERIOD       64.0e-6      /*  period of low priority clock     */
41 #define HI_PERIOD        1.0e-6      /*  period of high priority clock    */
42 #define LO_FREQ      15625.0         /*  frequency of low priority clock  */
43 #define HI_FREQ          1.0e6       /*  frequency of high priority clock */
44
45 #else   /*  iPSC/2  */
46
47 #define M_PERIOD         1.0e-3      /*  period of Intel's mclock()       */
48 #define M_FREQ           1.0e-3      /*  frequency for Intel's mclock()   */
49
50 #endif
```

300

```
51
52
53 /* ------------------------========  TYPE DEFINITIONS   =========-------------
54  *
55  *   The type 'ticks' is defined in an effort to make timing a bit more
56  *   transparent across the machines listed.
57  *
58  * ----------------===============================================-------------
59  */
60
61 #ifdef TRANSPUTER
62
63 typedef int ticks;
64
65 #else    /*  iPSC/2  */
66
67 typedef unsigned long ticks;
68
69 #endif
70
71
72
73
74
75 /* ----------------========       FUNCTION  DECLARATION   =========-------------
76  *
77  *   PURPOSE:    To get the time (in ticks) from the processor's clock.
78  *
79  *   INCLUDE:    <conc.h>                  (Logical Systems C, version 89.1)
80  *               "timing.h"
81  *
82  *   CALLS:      Time()                    (Logical Systems C, version 89.1)
83  *               mclock()                  (Intel iPSC/2 C)
84  *
85  *   CALLED BY:
86  *
87  *   PARAMETERS: None.
88  *
89  *   RETURNS:    The function samples the clock and returns ticks.  More
90  *               information on ticks, period, and frequency is given in the
91  *               definitions above.
92  *
93  *   EXAMPLE:    ticks t[2];
94  *
95  *               t[0] = clock();
96  *
97  * ----------------===============================================-------------
98  */
99
100
```

301

```
101 #ifdef PROTOTYPE
102
103     ticks clock(void);
104
105 #else
106
107     ticks clock(/* void */);
108
109 #endif
110
111
112
113
114
115 /* ------------========     FUNCTION   DECLARATION   =========------------
116  *
117  *   PURPOSE:    To force a delay of at least a given amount (in seconds) in
118  *               program execution.
119  *
120  *   INCLUDE:    <conc.h>                 (Logical Systems C, version 89.1)
121  *               "timing.h"
122  *
123  *   CALLS:      ProcGetPriority()        (Logical Systems C, version 89.1)
124  *               Time()                   (Logical Systems C, version 89.1)
125  *               mclock()                 (Intel iPSC/2 C)
126  *
127  *   CALLED BY:
128  *
129  *   PARAMETERS: The (float) argument tells the function the minimum time
130  *               (in seconds) to delay.
131  *
132  *   EXAMPLE:    delay(1.25);
133  *
134  * --------------===============================================------------
135  */
136
137 #ifdef PROTOTYPE
138
139     void delay(float seconds);
140
141 #else
142
143     void delay( /* float seconds */ );
144
145 #endif
146
147
148 /* ------------============= EOF  timing.h   =========------------ */
```

E. GAUSS FACTORIZATION CODE

The Gauss factorization code appears on the pages that follow. First, the code for partial pivoting is given. Since the complete pivoting case was very similar, most of it has been omitted to save space. The pivot election function, however, is shown in a fragment of **gfpcnode.c**, the node code for GF with Pivoting (Complete).

```
 1 # -----------------------------------------------------------------------
 2 #
 3 #    PURPOSE :   Makefile for Hypercube Gauss Factorization (GF) Program
 4 #    AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 5 #    DATE    :   26 August 1991
 6 #
 7 # -----------------------------------------------------------------------
 8
 9 ROOTCODE=gfpphost
10 NODECODE=gfppnode
11 HEADER=gf
12 NIF_FILE=gfpp
13
14
15 # ---------------------         OPTIONS AND DEFINITIONS      ----------------------
16 #
17 #    iPSC/2 Section (MDIR == MatLib directory)
18
19 MDIR=/usr/hartman/matlib/
20
21
22 #    Transputer Section
23 #
24 #    The following section establishes options and definitions, starting
25 #    with PP, the Logical Systems C Preprocessor. The '-dX' option (with no
26 #    macro_expression) is like '#define X 1'. Next the compilation options
27 #    for Logical Systems' TCX Transputer C Compiler are given.  The '-c'
28 #    means compress the output file.  The options beginning with '-p' tell
29 #    TCX to generate code for the appropriate processor:
30 #
31 #         -p2         T212 or T222
32 #         -p25        T225
33 #         -p4         T414
34 #         -p45        T400 or T425
35 #         -p8         T800
36 #         -p85        T801 or T805
37 #
38 #    Logical Systems' TASM Transputer Assembler is next.  The '-c' means
39 #    compress the output file (it can cut it in half)!  The '-t' is used
40 #    because the input to TASM will be from a language translator (TCX's
41 #    output) and not from assembly source code.
42 #
43 #    The final list tells TLNK which libraries to look at during linking.
44 #    It also establishes an entry point.  We use '_main' for the root node
45 #    and '_ns_main' for other nodes.
46
47 PPOPT2=-dPROTOTYPE -dTRANSPUTER -dT212
48 PPOPT4=-dPROTOTYPE -dTRANSPUTER -dT414
49 PPOPT8=-dPROTOTYPE -dTRANSPUTER -dT800
50 TCXOPT2=-cp2
```

```
51 TCXOPT4=-cp4
52 TCXOPT8=-cp8
53 TASMOPT=-ct
54 T2LIB=t2lib.tll
55 T4LIB=matlib4.tll t4lib.tll
56 T8LIB=matlib8.tll t8lib.tll
57 RENTRY=_main
58 NENTRY=_ns_main
59
60
61 # --------------------- DEFAULT ===> MAKE ALL -----------------------
62 #
63 #    Comment out one or the other....
64 #
65 #    all:     ipsc
66 #    run:     irun
67 #    clean:   iclean
68 all:      transputer
69 run:      trun
70 clean:    tclean
71
72
73
74
75 # -------------------------- ROOT CODE -----------------------------
76 #
77 #    iPSC/2 Section
78
79 ipsc: $(ROOTCODE) $(NODECODE)
80
81 $(ROOTCODE): $(ROOTCODE).o
82 cc $(ROOTCODE).o $(MDIR)allocate.o $(MDIR)clargs.o $(MDIR)commhost.o $(MDIR)generate.o
$(MDIR)epsilon.o $(MDIR)io.o $(MDIR)mathx.o $(MDIR)ops.o $(MDIR)timing.o -lm -host
-o $(ROOTCODE)
83
84 $(ROOTCODE).o:  $(ROOTCODE).c  $(HEADER).h
85
86
87 #    Transputer Section
88
89 transputer: $(ROOTCODE).tld $(NODECODE).tld
90
91 $(ROOTCODE).tld: $(ROOTCODE).trl
92 echo FLAG    c                  >  $(ROOTCODE).lnk
93 echo LIST    $(ROOTCODE).map >> $(ROOTCODE).lnk
94 echo INPUT   $(ROOTCODE).trl >> $(ROOTCODE).lnk
95 echo ENTRY   $(RENTRY)       >> $(ROOTCODE).lnk
96 echo LIBRARY $(T4LIB)        >> $(ROOTCODE).lnk
97 tlnk $(ROOTCODE).lnk
98
```

305

```
 99 $(ROOTCODE).trl: $(ROOTCODE).tal
100 tasm $(ROOTCODE).tal $(TASMOPT)
101
102 $(ROOTCODE).tal: $(ROOTCODE).pp
103 tcx  $(ROOTCODE).pp  $(TCXOPT4)
104
105 $(ROOTCODE).pp: $(ROOTCODE).c
106 pp   $(ROOTCODE).c   $(PPOPT4)
107
108
109
110
111
112 # --------------------------- NODE CODE  ----------------------------
113 #
114
115 #   iPSC/2 Section
116
117 $(NODECODE): $(NODECODE).o
118 cc $(NODECODE).o $(MDIR)allocate.o $(MDIR)commnode.o $(MDIR)generate.o $(MDIR)io.o
$(MDIR)mathx.o $(MDIR)ops.o $(MDIR)timing.o -node -lm -o $(NODECODE)
119
120 $(NODECODE).o: $(NODECODE).c $(HEADER).h
121
122
123 #   Transputer Section
124
125 $(NODECODE).tld: $(NODECODE).trl
126 echo FLAG   c                   >   $(NODECODE).lnk
127 echo LIST   $(NODECODE).map >>  $(NODECODE).lnk
128 echo INPUT  $(NODECODE).trl >>  $(NODECODE).lnk
129 echo ENTRY  $(NENTRY)          >>  $(NODECODE).lnk
130 echo LIBRARY $(T8LIB)          >>  $(NODECODE).lnk
131 tlnk $(NODECODE).lnk
132
133 $(NODECODE).trl: $(NODECODE).tal
134 tasm $(NODECODE).tal $(TASMOPT)
135
136 $(NODECODE).tal: $(NODECODE).pp
137 tcx  $(NODECODE).pp  $(TCXOPT8)
138
139 $(NODECODE).pp: $(NODECODE).c
140 pp   $(NODECODE).c   $(PPOPT8)
141
142
143
144
145
146 # --------------------------- EXECUTION  ----------------------------
147 #
```

306

```
148
149 irun: $(ROOTCODE) $(NODECODE)
150 $(ROOTCODE)
151
152 trun: $(ROOTCODE).tld $(NODECODE).tld $(NIF_FILE).nif
153 echo makecube first
154 ld-net $(NIF_FILE) -t -v
155
156
157 # --------------------------    CLEAN  UP   ----------------------------
158 #
159
160 iclean:
161 rm $(NODECODE).o
162 rm $(ROOTCODE).o
163 rm $(NODECODE)
164 rm $(ROOTCODE)
165
166 tclean:
167 del $(ROOTCODE).lnk
168 del $(NODECODE).lnk
169 del $(ROOTCODE).map
170 del $(NODECODE).map
171 del $(ROOTCODE).tal
172 del $(NODECODE).tal
173 del $(ROOTCODE).pp
174 del $(NODECODE).pp
175 del $(ROOTCODE).trl
176 del $(NODECODE).trl
177
178
179 # EOF gfpp.mak ----------------------------------------------------------
```

```
 1 ;   ----------------========    NETWORK INFORMATION FILE   ========-------------
 2 ;
 3 ;    SOURCE  :   gfpp.nif
 4 ;    VERSION :   1.0
 5 ;    DATE    :   14 September 1991
 6 ;    AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7 ;    USAGE   :   ld-net gfpp
 8 ;
 9 ;
10 ;   -------------===========    REFERENCES   ==============-------------
11 ;
12 ;    [1] Inmos. IMS B012 User Guide and Reference Manual. Inmos Limited,
13 ;            1988, Fig. 26, p. 28.
14 ;
15 ;
16 ;   -------------===========    DESCRIPTION   ==============-------------
17 ;
18 ;    Network Information File (NIF) used by Logical Systems C (version 89.1)
19 ;    LD-NET Network Loader.  This file prescribes the loading action to take
20 ;    place when the 'ld-net' command is given as in USAGE above.
21 ;
22 ;
23 ;   ----------------========    HARDWARE PREREQUISITES   ========-------------
24 ;
25 ;    NOTE: There are three node numbering systems: the one created by Inmos'
26 ;    CHECK program, the Gray code labeling, and the NIF labeling.  Since all
27 ;    three will be used on occasion, I will prefix node numbers with a C, G,
28 ;    or N to identify which system I am using!
29 ;
30 ;    The IMS B004 and IMS B012 must be configured correctly. The B004's T414
31 ;    has link 0 connected to the host PC via a serial-to-parallel converter,
32 ;    link 1 connected to the IMS B012 PipeHead, link 2 connected to the T212
33 ;    [communications manager (not used here)] on the B012, and link 3
34 ;    connected to the IMS B012 PipeTail (see [1]).  By the way, link 2 from
35 ;    the B004 goes to the the ConfigUp slot just under the PipeHead slot
36 ;    (this connects it to the T212).  Finally, the B004's Down link must run
37 ;    to the B012's Up link.
38 ;
39 ;
40 ;   ---------------====    SETTING THE C004 CROSSBAR SWITCHES   ====-------------
41 ;
42 ;    Once you have connected the hardware in the fashion mentioned above,
43 ;    the system is ready to be transformed to a hypercube.  Three codes by
44 ;    Mike Esposito are used here: t2.nif, root.tld, and switch.tld.  I have
45 ;    a batch file called 'makecube.bat' that performs a 'ld-net t2' also.
46 ;
47 ;    Mike's code passes instructions to the T212 on the B012; which, in-turn
48 ;    tells the C004's how to connect their switches.  After the code has
49 ;    executed, the (very specific) configuration that we are looking for
50 ;    will exist.  Specifically, the following (output from CHECK /R) is what
```

```
51 ;    this process gives us:
52 ;
53 ;            check 1.21
54 ;              # Part rate Mb Bt [  Link0   Link1   Link2   Link3 ]
55 ;              0 T414b-15 0.09 0 [   HOST     1:1     2:1     3:2 ]
56 ;              1 T800c-20 0.80 1 [    4:3     0:1     5:1     6:0 ]
57 ;              2 T2   -17 0.49 1 [   C004     0:2     ...    C004 ]
58 ;              3 T800c-20 0.80 2 [    7:3     8:2     0:3     9:0 ]
59 ;              4 T800c-20 0.76 3 [    9:3    10:2    11:1     1:0 ]
60 ;              5 T800d-20 0.90 1 [    8:3     1:2    10:1    12:0 ]
61 ;              6 T800d-20 0.76 0 [    1:3    12:2     7:1    11:0 ]
62 ;              7 T800d-20 0.76 3 [   13:3     6:2    14:1     3:0 ]
63 ;              8 T800d-20 0.90 2 [   14:3    15:2     3:1     5:0 ]
64 ;              9 T800c-20 0.77 0 [    3:3    13:2    15:1     4:0 ]
65 ;             10 T800d-20 0.90 2 [   16:3     5:2     4:1    15:0 ]
66 ;             11 T800d-20 0.90 1 [    6:3     4:2    16:1    13:0 ]
67 ;             12 T800d-20 0.77 0 [    5:3    16:2     6:1    14:0 ]
68 ;             13 T800d-20 0.77 3 [   11:3    17:2     9:1     7:0 ]
69 ;             14 T800c-20 0.90 1 [   12:3     7:2    17:1     8:0 ]
70 ;             15 T800c-20 0.90 2 [   10:3     9:2     8:1    17:0 ]
71 ;             16 T800c-20 0.76 3 [   17:3    11:2    12:1    10:0 ]
72 ;             17 T800d-20 0.88 2 [   15:3    14:2    13:1    16:0 ]
73 ;
74 ;    Here node C0 is the root transputer (on the IMS B004) and node C2 is
75 ;    the T212 (on the IMS B012).  The other sixteen nodes are the T800's
76 ;    that are used for the work.  A logical interconnection topology is
77 ;    described below.
78 ;
79 ;
80 ;    --------------================  TOPOLOGY  ================--------------
81 ;
82 ;    The physical interconnection scheme described above is an actual 4-cube
83 ;    with one exception.  The root node (C0) is situated BETWEEN nodes C1
84 ;    and C3 (which would be connected directly in the usual 4-cube).  This
85 ;    gives us two 3-cubes:  one whose node labeling is G0xxx and the other,
86 ;    whose node labeling is G1xxx (where the xxx represents all permutations
87 ;    of 3-bits).  These are the usual three cubes, and they will exist if we
88 ;    define the node numbering/labeling correctly.
89 ;
90 ;
91 ;    --------------================  STRATEGY  ================--------------
92 ;
93 ;    The node labeling established by the NIF is available via the variable
94 ;    _node_number (see <conc.h>) in source code.  Therefore, we would like a
95 ;    smart labeling scheme in the NIF file so that programming is easier.
96 ;    This, of course, is subject to the restriction that NIF labels begin
97 ;    with N1 and so on.
98 ;
99 ;    One such method would be to define a NIF labeling so that the Gray code
100 ;   label for a node would be (_node_number - 2).  In fact, this is
```

309

```
101 ;    possible and the adjacencies defined below allow us to realize this
102 ;    feature.  Below, node N0 is the host PC, node N1 is the root transputer
103 ;    (T414 on the B004), N2 through N17 correspond to G0 through G15 (the
104 ;    nodes of a 4-cube), and N18 is not used (but it's the T212).
105 ;
106 ;    -----------------=================================================-----------
107
108
109 host_server cio.exe; (default)
110
111 ;              TRANSPUTER      RESET     DESCRIPTION OF LINK CONNECTIONS
112 ; NODE         LOADABLE        COMES     --------------------------------
113 ;  ID          CODE (.tld)     FROM:     LINK0    LINK1    LINK2    LINK3
114 ; ====         ===========     =====     =====    =====    =====    =====
115    1,         gfpphost,       r0,         0,       2,        ,       10; B004
116    2,         gfppnode,       r1,         4,       1,       3,        6; B012
117    3,         gfppnode,       r2,        11,       2,       5,        7;
118    4,         gfppnode,       r5,        12,       5,       8,        2;
119    5,         gfppnode,       r3,         9,       3,       4,       13;
120    6,         gfppnode,       r7,         2,       7,      14,        8;
121    7,         gfppnode,       r9,         3,       9,       6,       15;
122    8,         gfppnode,       r4,         6,       4,       9,       16;
123    9,         gfppnode,       r8,        17,       8,       7,        5;
124   10,         gfppnode,       r11,       14,      11,       1,       12;
125   11,         gfppnode,       r13,       15,      13,      10,        3;
126   12,         gfppnode,       r16,       10,      16,      13,        4;
127   13,         gfppnode,       r12,        5,      12,      11,       17;
128   14,         gfppnode,       r6,        16,       6,      15,       10;
129   15,         gfppnode,       r14,        7,      14,      17,       11;
130   16,         gfppnode,       r17,        8,      17,      12,       14;
131   17,         gfppnode,       r15,       13,      15,      16,        9;
132 ; 18,         switch,         s1,          ,       1,        ,         ; T212
133
134
135 ; --------------============== EOF gfpp.nif ==================------------
```

310

```
 1  /* -------------========== PROGRAM INFORMATION ==========-------------
 2  *
 3  *   SOURCE  :   gf.h
 4  *   VERSION :   2.5
 5  *   DATE    :   21 September 1991
 6  *   AUTHOR  :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  *   SEE ALSO:   gfpc.mak        makefile for the complete pivoting case
 9  *               gfpp.mak        makefile for the partial  pivoting case
10  *               gfpchost.c      host code for the complete pivoting case
11  *               gfpphost.c      host code for the partial  pivoting case
12  *               gfpcnode.c      node code for the complete pivoting case
13  *               gfppnode.c      node code for the partial  pivoting case
14  *
15  *
16  * -------------============== REFERENCES  ==================-------------
17  *
18  *   [1] Gragg, William B.  MATLAB code and personal conversations, 1991.
19  *
20  *
21  * -------------============== DESCRIPTION  ==========-------------
22  *
23  *   This header file is shared by several programs (listed above).  Each of
24  *   these codes has something to do with a parallel implementation of Gauss
25  *   Factorization (GF).  Several pivoting strategies are supported.  Files
26  *   like gfpc*.* represent a COMPLETE pivoting strategy, and the files like
27  *   gfpp*.* give the corresponding code for the PARTIAL pivoting scheme.
28  *
29  *   The basic algorithm is from [1].  Parallelism is sought by distributing
30  *   the columns of A across the nodes of a multiprocessor system (using the
31  *   hypercube interconnection topology).  The program is designed for the
32  *   Intel iPSC/2 or a network of Inmos transputers.
33  *
34  *   The algorithm factors Q'AP = LU with P and Q permutation matrices, L
35  *   unit lower trapezoidal (r columns) and U upper trapezoidal with nonzero
36  *   diagonal elements (r rows).  The program is designed for a general
37  *   matrix, A.  It does not assume A square or sparse.  There is no effort
38  *   to optimize for this, or any other, special structure.  There is one
39  *   caveat:  I designed the code to gather data for square matrices of full
40  *   rank.  Therefore, I have tested the square case of random matrices very
41  *   carefully.  While the code should work for any general matrix, it has
42  *   not been carefully tested in other cases.  Additionally, since I sought
43  *   timing data for matrices of full rank, I have NOT addressed the problem
44  *   of gathering columns (back to the host) to the right of the final pivot
45  *   for rank-deficient matrices.  This would not be a difficult task, but I
46  *   did not make this effort since it has no bearing on my goal.
47  *
48  *   In the partial pivoting code, the search for pivots is carried out only
49  *   in the pivot column, so P is the identity (i.e., there are no column
50  *   interchanges).  Many of the remaining comments pertain to the complete
```

311

```
51  *   pivoting case, since it is the most challenging.  The changes for the
52  *   partial pivoting case should be evident in most cases.  At times, when
53  *   the changes are not necessarily evident, clarifying remarks address the
54  *   partial pivoting scheme.  This header file contains the majority of the
55  *   background and algorithm information, but if you're after a careful
56  *   study of the differences, compare the source codes. The algorithm below
57  *   gives a road map through the code.
58  *
59  *   ----------------===============================================---------------
60  */
61
62
63
64  /* -------------------============   ALGORITHM: BACKGROUND   =========-------------
65  *
66  *   1.) Preliminaries.  Consider A (m x n), a matrix of real numbers.  The
67  *   permutation vectors, p and q, characterize column and row permutations
68  *   (respectively).  The scalar, (g/a), is the growth factor.  The integer,
69  *   r, is a fairly reasonable determination of the 'numerical rank' of A.
70  *   The C language convention is followed, numbering rows and columns from
71  *   zero; and storing dynamic, two-dimensional arrays (matrices) in row-
72  *   major-order.  The 'pivot' will be that element located at A(k,k).  The
73  *   area (in A) below and to the right of the pivot [all A(i,j) where i > k
74  *   and j > k ] is called the 'Gauss transform area'.
75  *
76  *   2.) Communications and Coordination.  Let N be the number of processors
77  *   (workers) in the hypercube.  These nodes are labeled with a Gray code
78  *   { 0 .. (N - 1) }.  The root (host) node distributes the columns of A to
79  *   the nodes.  This is done cyclically, using the C modulus operator (%).
80  *   That is, column j will be sent to processor (j mod N).  Once the nodes
81  *   have their columns, they begin work.  Communication (for the complete
82  *   pivoting case) involves an election process for the next pivot, where
83  *   each of the nodes finds its best candidate and then the election finds
84  *   the best candidate in the global picture.  This is done in lg(N) steps
85  *   using the cubecast_from() function.
86  *
87  *   The partial pivoting case does not require the election process that
88  *   complete pivoting needs, but both methods look similar (in terms of
89  *   communication) after the elections are complete.  The node holding the
90  *   pivot column must perform the pivot column arithmetic and distribute
91  *   the resulting pivot column (also in lg(N) steps) to the other nodes.
92  *   Communications functions are not explained much in this code, but
93  *   details can be found in the files comm.h & comm.c.
94  *
95  *   3.) Pivoting Strategy.  The complete pivoting strategy's election
96  *   process (at each stage), determines the element in (the entire Gauss
97  *   transform area of) A that is largest in absolute value.  This element
98  *   wins the election and is 'moved' to A(k,k) for the upcoming stage.  It
99  *   isn't really moved...but p and q are updated so that we can keep track
100 *   of permutations.  During the search for the new pivot, candidates are
```

```
101  *   denoted A(s,t) = u.  The largest of the candidates is installed as the
102  *   next pivot.  There seems to be too much overhead associated with this
103  *   fancy indexing off of p[] and q[].  For the partial pivoting code, I
104  *   chose to ACTUALLY SWAP rows (if necessary) at each stage.  This makes
105  *   the 'pp' code a bit easier to read.
106  *
107  *   4.) Stopping.  The GF process is repeated until one of two criteria is
108  *   satisfied.  First, of course, we may run out of matrix.  Secondly, we
109  *   may find a pivot whose absolute value is less than our tolerance (tol).
110  *   In the latter case, we have a rank-deficient A.  Currently, the codes
111  *   recognize rank-deficiency and bail out of the iteration loop; but they
112  *   do not gather (to the host) all of the remaining columns to the right
113  *   of the last pivot.  This is discussed above.
114  *
115  *
116  *   -----------------========    ALGORITHM: THE GF PROCESS   ====================
117  *
118  *   0.) Initialization.  Let dim be the dimension of the hypercube.  Let
119  *   k = 0.  Search A and find the largest (in absolute value) element, u.
120  *   This is done at each node.  Once each node has a local candidate for
121  *   the next pivot, an election is held, dimension-by-dimension.  This
122  *   requires (dim) steps, and when it is finished, every processor knows
123  *   exactly the position and value of the next pivot.  Exception:  In the
124  *   partial pivoting code, the processor which has the pivot column simply
125  *   searches the (proper part of the) pivot column for the next pivot and
126  *   then informs the other processors.
127  *
128  *   1.) Status.  Every node knows the position and value of the next pivot,
129  *   namely u = A(s,t); and where it should be installed, A(k,k). The growth
130  *   rate is adjusted:  g = max[g, abs(u)].  If (u < tol), then A is rank-
131  *   deficient and we exit the loop (using the C 'break' statement).
132  *
133  *   2.) Permutations.  We account for the interchange of rows s and k and
134  *   columns t and k by swapping the elements of p[] that are indexed by k
135  *   and t and swapping the elements in q[] indexed by k and s.  This
136  *   (effectively) establishes the new pivot at A(k,k).  The column permu-
137  *   tation vector has no significance in the partial pivoting case since
138  *   it would never be changed.  The matrix, P, in this case, is simply the
139  *   identity.
140  *
141  *   3.) Adjust the Gauss Transform Area.
142  *
143  *       (a) In the (single) node that holds the new pivot's column (k),
144  *       divide every element below the pivot by the pivot value.  Broadcast
145  *       this column to every other node.  Node 0 updates the manager, who
146  *       uses this information to append to his copy of the resulting
147  *       (factored) A.
148  *
149  *       (b) Now every worker has the updated column k.  At every node, do
150  *       the following:  For every element A(i,j) [ where i > k and j > k ]
```

313

```
151  *        let A(i,j) = A(i,j) - (A(i,k) * A(k,j)).
152  *
153  * 4.) Pivot Search.  In the Gauss transform area, G, search for the
154  * element that is largest in absolute value.  Its position is A(s,t) and
155  * its value is u.  The candidates are chosen at the local (processor)
156  * level, then an election is held at the global level to determine the
157  * best candidate in the same manner that was described in step 0.
158  * Increment k.  Repeat the process (go back to step 1).  The obvious
159  * exceptions apply to the partial pivoting case.
160  *
161  *
162  * ---------------=========== NOTES FOR IMPROVEMENT  ========----------------
163  *
164  * Currently the code does not give full support for rank-deficiency.  It
165  * DOES break out of the loop, but everything to the right of the final
166  * pivot column will be garbage.  It would be relatively easy to add the
167  * necessary post-iteration rank-deficiency check and coalesce each of the
168  * remaining columns back to the manager, but this code was created to
169  * test the full-rank cases and take performance data.
170  *
171  * Secondly, there is the issue of whether it is better for the manager to
172  * receive each pivot column as it becomes available, or if all columns
173  * should be sent in at the end.  I'm not yet sure which method is better,
174  * but the current code keeps the root node up-to-date at each stage. This
175  * is probably the best solution to the problem above and would probably
176  * enhance performance during the iterations!  It REALLY SHOULD BE TESTED!
177  *
178  * There are many other questions that pertain to optimization that remain
179  * unanswered (especially in the complete pivoting case).
180  *
181  * ----------------==================================================-------------
182  */
183
184
185
186
187
188 /* ---------------=========== ALGORITHM: CONCLUSION  ========----------------
189  *
190  * 1.) Rank.  Set r, the rank of A, equal to the number of iterations that
191  *     were executed.  This is automatic in the manager (host) code since
192  * the integer, r, is used as the loop index.  The worker nodes use k for
193  * a loop index variable.
194  *
195  * 2.) Interchanges.  Row and column interchanges are not actually done in
196  * the complete pivoting code.  Instead, we maintain permutation vectors,
197  * p[] and q[].  You may note that while both vectors are used heavily
198  * during the GF process q[], in particular, comes in handy at the end to
199  * set A in order.  The partial pivoting code performs the actual inter-
200  * changes of rows.  At first, we would be inclined to believe that the
```

314

```
201  *   indexing by p[] and q[] leads to better performance, but there is no
202  *   clear timing evidence (at this point) that supports this idea.
203  *
204  *   3.) Factors.  The upper trapezoidal matrix, U, is the upper trapezoid
205  *   of (the resulting, factored) A (the diagonal of A and everything above
206  *   that).  The lower trapezoidal matrix, L, is formed by placing ones on
207  *   the diagonal of A; zeros above; and copying the lower trapezoid of A
208  *   (excluding the diagonal).  To form Q'AP, we use THE ORIGINAL copy of A
209  *   (not the factored, resulting A) and the matrices Q and P that are
210  *   implied by q[] and p[].  That is, in the end, we set Q[q[i]][i] = 1.0
211  *   for all i in { 0, 1, ..., (m-1) } and set P[p[j]][j] = 1.0 for all j
212  *   in { 0, 1, ..., (n-1) }.
213  *
214  *   ------------------------------------------------------------------------
215  */
216
217
218
219
220
221 /* ---------------------==========   MANIFEST  CONSTANTS   ==========-------------------
222  *
223  *
224  *   Section 1:  Communications Aids (Message Types and Type Selectors)
225  *
226  *   The following manifest constants simplify the communications effort.
227  *   The TRANSPUTER section is fairly general in nature.  The iPSC/2 section
228  *   specifies types and type selectors for csend() and crecv().  It IS
229  *   SIGNIFICANT that NODE_OFFSET is the largest of these.  It must remain
230  *   the largest so that (for all nodes n) the value of (n + NODE_OFFSET)
231  *   cannot be equal to one of the other message types (consider n == 0).
232  *
233  *   ------------------------------------------------------------------------
234  */
235
236
237 #ifdef TRANSPUTER
238
239 #define CUBESIZE   8         /*  change these for a cube of other dim    */
240 #define DIMENSION  3
241
242 #else /* iPSC/2 */
243
244 #define ARG_TYPE          1   /*  for passing command line argument info  */
245 #define COL_SIZE_TYPE     2   /*  for sending n part of size(A) ==> cols   */
246 #define COL_TYPE          3   /*  use this to send a column               */
247 #define PIVOT_TYPE        4   /*  candidate for next pivot                 */
248 #define PCOL_TYPE         5   /*  use this to send a pivot column          */
249 #define ROW_SIZE_TYPE     6   /*  for sending m part of size(A) ==> rows   */
250 #define NODE_OFFSET       7   /*  for sending messages from nodes          */
```

315

```
251
252 #endif
253
254
255 /* -----------------------------------------------------------------------
256  *
257  *   Section 3:  Timing
258  *
259  *   The root uses a two-dimensional array where the rows are indexed by the
260  *   node numbers and the columns use the following indexing.  The nodes, of
261  *   course, only need a one-dimensional array with indexing according to
262  *   the following scheme.  There a total of MAX_EVENTS elements in the
263  *   array, and indexing for a specific event is given by START_TIME, SETUP,
264  *   and so on.  The partial pivoting case does not use all of the events.
265  *
266  * -----------------------------------------------------------------------
267  */
268
269
270 #define MAX_EVENTS       18   /*  number of events that we want to time    */
271
272
273 #define DATA_SOURCE       0   /*  node number of source of the data        */
274 #define START_TIME        1   /*  t(0) ==> starting time for the node      */
275 #define SETUP             2   /*  from t(0) until starting to receive cols */
276 #define DISTRIB_COLS      3   /*  time to distribute columns               */
277 #define FIRST_PIVOT       4   /*  from receipt of last col to start iter   */
278
279 /*  The next two only apply to nodes zero and eight                        */
280 #define PCOLS_TO_HOST     5   /*  time spent passing pivot cols to host    */
281 #define PIVOTS_TO_HOST    6   /*  time spent passing pivots to host        */
282
283 /*  The next five kind of represent the big picture                        */
284 #define PIVOT_ELECTION    7   /*  time spent on pivot elections            */
285 #define UPDATING_PQ       8   /*  time spent updating permutations p and q */
286 #define PCOL_ARITHMETIC   9   /*  time spent on pivot column arithmetic    */
287 #define PCOL_DISTRIB     10   /*  time spent distributing pivot columns    */
288 #define UPDATING_G       11   /*  time spent updating the Gauss transform  */
289
290 /*  The next four are times from within update_G()                         */
291 #define PRLTIME          12   /*  pivot row location time                  */
292 #define LCTIME           13   /*  time to determine if a column is local   */
293 #define G_ARITHMETIC     14   /*  time spent on arithmetic within G        */
294 #define LOOPTIME         15   /*  time for both for() loops in update_G()  */
295
296 /*  The last two are back at the big picture level again                   */
297 #define ITERATION        16   /*  time checked before and after iteration  */
298 #define STOP             17   /*  the last time sampled by the node        */
299
300
```

316

```
301
302
303
304  /* -------------------------------------------------------------------------
305   *
306   *   Section 4:  General
307   *
308   * -------------------------------------------------------------------------
309   */
310
311  #define AFT            4   /*  number of digits to print after decimal  */
312  #define WIDTH          6   /*  number of characters (including decimal) */
313
314
315
316
317
318  /* -------------------------------------------------------------------------
319   *
320   *   Section 5:  A special flag used for the id field of a pivot.  When it
321   *               appears, it indicates that the sending node's part of A has
322   *   no elements as big as the tolerance, tol; and therefore this node's
323   *   candidate for pivot should not be considered.
324   *
325   * -------------------------------------------------------------------------
326   */
327
328
329  #define RANK_DEFICIENT -1
330
331
332
333
334
335  /* -------------============   TYPE  DEFINITIONS   ============------------- */
336
337
338  typedef struct {
339
340      int     id;
341      double  u;
342      int     s,
343              t;
344
345  } Pivot_Type;
346
347
348  /* -------------===============  EOF gf.h  ===============------------- */
```

317

```
 1 /* -------------========== PROGRAM INFORMATION ====================------------
 2  *
 3  *  SOURCE   :   gfpphost.c
 4  *  VERSION  :   2.0
 5  *  DATE     :   21 September 1991
 6  *  AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  *
 8  * -------------================ DESCRIPTION ========================------------
 9  *
10  * Gauss Factorization (GF) with Partial Pivoting:  Parallel Version.
11  * This is the manager portion of the code.  See [gf.h] for details.
12  *
13  * --------------========================================================------------
14  */
15
16 #include <stdio.h>
17 #include <string.h>
18
19 #ifdef  TRANSPUTER
20
21 #include <conc.h>
22 #include <stdlib.h>                          /*  addfree(), _heapend        */
23
24 #include <matrix.h>
25 #include <macros.h>
26 #include <allocate.h>
27 #include <clargs.h>
28 #include <comm.h>
29 #include <epsilon.h>
30 #include <generate.h>
31 #include <io.h>
32 #include <ops.h>
33 #include <timing.h>
34
35 #else   /* iPSC/2 */
36
37 #include "/usr/hartman/matlib/matrix.h"
38 #include "/usr/hartman/matlib/macros.h"
39 #include "/usr/hartman/matlib/allocate.h"
40 #include "/usr/hartman/matlib/clargs.h"
41 #include "/usr/hartman/matlib/comm.h"
42 #include "/usr/hartman/matlib/epsilon.h"
43 #include "/usr/hartman/matlib/generate.h"
44 #include "/usr/hartman/matlib/io.h"
45 #include "/usr/hartman/matlib/ops.h"
46 #include "/usr/hartman/matlib/timing.h"
47 #endif
48
49 #include "gf.h"
50
```

318

```
51
52
53 /* ------------============    MANIFEST  CONSTANTS    ===========------------
54  *
55  *   The following manifest constants are used to determine the size of the
56  *   option list, optv[]; indexing associated with valid command line
57  *   arguments; and selection constants for the user's choice of matrix type
58  *   [used in generate()].
59  *
60  */
61
62 #define NUMBER_OF_ARGS          3               /*  -d -t -v                    */
63
64 #define DIM                     0               /*  index into optv[]           */
65 #define TIMING                  1               /*   "    "     "               */
66 #define VERBOSE                 2               /*   "    "     "               */
67
68 #define SELECT_QUIT             0               /*  menu / matrix selection     */
69 #define SELECT_IDENTITY         1
70 #define SELECT_HILBERT          2
71 #define SELECT_RANDOM           3
72 #define SELECT_WILKINSON        4
73
74
75
76
77
78 /* ------------================    GLOBALS    ================------------ */
79
80
81 static char version[] = "Parallel GF with Partial Pivoting, Version 2.0";
82
83
84 #ifdef TRANSPUTER
85
86 Channel *ic[(CUBESIZE + 1)],
87         *oc[(CUBESIZE + 1)];
88
89 #else /*  iPSC/2  */
90
91 static char *cubename;
92
93 static char *nodecode = "gfppnode";
94
95 #endif /* TRANSPUTER */
96
97
98 static Arg_Struct *optv[NUMBER_OF_ARGS];
99
100
```

319

```
101
102
103
104  /* ------------========== FUNCTION DEFINITION ==========------------
105  *
106  *  The structure is defined more carefully in clargs.h, but the basic idea
107  *  is that we have an array of pointers to type Arg_Struct...in this case,
108  *  there are NUMBER_OF_ARGS valid arguments and the next few steps take
109  *  care of allocation and definition of them.  The -d argument allows the
110  *  user to enter the desired dimension of the hypercube, -t sets timing on
111  *  and -v is used to set verbose on.
112  */
113
114  void define_valid_args() {
115
116      static int interpret[] = { LONG };
117
118
119      install_complex_arg(DIM,     optv, "-d", interpret, 1);
120
121      install_simple_arg(TIMING, optv, "-t");
122      install_simple_arg(VERBOSE, optv, "-v");
123
124  }
125  /* End define_valid_args() ---------------------------------------- */
126
127
128
129
130
131  /* ------------========== FUNCTION DEFINITION ==========------------
132  *
133  *  A simple function to display the results....
134  */
135
136  #ifdef PROTOTYPE
137
138      void display_timing_data(Double_Matrix_Type *A,
139                               int            dim,
140                               double         a,
141                               double         eps,
142                               double         g,
143                               double         tol,
144                               int            r,
145                               double         **t)
146
147  #else
148
149      void display_timing_data(A, dim, a, eps, g, tol, r, t)
150
```

320

```
151         Double_Matrix_Type *A;
152         int              dim;
153         double           a,
154                          eps,
155                          g,
156                          tol;
157         int              r;
158         double           **t;
159
160 #endif
161 {
162     int aft,
163         cubesize = pow2(dim),
164         i,
165         m = A->rows,
166         n = A->cols,
167         width;
168
169
170 #ifdef TRANSPUTER  /* is measured in 64 microsecond ticks ==> 4-5 places */
171
172     aft   = 5;
173     width = 15;
174
175 #else  /* iPSC/2 is measured in milliseconds ==> three places*/
176
177     aft   = 3;
178     width = 13;
179
180 #endif
181
182     printf("-----------------========= TIMING  DATA  =========------");
183     printf("--------\n\n");
184
185     printf("                Hypercube of order %d ", dim);
186     (dim == 0) ? (printf("(1 processor)\n\n")) :
187                  (printf("(%d processors)\n\n", cubesize));
188
189     printf("Problem size ==> size(A) = (%d x %d).\n", m, n);
190     printf("Machine precision:  eps = %e\n", eps);
191     printf("Tolerance:          tol = %e\n", tol);
192     printf("Growth factor:      g/a = %e\n", (g/a));
193     printf("Rank:             rank(A) =%3d\n", r );
194     printf("Units for timing data:  = seconds\n");
195
196     for (i = 0; i < cubesize; i++) {
197
198         printf("\nNode %2d Data -----------------------------", i);
199         printf("-----------------------------\n\n");
200
```

321

```
201          printf("Setup and initialization:                      ");
202          printf("%*.*lf", width, aft, t[i][SETUP]);
203          printf("\nInitial column distribution:                 ");
204          printf("%*.*lf", width, aft, t[i][DISTRIB_COLS]);
205
206          if (i == 0) {
207
208              printf("\nTransmission of pivot columns to the host:    ");
209              printf("%*.*lf", width, aft, t[i][PCOLS_TO_HOST]);
210              printf("\nTransmission of pivots to the host:           ");
211              printf("%*.*lf", width, aft, t[i][PIVOTS_TO_HOST]);
212          }
213
214          printf("\nPerformance of pivot column arithmetic:       ");
215          printf("%*.*lf", width, aft, t[i][PCOL_ARITHMETIC]);
216          printf("\nDistribution of pivot columns:                ");
217          printf("%*.*lf", width, aft, t[i][PCOL_DISTRIB]);
218          printf("\nPerformance of updates and arithmetic in G:   ");
219          printf("%*.*lf", width, aft, t[i][UPDATING_G]);
220          printf("\nUpdate_G():  loop time including arithmetic:   ");
221          printf("%*.*lf", width, aft, t[i][LOOPTIME]);
222
223          printf("\n\nTime for all work inside main iteration loop: ");
224          printf("%*.*lf", width, aft, t[i][ITERATION]);
225          printf("\nTotal time from start to stop:                ");
226          printf("%*.*lf\n\n", width, aft, (t[i][STOP]-t[i][START_TIME]));
227      }
228
229 }
230 /* End display_timing_data() --------------------------------------------- */
231
232
233
234
235
236 /* -----------========== FUNCTION DEFINITION ==================-----------
237  *
238  *  This function distributes the columns of A to the nodes of the hyper-
239  *  cube. The loop variable, j, designates each column of A in turn. The
240  *  column buffer, cbuf[], copies from A the column to be transmitted.
241  *  After cbuf[] is filled, [i = (j mod cubesize)] means that node i will
242  *  get column j and the modulus operation seems to be a reasonable and
243  *  efficient scheme of distribution. Finally, the call to send() ships
244  *  the column out to the appropriate node.
245  *
246  * -----------------------------------------------------------------------
247  */
248
249 #ifdef PROTOTYPE
250
```

322

```
251      void distribute_columns(Double_Matrix_Type *A, int dim, double *cbuf)
252
253 #else
254
255      void distribute_columns(A, dim, cbuf)
256
257          Double_Matrix_Type *A;
258          int                 dim;
259          double              *cbuf;
260
261 #endif
262 {
263
264      int i,
265          j,
266          pos    = 42,                    /*  position of print head      */
267          rm     = LINE_LENGTH - 10;      /*  right margin (see matrix.h)  */
268
269      long cubesize   = pow2(dim),
270           sizeof_col = (long) (A->rows * sizeof(double));
271
272
273      printf("Distributing the columns of A to the nodes");
274
275      for (j = 0; j < A->cols; j++) {
276
277          for (i = 0; i < A->rows; i++) { cbuf[i] = A->matrix[i][j]; }
278
279
280          i = j % cubesize;                /*  column --> node i            */
281
282 #ifdef TRANSPUTER                          /*  node 0 has to sort 'em out   */
283
284          if (i < 8) {
285
286              send(0, (char *) cbuf, sizeof_col, cubesize);
287          }
288          else {
289
290              send(8, (char *) cbuf, sizeof_col, cubesize);
291          }
292
293 #else   /* iPSC/2 */
294
295          send(i, (char*) cbuf, sizeof_col, COL_TYPE);
296
297 #endif /* TRANSPUTER */
298
299          printf(".");
300
```

323

```
301         if (pos++ > rm) {
302
303             pos = 0;
304             printf("\n");
305         }
306
307     }
308
309     printf("\nColumn distribution complete.\n\n");
310
311 }
312 /* End distribute_columns() --------------------------------------------- */
313
314
315
316
317
318 /* --------------============ FUNCTION DEFINITION ============------------
319  *
320  * This function prompts the user for matrix size and type, then generates
321  * the matrix with a call to a function from generate.c.
322  */
323
324
325 #ifdef PROTOTYPE
326
327     Double_Matrix_Type *generate(int *m, int *n)
328
329 #else
330
331     Double_Matrix_Type *generate(m, n)
332
333         int                 *m,
334                             *n;
335 #endif
336 {
337     Double_Matrix_Type *A;
338
339     int matrix_type,
340         valid       = FALSE;
341
342
343     printf("Please enter the number of rows in A: ");
344     scanf("%d", m);
345     fflush(stdin);
346
347     printf("\n......and the number of columns in A: ");
348     scanf("%d", n);
349     fflush(stdin);
350
```

```
351      printf("\n\nSelect from the following list of matrices:");
352
353      while (!valid) {
354
355          printf("\n\n");
356          printf("    %d.) QUIT      \n", SELECT_QUIT    );
357          printf("    %d.) Identity \n", SELECT_IDENTITY );
358          printf("    %d.) Hilbert  \n", SELECT_HILBERT  );
359          printf("    %d.) Random   \n", SELECT_RANDOM   );
360          printf("    %d.) Wilkinson \n", SELECT_WILKINSON);
361          printf("\n>");
362          scanf("%d", &matrix_type);
363          fflush(stdin);
364
365          switch(matrix_type) {
366
367              case SELECT_IDENTITY  :
368              case SELECT_HILBERT   :
369              case SELECT_RANDOM    :
370              case SELECT_WILKINSON :     valid = TRUE;   break;
371
372              case SELECT_QUIT      :     exit(EXIT_SUCCESS);
373          }
374
375      } /* end while() */
376
377
378      switch(matrix_type) {
379
380          case SELECT_IDENTITY:
381
382              printf("\n\nGenerating A = identity(%d, %d).\n\n", *m, *n);
383
384              A = identity(*m, *n);
385              break;
386
387          case SELECT_HILBERT:
388
389              printf("\n\nGenerating A = hilbert(%d, %d).\n\n", *m, *n);
390
391              A = hilbert(*m, *n);
392              break;
393
394          case SELECT_RANDOM:
395
396              printf("\n\nGenerating A = mxrand(%d, %d).\n\n", *m, *n);
397
398              A = mxrand(*m, *n);
399              break;
400
```

325

```
401         case SELECT_WILKINSON:
402
403             printf("\n\nGenerating A = wilkinson(%d, %d).\n\n", *m, *n);
404
405             A = wilkinson(*m, *n);
406             break;
407     }
408
409
410     if (!A) {
411
412         printf("generate():  Allocation failure for the matrix A.\n");
413         exit(EXIT_FAILURE);
414     }
415
416     return(A);
417
418 }
419 /* End generate() -------------------------------------------------- */
420
421
422
423
424
425 /* ----------=========== FUNCTION DEFINITION  ====================----
426  *
427  *  Collect timing data from the nodes.  The Intel side of this function
428  *  takes advantage of the host's ability to receive from any node.  The
429  *  transputer side must receive every node's information from nodes zero &
430  *  eight (eight only becomes involved in the case of the hybrid 4-cube).
431  */
432
433 #ifdef PROTOTYPE
434
435     double **receive_timing_data(int cubesize)
436
437 #else
438
439     double **receive_timing_data(cubesize)
440
441         int     cubesize;
442
443 #endif
444 {
445     double  **dt;                        /*  (double) version of t[][]   */
446
447     int     i,
448             j;
449
450     long    tlen;                        /*  length of one node's data   */
```

326

```
451
452     ticks    **t;                        /*  raw timing data from nodes   */
453
454
455     /*
456      *  Perform allocation for the timing dt t[][].  The two-dimensional
457      *  array is indexed by node number for the rows and by event for the
458      *  columns.  For instance, t[i][j] means the time required for event
459      *  j at node i.  Actually, there is an extra row reserved at the end
460      *  of t[][] for totals:  t[cubesize][j] gives the total time for event
461      *  j across all nodes.
462      */
463
464     if (!(dt = (double **) malloc((cubesize+1) * sizeof(double*)))){
465
466         printf("receive_timing_data(): Allocation failure for dt[][].\n");
467         exit(EXIT_FAILURE);
468     }
469
470     for (i = 0; i < (cubesize + 1); i++) {
471
472         if (!(dt[i] = (double *)calloc(MAX_EVENTS,sizeof(double)))){
473
474             printf("Host:  Allocation failure for dt[%d].\n", i);
475             exit(EXIT_FAILURE);
476         }
477     }
478
479     if (!(t = (ticks **) malloc((cubesize+1) * sizeof(ticks*)))) {
480
481         printf("receive_timing_data(): Allocation failure for t[][].\n");
482         exit(EXIT_FAILURE);
483     }
484
485     for (i = 0; i < (cubesize + 1); i++) {
486
487         if (!(t[i] = (ticks *) calloc(MAX_EVENTS, sizeof(ticks)))) {
488
489             printf("Host:  Allocation failure for t[%d].\n", i);
490             exit(EXIT_FAILURE);
491         }
492     }
493
494     printf("Receiving timing data from the nodes");
495
496     tlen = (long) (MAX_EVENTS * sizeof(ticks));
497
498     for (i = 0; i < cubesize; i++) {
499
500         printf(".");
```

327

```
501
502 #ifdef TRANSPUTER
503
504         if (i < 8) receive(0, (char *) t[i], tlen, cubesize);
505         else       receive(8, (char *) t[i], tlen, cubesize);
506
507 #else  /* iPSC/2 */
508
509         receive(i, (char *) t[i], tlen, (i + NODE_OFFSET));
510
511 #endif /* TRANSPUTER */
512     }
513
514     printf("\n\n");
515
516
517     /*  Calculate totals, averages; place totals in t[cubesize] first....
518      *  then copy to dt[][] and record averages in dt[cubesize].
519      */
520
521     for (i = 0; i < cubesize; i++) {
522
523         for (j = 0; j < MAX_EVENTS; j++) t[cubesize][j] += t[i][j];
524     }
525
526     /*  Fill dt[][] with double values (in seconds).  The conversion
527      *  factors are borrowed from timing.h.
528      */
529
530     for (i = 0; i <= cubesize; i++) {
531
532         dt[i][DATA_SOURCE] = (double) t[i][DATA_SOURCE];
533
534         for (j = START_TIME; j < MAX_EVENTS; j++) {
535
536 #ifdef TRANSPUTER
537
538             dt[i][j] = ((double) t[i][j]) * LO_PERIOD;
539
540 #else
541
542             dt[i][j] = ((double) t[i][j]) * M_PERIOD;
543
544 #endif
545         }
546     }
547
548     /*  Convert totals to averages in dt[cubesize]                       */
549
550     for (j = START_TIME; j < MAX_EVENTS; j++) {
```

328

```
551
552          dt[cubesize][j] /= ((double) cubesize);
553      }
554
555
556      for (i = 0; i < (cubesize + 1); i++) free(t[i]);
557      free(t);
558
559      return(dt);
560 }
561 /* End receive_timing_data() ------------------------------------- */
562
563
564
565
566
567 /* ---------============= FUNCTION DEFINITION ============---------
568  *
569  *  This function analyzes the command line that the user supplied and sets
570  *  variables accordingly.  The valid arguments are given by define_valid_
571  *  args(), and the real work is passed off to interpret_args(), from the
572  *  clargs library.
573  */
574
575 #ifdef PROTOTYPE
576
577      void resolve_args(int argc, char *argv[],
578                        int *dim, int *timing, int *verbose)
579
580 #else
581
582      void resolve_args(argc, argv, dim, timing, verbose)
583
584          int  argc;
585          char *argv[];
586          int  *dim,
587               *timing,
588               *verbose;
589
590 #endif
591 {
592      int maxdim = 3,
593          valid  = FALSE;
594
595
596      interpret_args(argc, argv, NUMBER_OF_ARGS, optv);   /* see clargs.h */
597
598 #ifdef TRANSPUTER
599
600      *dim = DIMENSION;
```

329

```
601
602 #else   /* iPSC/2 */
603
604     if (optv[DIM]->found) *dim = (int) optv[DIM]->lsa[0];
605
606     switch (*dim) {
607
608         case 0:  case 1:  case 2:  case 3:  break;
609
610         default: while (!valid) {
611
612             printf("Enter desired cube dimension (0...%d): ", maxdim);
613             scanf("%d", dim);
614             fflush(stdin);
615
616             switch(*dim) {
617                 case 0:   case 1:   case 2:   case 3:
618                     valid = TRUE;
619                     break;
620             }
621         }
622     } /* end switch() */
623
624 #endif /* TRANSPUTER */
625
626     (optv[TIMING]->found)  ? (*timing  = TRUE) : (*timing  = FALSE);
627
628     (optv[VERBOSE]->found) ? (*verbose = TRUE) : (*verbose = FALSE);
629
630     printf("Argument resolution complete...\n\n");
631     printf("        Cube Dimension:   %d\n", *dim);
632
633     if (*timing)  printf("                    Timing:  ON\n");
634
635     (*verbose) ? (printf("        Verbose Mode:  ON\n\n")) :
636                  (printf("\n"));
637
638 }
639 /* End resolve_args() ------------------------------------------------- */
640
641
642
643
644
645 /* ---------------============ FUNCTION DEFINITION ==========-----------
646  *
647  */
648
649 #ifdef PROTOTYPE
650
```

330

```
651      void show_resulting_matrices(Double_Matrix_Type *A,
652                                   Double_Matrix_Type *AO, int *q)
653
654 #else
655
656      void show_resulting_matrices(A, AO, q)
657
658           Double_Matrix_Type *A,
659                              *AO;
660           int                *q;
661
662 #endif
663 {
664      Double_Matrix_Type *D,
665                         *L,
666                         *LU,
667                         *P,
668                         *QT,
669                         *QTA,
670                         *QTAP,
671                         *U;
672
673      int i,
674          j,
675          m = A->rows,
676          n = A->cols;
677
678
679      printf("Gauss Factorization Complete...\n\n");
680
681      strcpy(A->name, "A (after GF operations)");
682
683
684      /*  Allocate and form Q' and P ------------------------------------- */
685
686      if (!(QT = matalloc(m,m))) {
687
688          printf("Allocation failure for QT.\n");
689          exit(EXIT_FAILURE);
690      }
691
692      strcpy(QT->name, "Q Transpose");
693
694      for (i = 0; i < m; i++) {  QT->matrix[i][q[i]] = 1.0;  }
695
696
697      if (!(P  = identity(n,n))) {
698
699          printf("Allocation failure for P.\n");
700          exit(EXIT_FAILURE);
```

331

```
701      }
702
703      strcpy(P->name, "P   [ Partial (column) Pivoting ==> P == Identity ]");
704
705
706      /* Here, we slowly form Q'AP, keeping in mind that the A we are
707       * talking about is the original A....and we have labeled that one
708       * A0.  Therefore, we first form QTA (Q'A) as Q' * A0.  After we
709       * have QTA, we can multiply it (on the right) by P to get Q'AP,
710       * or QTAP as it is called here.
711       */
712
713      if (!(QTA = matalloc(m,n))) {
714
715          printf("Allocation failure for QTA.\n");
716          exit(EXIT_FAILURE);
717      }
718
719      strcpy(QTA->name, "Q' * (original) A");
720
721      if (matrix_product(QT, A0, QTA) == FAILURE) {
722
723          printf("matrix_product(QTA) Failure.\n");
724          exit(EXIT_FAILURE);
725      }
726
727
728      if (!(QTAP = matalloc(m,n))) {
729
730          printf("Allocation failure for QTAP.\n");
731          exit(EXIT_FAILURE);
732      }
733
734      strcpy(QTAP->name, "Q' * A * P");
735
736      if (matrix_product(QTA, P, QTAP) == FAILURE) {
737
738          printf("matrix_product(QTAP) Failure.\n");
739          exit(EXIT_FAILURE);
740      }
741
742
743      /* Next, we form L and U so that we can compare Q'AP ?=? LU.          */
744
745      L = zeros(m, n);        L->name = "L ";
746      U = zeros(m, n);        U->name = "U ";
747
748      for (i = 0; i < A->rows; i++) {
749
750          for (j = 0; j < A->cols; j++) {
```

332

```
751
752                if (i < j) { U->matrix[i][j] = A->matrix[i][j]; }
753
754                if (i == j) {
755
756                    L->matrix[i][j] = 1.0;
757                    U->matrix[i][j] = A->matrix[i][j];
758                }
759
760                if (i > j) { L->matrix[i][j] = A->matrix[i][j]; }
761            }
762        }
763
764        if (!(LU = matalloc(m,n))) {
765
766            printf("Allocation failure for LU.\n");
767            exit(EXIT_FAILURE);
768        }
769
770        strcpy(LU->name, "L * U");
771
772        if (matrix_product(L, U, LU) == FAILURE) {
773
774            printf("matrix_product(LU) Failure.\n");
775            exit(EXIT_FAILURE);
776        }
777
778
779        /*  Finally, we create a matrix of differences between the elements
780         *  found in QTAP (Q'AP) and LU.  If everything proceeded according
781         *  to the plan, this will be a matrix of zeros.
782         */
783
784        if (!(D = matalloc(m,n))) {
785
786            printf("Allocation failure for D.\n");
787            exit(EXIT_FAILURE);
788        }
789
790        strcpy(D->name, "Q'AP - LU");
791
792        for (i = 0; i < m; i++) {
793
794            for (j = 0; j < n; j++) {
795
796                D->matrix[i][j] = (QTAP->matrix[i][j] - LU->matrix[i][j]);
797            }
798        }
799
800        printmd(*A, WIDTH, AFT);
```

333

```
801      printf("\n\n");
802      printmd(*L, WIDTH, AFT);
803      printf("\n\n");
804      printmd(*U, WIDTH, AFT);
805      printf("\n\n");
806
807      printmd(*QT, WIDTH, AFT);
808      printf("\n\n");
809      printmd(*P, WIDTH, AFT);
810      printf("\n\n");
811      printmd(*QTA, WIDTH, AFT);
812      printf("\n\n");
813      printmd(*QTAP, WIDTH, AFT);
814      printf("\n\n");
815      printmd(*LU, WIDTH, AFT);
816      printf("\n\n");
817      printmd(*D, WIDTH, AFT);
818      printf("\n\n");
819
820 }
821 /* End show_resulting_matrices() --------------------------------------- */
822
823
824
825
826
827 /* --------------============ FUNCTION DEFINITION  ============---------
828  *
829  *   This is a simple function to physically swap the elements from row s to
830  *   the current pivot row, r.  It does not concern itself with column r or
831  *   any column j > r.
832  */
833
834 #ifdef PROTOTYPE
835
836      void swap_rows_left_of_pivot(Double_Matrix_Type *A, int r, int s)
837
838 #else
839
840      void swap_rows_left_of_pivot(A, r, s)
841
842          Double_Matrix_Type *A;
843          int                 r,
844                              s;
845
846 #endif
847 {
848      double tmp;
849
850      int j;
```

334

```
851
852
853      for (j = 0; j < r; j++) {
854
855          tmp = A->matrix[r][j];
856          A->matrix[r][j] = A->matrix[s][j];
857          A->matrix[s][j] = tmp;
858      }
859
860  }
861  /* End swap_rows_left_of_pivot() --------------------------------------- */
862
863
864
865
866
867  /* ----------------------========  FUNCTION  DEFINITION  ========----------------------
868   *
869   * This function performs updates to a permutation vector, v[], of length
870   * 'size'.  The pivot_index indicates the row or column where the next
871   * pivot has been located; and k indicates the stage, or the row and
872   * column where the pivot is to be installed.
873   */
874
875  #ifdef PROTOTYPE
876
877      void update_permutation(int v[], int size, int k, int pivot_index)
878
879  #else
880
881      void update_permutation(v, size, k, pivot_index)
882
883          int v[],
884              size,
885              k,
886              pivot_index;
887
888  #endif
889  {
890      int i;
891
892
893      i = v[k];      v[k] = v[pivot_index];      v[pivot_index] = i;
894  }
895  /* End update_permutation() --------------------------------------------- */
896
897
898
899
900
```

335

```
901 #ifdef PROTOTYPE    /* ===================================================== */
902
903     main(int argc, char *argv[])
904
905 #else
906
907     main(argc, argv)
908
909         int   argc;
910         char  *argv[];
911
912 #endif
913 {
914
915 /* ----------------========== VARIABLE DEFINITIONS ==================--------- */
916
917     double a,                    /* denominator of growth factor (g/a)  */
918            *cbuf,                /* col buffer holds one col at a time   */
919            **dtime,              /* doubles corresponding to ticks **t   */
920            eps       = epsd(),   /* machine precision (see machine.h)    */
921            g         = 0.0,      /* the growth factor                    */
922            root_time,            /* time measured at root for iterations */
923            tol;                  /* tolerance                            */
924
925     Double_Matrix_Type *A,       /* This A gets operated upon/changed    */
926                        *A0;      /* The original copy of A               */
927
928     int  cubesize,               /* number of processors in the cube     */
929          dim,                    /* dimension of the hypercube           */
930          i,
931          j,
932          m,                      /* number of rows in A                  */
933          me,                     /* root processor's id                  */
934          n,                      /* number of cols in A                  */
935          *q,                     /* row permutation vector               */
936          r,                      /* numerical rank estimate              */
937          timing,                 /* Boolean                              */
938          verbose;                /* Boolean                              */
939
940     long sizeof_col,             /* sizes, in bytes                      */
941          sizeof_int,
942          sizeof_pivot;
943
944     ticks      root_start,
945                t_root,           /* time measured at root transputer     */
946                **t;              /* time data: row => node, col => event */
947
948     Pivot_Type pivot;            /* pivot                                */
949
950
```

336

```
951 /* ------------=========== INITIALIZATIONS ============---------- */
952
953 #ifdef TRANSPUTER
954
955     /*  Add 1M to the heap to allow for generation of large matrices     */
956     addfree((void *) _heapend, 0x100000);
957
958 #endif
959
960     printf("\n%s\n\n", version);
961
962     define_valid_args();
963
964     resolve_args(argc, argv, &dim, &timing, &verbose);
965
966     A = generate(&m, &n);
967
968     sizeof_col   = (long) (A->rows * sizeof(double));
969     sizeof_int   = (long) sizeof(int);
970     sizeof_pivot = (long) sizeof(Pivot_Type);
971
972     if (!(cbuf = (double *) malloc(sizeof_col))) {
973
974         printf("main():  Allocation failure for cbuf[].\n");
975         exit(EXIT_FAILURE);
976     }
977
978     cubesize = POW2(dim);
979
980 #ifdef TRANSPUTER
981
982     initialize_hypercube(dim);
983
984 #else
985
986     cubename = initialize_hypercube(dim, nodecode);
987
988 #endif
989
990
991     me = myhost();
992
993     if (verbose) {
994
995         if (!(A0 = matalloc(m,n))) {
996
997             printf("Allocation failure for A0.\n");
998             exit(EXIT_FAILURE);
999         }
1000
```

337

```
1001          strcpy(A0->name, "Original A");
1002
1003          for (i = 0; i < A->rows; i++) {
1004              for (j = 0; j < A->cols; j++) {
1005
1006                  A0->matrix[i][j] = A->matrix[i][j];
1007              }
1008          }
1009          printf("\n\nA has been allocated and generated.\n\n");
1010          printmd(*A, WIDTH, AFT);
1011          printf("\n\nSending size(A) to the nodes.\n\n");
1012      }
1013
1014
1015 #ifdef TRANSPUTER
1016
1017      cubecast(me, dim, (char *) &m,      sizeof_int, cubesize);
1018      cubecast(me, dim, (char *) &n,      sizeof_int, cubesize);
1019      cubecast(me, dim, (char *) &timing, sizeof_int, cubesize);
1020
1021 #else  /*  iPSC/2  */
1022
1023      cubecast(me, dim, (char *) &m,      sizeof_int, ROW_SIZE_TYPE);
1024      cubecast(me, dim, (char *) &n,      sizeof_int, COL_SIZE_TYPE);
1025      cubecast(me, dim, (char *) &timing, sizeof_int, ARG_TYPE);
1026
1027 #endif
1028
1029      if (verbose) printf("\nSent size(A) to nodes.\n");
1030
1031      distribute_columns(A, dim, cbuf);
1032
1033      q = initial_permutation_vector(m);
1034
1035
1036      /* FINAL PREPARATIONS BEFORE STARTING THE ITERATION ------------------
1037       *
1038       *  Get the first pivot from node 0.  Initialize the growth factor
1039       *  variables, g and a, so that we can compute growth factor (g/a) as
1040       *  we go.  Set a reasonable tolerance.
1041       *
1042       * -------------------------------------------------------------------
1043       */
1044
1045 #ifdef TRANSPUTER
1046
1047      receive(0, (char *) &pivot, sizeof_pivot, cubesize);
1048
1049 #else  /* iPSC/2 */
1050
```

338

```
1051        receive(0, (char *) &pivot, sizeof_pivot, PIVOT_TYPE);
1052
1053 #endif /* TRANSPUTER */
1054
1055
1056      a = g = MAX(g, fabs(pivot.u));
1057
1058      tol = (MIN(m,n)) * g * eps;
1059
1060
1061      /*  BEGINNING OF ITERATION -----------------------------------------
1062       *
1063       *  We enter with A established and knowledge of the first pivot.
1064       *
1065       *  ----------------------------------------------------------------
1066       */
1067
1068 #ifdef TRANSPUTER
1069
1070      root_start = clock();
1071
1072 #endif
1073
1074      printf("Beginning iterations.\n\n");
1075
1076      for (r = 0; r < (MIN(m,n)); r++) {
1077
1078          if (pivot.id == RANK_DEFICIENT) break;
1079
1080          /*  We expect to receive cbuf[] in the correct (i.e., already
1081           *  swapped) order.  Before we stuff cbuf[] into A[][], we'll swap
1082           *  rows left of the pivot column, and then insert the new pivot
1083           *  column.
1084           */
1085
1086 #ifdef TRANSPUTER
1087
1088          receive(0, (char *) cbuf, sizeof_col, cubesize);
1089
1090 #else  /* iPSC/2 */
1091
1092          receive(0, (char *) cbuf, sizeof_col, PCOL_TYPE);
1093
1094 #endif /* TRANSPUTER */
1095
1096          g = MAX(g, fabs(pivot.u));
1097
1098          update_permutation(q, m, r, pivot.s);
1099
1100          if (pivot.s != r)  swap_rows_left_of_pivot(A, r, pivot.s);
```

339

```
1101
1102            for (i = 0; i < A->rows; i++) { A->matrix[i][r] = cbuf[i]; }
1103
1104            if (verbose) {
1105
1106                printf("Host:  Stage %d, Pivot value = %e.  ", r, pivot.u);
1107                printf("Growth factor = %e.\n", (g/a));
1108                printf("q = "); printvi(q, A->rows, WIDTH);
1109                printf("\n");
1110
1111            }
1112
1113            if (r < ((MIN(m,n)) - 1)) {
1114
1115 #ifdef TRANSPUTER
1116
1117                receive(0, (char *) &pivot, sizeof_pivot, cubesize);
1118
1119 #else  /* iPSC/2 */
1120
1121                receive(0, (char *) &pivot, sizeof_pivot, PIVOT_TYPE);
1122
1123 #endif /* TRANSPUTER */
1124
1125            }
1126
1127        } /* end for(r) ----------------------------------------------- */
1128
1129 #ifdef TRANSPUTER
1130
1131        t_root = (clock() - root_start);
1132
1133        if (timing) {
1134
1135            root_time = ((double) t_root) * LO_PERIOD;
1136
1137            printf("\n\nRoot transputer:  ");
1138            printf("Time for iterations: %8.4lf seconds\n\n", root_time);
1139        }
1140
1141 #endif
1142
1143
1144        free(cbuf);
1145
1146
1147        /*  I have selected the easy way out and assumed A has full rank.  If
1148         *  you did not make this assumption, you would need to collect the
1149         *  remaining columns at this point.
1150         */
```

340

```
1151
1152     if (timing) dtime = receive_timing_data(cubesize);
1153
1154
1155     /*  There is no more use for the nodes, so they can be released.  */
1156
1157 #ifndef TRANSPUTER
1158     printf("\n\nmain():  Killing and releasing cube.\n\n");
1159     killcube(ALL_NODES, ALL_PIDS);
1160     relcube(cubename);
1161 #endif
1162
1163     if (verbose) {  /*  Create and show Q', A0, P, L, U .... ----------- */
1164
1165         show_resulting_matrices(A, A0, q);
1166
1167     }
1168
1169
1170     if (timing) display_timing_data(A, dim, a, eps, g, tol, r, dtime);
1171
1172 }
1173 /* -------------------------- EOF  gfpphost.c  ======================= */
```

341

```
 1 /* --------------========== PROGRAM INFORMATION ========================
 2  *
 3  * SOURCE  :  gfppnode.c
 4  * VERSION :  2.0
 5  * DATE    :  21 September 1991
 6  * AUTHOR  :  Jonathan E. Hartman, U. S. Naval Postgraduate School
 7  * REMARKS :  See gf.h.
 8  *
 9  * -------------=====================================================
10  */
11
12 #include <math.h>
13
14 #ifdef TRANSPUTER
15
16 #include <conc.h>
17
18 #include <matrix.h>
19 #include <macros.h>
20 #include <allocate.h>
21 #include <comm.h>
22 #include <generate.h>
23 #include <mathx.h>
24 #include <ops.h>
25 #include <timing.h>
26
27 #else
28
29 #include "/usr/hartman/matlib/matrix.h"
30 #include "/usr/hartman/matlib/macros.h"
31 #include "/usr/hartman/matlib/allocate.h"
32 #include "/usr/hartman/matlib/comm.h"
33 #include "/usr/hartman/matlib/generate.h"
34 #include "/usr/hartman/matlib/mathx.h"
35 #include "/usr/hartman/matlib/ops.h"
36 #include "/usr/hartman/matlib/timing.h"
37 #endif
38
39 #include "gf.h"
40
41 #ifdef TRANSPUTER
42
43 Channel *ic[(CUBESIZE + 1)],
44         *oc[(CUBESIZE + 1)];
45
46 #endif
47
48
49 ticks t[MAX_EVENTS];
50
```

```
51
52
53
54
55  /* ------------------========== FUNCTION DEFINITION ===================----
56   *
57   *  This function is kind of an inverse for local_column().  Given some
58   *  column number (local_column) held at this node, the function returns
59   *  the corresponding column number in the global/host copy of the full-
60   *  sized A.  This could be implemented more efficiently as a macro.
61   */
62
63  #ifdef PROTOTYPE
64
65      int global_column(int local_column, int me, int cubesize)
66
67  #else
68
69      int global_column(local_column, me, cubesize)
70
71          int local_column,
72              me,
73              cubesize;
74
75  #endif
76  {
77      return(local_column * cubesize + me);
78  }
79  /* End global_column() ------------------------------------------------- */
80
81
82
83
84
85  /* ------------------========== FUNCTION DEFINITION ===================----
86   *
87   *  This function maps a column number in the global A (the full-sized A
88   *  held at the root processor/host) to the corresponding local column num-
89   *  ber.  If the global_column is not one that is held at this node, a
90   *  negative value (-1) is returned.
91   */
92
93  #ifdef PROTOTYPE
94
95      int local_column(int global_column, int me, int cubesize)
96
97  #else
98
99      int local_column(global_column, me, cubesize)
100
```

343

```
101        int global_column,
102              me,
103              cubesize;
104 #endif
105 {
106     if ((global_column % cubesize) != me)   return(-1);
107
108     return((int) global_column / cubesize);
109 }
110 /* End local_column() ---------------------------------------------- */
111
112
113
114
115
116 /* --------------=========== FUNCTION DEFINITION   =====================------
117  *
118  */
119
120 #ifdef PROTOTYPE
121
122     void do_pivot_column_arithmetic(Double_Matrix_Type *A, double *cbuf,
123                                     int k, int me, int cubesize)
124
125 #else
126
127     void do_pivot_column_arithmetic(A, cbuf, k, me, cubesize)
128
129        Double_Matrix_Type *A;
130        double              *cbuf;
131        int                 k,
132                            me,
133                            cubesize;
134
135 #endif
136 {
137     double  pivot_value;
138
139     int     i,
140             pivot_column;
141
142
143     pivot_column = local_column(k, me, cubesize);
144
145     pivot_value = A->matrix[k][pivot_column];
146
147
148     /*  Divide everything under the pivot by the pivot value          */
149     for (i = (k+1); i < A->rows; i++) {
150
```

```
151        A->matrix[i][pivot_column] /= pivot_value;
152     }
153
154
155     /*  This is somewhat redundant, and not optimal with respect to
156      *  efficiency, but it works and reads clearly, right?
157      */
158
159     for (i = 0; i < A->rows; i++)  cbuf[i] = A->matrix[i][pivot_column];
160
161 }
162 /* End do_pivot_column_arithmetic() ------------------------------------ */
163
164
165
166
167
168 /* ------------------========== FUNCTION DEFINITION  =====================
169  *
170  *  This function accepts the matrix, the global column number for this
171  *  stage (where the pivot will be taken from), and a pivot structure to be
172  *  filled....among other things....and 'returns' the row, s, and value, u,
173  *  of the new pivot in global column r (local column lc).
174  */
175
176 #ifdef PROTOTYPE
177
178     void locate_pivot(int me, int cubesize, Double_Matrix_Type *A, int r,
179                       Pivot_Type *pivot)
180
181 #else
182
183     void locate_pivot(me, cubesize, A, r, pivot)
184
185         int             me,
186                         cubesize;
187         Double_Matrix_Type *A;
188         int             r;
189         Pivot_Type      *pivot;
190
191 #endif
192 {
193     int i,
194         pivot_column;
195
196
197     pivot_column = local_column(r, me, cubesize);
198
199     /*  Initialize pivot row and value                                  */
200     pivot->s = r;
```

345

```
201      pivot->u = A->matrix[r][pivot_column];
202
203
204      for (i = (r+1); i < A->rows; i++) {
205
206          if (fabs(A->matrix[i][pivot_column]) > fabs(pivot->u)) {
207
208              pivot->s = i;
209              pivot->u = A->matrix[i][pivot_column];
210          }
211      }
212 }
213 /* End locate_pivot() ----------------------------------------------- */
214
215
216
217
218
219 /* ---------------=========== FUNCTION DEFINITION ====================-----
220  *
221  * Receive this node's columns from the root/host processor (manager),
222  * place them into the column buffer, then transfer them into A while
223  * the other processors are communicating with the root.
224  *
225  * The transputer scheme is a bit more involved.  Here nodes 0000 and 1000
226  * are connected to the root and they must receive for everyone.  They (0
227  * and 8) are not directly connected to everyone, so the columns must be
228  * passed out in cycles.  For instance, suppose we used the hybrid 4-cube.
229  * Then nodes 0 and 8 would receive bursts of 8 columns at a time.  They
230  * would keep the first one (we'll call it column 0 in some sort of rela-
231  * tive numbering scheme that abides by the C numbering convention), send
232  * the next one (col 1) in the 0x1 direction, the next to the 0x2 direc-
233  * tion, column 3 in the 0x1 direction, column 4 in the 0x4 direction,
234  * column 5 in the 0x1 direction, column 6 in the 0x2 direction, and
235  * lastly, column 7 in the 0x1 direction.  This makes cycle == 8 for nodes
236  * 0000 and 1000.  Similarly, nodes x001 have a cycle of four where they
237  * keep the first column to arrive and then send the next three to direc-
238  * tions 0x2, 0x4, and 0x2 in turn.  This distribution pattern is main-
239  * tained until all of the columns have been distributed.
240  */
241
242 #ifdef PROTOTYPE
243
244      void receive_columns(int              dim,
245                           int              node,
246                           Double_Matrix_Type *A,
247                           int              n,
248                           double           *cbuf,
249                           int              my_cols,
250                           int              colsize)
```

346

```
251
252 #else
253
254     void receive_columns(dim, node, A, n, cbuf, my_cols, colsize)
255
256         int                 dim,
257                             node;
258         Double_Matrix_Type  *A;
259         int                 n;
260         double              *cbuf;
261         int                 my_cols,
262                             colsize;
263
264 #endif
265 {
266     int cubesize = pow2(dim),
267         cycle,                      /*  length of typical col burst  */
268         dimeff    = MIN(3, dim),    /*  effective dimension          */
269         from,                       /*  node that I receive from     */
270         gc,                         /*  global column index          */
271         i,
272         idx,                        /*  index into to[]              */
273         lc        = 0,              /*  local column index           */
274         ldeff,                      /*  effective least_dimension()  */
275         nodeff    = (node % 8),     /*  effective node number        */
276         others,                     /*  no. of nodes in other 3-cube */
277         step,                       /*  for destination of cols rec'd*/
278         thehost   = myhost(),
279         to[8];                      /*  ==> direction to send to     */
280
281
282 #ifdef TRANSPUTER
283
284     ldeff = least_dimension(nodeff);
285
286     if (nodeff == 0) from = myhost();
287     else             from = node ^ pow2(ldeff - 1);
288
289     /*  cycle describes the length of a cycle that starts with me (node)...
290      *  then I receive several columns for others....then start over with
291      *  me.  The nodes in the highest dimension have cycle == 1 ==> self
292      *  only.  We also fill to[] with the directions that we will be
293      *  sending to within a given cycle.  Not all nodes use all 8 elements
294      *  of to[].  They only use the first cycle elements.  The step is the
295      *  difference between the column numbers received at this node during
296      *  a given burst of length cycle.
297      *
298      *  When we use the hybrid 4-cube, we are treating it as two 3-cubes,
299      *  so the variable others is set to 8.  This is because there are 8
300      *  other columns between every burst that comes to the 3-cube that
```

```
301        *   node is in.
302        */
303       cycle = pow2(dimeff - ldeff);
304
305       (dim == 4) ? (others = 8) : (others = 0);
306
307       step  = pow2(ldeff);
308
309       to[0] = 0;
310       to[1] = to[3] = to[5] = to[7] = pow2(ldeff);
311       to[2] = to[6] =                 pow2(ldeff + 1);
312       to[4] =                         pow2(ldeff + 2);
313
314
315       for (gc = node; gc < n; gc += (others + step)) {
316
317           receive(from, (char *) cbuf, colsize, cubesize);
318
319           for (i = 0; i < A->rows; i++) A->matrix[i][lc] = cbuf[i];
320
321           lc++;
322
323           for (idx = 1; idx < cycle; idx++) {
324
325               gc += step;
326
327               if (gc < n) {
328
329                   receive(from, (char *) cbuf, colsize, cubesize);
330
331                   directional_send(node, dim, to[idx], (char*) cbuf, colsize);
332               }
333           }
334
335       } /* end for(gc) */
336
337
338  #else  /* iPSC/2 */
339
340       for (lc = 0; lc < my_cols; lc++) {
341
342           receive(thehost, (char *) cbuf, colsize, COL_TYPE);
343
344           for (i = 0; i < A->rows; i++) {  A->matrix[i][lc] = cbuf[i];  }
345       }
346
347  #endif /* TRANSPUTER */
348
349  }
350  /* End receive_columns() ------------------------------------------ */
```

348

```
351
352
353
354
355
356  /* -----------========== FUNCTION DEFINITION ======================
357   *
358   * This function sends in the timing data that is held in t[].
359   */
360
361  #ifdef PROTOTYPE
362
363      void submit_timing_data(int node, int dim)
364
365  #else
366
367      void submit_timing_data(node, dim)
368
369          int node,
370              dim;
371
372  #endif
373  {
374      int  dimeff   = MIN(dim, 3),
375           dir,
376           i,
377           ld       = least_dimension(node % 8),
378           nodeff   = (node % 8),
379           root     = myhost();
380
381      long cubesize = pow2(dim),
382           tlen;
383
384
385      tlen = (long) (MAX_EVENTS * sizeof(ticks));
386
387  #ifdef TRANSPUTER
388
389      submit(node, dim, (char *) t, tlen, cubesize);
390
391      if (dimeff == ld) return;
392
393      if ((nodeff == 2) || (nodeff == 3)) {
394
395          if (dimeff > 2) {
396              directional_receive(node, dim, 0x4, (char *) t, tlen);
397              submit(node, dim, (char *) t, tlen, cubesize);
398          }
399          return;
400      }
```

349

```
401
402      if (nodeff == 1) {
403
404          if (dimeff > 1) {
405
406              directional_receive(node, dim, 0x2, (char *) t, tlen);
407              submit(node, dim, (char *) t, tlen, cubesize);
408          }
409
410          if (dimeff > 2) {
411
412              directional_receive(node, dim, 0x4, (char *) t, tlen);
413              submit(node, dim, (char *) t, tlen, cubesize);
414              directional_receive(node, dim, 0x2, (char *) t, tlen);
415              submit(node, dim, (char *) t, tlen, cubesize);
416          }
417
418          return;
419      }
420
421      if (nodeff == 0) {
422
423          if (dimeff > 0) {
424
425              /*  retrans from 1 or 9 ----------------------------------- */
426              directional_receive(node, dim, 0x1, (char *) t, tlen);
427              submit(node, dim, (char *) t, tlen, cubesize);
428          }
429
430          if (dimeff > 1) {
431
432              /*  retrans from 2 or 10 ---------------------------------- */
433              directional_receive(node, dim, 0x2, (char *) t, tlen);
434              submit(node, dim, (char *) t, tlen, cubesize);
435              /*  retrans from 3 or 11 ---------------------------------- */
436              directional_receive(node, dim, 0x1, (char *) t, tlen);
437              submit(node, dim, (char *) t, tlen, cubesize);
438          }
439
440          if (dimeff > 2) {
441
442              /*  retrans from 4 or 12 ---------------------------------- */
443              directional_receive(node, dim, 0x4, (char *) t, tlen);
444              submit(node, dim, (char *) t, tlen, cubesize);
445              /*  retrans from 5 or 13 ---------------------------------- */
446              directional_receive(node, dim, 0x1, (char *) t, tlen);
447              submit(node, dim, (char *) t, tlen, cubesize);
448              /*  retrans from 6 or 14 ---------------------------------- */
449              directional_receive(node, dim, 0x2, (char *) t, tlen);
450              submit(node, dim, (char *) t, tlen, cubesize);
```

350

```
451             /*  retrans from 7 or 15  ----------------------------------- */
452             directional_receive(node, dim, 0x1, (char *) t, tlen);
453             submit(node, dim, (char *) t, tlen, cubesize);
454         }
455     }
456
457
458 #else  /* iPSC/2 */
459
460     delay(1.0 + 2.0 * (float) node);
461
462     send(root, (char *) t, tlen, (node + NODE_OFFSET));
463
464 #endif /* TRANSPUTER */
465
466 }
467 /* End submit_timing_data() ----------------------------------------- */
468
469
470
471
472
473 /* -----------------============  FUNCTION  DEFINITION  =====================
474  *
475  *  This function performs the required operations on the Gauss Transform
476  *  area, G, of A and searches for the next pivot.
477  */
478
479 #ifdef PROTOTYPE
480
481     void update_G(Double_Matrix_Type *A, double *cbuf,
482                 int cubesize, int k, int me, int n, Pivot_Type *pivot)
483
484 #else
485
486     void update_G(A, cbuf, cubesize, k, me, n, pivot)
487
488         Double_Matrix_Type *A;
489         double          *cbuf;
490         int             cubesize,
491                         k,
492                         me,
493                         n;
494         Pivot_Type    *pivot;
495
496 #endif
497 {
498     int i,
499         j,
500         gc = 0,                         /*  global column number        */
```

351

```
501        lc = 0;                            /*  local column number to start */
502
503    ticks   start;
504
505
506    while ((gc = global_column(lc, me, cubesize)) <= k)  lc++;
507
508
509    /*  The pivot row is k and we know that lc is the first local column to
510     *  the right of k.  Now we must move through the Gauss Transform area,
511     *  all A(i,j) where i > k and j > k, and perform the operation:
512     *
513     *     A(i,j) = A(i,j) - A(i,k) * A(k,j) <==> A(i,j) -= cbuf[i]*A(k,j)
514     */
515
516    start = clock();
517
518    for (i = k+1; i < A->rows; i++) {
519
520        for (j = lc; j < A->cols; j++) {
521
522            A->matrix[i][j] -= (cbuf[i] * A->matrix[k][j]);
523
524        } /* end for(j) */
525
526    } /* end for(i) */
527
528    t[LOOPTIME] += (clock() - start);
529
530 }
531 /* End update_G() --------------------------------------------------------- */
532
533
534
535 /* ================================================================== */
536
537 main(){
538
539    double *cbuf;                    /*  column buffer holds one col of A      */
540
541    Double_Matrix_Type *A;          /*  this node's portion of the matrix A   */
542
543    int cubesize,                   /*  number of processors in the cube      */
544        dim,                        /*  dimension of the hypercube            */
545        gc,                         /*  global column number                  */
546        i,                          /*  generic integer and row ctr           */
547        j,                          /*  generic integer and col ctr           */
548        k,                          /*  index to pivot                        */
549        m,                          /*  number of rows in A (same local/all)  */
550        me,                         /*  id of this processor                  */
```

352

```
551         my_cols = 0,            /*  number of cols in local portion of A */
552         n,                      /*  number of cols in all of A          */
553         root,                   /*  host/root processor id              */
554         timing;                 /*  Boolean                             */
555
556    long sizeof_col,             /*  sizes, in bytes                     */
557         sizeof_int,
558         sizeof_pivot;
559
560    ticks      start,
561               starti;           /*  another start                       */
562
563    Pivot_Type pivot;
564
565
566
567    /* -------------======== INITIALIZATION WORK   ========------------ */
568
569    for (i = 0; i < MAX_EVENTS; i++) t[i] = 0;
570
571    start = t[START_TIME] = clock();
572
573
574 #ifdef TRANSPUTER
575
576    cubesize = CUBESIZE;
577    dim      = DIMENSION;
578    initialize_hypercube(dim);
579
580 #else
581
582    cubesize = (int) numnodes();
583    dim      = (int) nodedim();
584
585 #endif
586
587    t[DATA_SOURCE] = me = (int) mynode();
588    root                = (int) myhost();
589
590    sizeof_int   = (long) sizeof(int);
591    sizeof_pivot = (long) sizeof(Pivot_Type);
592
593
594    /*  BROADCAST THE SIZE(A) ------------------------------------------
595     *
596     *  All node processors need to know the number of rows and columns in
597     *  the matrix A [i.e., size(A)].  A broadcast to the entire cube,
598     *  cubecast(), is used to achieve this.  The nodes also need to know
599     *  whether or not to set timing on, so this value is passed too.
600     *
```

353

```
601        */
602
603 #ifdef TRANSPUTER
604
605        cubecast(me, dim, (char *) &m,      sizeof_int, cubesize);
606        cubecast(me, dim, (char *) &n,      sizeof_int, cubesize);
607        cubecast(me, dim, (char *) &timing, sizeof_int, cubesize);
608
609 #else  /* iPSC/2 */
610
611        cubecast(me, dim, (char *) &m,      sizeof_int, ROW_SIZE_TYPE);
612        cubecast(me, dim, (char *) &n,      sizeof_int, COL_SIZE_TYPE);
613        cubecast(me, dim, (char *) &timing, sizeof_int, ARG_TYPE);
614
615 #endif /* TRANSPUTER */
616
617        sizeof_col = (long) (m * sizeof(double));
618
619
620        /*  COLUMN BUFFER AND COUNTER -------------------------------------------
621         *
622         *  The column buffer, cbuf[], will be used to hold one column of A at
623         *  a time.  We will see cbuf[] used on a variety of occasions when we
624         *  must work with a column of A.  Allocate cbuf[] and determine the
625         *  number of columns that will be stored locally (my_cols).
626         *
627         */
628        cbuf = (double *) malloc(sizeof_col);
629
630        for (i = 0; i < n; i++) { if ((i % cubesize) == me) my_cols++; }
631
632
633        /*  ESTABLISH LOCAL A ---------------------------------------------------
634         *
635         *  Allocate storage space for this node's part of A (it is called A
636         *  even though it is only part of A).
637         */
638
639        A = matalloc(m, my_cols);
640
641        t[SETUP] = clock() - start;
642
643        start = clock();
644
645        receive_columns(dim, me, A, n, cbuf, my_cols, sizeof_col);
646
647        t[DISTRIB_COLS] = clock() - start;
648
649
650        /*  BEGIN ITERATION -----------------------------------------------------
```

354

```
651        *
652        *   1.) At the top of the for() loop we have just completed update_G(),
653        *       so the local candidate for the next pivot is situated in np[0].
654        *   The function elect_next_pivot() performs a series of directional_
655        *   exchange()s so that all local candidates compete in an election
656        *   process.  The winner is np[0].
657        *
658        *   2.) If all went well, np[0] contains the next pivot.  This informa-
659        *
660        *   3.) If this node has the pivot column [if (p[k] == gc)], it must
661        *       divide everything under the pivot by the value of the pivot and
662        *   distribute the column to all other nodes (node zero sends to host).
663        *
664        *   4.) Finally, this node must perform the computations across the
665        *       Gauss Transform area for the local portion of A.  The
666        *   update_G() function also locates the next pivot without special
667        *   expense.  Then it is time to go back to the top of the loop.
668        */
669
670      start = clock();
671
672      for (k = 0; k < (MIN(m,n)); k++) {
673
674          pivot.id = k % cubesize;
675          pivot.t  = k;
676
677          /*  know id; k ==> t; need s, u  */
678
679          if (pivot.id == me)  locate_pivot(me, cubesize, A, k, &pivot);
680
681          cubecast_from(pivot.id, me, dim, (char *) &pivot, sizeof_pivot);
682
683          if (me == 0) {
684
685              starti = clock();
686
687  #ifdef TRANSPUTER
688
689              send(root, (char *) &pivot, sizeof_pivot, cubesize);
690
691  #else   /* iPSC/2 */
692
693              send(root, (char *) &pivot, sizeof_pivot, PIVOT_TYPE);
694
695  #endif /* TRANSPUTER */
696
697              t[PIVOTS_TO_HOST] += (clock() - starti);
698          }
699
700          swap_rows(A, k, pivot.s);
```

355

```
701
702          starti = clock();
703
704          if (pivot.id == me) {
705
706               do_pivot_column_arithmetic(A, cbuf, k, me, cubesize);
707          }
708
709          t[PCOL_ARITHMETIC] += (clock() - starti);
710
711          starti = clock();
712
713          cubecast_from(pivot.id, me, dim, (char *) cbuf, sizeof_col);
714
715          t[PCOL_DISTRIB] += (clock() - starti);
716
717
718          if (me == 0) {
719
720               starti = clock();
721
722 #ifdef TRANSPUTER
723
724               submit(me, dim, (char *) cbuf, sizeof_col, cubesize);
725
726 #else  /* iPSC/2 */
727
728               submit(me, dim, (char *) cbuf, sizeof_col, PCOL_TYPE);
729
730 #endif /* TRANSPUTER */
731
732               t[PCOLS_TO_HOST] += (clock() - starti);
733          }
734
735          starti = clock();
736          update_G(A, cbuf, cubesize, k, me, n, &pivot);
737          t[UPDATING_G] += (clock() - starti);
738
739     }
740     /* END ITERATION [for(k...)] ---------------------------------------- */
741
742     t[ITERATION] = clock() - start;
743
744
745     free(cbuf);
746
747     t[STOP] = clock();
748
749     if (timing) submit_timing_data(me, dim);
750
```

356

```
751     return(SUCCESS);
752 }
753 /* ------------------========== EOF gfppnode.c ==========------------ */
```

```
1  /* --------------========== PROGRAM INFORMATION ================-----------
2   *
3   * SOURCE   :   gfpcnode.c
4   * VERSION  :   2.3
5   * DATE     :   17 September 1991
6   * AUTHOR   :   Jonathan E. Hartman, U. S. Naval Postgraduate School
7   * REMARKS  :   See gf.h.
8   *
9   * ------------==================================================-----------
10  */
11
12 #include <math.h>
13
14 #ifdef TRANSPUTER
15
16 #include <conc.h>
17
18 #include <matrix.h>
19 #include <macros.h>
20 #include <allocate.h>
21 #include <comm.h>
22 #include <generate.h>
23 #include <mathx.h>
24 #include <ops.h>
25 #include <timing.h>
26
27 #else
28
29 #include "/usr/hartman/matlib/matrix.h"
30 #include "/usr/hartman/matlib/macros.h"
31 #include "/usr/hartman/matlib/allocate.h"
32 #include "/usr/hartman/matlib/comm.h"
33 #include "/usr/hartman/matlib/generate.h"
34 #include "/usr/hartman/matlib/mathx.h"
35 #include "/usr/hartman/matlib/ops.h"
36 #include "/usr/hartman/matlib/timing.h"
37 #endif
38
39 #include "gf.h"
40
41 #ifdef TRANSPUTER
42
43 Channel *ic[(CUBESIZE + 1)],
44         *oc[(CUBESIZE + 1)];
45
46 #endif
47
48
49 ticks t[MAX_EVENTS];
50
```

358

```
51 /* ----------------============    FUNCTION  DEFINITION   ===========------------
52  *
53  *   After this node finds its candidate for next pivot, there must be a
54  *   comparison with all other nodes.  The local candidate starts in np[0].
55  *   Direction-by-direction, candidates are exchanged and the winner is
56  *   positioned in np[0].  If there is a tie, the candidate from the smaller
57  *   node number wins.  A RANK_DEFICIENT opponent is ignored (the local
58  *   candidate must be at least as good).  In the end, all processors have
59  *   identical entries in np[0].
60  */
61
62 #ifdef PROTOTYPE
63
64     void elect_next_pivot(int me, int dim, Pivot_Type *np)
65
66 #else
67
68     void elect_next_pivot(me, dim, np)
69
70         int         me,
71                     dim;
72         Pivot_Type *np;
73
74 #endif
75 {
76     int dir;
77
78     long cubesize = pow2(dim),
79          len      = sizeof(Pivot_Type);
80
81
82     for (dir = 1; dir < (int) cubesize; dir <<= 1) {
83
84         if (dir != 8) {
85
86             directional_exchange(me, dim, dir, (char *) &(np[1]),
87                                                (char *) &(np[0]), len);
88         }
89         else {
90
91             if ((me % 8) != 0) {        /*  we don't want 0 <--> 8 comm  */
92
93                 directional_exchange(me, dim, dir, (char *) &(np[1]),
94                                                    (char *) &(np[0]), len);
95             }
96         }
97
98
99
100
```

359

```
101          if (np[1].id != RANK_DEFICIENT) {
102
103              if (fabs(np[1].u) > fabs(np[0].u)) {
104
105                  np[0].id = np[1].id;    np[0].u  = np[1].u;
106                  np[0].s  = np[1].s;     np[0].t  = np[1].t;
107              }
108              else {
109
110                  if (fabs(np[1].u) == fabs(np[0].u)) {
111
112                      if (np[1].id < np[0].id) { /* smallest breaks tie */
113
114                          np[0].id = np[1].id;    np[0].u  = np[1].u;
115                          np[0].s  = np[1].s;     np[0].t  = np[1].t;
116                      }
117                  }
118              }
119
120          } /* end if(np[1].id....) */
121
122      } /* end for(dir) */
123
124
125      /*  Since there is no direct connection between nodes 0 and 8, we once
126       *  again destroy the beauty and generality of the hypercube so that we
127       *  can be sure that 0 and 8 have the best candidate for pivot.
128       */
129
130      if (dim == 4) {
131
132          if ((me % 8) == 0) {                  /*  Nodes 0000 and 1000          */
133
134              directional_receive(me, dim, 0x1, (char *) np, len);
135          }
136
137          if ((me % 8) == 1) {                  /*  Nodes 0001 and 1001          */
138
139              directional_send(me, dim, 0x1, (char *) np, len);
140          }
141      }
142 }
143 /* End elect_next_pivot() ----------------------------------------- */
144
145
146 /* This is only the first part of this file.  The rest would be similar to
147  * gfppnode.c
148  *
149  * --------------============== EOF gfpcnode.c  ==========------------- */
```

360

LIST OF REFERENCES

[1] P. Morrison and E. Morrison, editors. *Charles Babbage and His Calculating Engines.* Dover Publications, Inc., New York, 1961.

[2] John P. Hayes. *Computer Architecture and Organization.* McGraw–Hill Book Company, New York, Second edition, 1988.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[4] Michael Slater. *Microprocessor-Based Design: A Comprehensive Guide to Effective Hardware Design.* Prentice–Hall, Inc., Englewood Cliffs, NJ, 1989.

[5] INMOS Limited, Bristol, U.K. *Transputer Handbook,* 1989.

[6] C. Gordon Bell. The Future of High Performance Computers in Science and Engineering. *Communications of the Association for Computing Machinery,* 32(9):1091–1101, September 1989.

[7] J. J. Dongarra, James R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1978.

[8] Jack J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee and Oak Ridge National Laboratory, Author's electronic mail address: dongarra@cs.utk.edu., September 1991.

[9] Ware Myers. Caltech Dedicates World's Most Powerful Supercomputer. *IEEE Computer,* 24(7):96–97, July 1991.

[10] Parsytec. Parsytec News, Summer 1991.

[11] David Gilbert. Interview: Richard Hamming. *IEEE Computing Futures,* pages 10–17, Spring 1991.

[12] Kenneth G. Wilson. Grand Challenges to Computational Science. *North-Holland Future Generation Computer Systems,* 5:171–189, 1989.

[13] David Harel. *Algorithmics: The Spirit of Computing.* Addison–Wesley Publishing Company, Wokingham, England, 1987.

[14] David May. Towards General Purpose Parallel Computers. Invited lecture at Transputing '91, Sunnyvale, California, 25 April 1991.

[15] David Goldberg. What Every Computer Scientist Should Know About Floating–Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[16] Uno R. Kodres. Unpublished discussions, lectures, and course notes in computer science. Given at the U. S. Naval Postgraduate School, Monterey, California, 1991.

[17] Alston S. Householder. *Principles of Numerical Analysis*. McGraw–Hill Book Company, New York, 1953.

[18] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952. Research Paper 2379.

[19] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw–Hill Series in Supercomputing and Artificial Intelligence. McGraw–Hill Book Company, New York, 1987.

[20] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024–Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[21] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, Second edition, 1989.

[22] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Numerical Linear Algebra and Optimization*. Addison–Wesley Publishing Company, Redwood City, CA, 1991.

[23] William B. Gragg. Unpublished discussions, lectures, and notes in numerical linear algebra and scientific computing. Given at the U. S. Naval Postgraduate School, Monterey, California, 1991.

[24] William B. Gragg and John A. Trangenstein. Linear Algebra Lecture Notes. Department of Mathematics, University of California, San Diego, 1978.

[25] INMOS Limited, Bristol, U.K. *The T9000 Transputer Products Overview Manual*, 1991. Prentice–Hall, Inc., Englewood Cliffs, NJ.

[26] Gregory R. Bryant. Design, Implementation and Evaluation of an Abstract Programming and Communications Interface for a Network of Transputers. Master's thesis, U. S. Naval Postgraduate School, Monterey, CA, June 1988.

[27] Charles L. Seitz. The Cosmic Cube. *Communications of the Association for Computing Machinery*, 28(1):22–33, January 1985.

[28] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1988.

[29] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1989.

[30] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1985.

[31] Michael B. Esposito. Comparative Study of the Performance of Hypercube versus Dynamically Configurable Topologies in a Transputer Network. Master's thesis, U. S. Naval Postgraduate School, Monterey, CA, TBD 1991. To be published.

[32] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, Inc., New York, 1972.

[33] Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, Publishers, San Diego, CA, Third edition, 1988.

[34] Donald E. Knuth. Big Omicron and Big Omega and Big Theta. *SIGACT News*, pages 18–24, April–June 1976.

[35] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison–Wesley Publishing Company, Reading, MA, 1989.

[36] INMOS Limited, Bristol, U.K. *The Transputer Databook*, Second edition, 1989.

[37] Grady Booch. *Software Engineering with Ada*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, Second edition, 1987.

[38] Dimitri P. Bertsekas and Robert G. Gallager. *Data Networks*. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1987.

[39] Youcef Saad and Martin H. Schultz. Topological Properties of Hypercubes. (Research Report YALEU/DCS/RR–389):1–17, June 1985.

[40] Richard W. Hamming. *Coding and Information Theory*. Prentice–Hall, Inc., Englewood Cliffs, NJ, Second edition, 1986.

[41] Fred Buckley and Frank Harary. *Distance in Graphs.* Addison–Wesley Publishing Company, Redwood City, CA, 1990.

[42] William E. Boyce and Richard C. DiPrima. *Elementary Differential Equations and Boundary Value Problems.* John Wiley & Sons, Inc., New York, Fourth edition, 1986.

[43] Richard Haberman. *Elementary Applied Partial Differential Equations with Fourier Series and Boundary Value Problems.* Prentice–Hall, Inc., Englewood Cliffs, NJ, Second edition, 1987.

[44] Frank R. Giordano and Maurice D. Weir. *Differential Equations: A Modeling Approach.* Addison–Wesley Publishing Company, Reading, MA, 1991.

[45] Intel Corporation. *iPSC/2 Programmer's Reference Manual.*

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, VA 22304-6145

2. Commandant of the Marine Corps   2
    Code TE-06
    Headquarters, U. S. Marine Corps
    Washington, DC 20380-0001

3. Dudley Knox Library   2
    Code 52
    Naval Postgraduate School
    Monterey, CA 93943-5002

4. Chairman, Computer Science Department   2
    Code CS
    Naval Postgraduate School
    Monterey, CA 93943

5. Chairman, Department of Mathematics   2
    Code MA
    Naval Postgraduate School
    Monterey, CA 93943

6. Dr. Uno R. Kodres   3
    Code CS/Kr
    Naval Postgraduate School
    Monterey, CA 93943

7. Dr. William B. Gragg   1
    Code MA/Gr
    Naval Postgraduate School
    Monterey, CA 93943

8. Dr. John R. Thornton   1
    Code MA/Th
    Naval Postgraduate School
    Monterey, CA 93943