

CoMet : A Synthetic Benchmark for Message-Passing Architectures

Nalini Ganapati

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

The thesis “CoMet : A Synthetic Benchmark for Message-Passing Architectures” by Nalini Ganapati has been examined and approved by the following Examination Committee:

Dr. Jonathan Walpole
Assistant Professor
Thesis Research Adviser

Dr. Steve Otto
Assistant Professor
Thesis Research Adviser

Dr. Joseph E. Brandenburg
Intel Corporation

Dedication

To my parents.

Acknowledgements

I would like to thank Jonathan Walpole and Steve Otto for introducing me to the field and guiding me throughout. I would like to take this opportunity to thank Joe Brandenburg for his valuable comments on the thesis. I would also like to thank my husband and children for all their moral support and endurance while I worked on my thesis.

Contents

Dedication	iii
Acknowledgements	iv
Abstract	vii
1 Introduction	1
2 Overview of existing benchmarks	4
2.1 Synthetic Benchmarks	6
2.1.1 Whetstones	7
2.1.2 Dhrystones	7
2.2 Kernel/Algorithm Benchmarks	8
2.2.1 Livermore Fortran Kernels	9
2.2.2 NAS Kernels	9
2.2.3 LINPACK Benchmarks	9
2.3 Application Benchmarks	10
2.3.1 SPEC Benchmarks	10
2.3.2 Perfect Benchmarks	11
2.3.3 Euroben Benchmarks	12
2.3.4 Genesis	13
2.4 Summary	13
3 Description of the CoMet Benchmark	16
3.1 Methodology	17
3.1.1 Time measurement	18
3.1.2 Clock synchronization	18
3.1.3 Clock resolution	19
3.1.4 Reducing measurements to a single number?	19
3.1.5 Benchmark Topology	20

3.1.6	Coding of benchmark	20
3.1.7	Integrity of benchmarks	21
3.2	Structure of benchmark	21
3.2.1	The Basic Communication-Related Component	22
3.2.2	The Matrix-Related Component	23
3.3	Description of Benchmark Kernels	24
3.3.1	Echo	24
3.3.2	Pairwise Exchange	26
3.3.3	Broadcast	27
3.3.4	Global Reduction	27
3.3.5	Global Synchronization	28
3.3.6	Contention	28
3.3.7	Overlap of communication with computation	29
3.3.8	Update guard	31
3.3.9	Shift Matrix	33
3.3.10	Transpose Matrix	34
3.3.11	Row and Column Broadcast	34
3.4	Support Functions/Macros	35
3.4.1	Initialization and Cleanup Phases	35
3.4.2	Message Passing	37
3.4.3	Other related macros	39
3.5	Porting of CoMet	40
4	iPSC/860 Implementation	41
4.1	Overview of iPSC/860 architecture	41
4.2	Communication Protocols in NX/2	42
4.3	Porting the benchmark	45
5	Results and Analysis	51
5.1	Basic communication kernels	51
5.2	Matrix-related kernels	65
6	Conclusions	68
6.1	Future Work	69
	Bibliography	71

Abstract

CoMet : A Synthetic Benchmark for Message-Passing Architectures

Nalini Ganapati

Oregon Graduate Institute of Science & Technology, 1993

Supervising Professors:

Dr. Steve Otto

Dr. Jonathan Walpole

There is currently a proliferation of parallel supercomputers made possible by rapidly changing technologies. They have diverse architectures and support different programming models and parallel paradigms. An important and emerging class among parallel architectures are the distributed memory multiprocessors. We focus on the problem of evaluation and the need to provide common performance metrics for these distributed memory machines.

Of the existing benchmarks, uniprocessor benchmarks such as Whetstones and Dhrystones are not sufficient for benchmarking distributed memory architectures. Other benchmarks port full-length codes from a range of typical applications. However, since distributed memory machines have a partitioned address space, and communication between the individual processors in the system is implemented via message passing, porting application-level benchmarks to these machines requires extensive restructuring of the original code. It is also difficult to extract fundamental communication characteristics from the information returned by application-level benchmarks. Knowledge of these

fundamental characteristics guide users in porting and in making realistic guesses of the performance of their applications on distributed memory systems. In this context, we propose CoMet(**C**ommunication **M**etrics), a synthetic, but comprehensive, benchmark that is easily portable and scalable for distributed memory architectures. CoMet, which is written in C, is based on common communication patterns found in typical parallel scientific algorithms and adds to the information provided by other application-based benchmarks.

This thesis overviews existing benchmarks in the context of evaluating distributed memory message-passing machines, presents the CoMet benchmark and describes an implementation of CoMet on Intel's iPSC/860¹.

¹iPSC/860 is a registered trademark of Intel Corporation

Chapter 1

Introduction

There is currently a proliferation of parallel supercomputers made possible by rapidly changing technologies. They have very diverse architectures, and support different programming models and parallel paradigms. Broadly, they can be classified into those supporting SIMD(Single Instruction - Multiple Data) and MIMD(Multiple Instruction - Multiple Data) parallelism [Fly66], and as UMA(Uniform Memory Access), NUMA (NonUniform Memory Access) and NORMA(NO Remote Memory Access) machines based on the characteristics of memory accesses. Different parallel programming models exist to make efficient use of the parallel resources of such computers. This diversity is problematic when it comes to evaluating and comparing the parallel computer systems. Distributed memory architectures, based on message passing(DMMP), are examples of NORMA machines. They form a large and important class of parallel machines. This thesis focuses on the problem of evaluating the distributed memory class of machines and the need to provide common and meaningful performance metrics.

Earlier benchmarks have reported performance in terms of MIPS(Million Instructions per Second), Whetstones [Wei91], and Dhrystones [Wei84] [Wei91]. They evolved in the late 70's and early 80's before the advent of architectures based on reduced instruction sets, caches, etc. Consequently, they have become outdated and cannot completely measure the performance of even modern-day RISC-based workstations, leave alone supercomputers.

Evaluating a computer is a function of numerous issues that arise from the interplay of the application, the system software, the programming language, optimization effort, and

various other hardware and software characteristics. Isolating the effects of these different parameters while evaluating a computer is difficult, and ultimately only the resultant execution time for the application is important to the end user. Hence, the general trend in evaluating supercomputers is to compose benchmarks consisting of kernel and application programs that are representative of many scientific and engineering codes and base the performance metrics on the execution times of these codes. These benchmark codes are ported to the target system with hopefully minimal changes, and allow users to predict the performance of their own applications based on the benchmark results.

Although porting application-based codes for benchmarking purposes yields realistic performance numbers, it is difficult to extract fundamental performance characteristics for the target system from the benchmark results and is very time consuming. The basic performance characteristics in distributed memory machines not only allow users to model the performance of their application based on the characteristics, but also provide the user with guidance as to how to port the application. Also, porting most of the existing application-based benchmarks to distributed memory machines requires considerable effort in terms of extensive restructuring of the code and inserting explicit communication calls.

At this point, a few benchmarks are available specifically for evaluating distributed machines. These are Genesis from University of Southampton[AGH⁺92], NAS Parallel Benchmarks[BBDS92], MPLinpack[DS86] and CPEP (the CalTech Performance Evaluation Project)[MBF⁺90]. All the benchmark suites consist mostly of application-based benchmark programs with almost no synthetic codes to measure the basic communication characteristics of the distributed memory system. To add to the fundamental performance characteristics measured by the few synthetic kernels in the existing benchmarks, we have developed a more comprehensive, synthetic benchmark - CoMet(**C**ommunication **M**etrics) - that is easily ported onto parallel systems based on loosely-coupled networks and MIMD architectures. The benchmark includes many communication patterns found in typical scientific and engineering applications, and is intended to provide meaningful information on basic machine characteristics. The

benchmark is entirely written in C and has been implemented on Intel's iPSC/860¹. The results of the benchmark are presented in the form of graphs and in terms of fundamental numbers such as message latencies and communication bandwidth. We suggest that it is difficult and dangerous to reduce the results to a single number. Instead, our benchmark results, combined with other existing benchmarks, will enable users to construct realistic expectations of the performance of their own programs.

The rest of the thesis is organized as follows. In chapter 2, an overview of existing benchmarks in the context of evaluating distributed memory machines is presented. The structure of CoMet is described in Chapter 3. We discuss an implementation of the benchmark on the iPSC/860 in Chapter 4. The results of the iPSC/860 implementation, together with some analysis are presented in Chapter 5. Finally, Chapter 6 concludes the thesis and discusses possible future directions.

¹iPSC/860 is a registered trademark of Intel Corporation

Chapter 2

Overview of existing benchmarks

This chapter overviews existing benchmarks and performance metrics in the context of their ability to evaluate distributed memory - message passing based parallel systems (DMMP).

The general trend in evaluating computers is to compose benchmarks consisting of kernel and application programs and measure their execution times on the machines to evaluate performance [Cyb91] [Dix91] [BGL91] [AGH⁺92] [Hey91] [vdS91]. The types of code used in benchmarking, especially scientific benchmarking, can be categorized as follows

- Synthetic : These are code segments thought to generally reflect often used program constructs and basic machine functions. They are simple and easy to adapt to different systems. Whetstones and Dhrystones are examples of such benchmarks.
- Kernel/Algorithm : These are code segments extracted from real applications. They represent some often used algorithms and are thought to be representative of the significant portions of execution times in real applications. Examples are Livermore Loops, NAS kernels and LINPACK routines.
- Application Based : These code segments are implementations of actual applications solving existing, well-defined problems, usually in the scientific area. These benchmarks reflect the performance characteristics exhibited by real applications. The Perfect Benchmarks and the Euroben suite are some examples.

Apart from these general benchmarks, there are micro-measurements and global performance formulae to estimate overall performance of supercomputers taking into consideration the interplay between cache, pipeline, parallel architecture and user software (more precisely data structures) [SH91]. The results of these measurements represent the hardware characteristics of the computer better than the general benchmarks.

Traditionally, computer speeds are expressed in terms of MIPS (Million Instructions per second) or MFLOPS (Million Floating Point Operations per second). Some MIPS numbers reflect only the number of NOOP instructions executed per second. With the advent of newer RISC-based architectures, these numbers have to be interpreted with caution because increased MIPS ratings are often offset by an increase in the number of instructions required for high level language constructs. Because of these problems, MIPS has been redefined to denote a performance factor relative to the performance of VAX 11/780 [Wei91].

For the vast majority of supercomputer applications, the most appropriate low-level performance measurement unit is MFLOPS. Peak MFLOPS are a measure of the maximum number of floating point operations per second in ideal circumstances. An alternative approach is to calculate MFLOPS based on an application or a set of applications. However, porting applications to DMMP machines requires rigorous changes in terms of extensive restructuring of code and explicit communication calls. Furthermore, even if the applications were ported, many parallel algorithms add redundant arithmetic operations to reduce the overall number of steps leading to inflated MFLOPS numbers for the application on that particular system. Benchmarks such as LINPACK (discussed later), which use MFLOPS as a performance metric, assign a nominal FLOP count to the application to avoid this problem. Load imbalance and communication overheads also account for significant performance degradation in the DMMP machines. It is difficult to extract the fundamental characteristics of a system from just MFLOPS. Even if some application programs were ported, it would be hard to predict the behavior of other applications or guide the programmer in porting other codes from MFLOPS numbers alone.

Another performance measure widely used in benchmarking supercomputers, and probably the most misunderstood and confusing, is Speedup. Speedup of a benchmark program on N processors is defined to be the ratio of the elapsed time for the benchmark to run on a single node system with the elapsed time to run the same benchmark in parallel on N processors [Hoc91]. To a some extent, Speedup does reflect the scalability of the parallel architecture and the algorithm used. That is, a Speedup equal to M on N processors means that the benchmark application executes M times faster on the N processor system than it does on a single processor. However, speedup measures from different machines are not comparable, since each represents relative speedup from a single processor version of that machine, i.e. knowledge of absolute performance is lost while deriving Speedup. In fact, a system with the worst speedup may execute the benchmark program faster than another system whose speedup values are better. Hockney [Hoc91] defines Speedup as :

$$Speedup = T_1/T_p = one_processor_time/p_processor_time,$$

and characterizes it as a number without units. Comparing algorithms even on the same system is like comparing two numerical values in different units.

A related measure, Efficiency, is defined as the ratio of Speedup to the number of processors, and faces the same problems as Speedup when used as a performance metric. However, Speedup and Efficiency do indicate how well a particular architecture scales with respect to a particular algorithm. In that sense, Speedup and Efficiency can be used in conjunction with other metrics to reflect scalability of the multiprocessor architecture with respect to a particular algorithm.

2.1 Synthetic Benchmarks

Synthetic benchmarks are small programs written especially for benchmarking. They do not generally perform any useful computation, but are thought to capture the average characteristics of a real program and reflect the basic hardware characteristics of the computer system.

2.1.1 Whetstones

The Whetstone benchmark [Wei91] was one of the first synthetic programs designed solely for benchmarking. The benchmark is comprised of ten modules containing statements pertaining to integer arithmetic, floating point arithmetic, conditional statements, system calls, etc. Measurement is in terms of the number of Whetstone instructions executed per second from an intermediate language produced when the program sample is run with a Whetstone compiler. Weights are attached to the different modules so that the distribution of the Whetstone instructions is similar to that of the original program sample. The benchmark was intended for numerical and other scientific applications and as such has a high percentage of floating point data and operations.

The benchmark was developed in 1976, before many of the recent advances in computer architecture. It also does not represent the present day mix of instructions on advanced RISC based systems with register windows, on-chip caches, etc. Therefore, Whetstone numbers are to be interpreted with caution. Since the benchmark is very small, it is sensitive to reorderings of source code. A simple reordering of the library modules can lead to a 100% cache hit rate even for fairly small caches. The benchmark does not use any local variables, and hence there would be no visible performance improvement on a SPARC system, for example, which is based on register windows. On the other hand, since the benchmark uses a few global variables repeatedly instead of local variables, allowing a compiler to place the heavily used global variables in registers boosts the Whetstone performance numbers.

Since the Whetstone benchmarks do not completely capture the performance of present day computational units, it can be used in a limited way to measure the performance of DMMP systems.

2.1.2 Dhrystones

Dhrystone [Wei91] [Wei84], another synthetic benchmark, was developed along the lines of Whetstone by R.P.Weicker in 1984 to evaluate computers for non-numeric system

programming. It was originally written in Ada, however the popular versions are currently all written in C. The benchmark consists of 12 procedures and they are included in one loop with 94 statements. The results are reported as Dhrystones per second. No attempt is made to avoid compiler optimizations since the benchmark was intended to reflect actual programming style. A sizable part of the Dhrystone benchmark uses string manipulation and a simple tuning of the strings function library with an optimizing compiler in-lining string functions for this benchmark could lead to exaggerated performances. RISC machines generally outperform CISC machines on this benchmark because of the large number of registers and the locality of data and cache. The working set of this program fits into the caches of most modern-day workstations. However, within the main measurement loop, the benchmark contains no loops and on small cache microprocessors this results in cache misses for almost all instructions.

The benchmark was intended for systems programming environments and as such contains no floating point operations in its loop. The benchmark can be used in a limited fashion : to evaluate integer performance of small machines with simple architectures. It cannot, however be used to benchmark supercomputers because typical applications on these systems make widespread use of floating point operations.

2.2 Kernel/Algorithm Benchmarks

In one sense, kernel/algorithm benchmarks are an improvement over the synthetic benchmarks because the code segments used in the benchmarks more closely model real applications, i.e. they are frequently used program constructs or algorithms which account for much of an application's execution time. However, these benchmarks do not test every software and hardware aspect of the computer and in that sense fall short of being a complete benchmark. They are also harder to port fairly, and it is difficult to isolate the different hardware characteristics from the measurements.

2.2.1 Livermore Fortran Kernels

Livermore Fortran kernel benchmarks [Feo88] consist of 24 often used Fortran programs (kernels) that challenge a Fortran Compiler to produce the best optimized code. The benchmark evaluates the MFLOPS rate for each kernel and the overall average, harmonic and geometric means of the MFLOPS. They also present the best and worst case Fortran code to get a realistic picture of the floating point performance of the computer. The kernels contain a high number of floating point operations and array accesses and quite a few of them contain vectorizable code.

2.2.2 NAS Kernels

The NAS (Numerical Aerodynamic Simulation)[BGL91] set of benchmarks was developed at the NASA Ames Research Center. The benchmark consists of 7 Fortran programs, with each program consisting of a loop that iteratively calls particular subroutines. All the kernels contain mostly vectorizable code and hence are suited to measure the performance of tightly coupled systems.

2.2.3 LINPACK Benchmarks

The LINPACK [Don89] [Wei91] benchmark evaluates the performance of a system by its ability to solve a set of linear equations. The performance metric of LINPACK is reported in terms of execution times as well as MFLOPS. There are different versions of LINPACK for benchmarking and several optimized library routines are available to implement LU(lower-triangular, upper-triangular) factorization, back-substitution and other matrix-matrix techniques to solve the set of linear equations. One version of the benchmark uses LU factorization and back-substitution for dense linear systems of dimensions 100 or 1000. Included with these benchmarks is a driver routine which generates the test matrices and tests the solutions to within acceptable limits of accuracy.

This benchmark is popular among vendors. LINPACK has been ported onto many DMMP systems [DS86] and many vendors quote LINPACK MFLOPS numbers for their

systems. Unlike Whetstones and Dhrystones, the benchmark is highly sensitive, in a consistent way, to cache size, memory size and organization, optimizations, register allocation strategies, etc. It could be used to evaluate the single node performance of distributed memory systems. However, it would still be difficult to understand the behavior of the entire distributed memory system from just LINPACK numbers.

2.3 Application Benchmarks

The previous approaches of benchmarking based on extracting common code constructs and/or kernels and/or algorithms from real applications is to be cautiously interpreted, particularly with regard to its statistical soundness [BGL91]. To better understand the behavior of a system, the benchmarks in this category address this problem by drawing full length typical application programs from scientific and engineering fields as benchmarks to exercise almost every aspect of hardware and software of any given system.

2.3.1 SPEC Benchmarks

The SPEC (Systems Performance Evaluation Cooperative) Benchmark was the result of a consortium of computer manufacturers and vendors whose main goals were to come up with a fair, standard benchmark that did not favor any particular architecture and software configuration [Dix91]. The benchmark suite consists of 10 programs - 6 written in C and 4 written in Fortran - from a typical workstation load, usually a mix of compiling, editing, and running engineering and scientific applications - Table 2.1.

The performance metrics for the SPEC benchmarks are with respect to elapsed execution times for the programs in the benchmark suite and are reported as

- SPEC reference time (in seconds) which is the time taken by a DEC VAX 11/780 To run each program in the benchmark suite.
- SPEC ratio which is the ratio between the elapsed time for a particular machine and the SPEC reference time for each program in the suite.

GNU C Compiler	System software
ESPRESSO	Device simulation
SPICE 2GS	Circuit simulation
DODUC	Thermodynamics
NASA7	Fluid dynamics
LISP interpreter	List Processing
EQNTOTT	Logic Programming
MATRIX300	Matrix Multiplication
FPPP	Quantum Chemistry
TOMCATV	Mesh Generation

Table 2.1: Application codes comprising the SPEC benchmarks(release 1.0)

- SPECmark which is the geometric mean for the ten SPEC ratios for the machine.
- SPEC thurput which is a measure of work performed by a particular machine running a scaled load against the SPEC reference machine (VAX 11/780).

2.3.2 Perfect Benchmarks

The Perfect Benchmarks are the result of a collaboration from academia and various research centers. The entire suite consists of a collection of 13 application-level Fortran programs from scientific and engineering areas totaling over 60,000 lines of code. The main goals were to come up with an application-based benchmark which could identify, explain and possibly correct performance anomalies in high performance computing systems. In other words, the goal was to seek better insight into architecture and software rather than provide just benchmark numbers [Cyb91] [BGL91].

Each of the programs in the collection was originally written for CDC, CRAY, IBM and MARK III and solves some well-defined problem in the field. The general methodology adopted to obtain baseline measurements was to port each of the programs in the suite onto the target machines with as few changes as possible. Loosely speaking, the code remains invariant even if the program has a built-in bias towards a particular architecture. Once baseline measurements are obtained, manual optimization to the code

is allowed, including adding compiler directives, loop modification, I/O modification, code rewriting, etc. as long as the optimized code addresses the same problem as the original. The optimized code is then used to obtain the optimized measurements. The performance measurements are reported for each program in the suite in terms of CPU time, elapsed (refers to wall clock) time and MFLOPS. The benchmark also produces a verification output file for each of the programs. In addition to the measurements provided by the benchmark itself, an “optimization diary” is required to be maintained noting the type and level of optimization and the execution times obtained for every modification made.

2.3.3 Euroben Benchmarks

Application-based benchmarking may be meaningful because the large, real-life application programs which form part of the benchmark test almost every possible architecture and system software aspect of the target system. However, full-scale applications are not nearly as portable as the kernel/algorithm codes that were developed, in the first place, with portability in mind. The Euroben group of benchmarks addresses this problem and provides a layered approach to benchmarking. The Euroben benchmark code segments vary from very simple kernels to complete applications. The simple programs provide basic measurements of the performance of basic operations, whereas the more complicated codes provide more information on the behavior of the system with respect to particular applications.

The benchmark is distributed in four Fortran modules, targeted at 64-bit precision machines, and another benchmark for distributed memory machines named Genesis. The Genesis benchmark, developed at Southampton University, UK is separated from the other modules because of the rigorous changes required by the distributed memory machines in terms of explicit communication calls and a general restructuring of the code.

The contents of the first Fortran module are codes to test basic arithmetic operations, memory bank conflicts, intrinsic mathematical functions and to measure the memory and

communication bottlenecks. The second module consists of eight codes based on kernels like the LINPACK. The third module contains seven more complex algorithms including a fast elliptic solver. The fourth module contains ten full application codes ranging from circuit simulation to weather simulation.

Any combination of programs from any of the modules can be run as a benchmark. Hence, the benchmark does not reduce the performance measurement to a single number. The benchmark simply attempts to add to the information provided by the vendor in terms of providing a reasonable overview of the target system.

2.3.4 Genesis

The Genesis benchmark is distributed as part of the Euroben benchmark suite to address the special needs of evaluating distributed memory machines. The benchmark is written in Fortran 77 and follows the same layered approach as the Euroben suite. The synthetic part of the benchmark consists of programs relating to unidirectional/bidirectional transfer of messages, a matrix transpose algorithm and barrier synchronization. The rest of the benchmark is comprised of application codes related to matrix multiply, LU factorization, QR decomposition, FFT(Fast Fourier Transform), Poisson solvers, Helmholtz solvers, quantum chromodynamics, molecular dynamics, general relativity and local particle-mesh simulation.

2.4 Summary

Existing benchmarks can be categorized broadly into synthetic and application-based benchmarks. While the synthetic benchmarks are easy to port and provide valuable basic information of the system, they are only a simple representation of more complex programs. To obtain realistic performance statistics of systems, benchmark initiatives such as SPEC, Perfect and Euroben run full-length typical application programs and base their performance measurements on the execution times of the applications.

Distributed memory systems such as a loosely coupled network of workstations or

a hypercube, require considerable effort to port full application benchmarks such as SPEC and Perfect. Even among DMMP machines, communication hardware topologies vary widely. Common examples include the hypercube (Intel's iPSC/860), mesh (Intel's Delta¹ and Paragon²) and tree (Thinking Machine's CM-5³). Therefore, porting full length applications across machines with widely different communication topologies is unrealistic from a benchmarking perspective. In fact, as mentioned in the previous section, the Perfect Benchmarking effort characterized porting application level codes to massively parallel machines based on hypercubes and other topologies as difficult, and concludes that it is very difficult to come up with a comprehensive applications-based DMMP benchmark [Cyb91]. Furthermore, unless a range of applications is considered, it is unlikely that all aspects of the machine's communication system will be tested.

The existing distributed memory benchmarks, notably Genesis, concentrate primarily on evaluating systems based on full application codes. The performance of a DMMP system is dependent on not only the computational performance of its individual nodes but also on its communication capabilities and the extent to which computation can be overlapped with communication. Evaluation of these basic features and characterizing them with performance numbers is important because it can guide users to port their algorithms in specific ways. Performance evaluation of a single node is easily obtained by running benchmarks such as LINPACK on any individual node. For other characteristics, the Genesis benchmark contains a few synthetic programs to provide basic information about the message-passing capabilities of the DMMP system. However, there are too few synthetic programs, and they do not provide the user with information regarding the behavior of the system under loads or the extent to which computation can be overlapped with communication. Also, it would help users in predicting performances of their own applications, if the benchmark was more comprehensive and included typical communication patterns without incurring the expense of porting large applications. Thus,

¹Delta is a registered trademark of Intel Corporation

²Paragon is a registered trademark of Intel Corporation

³CM-5 is a registered trademark of Thinking Machine Corporation

our approach, discussed in the next chapter, is to add to the information provided by the Genesis benchmark by building a synthetic benchmark to simulate many commonly occurring computation and communication patterns in real scientific applications under different conditions.

Chapter 3

Description of the CoMet Benchmark

Parallel Computers based on distributed-memory differ from single processor machines and shared-memory multiprocessors in that work has to be explicitly distributed to processing nodes that can only communicate by exchanging messages. Running one's own applications is the best way to evaluate any system, but porting applications can take a lot of work and time and hence is often unrealistic. The next best way to evaluate a system [DMW87] is to have a benchmark which :

1. is written in a high-level language, making it portable across different machines,
2. is representative of some kind of programming style (for example, systems programming, numerical programming, or commercial programming),
3. can be measured easily, and
4. is widely available.

For parallel systems based on a number of processors, one could add to the list, the capability for the benchmark to scale with the number of processors in the system.

Our benchmark is written in C, a high-level language. We have strived to make it representative of communication patterns commonly found in scientific processing on distributed architectures. However, in doing so, we have excluded all architectures that are not based on message passing. In fact, some of the requirements mentioned above are contradictory [DMW87]; it is almost impossible to have a benchmark representative of some style without some complexity, making measurements on a wide range of machines

difficult. We justify the exclusion of other architectures because of the special requirements of loosely coupled architectures and of the applications that run on them. If we characterize the computational capacity of a single processor as the *peak performance*, then peak performance degradation on most systems based on multiple processors and loosely coupled environments stems not only from load imbalances but also from inefficient communication. Since, load balance is usually dependent on the type of algorithm used to implement the application, and since our benchmark is not application-level, we do not capture it. Communication efficiency, on the other hand, is determined by operating system hardware and software. We attempt to capture this information through the use of a comprehensive, synthetic benchmark that can be scaled to run on any number of processors. We believe our efforts at benchmarking distributed memory machines add to the information provided by other existing benchmarks described in the previous chapter.

3.1 Methodology

Message passing programs that run on DMMP computers consisted of two parts - a host program running on a front end machine and a node program running in parallel on the nodes that comprise the DMMP system. The host program would then load the node programs onto required nodes and also serve the I/O requirements for the entire system. The current trend is to adopt a Single Program, Multiple Data (SPMD) style of programming. The same program is loaded on all the node processors with each of the processors proceeding at its own pace between points of synchronization. I/O requirements are usually served by independent file subsystems, for example the iPSC/860 is served by its concurrent file systems attached directly to the system.

Our benchmark adopts the SPMD programming style and the benchmark program is loaded on all the nodes. Only one instance of the benchmark is run on each node as our measurements are restricted to evaluating communication in the systems being evaluated. A host program is optional and the responsibility of the user. The function

of the host program or a shell script, when necessary, is limited to

1. acquire the required number of nodes to run the benchmark on,
2. load the benchmark programs on the acquired nodes and
3. number each node uniquely in the range $0..P-1$,

where P is the number of acquired processors. Numbering the node processors can be done either by the host program or the DMMP system itself as long as the node numbers are mapped uniquely in the range $0..P-1$.

3.1.1 Time measurement

For most benchmarks, the basic measurement is time and all other quantities and measurements are derived from this fundamental measure. Benchmark times can be further derived from wall-clock time, CPU time, or any other time measure offered by the computer system. Different systems have different interpretations of CPU time and as such have different components of the total time included in the CPU time. Even if the interpretation was uniform among different systems, systems implement blocking message passing primitives by either busy waiting or by suspending the process making time measurements based on CPU time further dependent on implementation. An alternate approach is to base measurements on elapsed time. The main disadvantage of measuring program performance in terms of elapsed time is the sensitivity to varying system loads. We base our time measurements on elapsed times, but execute the benchmark on a dedicated system to alleviate the timing problems sensitive to system load.

3.1.2 Clock synchronization

Because DMMP systems are loosely coupled environments, many systems do not have a global or synchronized clock. Instead, there is a clock on each of the nodes. This leads to problems of clock synchronization. To avoid these problems, we base all our measurements on timings obtained on one designated node processor in the distributed memory system.

3.1.3 Clock resolution

In kernel benchmarks, generally the message start-up times are a few micro-seconds or less. Unfortunately, clock resolution provided on workstations, for example, is often on the order of 10ms. To obtain more accurate timing figures, we enclose our benchmark codes in a repeat loop and time the entire loop. Also, to prevent optimizing compilers from removing the repeat loop, the benchmark codes enclosed with such loops have a call to an external dummy function from within the loop using the increment variable as the dummy function parameter. To avoid the overheads of executing the repeat loop and the call to the dummy function, the repeat loop is separately timed without the kernel code and the execution time for the loop is subtracted from the total elapsed time to obtain the true elapsed time.

3.1.4 Reducing measurements to a single number?

All our performance metrics are derived from the true elapsed times for the different parts of the benchmark. SPEC[Dix91][BGL91] reduces the benchmark elapsed times to a SPECmark and then to a harmonic mean of the 10 SPECmarks. On the one hand, single numbers are attractive because they summarize the performance of the computer in a single number allowing a straight-forward comparison of different parallel systems. On the other hand they can be quite meaningless, especially if it is difficult to derive some intuitive meaning from the number. Performance between parallel machines can vary by factors of a thousand or more making it really difficult to arrive at a single number that captures these differences of magnitude among parallel systems. Also, we would like users to make realistic guesses of the performance of their own codes based on our benchmark. A single number would imply a uniform performance on all application codes when in reality most codes perform differently. In this context our benchmark will not yield a single number. Instead, it will focus on graphing the elapsed times of the various kernels in the benchmark with respect to another parameter such as message size, matrix size or number of processors. The basic communication performance parameters,

such as startup times and bandwidths, should be interpreted from these graphs.

3.1.5 Benchmark Topology

DMMP systems can have varied architectures such as those based on a network of workstations or a hypercube or a tree-based communication network (CM-5) [Thi91]. It is unrealistic to expect the benchmark kernels to be completely architecture independent. We have instead focussed on capturing the topologies of the hardware into uniform data structures based on important data structures in parallel, scientific programs. Communication properties of some networks are dependent on one-hop links between adjacent nodes and multi-hop links between not-adjacent nodes. The benchmark contains a function which, given a node, identifies another node n -hops away. For matrix manipulations, one wants to identify nearest neighbor nodes for operations such as shift the matrix in a particular dimension, update a guard wrapper, etc. We have defined a two-dimensional matrix of processors, that is specified by the user. This data structure captures nearest neighbor mappings which allow a two-dimensional application matrix, to be block decomposed onto the underlying nodes in a manner that keeps adjacent blocks in the decomposed matrix on adjacent or nearest nodes. The matrix that captures the topology of the hardware for matrix manipulations, can be specified by hand or in any other way. For example, we have used gray codes to specify the matrix in our implementation for the iPSC/860 [FJG⁺88].

3.1.6 Coding of benchmark

Although there have been several attempts at message passing standards, such as PARMACS[BDH90] and PICL[Gei90], none have been completely accepted as standards. Rather than build our libraries based on PARMACS or PICL, we have implemented them using high-level macros and functions to describe the different communication patterns that a user is allowed to implement or modify. This has the advantage of not adding any more overhead than is absolutely necessary while invoking the communication library functions from the kernels in the benchmark.

3.1.7 Integrity of benchmarks

To enhance uniformity of comparisons and to reduce the chances of reporting misleading information on the part of users, we specify the functions/macros the user can modify. The user is not supposed to alter any other parts of the benchmark program. Also, we allow the user/vendor to use any level of optimization on the premise that if the vendor went ahead to build special compiler techniques that optimize communication arising out of common matrix manipulations, all users would ultimately benefit.

3.2 Structure of benchmark

To fully evaluate a loosely coupled parallel system, we need to evaluate the performance of each individual processor in terms of its scalar/vector capabilities, the performance of the system as a whole by simulating as many commonly occurring communication patterns as possible in both idle and loaded environments, and the extent to which communication can be overlapped with computation to utilize separate communication channels. The performance of an individual node is left to the user and can be easily evaluated using LINPACK or any of the numerous existing benchmarks and the scalar and vector capabilities of the node can be calculated in terms of r_∞ and $n_{1/2}$ ¹. The CoMet benchmark focuses on evaluating DMMP systems in terms of their communication-related capabilities.

CoMet is basically organized in two levels. The first is based on transfer of messages among different nodes with varying message sizes, and has the goal of measuring basic communication capabilities. The second is based on operations on a 2-D block-decomposed matrix to measure the performance of typical communication patterns.

The fundamental communication-related performance characteristics of a system based on distributed memory are the latency and the throughput with which messages can be transferred between the various nodes in the configuration. Low latency and

¹ r_∞ describes the asymptotic performance that is approached as the vector length goes to infinity, while $n_{1/2}$ is the vector length needed to achieve half the asymptotic performance [Hoc91].

high bandwidth are the most desirable characteristics of any DMMP system since almost all other measurable quantities in the system are affected by these parameters. Therefore, the first level of the CoMet benchmark measures these basic characteristics of a using synchronous and asynchronous (when available on the system) message-passing semantics.

Although scientific computation gives rise to a variety of communication patterns, matrix manipulation is at the heart of many scientific applications. The performance of many applications therefore, depends heavily on the efficiency of matrix-related operations. Therefore, the second level of the CoMet benchmark measures the system performance for various common matrix operations.

3.2.1 The Basic Communication-Related Component

CoMet’s basic communication-related benchmarks are organized into a number of ‘kernels’, each of which transfers varying length messages between system nodes.

The first benchmark kernel, **echo**, measures the cost of unidirectional message transfer between various pairs of nodes. Since many DMMP systems also provide bi-directional channels, we also provide a **pairwise exchange** kernel which measures the cost of bi-directional message exchange. However, this kernel has the potential to deadlock where non-buffered semantics are used. In addition to using unidirectional and bidirectional message transfers, many applications also use broadcasts and global reductions. For example, an application might search for an optimal solution by performing local searches for a maximum/minimum and broadcasting the local optimal solution to all other nodes. Hence, we have included **broadcast** and **global sum** kernels in the benchmark. Since broadcast and some flavor of global reduction are commonly used primitives, many systems provide explicit support for them (generally through optimized library routines). Finally, many applications also require some form of global synchronization. The **barrier synchronization** kernel measures the cost of global synchronization. Again, many systems have explicit support for this function in the form of specialized hardware or efficiently coded libraries.

Since applications do not always execute on otherwise idle systems, and since many DMMP systems allow communication to be overlapped with computation, we extend the above kernels to measure :

1. inter-node communication performance on loaded systems, and
2. the extent to which computation and communication can be overlapped.

We have measured the effect of the **echo** kernel in the presence of measured loads to characterize communication under contention. As with the **echo** kernel, **echo with contention**, is constructed for varying message lengths.

Finally, we have constructed a kernel to examine the extent to which communication and computation can be overlapped by first executing a **pairwise exchange** kernel and a computation based on DAXPY synchronously and then executing and timing them asynchronously. The difference between the two timing figures illustrates the ability of the machine to overlap communication and computation. Each of the above kernels is described in detail in Section 3.3.

3.2.2 The Matrix-Related Component

While the kernels in the previous level were limited to measuring the basic communication characteristics of a DMMP system, this conceptually higher level covers the kernels associated with matrix manipulation. We have implemented the following kernels, modeled on typical matrix manipulation patterns : **update guard**, **shift**, **transpose**, and **row/column broadcast**. These kernels are based on a two dimensional matrix which is block decomposed, with a one-element guard-wrapper² over a set of acquired nodes. The matrix is initialized with values that the user is not allowed to modify and a verification module is used to check the resultant matrix. All the matrix operations are invoked as macros or functions that can be in-lined, and the user is free to implement

²The guard wrapper conceptually surrounds the part of the matrix local to the node. It is used to store the non-local values of the matrix that are resident on neighboring nodes. The motivation behind the guard wrapper lies in the fact that a node can access some of the non-local values of the global matrix by simply referring to the values in this guard wrapper - see Figure 3.1.

Figure 3.1: View of the distributed matrix on each node. The shaded portions refer to the guard wrapper, while the inner values of the matrix refer to the parts of the matrix that remain local to the node.

the macros/functions in any efficient way. However, the user is not allowed to modify the setup and verification modules.

3.3 Description of Benchmark Kernels

3.3.1 Echo

The first benchmark kernel - **echo** - is based on a simple uni-directional transfer of messages between nodes (Figure 3.2). Based on our assumption that the clocks on a multi-processor system need not be synchronized, we measure time only on a single test node. Hence, to derive the time taken for a message to be transferred onto another node, a test node sends a message to another node and waits for the message to be sent back, half this time is considered to be the time taken for a single uni-directional transfer. The echo test is constructed for different sizes of messages and between nodes varying numbers of hops away.

```
begin
  for (hop = 0 to hop = maximum hops possible)
    begin
      for (message size = 0 to message size = over a page size)
        begin
          t1 = get time;
          if (test node) then
            send message to another node;
            receive message from other node;
          else
            if (communicating node)
              receive message from test node;
              send message back to test node;
            end if
          end if
          t2 = get time;
          echo time on test node = t2 - t1;
        end for
      end for
    end for
  end
```

Figure 3.2: Pseudo-code for **echo**

```

begin
  for (hop = 0 to hop = maximum hops possible)
  begin
    for (message size = 0 to message size = over a page size)
    begin
      t1 = get time;
      if (test node) then
        send message to communicating node;
        receive message from communicating node;
      else
        if (communicating node)
          send message to test node;
          receive message from test node;
        end if
      end if
      t2 = get time;
      pairwise exchange time on test node = t2 - t1;
    end for
  end for
end

```

Figure 3.3: Pseudo-code for **pairwise exchange**

3.3.2 Pairwise Exchange

Pairwise Exchange (Figure 3.3) is a variation of the echo benchmark. It measures the time to exchange messages between two nodes, much like the exchange of boundary information while solving partial difference equations using domain decomposition. This kernel typically executes faster than the corresponding **echo** kernel in most systems because of bidirectional communication links which allow nodes to simultaneously start the pairwise exchange operation. In the **echo** kernel, the communicating node has to wait before sending a message back to the test node. This kernel assumes buffered message passing semantics for the simultaneous sends between the communicating nodes, in systems that do not support such semantics, there is a potential danger of deadlocks.

```

begin
  for (message size = 0 to message size = over a page size)
    begin
      t1 = get time;
      if (test node) then
        broadcast message to all nodes;
      else
        begin
          receive message broadcasted;
        end if
      barrier();end if
      t2 = get time;
      broadcast time on test node = t2 - t1;
    end for
  end

```

Figure 3.4: Pseudo-code for **broadcast**

3.3.3 Broadcast

The broadcast benchmark (Figure 3.4) measures the time taken to broadcast a message to all other nodes. This kernel can be implemented using a series of sends. However, broadcast is a commonly used communication primitive and so efficiently coded library routines are usually available. Hence, many systems implement it using an efficient library routine based on optimal communication algorithms. As with the previous kernels, broadcast is constructed for varying message sizes.

3.3.4 Global Reduction

The **global reduction** benchmark is an extension of the broadcast benchmark in the sense that it requires each node to send a value to all other nodes. Our global reduction benchmark requires each node to calculate the global sum of the individual elements of a double precision vector stored on each node. The kernel is constructed for varying vector lengths.

```

begin
  double precision : vector;

  for (vector length = 0 to vector length = max_vec_length)
    begin
      t1 = get time;
      number := global sum of number on all nodes;
      t2 = get time;
      global sum time on test node = t2 - t1;
    end
  end

```

Figure 3.5: Pseudo-code for **global reduction**

3.3.5 Global Synchronization

Finally, the **global synchronization** kernel measures the minimum time required to complete a global barrier synchronization. Global synchronization can be a very expensive operation because all the processors must communicate with each other. This is also a difficult value to measure because of the potential for the nodes to be at completely different stages of execution before synchronization. Hence, our **global synchronization** kernel is constructed in a manner that effectively measures the *least* time required to complete a barrier synchronization. This is accomplished by repeatedly calling and timing global synchronizations on a single node.

While **echo**, **broadcast**, **pairwise exchange** and **global reduction** benchmarks lead to graphs of message size versus time to execute the kernel, the broadcast, **global reduction** and **global synchronization** benchmarks produce graphs of number of processors versus time.

3.3.6 Contention

All the benchmarks mentioned above have been measured on previously idle systems. To study the effects of contention, rather than rely on random load, we have introduced

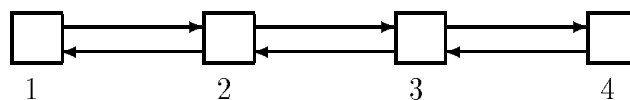


Figure 3.6: A simplified view of the logical topology for the contention kernel. The arrows in the figure denote communication links and the boxes denote nodes. Nodes 2 and 3 run the **echo** kernel while nodes 1 and 4 provide cross traffic.

measured loads into the system while running the **echo** kernel. Our contention benchmark chooses 4 nodes such that two nodes continuously exchanging messages introduce load on the communication links between the other two nodes running the **echo** kernel - see Figure 3.6. The identity of the nodes 1, 2, 3 and 4 in the figure is dependent on the architecture. For example, we have chosen node '0' (relates to 1 in the figure) to continuously send messages to node '7' (relates to 4) on a 8-node iPSC/860 hypercube. The messages from these nodes pass through nodes '1' (relates to 2 in the figure) and '3' (relates to 3). We then calculate the communication characteristics of nodes '1' and '3' by simultaneously executing the **echo** kernel. The degree of load introduced can be easily varied by transferring messages of different sizes. In general, since this kernel is architecture dependent, the nodes to be identified for running the benchmark is based on the hardware topology of the target system.

3.3.7 Overlap of communication with computation

Usually, computation not dependent on the results of an initiated message transfer can take place simultaneously with synchronous communication. Hence, some computation can always be overlapped with communication using asynchronous message passing primitives. To examine the degree to which this overlap can take place, the overlap benchmark first runs the pairwise exchange kernel with a DAXPY computational part using synchronous communication, then repeats the pairwise exchange kernel, overlapping asynchronous communication with the DAXPY computational part. The amount of computation is varied by changing the lengths of the vectors for the DAXPY and

```

begin
  if (node == 1 or node == 4) then
    forever loop
      if (node == 1) then
        send message to fourth node;
      end if
      if (node == 4) then
        receive message from first node;
      end if
    end for
  else
    if (node == 2 or node == 3) then
      t1 = get time;
      if (node == 2) then
        send message to third node;
        receive message from third node;
      else
        if (node == 3)
          receive message from second node;
          send message back to second node;
        end if
      end if
      t2 = get time;
      echo time with contention (on node 2) = t2 - t1;
    end if
  end if
end

```

Figure 3.7: Pseudo-code for **echo with contention**

```

begin (synchronous version)
   $t1 = \text{get time};$ 
  if (test node) then
    send message to other node;
    receive message from other node;
  else
    if (communicating node) then
      send message to test node;
      receive message from test node;
    end if
  endif
  perform DAXPY computation (vector length);
   $t2 = \text{get time};$ 
  time without overlap on test node =  $t2 - t1$ ;
end (synchronous version)

```

Figure 3.8: Pseudo-code for synchronous portion of the **overlap** kernel

the amount of communication is varied by changing the message sizes. The kernel is constructed for different message sizes and vector lengths to study the extent of overlap at varying levels of communication and computation (Figures 3.8 and 3.9).

3.3.8 Update guard

The function **update_guard()** takes a pointer to the local matrix and the dimension of the local matrix as parameters and updates the non-local elements in the guard wrapper by communicating with neighboring nodes (figure 3.11). Identities of the neighboring nodes are specified in the global values **pred**, **succ**, **top** and **bottom** for every node by dereferencing the adjacency matrix. The northern portion of the guard wrapper thus contains values from the **top** node, the southern portion of the wrapper contains values from the node **bottom**, and the eastern and western parts of the wrapper contain values from the nodes identified by **pred** and **succ** respectively - see Figure 3.10.

```

begin (asynchronous version)
   $t1 = \text{get time};$ 
  if (test node) then
    post a send message to other node;
    post a receive message from other node;
  else
    if (communicating node) then
      post a send message to test node;
      post a receive message from test node;
    end if
  endif
  perform DAXPY computation;
  wait for sends and receives to complete;
   $t2 = \text{get time};$ 
  time with overlap on test node =  $t2 - t1$ ;
end (asynchronous version)

```

Figure 3.9: Pseudo-code for asynchronous portion of the **overlap** kernel

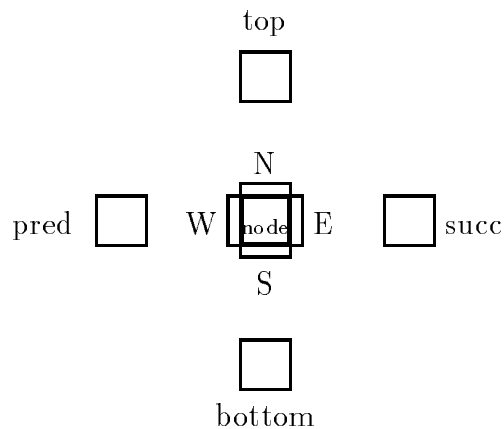


Figure 3.10: View of neighboring nodes with respect to a node and its guard wrapper

```

begin
     $t1 = \text{get time};$ 
    read northern guard wrapper values into a vector;
    send vector to top node;
    read southern guard wrapper values into a vector;
    send vector to bottom node;
    read eastern guard wrapper values into a vector;
    send vector to succ node;
    read western guard wrapper values into a vector;
    send to pred node;
    receive vectors from top, bottom, pred and succ nodes;
    update southern, northern, eastern and western guard wrapper values;
     $t2 = \text{get time};$ 
    update time =  $t2 - t1$ ;
end

```

Figure 3.11: Pseudo-code for `update_guard(local matrix, local dimension)`

3.3.9 Shift Matrix

Shifting a distributed matrix by N/P elements, where N/P is the number of matrix elements divided by the number of processors in the direction being shifted, is a commonly used matrix operation, e.g. it is used in matrix multiplication. The `shift_matrix()` benchmark (Figure 3.12) shifts the matrix by N/P elements. The local matrix with the guard wrapper has been implemented as an array in our benchmark. To keep matters simple and to avoid having to build too many buffers, the benchmark kernel shifts the whole local matrix along with the guard wrapper. In addition to taking a pointer to the local matrix and the local matrix dimension as parameters, this function call also has the direction to be shifted as an additional parameter. The enumerated variable `NORTH` is used as the direction to be shifted if the matrix has to be shifted upward and the enumerated variable `EAST` is used if the matrix has to be shifted sideward. All the nodes receive the shifted part of the matrix into their original data structure referred to by `local_matrix`.

```

begin
   $t1 = \text{get time};$ 
  if (direction is NORTH) then
    send local matrix to top node;
  else
    if (direction is EAST) then
      send local matrix to succ node;
    end if
  end if
  receive matrix into local buffers;
   $t2 = \text{get time};$ 
   $\text{shift matrix time} = t2 - t1;$ 
end

```

Figure 3.12: Pseudo-code for **shift_matrix**(*local matrix*, *local dimension*, *direction*)

3.3.10 Transpose Matrix

Matrix transpose is included as a benchmark kernel because it is an important application in its own right and yields a completely different communication pattern from the shift kernel discussed above - see figure 3.13. The kernel determines the node with which to exchange the local matrix by dereferencing the adjacency matrix. The nodes then exchange the local matrices and perform an internal transpose of the local matrix to complete the operation. Transpose can also be implemented using the divide and conquer approach. Interestingly, the divide and conquer approach leads to a completely different pattern and this approach could be implemented in a future extension of the benchmark.

3.3.11 Row and Column Broadcast

Finally, many matrix manipulations require the broadcast of just row or column of the matrix to the other processors. LU decomposition is an example of this category of application. This benchmark kernel - see figure 3.14 - consists of two sub-kernels **row_broadcast**() and **col_broadcast**() which take as additional inputs the row or column (in global units) to be broadcasted. If the node possesses part of the specified row/column, it broadcasts that part to all other nodes possessing parts of the

```

begin
    t1 = get time;
    identify transpose node;
    send local matrix to transpose node;
    receive matrix into local buffers;
    transpose local matrix;
    t2 = get time;
    transpose time = t2 - t1;
end

```

Figure 3.13: Pseudo-code for **transpose_matrix**(*local matrix*, *local dimension*)

matrix related to the same column/row. Otherwise, the node waits to receive part of the row/column. All the nodes update the values in their respective local matrices based on the broadcast values.

3.4 Support Functions/Macros

This section outlines the definitions of the different macros used in the benchmark together with some motivation for their usage.

3.4.1 Initialization and Cleanup Phases

Some systems require explicit setup and cleanup phases. An example is PVM[Sun90] where UNIX processes have to be enrolled as PVM processes and have to set up the message passing system during an initialization phase. PVM programs also have to call a clean-up routine before exiting. Hence, to be broadly applicable, we divide CoMet into three phases - the initialization phase, the benchmark kernels' phase and the cleanup phase. During initialization, the function **initialise**() performs the required initializing of the message passing systems. The function **get_node_id**() returns a unique node number starting at '0' for all the nodes in the system. The function **get_num_nodes**() returns the number of nodes the benchmark program has been loaded on.

The matrix manipulation operations operate on a 2-D matrix, block decomposed over

```

begin
   $t1 = \text{get time};$ 
  if (node possesses part of the row) then
    identify nodes to which the part of the row has to be broadcast;
    loop for all identified nodes
      send local part of the row to identified node;
    end loop
  else
    receive broadcast row;
  end if
   $t2 = \text{get time};$ 
  time to broadcast row =  $t2 - t1$ ;
end

```

Figure 3.14: Pseudo-code for **row_broadcast**(*local matrix*, *local dimension*, *row*)

all the acquired nodes of the parallel system. To decompose the matrix, the processors are mapped onto a two dimensional mesh. The dimensions of the mesh are returned as global variables **proc_dim1** and **proc_dim2** by the function **get_proc_dimensions()**. Each processor in the system is represented by a point in the mesh. Since most communication in matrix manipulations takes place between neighbors, to minimize communication overhead any two adjacent points(processors) in the mesh have to be physically as close as possible to each other. The function **get_adjacent_neighbors()** forms the processor mesh with dimensions **proc_dim1** and **proc_dim2** and returns the node numbers of its nearest neighbors as global variables - **top**, **bottom**, **pred** and **succ**. The function also identifies the row **node_row** and column **node_col** at the node in the mesh. Finally, to aid kernels such as the **echo** kernel that require the identity of nodes several hops away in terms of communication links, the function **get_hop_node()** returns the identity of a node any number of communication links away. The number of links is specified as an input parameter to the **get_hop_node** function. The **clean_up()** function calls any required system primitives before exiting.

3.4.2 Message Passing

Programs written for distributed memory systems usually adhere to a message passing paradigm for communication. The macros described below represent both asynchronous³ and locally synchronous⁴ sending and receiving of messages. The user has to specify the address and length of a buffer in the application's address space. Messages sent are identified by type and the node the message is intended for. Messages received are matched only by the type specified to the macro. The semantics of sending are thus limited to the nodes specified to the macro.

The locally synchronous communication macros used in the benchmark are **blocked_send**, **blocked_recv** and **blocked_broadcast**. The macros for asynchronous communication implement the non-blocking versions of the **blocked_send** and **blocked_recv** to allow for overlap of computation and communication. The syntax of the non-blocking communication macros **unblocked_send** and **unblocked_recv** are the same as the blocking ones except that the non-blocking macros return an integer which uniquely identifies the instance of the non-blocking call.

The **blocked_send** macro is used to send a buffer containing the message labeled as type across to another node specified as a parameter. This call blocks until the application's buffers are copied to the OS's communication buffers. The application buffer can be reused as soon as the call returns.

blocked_send(*type*, *message*, *message_length*, *node*)

type = identity label for message

message = buffer containing the message

message_length = length of the buffer containing the message in bytes

³In the asynchronous model[FJG⁺88][For], sends and receives are non-blocking and return immediately. However, the sending process cannot reuse the message buffer until the message is transferred. Similarly, the receiving process cannot use the contents of the message buffer until the message is completely transferred. Systems can monitor the completion of asynchronous sends/receives by either waiting for the message transfer to complete or by probing the status of the message transfer.

⁴In the locally synchronous model[For], a send blocks until the application buffer is copied into system space and is available for reuse. Similarly, a receive blocks until the contents of the message are completely copied into the process's application buffer.

node = destination processor/node number

The **blocked_recv** macro is blocked until a message that matches the type specified as a parameter is copied into the receive buffer in the application's address space.

blocked_recv(*type*, *message*, *message_length*)

type = identity label for message

message = buffer containing the message

message_length = length of the buffer containing the message in bytes

The **blocked_broadcast** macro is used to send a message labeled by type to all other nodes in the system. This call also blocks until the application-level buffers are copied to the communication system buffers.

blocked_broadcast(*type*, *message*, *message_length*)

type = identity label for message

message = buffer containing the message

message_length = length of the buffer containing the message in bytes

Probing

In some types of message-passing programs, the type of the next message to be read may not be strictly predictable. In such cases, probing is useful to determine if any messages of a particular type have arrived. The macro **unblocked_probe_msg(type)** returns a '1' if a message of the specified type has arrived and a '0' if there is no such message has arrived.

unblocked_probe_msg(*type*)

type = identity label for message

Wait For

In our model of asynchronous communication, before reclaiming the buffer (after calling **unblocked_send**) and before using the contents of the message buffer (after posting

an **unblocked_recv**), the execution of the non-blocking calls has to be complete. The **wait_for_send_to_complete** and **wait_for_recv_to_complete** macros serve to block further execution until the non-blocking calls complete.

wait_for_send_to_complete(*send_id*)

send_id = integer returned by **unblocked_send** macro

wait_for_recv_to_complete(*recv_id*)

recv_id = integer returned by **unblocked_recv** macro

Reduce

To support the global reduction benchmark kernel, the **global_sum** macro returns the sum of a double precision variable over all the processors.

sum = **global_sum**(*local_variable*)

local_variable = local double precision variable

sum = sum of the local double precision variable on all processors

Barrier

To set the entire system to a well-defined state, the executing node programs must be synchronized. The macro **barrier** when executed on all nodes blocks further execution until all other nodes have reached the **barrier**. The **barrier** macro is executed at the start of all the benchmark kernels to bring the nodes to the same point in execution.

3.4.3 Other related macros

A benchmark program measures elapsed times for different code sections. To measure the elapsed times, the macro **get_time** returns double precision time in seconds.

To measure startup and other times less than a few microseconds, the benchmark places repeat loops around the different sections of code that are to be timed. The

increment value for the repeat loops is specified by the macro variable **max_times**. Also, most of the benchmark kernels operate on varying message sizes. It is interesting to observe the kernel behavior at small message sizes to have an idea of the system overhead, and at buffer lengths slightly exceeding the processor's page sizes to have an idea of the effect of memory mapping between the user space and kernel space, etc. The benchmark's message sizes are therefore incremented by small increment values and the maximum message length is dependent on the page size specified by the macro **page_size**. For the overlap kernel, the computational part is based on different vector sizes as well as different message lengths. The maximum vector length is specified by the macro **max_vec_len**.

For error detection, we assume that error is flagged to the program by returning a value smaller than zero and that error-handling is left to the user. The function **error_exit**, prints out the input error message, identifying the node where it occurred, and halts the execution of the programs running on all the other nodes of the system.

3.5 Porting of CoMet

We have organized CoMet into 5 modules : main.c consisting of all the kernels; init.c consisting of the initialization routines, the setting up of the adjacency matrix, etc.; setup.c consisting of routines to set up the matrix for the matrix-related kernels; mat_ops.c consisting of functions implementing the matrix-related operations; check.c to verify the resultant matrix after the manipulations on the matrix are completed and finally repp.c consisting of functions to report all values derived while executing the benchmark. Macro definitions and other function prototype declarations are included in prototypes.h and extern.h. Users are not allowed to modify the modules setup.c and check.c, although they can implement the matrix-related operations in any way.

Chapter 4

iPSC/860 Implementation

4.1 Overview of iPSC/860 architecture

The iPSC/860 consists of a number of Intel's i860 processors interconnected with a hypercube communication network having a hypercube topology. The number of processors in any hypercube configuration can range from 1 onwards in powers of 2. Each node in the hypercube is connected to the hypercube communication network via a direct connect module(DCM). The DCMs support 8 channels to connect to direct neighbor nodes and these channels are bit-serial and bidirectional with a bandwidth of 2.8 Mbytes per second. Since one channel in the DCM is reserved for special I/O, the maximum size of the iPSC/860 is 128. Communication requests between any two nodes are fulfilled dynamically by building a path along free channels using an e-cube algorithm [Nug88]. The paths are freed once the communication request has been fulfilled.

The nodes in an iPSC/860 are built from RISC-based *i860^{XR}* processors which have a clock cycle of 25ns, communication-related hardware and 8-16MB of DRAM with 70ns access time. The i860 has 32 registers, and like typical RISC processors, completes most instructions in one-cycle. It has a 4-stage pipeline consisting of fetch, decode, execute and write phases. The floating point unit has 32 registers which are also accessible as 16 64-bit registers. The processor has a 4KB two-way set-associative instruction cache and a 8KB two-way set-associative write-back data cache. The MMU has a 64-entry, four-way set-associative TLB that implements a paged virtual memory of 4GB with a 4KB pagesize.

The i860 processor has the capability of operating in a dual-instruction mode that can simultaneously execute an instruction in both the core as well as floating point unit. Peak performance rates for the processor are 80 MFLOPS in 32-bit precision and 60 MFLOPS in 64-bit precision. The overall peak performance of a maximal iPSC/860 hypercube with 128 nodes is 10.24 GFLOPS in 32-bit precision and 7.68 GFLOPS in 64-bit precision. Actual performance achieved with present-day compilers is less than one-third of the peak performance.

Each node on the iPSC/860 hypercube runs the NX/2 node operating system. NX/2 [Pie88] [Int91] performs process management and message passing and allows only one application to execute per node at a time on each node. The 8th DCM channel can be used to access a concurrent file system(CFS) which is a collection of additional Intel 386-based I/O nodes. The hypercube can be accessed from an attached host, called the System Resource Manager(SRM), which is an Intel 301 microcomputer with an 80386/387-based processor-pair running at 16 MHz and having upto 8 MB of memory. The SRM plays the role of host-server to the hypercube, aiding in program development, allocation and de-allocation of sub-cubes, and in loading user programs onto the subcubes.

4.2 Communication Protocols in NX/2

The communication protocol used by NX/2 is based on buffers. The system buffers all incoming and outgoing messages in such a way that no rendezvous is necessary for any two communicating processes. Each NX/2 instance has a number of 100-byte buffers reserved for all other nodes that can communicate with that node. Each NX/2 also remembers the number of free buffers on all other nodes that have buffers reserved for it. Applications identify the memory area containing the message to be communicated by passing a pointer to it as a parameter to the message passing primitive. In the blocking version of the send primitive, the application is then blocked from further execution until the message has been successfully copied into the system buffers.

For messages smaller than or equal to 100 bytes, NX/2 on the sending node determines if there is a free buffer reserved for it on the receiving node. If a free buffer exists, NX/2 decrements its count of free buffers available for it on the receiving node and sends the data. If there are no available buffers, the executing application is blocked until a buffer becomes available. The system piggybacks buffer status over network traffic.

For messages that are longer than 100 bytes, the allocation by NX/2 is done dynamically in a circuit-switched manner. A short message is sent requesting that a buffer, equal in size to the length of the message, is allocated. The sending node waits for an acknowledgement before sending the data.

Three different semantics of sending and receiving of data are provided :

1. blocking send/receive (locally synchronous),
2. non-blocking send/receive (asynchronous) and
3. interrupt-driven send/receive (also asynchronous).

Using the blocking primitives, the sender is blocked until the message has been copied into the system buffers and the receiver is blocked until the message data has been copied into the receive buffer in its address space. The C version of the calls are

```
csend(message_type, buffer, buffer_size, id_of_receive_node, process_id),  
crecv(message_type, buffer,buffer_size).
```

At present NX/2, supports only a single process executing at a time on each node. The parameter **process_id** is therefore unused and is to be specified as an integer '0' in the calls.

For the non-blocking primitives, the sender and receiver initiate the communication with a call and return immediately with an identifier for the message that is being sent/received. Before re-using the contents of either its send or receive buffers, the application must check for completion of the non-blocking communication operations. This behavior is accomplished through the following calls :

```

message_id = isend(message_type, buffer, buffer_size, id_of_receive_node,
                    process_id),
message_id = irecv(message_type, buffer, buffer_size),
msgwait(message_id)

```

The last type of communication semantics is implemented using

```

hsend(message_type, buffer, buffer_size, id_of_receive_node, process_id,
        handler),
hrecv(message_type, buffer, buffer_size, handler).

```

These calls return as soon as possible as do `isend` and `irecv`, however instead of requiring the application to explicitly check for completion, `NX/2` directly invokes a handler specified with the call.

The communication macros in the CoMet kernels for the iPSC/860 are mapped onto `csend`, `crecv`, `isend` and `irecv` to implement locally synchronous and asynchronous message-passing.

In addition to the basic message passing primitives, the programmer is also provided with system calls to search for specific messages identified by their message types. Again, these calls are implemented in both blocking and non-blocking versions. In the non-blocking form, the call returns a flag to indicate the result of the search for the message of a specific type.

blocking form :

```
cprobe(message_type)
```

unblocking form :

```
flag = iprobe(message_type)
```

4.3 Porting the benchmark

The benchmark was ported to an Intel 40MHz iPSC/860 with 16 nodes, each with 8MB of memory and running NX/2 version 3.2. The C compiler **icc**, was version 2 from Portland Group, Inc (PGI). The compiler options used for compiling the benchmarks were -O2 and the tests were carried out between October 1992 and December 1992.

To port the benchmark to the iPSC/860, we

1. mapped all the macro calls to iPSC/860 message passing primitives,
2. set all the macro variables,
3. mapped hardware hypercube topology onto a two-dimensional grid,
4. wrote the functions to implement the matrix manipulation routines, and
5. wrote a shell script to extract and organize all relevant output data into a form acceptable to gnuplot - a graph plotting tool.

The iPSC/860 does not require any special initialization on starting CoMet other than loading the various kernels on the nodes. Neither does it require any clean up while exiting. Only one process is initiated on each node and since the nodes are numbered contiguously starting with '0', by the operating system itself, porting of CoMet to the iPSC/860 was relatively straightforward. CoMet's **get_node_id()** and **get_num_nodes()** were mapped onto NX/2 primitives **mynode()** and **numnodes()** respectively.

Both the benchmark macros and iPSC/860 message passing primitives are based on buffered communication making it straightforward to map the macros to the message passing calls. Macros **blocked_send()** and **blocked_recv()** were mapped onto **csend()** and **crecv()**, and macros **unblocked_send()** and **unblocked_recv()** were mapped onto **isend()** and **irecv()** respectively. To implement **blocked_broadcast**, the macro was

mapped onto **csend()** with the input parameter, identifying the node the message is intended for, set to '-1'. The **wait_for_send_to_complete()** and **wait_for_recv_to_complete()** macros were both mapped onto the system call **msgwait()**. The macro implementing the search for specific messages **unblocked_probe_msg()** was mapped onto **iprobe()** and the macro **barrier()** was mapped onto **gsynch()**.

To measure elapsed times for the various kernels in the benchmark, the macro **get_time()** was mapped onto the system call **dclock()** which returns double precision time in seconds since the booting of the hypercube. The elapsed times for the benchmark were found to be converging when the kernels were averaged on about 1000 runs. Hence, to obtain reliable measurements of the elapsed time for the different sections of code, the macro **max_times** was set to '1000'. The macro **page_size** was set to the natural page size of the i860 microprocessor, that is '4096'.

To measure the overlap of computation and communication, the macro specifying maximum vector length for the DAXPY part, **max_vec_len**, was set, somewhat arbitrarily, to '200000' based on the memory available in the system and to observe significant overlaps of communication with computation. The message lengths in the communication part of the kernel ranged from 0 to about 85000, to observe the behavior of small to large messages in the benchmark kernels.

Table 4.1 lists the mapping of the benchmark macros onto iPSC/860 message passing calls or the values they are set to.

To map the hypercube topology onto a two-dimensional mesh, gray code [FJG⁺88] has been used - refer Figure 4.1. The elements in the mesh represent nodes in the system such that adjacent elements in the matrix are physically neighbors as well. The dimensions of the mesh are determined by variables **proc_dim1** and **proc_dim2**. If the number of nodes allocated to the application running the benchmark is a perfect square, the two dimensions are equal to the square root of the number of nodes allocated, otherwise **proc_dim1** is twice **proc_dim2** and is equal to the square root of twice the number of processors allocated to the running benchmark. The function **get_hop_node()** which returns the identity of nodes 'n' hops away, is defined using bit arithmetic properly

<code>page_size</code>	4096
<code>max_times</code>	1000
<code>max_vec_len</code>	200000
<code>get_time()</code>	<code>dclock()</code>
<code>blocked_broadcast(msgtyp, msg, msg_size)</code>	<code>csend(msgtyp, msg, msg_size, -1, 0)</code>
<code>blocked_send(msgtyp, msg, msg_size, node)</code>	<code>csend(msgtyp, msg, msg_size, node, 0)</code>
<code>blocked_recv(msgtyp, msg, msg_size)</code>	<code>crecv(msgtyp, msg, msg_size)</code>
<code>unblocked_probe_msg(msgtyp)</code>	<code>iprobe(msgtyp)</code>
<code>barrier()</code>	<code>gsync()</code>
<code>global_sum(num, dummy)</code>	<code>gdsum(&num, 1, &dummy)</code>
<code>unblocked_send(msgtyp, msg, msg_size, node)</code>	<code>isend(msgtyp, msg, msg_size, node, 0)</code>
<code>unblocked_recv(msgtyp, msg, msg_size)</code>	<code>irecv(msgtyp, msg, msg_size)</code>
<code>wait_for_send_to_complete(id)</code>	<code>msgwait(id)</code>
<code>wait_for_recv_to_complete(id)</code>	<code>msgwait(id)</code>

Table 4.1: Mapping of macros

measure characteristics of the hypercube configuration - Figure 4.2.

Finally, to implement the matrix manipulation routines, the functions **update_guard**, **shift_matrix**, **transpose**, **row_broadcast** and **col_broadcast** were completely written using a combination of **csend**, **crecv**, **isend** and **irecv**. For **update_guard**, the northern, southern, eastern and western values were assembled into separate buffers, and the four assembled buffers were sent to the neighboring nodes **top**, **bottom**, **pred** and **succ** respectively. To update its own guard wrappers, each node receives messages from the four neighboring nodes and copies them into its respective guard wrappers. Instead of naively implementing the function without any overlap of computation with communication, we have overlapped the assembling of buffers with communication - see Figure 4.3.

For **shift_matrix**, the entire local matrix was sent to a neighboring node using **csend** and the shifted matrix was received into the same buffer using **crecv**. The **transpose** function was implemented in a straightforward way with each node exchanging its buffer with another node. The identity of the node to exchange messages with was obtained

```

for (i=0; i<proc_dim2; i++) {
    for (j=0; j<proc_dim1; j++) {
        *(adj_matrix+i*proc_dim1+k) = gray(i*proc_dim1+j);
        k += value;
    }
    value *= -1;
    if (value < 0)
        k = proc_dim1-1;
    else
        k = 0;
}
value = 1;
for (i=0; i<proc_dim2; i++) {
    for (j=0; j<proc_dim1; j++) {
        if (*(adj_matrix+ i*proc_dim1+j) == node_id) {
            value = 0;
            break;
        }
    }
    if (value == 0)
        break;
}
node_row = i;
node_col = j;
pred = *(adj_matrix+i*proc_dim1+((j+proc_dim1-1) % proc_dim1));
succ = *(adj_matrix+i*proc_dim1+((j+1) % proc_dim1));
top = *(adj_matrix+((i+proc_dim2-1) % proc_dim2)*proc_dim1+j);
bottom = *(adj_matrix+((i+1) % proc_dim2)*proc_dim1+j);

```

Figure 4.1: C code fragment for mapping the hypercube topology to a mesh

```

for (i=0; i<hop; i++) {
    if (node & temp)
        node = -= temp;
    else
        node += temp;
    temp = 2*temp;
}
return node;

```

Figure 4.2: C Code fragment returning the identity of a node 'hop' hops away

```

begin
    assemble north guard wrapper into buffer;
    initiate send to north neighbor top;
    assemble south guard wrapper into buffer;
    initiate send to south neighbor bottom;
    assemble east guard wrapper into buffer;
    initiate send to east neighbor succ;
    assemble west guard wrapper into buffer;
    initiate send to west guard wrapper pred;
    initiate receive from bottom to fill north guard wrapper;
    initiate receive from top to fill south guard wrapper;
    initiate receive from pred to fill west guard wrapper;
    initiate receive from succ to fill east guard wrapper;
    wait for all sends and receives to complete;
end

```

Figure 4.3: Pseudo code for **update_guard** on iPSC/860

```

row = global_row/local_dim;
local_row = global_row%local_dim;
ptr = local_matrix+(local_dim+2)*(++local_row)+1;
if (node_row == row) {
    for(i=0; i<proc_dim2; i++) {
        if (i != node_row)
            csend(ROW_CAST, ptr, local_dim*sizeof(double),
                (i*proc_dim1+node_col), 0);
    }
    for (i=1; i<proc_dim2; i++)
        crecv(GOTIT, null, 0);
}
else {
    crecv(ROW_CAST, ptr, local_dim*sizeof(double));
    csend(GOTIT, null, 0, infonode(), 0);
}

```

Figure 4.4: C code fragment for **row_broadcast**

by dereferencing the processor mesh with the **node_row** and **node_col** transposed. Each node, after receiving the local matrix from another node performed an internal transpose to complete the operation. Finally, the **row_broadcast** and **col_broadcast** were implemented using **csend** and **crecv** to broadcast to all the nodes not possessing the said row or column of the matrix - see Figure 4.4.

Chapter 5

Results and Analysis

CoMet was implemented and executed on 2, 4, 8 and 16 nodes of the iPSC/860. The results obtained from running the benchmark are presented in the form of graphs with a brief analysis for each of them. We have not attempted to reduce the results to a single number. Instead, we have tried to derive values that reflect the basic characteristics of the machine such as the start-up time and bandwidth for the **echo** and **pairwise exchange** kernels.

5.1 Basic communication kernels

The fundamental characteristics associated with message passing hypercubes are the start-up time and time taken per byte of data. We derived these characteristics by running the **echo** kernel and plotting the elapsed times for different message sizes in a graph. If $t_{start-up}$ is the message latency for a 0-byte message to be echoed and $t_{per-byte}$ is the transfer time for one byte, since the elapsed time increases linearly (except for the jumps which will be explained later) with message size, the time taken $t(N)$ for a message of N bytes to be echoed from a neighboring node that is a hop away, empirically is

$$t(N) = t_{start-up} + Nt_{per-byte}.$$

For messages to be echoed between nodes that are more than a hop away, say n communication links away, there is an additional overhead proportional to the number of hops

as seen in figure 5.1. The time taken $t(N)$ for a message of N bytes to be echoed from a node n hops away thus becomes

$$t(N) = t_{start-up} + Nt_{per-byte} + (n - 1)h$$

where h is the incurred overhead per hop and n is the number of hops. Bomans and Roose [BDH90], Dunigan [Dun91][Dun92], Berrendorf and Helin [BH92] have all used the above empirical formulae in interpreting the characteristics of communication on the iPSC/860. From the graph, we observe jumps at around 100, 2048 and 4096 bytes. The first jump can easily be explained, the operating system uses different protocols for messages of size smaller than or equal to 100 bytes and for those greater than 100 bytes. Messages of size 4096 bytes are guaranteed to lie across page boundaries, but it is interesting to note that message transfer starts getting faster at around 2048 and 4096 bytes and we have not been able to attribute a reason for the faster rates at these message sizes. Ignoring the jumps, using a linear least-squares fit to our measured communication times, we arrive at $t_{start-up}$ to be around 79us for transmitting messages less than 100 bytes and 156us for messages greater than 100 bytes. We arrive at $t_{per-byte}$ to be around 0.63us for messages less than 100 bytes and 0.41us for messages greater than 100 bytes. and the overhead h to be 25us for messages greater than 100 bytes. Adding hops seems to add a small overhead to the total transmission time. Hence, our conclusion is that wire time by itself does not account for most of the transmission time.

To have an idea of bandwidth (bytes transferred per unit time), we also plotted the graph (Figure 5.2) with data rates (basing the data rate calculation on elapsed time) against message size. The iPSC/860 had a peak bandwidth of 2.5 MB/s for the echo kernel between adjacent nodes. Also, half the peak bandwidth was achieved around message sizes of 486 bytes and ninety percent of the bandwidth was achieved around 5000 bytes. We use the message sizes that achieve 50% and 90% the peak bandwidth, characterized as 50% and 90% loads, as input to the contention kernel.

For the broadcast kernel, we plotted the elapsed times for different machine sizes with respect to the message size - see Figure 5.3. Predictably, we observed the jump

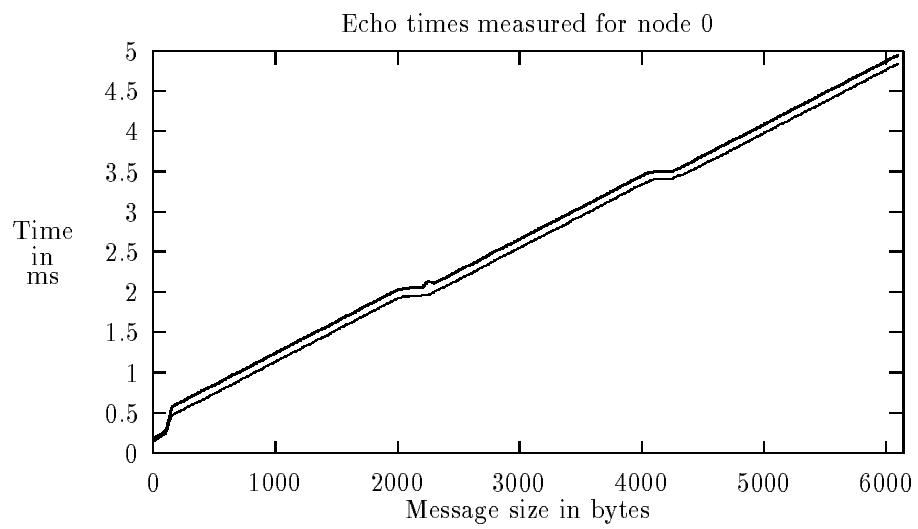


Figure 5.1: Echo kernel run on 16 nodes. The lower line represents the echo times between adjacent nodes and the upper line represents the echo times between nodes 4 hops apart. The time for a single send/rcv is obtained by halving these numbers.

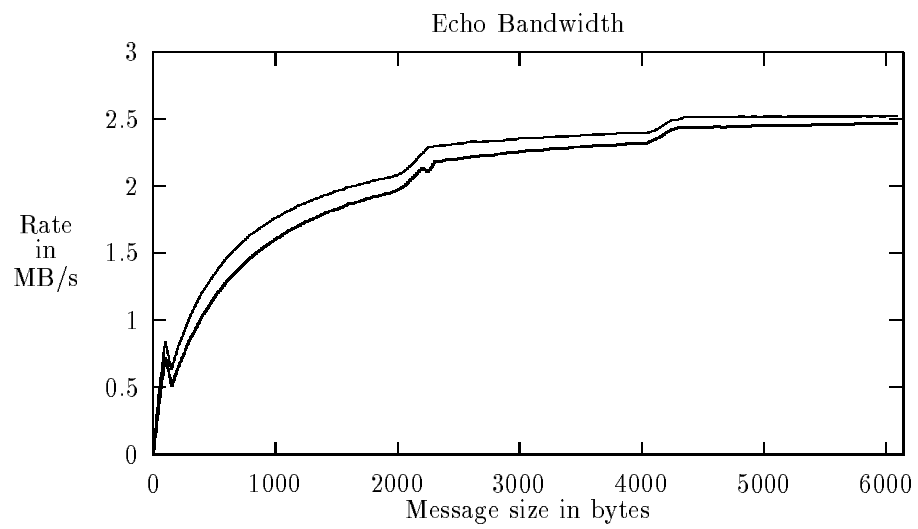


Figure 5.2: Echo kernel run on 16 nodes. The upper line represents the bandwidth achieved between adjacent nodes and the lower line represents the bandwidth achieved between nodes 4 hops apart.

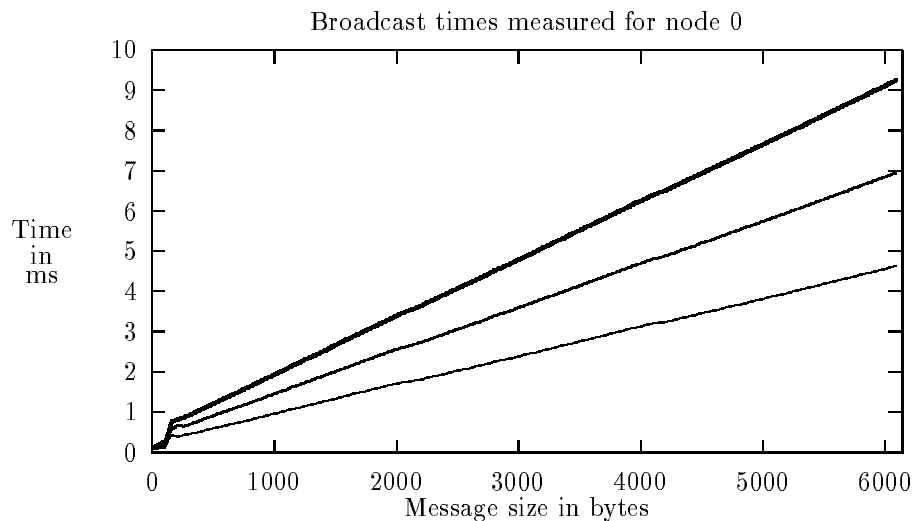


Figure 5.3: Broadcast kernel. The lowest line represents the broadcast times when the kernel was run on 4 nodes, the middle line represents the elapsed times on 8 nodes and the upper line represents the times on 16 nodes.

at a message size of 100 bytes for the change of protocol. Otherwise, the time taken to broadcast messages to all other nodes increased in a linear fashion with respect to message size for each machine size. As expected, the broadcast times for the different machine sizes increased linearly with the log number of processors, as seen in the constant slope changes in the figure for 4, 8 and 16 node combinations.

To interpret the results obtained by running the pairwise exchange kernel, we plotted the elapsed times for the kernel running on messages of different sizes against message size - see Figure 5.4. The curve obtained can again be empirically interpreted by the equation

$$t(N) = t_{start-up} + Nt_{per-byte} + (n - 1)h.$$

Since the communication channels in the iPSC/860 are bidirectional, we observed the pairwise exchange between nodes to be faster than the results obtained by running the

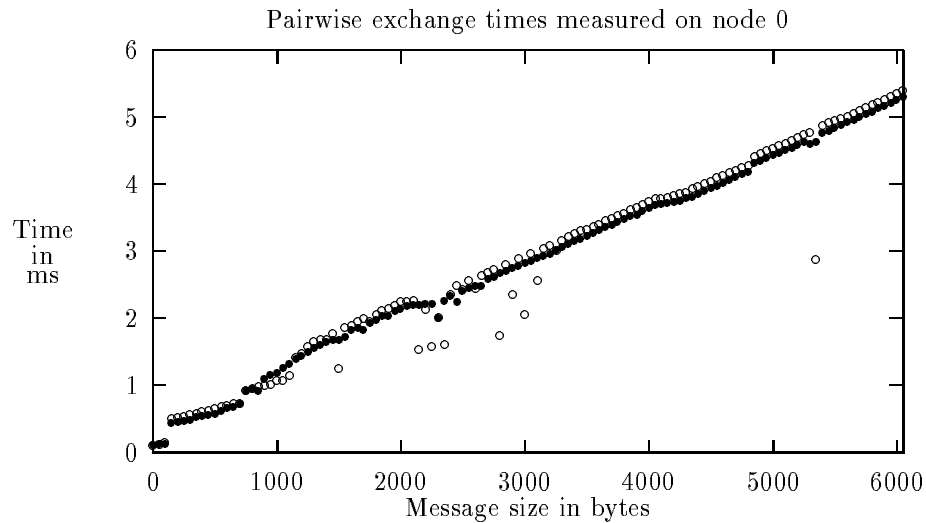


Figure 5.4: Pairwise exchange kernel run on 16 nodes. The filled circles represent elapsed times for a pairwise exchange between adjacent nodes and the empty circles represent elapsed times between 4-hop nodes.

echo kernel for the same message size. From the graph, we also observe some points where the pairwise exchange completes a lot faster than expected. We are not sure of the exact sequence of events leading to the faster communication, but it could be because the exchanging nodes were synchronized in such a way that the receive had already been posted before the message arrived.

All the results discussed so far were in contentionless environments. We have plotted the elapsed times for the echo test between nodes 1 and 3 under 50% and 90% loads between node 0 and 7 - see Figure 5.5. Since, the iPSC/860 uses circuit switching to rout its messages greater than 100 bytes, the effect of contention depends on whether the communicating nodes 0 and 7 established circuits spanning nodes 1 and 3 and can vary from iteration to iteration. Predictably though, we observe from the graph that the echo test runs slower under increased loads between nodes 0 and 7.

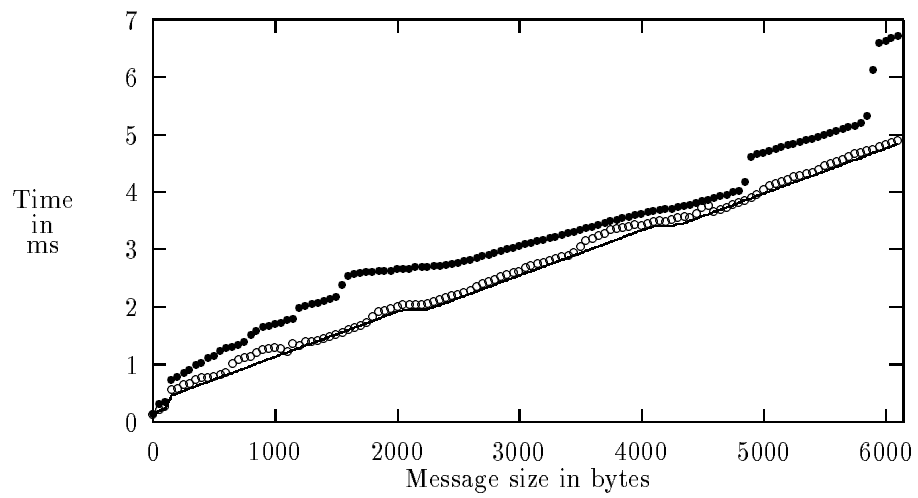


Figure 5.5: Contention kernel run on 8 nodes. The empty circles represent echo times under contention from 50% load and the filled circles represent echo times under 90% load

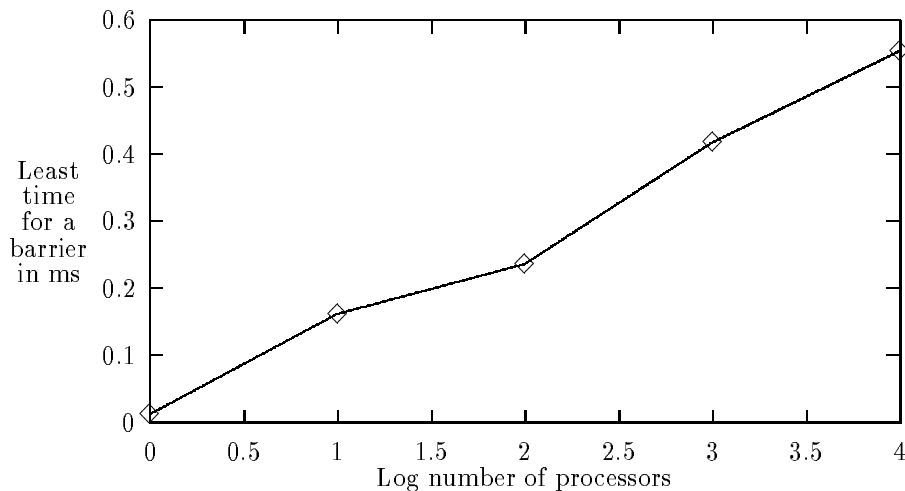


Figure 5.6: Barrier synchronization

To interpret results obtained for barrier synchronization and global reduction, we gathered the numbers obtained for different numbers of nodes and plotted the minimum time required for a barrier synchronization (Figure 5.6) and the time taken for a global sum of a single double precision number (Figure 5.7) against the log of the number of processors in the subcube running the benchmark. Interestingly, the global reduction times increase almost linearly with the log of the number of processors. This is a characteristic of a good global sum algorithm. The time to barrier synchronize also increases in approximately a logarithmic fashion with the number of nodes.

Finally, the last graphs (Figure 5.8, Figure 5.12, Figure 5.10, Figure 5.11 and Figure 5.9) in this section pertain to the overlap of computation and communication. Because of separate communication and computation hardware in the iPSC/860, communication and computation can be partially overlapped. The figures display results for message sizes varying from 0 to 85000 and for vector lengths varying from 0 to 200000 in double precision units. The version of the kernel that uses asynchronous primitives to achieve

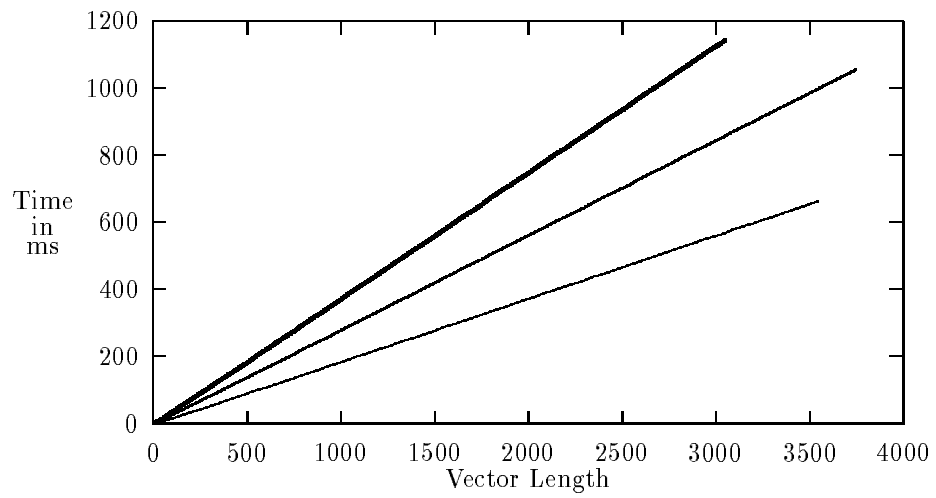


Figure 5.7: Global Sum. The lowest line represents the broadcast times when the kernel was run on 4 nodes, the middle line represents the elapsed times on 8 nodes and the upper line represents the times on 16 nodes.

overlap shows an improvement over the version that uses synchronous calls in all cases except for very low message sizes where the overhead involved in setting up the asynchronous call (isend and iwait) does not match the gain resulting from the overlapped communication. We observed improvements of over 33% when overlapping a pairwise transfer of large messages with double precision computation involving vector lengths. We also observed that even if there is little computation, as messages get larger, the overlapped version of the kernel gets faster because the message is copied directly into user space, avoiding a memory copy. We notice from Figure 5.10, that at some points, the kernel completes a lot faster than expected. The results seem to be similar to that obtained while running the pairwise exchange kernel. Again, like in the results obtained by running the pairwise exchange kernel, we are not sure of the sequence of events except that it could be possible that the communicating nodes were synchronized such that the receive had already been posted facilitating the faster communication.

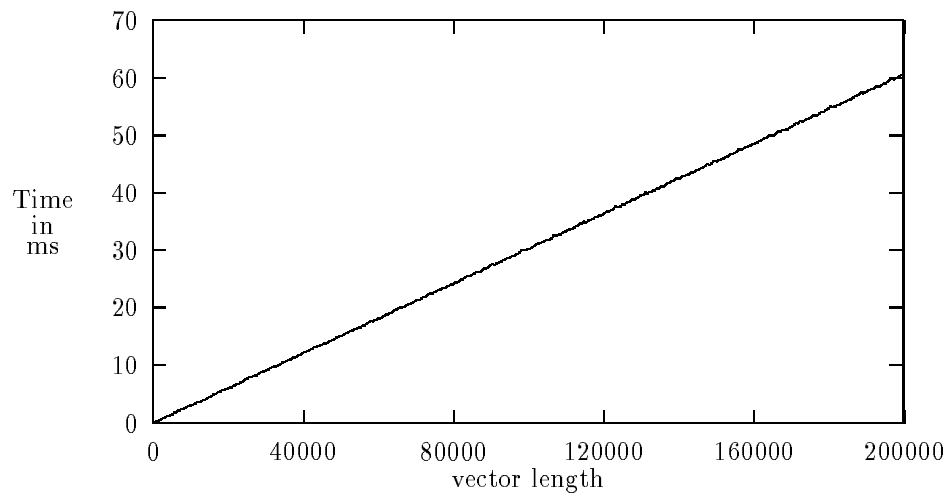


Figure 5.8: DAXPY run on 4 nodes

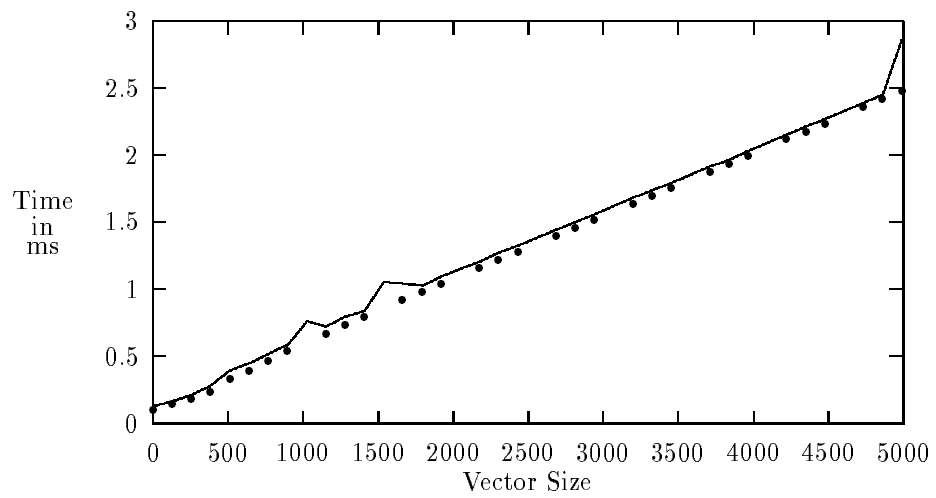


Figure 5.9: Overlap of computation and communication at message size '0'. The dots represent the times for the non-overlapped version and the line represents the times for the overlapped version of the kernel.

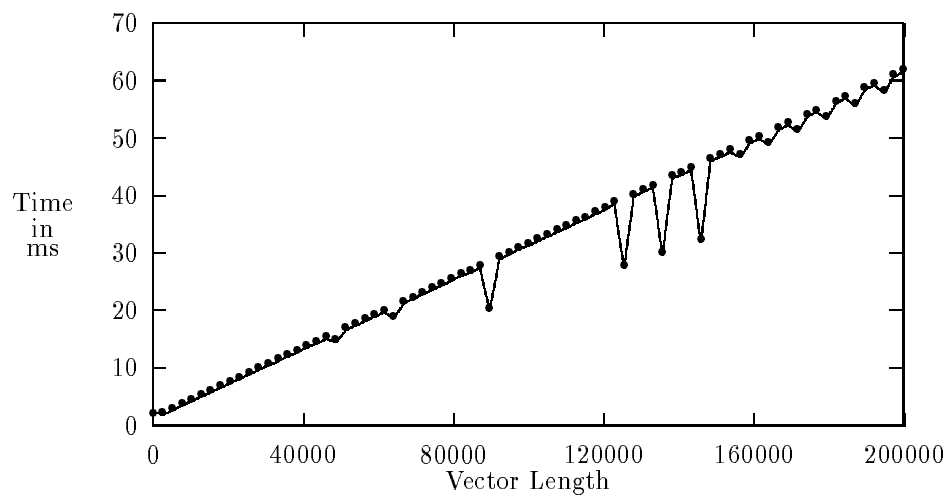


Figure 5.10: Overlap of computation and communication at message size '2536'. The dots represent the times for the non-overlapped version and the line represents the times for the overlapped version of the kernel.

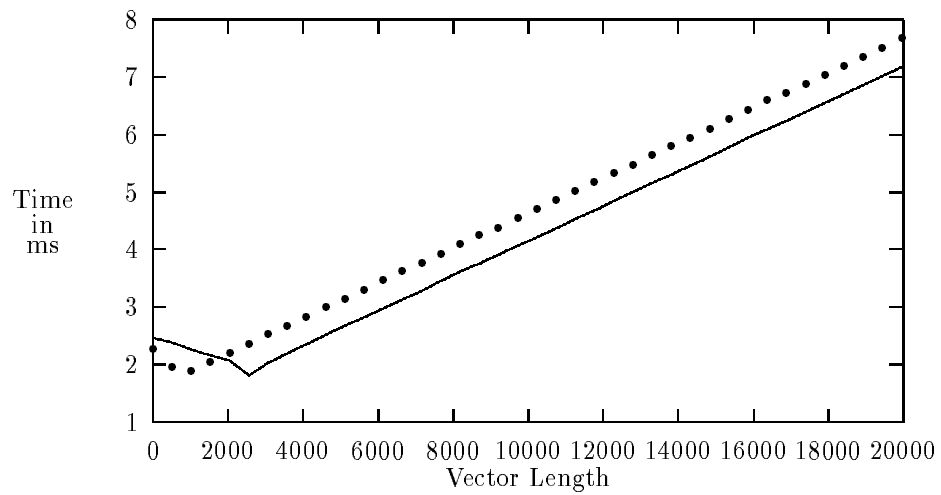


Figure 5.11: Overlap of computation and communication at message size '2536'. A zoomed-in view of figure 5.10 for low message sizes. The dots represent the times for the non-overlapped version and the line represents the times for the overlapped version of the kernel.

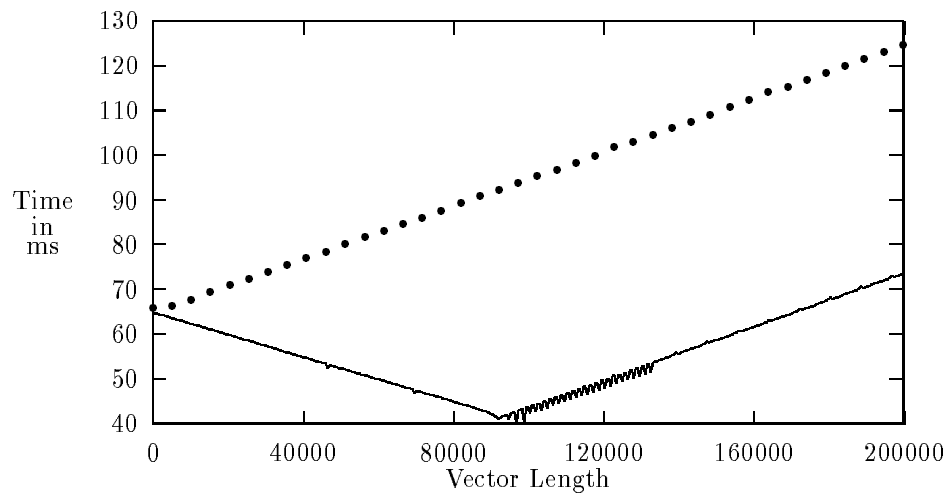


Figure 5.12: Overlap of computation and communication at message size '85000'. The dots represent the times for the non-overlapped version and the line represents the times for the overlapped version of the kernel.

5.2 Matrix-related kernels

The hypercube topology had to be mapped onto a two-dimensional grid to allow for a block decomposition of a two-dimensional matrix on it. Since most matrix manipulations for achieving matrix multiplication involve neighboring block values, the hardware topology was mapped in such a way that the adjacent nodes of the hypercube also possessed adjacent blocks of the matrix. Using a gray code function, we could map the hypercube topology onto a two-dimensional grid keeping the constraint of adjacency completely satisfied, i.e. adjacent blocks of the application matrix could be decomposed onto adjacent nodes. Thus, most of the matrix operations were simple data transfers between adjacent nodes. Hence, we find a linear relationship between **update_guard** (Figure 5.13), **row_broadcast** (Figure 5.15) and **col_broadcast** (Figure 4.4) times and matrix sizes. The relationship between **shift_matrix**(Figure 5.14) and **transpose**(Figure 5.16) times and the matrix size is square. The matrix size in the graphs refer to the matrix dimension. Figure 4.4 has two different curves for row and column broadcast because of the extra time involved in assembling the column into a buffer before broadcasting the column to the appropriate nodes.

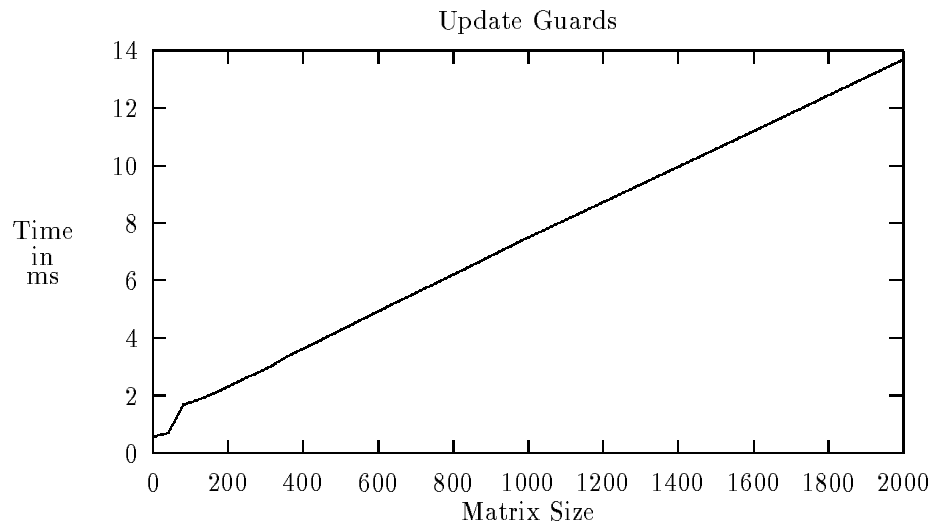


Figure 5.13: Update guard wrapper on 16 nodes

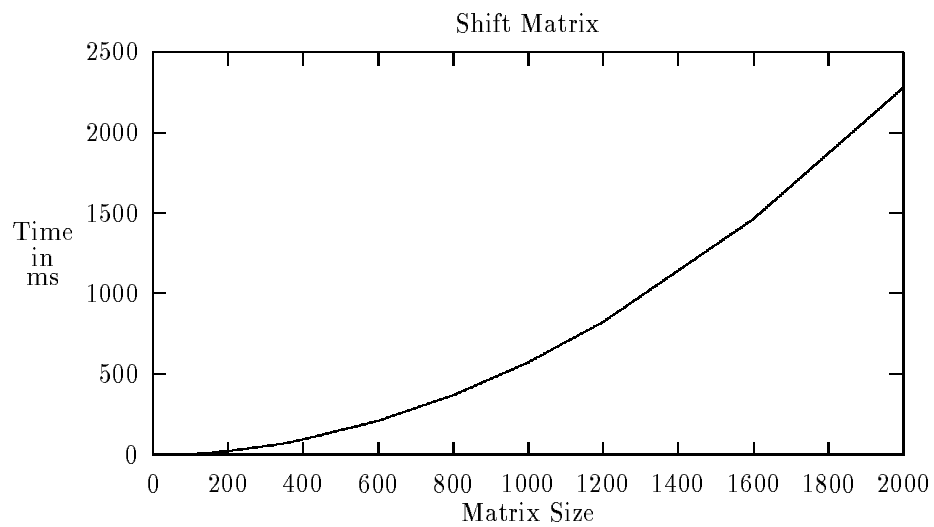


Figure 5.14: Shift matrix on 16 nodes

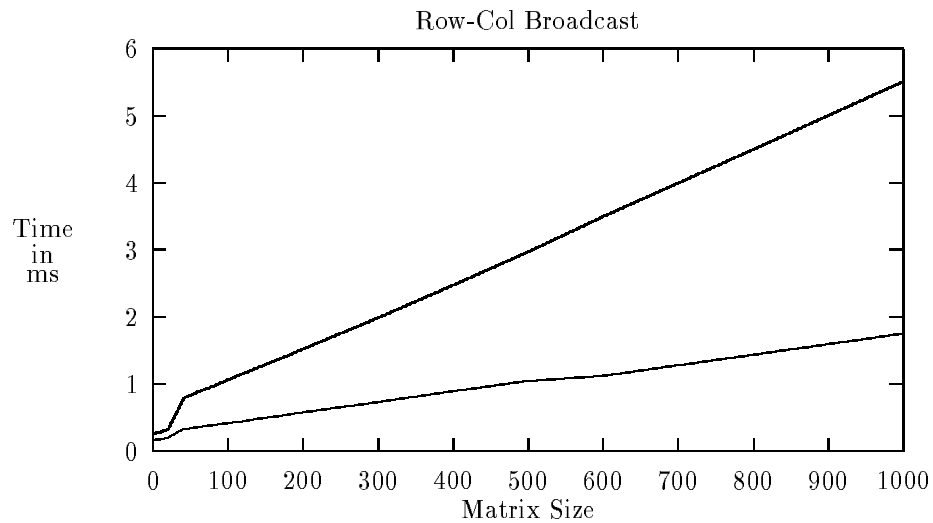


Figure 5.15: Row and Column Broadcast on 16 nodes

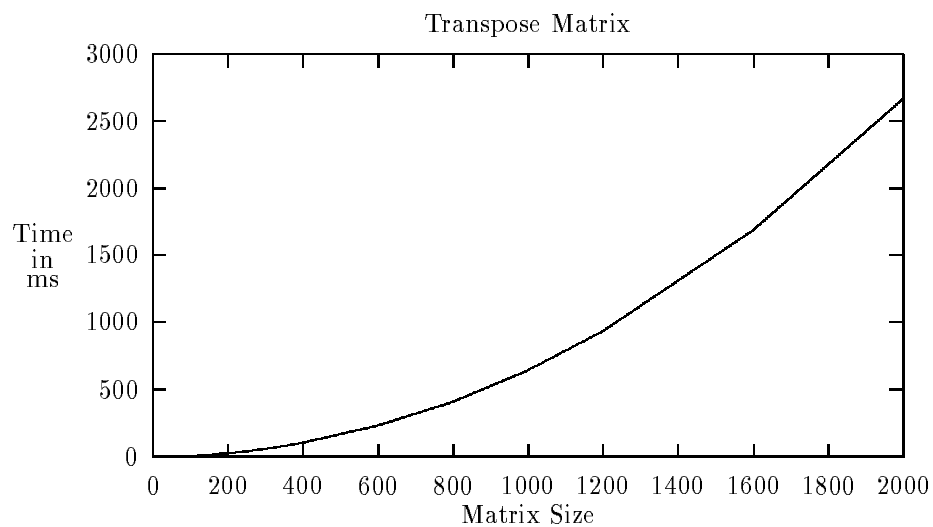


Figure 5.16: Transpose matrix on 16 nodes

Chapter 6

Conclusions

To summarize, we have developed a benchmark, CoMet, for parallel systems that communicate using message passing. The benchmark program consists of a set of 8 files (init.c, main.c, setup.c, check.c, mat_ops.c, repp.c, prototypes.h and extern.h). The benchmark along with a Make file, user's guide and a shell script to execute the benchmark and organize the results is available by anonymous ftp from cse.ogi.edu at /pub/dsrg/CoMet.

We have implemented the benchmark on Intel's iPSC/860 - a parallel system based on i860 RISC processors and a hypercube interconnect topology. We have plotted graphs based on the elapsed times for the various benchmark kernels and have derived communication characteristics such as start-up time, transfer time per byte and the overheads incurred for non-adjacent nodes communicating. At this point of time, existing distributed memory benchmarks such as Genesis and CPEP concentrate on porting full-length applications to evaluate the distributed memory architectures. Although the Genesis suite of benchmarks follows the layered approach to benchmarking, the synthetic kernels included in the benchmark are too few to allow users to model their applications based on the benchmark. We have adopted the synthetic approach of simulating many common communication patterns found in scientific processing to benchmarking the distributed memory message passing systems because of the tremendous effort in terms of time and manpower required to port full applications to these systems. We attempt to add to the information provided by the other benchmarks so as to guide users in developing and in constructing realistic expectations of the performance of their applications based on the benchmark results.

The benchmark is organized into basically two parts - kernels that are basic communication related and kernels that are matrix manipulations based on a block decomposition of a two-dimensional matrix. While the results obtained from running the basic communication related kernels directly relate to the fundamental message passing characteristics of the distributed architectures; the matrix related kernels are based on commonly occurring patterns emanating from matrix manipulations. We have also attempted at measuring the effects of contention by communicating messages in both idle and busy conditions. We believe that overlapping communication with computation results in significant gains especially when large messages are involved in the communication and have also attempted at measuring the extent to which communication can be overlapped with computation. The results obtained from running these benchmarks can thus be used in constructing simulators to predict performances of parallel systems and in other types of performance modeling.

6.1 Future Work

We have strived to make our benchmark as portable as possible and have successfully ported the benchmark to the iPSC/860. Future work could include successfully porting the benchmark on PVM¹ and to the numerous other existing distributed memory architectures such as the CM-5. Our design of the benchmark has been influenced by the iPSC/860 architecture. This may be a problem while porting CoMet to architectures completely different from the iPSC/860 and to systems that support different parallel paradigms. But, the Message Passing Interface Forum[For] has proposed a message passing interface standard very similar to the iPSC/860 message passing interface. A good extension would be to port the benchmark to the message passing interface and if the standard gets widely accepted, CoMet would easily be ported to a number of distributed memory machines.

¹PVM : Parallel Virtual Machine [Sun90] provides the user with a message passing interface over a network of loosely coupled UNIX workstations - see Appendix for some initial results on porting PVM

Further future work could include extending the benchmark to include kernels that measure characteristics of interrupt-driven sends and receives such as in `hsend/hrecv` of `iPSC/860`. In fact, Eicken, et al.[vECGS92] have proposed hardware support for message-driven sends and receives to achieve greater overlaps of communication and computation. The benchmark could be made more comprehensive by including many more common communication patterns from scientific computing. Also, in reality, many applications require access to large amount of data resident in secondary memory and our benchmark could be extended to measure the basic characteristics of the interaction of the distributed memory system with the Input/Output subsystem.

Bibliography

- [AGH⁺92] Cliff Addison, Vladimir Getov, Tony Hey, Roger Hockney, and Ivan Wolton. The genesis distributed-memory benchmarks. *Proc. of Parallel Processors - Benchmarking and Assessment*, March 1992.
- [BBDS92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. In *Proceedings - Supercomputing '92*, 1992.
- [BDH90] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing* 15, 1990.
- [BGL91] M. Berry, G. Cybenko, and J. Larson. Scientific benchmark characterisations. *Parallel Computing* 17, 1991.
- [BH92] Rudolf Berrendorf and Jukka Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice and Experience*, Vol 4, May 1992.
- [Cyb91] George Cybenko. Supercomputer performance trends and the perfect benchmarks. Technical Report 1093, CSRD, University of Illinois at Urbana-Champaign, April 1991.
- [Dix91] Kaivalya M. Dixit. The SPEC benchmarks. *Parallel Computing* 17, 1991.
- [DMW87] Dongarra, Martin, and Worlton. Computer benchmarking : paths and pitfalls. *IEEE Spectrum*, July 1987.

- [Don89] J. Dongarra. Performance of various computers using standard linear equations software in a fortran environment. Technical Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, TN, 1989.
- [DS86] J. Dongarra and D. Sorensen. Linear algebra on high performance computers. In *Proceedings of Parallel Computing*, 1986.
- [Dun91] T. H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing 17*, 1991.
- [Dun92] Thomas Dunigan. Communication performance of the intel touchstone delta mesh. Technical Report ORNL/TM-11983, Oak Ridge National Laboratory, January 1992.
- [Feo88] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing 7*, 1988.
- [FJG⁺88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on Concurrent Processors*, volume I. Prentice Hall, Inc., 1988.
- [Fly66] M. J. Flynn. Very high speed computers. In *Proceedings of IEEE*, 54, December 1966.
- [For] The Message Passing Interface Forum. Draft 1.0 of MPI. Technical report. Send electronic mail to netlib@ornl.gov with "send info from mpi" in the message body. This will tell you the current state of the MPI standardization effort.
- [Gei90] G. A. Geist. A user's guide to PICL: a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, September 1990.
- [Hey91] Anthony J. G. Hey. The genesis distributed memory benchmarks. *Parallel Computing 17*, December 1991.

- [Hoc91] R. Hockney. Performance parameters and benchmarking. *Parallel Computing 17*, 1991.
- [Int91] Intel Corporation. *iPSC/860 Programmer's Manual*, 1991.
- [MBF+90] P. Messina, C. Baillie, E. Felten, P. Hipes, R. Williams, A. Alagar, A. Kamrath, R. Leary, W. Pfeiffer, J. Rogers, and D. Walker. Benchmarking advanced architecture computers. *Concurrency: Practice and Experience 2(3)*, 1990.
- [Nug88] Steve Nugent. The iPSC/2 direct-connect technology. In *Proceedings of the 3rd conference on Hypercube Concurrent Computers and Applications*, 1988.
- [Pie88] Paul Pierce. The NX/2 operating system. In *Proceedings of the 3rd conference on Hypercube Concurrent Computers and Applications*, 1988.
- [SH91] Willi Schonauer and Harmut Hafner. Performance estimates of supercomputers. *Parallel Computing 17*, 1991.
- [Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience Vol. 2(4)*, December 1990.
- [Thi91] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, October 1991.
- [vdS91] Aad J. van der Steen. The benchmark of the Euroben group. *Parallel Computing 17*, December 1991.
- [vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauer. Active messages: a mechanism for integrated communication and computation. *Communications of ACM*, July 1992.
- [Wei84] R. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of ACM*, 1984.

- [Wei91] R. P. Weicker. A detailed look at some popular benchmarks. *Parallel Computing* 17, 1991.