Applications Benchmark Set for Fortran-D and High Performance Fortran

A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski
Manish Parashar, Tomasz Haupt, Kim Mills
Ying-Hua Lu, Neng-Tan Lin and Nang-kang Yeh

Contents

Applications Benchmark Set for Fortran-D and High Performance Fortran

A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski
Manish Parashar, Tomasz Haupt, Kim Mills
Ying-Hua Lu, Neng-Tan Lin and Nang-kang Yeh

Keywords: CompilerBenchmarking, FORTRAN 90D, FORTRAN 77D, HPF

ABSTRACT

Fortran77, the currently accepted Fortran standard worldwide, is essentially a sequential language which hides the parallelism of a problem in sequential constructs like loops, etc. Consequently, scientists wishing to use parallel computers must rewrite their programs in an extension of Fortran that explicitly reflects the architecture of the underlying machine, such as a message-passing dialect for MIMD distributed-memory machines, array syntax for SIMD machines, or an explicitly parallel dialect with synchronization for MIMD shared-memory machines. This conversion is difficult, error prone, and the resulting parallel codes are machine-specific.

To overcome these problems a new Fortran standard, or more precisely, a standard of Fortran extensions, are necessary to establish a machine independent programming model that is easy to use and yet, is acceptably efficient on different parallel architectures. Thus research is now concentrated on the provision of appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on a sequential machine. Several proposals have been put forth in recent months for a set of language extension to achieve this. To coordinate these efforts the High Performance Fortran Forum (HPFF) has been created. HPFF is a coalition of industrial and academic groups working to develop an industry-wide standard of extensions to Fortran which provide support for high performance programming on a wide variety of machines, portable from workstations to massively parallel SIMD and MIMD supercomputers. Fortran-D, a version of Fortran enhanced with data decomposition directives, can provide such a programming model. We believe that it can make parallel computing truly usable.

One of the elements of this program is to establish a reliable validation strategy. In order to evaluate the efficiency of automatic data partitioning schemes, a dedicated benchmarking suite is being developed at NPAC and is described in this paper. Currently, the suite is oriented towards validation of the Fortran-D compiler, which is also being developed at NPAC in collaboration with Rice University. We plan to augment it with applications written in other proposed HPF dialects. [1]

[1]This work was sponsored by DARPA under contract #DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. Additional support was also provided by CRPC under NSF Cooperative agreement Nos. CCR-9120008 and CDA-8619893 with support from the Keck foundation

1 INTRODUCTION

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to scientists and engineers. Unfortunately, its advantages are hindered by the difficulty of parallel programming and the lack of portability of resulting programs. As a result, although parallel computers have been commercially available for some time, their use has been mostly limited to academic and research institutions. It is our goal, at the Northeast Parallel Architectures Center (NPAC) at Syracuse University, to introduce parallel computing to industry. Since Fortran has been the language in which most industrial and governmental software investments have been made, we have decided to focus on ways to make Fortran portable to parallel computers with an acceptable efficiency. Fortran-D, a version of Fortran enhanced with data decomposition directives [13], can provide such a programming model. We believe that it can make parallel computing truly usable.

Since its introduction over three decades ago, Fortran has been the language of choice for scientific programming. Fortran77, the currently accepted Fortran standard worldwide, is essentially a sequential language which hides the parallelism of a problem in sequential constructs like loops, etc.. Consequently, scientists wishing to use parallel computers must rewrite their programs in an extension of Fortran that explicitly reflects the architecture of the underlying machine, such as a message-passing dialect for MIMD distributed-memory machines, array syntax for SIMD machines, or an explicitly parallel dialect with synchronization for MIMD shared-memory machines. This conversion is difficult, error prone, and the resulting parallel codes are machine-specific.

To overcome these problems a new Fortran standard, or more precisely, a standard of Fortran extensions, are necessary to establish a machine independent programming model that is easy to use and yet, is acceptably efficient on different parallel architectures. Thus, research is now concentrated on the provision of appropriate high-level language constructs to enable users to design programs in much the same way as they are accustomed to on a sequential machine. Several proposals have been put forth in recent months for a set of language extension to achieve this (like Fortran-D [6], Vienna Fortran [2], Digital Equipment Corporation's High Performance Fortran proposal [20], the language extensions to cf77 planned by Cray Research Inc. [41], suggested extension from Thinking Machines Co. [44], and others); and research towards language definition is accompanied by research in compiler technology. To coordinate these efforts the High Performance Fortran Forum (HPFF) has been created. HPFF is a coalition of industrial and academic groups working to develop a industry-wide standard of extensions to Fortran which provide support for high performance programming on a wide variety of machines, portable from workstations to massively parallel SIMD and MIMD supercomputers.

One of the elements of this program is to establish a reliable validation strategy. In order to evaluate the efficiency of automatic data partitioning schemes, a dedicated benchmarking suite is being developed at NPAC and is described in this paper. Currently, the suite is oriented towards validation of the Fortran-D compiler, which is also being developed at NPAC in collaboration with Rice University. We plan to augment it with applications written in other proposed HPF dialects. This paper is organized as follows. The remaining part of the introduction describes the main features of Fortran-D. Chapter 2 provides an overview of NPAC's benchmark suite. Chapter 3 briefly summarizes the architectures of target machines used to develop the parallel codes included in this suite. The next chapters contain descriptions of the benchmark applications, their implementation strategies on different architectures and their performance. Chapter 4 describes the Purdue benchmarking set. Chapter 5 describes parallelization of a set of matrix factorization routines from the LAPACK library. Chapter 6 describes the problem of electromagnetic scattering from a plane conductor. Chapter 7 describes modeling of stock option pricing. Final remarks and conclusions are summarized in the last chapter.

1.1 Outline of the Fortran-D Design

Fortran-D is a machine independent set of compiler directives to Fortran77 and/or Fortran90 and is expected to be the core of the emerging HPF standard. The model of parallel computation adopted for Fortran-D is based on the idea of "Annotated Complete Program" [43]. The programmer writes standard serial code and parallelization is achieved by providing the compiler with indications as to how data structures should be partitioned among processors. In addition, the programmer may specify how the program can be parallelized, in particular, by indicating loops that can be performed in parallel. If these compiler directives are ignored the program can be run without change on a sequential machine. The compiler for parallel machines can use the directives not only to decompose data structures but also to infer parallelism, based on the principle that only the owner of a datum computes its value. In other words, the compiler attempts to find additional parallelism beyond that explicitly indicated by the programmer.

The advantage of this approach is that serial programs annotated solely with data distribution declarations are easier to write than explicitly parallel programs and promise reasonably low cost of porting existing sequential applications to parallel supercomputers. The key question regarding this model is how well a sophisticated compiler will be able to identify parallelism, and generate efficient communications, and how efficient the resulting parallel program will be in comparison to explicitly parallel programs for the same algorithms. Currently, this is an open research issue. The NPAC benchmark suite is meant to provide "experimental data" to address these questions.

Perhaps the most important intellectual step in preparing a program for execution on a distributed-memory parallel computer is to choose a decomposition for the fundamental data structures used in the program. Once selected, this data domain decomposition scheme often determines the parallelism in the resulting program. Unfortunately, there are no existing tools to help the programmer

perform this important step correctly. From this point of view an important issue is the Vchoice between Fortran-77 and Fortran-90 as the base language for the High Performance Fortran. The argument for Fortran-90 is that it may prove to be better suited for a data-parallel programming style and thus simplifying the programmer's task of selecting the most efficient decomposition. Again, it is an open research issue. To address this question, there are two dialect of Fortran-D under development, Fortran77-D (Rice University) and Fortran90-D (NPAC), and the NPAC's benchmark suite is unique in the sense that it provides codes in both Fortran dialects.
The detailed description of the Fortran-D language and the compiler can be found elsewhere [13, 6, 7]. The exact syntax of the language is a subject of research at NPAC and Rice University as well as a subject of discussion at the High Performance Fortran Forum.

2 OVERVIEW OF THE HPF/FORTRAN-D BENCHMARKING SUITE

2.1 Validation Strategy

The primary purpose of the suite is to provide a fair test for the prototype Fortran-D compiler and data partitioner. Our validation strategy involves providing source codes of each application in 6 Fortran dialects:

1. Fortran77
2. Fortran77+hand coded message passing (Fortran77+MP)
3. Sequentialized version of Fortran77+message passing (Fortran77||)
4. Fortran77-D
5. Fortran90
6. Fortran90-D

Version 1, written in plain Fortran77, is the original application. Version 2 is an optimized, to the best of our knowledge, hand-coded message passing version of the original application. We do not claim that this algorithm is the optimal for a the target machine. The target machine is the Intel's hypercube iPSC/860, and message-passing is implemented using the proprietary Intel message passing library and/or portable communication libraries like Express [40] or PICL [16, 15]. Version 3 is based on the exact same algorithm as the massage passing program version 2 except that all explicit message-passing and blocked loops are removed. Fortran77-D, version 4, is based on version 3 extended by the Fortran-D compiler directives. Version 5 is the Fortran90 version of the original application. Actually, we provide two versions, one written in TMC's CMFortran (to be run on CM2 and/or CM5) and DEC/Maspar's mpfortran (to be run on DECmpp 12000). Finally, version 6 is Fortran90 + Fortran-D directives. For the HPFF purposes we will augment the suite by codes written in other HPF dialects, when available.

To validate the Fortran77-D and Fortran90-D compilers we will compare the running time of the hand-coded Fortran77+MP (version 2 above) with the running time of the output obtained by compiling the Fortran77-D (version 4) and Fortran90-D (version 6), respectively. Since we do not yet expect Fortran-D compiler to perform high-level algorithm changes, we decided to use sequentialized version of the message-passing codes to build the Fortran-77D programs.

An important feature of this suite is that we provide both SIMD and MIMD version of each application. This allows for an independent evaluation of how well different algorithms may be implemented on different types of computer's architectures.

2.2 Current Status

Currently, the NPAC's benchmarking suit consists of 45 items, divided into ten groups, as listed in Table 1 and 2. It is expected that more applications will be added soon.

The first three groups, Vsuite, General Mathematical Applications and the Purdue Set, collect applications designed for the initial test of the Fortran-D compiler. The applications are simple, but diverse. They selectively address different aspects of parallel computing and thus enable systematic, clear testing of the compiler at the development phase. Of particular value in this context is the Purdue Set, a benchmarking suite to test parallel language designs proposed by the Purdue University Group [42]. Actually, most of the computational problems included in the set have been extracted from larger computations and thus are somewhat artificial by themselves. Nevertheless, the suite does comprise a rich sample of practical computations, as shown in Table 3 . The results of the Fortran-D compiler tests based on the Purdue Set will be presented in a forthcoming paper.

The problems of the Vsuite, General Mathematical Applications, and Purdue Set are too small and simple to demonstrate the power of massively parallel supercomputers. Therefore, we deliberately avoid a direct comparison of their performance on different architectures. However, they help to understand the principles of data parallel programming. In fact, they are used as a teaching aid for training purposes as well.

The other seven groups comprise a suite of complete, "real life" applications coming from independent research on parallel algorithms in linear programming, matrix algebra, computational physics, financial modeling, weather and climate modeling, electro-magnetic fields simulation, and Numerical Aerodynamic Simulation (NAS) benchmarking set [1]. Selected applications are described in the chapters 4, 5, 6, and 7. These applications along with other applications currently under development, will be used for the actual validation of the compiler.

Applications	F77	F77+MP	CMF	F77‖	F77D	F90D
I. Vsuite						
5 Simple Examples			X			X
II. General Math. Applications						
1. Gaussian Elimination	X	X	X	X	X	X
2. Π Integration	X	X	X	X	X	X
3. Laplace Solver	X	X	X	X	X	X
4. 2DFFT	X	X	X	X	X	X
III. Purdue set (J.R. Rice set)						
1. Trapezoidal rule	X	X	X	X	X	X
2. reduction functions 1	X	X	X	X	X	X
3. reduction functions 2	X	X	X	X	X	X
4. reduction functions 3	X	X	X	X	X	X
5. simple search	X	X	X	X	X	X
6. tridiagonal set of lin. equations	X	X	X	X	X	X
7. Lagrange interpolation	X	X	X	X	X	X
8. divided differences	X	X	X	X	X	X
9. finite differences	X	X	X	X	X	X
10. Fourier's moments	X	X	X	X	X	X
11. array's construction	X	X	X	X	X	X
12. WHERE construct	X	X	X	X	X	X
13. Simpson's and Gauss' integration	X	X	X	X	X	X
14. Chebyshev interpolation	X	X	X	X	X	X

X= Available Now, A=Available by August,

S= Available by October, D= Available by December 1992

Table 1: Current status of elementary applications of the Fortran-D Benchmarking Suite

Applications	F77	F77+MP	CMF	F77‖	F77D	F90D
IV. Linear Programming						
1. Modified Simplex Method	*X*	*X*	*X*	*X*	*X*	*X*
V. LAPACK						
1. Block LU Factorization-SDOT	*X*	*X*	*X*	*X*	*X*	*X*
2. Block LU Factorization-SAXPY	*X*	*X*	*X*	*X*	*X*	*X*
3. Block LU Factorization-GAXPY	*X*	*X*	*X*	*X*	*X*	*X*
4. Block QR Factorization	*X*	*X*	*X*	*X*	*X*	*X*
5. Block Cholesky Factorization	*X*	*X*	*X*	*X*	*X*	*X*
VI. Physics						
1. Conventional Spin	*X*	*X*	*X*	*A*	*A*	*A*
2. Cluster Spin	*A*	*A*	*A*	*A*	*A*	*A*
VII. Financial						
1. Option pricing	*X*	*A*	*X*	*A*	*A*	*A*
VIII. Weather and Climate						
1. Climate Model (Keppenne)	*X*	*X*	*X*	*A*	*A*	*A*
2. Severe Storm Model (CAPS)	*X*	*S*	*X*	*S*	*S*	*S*
IX. Electromagnetic Field						
1. EM Scattering from Slots(TE case)	*X*	*A*	*X*	*A*	*A*	*A*
2. EM Scattering from Slots(TM case)	*X*	*D*	*D*	*D*	*D*	*D*
3. EM Near-Field To Far-Field Trans.	*X*	*S*	*S*	*S*	*S*	*S*
X. NAS set						
1. Embarrassingly Parallel	*X*	*X*	*X*	*S*	*S*	*S*
2. Multigrid	*X*	*X*	*X*	*S*	*S*	*S*
3. Conjugate Gradient	*X*	*X*	*X*	*S*	*S*	*S*
4. 3DFFT PDE	*X*	*X*	*X*	*S*	*S*	*S*
5. Integer Sorting	*X*	*X*	*X*	*S*	*S*	*S*
6. LU-solver	*X*	*X*	*X*	*S*	*S*	*S*
7. Pentadiagonal Solver	*X*	*X*	*X*	*S*	*S*	*S*
8. Block Tridiagonal Solver	*X*	*X*	*X*	*S*	*S*	*S*

X= Available Now, A=Available by August,

S= Available by October, D= Available by December 1992

Table 2: Current status of real life applications of the Fortran-D Benchmarking Suite

3 ARCHITECTURES

For the evaluation of the algorithms the Connection Machine CM2, the DECmpp-12000 and the Intel iPSC/860 is used. A brief description of these machines follows.

3.1 The Connection Machine CM-2 Architecture

The Connection Machine 2 system [2] is a massively data parallel computer with 65536 processors [46]. Each processor has 1 Megabit of local memory, so the machine has an overall memory capacity of 8 Gigabytes. The processors (and their associated memories) are arranged 16 to a chip in hardware. Each "node" consists of a pair of chips (32 processors) and is supported by a floating-point accelerator.

The CM-2 operates in "single-instruction multiple-data"(SIMD) mode; that is, an identical instruction set is broadcast to each processor. Therefore, it is very efficient for problems which require simultaneous operation on all of the data present. The CM-2 is further characterized by a very sophisticated communications network linking all of its processors. The above mentioned features make the CM-2 well-suited for many problems in computational science and engineering.

The CM-2 system uses a conventional front-end computer. The front-end executes the control structure of programs, issuing commands to the CM-2 processors whenever necessary.

3.2 The DECmpp 12000 Machine Architecture

The DECmpp 12000 computer system [3] includes a RISC-style Array Control Unit with a Harvard architecture and a data-parallel processor (the data parallel unit or DPU) which includes 8192 Processor Elements (PEs) [5, 31]. Each PE is a load/store arithmetic processor with dedicated register space and RAM. Each PE has a 1.8-MIPS control processor, forty 32-bit registers, and 16 KBytes of RAM. The peak performance is 650 MFLOPS DP, 117 Gbyte/sec overall memory bandwidth, and 1.5 Gbtye/sec overall router bandwidth.

The computational core of the machine system is an array of PEs arranged in a rectangular two-dimensional lattice which execute the same tasks in parallel (SIMD parallelism). This core is tightly coupled to a Sun 4 front-end host and also to high-speed overlapping I/O subsystem.

DECmpp 12000 is in the MP-1 family of Data-Parallel computers. The unique characteristics of the MP-1 architecture are the combination of scalable architecture in terms of the number of processing elememts, system memory, and system communication bandwidth; "RISC-like" instruction set design that leverages optimizing compiler technology; and adherence to industry standard floating

[2] Connection Machine is a registered trademark of Thinking Machines Corporation

[3] DECmpp 12000 is a registered trademark of Digital Equipment Corporation

point design, specifically VAX and IEEE floating point. The architecture provides not only a high computational capability, but also a mesh and global interconnect style of communication.

3.3 The Intel iPSC/860 Architecture

The Intel iPSC/860 is an Intel i860 processor based hypercube attached to a 80386 host processor (System Resource Manager). Each i860 node has an 8 KByte cache and 8 to 16 MBytes of main memory. The clock speed is 40 MHz and each node has a theoretical peak performance of 80 MFLOPS for single precision and 40 MFLOPS for double precision. The i860 processor uses pipelining and instruction caching allowing it to process more than one instruction per clock cycle. Communication is supported by direct-connect modules present at each node [39] which allow the nodes to be treated as though they are directly connected. The direct-connect modules control eight bidirectional channels, each supporting a peak bandwidth of 2.8 Mbyte/sec. The communication time for a message is a linear function of the size of the message. Hence, the time, t_m to transmit a message of length n bytes from an arbitrary source node to an arbitrary destination node is given by:

$$t_m = t_s + t_b \times n$$

where t_s is the fixed startup overhead and t_b is the transmission time per byte.

4 PURDUE BENCHMARKING SET

General Features of the Purdue Set

The comprehensive discussion of the Purdue Benchmarking Set can be found elsewhere [12]. In this paper we outline the most important features and conclusions.

The Purdue set is a collection of 17 computational problems proposed by the Purdue University group [42]. Actually, most of the problems of the set have been extracted from larger computations and thus are somewhat artificial by themselves. However, they do represent a sampling of practical computations, even though not always the most efficient algorithms have been selected. The great value of the Purdue set is that the problems address many different aspect of parallel computing. In particular, they represent various schemes of data dependencies, from very simple with almost no interprocessor communication required to more complex with large communication overhead to irregular problems, difficult to parallelize.

There are many advantages of including simple applications to the compiler benchmarking suite. First of all, they are very useful for initial tests of a compiler performance. This includes comparison of automatic parallelization performed by the compiler to hand written codes and "hand compiled"

Table 3: List of the problems of the Purdue Set

1. evaluate the trapezoidal rule estimate of an integral of f(x)
2. compute the value of $e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} \left(1 + \frac{0.5}{-|i-j|+0.001}\right)$
3. compute the value of $S = \sum_{i=1}^{n} \prod_{j=1}^{m} a_{ij}$
4. compute the value of $R = \sum_{i=1}^{n} \frac{1}{x_i}$
5. given a table of the i-th student's score on the j-th test calculate the average score of each student, increase all above average by 10%, give the lowest score that is above average and finally find a genius - a student who has all scores above average.
6. compute polynomial interpolant values of f(x) using Lagrange interpolation formulas:
$$p(x) = \sum_{i=1}^{n} f(x_i)\, l_i(x)\,;\; l_i(x) = \frac{\prod (x - x_j)}{\prod (x_i - x_j)} \tag{1}$$
7. compute the first M columns of the divided difference table $f[x_i, x_{i+1}, ..., x_{i+k}]$, where
$$f[x_i, x_{i+1}, ..., x_{i+k}] = \frac{f[x_{i+1}, ..., x_{i+k}] - f[x_i, ..., x_{i+k-1}]}{x_{i+k} - x_i} \tag{2}$$
8. given an array $u_{i,j}$ replace each value by the average of its value plus those of all its neighbors, i.e. $u_{i,j} = \left(\sum_{neighbors} u_{ij}\right) / (Number\ of\ neighbors)$
9. do a logarithmic transformation $d_i = \lg(1 + d_i)$ for a set of data d_i , i=1,N, and compute first four Fourier moments $F_j = \sum_{i=1}^{N} d_i \cos(\pi j / (N+1))$
10. given the m by m matrix A, the 1 by m vector R, the m by 1 vector C, and a number a, construct the array
$$ABIG = \begin{pmatrix} A & C \\ R & a \end{pmatrix} \tag{3}$$
11. For given vectors a, b, c, and d of dimension N, compute the new vector $a_i = a_i \sin b^i$. $a_i < \cos(c_i)$ then $a_i = a_i + c_i$ otherwise $a_i = a_i - d_i$ and then compute $e = \sum_{i=1}^{N} a_i^2$
12. carry out a test of there methods to integrate a function
13. carry out a comparison of two types of interpolation points (equidistant and Chebyshev spaced) for Hermite interpolation using piecewise cubic polynomials.
14. carry out a test of three methods to integrate a function
15. carry out a comparison of two types of interpolation points (equidistant and Chebyshev spaced) for Hermite interpolation using piecewise cubic polynomials.

programs. The latter is expected to be a very powerful tool to determinate the completeness of the proposed compiler directives. In addition, since the original fortran77 codes are easy to understand, they can be used for educational purposes as well.

It is a well known fact that parallel supercomputers are efficient for sufficiently large problems only. Therefore, the problems of the Purdue set has been modified by increasing size of matrices, and some parts of the code are computed several times in a loop in order to increase the number of arithmetical operations. Few minor bugs in the fortran77 codes has been fixed as well. Finally, we decided to drop 3 of 17 applications of the original set because very similar applications already exist in our benchmarking suite. The problems used in the suite are listed in Table 1.

Parallelization of the Purdue Set

The performance of the parallelized versions on the Thinking Machines 16k-processors CM2 (SIMD architecture), and Intel's iPSC/860 (MIMD), is demonstrated in Figures 1 and 2, respectively. For the CM2, the speedup

$$s = \frac{T_{DECstation\ 5000/120}}{T_{CM2}} \qquad (4)$$

is shown as a function of the problem size, while for iPSC/860 the speedup defined as

$$s = \frac{T_1}{T_n} \qquad (5)$$

is plotted as a function of number of processors, n, for fixed problem size. T is elapsed time of execution excluding loading and I/O.

Deliberately, we avoid the direct comparison of different architectures. The reason for that is we recognize the Purdue set as not representative enough for that purpose. Instead, our goal is to demonstrate that the Purdue set has been efficiently implemented on both SIMD and MIMD machines and therefore can be used for validation of the compiler.

The first 4 problems (see Table 3) are very regular and can be easily expressed in array syntax of Fortran90. The resulting code can be compiled for a SIMD machine 'as is' or converted for MIMD architecture by aligning Fortran90 arrays with, in this case, a cyclic decomposition to be distributed over nodes.

Problem 5 is hardly a real life application, however it introduces an interesting structure of the do-loops. Since the most efficient implementations on SIMD and MIMD architectures require different loop arrangements, it is a valuable test for the compiler performance.

Problem 6 deals with interpolation. In principle two extreme cases can be considered here: interpolation using high order Lagrange polynomials or interpolation of many points using the low order polynomials. We parallelized the Fortran77 code assuming the latter case as we found it more practical but leads to very asymmetric matrices, i.e. matrices with dimension in one direction much larger than in the other. This requires a careful mapping of the problem on the processors grid to

minimize communication overhead. For a distributed memory machine a cyclic decomposition is efficient, however, the redundant computation of the Lagrange coefficients is difficult to avoid.
Problem 7 addresses interpolation in a different way. The algorithm can be easily expressed in Fortran90, even though the interprocessor communication is quite complex. Thus the efficiency of the SIMD code depends to large extend on the compiler (CMF, mpfortran) performance. Appropriate compiler directives help increasing the speedup. The most effective MIMD implementation requires a block decomposition.
The next problem involves an intensive "next neighbor" communication. Again, it is easy to convert to the array syntax for SIMD architectures. For distributed memory systems the natural decomposition is a block one. This algorithm is a very good example to demonstrate how tedious (and error prone!) may be coding a MIMD machine without fixing the number of processors, even if the pattern of message passing is very simple and regular.
Problems 10 and 11 address the opposite aspects of parallel computing: interprocessor communication without arithmetic operations and asynchronous floating point operations, respectively, while problem 9 is a mixture of both. Problem 10 can be very conveniently and clearly coded using array sections, but a careful alignment is crucial to minimize communication overhead. The allocatable and assumed-shape arrays of Fortran90 allow for easy changes of the size of arrays, however, to take a full advantage of this, a dynamic alignment should be possible as well. This feature is not supported either by CM-fortran of Thinking Machines or mp-fortran of Maspar/DEC. The MIMD implementation of problem 10 require a block decomposition and the dynamic alignment of the arrays to the decomposition would be desired in Fortran-D too. The polymorphism of Fortran90's intrinsic functions makes possible to avoid tedious coding of do-loops in problem 11. The best load balance for this problem is achieved by a cyclic decomposition.
Finally, problems 12 and 13 compare different method of integration and interpolation. The real value of the comparison lies in the different ratio of communication to computation. For an efficient SIMD implementation of problem 12 one would need a possibility to define elemental (or even better, polymorphic) functions. Unfortunately, this feature is not supported yet by CM-fortran nor mp-fortran, even so the user defined polymorphic functions are part of the ISO/ANSI standard of Fortran90. In our implementation, we defined a subroutine taking an assumed-shape array of the function arguments as an input and returning an assumed-shape array of function values. For more advanced applications there may be necessary to dynamically redistribute and/or realign parameters of functions and subroutines, especially when using libraries of compiled modules. We acknowledge this feature as a part of the Fortran-D design. Problem 13 indicate to another language construct which may be very useful for programmers: nested WHERE constructs. Apparently, they are natural "parallel equivalents" of nested IF constructs and would simplify the conversion of Fortran77 do-loops into array syntax. In spite of the fact that the nested WHERE may prove

to be very inefficient for SIMD architectures, they may be very useful in Fortran90D.

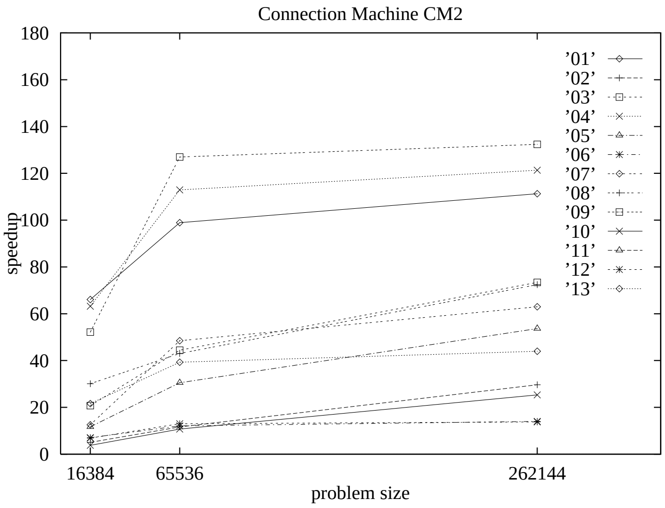


Figure 1: Speedups for the Purdue problems on CM2 over DECstation 5000/120

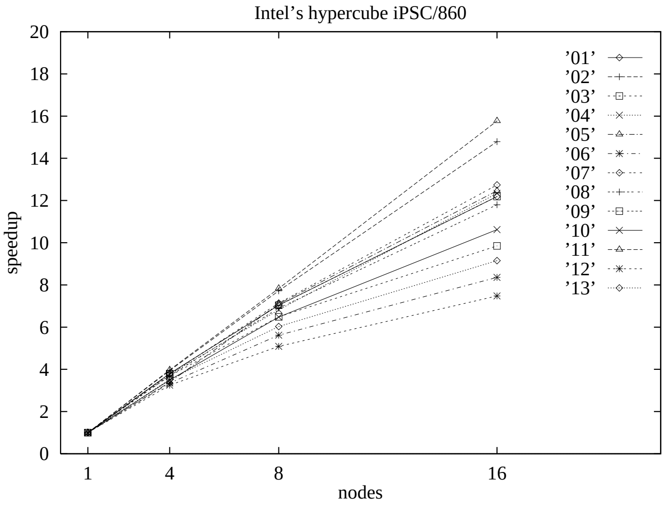


Figure 2: Speedups for the Purdue problems on Intel's hypercube iPSC/860

Remarks on the Purdue Set

Although the problems of the Purdue set do not allow to demonstrate the power of massively parallel supercomputers, they help to understand the principles of (data) parallel programming. In fact, they are used for training purposes as well as for initial tests of the Fortran-D compiler.
One of the most important conclusions from parallelization of the Purdue set is that all algorithms included in the set can be easily expressed using the array syntax of the Fortran90 and such conversion leads generally to more compact and yet more clear codes. Moreover, in this way data parallelism is easier to understand and easier to implement. The directives of the Fortran-D compiler seems to be sufficient for efficient automatic conversion of the Fortran90 codes to source codes for MIMD machines. It remains to be shown, however, that these conclusions hold for more complex applications, in particular those not well suited for the SIMD architecture.

5 LAPACK SUBSET

The LAPACK subset consists of a collection of Matrix factorization routines commonly used in engineering and scientific applications.

5.1 LU-Factorization

Let $\mathbf{A}\vec{x} = \vec{b}$ denote a dense system of linear equations, where $\mathbf{A}$ is a n-by-n matrix and $\vec{b}$, $\vec{x}$ are vectors of dimension n. One method to find $\vec{x}$ is to factorize $\mathbf{A}$ into a unit lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, solve $\mathbf{L}\vec{y} = \vec{b}$ for $\vec{y}$ and then solve $\mathbf{U}\vec{x} = \vec{y}$ to get $\vec{x}$. The computational complexity of the substitution steps is $O(n^2)$, while that of the factorization steps is $O(\frac{2}{3}n^3)$. Therefore, it is worthwhile to parallelize decomposition of $\mathbf{A}$. On distributed and shared memory MIMD machines with a hierarchical memory structure, blocked algorithms increase the number of computations per memory access and are one of the most efficient ways to improve performance [36, 14]. The benchmark suite contains the jik-SDOT, jki-GAXPY, and kji-SAXPY blocked algorithms for $\mathbf{LU}$ factorization which are suited to the column oriented Fortran. These algorithms are based on the efficient jik-noblock algorithm used to factorize a single matrix block [36].

5.1.1 Parallel Blocked jki-GAXPY

The sequential jki-GAXPY algorithm computes a block column of both matrices $\mathbf{L}$ and $\mathbf{U}$ at the j^{th} step of the elimination. The following operations are required (compare with Figure 3):

0. Initialize: Start with first block. $j \leftarrow 1$

1. Pivot and Update $U_2^{(j)}$: Apply previous interchanges to the block $U_2^{(j)}$.

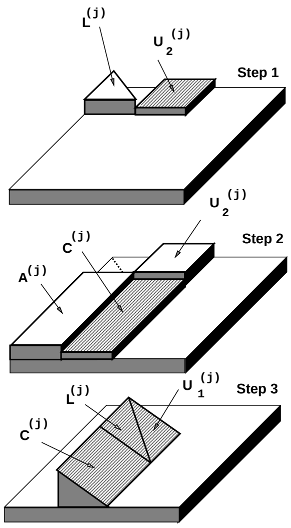


Figure 3: *jki*-GAXPY

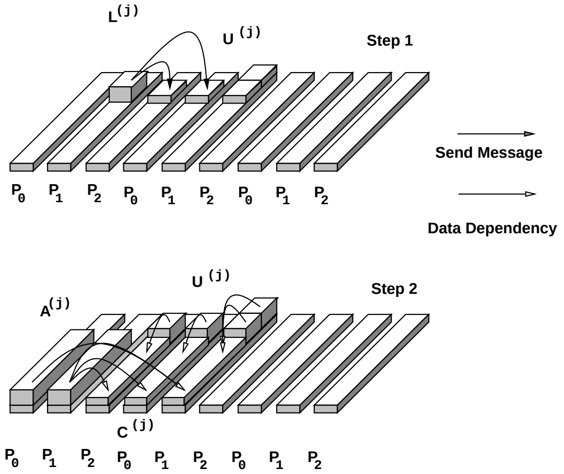


Figure 4: Parallel LU GAXPY a) update U_k, b) update C_k

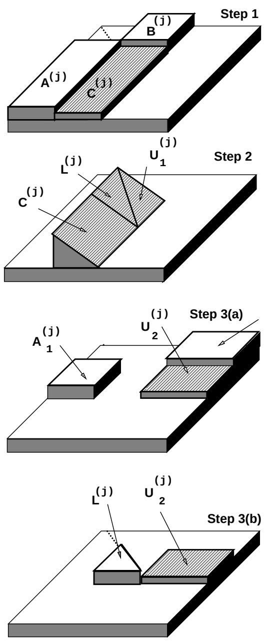


Figure 5: *jik*-SDOT

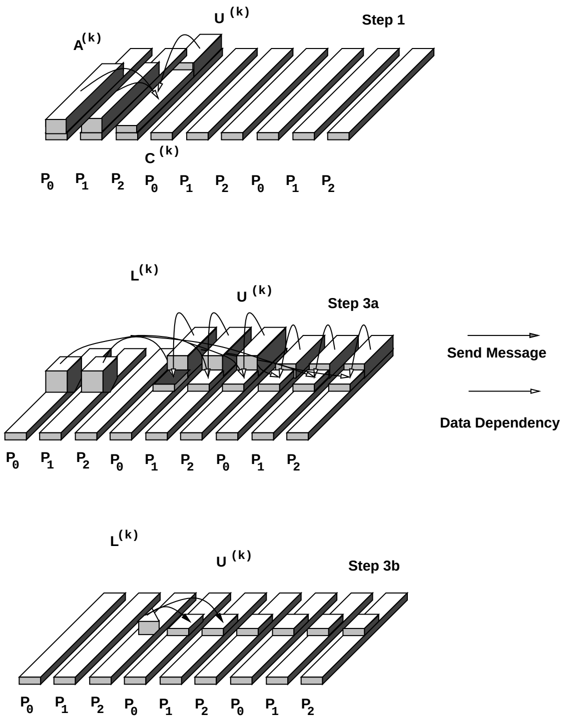


Figure 6: Parallel LU SDOT

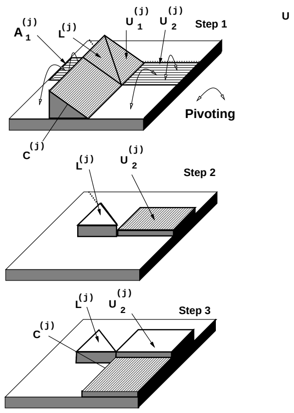


Figure 7: kji-SAXPY

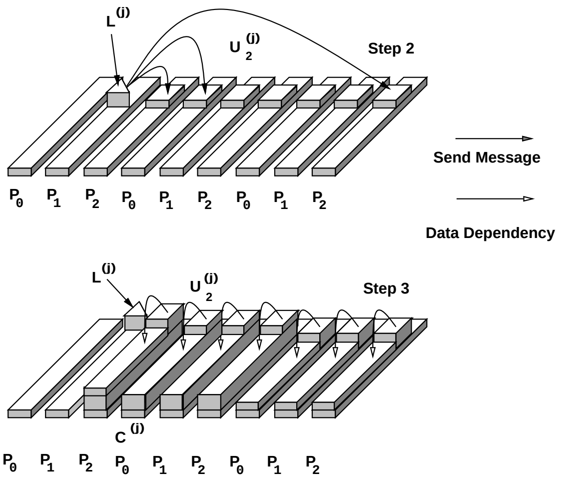


Figure 8: Parallel LU SAXPY

The j^{th} superdiagonal block of U is computed:
$U_2^{(j)} \leftarrow \left(L^{(j)}\right)^{-1} U_2^{(j)}$

2. Update $C^{(j)}$: The j^{th} diagonal and subdiagonal blocks of C are computed:
$C^{(j)} \leftarrow C^{(j)} - A^{(j)} U_2^{(j)}$

3. Factorize $C^{(j)}$: The j^{th} block column is factorized into **LU** factors using a noblock algorithm (jik-noblock).

4. Iterate: IF no more blocks THEN stop ELSE GOTO Step 1.

Because the sequential algorithm updates only one block at a time, a corresponding parallel version of this algorithm should be restricted to this block. Nevertheless, to be efficient the parallel algorithm works in a pipelined fashioned way on different processors. Therefore, the data is distributed in a block cyclic manner over the processors as shown in Figure 4. In this and the following figures data dependencies are expressed by the height of the matrix element in the figure. If a datum is higher than another one, then this datum has to be calculated first. Information is exchanged between the processors after the factorization step is completed. Looking at the data dependencies of the sequential algorithm it is clear that in order to update the matrix each processor has to know the lower triangular matrix L computed so far and $A^{(j)}$ even if they are generated in another processor. Hence, each processor has to store a complete copy of the factorized matrix.

One difference between the sequential and the parallel algorithm is that in the parallel algorithm the computation of C is distributed over time. In case of n processors, $n-1$ intermediate results are needed to complete the computation of C in each processor. Only one processor does the factorization at a time. This processor also updates more parts of the submatrix U than the other processors, which only updates one part. In later steps more time is spent on updating than on factorization which makes the algorithm efficient.

The disadvantage of this algorithm is clearly the need for storing the matrix in each processor. This data redundancy limits the use of this algorithm to small matrices. The advantage of the algorithm is that it is fast due to the data redundancy and the pipelined execution. To allow even bigger matrices to be calculated on a parallel machine with restricted memory capacity, the SDOT and SAXPY based algorithms are useful.

5.1.2 Parallel Blocked jik-SDOT

In the blocked jik-SDOT [9] algorithm, one block column of **L** and one block row of **U** are computed in each iteration. The basic steps involved in the j^{th} iteration are shown in Figure 5 along with the data dependencies involved in each step. In the j^{th} iteration, the j^{th} block depends on all of the $j-1$ previously factorized blocks.

0. Initialize Start with the first block $j \leftarrow 1$

1. Update $C^{(j)}$ The diagonal and subdiagonal blocks of the j^{th} block column are computed using SGEMM:

$C^{(j)} \leftarrow C^{(j)} - A^{(j)}B^{(j)}$

2. Factorize $C^{(j)}$ and Pivot The j^{th} block column, $C^{(j)}$ is factorized into **LU** factors using the jik-noblock algorithm.

The row interchanges are applied to blocks on both sides of the current block.

3. Update $U_2^{(j)}$ a) The j^{th} block row of **U** is updated (SGEMM):

$U_2^{(j)} \leftarrow U_2^{(j)} - A^{(j)}E^{(j)}$

b) The j^{th} block row of **U** is computed (STRSM):

$U_2^{(j)} \leftarrow \left(L^{(j)}\right)^{-1} U_2^{(j)}$

4. Iterate: IF no more blocks remaining THEN stop ELSE goto Step 1.

With the matrix laid out onto the processor (as shown in Figure 6) in a manner similar to that presented in the parallel GAXPY algorithm, this dependency requires that the factorized matrix is stored in each processor. As a consequence, the size of the matrix which can be factorized by this algorithm is limited by the available memory at each node. This limitation can be overcome to a certain extent by observing that in each iteration, the block to be factorized depends only on the portion of the factorized submatrix which includes and which is located below the current block row. In the implementation the factorized submatrix is stored in a work array. At the beginning of each iteration, the work array is reshaped so as to retain only that portion of the factorized submatrix required for subsequent computations, thereby overcoming the memory limitation.

The structure of the algorithm requires that the blocks of the matrix are factorized in a sequential order. A pipelined approach is used to avoid this inherent bottleneck [10]. In this approach, the iterations of the algorithm are pipelined so as to overlap the factorization of the j^{th} block column (steps 1 and 2) with the update of the block row associated with the $j-1^{th}$ block column (step 3). Figure 6 shows the layout of the matrix onto the processor along with the operations in the third iteration. The activities of each of the processors in the pipelined algorithm are shown in Figure 9 for a four processor system.

Although, using the pipelined implementation did provide some performance improvement for large matrices when compared with the non pipelined version, this improvement was very limited. The reason being that the amount of work involved in updating and computing the block row, i.e. step 3 is small compared to work required to update and factorize the subdiagonal block which is only done on one processor at a time. This unbalance of work along with the overhead involved in reshaping the work matrix in each iteration prevents the pipeline from remaining full and limits the improvement in performance that can be obtained.

5.1.3 Parallel Blocked kji-SAXPY

In the j^{th} step, one block column of **L** and one block row of **U** are computed and the corresponding transformations are applied to the remaining sub-matrix. The basic steps and data dependencies involved in the j^{th} iteration are shown in Figure 7. In the j^{th} iteration, the j^{th} block depends on $j-1^{th}$ factorized block:

0. Initialize Start with the first block

$j \leftarrow 1$

1. Factorize $C^{(j)}$ The j^{th} block column is factorized into **LU** factors, $L^{(j)}$ & $U_1^{(j)}$, using the jik-noblock algorithm (SGETF2).

The row interchanges (pivoting) are applied to blocks on both sides of the current block.

2. Update $U_2^{(j)}$ The j^{th} block row of **U** is computed using STRSM:

$$U_2^{(j)} \leftarrow \left(L^{(j)}\right)^{-1} U_2^{(j)}$$

3. Update $C^{(j)}$ Updating the remaining matrix using a block outer product (SGEMM):

$$C^{(j)} \leftarrow C^{(j)} - L^{(j)} U_2^{(j)}$$

4. Iterate: IF no more blocks remaining THEN stop ELSE goto Step 1.

Figure 8 shows the layout of the matrix onto the processing nodes. Like in the SDOT algorithm a pipelined approach is used.

Figure 8 shows the operation of the parallel blocked algorithm in the second iteration. Here processor 1 has factorized its panel and has shipped the factorized panel to the other processors. Now in step 2, each processor uses the value of $L^{(j)}$ it received from the current processor to update its portion of $U_2^{(j)}$. In step 3, the values of $L^{(j)}$ and $U_2^{(j)}$ are used to update $C^{(j)}$, that is the sub-diagonal sub-matrix. The activities of each of the processors during the iterations of the pipelined

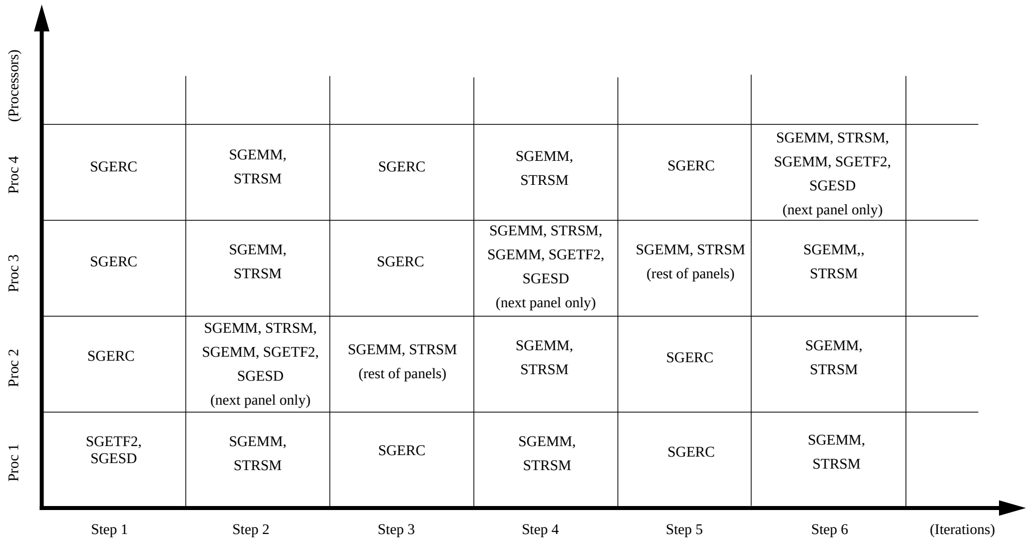


Figure 9: Spacetime diagram for the pipelined jik-SDOT algorithm (LU)

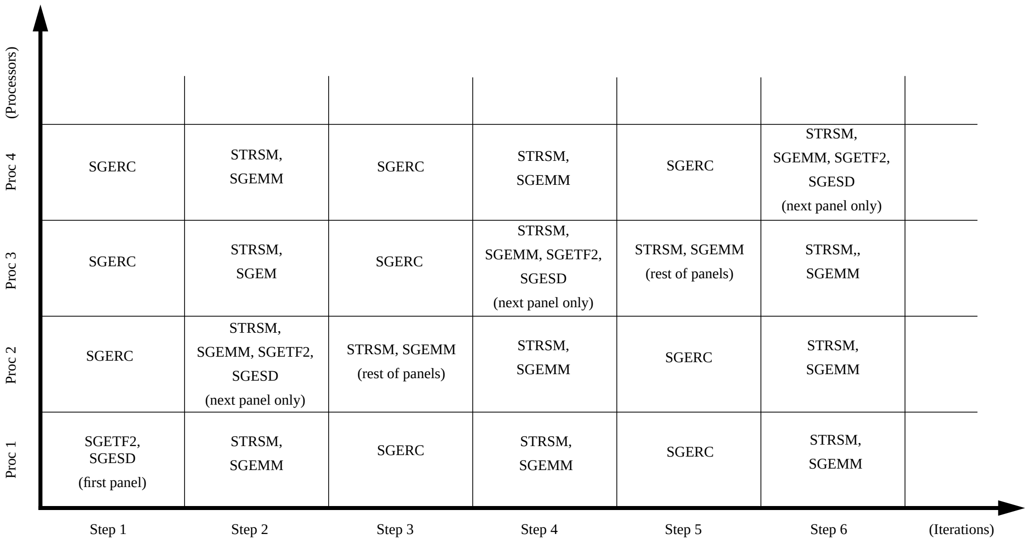


Figure 10: Spacetime diagram for the pipelined kji-SAXPY algorithm (LU)

algorithm for a four processor system are shown in the form of a spacetime diagram in Figure 10. A more detailed description of the algorithms can be found in [14].

Results

DM-MIMD Architecture The two upper diagrams in Figure 11 show the results of the three algorithms on the iPSC/860 of dimension 4. The GAXPY algorithms delivers the best performance for small matrices. Close to performance of the GAXPY algorithm is the perfomance of the SAXPY algorithm The SDOT algorithm produces very little improvement over the noblock algorithm due to the overhead of reshaping and the data dependencies. Optimal block sizes for the parallel GAXPY and SAXPY were found to be in the 8±4 range with decreasing performance for very small block sizes due to increased communication, and for very large block sizes due to poor load balancing. However, in the case of the parallel SDOT there exists a slight increase in the performance with increasing block sizes due to the decrease in the number of reshapes required. The two bottom diagrams of Figure 11 show the scalability of the SAXPY and SDOT algorithms with matrix sizes.

SIMD Architecture Experimentation with the three algorithms on the CM2 and DECmpp 12000 has shown that the blocked algorithms are not suited to SIMD architectures. On the DECmpp 12000, the three algorithms performed poorly compared to the noblock algorithm with unit block size (Figures 12 and 13). The main reason for this is the mapping of the blocks onto the processor grid. The static mapping provided by the DECmpp leads to a an inefficient utilization of the machine. Similar results were also observed on the CM2. Blocking on the CM2 leads to poor VP (Virtual Processing) ratios which in turn leads to poor performance. Thus, the blocked algorithms perform poorly on a SIMD machine since they do not make use of the available processors. The main reason why pivoting and updating in MIMD machines is split into blocks is that there are limited processors available. On a SIMD machine like the connection machine or the DECmpp the pivoting and updating can be done in parallel on the whole matrix instead of a single matrix block. Another motivation for blocked algorithms on MIMD machines is the hierarchical memory structure. This does not apply to SIMD machines. The matrix is stored directly into the main memory of the processors. The hardware is responsible for a high bandwidth between external memory and the memory of the processors. Therefore, it is not necessary to use blocked algorithm to achieve good performance on a SIMD machine.

Conclusion

This application describes the Fortran-oriented methods for block **LU** factorization. These methods are also applicable on shared memory parallel vector computers [36]. The numerical results and performance comparisons show the following:

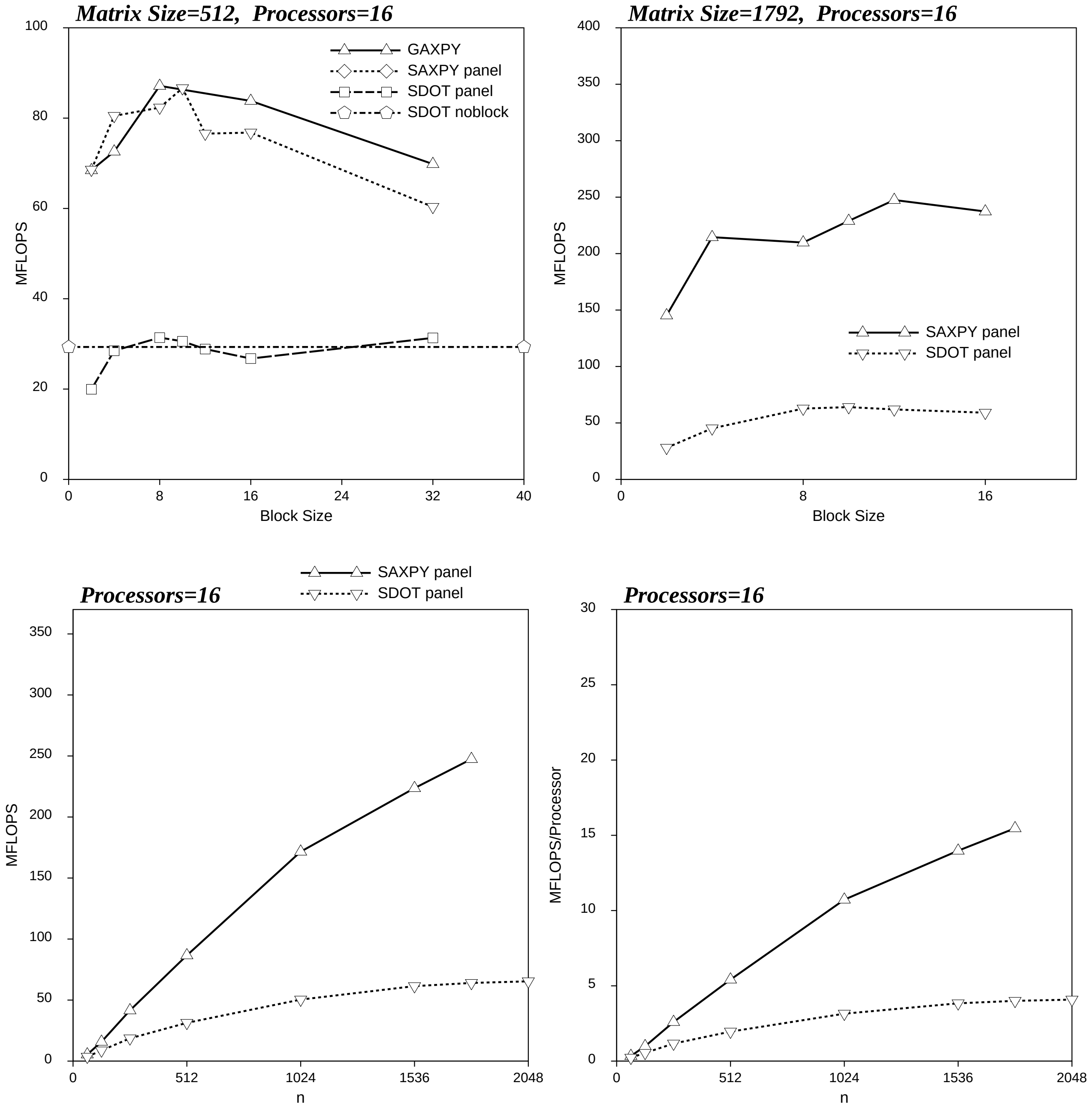


Figure 11: Performance of the different LU algorithms on the iPSC/860.

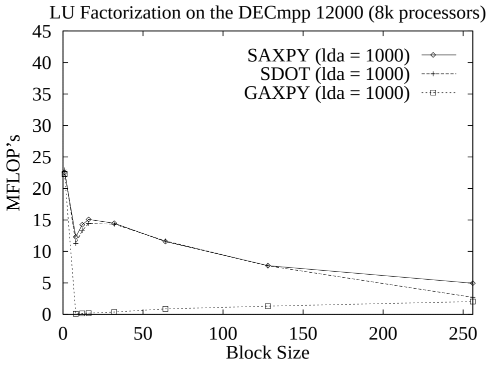


Figure 12: Performance of the different LU algorithms on the DECmpp 12000.

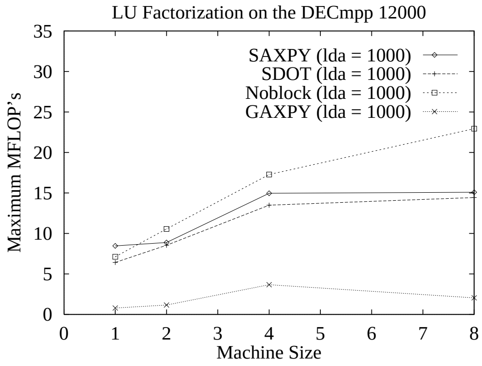


Figure 13: Performance of the different LU algorithms on the DECmpp 12000.

GAXPY

- The parallel GAXPY algorithm is very fast for small matrices.
- Because of the need to store the matrix in each node, the memory capacity of the node limits the maximal problem size for this algorithm.

SDOT

- With the help of reshaping, the paneled version of the parallel SDOT algorithm is able to factorize large matrices.
- Since the reshaping is time insensitive, this algorithm has the worst performance of the three studied.

SAXPY

- The data dependencies inherent in the parallel SAXPY algorithm are most suited for distributed memory MIMD architectures.
- No reshaping of the matrix is necessary since only a small portion of the factorized matrix has to be stored in each processor.
- Parallel SAXPY provides an efficient algorithm which scales effectively with the matrix size and can be used with a wide of number of processor.

Furthermore, the best performance is achieved at block sizes where the computation at each node outweighs the tradeoff between high load balancing (small block sizes) and low communication overhead (large block sizes). This optimal block size is dependent on the algorithm used, the size of the matrix and the number of processors available.
The three algorithms performed very poorly on SIMD architectures due the mapping of the blocks onto the machine. On the CM2, blocking minimizes the VP (virtual processing) ratio, leading to degraded performance. On the DECmpp 12000, the layout of the blocks is static and the

software does not support dynamic mapping. This leads to an inefficient utilization of the available processors.

We make the following recommendations:

- The parallel GAXPY algorithm should be used in case of small matrices and few available processors.
- The parallel SAXPY algorithm should be used for larger matrices or if the matrix size varies over a wide range and the number of processors is variable.
- The block size should be chosen depending on the algorithm used, the size of the matrix, and the number of processors used, so as to maximize performance.
- The algorithms are not suited for SIMD machines since blocked algorithms for matrices smaller than the available number of processors lead to a non efficient utilization of the available processors.

The Language Aspect As expected writing the algorithms in Fortran90-D and Fortran77-D was a mayor help for developing the code. Not only that the programs became much shorter but also more readable. The expectations concerning the time spend for developing or rewriting the Fortran90 and 77-D code has been drastically smaller than for the Fortran77+MP code.

It might be useful to specify a *pipeline* language construct to support the specific needs for the MIMD machines in matrix algebra computations. Also there is a need for providing BLAS routines in Fortran90 and Fortran90-D. There are two ways to do so: a) with a library, b) with an automatic translation tool [48]. Reshaping of arrays should be avoided since they are very cost-intensive.

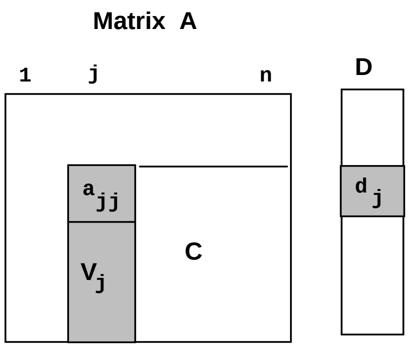
Since the blocked algorithms are not suitable for MIMD machines it gives a hint that there exists algorithms which are very difficult to parallelize on different target machines with the same success in the performance.

5.2 QR Factorization

QR method is one of LAPACK factorization algorithms. QR method is more stable than LU factorization and more general than the Cholesky decomposition, introduced later. QR is best suited for ill conditioned matrices if they are not truly singular. The sequential QR method uses a columnwise annihilation process to achieve a triangularization of the input matrix A. A series of *Householder reductions* applied to the matrix A, leads to a matrix of the form $A = QR$. In this matrix Q is an upper and R is a lower triangular matrix. The complexity of the QR factorization is $O(\frac{4}{3}n^3)$[19]. Thus the algorithm is approximately two times slower than LU factorization.

This algorithm uses only matrix-vector multiplications and outer product updates. These are BLAS level 2 operations. Figure 14 shows the noblock QR Factorization algorithm used as basis

for the blocked algorithm. The blocked QR algorithm (Figure 15), is obtained by reformulating the algorithm with the help of matrix blocks and matrix-matrix operations (level 3 BLAS).
Like in the LU factorization algorithm the blocked algorithm, as shown in Figure 16, makes efficiently use of the hierarchical memory and limited number of processors. Thus the ratio between computation and communication is increased. The communication scheme between the processors in a DM-MIMD architecture is a ring structure and the data is distributed in block cyclic way over the processors as in the parallel LU algorithms.
First, the non-block QR algorithm factories in its first processor the first block to obtain a householder submatrix. The resulting Householder submatrix is submitted to the neighbor processors and the next processor will update and factorize its matrix block. In the meanwhile all other processors update their matrix blocks. This process is repeated till the final QR factorization is obtained in one of the processors. If in each iteration the communication time is close to the computation time for updating and factorizing, then we get the best performance because the idle times in the pipeline are reduced.



Loop $j = 1, n$

$$b = -\mathrm{sign}(a_{jj})\ |V_j|$$

$$d_j = (b - a_{jj})/b$$

$$V_j = V_j / (a_{jj} - b)$$

$$a_{jj} = b$$

$$C = C - d_j * V_j * V_j^T * C$$

Figure 14: Non-block QR Factorization Diagram

Results of the QR method

Like with the LU factorization algorithms the effort spend in writing the Fortran90 code was very short. The time for writing the Fortran77+MP was considerably longer.
Figure 18 displays for a Hypercube with 32 processors the performance obtained with different block sizes. Figure 17 shows the optimal performance (based on optimal block size) for different matrix size and number of processors. For larger matrices the optimal block size is 12. Figure 19 and 20 show similar results obtained for the DECmpp. The optimal block size for this machine is 64 for larger and 1 for smaller matrices.

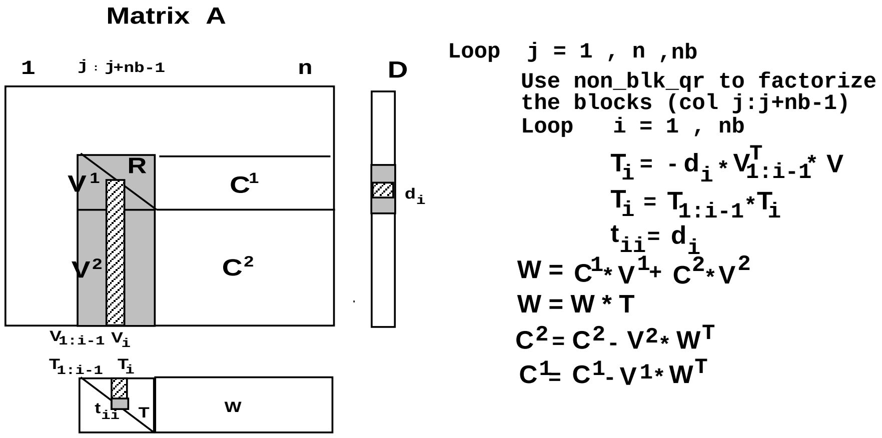


Figure 15: Block QR Factorization Diagram

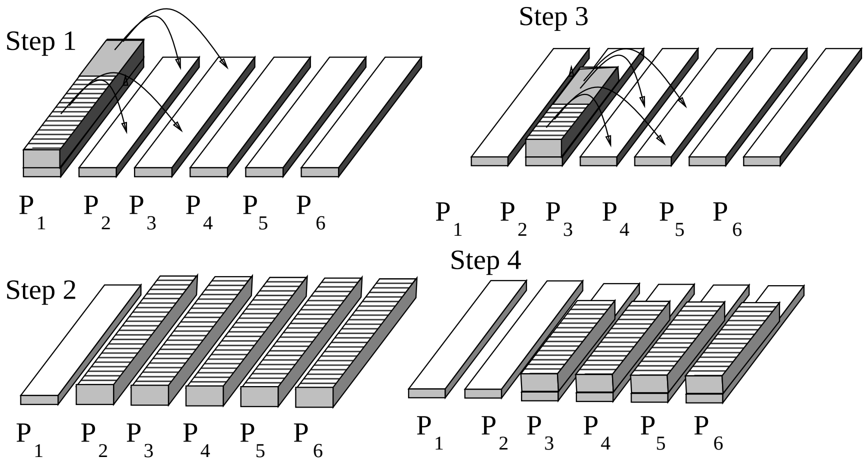


Figure 16: The first four steps of the Parallel Blocked QR Factorization

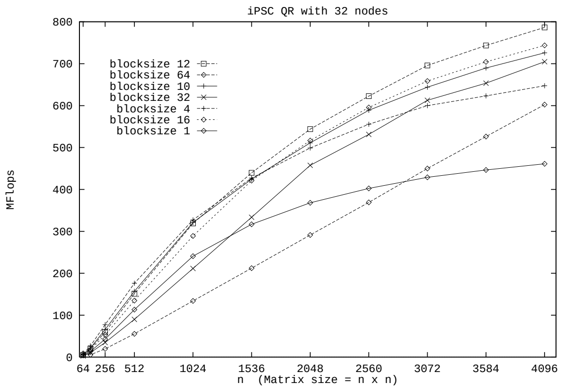


Figure 17: Optimal MFLOPS of QR for different matrix sizes on iPSC/860

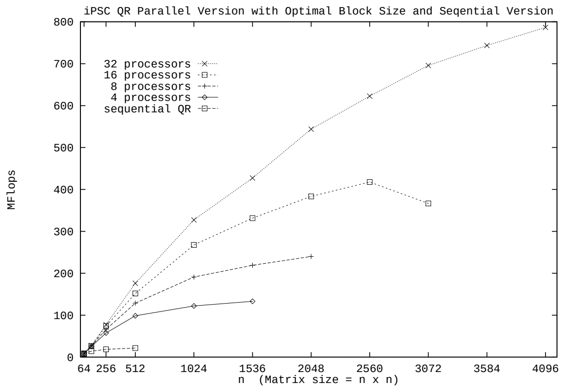


Figure 18: Performance of QR for different block and matrix sizes on a 32 node iPSC/860

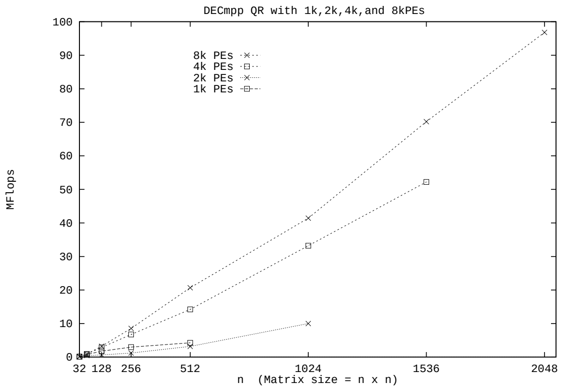


Figure 19: Performances of QR on DECmpp-12000 for the optimal block size

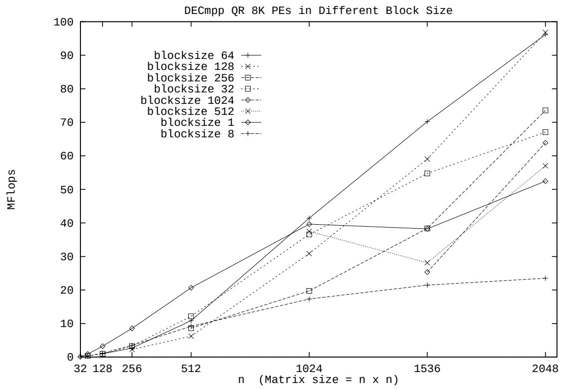


Figure 20: Performances of QR on DECmpp-12000 for 8K processors and different block sizes

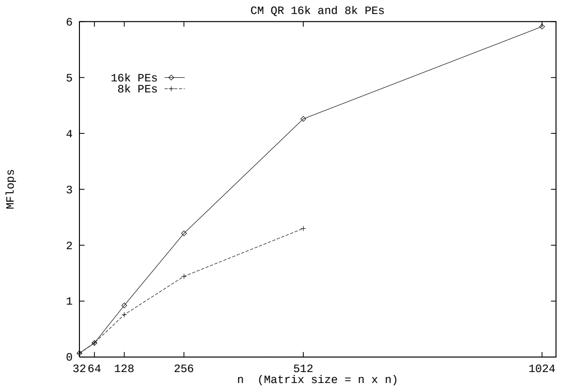


Figure 21: Performances of QR on the CM2

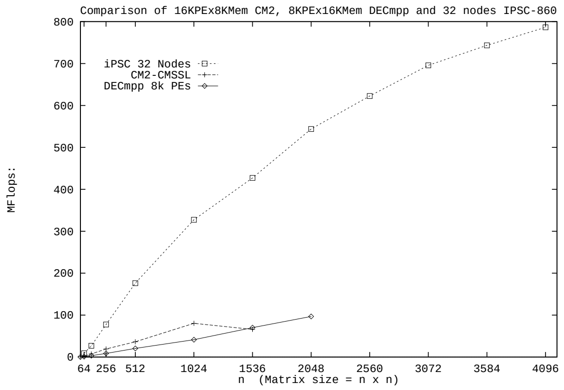


Figure 22: Comparison of QR among iPSC/860, DECmpp and CM2

The Figure 21 shows the poor performance obtained for the CM2 using the parallel blocked algorithms. The reasons for that are the same as for the parallel LU factorization algorithms. Figure 22 compares the performance among iPSC/860, DECmpp-12000 and the CMSSL library function call (written in assembly language) for the QR method on the CM2.
The higher performance on the iPSC/860 in comparison to the DECmpp-12000 and CM2 shows that the distributed MIMD computer systems can be very efficient in solving matrix algebra problems. Thus it is desirable to parallelize block based algorithms on distributed memory MIMD architectures.

5.3 Cholesky Factorization

An important special case of linear systems $Ax = b$ occurs when A is both symmetric and positive definite. One efficient method to solve this special systems is to apply Cholesky factorization to yield $A = LL^T$, where L is a lower triangular matrix with positive diagonal elements. Since a symmetric positive definite matrix has a "weighty"diagonal, pivoting is not necessary for the numerical stability of the factorization process. Because of the symmetry of A only half of arithmetic operations of the LU factorization are necessary. Readers may consult [18, 19] for more information about symmetric positive definite matrices and the Cholesky factorization.

Algorithms and Implementations

In literature, the forms of Cholesky decomposition are often categorized into three classes as shown in Figure 23: Row-, Column- and Submatrix-Cholesky factorization[19, 17]. Since Fortran is column-oriented only the two latter forms are considered here:

Column-Cholesky (GAXPY-Cholesky) Each column of the Cholesky factor L is computed one by one using the previously factorized columns. The matrix-vector multiplication is the basic operation in this version, hence it is also called GAXPY-Cholesky.

Submatrix-Cholesky (SAXPY-Cholesky) Outer product (rank-1) updating are repeated on the remaining submatrix with the "currently" factorized column. The size of the submatrix is decreasing from iteration to iteration. The idea is similar to the parallel SAXPY-LU factorization, so we call it SAXPY-Cholesky.

The effort spent in rewriting the code in Fortran90 is modest as for the LU and Cholesky factorization algorithms. The Fortran77 BLAS routines have to be substituted by the appropriate Fortran90 vector operations and/or standard functions. Table 4 shows the translation of BLAS routines to correspond Fortran90 routines.
For DM-MIMD machines, the implementation is not straightforward and pipelining is introduced to get a better performance. Like the SAXPY-LU factorization, the SAXPY-Cholesky method has fewer data dependencies, resulting in shorter messages and better utilization. The Reader may consult [10] and [17] for details.

Results and Conclusion for the Cholesky Factorization

For the CM-2 and DECmmp-12000 the noblock SAXPY-Cholesky is superior to or approximately equal to the noblock GAXPY-Cholesky at problem sizes smaller than 256 using different numbers of processors. Hence, the noblock SAXPY is used as basis of the blocked Cholesky algorithms.
The SAXPY-Cholesky has a better performance than the GAXPY-Cholesky on the CM-2. In contrast, the GAXPY-Cholesky is superior to SAXPY-Cholesky on DECmmp-12000 (Figures 25 and 26). The best block size is 32 for 1k, 64 for 4k, and 128 for 8k processors on DECmmp-12000 and 128 for both 8k and 16k processor on CM-2. Note that the best block sizes are very sensitive to the physical hardware architectures.
On the iPSC/860 24 is the suitable block size for larger matrix sizes (Figure 29). At this block size the tradeoff between communication overhead and load balance is minimal. Further observation from the experimentation with Cholesky factorization shows:

- Blocked based algorithms are better than their noblock counterparts on both SIMD and DM-MIMD architectures.
- DM-MIMD architectures could give more promising performance than the SIMD architectures.
- Automatic translation of BLAS routines from Fortran77 to Fortran90 and Fortran90-D is expected.

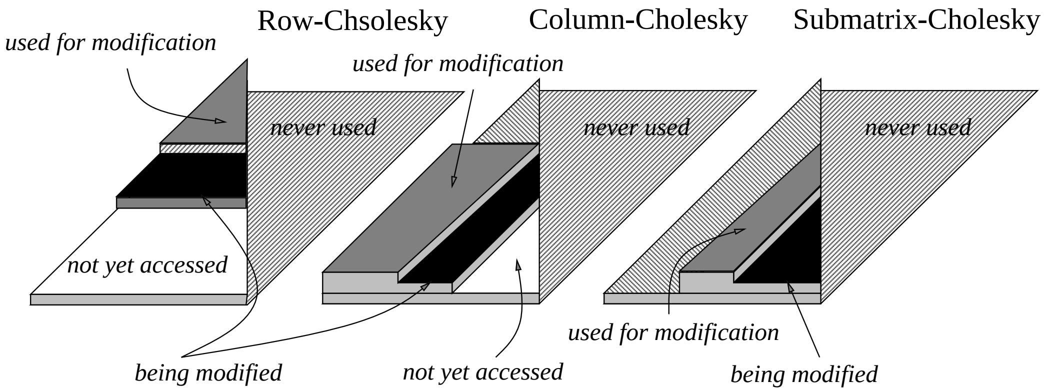


Figure 23: Three forms of Cholesky decomposition

Table 4: Translation of Fortran77 BLAS routines to Fortran90 BLAS routines

Algorithms	BLAS	Fortran90 facilities	
		Vector Operations	Intrinsic Functions
Noblock SAXPY	SSCAL	X	/
	SSYRK	X	spread
Noblock GAXPY	SDOT		dot-product
	SGEMV	X	matmul
	SSCAL	X	/
Blocked SAXPY	STRSM	X	spread
	SSYRK	X	matmul, transpose
Blocked GAXPY	SSYRK	X	matmul, transpose
	SGEMM	X	matmul, transpose
	STRSM	X	spread

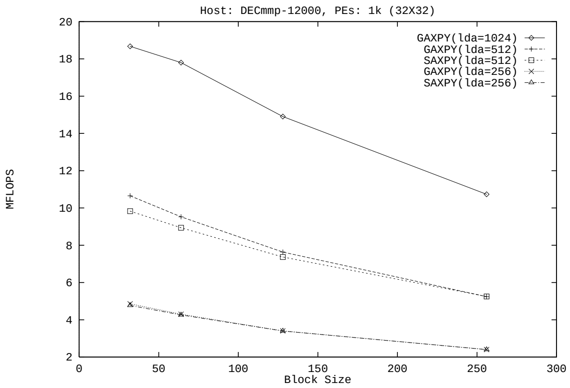


Figure 24: Performance of SIMD Cholesky factorization on DECmmp-12000

The matrix algorithms are inherently sequential with high data dependency. A pipelined approach is an efficient way to parallelize these algorithms. It addresses the need for a *pipeline* directive for efficient performance on DM-MIMD platforms. All algorithms perform *poorly* on SIMD platforms, even with no blocking and the use of vendors supplied matrix multiplycation. This addresses the need for more efficient Fortran90 compilers (even assembly coded, vendor supplied routines are very poor) and more *constructs* to be added.

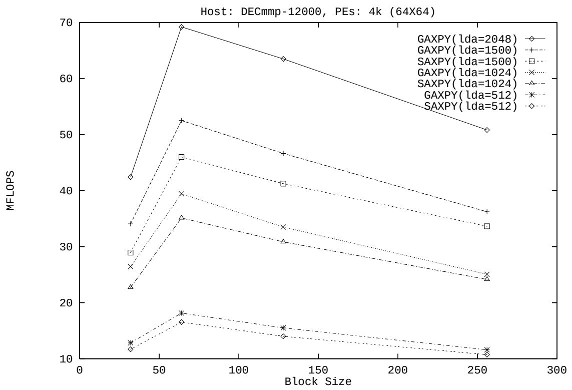


Figure 25: Performance of SIMD Cholesky factorization on DECmmp-12000

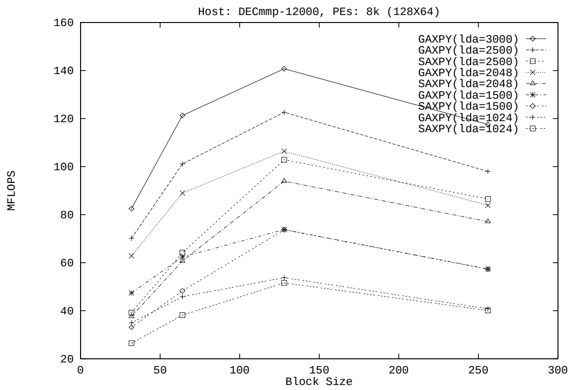


Figure 26: Performance of SIMD Cholesky factorization on DECmmp-12000

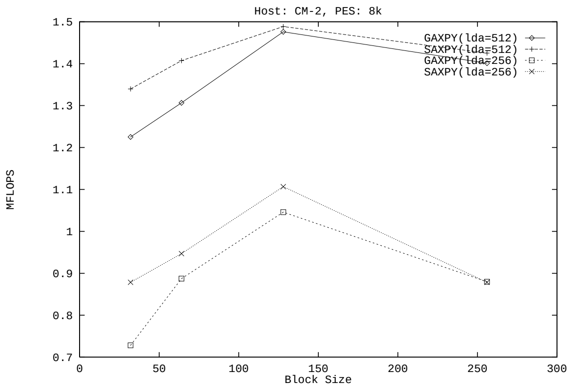


Figure 27: Performance of SIMD Cholesky factorization on CM-2

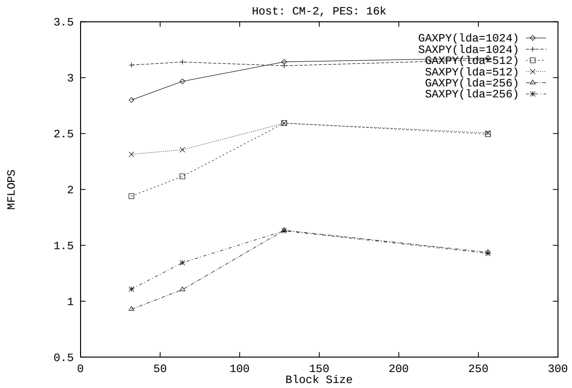


Figure 28: Performance of SIMD Cholesky factorization on CM-2

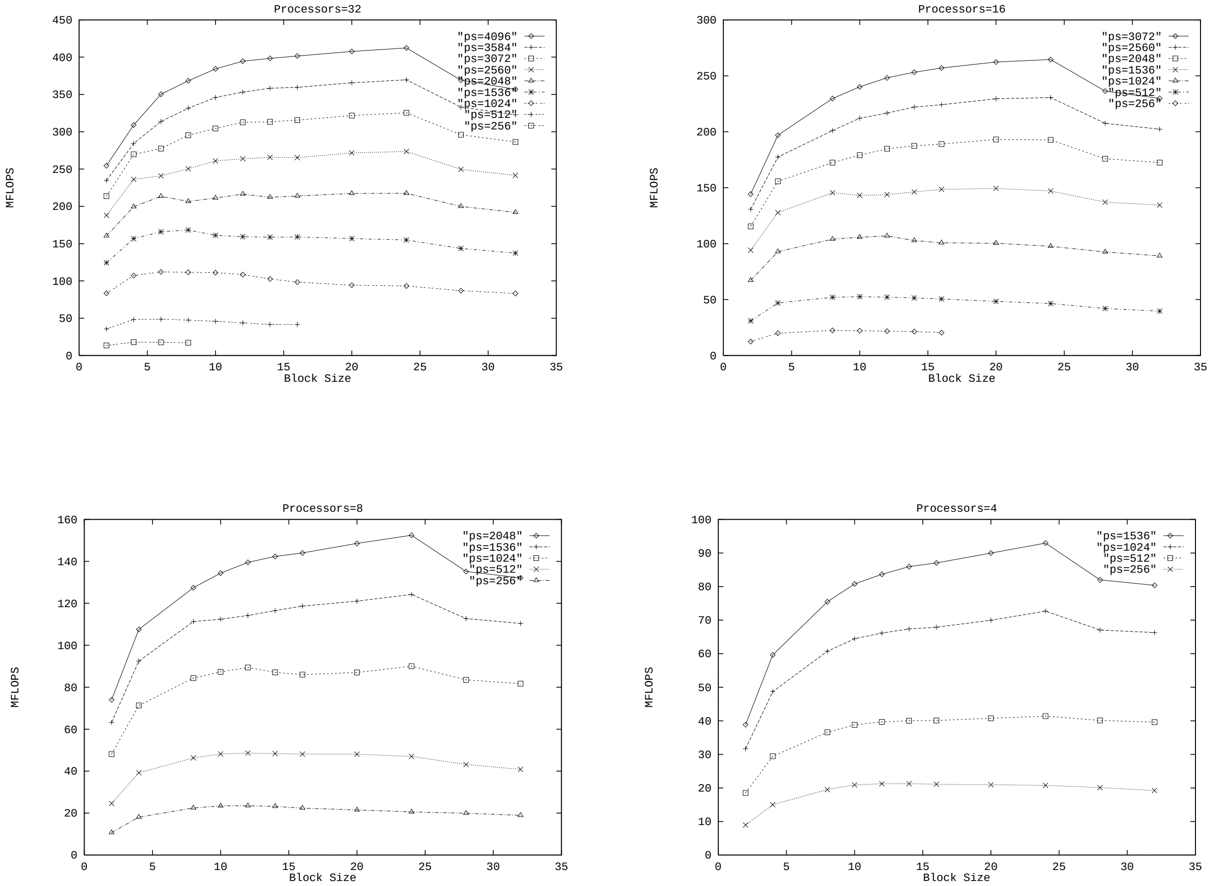


Figure 29: Performance of MIMD Cholesky factorization on iPSC/860, where ps denotes the problem size

6 ELECTROMAGNETIC SCATTERING FROM CONDUCTING BODY

This application studies the problem of electromagnetic scattering from a plane conductor containing multiple apertures. The apertures are terminated or interconnected by a microwave network, internal to the plane, through two waveguides. Electromagnetic field (EM) imitation is a widely encountered problem in many practical engineering applications. Airplane Signature, Lightning protection for aircrafts, EMP (Electromagnetic Pulse) coupling, EMC (Electromagnetic Compatibility), ESD (Electrostatic Discharge), integrated circuit design, electromagnetic biology, and many other applications are of considerable interest [4, 27, 38]. To get a numerical solution with acceptable error on traditional computers is usually a very time consuming task. A typical EM application code runs on a workstation for days. This is mostly because of inherent bottlenecks, typically in memory access. Massively parallel machines offer a very attractive approach for such implementations.

In the study, a matrix method of solution is outlined using the method of moments that was first developed to simulate EM fields by R.F. Harrington [22]. A general procedure for formulating problems involving electromagnetic coupling through apertures in conducting bodies is surveyed in [22, 23, 24, 33]. The loaded aperture cases can be formulated in a similar way.

Figure 30 shows the structure, in which a plane wave is incident on a conducting plane containing two slots terminated by a microwave network. In region A, a plane wave is incident on a conducting plane containing two slots. The width of the two slots are a and b respectively. The distance between apertures is $2d$. In region B, two parallel plate waveguides are connected by a microwave network [Y]. The lengths of waveguides are l_1 and l_2, respectively.

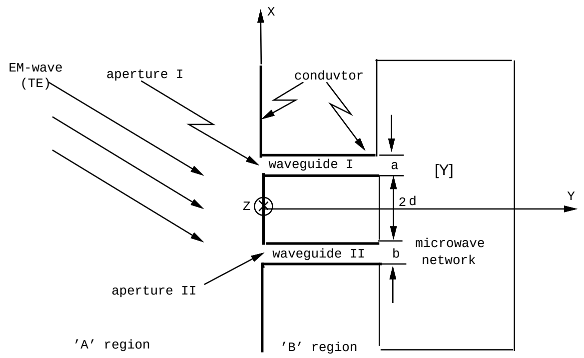


Figure 30: EM field scattering from two slots

By using the equivalence principle, the boundary condition, and the image method [21, 26, 45], we have [28]

$$-\sum_j \underline{H}_t^{ai}(\underline{M}_j) - \sum_j \underline{H}_t^{bi}(\underline{M}_j) = \underline{H}_{ti}^{sc} \tag{6}$$

$$\text{over } A_i \ (i = 1, 2)$$

For region A, we can use the image method and so, we have

$$\underline{H}_z(\underline{\rho}) = \frac{\epsilon}{4j} \sum_i \int_{A_i} 2\underline{M}_i(\underline{\rho}') H_0^{(2)}(K \mid \underline{\rho} - \underline{\rho}' \mid) \, dx \tag{7}$$

where, $\underline{\rho} = \underline{u}_x x + \underline{u}_y y$ is the field point, $\underline{\rho}' = \underline{u}_x x'$ is the source point, and A_i is the area of the i^{th} aperture.

By using the method of moments [22, 23, 32], the formulations can be interpreted into a generalized network formulation. We choose N expansion functions in each aperture and assume that the linear combination

$$\underline{M}_z^j = \sum_{n=1}^{N_j} V_n^j \underline{M}_n^j \qquad (j = 1, 2) \tag{8}$$

Using Galerkin's method with pulse basis functions, we have,

$$\{ \left[\ Y^a \ \right] + \left[\ Y^b \ \right] \} \vec{V} = \vec{I} \tag{9}$$

and $\left[\ Y^a \ \right]$ can be written as

$$\left[\ Y^a \ \right] = \begin{bmatrix} \left[\ Y_{11}^a \ \right]_{I \times I} & \left[\ Y_{12}^a \ \right]_{I \times J} \\ \left[\ Y_{21}^a \ \right]_{J \times I} & \left[\ Y_{22}^a \ \right]_{J \times J} \end{bmatrix} \tag{10}$$

where, I, J are the number of mode functions we use for the mode function expansion on apertures 1 and 2 respectively.

The elements of $\left[\ Y_{11}^a \ \right]_{I \times I}$ are

$$Y_{mn}^{a11} = \sum_{k_i=1}^{K_1} \sum_{k_j=1}^{K_1} A_{k_i,m}^1 A_{k_j,n}^1 \int_{d+(k_i-1)\Delta_1}^{d+k_i\Delta_1} dx \int_{d+(k_j-1)\Delta_1}^{d+k_j\Delta_1} \frac{K}{2\eta} H_0^{(2)}(K|x - x'|) \, dx' \tag{11}$$

where m and n are numbers of pulse functions on apertures 1 and 2 respectively.

The elements of $\left[\ Y_{12}^a \ \right]_{I \times J}$ are

$$Y_{mn}^{a12} = \sum_{k_i=1}^{K_1} \sum_{k_j=1}^{K_2} A_{k_i,m}^1 A_{k_j,n}^2 \int_{d+(k_i-1)\Delta_1}^{d+k_i\Delta_1} dx \int_{-d-b+(k_j-1)\Delta_2}^{-d-b+k_j\Delta_2} \frac{K}{2\eta} H_0^{(2)}(K|x - x'|) \, dx' \tag{12}$$

The elements of $\left[\ Y_{21}^a \ \right]_{J \times I}$ are

$$Y_{mn}^{a21} = \sum_{k_i=1}^{K_2} \sum_{k_j=1}^{K_1} A_{k_i,m}^2 A_{k_j,n}^1 \int_{-d-b+(k_i-1)\Delta_2}^{-d-b+k_i\Delta_2} dx \int_{d+(k_j-1)\Delta_1}^{d+k_j\Delta_1} \frac{K}{2\eta} H_0^{(2)}(K|x - x'|) \, dx' \tag{13}$$

The elements of $\left[\ Y_{22}^{a}\ \right]_{J\times J}$ are

$$Y_{mn}^{a22} = \sum_{k_i=1}^{K_2}\sum_{k_j=1}^{K_2} A_{k_i,m}^2 A_{k_j,n}^2 \int_{-d-b+(k_i-1)\Delta_2}^{-d-b+k_i\Delta_2} dx \int_{-d-b+(k_j-1)\Delta_2}^{-d-b+k_j\Delta_2} \frac{K}{2\eta} H_0^{(2)}(K|x-x'|)\, dx' \tag{14}$$

$\left[\ Y^{b}\ \right]$ is the region B admittance, which can be proved to be equal to the equivalent microwave admittance [28]. In $\left[\ Y^{b}\ \right]$, all elements are zero except for $Y_{1,I+1}^{b}$,$Y_{I+1,1}^{b}$, and the elements on the diagonal. The elements on the diagonal are

$$Y_{ii} = \frac{1}{Z_{ci}} \qquad\qquad Y_{ij} = 0\,(i \neq j)\,(i,j \geq 2) \tag{15}$$

where Z_{ci} is the characteristic impedance of the waveguide.
The four special elements are $Y_{1,1}^{b} = A_1, Y_{I+1,1}^{b} = \frac{B_1}{\Delta_2}, Y_{I+1,1}^{b} = A_2$, and $Y_{22}^{b} = \frac{B_2}{\Delta_2}$.

$$\Delta_1 = \Delta_2 = \begin{vmatrix} \cosh K_1 l_1 + Z_{c1} Y_{11}^L \sinh K_1 l_1 & Y_{12}^L Z_{c2} \sinh K_2 l_2 \\ Z_{c1} Y_{21}^L \sinh K1 l_1 & \cosh K_2 l_2 + Z_{c2} Y_{22}^L \sinh K_2 l_2 \end{vmatrix} \tag{16}$$

$$A_1 = \begin{vmatrix} \frac{1}{Z_{c1}} \sinh K_1 l_1 + Y_{11}^L\ cosh K_1 l_1 & Z_{c2} Y_{12}^L \sinh K_2 l_2 \\ Y_{21}^L \cosh K_1 l_1 & \cosh K_2 l_2 + Z_{c2} Y_{22}^L \sinh K_2 l_2 \end{vmatrix} \tag{17}$$

$$A_2 = \begin{vmatrix} \cosh K_1 l_1 + Z_{c2} Y_{11}^L \sinh K_1 l_1 & \frac{1}{Z_{c1}} \sinh K_1 l_1 + Y_{11}^L \cosh K_1 l_1 \\ Z_{c1} Y_{21}^L \sinh K_1 l_1 & Y_{21}^L \cosh K_1 l_1 \end{vmatrix} \tag{18}$$

$$B_1 = \begin{vmatrix} Y_{12}^L \cosh K_2 l_2 & Z_{c2} Y_{12}^L \sinh K_2 l_2 \\ \frac{1}{Z_{c2}} \sinh K_2 l_2 + Y_{22}^L \cosh K_2 l_2 & \cosh K_2 l_2 + Z_{c2} Y_{22}^L \sinh K_2 l_2 \end{vmatrix} \tag{19}$$

$$B_2 = \begin{vmatrix} \cosh K_1 l_1 + Z_{c1} Y_{11}^L \sinh K_1 l_1 & Y_{12}^L \cosh K_2 l_2 \\ Z_{c1} Y_{21}^L \sinh K_1 l_1 & \frac{1}{Z_{c2}} \sinh K_2 l_2 + Y_{22}^L \cosh K_2 l_2 \end{vmatrix} \tag{20}$$

$$\vec{I} = \begin{bmatrix} \vec{I}_1 \\ \vec{I}_2 \end{bmatrix}$$

and

$$I_{k_i}^1 = \sum_{k_m=1}^{K_m} 2A_{k_i,m}^1 \frac{\sin(K\Delta_1 \cos\phi_{in})}{K\Delta_1 \cos\phi_{in}}\, e^{-jK\hat{x}\cos\Phi_{in}} \tag{21}$$

$$I_{k_j}^2 = \sum_{k_m=1}^{K_m} 2A_{k_j,m}^2 \frac{\sin(K\Delta_2 \cos\phi_{in})}{K\Delta_2 \cos\phi_{in}}\, e^{-jK\hat{x}\cos\Phi_{in}} \tag{22}$$

Figure 31 presents a flow chart of the algorithm.

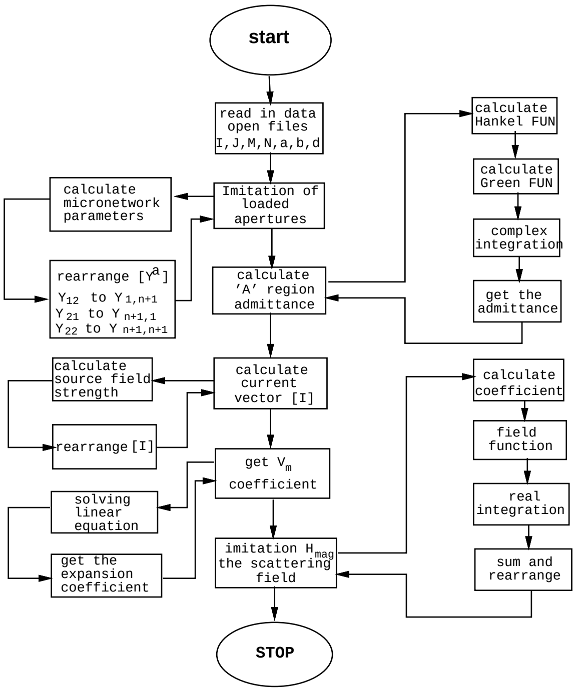


Figure 31: Flow chart for the EM scattering application program

Implementation and Results for the Electromagnetic Scattering

From the analysis of the sequential code, we know that the core calculations in the main program are the evaluations of the elements of the region A admittance matrix. This matrix comes from the Moment Method solution in which we use pulse base mode function expansion. In the expansion, the mode functions are an independent function set and the symmetric product is linear. For this reason, the Moment Method has a lot of parallel opportunities and maps naturally to SIMD machine (like CM-2 and DECmpp12000).

The basic premise of these calculations is that they can be performed in parallel since they are completely independent. For example, to calculate the elements of the region A admittance matrix for a 10x10 mode function expansion, we must use a 512x512 pulse base on the two apertures and for every pulse base we have to call the Hankel function subroutine four times. Hence, for all the elements, the Hankel function needs to be evaluated 1.048576×10^8 times. So, if we can make a rank four array, 10x10x512x512, we can have the processors calculate them simultaneously and independently. By using this on a CM-2 with 16384 processors, the Hankel function will be evaluated by each processor only 6400 times. For the other parts of the code, we used every opportunities to use the efficient assembly coded scientific subroutine library and the array syntax

that maps naturally on the CM-2 [46] and DECmpp 12000 [31].
We have developed two parallel versions from the Fortran 77 sequential EM scattering application code. One is built around the efficient use of the Connect Machine Scientific Subroutine Library (CMSSL). The second version is written in standard Fortran 90 which is machine independent and runs on both the CM-2 and the DECmpp 12000. The scattering magnetic field pattern can be found in reference [28].
For small array sizes, for example 10x10x20x20, the Fortran 90 version runs about 160 times faster on CM-2 and DECmpp-12000 than on a SPARC station. But, for larger array sizes, for example 10x10x512x512, a sequential computer like the SPARC station, requires 39 hours to calculate one pattern. The CM-2 with 16k processors however, needed only 35 seconds for the same problem while the DECmpp 12000 with 8k processors needed 58 seconds. [29, 37]. The scattering field is shown in Figure P:lu:imit.

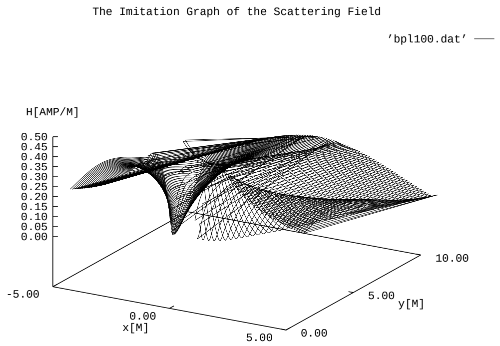


Figure 32: Imitation of the scattering magnetic field in space

Figure P:lu:perf shows the performance of the Fortran 90 code on DECmpp 12000. It shows the scalability of performance with machine size for different admittance matrix sizes. It is clear from the figure that the most effective machine size is dependent on the problem size. As seen from the figure the most effective machine size for an array size of 512 is 8K, for an array size of 256 is 4K, and for an array sizes of 128 and 64 is 2K. This due to the fact that the elapsed time decreases substantially by increasing the machine size upto the above mentioned sizes and does not show any substantial change for any further increase in the machine sizes. This is because the array size is no longer large enough to amortize the start-up overhead time for the large machine size.

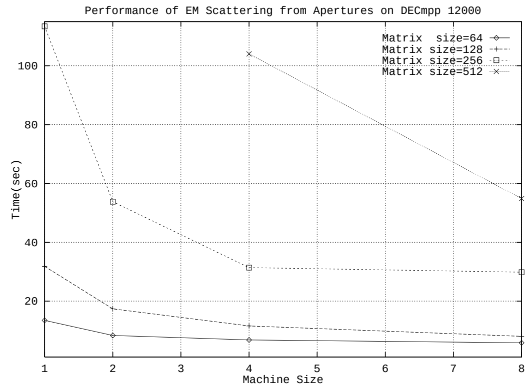


Figure 33: The performance of EM Scattering code on DECmpp 12000

7 STOCK OPTION PRICING

This application is part of NPAC's program to apply high performance computing to problems in industry, and is based on a collaboration between NPAC and the School of Management at Syracuse University. In this project, we implemented a set of stock option pricing models in Fortran90 on the Connection Machine-2 and the DECmpp-12000. In this paper, we describe a single pricing model, the binomial approximation model with American call and stochastic volatility (early exercise). Following the natural structure of the problem, our initial approach was to express the model in two-dimensional arrays. This approach requires dynamic distribution of data on the target machines. Performance of the code using dynamic arrays on the DECmpp-12000 is quite sensitive to the distribution of data on the physical processor grid. In contrast, performance of the same code on the CM-2 is not sensitive to the data distribution. In a second modeling approach, we implemented one-dimensional arrays with a static data distribution. Performance on both the CM-2 and DECmpp-12000 was similar and approached 50 times faster than a sequential Fortran77 version of the code running on a SUN4 workstation (25 MHZ). We use the pricing model as an application benchmark code to illustrate the issue of data distribution and performance on the CM-2 and DECmpp-12000.

Option pricing

Stock options are contracts giving the holder of the contract the right to buy or sell the underlying stock at some time in the future. Option contracts are traded just as stocks are traded, and option pricing models are used by traders and financial managers to guide speculative and hedging strategies in the market.

Since the opening of the first organized options exchange, and the introduction of the Black-Scholes option pricing model [3], finance researchers have sought improved methods to price options with stochastic volatility on American contracts. Key model parameters include volatility (or variance) of the underlying asset σ, variance of the volatility ξ, and correlation between asset price and volatility ρ. Following [8, 25, 11], we briefly summarize the equations describing continuous time movement of stock price and volatility (variance of stock price) over the life of an option contract. Modeling these processes is based on an assumption that stock price and volatility follow a constant drift except at times when new information enters the market and the stock jumps to a new value. Volatility, σ, and stock price, S, follow stochastic processes represented as

$$\frac{d\sigma^2}{\sigma^2} = \mu_\sigma dt + \xi d\widetilde{W} \tag{23}$$

$$\frac{dS}{S} = \mu_s dt + \sigma d\widetilde{Z} \tag{24}$$

where $\widetilde{W}$ and $\widetilde{Z}$ are standard Weiner processes with correlation ρ, μ_σ is the drift of the variance process and μ_s is the drift of stock price (both constants) and ξ is the volatility of the variance (not directly observed, but estimated from data). Weiner processes generate continuous paths that are in constant motion no matter how small the time step.

Binomial approximation models represent the continuous time processes described above as a lattice of discrete up/down movements or jumps in stock price and volatility. For example, the magnitude of the increase (u) or decrease (d) in variance for any given time period is based on new information from the market and is expressed as

$$u = e^{(\mu_\sigma - \xi^2/2)\Delta t + \xi\sqrt{\Delta t}} \tag{25}$$

$$d = e^{(\mu_\sigma - \xi^2/2)\Delta t + \xi\sqrt{\Delta t}} \tag{26}$$

The probability of an increase or decrease is equally likely, and both paths are represented in the binomial lattice.

With the introduction of correlation, ρ, the variance of stock price after i periods with j upward movements and $i - j$ downward movements is now defined as

$$\sigma^2 = \left(\sigma^2_{0,0}\right) u(\rho)^i d(\rho)^{i-j} \tag{27}$$

In the limit, as Δt approaches zero, the binomial process approaches the continuous time process

$$d\sigma^2 = \mu_\sigma \sigma^2 dt + \xi \sigma^2 d\widetilde{W} \tag{28}$$

The magnitude of increases (U) and decreases (D) within the stock price are then defined as

$$U_{i,j} = e^{r_f - \sigma(2_{i,j}/2)\Delta t + \sigma_{i,j}\Delta t} \tag{29}$$

$$D_{i,j} = e^{r_f - \sigma(2_{i,j}/2)\Delta t - \sigma_{i,j}\Delta t} \tag{30}$$

American options incorporate early exercise, which means that the option can be exercised at any time during the life of the contract. Pricing American option contracts with the binomial model requires tracking price movements within the lattice from the time of dividend payout to contract maturity. We use American pricing, but do not describe implementation of this model in this paper.

Application of Model to Market Data

We applied a binomial option pricing model incorporating stochastic volatility and American call to a set of Chicago Board of Options Exchange (CBOE) market data, for a number of stocks, for the time period January, 1988.

The binomial pricing model uses a binary tree or lattice to represent discrete up/down moves in volatility and price over time. After T time periods, 2^T stock prices are represented in the terminal period of the lattice, and a single model price is derived from this distribution. A fuller discussion of our comparison of pricing models and market data is presented in [34].

Implementation of the Pricing Model

First, we describe the structure of the binomial pricing model. The model is then implemented in two forms– two-dimensional arrays with dynamic data distribution, and one-dimensional arrays with static data distribution.

A binomial lattice is illustrated in Figure 34 showing asset price or volatility of price in the vertical axis and time in the horizontal axis. Based on a previous comparison of model sizes [34], we divide the life of an option contract into $T = 17$ periods.

Important elements of the model include initial price (S_0) and volatility (σ_0) or (V_0), time of dividend payout (t_{div}), the $2^{t_{\text{div}}}$ nodes at time of the dividend where t_{div} ranges over values 1 to $T-1$, and the 2^T nodes at terminal time T. A single option price C_0, is estimated from a weighted average of the 2^T prices at time T and discounting to the present time T_0.

We designate the time steps in our model from 1 to t_{div} as stage 1 of the model, and timesteps from t_{div} to maturity T as stage 2 of the model. Although we do not describe details here, this breakdown of the American pricing model allows us to track price movements after dividend payout and determine percentages of early exercise.

Figure 35 illustrates the $2^{t_{\text{div}}}$ nodes in the binomial lattice at time of dividend. The value of t_{div} ranges from 1 to $T-1$ and defines the shape of the two-dimensional Fortran array $(1 : 2^{t_{\text{div}}}, 1 : 2^{T-t_{\text{div}}})$. The value t_{div} comes from market information (each option record has its own value t_{div}) and is not accessible to the model until run-time, requiring dynamically allocated arrays.

At the close of stage 1 in our model, there are $2^{t_{\text{div}}}$ nodes in the lattice. After dividend payout, and the onset of stage 2 of the model, up/down movements of price (and volatility) for each node are

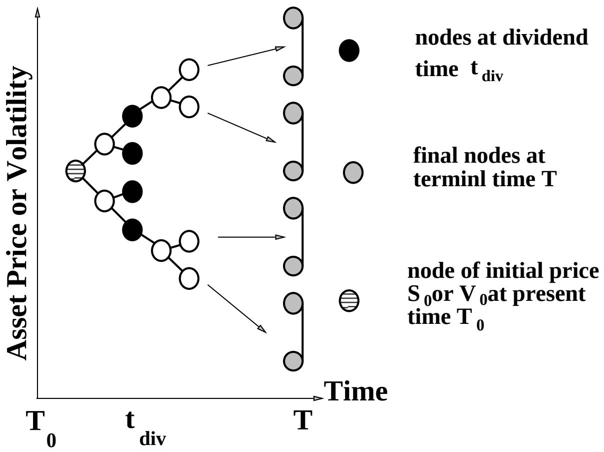


Figure 34: Two-dimensional structure of lattice

represented by a subtree of the lattice and expressed in the second dimension of the array of size $2^{T-t_{\mathrm{div}}}$. As illustrated in Figure 35, when $t_{\mathrm{div}} = 2$, there are $2^{t_{\mathrm{div}}}$ or 4 rows in the two-dimensional array. After dividend payout, stage 2 of the model, further up/down moves of price and volatility are expressed in the 2^{T-2} columns of the two-dimensional array.

Model Implementation 1: Two-Dimensional Dynamic Arrays

Following the illustration of Figure 35, we expressed the two-dimensional structure of the binomial lattice in two-dimensional Fortran arrays. This approach requires dynamic two-dimensional arrays which vary in shape according to the value of t_{div}, a value which becomes known only at run time. We observed large differences in performance between the CM-2 and DECmpp-12000 implementations of the two-dimensional array model. CM-2 performance is not sensitive to the shape of dynamically sized two-dimensional arrays. Model run time remains relatively constant for an range of t_{div} values and associated array shapes. In contrast, DECmpp-12000 performance is highly sensitive to the shape of dynamically sized two-dimensional arrays. When values of t_{div} result in asymmetric arrays, performance falls below the performance of a sequential workstation implementation of the pricing model.

Poor performance of the DECmpp with two-dimensional asymmetric arrays is due to inefficient mapping of data to the processor grid and resultant load imbalance. An examination of the data mapping patterns with t_{div} values of 1, or 16 illustrates this problem Figure 36 A, B. For asymmetric arrays, the cut and stack method of data distribution used by the DECmpp operating system [30] maps only a portion of data to the processor grid, with most of the data "wrapped" into memory. A more efficient mapping of data is shown in Figure 36 C. In this example, the value of t_{div} results in

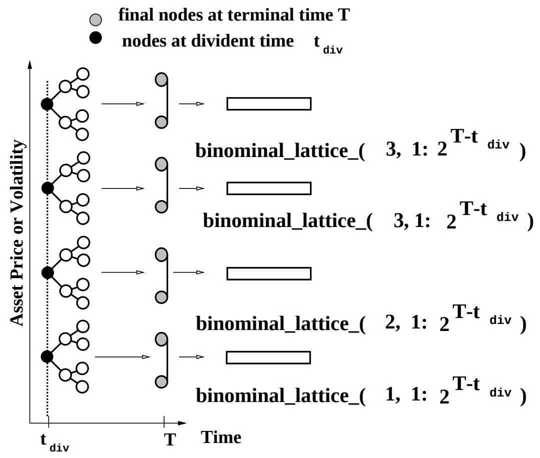


Figure 35: Binomial Lattice Expressed as Two-Dimensional Array

a symmetric two-dimensional array and the operating system efficiently maps data to the physical processor grid.

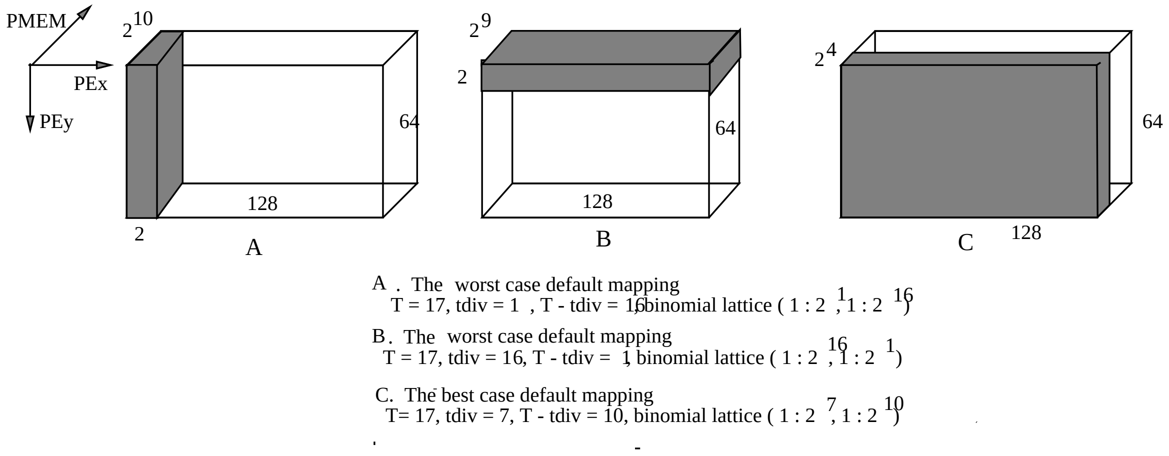


Figure 36: Data Distribution

In comparison, the CM-2 operating system arranges array elements “horizontally” (arrays of multidimensional rank are mapped as a one-dimensional array) across processors, one element per processor. This approach fills up the processor grid so as to maximize the number of physical processors in use and minimize virtual processor looping [47].

In summary, in this first model implementation, we express the movement of stock price change over time in two-dimensional Fortran arrays. The value of t_{div}, which is known only at run time, determines the shape of the two-dimensional arrays. We dynamically size arrays by passing the value of t_{div} to a subroutine. When asymmetric arrays result in this scheme of dynamic data distribution, performance is degraded on the DECmpp-12000 but remains constant on the CM-2.

Model Implementation 2: One-Dimensional Static Arrays

This application code happens to require nearest neighbor communication along only one axis. This feature allowed us to express the same binomial model described above in one-dimensional Fortran arrays with a static data distribution.

In this implementation, we represent the the 2^T nodes of the binomial lattice in a Fortran array of size $(1 : 2^T)$. Figure 37 illustrates this one-dimensional array model for $t_{\text{div}} = 2$. At the end of stage 1 of the model, just prior to dividend payout, there are $2^{t_{\text{div}}}$ nodes in the lattice. In stage 2 of the model, we map each node to a section of the array of size $2^{T-t_{\text{div}}}$ and evolve the volatility and price lattice forward in time. We use the Fortran90 intrinsic function eoshift inside a loop first to calculate, then to communicate values representing possible up/down moves in price and volatility through the array section.

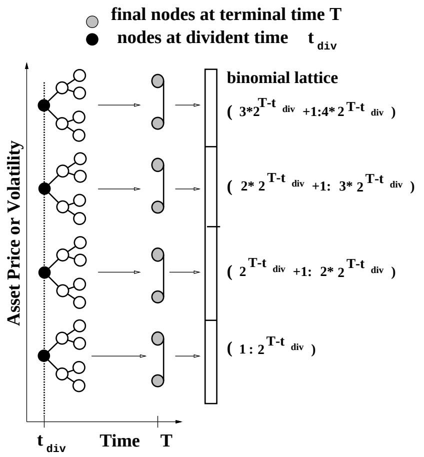


Figure 37: One-Dimensional Static Arrays

The DECmpp operating system maps *one-dimensional* arrays to the processor grid in raster-scan fashion. For a model of $T = 17$ periods, the 2^{17} one-dimensional array completely fills the 128×64 physical processors of the 8k DECmpp, then is layered into memory. By using a one-dimensional array to represent the binomial lattice, we fix the number of required layers to 2^4 for any value of t_{div}, and program performance remains constant.

Performance of the one-dimensional model implementation using static data distribution on the

CM-2 is similar both to the two-dimensional implementation described above, and the one-dimensional implementation on the DECmpp-12000.

In summary, this application code has a substantial communication requirement but happens to regular and along only one axis. This feature allowed us to express the pricing model in one-dimensional arrays with a static data distribution. A refinement of this approach, expressed through compiler directives, combines static distribution of a parallel array dimension to fill the 8K processor grid, with serial in-processor arrays to reduce communication [35].

Results

We implemented the binomial pricing model using static and dynamic data distribution and compared performance between the CM-2 and DECmpp-12000. Our first model implementation expresses the natural two-dimensional structure of a binary tree in two-dimensional Fortran arrays. This approach was based on dynamically sized arrays defined by the value of (t_{div}), which is known at run time. The DECmpp operating system inefficiently maps dynamically distributed, asymmetric two-dimensional arrays to the fixed processor grid. Performance is slower in some cases than a sequential version of the model running on a workstation. CM-2 performance is not sensitive to dynamically distributed, asymmetric, two-dimensional arrays.

Remarks on the Stock Option Pricing Application

We used option pricing as a benchmark application code to examine the issue of dynamic vs. static data distribution on the CM-2 and DECmpp-12000. This issue became apparent while performing a large scale comparison of option pricing models and historical market data, which requires parallel models and porting these codes developed on the CM-2 to the DECmpp-12000. Performance differences between the two machines pointed out the data distribution issue.

We were able to achieve similar levels of performance between the CM-2 and the DECmpp-12000 by expressing the model in one-dimensional arrays with static data distribution. This approach does not follow naturally from the application structure and requires more programming effort to implement. Our experience here points out the benefit of using real application codes to evaluate parallel language and system software design.

8 CONCLUSION

We have developed an applications test suite to test and evaluate the Fortran-D language design. Eventually this application suite may evolve into a test suite to certify a HPF compiler. We contrast this with our existing Fortran-77/90D test suite where we have coded the applications in at least six Fortran dialects. Currently, the test suite consists of 45 applications. The 23 elementary applications are being used for initial test of the Fortran-D compiler as well as a teaching aid that have been made available for researchers nationwide for programming in all existing Fortran dialects. The 22 real life applications coming from independent research on parallel algorithms in different scientific areas are also made available. Our experience here points out the benefit of using real application codes to evaluate parallel language and system software design as mentioned above for the selected applications. It has been shown that Fortran-D is highly efficient for the implementation of some real applications although more directives might be needed for some other applications.

From our experiments we can strongly say that DM-MIMD architectures are better suited for many real applications than SIMD. This adresses the need for a Fortran-D like compiler to handle the burdensome messgae passing.

Availability

To obtain a copy of the software used in this study, send a one-line e-mail message “send index” to npaclib@minerva.npac.syr.edu or anonymous ftp form minerva.npac.syr.edu. Minerva is a free software distribution electronic service. The index lists information on how to access all the programs used in this study. Users who have problems accessing these programs should send e-mail to the authors at agm@nova.npac.syr.edu.

Acknoledgement

The presented research is sponsored by DARPA under contract #DABT63-91-k-0005. The content of the information does not necessary reflect the position or the policy of the Government and no official endorsement should be inferred.

Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR-9120008 and CDA-8619893 with support from the Keck Foundation.

We would like to thank Jack Dongarra for making a preliminary version of LAPACK available to us and to Susan Ostrouchov for her help with **LU** factorization on the iPSC/860.

References

[1] Bailey, D., Barton, J., Lasinski, T., and Simon, H. The NAS Parallel Benchmarks. Tech. Rep. RNR-91-002, NAS systems Division, NASA AMes Research Center, Moffett Field, CA 94035, August 1991.

[2] Benker, S., Chapman, B., and Zima, H. Vienna Fortran 90. In *Proceedings of Scalable High Performance Computing Conference* (Williamsburg, VA, USA, 1992), p. p. 51.

[3] Black, F., and Scholes, M. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy 81* (May-June 1973), pp. 637–59.

[4] Bladel, J. V., and Butler, C. M. *Aperture Problems*. Sythoff and Noordhoff International Publishers, 1979. Ed. J Skwirsynski.

[5] Blank, T. The MasPar MP-1 Architecture. *Proceedings of the IEEE Compcon Spring 1990* (February 1990), pp. 20–24.

[6] Choudhary, A., Fox, G. C., and Mohamed, A. G. Fortran D Compiler for MIMD Machines. Tech. Rep. SCCS 275, Northeast Parallel Architectures Center, Syracuse University, 111 College Place, Room 3-201, Syracuse, NY 13244-4100, 1992.

[7] Choudhary, A., Fox, G. C., Ranka, S., Hiranandani, S., Koelbel, C., and Tseng, C.-W. Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines. Tech. Rep. SCCS 25, Northeast Parallel Architectures Center, Syracuse University, 111 College Place, Room 3-201, Syracuse, NY 13244-4100, 1992.

[8] Cox, J., Ross, S., and Rubinstein, M. Option Pricing: A Simplified Approach. *Journal of Financial Economics 7* (1979), pp. 229–63.

[9] Dayde, M. J., and Duff, I. S. Level 3 BLAS in **LU** Factorization on the CRAY-2, ETA-10P, and IBM 3090-200/VF. *The International Journal of Supercomputer Applications, Massachusetts Institute of Technology Vol. 3*, No. 2 (1989), pp. 40–70.

[10] Dongarra, J., and Ostrouchov, S. LAPACK Block Factorization Algorithms on the Intel iPSC/860. Tech. Rep. LAPACK Working Note 24, Department of Computer Science Technical Report, University of Tennessee, 1990.

[11] Finucane, T. Binomial Approximations of American Call Prices with Stochastic Volatilities. *to be published in Journal of Finance* (1992).

[12] Fox, G. C., Haupt, T., and Mohammed, A. G. Purdue Set - Problems to Test Parallel Languages. Tech. rep., in preparation, 1990.

[13] Fox, G. C., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and Wu, M.-Y. Fortran D Language Specifications. Tech. Rep. SCCS 42C, Northeast Parallel Architectures Center, Syracuse University, 111 College Place, Room 3-201, Syracuse, NY 13244-4100, 1990.

[14] Fox, G. C., Mohamed, A. G., von Lasewski, G., Parashar, M., Lin, N.-T., and Yeh, N. High Performance Scalable Matrix Algebra Algorithms for Distributed Memory Architectures. Tech. Rep. SCCS-271, Northeast Parallel Architectures Center, Syracuse University, New York, April 1992. and CRPC-TR92210, Center for Research on Parallel Computation, Rice University, Houston, Texas.

[15] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. PICL, A Portable Intrumented Communication Library, C Reference Manual. Tech. Rep. Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, July 1990.

[16] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. A User's Guide to PICL, A Portable Intrumented Communication Library. Tech. Rep. Tech. Rep. ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Oct 1991.

[17] George, A., Heath, M. T., and Liu, J. W.-H. Parallel Cholesky Factorization on a Shared-Memory Multiprocessor. *Linear Algebra and Applications 77* (1986), pp. 165–187.

[18] George, A., and Liu, J. W.-H. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[19] Golub, G. H., and Loan, C. F. V. *Matrix Computations.* John Hopkins University Press, 1989.

[20] Group, M. P. S. *High Performance Fortran*, report ml01-5/u46 ed. Digital Equipment Corporation, 1992.

[21] Harrington, R. F. *Time-Harmonic Electromagnetic Fields.* McGraw-Hill Book Company, New York, 1961.

[22] Harrington, R. F. Matrix Methods For Field Problems. *Proc. IEEE vol. 55, No. 2* (Feb. 1967), pp. 136–149.

[23] Harrington, R. F. *Field Computation by Moment Methods.* Krieger Publishing Co., Malabar, FL, 1982, c1968.

[24] Harrington, R. F., and Mautz, J. R. A Generalized Network Formulation for Aperture Problems. *IEEE Transactions on Antennas and Propagation 24*, 6 (Nov. 1976), pp. 870–873.

[25] Hull, J., and White, A. The Pricing of Options on Assets with Stochastic Volatilities. *Journal of Finance 42* (1987), pp. 281–300.

[26] Jordon, E. C., and Balmain, K. G. *Electromagnetic Waves and Radiating Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1969.

[27] Kurokawa, K. *An Introduction to the Theory of Microwave Circuits*. Academic Press, New York, 1969.

[28] Lu, Y., and Harrington, R. F. Electromagnetic Scattering from a Plane Conducting Two Slots Terminated by Microwave Network(TE Case). Tech. rep., Report TR-91-2, 1991.

[29] Lu, Y., Mohamed, A. G., and Harrington, R. Implementation of Electromagnetic Scattering From Conductors Containing Loaded Slots on the Connection Machine CM-2. Tech. rep., CRPC-TR92209, March 1992.

[30] MasPar Computer Corporation. *Maspar Fortran User Guide*, version 1.1, revision a2, pp. 2–9 ed., August 1991.

[31] MasPar Computer Corporation. *MasPar System Overview*, pn 9300-9001-00, revision a5 ed., Aug. 1991.

[32] Mautz, J. R., and Harrington, R. F. Modal Analysis of Loaded N-Port Scatterers. *IEEE Transactions on Antennas and Propagation 21*, 2 (March 1973).

[33] Mautz, J. R., and Harrington, R. F. Transmission from a Rectangular Waveguide into Half Space Through a Rectangular Aperture. Tech. rep., Report TR-76-5, 1976.

[34] Mills, K., Vinson, and M., Cheng, G. A Large Scale Comparison of Option Pricing Models with Historical Market Data. Tech. rep., SCCS 260, Syracuse Center for Computational Science, May 1992.

[35] Mills, K., Vinson, M., and Cheng, G. Load Balancing, Communication, and Performance of a Stock Option Pricing Model on the Connection Machine-2 and DECmpp-12000. Tech. rep., SCCS 273, Syracuse Center for Computational Science, May 1992.

[36] Mohamed, A. G., Fox, G. C., and von Laszewski, G. Blocked LU Factorization on a Multiprocessor Computer. Internal Report SCCS-94b, Northeast Parallel Architectures Center, Syracuse University, 111 College Place, Syracuse NY 31244-4100; CPRC-TR92212, Center for Research on Parallel Computation, Rice University, Houston, TX 77251-1892, April 1992. To appear in: Microcomputers in Civil Engineering.

[37] Mohamed, A. G., Lu, Y., and Harrington, R. Implementation of Electromagnetic Scattering From Conductors Containing Loaded Slots on DECmpp-12000 Computer. Tech. rep., SCCS 287, April 1992.

[38] N. N. Wang, J. H. Richmond, M. C. G. Sinusoidal Reaction Formulation for Radiation and Scattering from Conducting Surfaces. *IEEE Trans. on Antennas and Propagation vol. AP-23, 3* (May 1975), pp. 376–382.

[39] Nugent, S. F. The iPSC/2 Direct-Connect Technology. *Third Conference on Hypecube Concurrent Computers and Applications 1* (1988), pp. 51–60.

[40] Parasoft Corporation. *Express Reference Manual*, 1988.

[41] Pase, D., MacDonald, T., and Meltzen, A. MPP Fortran Programming Model. Cri report, Cray Research Incorporation, March 1992.

[42] Rice, J. R., and Jing, J. Problems to Test Parallel and Vector Languages. Tech. rep., CSD-TR-1016, 1990.

[43] Rosing, M., Schnabel, R. B., and Weaver, R. P. Scientific Programming Languages for Distributed Memory Multiproicessors: Paradigms on Research Issues. In *Languages Compilers and Run-Time Environments for Distributed Memory MAchines*, J. Saltz and P. Mehrotna, Eds. North Holland, Amsterdam, London, New York, Tokyo, 1992.

[44] Steele, G. Talk presented ath the HPFF in Dallas, TX. available via ftp on titan.rice.edu:/public/HPFF, Jan. 1992.

[45] Stratton, J. *Electromagnetic Theory.* McGraw-Hill Book, Co., New York, 1941.

[46] Thinking Machine Corporation. *Connection Machine Model CM-2 User's Guide*, version 6.1 ed., October 1991.

[47] Thinking Machines Corporation. *CM Fortran Reference Manual*, version 5.2-0.6, pp. 368 ed., 1989.

[48] von Lasewski, G. Proposal to extend FORTRAN-D with BLAS derectives. Tech. Rep. unbulished internal report, Northeast Parallel Architectures Center, Syracuse University, June 1992.