appears in International Parallel Processing Symposium, Apr. 1996

# Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism

Shankar Ramaswamy Eugene W. Hodges IV Prithviraj Banerjee

Center for Reliable and High-Performance Computing
469 Computer and Systems Research Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St.
Urbana, IL 61801
E-mail: {shankar, ewhodges, banerjee}@crhc.uiuc.edu

## Abstract

*In this paper we suggest a new approach aimed at reducing the effort required to program distributed-memory multicomputers. The key idea in our approach is to automatically convert a program written in a library-based programming language (*MATLAB*) to a parallel program based on the ScaLAPACK parallel library. In the process of performing this conversion, we apply compiler optimizations that simultaneously exploit task and data parallelism. As our results show, our approach is feasible and practical and our optimization provides significant performance benefits.*

## 1. Introduction

Distributed-memory multicomputers (DMMs) offer high performance at relatively low hardware costs making them very attractive for parallel processing. Unfortunately, a tremendous effort is required to write parallel programs that efficiently utilize the hardware in these machines.

A promising approach to lessening the burden of parallel programming is the use of parallel libraries written by experts. Parallel libraries offer high levels of performance, rapid prototyping, and do not require the programmer to understand the underlying architecture or algorithm. However, a drawback of this approach is that libraries often do not offer the flexibility of custom software.

In this paper, we propose an approach for making programming using libraries easier while also offering some flexibility. We have built a compiler prototype that maps programs written in a high-level, library-based programming language (MATLAB [10]) onto a parallel library (ScaLAPACK [3]). By automating the mapping process, we enable the use of compiler optimizations for performance improvements. We describe one such optimization, the simultaneous exploitation of task and data parallelism, and demonstrate its effectiveness.

### 1.1. Related Work

MATLAB is a commercially available software package from The MathWorks Inc. that is tailored to rapid prototyping of applications by scientists and engineers. MATLAB is widely used in research and industry for numerical computations. Matrices are basic data types in the MATLAB language and can be used directly as operands or function call parameters. Also, the core computational intrinsics in MATLAB were originally based on the LINPACK [5] and EISPACK [15] libraries. Thus the use of ScaLAPACK for our proposed parallel version follows naturally.

The ScaLAPACK library of linear algebra routines has evolved from the LAPACK [1], EISPACK and LINPACK libraries. It is aimed at providing scalable performance on distributed-memory multicomputers. The routines use block cyclic data distributions for matrices to obtain good performance and minimize data movement between various levels in the memory hierarchy. The ScaLAPACK library uses the Basic Linear Algebra Communication Subroutines (BLACS) [6] for all interprocess communication. The latest version of BLACS supports the execution of routines in different processor partitions, making the generation of task and data parallel code using ScaLAPACK possible.

De Rose et al. [14] are working on a MATLAB compiler to facilitate rapid prototyping of numerical libraries and programs on shared-memory multiprocessors as well as optimized code for workstations. Their compiler performs de-

This research was supported in part by an IBM Partnership Award and in part by an ONR Fellowship. We are also grateful to the National Center for Supercomputing Applications for providing access to their computers.

tailed analysis of a MATLAB program and translates it into a Fortran 90 program augmented with directives for parallelization and calls to any available parallel library routines. At this point the compiler from De Rose et al. exploits only data parallelism. Our MATLAB compiler targets distributed-memory multicomputers and exploits both task and data parallelism.

## 2. Language Description

The language supported by our compiler currently is a subset of MATLAB with additional directives for specifying type and size information for variables. These directives reduce run-time overhead by obviating the need for dynamic inference and are in the form of MATLAB comments to ensure that the code can also be used in the MATLAB computing environment.

A MATLAB program is assumed to be composed of a set of functions or modules with one distinguished function called `main`. Program execution is assumed to start with the first statement in `main`. A function has the following components:

**Header**: provides a declaration for the function name, its inputs and outputs.

`function [`*outputs*`] = `*name*`(`*inputs*`)`

Examples:

```
function [a,b,c] = foo1(x,y)
function a = foo2(x,y,z)
```

**Type and size directives** : provide type and size information for matrices used in the function.

`%! ` *category type name*`(`*dimensions*`) ;`

Examples:

```
%! local float foo1(128,128);
%! input int foo2($,$);
%! output double foo3($),foo4;
```

where *local* variables are used only in the body of the function, *input* variables are arguments passed in, and *output* variables are modified by the function. Types for matrices include integer (*int*), single precision (*float*) and double precision (*double*); MATLAB's complex is currently only indirectly supported. Dimensionality information is required for all variables. However, size information is optional; it is not required for variables whose sizes are invocation dependent. A $ for a particular dimension denotes the size in that dimension is invocation dependent. The compiler will automatically determine the sizes for each invocation of the function; this is described in a later section.

**Statements**: represent the computation performed in the function. A simple assignment statement has the form:

`[`*outputs*`] = ` *expression*

where *expression* can use any of the MATLAB operators or intrinsic functions. For details on MATLAB operators, refer to the MATLAB Reference Guide [10].

Examples:

```
[a,b,c] = foo1(x,y);
a = x * y + z * foo2(u,v,w);
```

At this point, the compiler prototype only allows references to entire matrices in expressions (MATLAB allows for submatrix references).

A loop has the form:

```
for loopvar = start : stop
    statements
end
```

Example:

```
for i = 1:100
    a = x * y - z;
    b = foo1(z);
end
```

The `while` loop and `if` conditionals are also handled by the compiler and have the following forms:

```
while (var1 operator var2)
    statements
end
```

and

```
if (var1 operator var2)
    statements
else
    statements
end
```

The benchmark `bicg` makes use of the `while` and `if` constructs, but they are beyond the scope of this paper.

Figure 1 shows a sample MATLAB program for multiplying two complex matrices.

```
function main
%! local float areal(256,256),aimag(256,256);
%! local float breal(256,256),bimag(256,256);
%! local float creal(256,256),cimag(256,256);

creal = areal * breal - aimag * bimag;
cimag = areal * bimag + aimag * breal;
```

**Figure 1. MATLAB program for multiplying complex matrices.**

## 3. Task and Data Parallelism

The simultaneous exploitation of task and data parallelism [13], [16], [7] provides significant benefits for many application programs. Exploiting task and data parallelism involves using a task graph representation for programs to decide on the number of processors to use for each task (allocation) and an execution (schedule).

The task graph representation we use is called the *Macro Dataflow Graph* (MDG) and is similar to the Hierarchical Task Graph (HTG) [8] representation. A MATLAB program is assumed to consist of a set of modules or functions with one distinguished function called `main`. This function is assumed to be at the topmost level and is not invoked from any other function. The MDG representation for a MATLAB program is a weighted directed acyclic graph whose tasks correspond to one of:

1. Basic operators or Intrinsic functions (Simple tasks)

2. Loop constructs (Loop tasks)

3. User defined functions (UserDef tasks)

Edges in the MDG correspond to precedence constraints that exist among the operations corresponding to the tasks. There are two distinguished tasks in the MDG called START and STOP. START precedes all other tasks, and STOP succeeds all other tasks. The MDG is hierarchical in structure, i.e., tasks in the MDG can themselves contain other MDGs. The Loop tasks and UserDef tasks are hierarchical. In the case of a Loop task, its MDG corresponds to the statements in the loop body; for a user-defined function, its MDG corresponds to the statements in the function. The topmost MDG corresponds to `main`. Although MATLAB allows recursion, the compiler prototype does not support this feature.

The weights of the tasks in the MDG correspond to the cost of executing the corresponding operators or functions; we refer to this cost as the processing cost of the function. Processing costs depend on the sizes of the operands and the number of processors used to execute the operator or function. Our programming paradigm dictates that all user-defined functions must ultimately be composed of intrinsic operators and functions, which in our case will be ScaLAPACK or other library tasks. Therefore, knowing the costs of ScaLAPACK or other library tasks, we can compute the cost of a given user-defined function. We use a mathematical model for the costs of data-parallel scientific tasks as a function of the number of processors used to execute it [13]; the parameters of the model are task and input size dependent. In order to obtain these parameters, we profile and model the execution times of ScaLAPACK or other library tasks as a function of data set sizes and the number of processors used to execute them. The parameters need to be computed only once for each machine.

Edge weights in the MDG correspond to the costs of redistributing data between the execution of operators or functions; we refer to this cost as data redistribution costs. The redistribution is needed to ensure that all operands have the appropriate distribution across the processors used to execute the operator or function. We have developed mathematical models for these costs also [13]; they are a function of the size and distribution of the matrix being redistributed as well as some basic machine parameters that need to be measured only once for a new machine.

Based on the edge and task weight models, we have an allocation and scheduling algorithm for exploiting task and data parallelism in MDGs with only Simple tasks. The algorithm is based on convex programming techniques; a detailed description of this algorithm can be found in [13],[12]. In this paper, we only discuss the extensions to our algorithm for handling allocation and scheduling for Loop tasks and hierarchical MDGs.

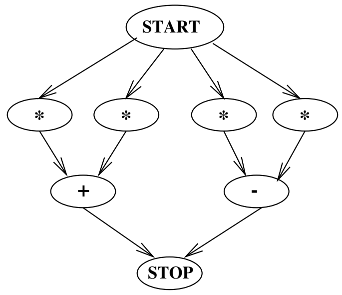
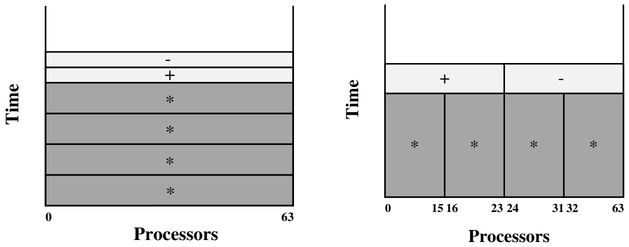



**Figure 2. MDG for complex matrix multiply.**




(a) Purely data parallel (b) Task and data parallel

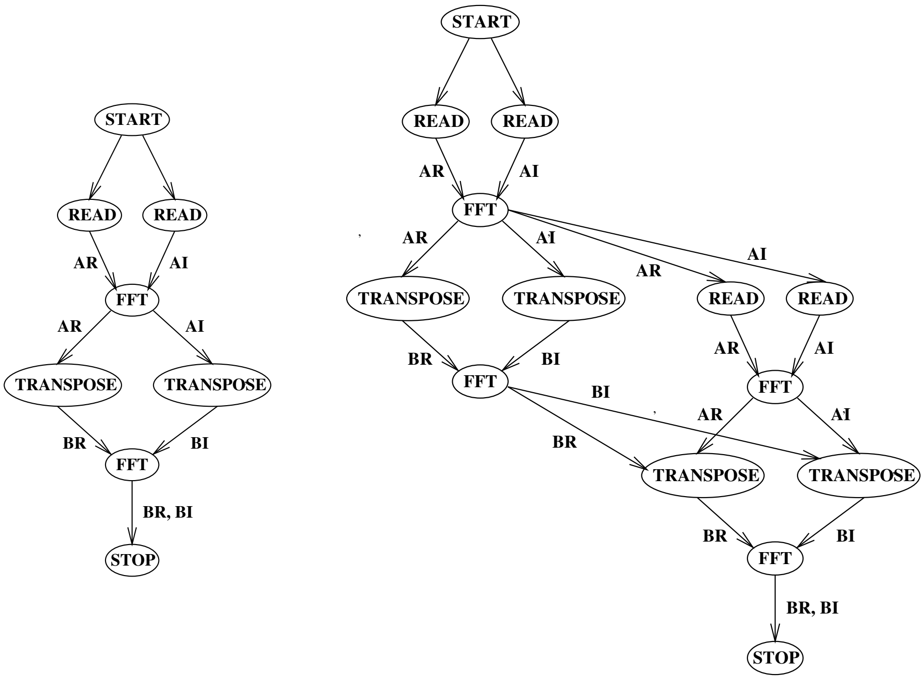
**Figure 3. Allocation and scheduling schemes for complex matrix multiply.**

### 3.1. Example

The following example illustrates the usefulness of proper allocation and scheduling. Figure 2 shows the MDG for the complex matrix multiply code shown in Figure 1. Given a 64-processor CM-5, there could be many ways in which we can allocate and schedule this MDG. For instance, the simple scheme shown in Figure 3(a) would execute the tasks one after another on all 64 processors. In this case, we have measured an execution time of 8.5 seconds. On the other hand, a better way of executing the MDG, as shown in Figure 3(b), would be first to execute the four matrix multiplies concurrently using 16 processors each followed by the matrix add and subtract concurrently on 32 processors each. The finish time decreases to 7.5 seconds. The first scheme is exploiting pure data parallelism, with each task using all available processors. However, the second scheme is exploiting both task and data parallelism: tasks execute concurrently and share available processors.

Good allocation and scheduling speeds program execution because of more efficient execution. Beyond a certain

```
for i = 1:100
  read(ar);
  read(ai);
  fft(ar,ai);
  br = ar';
  bi = ai';
  fft(br,bi);
end
```




(a) Example loop (b) MDG for loop (c) MDG after unrolling once

**Figure 4. Example illustrating unrolling loops.**

number of processors, most real applications execute more inefficiently as the number processors grows if the data set is not also scaled. Task parallelism can be used to control the degree of data parallelism used for each individual task and thus increase efficiency.

### 3.2. Loop Task Allocation and Scheduling

As discussed above, we have previously developed an algorithm called SIMUL [12] based on convex programming techniques for allocation and scheduling MDGs with only Simple tasks. In order to handle MDGs with Loop tasks, we present below our new loop allocation and scheduling approach (LOOP); this algorithm uses the SIMUL algorithm. Given the MDG corresponding to a loop body, allocation and scheduling of the loop on $P$ processors consists of the following steps:

1. Copy the MDG into a temporary (*TEMP*). Set the unroll factor (*UF*) to 1.

2. Allocate and schedule *TEMP* using the SIMUL algorithm. Average the finish times obtained with the current value of *UF*. Check to see if this average time improves upon the average time of the previous iteration. If not, go to the last step; else, proceed to next step.

3. Unroll the loop by adding another copy of the MDG to *TEMP*. This unrolling process enforces all of the loop carried precedence constraints that exist among variables involved in the loop body.

4. Increment *UF*.

5. Repeat starting at Step 2.

6. Replace the input MDG with *TEMP* and use *UF* to modify the loop bounds.

The motivation for unrolling the loop is to try to increase the available task parallelism by overlapping the execution of tasks from different iterations. Increasing the task parallelism provides opportunities for increased performance. Consider the MATLAB loop shown in Figure 4(a) and the MDG in Figure 4(b) corresponding to the body of the loop. Figure 4(c) shows the MDG corresponding to unrolling the loop body once. Unrolling the just once has significantly increased the amount of available task parallelism, providing improved performance.

### 3.3. Hierarchical MDG Allocation and Scheduling

Since MATLAB programs often result in a hierarchical structure for the MDG, we have developed the following new algorithm for allocation and scheduling a hierarchical MDG:

1. At each level, check to see if the parent of the task graph at this level is a Loop task or a UserDef task. If it is a

Loop task, use the LOOP allocation and scheduling algorithm; otherwise, for a UserDef task, use the SIMUL allocation and scheduling algorithm.

2. After allocating and scheduling at this level, traverse the list of tasks at one level lower in the task graph and find any Loop tasks or UserDef tasks. If at least one is found, allocate and schedule the MDGs represented by these tasks using the allocations computed in step 1; otherwise, terminate.

## 4. The Compiler

The important phases in the compilation process are:

**Parsing**: The MATLAB source code is parsed, and an abstract syntax tree is constructed. The data declaration directives provide type information for all of the matrices and scalars. In addition, they provide size information for all of the matrices in the `main` module and for any matrices in user-defined modules that have static sizes.

**Matrix size analysis**: This pass analyzes the structure of the program in a bottom-up manner and computes matrix size information. At the lowest level, we have operators and intrinsic functions for which we know how output sizes map to input sizes. For example, we know that the multiplication of an $M \times N$ matrix by an $N \times P$ matrix results in an $M \times P$ matrix. Using the mapping for operators and intrinsic functions, we traverse the abstract syntax tree bottom-up and map input sizes to output sizes for the matrices of all user-defined functions before finally computing the mappings for the `main` module. At this point the size of every matrix that is invocation dependent in functions is tied to the sizes of the matrices in `main`. Therefore, knowing the size of matrices in `main` we can determine the size of all of the matrices at each call site for user-defined functions.

Consider the MATLAB program shown in Figure 5. Here, we have two modules, `main` and a user-defined function `cmmul`, and `main` contains calls to `cmmul` at two different places with two different sets of matrix sizes. During the analysis phase, we use the properties of the multiplication and addition operators in `cmmul` to tie the dimension sizes of `zr` to the size of the first dimension of `xr` and the second dimension of `yr`. Therefore, for the first call to `cmmul` in `main` we use the sizes of the matrices corresponding to `xr` (`ar`) and `yr` (`br`) to determine that the size of `zr` for that call is $200 \times 250$. Since `dr` in `main` corresponds to `zr` in `cmmul`, we determine its size to be $200 \times 250$. Similarly, for the second call to `cmmul`, the size of `er` is determined to be $50 \times 50$. The mappings for the inputs to the outputs of the intrinsic `fft2` are known.

**MDG construction**: After analyzing the matrix size information in the program, we are ready to build the corresponding MDG by using the size information to compute the weights of the tasks in the MDG. As described in Section 3, the weighting functions for tasks have parameters that are input size dependent.

```
function main

%! local float ar(200,50), ai(200,200);
%! local float br(50,250), bi(50,250);
%! local float cr(250,50), ci(250,50);
%! local float dr($,$), di($,$);
%! local float er($,$), ei($,$);

[dr,di] = cmmul(ar,ai,br,bi);
[dr,di] = fft2(dr,di);

[er,ei] = cmmul(er,ei);
[er,ei] = fft2(er,ei);

function [zr,zi] = cmmul(xr,xi,yr,yi)

%! local float zr($,$), zi($,$);
%! local float xr($,$), xi($,$);
%! local float yr($,$), yi($,$);

zr = xr * yr - xi * yi;
zi = xr * yi + xi * yr;
```

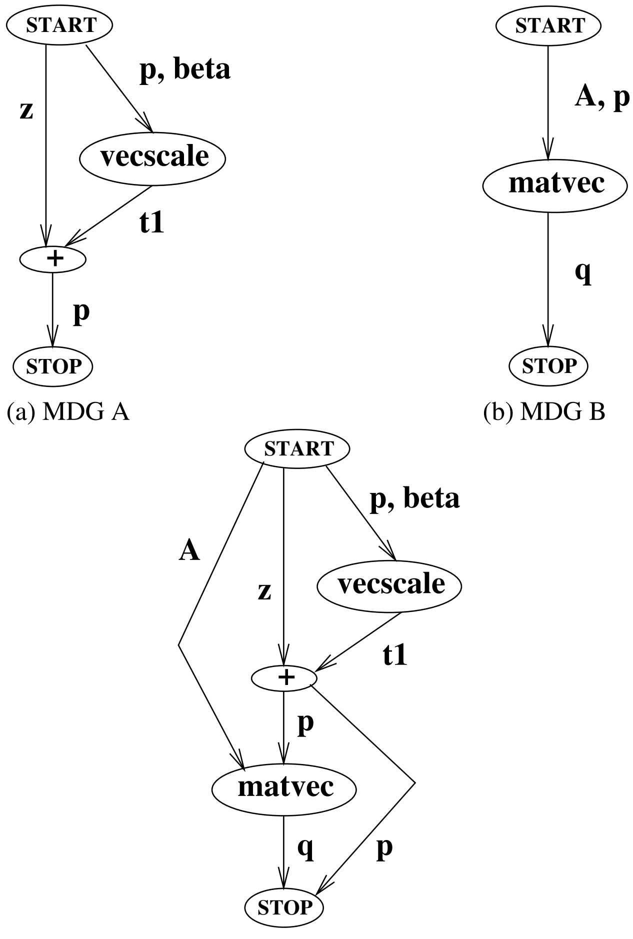
**Figure 5. Example for matrix size analysis.**

The MDG is constructed in two passes: the first pass builds MDGs separately for `main` and each user-defined function, and the second pass uses these MDGs to create a single hierarchical structure for the entire program. The second pass starts with `main` and proceeds top down.

The MDG for a function is built incrementally by traversing in order the list of statements in the function. As each statement is encountered, we build an MDG for it and then combine it with the MDG representing the previous statements. The START and STOP tasks play an important role in this process because they are used to capture all of the reads and writes to variables in an MDG. The set of all variables $\mathcal{R}$ that are read by the tasks in an MDG consists of the labels on the arcs exiting START; similarly, the labels on arcs entering STOP comprise the set of all variables $\mathcal{W}$ that are modified.

Figure 6 illustrates the process of combining MDGs. When combining MDG A with MDG B, we first examine the START node of MDG B to determine $\mathcal{R}_B$. Then, we use the STOP node of MDG A to determine $\mathcal{W}_A$. The intersection of $\mathcal{R}_B$ and $\mathcal{W}_A$ will determine which tasks in MDG B must be preceded by tasks in MDG A; edges are then added between corresponding tasks in MDG A and MDG B for these precedence constraints.

As previously described, statements in a MATLAB program may contain assignment statements or loop statements.




(a) MDG A (b) MDG B (c) MDG C

**Figure 6. Example of combining MDGs.**

In the case of assignment statements involving only operators, applying the process we have described above is straightforward. For assignments involving intrinsic or user-defined functions, we also need to consider any expressions that may occur in the arguments to these functions while building the MDG. For loop statements, we build an MDG for the loop body and use the sets $\mathcal{R}$ and $\mathcal{W}$ of this MDG at the upper level.

**Allocation and Scheduling**: The algorithms desribed in the previous section are used to make allocate and schedule the tasks in a top-down fashion for a hierarchical MDG.

**Code Generation**: Operators and intrinsic functions are mapped to the appropriate ScaLAPACK or other library routines, and calls to data redistribution routines, described in [11], are generated. Each of the calls to a library or data redistribution routine requires a separate BLACS context [6] to execute correctly. We generate calls to create these contexts with the appropriate set of processors. C code is generated, and each source file contains a single function, corresponding to a user-defined function or loop statement.

## 5. Results

We measured the performance of our MATLAB compiler using four benchmark programs. Two of these, `cmmul` and `cfd`, have been used by us in the past and contain only Simple tasks. The other two benchmarks, `poly` and `bicg`, contain a Loop tasks and conditionals. A brief description for each of the benchmarks is given below:

1. `cmmul`: multiplication of $256 \times 256$ complex matrices.
2. `cfd`: spectral CFD code using $256 \times 256$ mesh [9].
3. `poly`: polynomial multiplications using FFTs [4].
4. `bicg`: BiConjugate Gradient iterative method for solving systems of linear equations [2].

We used a 128-processor CM-5 for all our experiments, with copies of ScaLAPACK and BLACS libraries for the CM-5 that we installed. Speedups obtained for our benchmark programs are shown in Figure 7. The data corresponding to purely data-parallel execution are marked "SPMD," and the data for task and data parallel execution are marked "MPMD." From the speedup graphs, it is clear that a combined task and data-parallel approach provides a significant increase in performance for applications.

## 6. Conclusions and Future Work

In this paper we present a new hybrid approach for scientific programming on distributed-memory multicomputers. We use a high-level language (MATLAB) to generate code that uses a standard parallel library (ScaLAPACK). Besides easing the programming process, our hybrid approach provides new opportunities for compiler optimizations. We have described one such optimization: the simultaneous exploitation of task and data parallelism. This optimization can provide significant performance benefits for many applications, as evident from our results.

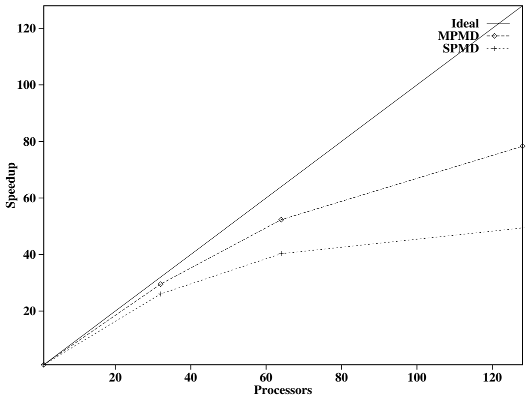
Currently, our compiler allows for simple assignment statements and simple loops. In the future, we plan to extend this work to provide more complete support for submatrix references and other features of the MATLAB language.

Detailed information on the allocation and scheduling algorithms and other aspects of this work are available at http://www.crhc.uiuc.edu/Paradigm.

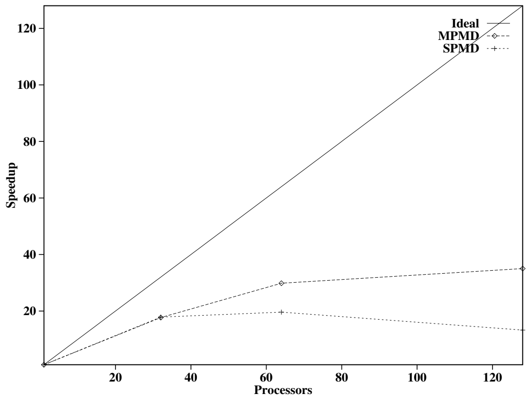
## References

[1] E. Anderson et al. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proc. of Supercomputing '90*, pages 2–11, New York, NY, Nov. 1990.

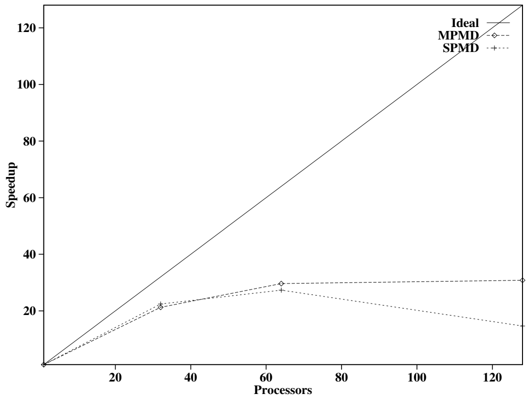
[2] R. Barett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.



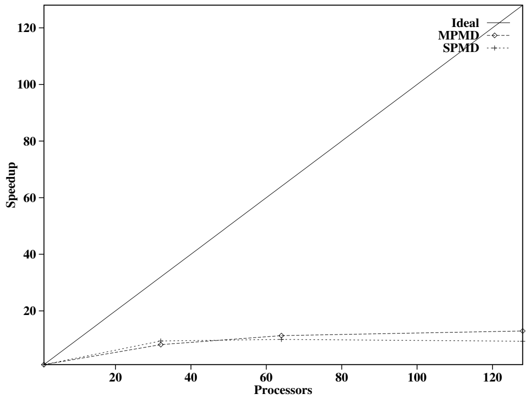

(a) `cmmul`




(b) `cfd`




(c) `poly`




(d) `bicg`

**Figure 7. Speedups obtained for benchmark programs on a 128-processor CM-5.**


[3] J. Choi et al. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Frontiers '92: The 4th Symp. on the Frontiers of Massively Parallel Computation*, pages 120–127, McLean, VA, Oct. 1992.

[4] M. Dhagat, R. Bagrodia, and M. Chandy. Integrating Task and Data Parallelism in UC. In *Proc. of the 24th Int'l Conf. on Parallel Processing*, pages 29–36, Oconomowoc, WI, Aug. 1995.

[5] J. J. Dongarra et al. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[6] J. J. Dongarra and R. C. Whaley. LAPACK Working Note 94: A User's Guide to the BLACS v1.0. Technical Report UT-CS-94-281, Univ. of Tennessee, Knoxville, TN, Mar. 1995.

[7] I. Foster et al. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 293–300, Knoxville, TN, May 1994.

[8] M. Girkar and C. D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):166–178, Mar. 1992.

[9] S. L. Lyons, T. J. Hanratty, and J. B. McLaughlin. Large-scale Computer Simulation of Fully Developed Channel Flow with Heat Transfer. *International Journal of Numerical Methods for Fluids*, 13(8):999–1028, Nov. 1991.

[10] The Mathworks, Inc., Natick, MA. *MATLAB Reference Guide*, 1992.

[11] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, Feb. 1995.

[12] S. Ramaswamy and P. Banerjee. Simultaneous Allocation and Scheduling Using Convex Programming Techniques. *Parallel Processing Letters*, Dec. 1995. to appear.

[13] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. In *Proc. of the 23rd Int'l Conf. on Parallel Processing*, pages II:116–125, St. Charles, IL, Aug. 1994.

[14] L. D. Rose et al. A MATLAB Compiler and Restructurer for the Development of Scientific Libraries and Applications. In *Proc. of the 8th Work. on Langs. and Compilers for Parallel Computing*, pages 18.1–18.18, Columbus, OH, Aug. 1995.

[15] B. T. Smith et al. *Matrix Eigensystem Routines - EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1976.

[16] J. Subhlok et al. Exploiting Task and Data Parallelism on a Multicomputer. In *Proc. of the 4th ACM SIGPLAN Symp. on Principles & Practices of Parallel Prog.*, pages 13–22, San Diego, CA, May 1993.