Preliminary Results from a Parallel MATLAB Compiler

Michael J. Quinn, Alexey Malishevsky, Nagajagadeswar Seelam, and Yan Zhao
Department of Computer Science
Oregon State University
Corvallis, OR 97331

Abstract

We are developing a compiler that translates ordinary MATLAB scripts into code suitable for compilation and execution on parallel computers supporting C and the MPI message-passing library. In this paper we report the speedup achieved for several MATLAB scripts on three diverse parallel architectures: a distributed-memory multicomputer (Meiko CS-2), a symmetric multiprocessor (Sun Enterprise Server 4000), and a cluster of symmetric multiprocessors (Sun SPARCserver 20s). By generating code suitable for execution on parallel computers, our system multiplies the gains achievable by compiling, rather than interpreting, MATLAB scripts. Generating parallel code has an additional advantage: the amount of primary memory available on most parallel computers makes it possible to solve problems too large to solve on a single workstation.

1. Introduction

Many problems that scientists and engineers want to solve are computationally intensive. Since parallel computers offer the potential for high computation rates, you might assume that most computational scientists would have embraced parallel computing by now. However, two significant obstacles have prevented the widespread adoption of this new paradigm. First, taking advantage of parallel computers has usually required writing code in a new programming language. Second, even when scientists have gone to the trouble of developing parallel programs, the relatively short life-span of parallel computers has meant that large investments in coding have often yielded only short-term benefits. For this reason, the use of parallel computers by computational scientists and engineers is still the exception, rather than the rule.

However, the emergence of MATLAB as a popular software package for numerical computation, data analysis, and graphics may change the face of parallel computing. Unlike FORTRAN 77, MATLAB contains vector and matrix data types and operations manipulating these data types. Getting maximum performance from the MATLAB interpreter involves learning how to eliminate loops by replacing series of scalar operations with single vector or matrix operations. The resulting scripts are amenable to compilation for parallel computers.

At Oregon State University, we have witnessed scientists switch from FORTRAN to MATLAB because it is a superior environment for developing numerical models. We have heard them explain that they plan to debug their models in MATLAB using a small data set, then convert their MATLAB scripts to FORTRAN when it is time to run the model on real data. When the time to re-implement the program in FORTRAN arrives, they often change their minds, choosing instead to wait for the MATLAB interpreter to execute the script on a large data set, even if it requires several CPU days.

These two actions—writing MATLAB scripts amenable to parallelization and choosing to stick with MATLAB even when solving large problems—create an environment in which a parallel MATLAB compiler has high utility. We are implementing a compiler, nicknamed ‘Otter’, that translates MATLAB scripts into SPMD-style C programs with calls to the standard MPI message-passing library. The resulting codes can be executed on sequential or parallel computers equipped with a C compiler and the MPI library (Figure 1).

Our goal is not to implement every function supported by MATLAB. Rather, we view our system as a vehicle for exploring the feasibility of compiling high-level languages into code that executes efficiently on parallel computers.

In this paper we will describe related parallel MATLAB projects, outline the design of our compiler and run-time library, and present benchmark results from several applications executing on three diverse parallel architectures.

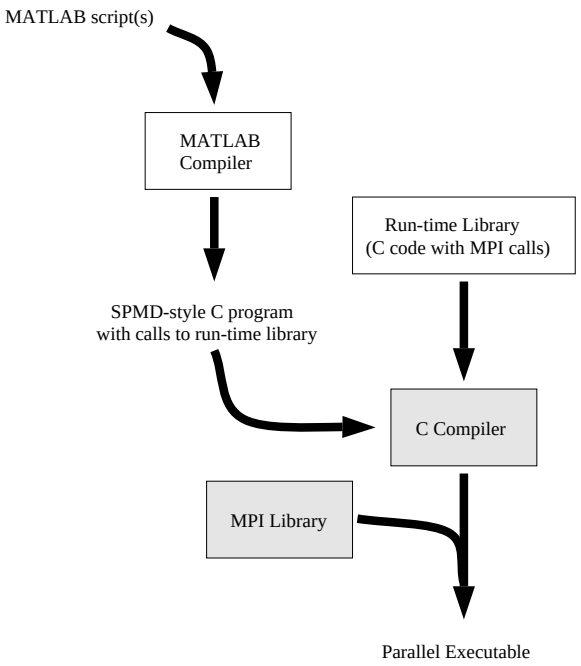

Figure 1. The 'Otter' MATLAB compiler and run-time library enable MATLAB scripts to be compiled and executed on any parallel computer supporting a C compiler and the MPI message-passing library.

2. Related work

Other high-performance MATLAB projects can be divided into two categories: interpreter-based systems and translators. Interpreter-based systems run MATLAB interpreters in parallel. They extend MATLAB to allow the distribution and collection of submatrices between a "manager" processor and a collection of "worker" processors. The MATLAB Toolbox, developed at the University of Rostock in Germany, targets a collection of UNIX workstations connected on a local area network [10]. MultiMATLAB, developed at Cornell University, initially targeted conventional parallel computers, such as the IBM SP-2, but can also be executed on networks of workstations [11]. Wake Forest's Parallel Toolbox is an SPMD-style programming environment allowing programmers to access MATLAB interpreters running on a network of workstations [6]. Paramat, from Alpha Data Corporation in the United Kingdom, targets a PC add-on multiprocessor board containing Digital Alpha CPUs [8]. These systems differ from our software in two ways. First, the nodes execute MATLAB interpreters, rather than compiled code. Second, they add explicitly parallel extensions to the MATLAB language, which puts an additional burden on the user.

We are aware of five MATLAB compilers (or translators). Two of these compilers generate sequential code. The MathWorks now sells a compiler which translates MATLAB scripts into C code. MATCOM, sold by MathTools, translates MATLAB into C++ code suitable for compilation and execution on single-CPU systems [9].

The other three compilers target parallel computers. CONLAB, from the University of Umea in Sweden, translates MATLAB into C with calls to the PICL message-passing library [4]. It requires that the programmer make use of parallel extensions to MATLAB.

Perhaps the most sophisticated system is RTExpressä, marketed by Integrated Sensors, which translates MATLAB scripts into C with calls to MPI [7]. Through a graphical user interface, the programmer selects portions of the MATLAB script to be executed in parallel, indicating the desired number of processors and the type of parallelization desired (pipeline, data parallel, task parallel, or round robin). The system then generates a program suitable for execution on a parallel computer.

The project most similar to ours is the FALCON compiler developed at the University of Illinois [2, 3]. It translates MATLAB scripts, without any parallel extensions or user annotations, into Fortran 90 code. However, there is a significant difference between our compiler and the FALCON project. The Otter compiler translates MATLAB code directly into SPMD C programs with calls to MPI. The FALCON compiler generates Fortran 90 code, which needs another translation step in order to run on a parallel computer. Significantly, no performance results for the FALCON compiler on multiple-CPU platforms have been reported.

MATLAB implementations targeting parallel computers are summarized in Table 1.

3. Compiling

DeRose has provided detailed explanations of the process of translating MATLAB into Fortran [2, 3]. Our compiler design is similar to his design in many respects. Hence in this section we will simply highlight the steps of our multi-pass compiler, documenting the most significant differences between the two compilers. In particular, we will describe what needs to be done to generate an SPMD parallel C program suitable for execution on a distributed memory parallel computer.

MATLAB is an imperative language. The individual statements resemble those in BASIC or FORTRAN. MATLAB programmers produce M-files. An M-file may either be a script—a series of statements with no input parameters and no return values—or a function with optional parameters, local variables, and one or more return values.

Name	Site	Implementation
Paramat	Alpha Data Parallel Systems, UK	Interpreter
CONLAB	University of Umea, Sweden	Compiles to C/PICL
FALCON	University of Illinois	Compiles to Fortran 90
MATLAB Toolbox	University of Rostock, Germany	Interpreter
Multi-MATLAB	Cornell University	Interpreter
Otter	Oregon State University	Compiles to C/MPI
Parallel Toolbox	Wake Forest University	Interpreter
RTExpressa	Integrated Sensors	Compiles to C/MPI

Table 1. Experimental and commercial MATLAB-based systems targeting parallel computers. Only FALCON and Otter generate parallel code from "pure" MATLAB (i.e., MATLAB without any extensions).

A MATLAB "program" consists of a script, together with those M-files reachable through a chain of one or more references from the original script.

Otter is a multi-pass compiler designed to facilitate the later addition of optimization passes. The first pass consists of constructing the parse tree for the initial script. We have constructed the scanner and parser using *lex* and *yacc*. We are able to scan and parse all of MATLAB's syntactic constructs with one exception. The MATLAB interpreter allows both commas and white space to delimit elements of a list. To simplify scanning and parsing, we do not support the use of white space to delimit list elements. The parser augments the parse tree with additional links to simplify code analysis. The result is an abstract syntax tree (AST).

The second pass resolves all of the identifiers used in the program. Beginning with the original script, it determines which identifiers correspond to variables and which correspond to functions. User M-file functions identified during this pass are scanned, parsed, and eventually subjected to the same identifier resolution algorithm. At the end of this pass every M-file in the user's program has been added to the AST. Unlike DeRose, we do not in-line M-file functions where they are called. This makes the propagation of type information more difficult, but it reduces the size of the C programs emitted by the compiler.

The third pass of the compiler determines the type, shape, and rank of the variables and stores this information in the symbol table. Correctly identifying these attributes is essential to generating programs that execute efficiently. For example, variables may have one of four types: literal, integer, real, and complex. Recognizing that a variable is of type real rather than type complex saves half the memory and significantly reduces the amount of time to perform operations such as addition or multiplication using that variable. A variable may have either scalar or matrix rank. Each matrix variable has an associated shape, i.e., the number of rows and columns.

As much as possible, type and rank information is determined at compile time. MATLAB, designed as an interpreted language, allows the attributes of a variable to change during a program's execution. We solve this problem by transforming the program into static single assignment form, which ensures each variable is only assigned a value once [1]. Once the program is in static single assignment form, a static inference mechanism extracts information about variables from input files, constants, operators, and functions. If the user's program initializes a variable through external file input, a sample data file must be present, so that the compiler can determine the type of the variable as well as its rank. Shape information can be collected and propagated at run time.

Expression rewriting constitutes the fourth pass of the compiler. The output of the Otter compiler is a loosely synchronous, SPMD-style C program with calls to a runtime library which we have developed. The only parallelism we are exploiting is the data parallelism inherent in vector and matrix operations. For this reason our design is able to take advantage of many concepts developed by groups implementing compilers for data-parallel languages such as Fortran 90, High Performance Fortran, and C* [5]. In particular,

1. Scalar variables are replicated across the set of processors performing the computation.
2. Matrices are distributed among the local memories of the processors. Matrices of identical size are distributed identically.
3. The "owner computes" rule determines which processors are responsible for executing which operations.
4. Synchronization among processors is accomplished through message passing.
5. One processor coordinates all I/O operations.

Given these assumptions, the compiler is able to determine which terms and subexpressions may involve interprocessor communication. The compiler must modify the AST to bring these terms and subexpressions to the statement level, where they can be translated into calls to the run-time library. After this has been done, some element-wise matrix operations may remain. Because the C programming language does not support element-wise arithmetic operations on arrays, `for` loops must be inserted to perform these operations one element at a time.

For example, consider the following MATLAB statement involving matrices `a`, `b`, `c`, and `d`:

```
a = b * c + d(i,j);
```

Because matrices are distributed among the memories of the processors, the multiplication of matrices `b` and `c` involves interprocessor communication. Hence this operation must be performed via a call to the run-time library. Matrix element `d(i,j)` is owned by a single processor; it must be broadcast to the other processors. Since matrices of identical size are allocated to processors identically, matrix addition can be performed without any interprocessor communication. The compiler creates a `for` loop which enables each processor to adds its share of the matrix elements. The resulting code follows:

```
ML__matrix_multiply(b, c, ML__tmp1);
ML__broadcast(&ML__tmp2, d, i-1, j-1);
for (ML__tmp3 = ML__local_els(a)-1;
     ML__tmp3 >= 0; ML__tmp3--)
{
   a->realbase[ML__tmp3] = ML__tmp1->
      realbase[ML__tmp3] + ML__tmp2;
}
```

The fifth pass of the compiler looks for statements manipulating individual elements of matrices, such as:

```
a(i,j) = a(i,j) / b(j,i)
```

Statements such as this must be surrounded by a conditional, so that only the processor owning the matrix element referenced on the left-hand side of the statement actually performs the operations on the right-hand side and assigns the result. For example, the compiler translates the statement shown above into the following C code:

```
ML__broadcast(&ML__tmp1, b, j-1, i-1);
if (ML__owner(a, i-1, j-1))
{
   *ML__realaddr2(a, i-1, j-1) =
      *ML__realaddr2(a, i - 1, j - 1) /
      ML__tmp1;
}
```

The processor storing matrix element `b(j,i)` broadcasts the value to the other processors. Only the processor owning `a(i,j)` executes the assignment statement. (The indices in the output code are decremented by one, because C arrays are 0–based, while MATLAB arrays are 1–based.)

The sixth pass of the compiler performs "peephole" optimizations, looking for ways in which a sequence of run-time library calls can be replaced by a single call.

The final compiler pass traverses the AST, emitting C code interspersed with calls to the run-time library.

4. Run-time library

The run-time library is responsible for the allocation and de-allocation of vectors and matrices. It performs all matrix/vector operations which require interprocessor communication on a distributed memory parallel computer.

Every matrix and vector is represented on each processor by a C structure named MATRIX which contains global information about its type, rank, and shape. This structure also contains processor-dependent information, such as the total number of matrix elements stored on a particular processor and the address in that processor's local memory of its first matrix element. The run-time library function `ML__init` initializes the fields of the MATRIX structure. In the previous section we saw how the compiler is able to use this information to generate `for` loops to perform element-wise vector and matrix operations.

Run-time library functions perform all matrix manipulations except element-wise operations. For example, the function `ML__owner`, passed a pointer to a MATRIX structure and a pair of indices, returns 1 if and only if the processor calling the function stores that matrix element. Function `ML__print_matrix` prints the elements of a matrix. Function `ML__matrix_vector_multiply` is passed pointers to a matrix and two vectors and does the necessary computations and interprocessor communications to accomplish the matrix-vector multiplication. When the function returns, the third argument points to the result vector.

Data distribution decisions are made within the run-time library, simplifying the design of the compiler and making it easier to experiment with alternative data distribution strategies. In our initial implementation of the compiler, matrices are distributed in row-contiguous fashion among the memories of the processors, while vectors are distributed by blocks to the processors.

5. Single processor performance

We have benchmarked the performance of our compiler versus The MathWorks' interpreter and a commercial compiler on four applications.

The first application solves a positive definite system of 2048 linear equations using the conjugate gradient algorithm. The program makes extensive use of matrix-vector multiplication and vector dot product.

The second benchmark script is an ocean engineering application from the Department of Civil Engineering at Oregon State University. It evaluates the nonlinear wave excitation force on a submerged sphere using the Morrison equation. It requires vector shifts, outer products, and calls to the built-in function `trapz2`.

The third benchmark script performs an *n*-body simulation for 5,000 particles. This algorithm uses the built-in function `mean`. In addition, it exercises the run-time library's broadcast function.

The fourth benchmark script computes the transitive closure of a 512×512 matrix through repeated matrix multiplications. It was chosen to test the speed of the run-time library's implementation of matrix multiplication.

We have benchmarked the performance of The MathWorks' interpreter, version 2.0.2 of MathTools' MATCOM compiler (without BLAS calls), and our compiler (without BLAS calls). The platform is a single UltraSPARC CPU of the Sun Enterprise Server 4000. As Figure 2 illustrates, for these scripts our compiler always outperforms The MathWorks' interpreter. Our compiler is competitive with the MATCOM compiler, outperforming it on two benchmark scripts and underperforming it on the other two.

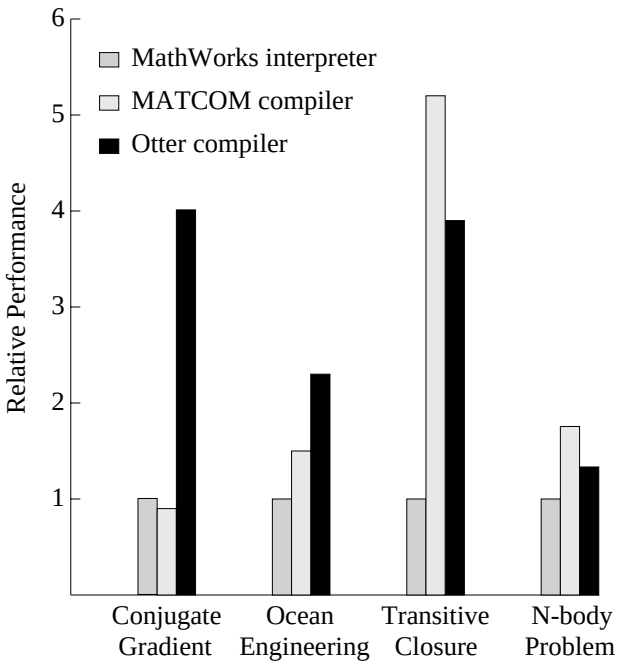

Figure 2. Relative performance of The MathWorks' interpreter, MATCOM compiler, and OSU's Otter compiler on four benchmark applications executing on a single UltraSPARC CPU of a Sun Enterprise Server 4000.

6. Performance on diverse parallel architectures

Our compiler emits SPMD-style parallel programs, allowing us to take advantage of many processors to multiply the gains achieved by compiling, rather than interpreting, MATLAB scripts. This is the principal advantage of our compiler over commercial MATLAB compilers that emit sequential code. In this section we present the performance of the four benchmark scripts described in the previous section. We have executed these scripts on three diverse parallel architectures: a distributed memory multicomputer (16–CPU Meiko CS-2), a distributed memory cluster of symmetric multiprocessors (four Sun SPARCserver 20s with 4 CPUs each), and a symmetric multiprocessor (8–CPU Sun Enterprise Server 4000).

The performance of the conjugate gradient script on our three target architectures is illustrated in Figure 3. The figure plots execution speed relative to the speed of The MathWorks' interpreter. For example, the compiled script executing on 16 CPUs of the Meiko CS-2 executes 50 times faster than the interpreter executing the script on a single CPU of the Meiko CS-2.

Figure 4 plots the performance achieved by the compiled ocean engineering script on our three target systems. The speedup achieved on this application is not as good because the size of the data set is relatively small, and most of the operations performed have $O(n)$ time complexity. As a result, the grain size of the typical computation is relatively small, increasing the overall impact of interprocessor communication.

The performance of the compiled script on the *n*-body simulation script on our three target architectures is illustrated in Figure 5. Again, the preponderance of $O(n)$ operations limits the opportunities for speedup through parallel execution.

Our fourth application is transitive closure. The script computes the transitive closure of an $n \times n$ matrix through $\log n$ matrix multiplications. The conventional sequential matrix multiplication algorithm requires $O(n^3)$ floating-point operations. Hence this script would seem to be a good candidate for parallel execution. The performance

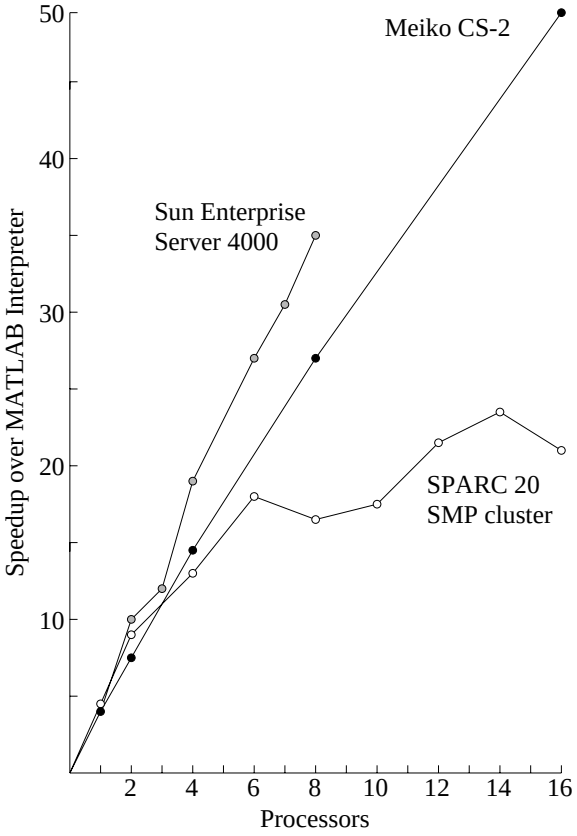

Figure 3. Performance of compiled conjugate gradient script relative to the performance of the MATLAB interpreter on a single CPU.

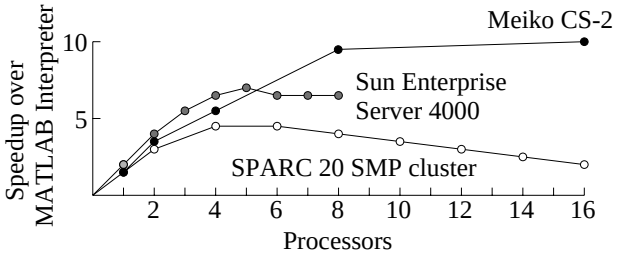

Figure 4. Performance of ocean engineering script relative to the performance of the MATLAB interpreter on a single CPU.

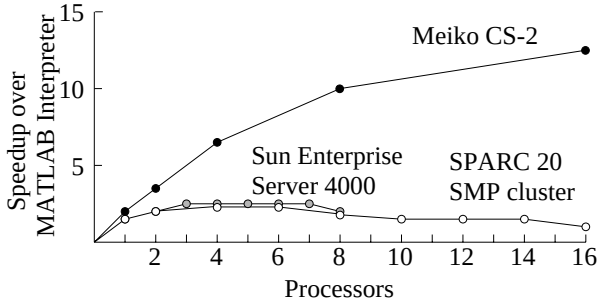

Figure 5. Performance of the *n*-body simulation script relative to the performance of the MATLAB interpreter on a single CPU.

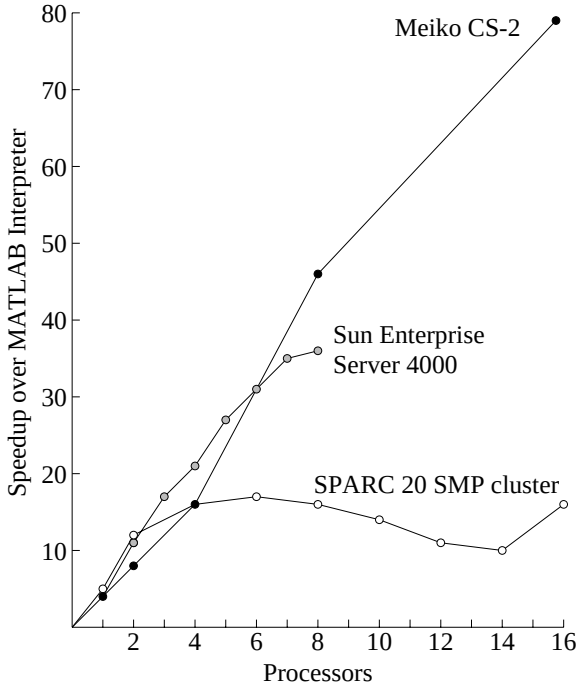

Figure 6. Performance of transitive closure script relative to the performance of the MATLAB interpreter on a single CPU.

results, illustrated in Figure 6, bear this out. The compiled program executes 78 times faster on 16 nodes of the Meiko CS-2 than the interpreted program executes on a single processor.

Of the three parallel architectures used as test beds in this study, the Meiko CS-2 provides the best balance between processor speed, message latency, and aggregate message-passing bandwidth. As a result, it generally achieves greater speedup than the other two parallel systems. The most unbalanced system is our cluster of SPARC 20 SMPs, which are connected by an Ethernet. The relatively high latency and low bandwidth of the interconnection network puts a severe damper on speedup achieved beyond four CPUs (the number of CPUs in a single SMP).

7. Summary

We have designed a compiler that translates MATLAB scripts into SPMD-style C code suitable for execution on parallel computers supporting the MPI message-passing library. The compiler detects data-parallel vector and matrix operations and generates code (or calls run-time library functions) that distribute the work across a set of processors. Currently our system implements a small number of MATLAB functions.

Our preliminary results indicate that the speedups achieved through the parallel execution of MATLAB scripts can vary widely. Two important determinants are the sizes of the matrices being manipulated and the complexity of the operations being performed on them. When the script calls for operations with complexity $\Omega(n^2)$ to be performed on matrices containing several hundred thousand elements or more, the performance improvement over The MathWorks' interpreter can be significant.

Translating MATLAB scripts into parallel code has an additional, very important advantage: larger problems can be solved. It is infeasible for the MATLAB interpreter to solve problems where the aggregate amount of data being manipulated exceeds the primary memory capacity of a workstation. In contrast, a parallel computer may have far more primary memory than an individual workstation.

Acknowledgements

This work was supported by the National Science Foundation under grant CDA-9216172.

Michael Quinn gratefully acknowledges the support of Tony Hey, the University of Southampton, and the United Kingdom's Science and Engineering Research Council, which provided him the opportunity to begin working on this compiler while he was on sabbatical in Southampton during the 1995–96 academic year.

References

[1] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991), pp. 451–490.

[2] DeRose, L. A., "Compiler Techniques for MATLAB Programs," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign (1996).

[3] DeRose, L., and Padua, D., "A MATLAB to Fortran 90 Translator and its Effectiveness," in *Proceedings of the 10th ACM International Conference on Supercomputing* (May 1996).

[4] Drakenberg, P., Jacobson, P., and Kågström, B., "A CONLAB Compiler for a Distributed Memory Multicomputer," in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Volume 2 (1993), pp. 814–821.

[5] Hatcher, P. J., and Quinn, M. J., *Data-Parallel Programming on MIMD Computers*, The MIT Press, 1991.

[6] Hollingsworth, J., Liu, K., and Pauca, P., "Parallel Toolbox for MATLAB PT v. 1.00: Manual and Reference Pages," Technical Report, Wake Forest University (1996).

[7] Integrated Sensors home page. `http://www.sensors.com/isi`

[8] Kadlec, J., and Nakhaee, N., "Alpha Bridge: Parallel Processing under MATLAB," in *Proceedings of the Second MathWorks Conference* (1995).

[9] MathTools home page. `http://www.mathtools.com`

[10] Pawletta, S., Drewelow, W., Duenow, P., Pawletta, T., and Suesse, M., "A MATLAB Toolbox for Distributed and Parallel Processing," in *Proceedings of the MATLAB Conference 95*, Cambridge, MA (1995).

[11] Trefethen, A. E., Menon, V. S., Chang, C.-C., Czajkowski, G. J., Myers, C., and Trefethen, L. N., "MultiMATLAB: MATLAB on Multiple Processors," Technical Report 96–239, Cornell Theory Center, Ithaca, NY (1996).