COMPILING PARALLEL MATLAB FOR GENERAL DISTRIBUTIONS USING TELESCOPING LANGUAGES

Mary Fletcher, Cheryl McCosh, Guohua Jin, and Ken Kennedy

Rice University
Department of Computer Science
Houston, TX 77005-1892

ABSTRACT

Matlab is one of the most popular computer languages for technical and scientific programming. However, until recently, it has been limited to running on uniprocessors. One strategy for overcoming this limitation is to introduce global distributed arrays, with those arrays distributed across the processors of a parallel machine. In this paper, we describe the compilation technology we have designed for Matlab D, a distributed-array extension of Matlab. Our approach is distinguished by a two-phase compilation technology with support for a rich collection of data distributions. By precompiling array operations and communication steps into Fortran plus MPI, the time to compile an application using those operations is significantly reduced. This paper includes preliminary results that demonstrate that this approach can dramatically improve performance, scaling well to at least 32 processors .

Index Terms— Parallelizing compilers

1. INTRODUCTION

The scientific application development community today is bifurcated. Most scientists prefer to develop algorithms and applications in a high-level language like Matlab, appropriately specialized to their domains through the use of libraries or "toolboxes." On the other hand, those producing computationally intensive applications, for which performance is essential, typically write parallel programs in conventional languages like C, C++, and Fortran with calls to MPI. This approach is both tedious and error prone, so most applications are produced by expensive professional programmers. For this reason, many scientists and engineers have little or no access to high-end computing for the solutions of their problems.

Many of these problems could be ameliorated if Matlab and similar languages could be extended and compiled to make efficient use of parallelism. There are two general strategies being pursued by the community to achieve this goal.

The first is to extend Matlab by the use of either explicit parallel constructs or by allowing multiple Matlab programs, running on different processors, to exchange messages using MPI. Mathworks itself has produced a product that supports the latter [1], and the company has indicated that it will add explicit parallelization extensions, such as unsynchronized loops, in the near future [2]. We believe that these strategies, though clearly useful, will not find wide acceptance in the Matlab community because of the complexity of parallel programming, particularly using MPI. Users who have turned to Matlab because of its simplicity will not easily relinquish that advantage.

The second approach to parallel computing in Matlab is to extend the language by adding arrays that are global but distributed across the memories of a parallel processor. This strategy, which is employed by Interactive Supercomputing [3] and several research projects [4, 5, 6], involves the use of parallel versions of matrix operations, with the required communication built in. Several of these use ScaLAPACK, the parallel version of LAPACK, for these operations. While this approach is extremely successful for a number of problems, its performance is limited on problems not well suited to the set of distributions supported by the underlying parallel libraries. To support a wider variety of distributions, the primitive matrix operations would need to be rewritten by hand, with communication routines coded to handle the data exchange needed for operations on arrays with different distribution. If there are *n* different distributions and *m* operations, this would require that nm different operation implementations and n^2 communication patterns be explicitly coded, which would be a significant implementation effort.

The goal of our project at Rice University is to use automatic methods to make it feasible to extend Matlab with a rich collection of array distributions without the implementation effort of explicitly recoding the library operations for each distribution. Furthermore, to achieve the highest possible performance, we plan to translate the extended Matlab programs to Fortran plus MPI.

The language we are developing, which we call *Matlab D*, extends Matlab with the addition of primitive operations to allocate distributed arrays with different distribution types taken as parameters, much as sparse arrays are allocated in the current Matlab product. The compilation system for this language is based on two existing technologies: a High Performance Fortran (HPF) compiler developed at Rice over the past decade and a compiler generation framework called *telescoping languages*.

HPF is an extension of Fortran 90 with directives for the compiler about data distributions [7]. The Rice HPF compiler generates Fortran code with MPI calls using the directives to determine how to partition the data and computation among the processors. It generates all of the interprocessor communication operations required to correctly implement the program and performs many optimizations to reduce the cost of communications [8].

Telescoping languages is a strategy developed at Rice for generating optimizing compilers for domain-specific languages based on libraries of fundamental domain operations. As shown in Figure 1, the telescoping languages system operates in two phases. The Library Analysis and Preparation Phase, which is performed infrequently, specializes the domain libraries in advance to the calling

This material is based on work supported by the National Science Foundation under Grant No. CCF-0444465, and also by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory. Matlab is a registered trademark of The MathWorks, Inc.

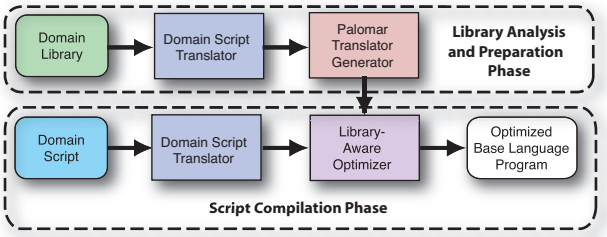

Fig. 1. Telescoping-Languages Framework

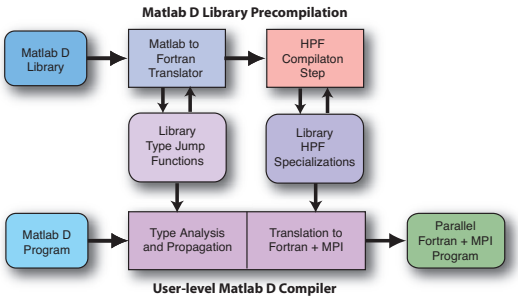

Fig. 2. Matlab D Framework

contexts that will likely exist in programs presented to the generated optimizing compiler. The Script Compiler Phase analyzes the user program to determine the types of operands at each library call and select the variant of the called routine that is optimized for those operand types. The telescoping languages framework is being used to produce a compiler for sequential Matlab programs [9].

Our strategy is to use these two technologies in combination to produce the specialized operations and communications needed to implement Matlab D. In advance, the Matlab compilation framework would produce generic HPF routines for each mathematical array operation and each communication operation. These routines would then be specialized by the HPF compiler for each supported pattern of distributions for the operands. Then, when the user presents a Matlab D program, the compiler would perform an analysis to determine the distribution type for each array in the program and, for each operation, select the specialized primitives needed to implement it. Thus, this strategy automates the process of hand coding operations for each distribution. The Matlab D compiler will be able to produce good code for each distribution type it supports. This will make it possible to significantly expand the set of applications that can be efficiently implemented in Matlab. For example, the highest performance on the NAS parallel benchmarks is achieved by using a distribution called *multipartitioning*, which is currently supported by the Rice compiler [8, 10].

This paper gives an overview of our strategy for Matlab D, focusing on the issue of data movement. An extended example shows how the compilation process achieves high performance without sacrificing productivity.

2. MATLAB D STRATEGY

The Matlab D compiler [11] extends both phases of the telescoping-languages compiler. In the first phase, shown in the top of Figure 2, the compiler specializes the library routines and generates multiple variants in HPF. After library precompilation, the compiler translates and specializes Matlab scripts using the specialized library routines, as shown in the bottom of Figure 2.

The first step in the library preprocessing phase is to perform type and distribution analysis. Matlab is a weakly typed language, so, in order to translate the routines to Fortran, a strongly typed language, valid type configurations for the variables must be determined. As part of type analysis, the compiler performs distribution analysis to track the best possible distributions of the arrays through the program based on how they are declared and used.

The compiler generates a variant for each valid type configuration found. The compiler determines where redistribution is needed based on the distributions for each variant. For example, if arrays B and C are added in an operation to form array A but their elements are not aligned correctly on the processors, the compiler inserts calls to redistribute B and C before the operation. The element-wise array routines operate on local data and do not perform communication.

Where possible, the compiler specializes calls by replacing them with calls to the variant of that routine best specialized for the context. The compiler then translates the variant to HPF. After the HPF variants are produced, the HPF compiler produces Fortran with MPI from each of the variants, and a Fortran compiler generates an object file from each of the Fortran with MPI variants. Information about the generated variants is recorded in the specialization database.

User-script compilation is similar to library precompilation. The user-level compiler uses the specialization database and the type information from the library preprocessing phase. The compiler performs type and distribution analysis, and based on the distribution information, determines where to insert calls to data movement routines. The compiler also uses the type information to uncover optimization opportunities, such as using shadow regions on the processors in order to avoid local copies during shift operations.

During specialization, the compiler replaces library calls in the script with calls to the specialized variant best-suited for the calling context. After specialization, the script is translated to Fortran with calls to the precompiled variants. By using calls to precompiled variants, the user scripts can be translated directly to Fortran, instead of to HPF, since the data movement and partitioning required is contained in variants of the library routines called.

3. COMPILATION EXAMPLE

This section describes user script compilation in more detail using the following stencil calculation as a running example.

```
A = distributed1(n, block)
B = distributed1(n, block)
C = distributed1(n, block)
...
A(2:n-1) = B(1:n-2) + C(3:n)
```

Distribution analysis shows that the elements accessed in the addition are not aligned on the processors. The data in B needs to be shifted by 1 and the data in C needs to be shifted by -1 to be aligned correctly with A. To mark that data movement is needed, the compiler inserts copies to temporaries. The compiler also replaces the Matlab primitive "+" with a call to the generic add library routine.

```
A = distributed1(n, block)
B = distributed1(n, block)
C = distributed1(n, block)
...
T1 = get(B, 1, n-2)
T2 = get(C, 3, n)
A(2:n-1) = add(T1, 1, n-2, T2, 3, n)
```

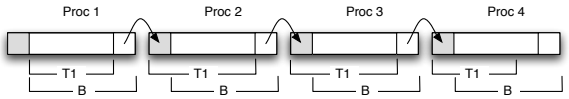

Fig. 3. Overlap areas (shaded) for array B over four processors

The compiler next translates the script into Fortran. Calls to MPI are encapsulated in the data movement library routines. The compiler generates templates, or virtual arrays, to guide array alignment. A template is created with the chosen distribution and size, and arrays are aligned on the processors relative to the template.

```
call mpi_init(ierr)
...
! create a template descriptor
t = gen_templateB(n)
! create array descriptors
A = distributedB1(n, t, 0)
B = distributedB1(n, t, 0)
C = distributedB1(n, t, 0)
...
T1 = distributedB1(n-2, t, 1)
T2 = distributedB1(n-2, t, 1)
call moveB1(B, 1, n-2, T1)
call moveB1(C, 3, n, T2)
call addB1(A,2,n-1,T1,1,n-2,T2,3,n)
...
call mpi_finalize(ierr)
```

Thus, B and C are shifted using the move routine, and then the arrays are added, each operation using the routine specialized for one-dimensional, block distributed arrays.

In this example, copying B and C into temporaries to align the data is inefficient because the majority of the data is copied locally. To save time and space, instead of copying arrays B and C, the temporaries are created to point into the existing storage for B and C, plus a small amount of space for communicated data known as overlap areas or shadow regions. Shifts of small amounts become updates of the overlap areas instead of full copies.

Figure 3 shows array B with an overlap area of one element on each processor. The temporary array T1 refers to the same storage, but beginning in the overlap area on all processors but the first. The size of the overlap area is determined during type inference, and calls to move are replaced with calls which update the overlap areas during specialization.

4. EXPERIMENTAL RESULTS

We evaluated our strategy on two stencil codes running on an AMD cluster with 275 dual core 2.2 GHz Opteron CPUs. Each core of the cluster has 2GB of memory. Each Opteron processor has a 1MB L2 cache. Processors are interconnected with a fast Cray RapidArray.

The two codes tested in this experiment are a 1D 2-point stencil using two double precision arrays each with 16M elements, and a 2D 4-point stencil using two 4K $\times$ 4K double precision arrays. Both arrays were partitioned across the processors with BLOCK distribution. The generated code was compiled with the PGI Fortran compiler *pgf77* 6.0.2 with -O3 and linked with the MPICH library version 1.2.6.

We compare the performance of the generated Fortran versions (both parallel and sequential) with sequential Matlab versions using Matlab 7.2. For each experiment, we present the best execution time from multiple runs. To illustrate the benefit of using overlap areas to avoid local copying, we timed both stencil codes with and without the use of overlap areas.

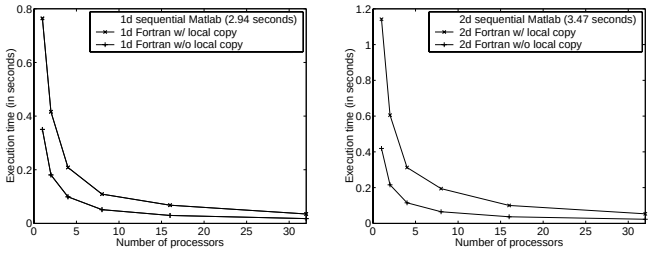

Fig. 4. Execution time of the stencil computation

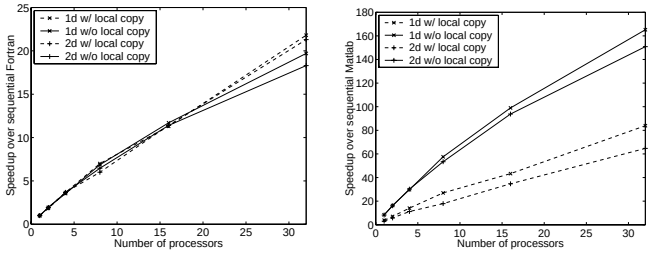

Fig. 5. Speedups over sequential Fortran and Matlab codes

Figure 4 shows the execution time of the generated parallel codes on up to 32 processors. The sequential Matlab execution time was 2.94 seconds for the 1D case and 3.47 seconds for 2D case. Both sequential Fortran codes with and without local copying performed significantly better than the Matlab code, with speedups of 3.8 and 8.4 respectively. Eliminating local data copying improved performance by up to a factor of 2.7.

Figure 5 presents the speedups of the parallel Fortran codes over the sequential Fortran and Matlab codes. The parallel codes scale well up to 32 processors. The less than perfect speedups are due to the smaller data sizes on larger number of processors. The speedups over the sequential Matlab include the contributions from the fast sequential execution of the generated Fortran codes.

5. IMPLEMENTATION STATUS AND FUTURE WORK

We are currently extending our sequential Matlab compiler to construct the Matlab D compiler. We have implemented a preliminary distribution analysis using the existing type analysis framework, which already has support for intrinsic type, size, and shape problems. We are adding our algorithms to insert calls to the move routines and to handle shadow regions to the specialization structure. Code generation in the backend will be extended to output HPF directives and MPI initialization.

Once the prototype is constructed, we plan to extend our distribution analysis algorithm to recognize that the distribution of an array can change during execution through redistribution. The current implementation relies on the assumption that each array has the same base distribution throughout the program, and that arrays can only be shifted. We also plan to extend the base library to efficiently handle routines, such as matrix multiplication and FFT, which cannot avoid communication by redistribution in advance.

6. RELATED WORK

There are a number of projects focused on the parallelization of Matlab. Several of these are similar to Matlab D in the sense that they compile Matlab to C or Fortran, either generating assertions for a parallel Fortran compiler or calls to parallel numerical libraries. Included in this class are Falcon [12], Otter [13], CONLAB [14], and ParAL [15]. While these projects obtain improved performance by compiling to low-level languages, they do not take advantage of the performance possible from being able to automatically specialize libraries for additional distributions.

Another strategy is to implement MPI and PVM routines that users can call directly from Matlab scripts, as in MatlabMPI [16] and the MPI and PVM Toolboxes [17]. These toolboxes give the user more control over the details of parallelism but, because MPI and PVM are low-level and difficult to use, they should result in lower levels of productivity than global array parallelization.

Some projects, such as Star-P [3], LAPACK for Clusters [6], MultiMATLAB [18], and the Matlab Distributed Computing Toolbox and Engine [1], provide mechanisms for users to send commands to a parallel backend. These, along with the projects implementing MPI/PVM, rely on the Matlab interpreter, which can be significantly slower than compiled Fortran. Also, the specialization pass in the Matlab D compiler can find and perform optimizations which these projects do not, such as the use of overlap areas.

7. CONCLUSION

The global distributed array strategy, which Matlab D represents, is the most promising approach to introducing parallelism into Matlab without significantly reducing the productivity of end users. By using telescoping languages coupled with HPF compilation, the Matlab D compilation framework makes it possible to support a rich collection of distribution without laborious hand coding of array operation libraries. It also keeps compilation times low by precompiling the operation and communication libraries. Our preliminary results for small stencil kernels show that the overall approach promises to yield dramatic performance improvements over sequential Matlab implementations, with excellent scaling. Over the coming few months, we plan to expand our implementation and experiments to a much larger class of applications and distributions.

8. REFERENCES

[1] The Mathworks, "Distributed computing toolbox," http://www.mathworks.com/products/distribtb/.

[2] Cleve Moler, "HPC challenge awards: Class 2 - productivity," presented at Supercomputing 2005, Seattle, November 2005.

[3] Ron Choy and Alan Edelman, "Parallel MATLAB: Doing it right," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[4] Jeremy Kepner and Nadya Travinin, "Parallel Matlab: The next generation," *High Performance Embedded Computing*, 2003.

[5] Aakash Dalwani, Neil Ludban, David Hudak, and Ashok Krishnamurthy, "High-performance parallel Octave on Itanium using ParaM," presented at the Gelato ICE Itanium Conference and Expo, San Jose, April 2006.

[6] Piotr Luszczek and Jack Dongarra, "Design of interactive environment for numerically intensive parallel linear algebra calculations," in *Proceedings of the International Conference on Computational Science (ICCS) 2004, Krakow, Poland*, 233 Spring Street, New York, NY 10013, 2004, Springer.

[7] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele. Jr., and Mary E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, 1994.

[8] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi, "High Performance Fortran compilation techniques for parallelizing scientific codes," in *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1998, pp. 1–23, IEEE Computer Society.

[9] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey, "Telescoping languages: A system for automatic generation of domain languages," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 387–408, 2005.

[10] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarria-Miranda, "Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 887–911, 2003.

[11] Mary Fletcher, Cheryl McCosh, Ken Kennedy, and Guohua Jin, "Strategy for compiling parallel Matlab for general distributions," Tech. Rep. TR06-877, Rice University, Houston, TX, 2006.

[12] Luiz De Rose and David Padua, "A MATLAB to Fortran 90 translator and its effectiveness," in *ICS '96: Proceedings of the 10th international conference on Supercomputing*, New York, NY, USA, 1996, pp. 309–316, ACM Press.

[13] Michael J. Quinn, Alexey Malishevsky, and Nagajagadeswar Seelam, "Otter: Bridging the gap between MATLAB and ScaLAPACK," in *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 1998, p. 114, IEEE Computer Society.

[14] Peter Drakenberg, Peter Jacobson, and Bo Kgstrm, "A CONLAB compiler for a distributed memory multicomputer," *The Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 814–821, 1993.

[15] Igor Z. Milosavljevic and Marwan A. Jabri, "Automatic array alignment in parallel Matlab scripts," in *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 1999, pp. 285–289, IEEE Computer Society.

[16] Jeremy Kepner, "MatlabMPI," *Journal of Parallel Distributed Computing*, vol. 64, no. 8, pp. 997–1005, 2004.

[17] Javier Fernández, Antonio Cañas, Antonio F. Díaz, Jesús González, Julio Ortega, and Alberto Prieto, "Performance of message-passing MATLAB toolboxes.," in *High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal, 2002*, 2002, pp. 228–241.

[18] Anne E. Trefethen, Vijay S. Menon, Chi Chang, Grzegorz Czajkowski, Chris Myers, and Lloyd N. Trefethen, "MultiMATLAB: MATLAB on multiple processors," Tech. Rep., Cornell University, Ithaca, NY, USA, 1996.