Implementing a Run-time Library for a Parallel MATLAB Compiler

Alexey Malishevsky

Major Professor: Dr. Michael J. Quinn

A Research Paper submitted to
Oregon State University
in partial fulfillment of the the requirements for the degree of
Master of Science

Department of Computer Science
Oregon State University
Corvallis, OR 97331

Presented: April 9, 1998

Acknowledgments

Thanks to Dr. Michael J. Quinn for offering me a part in his remarkable Otter project, for letting me participate in the HPDC conference, and for supporting my stay here while I have been working on my degree. I appreciate his helping me to learn many techniques of parallel programming and his enormous patience while waiting for me to complete my course work prior to working on the project.

Thanks to Dr. Thomas Dietterich and Dr. Margaret Burnett for serving on my committee and for giving me valuable feedback on my report.

Contents

List of Figures

Abstract

Programming parallel machines has been a difficult and unrewarding task. The short lifespan of parallel machines and their incompatibility have made it difficult to utilize them. Our goal here is to create an environment for parallel computing which allows users to take advantage of parallel computers without writing parallel programs. MATLAB is accepted everywhere as a standard of computing, being very powerful, portable, and available on many computers. Its slow speed limits its utility as a language for manipulating large data sets. However, the emergence of publicly available parallel libraries like ScaLAPACK has made it possible to implement a portable MATLAB compiler targeted for parallel machines. My project was the implementation of the run-time support for such a compiler. The system implements a subset of MATLAB functionality. Benchmarking ten MATLAB scripts reveals performance gains over the MATLAB interpreter. In addition, I have studied various data distributions and their effect on parallel performance. This report describes the parallel MATLAB compiler, the run-time library, and its implementation. It presents performance data collected on a variety of parallel platforms as well as the effect of choice of the data distributions on performance.

1 Introduction

There has been and will always be a need for high speed computers, particularly in engineering and science where various models are considered. Over time, with the development of more complicated machinery, more precise and finer-grained mathematical models have to be employed, which require increasingly powerful computers. Computational scientists have always looked to parallel machines, because they offered much higher computational power then sequentional ones. However, parallel machines are not yet widely adopted because of the relative difficulty of parallel programming and the lack of parallel software. The relatively short lifespan of parallel machines and their incompatibility have made it very difficult to develop parallel software. In addition, many users find it very difficult to develop and test a model on parallel computers.

MATLAB has changed, in some sense, the whole way of scientific computing. MATLAB combines the user-friendly interactive interface with powerful graphics, a great variety of functions which implement linear algebra algorithms, and a high level language with primitives for matrix and vector operations [14, 15]. Its widespread availability and portability of scripts to many sequential machines have made MATLAB a very popular computational tool (over 100,000 licenses have been sold). Many people find it very easy to compute a physical model or any other model in MATLAB. However, its liability is its slow performance. Because MATLAB is an interpreted language, the same computations can run an order of magnitude slower under MATLAB than when coded in FORTRAN. So, after a model is created and tested on small data sets, it must be re-coded in another language like FORTRAN before it can quickly execute on realistic data sets. So, there is a tradeoff between execution time and code development time: spending time to re-code the problem in FORTRAN or C saves CPU time to run it.

If we can automate the re-coding step, we will kill two birds with one stone: keep the development time short and execute the program quickly. To this end, we have developed the parallel MATLAB compiler, nicknamed 'Otter'. It compiles MATLAB scripts into C code, which can be executed efficiently on parallel machines. Essentially, it emits an SPMD-style program with calls to a run-time library. The successful implementation of Otter has eliminated the re-coding step in many cases. It automates the development of code for multiple-CPU computers which has been a problem for many years. Compiled MATLAB scripts execute on networks, shared memory machines, distributed memory machines, and any other machines supporting the MPI library. In addition, by distributing the data, the amount of primary memory available becomes

much greater and allows us to solve much bigger problem sizes than a MATLAB interpreter could handle. Another great benefit of Otter is that it utilizes optimized parallel linear algebra packages, like ScaLAPACK. Hence, we can take advantage of parallel libraries without learning how to invoke functions directly or worrying about data distribution. Finally, by just compiling a script, we can observe an increase in performance over an interpreter. This report describes my part of this project, the development of run-time library support for the compiler.

2 Previous Work

A significant amount of work has been done in the area of parallel MATLAB. Both parallel interpreters and parallel compilers have been developed. Parallel interpreters force users to add parallel instructions into MATLAB code. The MATLAB Toolbox (University of Rostock in Germany) generates code for networks of workstations [16]. MultiMATLAB (Cornell University) can be executed on both parallel computers and networks [20]. Parallel Toolbox (Wake Forest) is a SIMD-style programming environment which allows the user access MATLAB interpreters running on a network of workstations (NOW) [10]. Paramat (Alpha Data Corporation) is designed for PC add-on board with Digital Alpha CPUs [12].

Some compilers generate sequential programs, while others emit parallel programs. A compiler from MathWorks produces sequential C code. MATCOM (MathTools) produces sequential C++ code [13]. CONLAB (University Umea in Sweden) translates into C with calls to the PICL message-passing library [8]. This system requires the programmer to use parallel extensions to MATLAB. RTExpressa (Integrated Sensors) translates MATLAB scripts into C code with MPI calls [11]. Using a GUI interface, a programmer selects a portion of a MATLAB script to run in parallel on specified machines and processors. The types of parallelization are also specified, which can be data parallel, task parallel, round robin, or pipeline. FALCON (University of Illinois) translates ordinary MATLAB into FORTRAN 90 code [5, 6]. However, another translation step is needed to produce a parallel executable. Otter is very similar to FALCON.

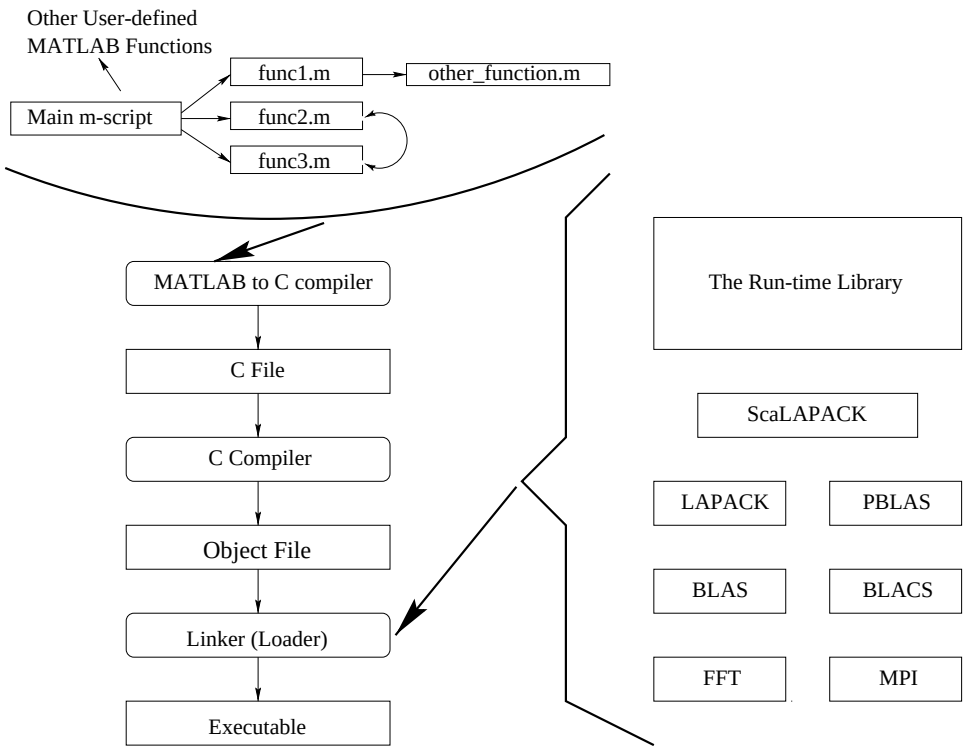

Figure 1: The Compilation Process

3 MATLAB to C Compiler

3.1 An Overview of the Compilation Process

A MATLAB program is a ".m" file containing a script (sequence of statements) together with functions, which are callable from this script. The Otter compiler is a multi-pass compiler. As a result, it allows easy additions of new optimizations by simply adding extra passes.

In the first step, the compiler constructs the parse tree for the main script to be compiled. Lex and yacc are used to construct the parser. Next, several additional links are created by the parser to construct an abstract syntax tree (AST).

In the second pass, all identifiers are resolved to be variables or functions. For every function called from the script, the same process is repeated. At the end of this step, every function is added to the AST.

In the third pass, the type, shape, and rank are determined for each variable. A variable can be of the literal, integer, real, or complex type. If possible, the compiler will determine the rank and shape of a matrix. If not, the determination will be postponed until run-time. Because

MATLAB allows variables to change type dynamically, the static single assignment representation [4] and variable renaming are used to cope with dynamically changed types. When the static single assignment is resolved for all variables, we use a type inference mechanism to propagate type and shape information through expressions and function calls (and even data files). The shape information is not essential and can be postponed until run-time (if possible, the compiler will also collect the shape information).

In the next step (expression rewriting), the AST begins to resemble a SPMD-style C program. The data distribution decisions match those made for other data-parallel languages [9]:

- scalar variables are replicated,
- all matrices are distributed, and all matrices of the same shape are distributed the same way,
- the "owner computes" rule is used,
- synchronization is accomplished via message passing, and
- one node is in charge of I/O operations.

Under these assumptions, the compiler determines which terms and sub-expressions require interprocessor communications and modifies the AST to move these terms and sub-expressions to the statement level, where they are transformed into the run-time library calls. For elementwise matrix operations, loops are generated to operate on the local data.

The sixth pass performs "peephole" optimizations. If possible, several calls to the run-time library are replaced by one call.

The final pass traverses the AST and emits C code.

If a script calls a function defined in an m-file, this function is compiled and added into the C file as a C function. Parameters and return values are defined as C function parameters. If a matrix parameter is not changed, it is passed as a reference; otherwise, it is copied to a temporary variable.

MATLAB expressions like

```
k = 10.0;
a = b * c + d * k;
```

(where `a, b, c`, and `d` are matrices and `k` is a scalar)

are translated into

```
   k_s_r = 1.000000000000e+01;
   ML__matrix_multiply(b_m_r, c_m_r, ML__tmp1);
   ML__matrix_init(&a_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp1, 0, 4);
   for (ML__tmp2 = ML__local_els(a_m_r) - 1; ML__tmp2 >= 0; ML__tmp2--) {
      a_m_r->realbase[ML__tmp2] = ML__tmp1->realbase[ML__tmp2] +
         d_m_r->realbase[ML__tmp2] * k_s_r;
   }
```

A run-time library function `ML__matrix_multiply()` performs the matrix multiplication `b * c` and saves the result into a temporary variable. Next, the compiler generates one loop to perform the addition and multiplication by a scalar (the compiler attaches suffixes to the names of variables).

Here is another example. The statement:

```
a(i, j) = 5.1;
```

is translated to:

```
   if (ML__owner(a_m_r, i_s_i - 1, j_s_i - 1)) {
      *ML__realaddr2(a_m_r, i_s_i - 1, j_s_i - 1) = 5.100000000000e+00;
}
```

Here, the owner node sets a value to the matrix element.

Also, the statement:

```
ak = a(i, j);
```

is translated to:

```
   ML__broadcast(&ML__tmp3, a_m_r, i_s_i - 1, j_s_i - 1);
   ak_s_r = ML__tmp3;
```

Here, the owner broadcasts the value of the matrix element.

3.2 Inline Loop Generation vs. Calls to the Run-time Library for Elementwise Operations

We have studied two ways to deal with elementwise matrix operations. A statement can include several matrix additions, subtractions, elementwise multiplications, multiplication/addition/subtraction/division by a scalar, etc.. A statement can be split into several stages by the use of temporary matrices and by making calls to optimized run-time library subroutines to perform these elementwise operations. Another way to deal with such elementwise operations is to include inline C "`for`" loops to actually perform them. The disadvantage of this method is that these loops are not optimized as much as loops in the run-time library could be. However, this method allows us to use a technique, called "loop fusion" [21] to merge several loops together. This avoids temporary storage in many cases and eliminates the overhead of multiple loops. We have found, that the latter method combined with "loop fusion" is usually about twice as fast as making run-time calls for pairwise operations and accounted for a significant decrease in execution time of benchmarks.

4 Looking for the Best Data Distribution

In the early stages of the project I tried to select the best data distribution for matrices and vectors by conducting the series of experiments. I tried to model the performance of dot product, matrix-matrix, matrix-vector, and vector-matrix multiplication for column-wise, row-wise, and block-wise matrix distributions and replicated and distributed vectors. I implemented every case using MPI. In total, I wrote 17 benchmark programs and evaluated their performance. The results were disappointing. For example, cases which in theory should have had the same execution time actually varied greatly. The explanation lies in the C compiler. Every such benchmark has its own implementation of the multiplication of the local blocks, resulting in different usage of index variables, differences in the number of variables, and differences in addressing styles, which cause the compiler to produce differently optimized code. In several cases, index variables were allocated on the stack, in some cases in the data segment, and in the other cases in registers. These differences can cause a two-to-one performance variation. In order to avoid this kind of problem, I created a generic subroutine to implement the multiplication of two local sub-blocks with all proper offsets and indexing. I rewrote every benchmark to invoke this function. However, I discovered another major problem. I had chosen Meiko CS-2 as the benchmark machine. While it is much more predictable than SMP's and networks, its communication bandwidth can vary from less then 1 Mbyte/sec to 40 Mbytes/sec, depending on whether the communication co-processor has already pre-paged data to transmit. The large variance in communication speed makes precise performance prediction impossible.

5 Overview of the Run-time Library

The run-time library initializes the parallel environment, does allocation, initialization, and destruction of matrices and vectors, performs operations which involve the interprocessor communication, and takes care of all other non-trivial operations.

The run-time library is written in C and incorporates a multitude of files:

- `matlab_common_interface.c` - an interface to the run-time library, externally visible functions, providing run-time selection of a subroutine of the correct type. All calls to run-time library go through this file's functions with few exceptions. It is a gateway to the run-time library.
- `matlab_generic.c, matlab_generic_trapz.c, matlab_generic_minmax.c` - collections of subroutines, like min/max and trapz, written for a generic data type (see Section 6.2 for more details);
- `matlab_common_group.c` - group communications subroutines, most of which are simply interfaces to ScaLAPACK or MPI communication subroutines;
- `matlab_common_mem.c` - memory management subroutines;
- `matlab_common_arifm.c` - subroutines for scalar arithmetic, including complex arithmetic;
- `matlab_common_scalapack_interface_mm.c` - ScaLAPACK interface subroutines which provide matrix operations (multiplication, inverse, transpose, etc.);
- `matlab_common_scalapack_interface_LA_decomp.c` - ScaLAPACK interface subroutines which provide various matrix factorization (lu, qr, svd, etc.);
- `matlab_common_scalapack_interface_misc.c` - ScaLAPACK interface subroutines which provide various other functionality, and C code, which implements some missing ScaLAPACK functionality (performs post-processing);
- `matlab_common_matrix_initialize.c` - initialization of the MATRIX data structure, and calculation of the shape of the result for various operations;
- `matlab_common_matrix_print.c` - printing functions;
- `matlab_common_load.c` - subroutines which load data from files;
- `matlab_common_coerce.c` - type coercion subroutines;
- `matlab_common_fft.c` - interface subroutines to a parallel FFT library;

- `matlab_scalapack.c` - a collection of subroutines which perform matrix initialization, BLACS context initialization, and conversion between old and new data distribution styles;
- `matlab_blacs_defs.h` - BLACS constants, structures, and the declarations of prototypes for BLACS functions;
- `matlab_generic_redefs.h` - redefinitions for `int, double`, and `complex` types for the "generic" code;

Currently, the run-time library consists of about 7400 lines of C code. Its files occupy about 190 Kbytes.

5.1 Stages of Development

The run-time library has been developed twice: the MPI version and the ScaLAPACK version.

5.1.1 The MPI Version

In order to quickly create a small subset of MATLAB functionality, the run-time library has been implemented on the top of MPI. To simplify code development, only a contiguous row-wise data distribution was used. The vectors were treated separately and were distributed evenly among all available processes. This data distribution did not provide the optimal performance; however, it gave relatively good results. The original run-time library included matrix multiplication, matrix-vector multiplication, vector-matrix multiplication, vector outer product, dot product, mean (matrices and vectors), LU factorization, 1-D and 2-D contiguous array slicing, trapz calculation (trapezoidal numerical integration [14]), / (right division, was implemented only for square matrices), \ (left division, was implemented only for square matrices), matrix inversion, matrix transpose, matrix power, and functions to support the run-time environment.

5.1.2 The ScaLAPACK Version

After the first results were collected and published [18], I completely redesigned the library and rewrote almost all of its functions. Now, it supports a much greater subset of MATLAB functionality and uses more general and efficient data distribution. I switched from the written-from-scratch C code to the utilization of publicly available parallel libraries. The ScaLAPACK library allowed us to efficiently support many linear algebra functions such as chol, lu, qr, eig, \ and / (for non-square matrices), poly, polyval, roots, rank, det, etc.. This transition has been done in several

steps. First, the old run-time library coexisted for a while with the ScaLAPACK code. Two data distributions were supported: the old row-wise data distribution and the ScaLAPACK-style data distribution (described in Section 5.1.3). Then, function by function, the old code was replaced, wherever possible, with calls to the ScaLAPACK library. As a result, the row-wise data distribution has now been totally abandoned in favor of the more general ScaLAPACK data distribution. As a result, the highly efficient optimized parallel code has been used for the multitude of linear algebra algorithms. The new functions store matrices in the FORTRAN-style column-major order.

5.1.3 Data Distributions

The Contiguous Row-wise Data Distribution In the early MPI version of the run-time library, all matrices were distributed by continuous groups of rows. This data distribution provided good speedups for benchmarks and allowed relatively easy implementation. Vectors were treated separately and were always distributed over all available nodes, regardless of their orientation.

The Block-Cyclic Data Distribution In the ScaLAPACK version, a more elaborate data distribution has been used. It provides better performance at the expense of higher implementation complexity, but the main advantage of this distribution is that it is relatively general. With only slight parameter changes, we can select row-wise, column-wise, or block-wise data distributions. For rows, we select some block size factor b and the number of rows p in the 2-D $p \times q$ virtual processor grid. The rows of a matrix are assigned to processors in an interleaving manner. This dimension is spread out evenly by allocating b consecutive rows to the first row of processors in the virtual processor grid, then b following consecutive rows to the next row of processors in the virtual processor grid, and so on, with wraparound over p sets. The columns of a matrix are distributed in a similar way over q columns of the $p \times q$ virtual processor grid. In this way, rows and columns are distributed independently in the corresponding dimensions of the 2-D virtual processor grid [7].

The Choice of the Best Data Distribution As explained later, many operations depend heavily on the data distribution strategy, so the choice of the data distribution is very important and can greatly affect the performance.

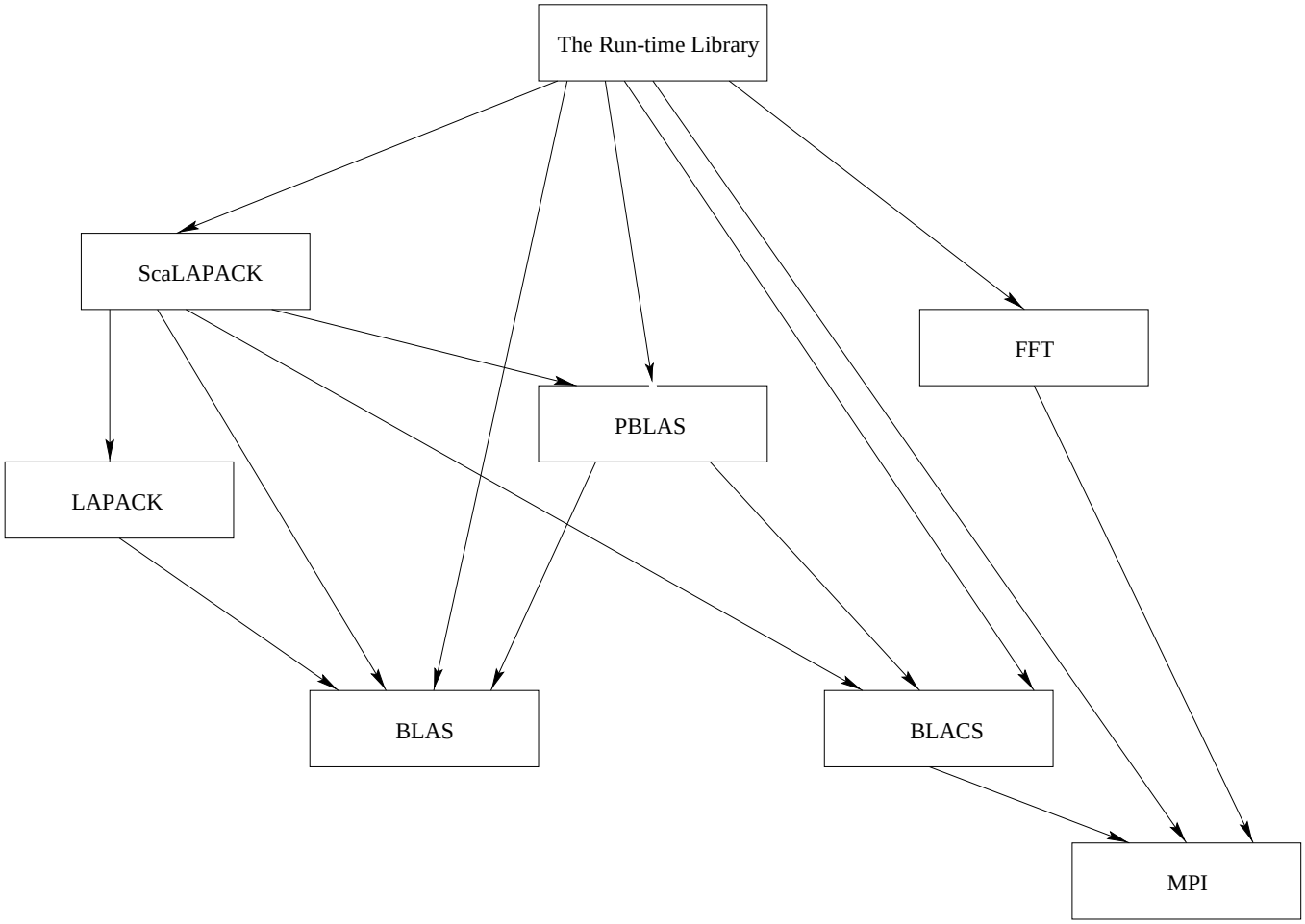

Figure 2: The Diagram of Module Usage [3]

5.2 Diagram of Module Usage

Otter includes the <matlab-to-C> compiler itself, the run-time library, and the parallel FFT package (freeware, written by Matteo Frigo and Steven G. Johnson at Massachusetts Institute of Technology). ScaLAPACK, PBLAS, BLAS, BLACS, and MPI libraries are freeware software developed by the joint efforts of the University of Tennessee, Knoxville, and Oak Ridge National Laboratory. These libraries can be downloaded in the source code from the WWW site "http://www.netlib.org" and installed in the system.

ScaLAPACK (Scalable Linear Algebra PACKage) is the scalable distributed memory parallel subset of the LAPACK (Linear Algebra PACKage) library. ScaLAPACK relies on the LAPACK library (for local computations), the BLACS library (for communications), the PBLAS library (for matrix and vector operations), and the BLAS library. The PBLAS (Parallel Basic Linear Algebra Subroutines) library is the distributed memory parallel version of the BLAS library.

The PBLAS library relies on the BLAS library (for local computations) and the BLACS library (for communications) [2]. The BLAS (Basic Linear Algebra Subroutines) library provides basic matrix and vector operations, like multiplication, dot product, etc.. The FFT library relies on the MPI library. The BLACS (Basic Linear Algebra Communication Subroutines) library is a communication layer for many distributed memory packages. It is a completely portable source-code distribution which provides the communication subroutines and manages "contexts" (virtual processor grids with distribution parameters) [7]. The BLACS library itself relies on MPI (or PVM) libraries [7]. The run-time library makes extensive use of the ScaLAPACK library (to implement linear algebra functions), the BLAS library, the PBLAS library (to provide matrix and vector operations), the BLACS library (to manage virtual processor grids and parallel environments), and the MPI library (to implement interprocessor communication for the run-time library functions). In addition, the run-time library utilizes a set of communication subroutines from ScaLAPACK's REDIST and TOOLS libraries to implement data redistribution. Figure 2 shows the dependencies between these libraries. ScaLAPACK, PBLAS, and BLACS libraries support 2-D block-cyclic data distribution.

5.3 ScaLAPACK

As we have seen, ScaLAPACK is a parallel extension of LAPACK. ScaLAPACK includes three classes of subroutines: driver, computational, and auxiliary. Driver subroutines are intended to solve complete problems, such as solving the systems of linear equations, finding eigenvalues, or calculating the singular value decomposition. Computational routines perform "distinct computational task[s]," like the LU factorization or Hessenberg form reduction [1]. Auxiliary routines are intended to perform subtasks of the algorithms and common low-level computations. In addition, ScaLAPACK provides data redistribution subroutines for all precisions, which support the copying of sub-matrices with different distribution parameters. `Real, double precision, complex`, and `double precision complex` data types are supported by ScaLAPACK subroutines. The types of matrices supported by ScaLAPACK are general band (diagonally dominant), general tridiagonal, general, Hermitian, orthogonal, symmetric, triangular, complex unitary, and several combinations of them [1].

Here is an example of what the C interface to a ScaLAPACK subroutine looks like:

```
pdgesvd_("V", "V", &n, &m,
        temp->matrix_internal2.base, &ia, &ja, temp->matrix_internal2.desc,
        val,
        U->matrix_internal2.base, &ia, &ja, U->matrix_internal2.desc,
        VT->matrix_internal2.base, &ia, &ja, VT->matrix_internal2.desc,
        work, &lwork, &info);
```

5.4 The Current Run-time Library Functionality

\	/	array slicing	chol
cond	det	dot	eig
fft	fliplr	flipud	hess
ifft	imag	interp1	inv
lu	matr.-vec. mult.	matrix mult.	matrix power
max	mean	min	norm
outer product	poly	polyval	qr
rank	real	roots	svd
transpose	trapz	vec.-matr. mult.	

Norm, rank, cond, eig, svd, roots, and poly are implemented only for the real data type. Cholesky is implemented only for symmetric and positively definite matrices. Eigenvectors are implemented only for symmetric matrices. Interp1 only supports linear interpolation. Most of the functions are implemented only with one or few options/features.

6 The Implementation of the Run-time Library

6.1 Design Principles

The design of the run-time library is based on these principles/assumptions:

- For 2-D matrices, never store at any point of time more than $O(matrix\_size/numprocs)$ data items on any node (with one exception: for I/O).
- Optimize highest complexity operations.
- Assume no specific platform or topology.
- Assume distributed memory platform (no assumption of platform is made; unavailability of shared memory is assumed).
- Use standard libraries as much as possible.
- Never duplicate the library code (see Section 6.2).
- Consider using generic cases, but only if this usage does not hurt the performance.
- Assume large data sets.
- Assume that no optimization for the latency of the run-time library is required (it allows us not to worry about the performance on small problem sizes).

6.2 Data Types

The run-time library supports three basic MATLAB data types: integer, real, and complex. In order to reduce the code size, eliminate code duplication, and improve maintainability, code has been written for only the generic type. The code uses `TYPE` in all data declarations and function names, like `trapz_TYPE`. This generic code is included and compiled three times. Every time, `TYPE` is redefined to be `int`, `double`, or `complex`. Also, every time, the function names are redefined. So, for example, for trapz, a C compiler actually sees three functions: `trapz_int`, `trapz_double`, and `trapz_complex`. On the higher level, the proper function is selected at run-time, depending on the type. As a result, the code is efficient and easy to maintain. In formulas for integers and reals, C expressions are directly used; but for the complex data type, function calls are made (like `add_complex()`). All of these are selected at compile time by the compiler directives `#ifdef`, `#else`, and `#endif`.

6.3 Memory Management

Frequently, the run-time library needs some temporary storage. The simplest method would be to use `malloc()` and `free()` calls, but this would be too slow. Therefore, because the usage of the temporary storage goes in stack-like fashion, a stack is used to allocate any temporary storage needed inside a function call. This storage is kept as long as a function call is active; when the call completes, this storage is deallocated. At the beginning of each function which needs temporary storage, we save the current state of the stack by calling `stack_start()`. Then, we allocate memory from the stack by `stack_alloc(n)` calls, and, at the end, we reset the stack's state to the state prior to these allocations by the call `stack_reset(frame)` (where `frame` is an integer, returned by `stack_start()`). For each memory allocation, we just take `n` bytes from the fixed size array and increment the current pointer by `n`. A function `stack_reset(frame)` sets the pointer to the `frame` position, effectively freeing all allocations between `stack_start()` and `stack_reset(frame)`. So, every function is effectively contained between the pair of calls `stack_start()` and `stack_reset()`. During the initialization stage the `stacksize` parameter is set and can be adjusted by a command-line parameter.

On the other hand, when we need permanent storage, we use the `heap` array (for example, `ML__matrix_init()` and `ML__init()` calls for all matrices external to the run-time library which are defined by the compiler). It is similar to the `stack` except it cannot be reset and reused. These methods make memory allocations very fast, because they are trivial. These arrays are allocated during the initialization stage. If the `heap` array is used up, additional storage is allocated as needed. However, when the `stack` is overflowed, the program is terminated. The sizes for the `stack` and `heap` are specified by command-line parameters and can be adjusted before the running of the program. In addition, when memory is a constraint, we can turn on dynamic memory management, which allocates and deallocates memory as needed from the global heap. This makes the program run much slower, but reduces the possibility of thrashing (the data set cannot fit into the primary memory) because of the smaller memory requirements. Here, the memory is allocated and deallocated through `malloc()` and `free()` calls.

6.4 The Type and Shape of Matrices

The run-time library takes care of the type coercion between arguments, the shape and rank of the result; and the compiler takes care of the correct type propagation. The shape and rank of a

matrix are calculated at run-time. The size of the result is computed and set inside any call into the run-time library, except for a `shift` function. Inside a run-time library call, arguments are coerced to the type of the result with few exceptions. The type coercion of the arguments is done via a simple call to `conform()` function. If the types do not match, it allocates the space for a matrix of the required type and coerces data to this type. Otherwise, it just returns a pointer to the same matrix.

Currently, the compiler employs the variable renaming technique and, thus, all results are of the general type of the arguments (with few exceptions). The reason why the library itself does not select the type of the result lies in allowing the compiler to optimize the management of dynamic types. In other words, instead of renaming and copying data each time the type changes, the compiler could set the type once and never change it (what it actually did before variable renaming has been implemented). So, the compiler just allocates 1x1 matrices, and the run-time library changes their size as needed. However, the compiler sets the dimensions of a matrix when elementwise operations are performed on matrices, and C loops are generated by the compiler to do this.

6.5 Initialization and Data Structures

The initialization of the run-time library includes the parsing of the command-line parameters, setting the block sizes and the shape of the virtual processor grid, the size of the stack and the size of the heap, the allocation of memory for the heap and stack; and the initialization of the MPI library, the BLACS library, BLACS contexts, communication groups, and arrays of pointers to functions.

6.5.1 The Initialization of the MPI Library and the Creation of Communication Groups

First of all, the MPI library is initialized via a `MPI_Init` call. Next, the number of processors and the coordinates of each processor in the virtual processor grid are obtained. After the BLACS initialization, two communications groups are created, based on the BLACS 2-D virtual processor grid (vertical: columns, and horizontal: rows).

```
typedef struct rt_matrix
      {
       void *base_addr;          // the address to the local data
       int *intbase;             // ---||---
       double *realbase;         // ---||---
       complex *complexbase;     // ---||---
       int data_type;            // the data type
       int dims;                 // 1 - for vectors, 2 - for matrices
       int rows;                 // the number of rows
       int cols;                 // the number of columns
       int local_els;            // the number of local elements
       int orientation;          // vector's orientation: row or column

       matrix matrix_internal,   // matrices tied to specific BLACS contexts
              matrix_internal2,
              matrix_matlab,
              matrix_single,
              matrix_single_t,
              matrix_default;
          } MATRIX;
```

Figure 3: The `MATRIX` Data Structure

6.5.2 The `blacs` Data Structure, BLACS contexts, and their Initialization

First, we initialize the BLACS environment via `Cblacs_pinfo()` and `Cblacs_setup()` calls. Next, we allocate and initialize several BLACS contexts (the creation of the virtual processor grids): contexts in the shape of a row, a column, and a square (approximately); a context consisting only of the node number zero, and a context compatible with the contiguous row-wise data distribution (essentially similar to the row-shaped context). Each context describes the logical layout of the nodes. Each context is represented by an instance of the `blacs` data structure which includes the context descriptor, the dimensions of the corresponding virtual processor grid, the position of the current processor in it, and the total number of processors.

6.5.3 The Initialization of Arrays of Pointers to Function Names

Some functions have three versions: integer, real, and complex. In order to avoid the multitude of `switch` statements, arrays of pointers have been created for several frequently used functions. By simple accessing of the proper array's element, we can select the proper function by the dynamic type, ex. `(*(Zero_X[type]))(...)`. Thus, several arrays of pointers are initialized here.

```
typedef struct
 {
  int rows, cols,     // the matrix's dimensions: the number of rows and columns
      data_type,      // the data type

      row_blocks,
      col_blocks,     // the block sizes for rows and columns correspondingly

      local_rows,
      local_cols,     // the number of local rows and columns

      desc[DLEN_],    // array descriptor
      lld,            // the leading dimension
      ictxt,          // the associated BLACS context

      nprow, npcol,   // the dimensions of the virtual processor grid
      myprow, mypcol, // the position of the processor in the virtual processor grid
      transposed,     // flag, indicating whether the column-major order is used
      allocated,      // the size of the memory allocated
      local_els,      // the number of local elements
      temp;           // flag, indicating whether it is the temporary matrix
  void * base;
 } matrix;
```

Figure 4: The `matrix` Data Structure

6.5.4 Matrix Data Structures

Each matrix is represented by the `MATRIX` data structure, which includes the data type, the pointer to local data, the number of dimensions (1 or 2), the size of a matrix, etc.. In addition, it includes

```
typedef struct
 {
  int nprocs,  // the total number of processors
      mypnum,  // the id of the processor

      npcol,   // the shape of the virtual processor grid
      nprow,

      ictxt,   // the corresponding BLACS context number

      myprow,  // the coordinates of the processor in the virtual processor grid
      mypcol;
 } blacs;
```

Figure 5: The `blacs` Data Structure

several structures of type `matrix`. These structures represent matrices tied to specific BLACS contexts. They are: `internal` (the virtual processor grid: 2-D for matrices and 1-D for vectors), `internal2` (2-D processor grid, uniform for every matrix), `matlab` (the row-wise simulation of data distribution), `default` (the default data distribution which currently points to `internal` or `matlab`, what a compiler actually sees outside the library), and `single` (the one node context, needed to gather/scatter matrices). For 2-D matrices, `internal` equals `internal2`, and the data are allocated only once. For vectors, on the other hand, the context depends on the vector's orientation. For row vectors, the virtual processor grid has the shape of a single row, and for column vectors it has the shape of a single column. In other words, the vectors are treated separately. Their data are duplicated in two contexts: the 1-D virtual processor grid (vectors are effectively distributed on all available processors), and the 2-D virtual processor grid, so the vectors are just a special case of 2-D matrices (required for many ScaLAPACK functions). By default, vectors are distributed on the 1-D processor virtual processor grid, and the data are redistributed when needed. This provides good data distribution for vectors when we perform vector operations, and it does not hurt to redistribute vectors occasionally. As a result, the transposing of vectors is trivial (just copy local data, if needed).

The `matlab` data structure includes blocking factors for both dimensions, the data type, the global dimensions, the local dimensions, the number of local elements, the pointer to the local data, the amount of allocated bytes, the allocation flag, the BLACS context descriptor, the dimensions

of the corresponding virtual processor grid, and the position of the processor in this grid.

During the matrix initialization, these structures are filled, and memory for the the local data is allocated if necessary. During the execution, matrices are frequently re-initialized to different sizes. Calculations are very insignificant. Also, the local space is re-allocated only in the case when the needed amount of memory becomes greater than allocated; so, if the memory requirements become smaller, no memory re-allocation is done. The space can be allocated from the stack or heap. If dynamic allocation is turned on, re-allocation is done every time when the changes of the memory requirements for the local data block exceed a predefined threshold.

6.6 Error Handling

Because the compiler cannot perform all error checking during the compilation stage, it is up to the run-time library to give necessary feedback to the user about all errors encountered during the execution, such as incompatible matrix dimensions or out of range array references (generated normally by the MATLAB interpreter). Normally, the program is terminated with an appropriate error message, such as "incompatible matrix dimensions in outer product." In addition to the user generated errors, internal errors, such as running out of memory, are handled in a similar way. If any error occurs, the `ML__error` function is invoked, which normally prints an error message and terminates the program. Sometimes, errors are not fatal. The run-time library detects non-fatal errors and attempts to correct them by restarting the process. It can be a little tricky because several programs need to be restarted. Because it is an SPMD-style program, some errors will be generated by all processors. In this case, we reset the calculations, inform the user, and restart the program with corrected parameters.

6.7 Run-time Statistics

Sometimes, it is useful for a user to know some run-time statistics of a script. By a command-line parameter, the memory allocation statistics, such as stack and heap usage, memory allocation overhead, the shape of the virtual processor grid, the block size, and other data can be printed by the program.

6.8 Implementation of Functions

All functions are implemented based on the assumption of large problem sizes. The implementation is scalable, and the data are distributed evenly among all available processors. Some functions are available in three data types: integer, real, and complex.

6.8.1 Trapz

In `trapz(x,y)`, which calculates a discrete integral defined by vector x and matrix y, we assume that $y \gg x$. First of all, we all-gather the x vector (we could avoid doing this, but this takes a very negligible amount of time and space). In the original implementation, with no communication, we computed the inner part based on the assumption of the contiguous row-wise data distribution. Next, in order to finish the computations, we needed to communicate the border rows. Each node sent its first row to its predecessor and the last row to its successor. The communications and computations were efficiently overlapped: first, we initiated the data transfers; second, we performed local computations; third, we attempted to receive data; forth, we completed the rest of computations. Here, processors attempted to receive the data only when they actually needed them. In addition, we proceeded row by row, in order to utilize the cache effectively (in the MPI version of the run-time library, matrices were kept in the C-style row-major order). In the ScaLAPACK version, when the data distribution became more general, the trapz algorithm had been simplified. If the virtual processor grid has a single row shape we proceed directly, working only on the local data, and redistributing the resulting vector at the end. If the shape of the virtual processor grid is not a single row, we redistribute the matrix y, so y is distributed over the virtual processor grid with a row shape. In addition, $\frac{1}{2} \times (x_i - x_{i-1})$ multipliers are pre-calculated and stored to improve the performance.

6.8.2 Array Slicing

The ability to use array slicing is very important for MATLAB scripts. It is done here by using two mirror functions: `redistribute` and `shift`. The problem with array slicing is that multiple-node data redistribution is hurt badly by the interprocessor communication.

Redistribute This function takes a matrix and copies its rectangular sub-matrix into a new matrix. In the original implementation, we "compacted" the data first, removing the empty space

at the left and right sides. Next, we needed to redistribute the compacted rows. For each node, we figured out what blocks of contiguous rows we would need to send and where. Next, we initialized all data transfers (if the source and destination were the same, we used memory copy). It was important to use non-blocking sends here because, if the data set was large enough, MPI could deadlock, waiting for the corresponding receive. Next, we performed the corresponding receives. Special care was taken for vectors.

In the latest version, we use a ScaLAPACK function `Cpdgemr2d/Cpzgemr2d/Cpigemr2d` to accomplish the data redistribution.

Special care is taken when the 1-D array slice is used for a 2-D matrix. The easiest way is to gather/scatter on one node, but this approach is too inefficient and we also do not want to gather a 2-D matrix on one node (this would contradict our distributed memory criterium). An algorithm for extracting a 1-D array slice from a 2-D matrix and storing it into a 1-D vector is the following:

1. Determine the borders where this 1-D array slice is contained within a given 2-D matrix (starting and ending rows and columns).
2. Extract this rectangle by redistributing the original 2-D matrix to the new smaller 2-D matrix.
3. Transform a 2-D matrix into a 1-D vector by creating the two virtual processor grids for a matrix and for a vector in such a way that local elements of the matrix correspond directly to the local elements of the vector.
4. Redistribute this pseudo-vector to its default virtual processor grid, and copy local data to the vector storage.
5. Use 1-D redistribution to extract the sub-vector from the vector which actually needs to be put to the destination.

The point here is that we never have more than $\theta(n^2/numprocs)$ elements on any node. This is a little slower than gather/scatter approach, but it achieves some speedup on multiple processors.

Shift The shift function is the exact mirror of the redistribute function. It copies a matrix into the rectangular sub-matrix of another matrix. Its implementation is the exact mirror of the redistribute function, so it will not be described here. In the ScaLAPACK implementation, the

starting position is specified in the destination matrix, as are the number or rows and columns to be copied from the source matrix.

6.8.3 Matrix Multiplication

In the original implementation, p iterations were performed (where p is the number of processors). Each iteration performed partial multiplication of the part of the local block of matrix A by the whole received block of matrix B (on i-th iteration, i-th node broadcasted its local part of matrix B). Totally, p iterations were performed, and eventually the whole matrix B was broadcasted to everybody. However, only $\theta(n^2/p)$ elements were stored on any node at any one time. In the ScaLAPACK implementation, this is done completely via a PBLAS function `pdgemm/pzgemm`, which has a number of restrictions (the matrices must be distributed the same way, and block sizes must be the same in both dimensions). Because integers are not supported here, we convert integers to reals and convert the result back to integers. The use of integer matrix multiplication is actually detrimental to the performance. I have found that floating point operations are actually executed faster then integer operations on a Sparc CPU. If any of the arguments is a vector, we select the proper function: matrix-vector multiplication, vector-matrix multiplication, outer product, or dot product. These specialized functions are faster then the generic matrix multiplication.

6.8.4 Matrix-Vector Multiplication

In the original implementation, the 1-D vector was gathered on every processor. Next, with no communication (due to row-wise data distribution), we calculated the local data (multiplied the local part of the matrix to the whole vector to get the local piece of the resulting vector). In the ScaLAPACK version, we utilize a PBLAS function `pdgemv/pzgemv` to perform this multiplication. We redistribute the vector from its 1-D processor grid to the 2-D virtual processor grid, as PBLAS requires, and redistribute the result back. The same trick for the integer matrix is used as in the matrix multiplication.

6.8.5 Vector-Matrix Multiplication

In the original implementation, we calculated the partial sums, by performing the multiplication of the local part of the 1-D row vector by the local part of the matrix. Next, we performed the global reduction of n elements simultaneously to calculate the whole resulting vector. Finally,

we distributed this resulting vector over all nodes. In the ScaLAPACK version, we use the same PBLAS function `pdgemv/pzgemv` as in matrix-vector multiplication. This subroutine allows us to specify whether the original or transposed matrix is to be used in the multiplication, so we use the fact that $x * A = A^T * x^T$. We transpose vector x (it is very fast) and call this PBLAS function `pdgemv/pzgemv`. The same method for the integer type is used here as in the matrix multiplication.

6.8.6 Local Multiplication

In the original version, in matrix-matrix, matrix-vector, and vector-matrix multiplication, we used the same function to multiply the local parts of matrices. This function was highly optimized, and it transposed the local data in the matrix B to optimize the cache performance. Also, we pre-calculated and stored the addresses of each row for both local matrices A and B. So, indexing was speeded up, because referencing `A[i][j]` was very fast. Then, for the inner loop, we calculated the starting address for each sub-array and performed the dot product for two vectors. All inner loop calculations were kept in registers, except data themselves. However, in the ScaLAPACK version, this function has been removed.

6.8.7 Outer Product

In the original version, we gathered the vector y on every node and performed the multiplications of the local part of vector x by the vector y in order to calculate the local data. No communication was required at this point. In the ScaLAPACK version, we use the function `pdger/pzgeru` to accomplish this job. Both vectors are redistributed temporarily to the 2-D virtual processor grid. For the integer type, we use the same method as in the matrix multiplication.

6.8.8 Dot Product

Originally, dot product was performed by local calculations and the global all-to-all reduction via an MPI library call. But now, for the local computations, we utilize a BLAS function `ddot/zdotu`. Both vectors are kept distributed on their corresponding 1-D contexts. The same method is used here for the integer type.

6.8.9 Mean

The local means are calculated, and the result is globally reduced. A matrix mean (a mean is calculated for every column) can be calculated by calculating the local means for every local part of every column, and the result is reduced in contexts of vertical communication groups. So, the result represents the vector of means for corresponding columns. Later, we redistribute this vector to its default 1-D virtual processor grid. All three types are supported here.

6.8.10 Matrix Inversion

In the original version, the matrix inversion was implemented by doing LU factorization and by using forward and backward substitutions for every column of the identity matrix. The forward and backward substitutions were implemented in a trivial way by just doing them simultaneously for every column of the identity matrix, and then combining the result. In the ScaLAPACK version, matrix inversion is done by the function calls `pdgetrf/pzgetrf` (to compute LU factorization) and `pdgetri/pzgetri` (to compute actual inverse).

6.8.11 Matrix Back Division

Matrix back division in MATLAB means the linear system solving: $X = A\backslash B \Longrightarrow AX = B$. X and B can be matrices, which means that several linear systems are solved simultaneously. If matrix A is not square, another function is called which solves the least squares problem. It is determined at run-time. Originally, "\" was implemented only for square matrices, such that $A\backslash B = A^{-1} \times B$. Now, in the ScaLAPACK version, we can handle any matrix which is allowed by MATLAB by performing the pair of ScaLAPACK function calls: `pdgetrf/pzgetrf` (the LU factorization) and `pdgetrs/pzgetrs` (linear system solving). For the least squares problem, a ScaLAPACK function `pdgels/pzgels` is called.

6.8.12 Matrix Division

Matrix division was implemented originally for only square matrices, such as $A \times B^{-1}$. Now, we use the precise formula for the matrix division: $A/B = (B^T\backslash A^T)^T$. Here, we allocate several temporary matrices, transpose them, perform the back division, and transpose the result back.

6.8.13 Transpose

The original implementation of the transpose function included a personalized all-to-all broadcast. First of all, for every pair of nodes, we selected the data to be sent from node i to node j. Then, we copied these data into buffers. After this, we transferred these data from all senders to all receivers simultaneously by initializing all transfers, and then accepting the data by the destination nodes. When received, the data were copied to their correct places. This implementation was highly scalable. A vector was not physically transposed, only its `orientation` field was changed and the local data copied. Now, we use a ScaLAPACK function `pdtran/pztranu` to transpose the matrix. For vectors, we simply copy the local data. For matrices of the complex type, data are also conjugated (like MATLAB does). Because only complex and real types are supported, for an integer matrix, we use a subroutine for the floating data type to trick the ScaLAPACK library (floats and integers are the same length here). If this does not work, the integer data type can always be converted to the double precision data type.

6.8.14 Eigenvalues and Eigenvectors

In order to calculate eigenvalues and eigenvectors of the symmetric matrix, we use a ScaLAPACK subroutine `pdsyev`. Eigenvalues are replicated on each node, so we distribute them over all available processors. For non-symmetric matrices, we use a ScaLAPACK function `pdgeevx` to calculate eigenvalues. There is no support yet for eigenvalues of complex matrices or for the non-symmetric eigenvectors problem.

6.8.15 Svd

For the singular value decomposition, we call a ScaLAPACK function `pdgesvd` for the real data type. Later, once ScaLAPACK will incorporate the singular value decomposition for the complex data type, the run-time library will support it.

6.8.16 Roots

Here, we employ the same algorithm as the MATLAB uses by using the companion matrix. The matrix consists of zeroes, except for elements on its first sub-diagonal, which are ones, and for the first row which consists of the coefficients $c(2:n)$ divided by $-c(1)$, where c is the vector of the coefficients. The roots are the eigenvalues of this companion matrix. Matrix allocation, zeroing,

and setting the sub-diagonal are done in parallel, but the filling of the first row takes $\theta(n)$ time, so it is not parallelized here because the calculation of eigenvalues takes $\theta(n^3)$ time.

6.8.17 Rank, Norm, and Cond

All these values are calculated the same way, so only one function is used here. First of all, the svd (singular value decomposition) is calculated (only singular values, not vectors). Next, for the norm, the largest singular value is found; for the rank, the number of singular values greater then some threshold is found; and, for the conditional number, the smallest and the largest singular values are found and their ratio is computed. Because these computations take only $\theta(n)$ time, and the calculation of the svd takes $\theta(n^3)$ time; rank, norm, and cond are computed sequentially, once the svd is computed. Currently, the complex data type is not supported here, because the svd does not support it.

6.8.18 Determinant

In order to find the determinant of a matrix, we find the LU factorization and compute the product of the diagonal elements of the upper triangular matrix U. Because, the product takes only $\theta(n)$, compared to $\theta(n^3)$ time for the LU calculation, the product calculation is not parallelized. In addition, because the result is pivoted, we compute the determinant of the permutation matrix P, which is 1 or -1, and multiply it by the computed product. The idea behind the computing of the $det(P)$ relies on the definition of the determinant.

Take an $n \times n$ original matrix. For every element in the first row (or column), compute recursively the determinant of $(n-1) \times (n-1)$ matrix (derived from the original by crossing the i-th row and j-th column), and multiply it by the $A_{i,j} \times (-1)^{i+j}$ (i and j are indices of the picked element). Add all these products.

After constructing the permutation matrix P, it is redistributed to the contiguous column-wise data distribution, and condensed into a single vector v. An element v_i is the position of 1 in i-th column in matrix P. Then, we all-gather this vector and compute the determinant of P, which is 1 or -1. Because matrix P has only a single 1 per row and per column, we only need to compute the determinant recursively once for each column, considering the element in a column which is equal to one. The matrix condensing is done in parallel, but the actual determinant calculation is done sequentially because it is only $\theta(n^2)$. The memory requirements are $\theta(n^2/numprocs)$.

6.8.19 Poly

Here, we applied the same algorithm as MATLAB does. First, we find the eigenvalues of the matrix and store them into vector e. Next, we use the formula to find the polynomial coefficients (stored in the vector c), given its n values:

$c = [1, 0, ..., 0]$;

for $j = 1$ to n

$$c_{2...j+1} = c_{2...j+1} - e_j \times c_{1...j}$$

Because this only takes $\theta(n^2)$ time compared to $\theta(n^3)$ time for eigenvalues calculation, it is done sequentially. Finally, we distribute the vector c.

6.8.20 Polyval

The Horner method is used to evaluate the polynomial at locally stored points of the matrix (containing the points to evaluate). If several points are given as a matrix, this method achieves a linear speedup because no communication is required. However, small overhead occurs when we all-gather the polynomial coefficients.

6.8.21 Min and Max

Both min and max functions are implemented by the same code. The "<" or ">" operation is defined and the code is included and compiled. Each processor operates on the sub-block of the matrix that it owns. It finds the min/max value and its index for every local column. As a result, each processor ends up with a row vector containing the min/max values and corresponding global indices for each value (it immediately converts the local indices to the global indices). The size of this local vector is equal to the number of columns stored locally. Next, by using the vertical communication groups, these vectors are gathered on the processors, which lie on the first row of the virtual processor grid. A processor there receives all data values and indices from all the processors lying on the same column of the virtual processor grid. After the reduction is accomplished, the processors on the first row of the virtual processor grid contain the result: min/max values and their global indices. Finally, this vector, still distributed over the original 2-D virtual processor grid, is redistributed to its 1-D virtual processor grid.

6.8.22 Matrix Power

Only integer matrix power was implemented. We use the repeated squaring algorithm which needs only $\log(power)$ matrix multiplications. Its performance depends on the performance of the matrix multiplication.

6.8.23 QR Factorization

QR factorization is performed in two steps. First, by using a ScaLAPACK function `pdgeqrf/pzgeqrf`, we compute matrix R and elementary reflectors. Second, by applying a ScaLAPACK subroutine `pdorgqr/pzungqr`, we compute matrix Q. Third, we need to extract the matrix R (it occupies only a portion of an $n \times m$ result matrix) by using a 2-D redistribute function.

6.8.24 Vector Init

If a vector is initialized this way,

```
vec = [start:step:stop],
```

the call to the run-time library's function `ML__vector_init` is made. For the first element in the local data block, the current global index and the value for this element are calculated. When we move to the next element, we increment the current global index. When we finish the current block and move to the next block, we increment the current global index by $block\_size \times (number\_of\_processors - 1)$. On each step, the value is recalculated from the current global index, the starting value, and the step value. This initialization is done efficiently in parallel.

6.8.25 Real/Imag

These are trivial functions performed on local parts of matrices which extract real or imaginary parts of complex elements, respectively.

6.8.26 Conj

This function is trivial. For the local data, it negates the imaginary part. For real and integer types, data are copied locally. No communication is required here.

6.8.27 Flipud

In order to implement this function (which flips the rows of a matrix), we need to redistribute the data so the local swaps are to be performed, therefore, the virtual processor grid must be a single row here. We scan and swap local elements, which lie in the opposite rows, so that any column lies completely locally. After the flips are performed, the matrix is redistributed to the original context it occupied prior to flipping.

6.8.28 Fliplr

This function flips matrix columns. Its implementation is the exact mirror of the flipud function. Here we use the virtual processor grid in the shape of a single column, and swap columns instead of rows.

6.8.29 Hess

In order to calculate the upper Heisenberg matrix, a ScaLAPACK subroutine `pdgehrd/pzgehrd` is invoked directly.

6.8.30 Indexing

In order to provide array indexing, several functions are used. `G2L`, `L2G`, `ML_node`, `ML_addr2`, `ML_addr1`, and `ML_owner` are based on ScaLAPACK functions to convert global to local and local to global indices, and to find the node number from the indices. We need only to calculate the address from the local indices and the processor ID from the coordinates of the processor in the virtual processor grid. This is trivial because processors are laid out in the row-major order in the virtual processor grid by processors' ID's. In addition, `ML_broadcast` is used to broadcast a matrix element to every processor. This is trivial, given previously described functions.

6.8.31 Fft and ifft

In order to calculate FFT, a call to the FFT library is made. Because this library prefers even data distribution, we redistribute the vector, so that every node has the same number of elements and data lie in contiguous chunks. If p (the number of processors) does not divide n (the size), n is increased by $(p - (n \bmod p))$ and zeroes are added to its end. After FFT is computed, the result is truncated (if necessary) and redistributed to its default distribution.

Ifft works in a similar way.

6.8.32 Interp1

`Interp1(xd,yd,x)` only supports linear interpolation. First, we all-gather the vectors xd and yd. Then, we interpolate the local elements of x. If $length(x) \gg length(xd)$, the speedup will be linear.

7 Benchmark Scripts

Several benchmark scripts have been taken from the MathWorks home page ("http://www.mathworks.com") and modified if necessary. These are wing, mortgage, centroid, linear interpolation, rector, and bar impact.

7.1 Conjugate Gradient

Conjugate gradient is an iterative method which solves the system of linear equations $A \times x = b$ ($n = 2048$). The matrix A is a symmetric positive definite matrix. Each iteration takes $\theta(n^2)$ operations and consists of several dot products, a matrix-vector multiplication, and several vector additions and subtractions. Conjugate gradient is a very good example of a scalable script. It gives a very good speedup, and outperforms the interpreter even on one node. The execution time is dominated by the matrix-vector multiplications. The number of iterations is quite small, because it rapidly converges to a solution.

7.2 Transitive Closure

Transitive closure finds the transitive closure of a graph represented by an $n \times n$ adjacency matrix ($n = 512$). A script applies $\log n$ matrix multiplications in a sequence. After $\log n$ multiplications, it replaces any number > 0 with 1 in every matrix's element. Each iteration takes $\theta(n^3)$ operations. The speedup is good because the matrix multiplication is very scalable. In addition, the speedup even on one node is very substantial due to better implementation of the sparse matrix multiplication.

7.3 Ocean

Ocean script models the forces on a sphere submerged in the water by using the Morison equation (the grid size is $n = 8192$ by $m = 95$). It includes dot products, outer products, trapz calculation (discrete integrals), and elementwise operations on matrices. It loads data from a file and performs the simulation. Its time complexity is $\theta(n \times m)$. Here, trapz accounts for a large part of the execution time and is a very good candidate for parallelization.

7.4 An n-body Problem

An n-body problem simulates the movement of particles under gravitational pull (the problem size is $n = 5000$). Each iteration calculates all $\theta(n^2)$ interactions between particles. So, for each pair of particles, a force is calculated and added to the particle's total vector of force. Each iteration takes $\theta(n^2)$ operations. However, here, we need $\theta(n)$ communications. Each set of $\theta(n)$ operations (which calculates the interactions between one particle and all other particles) requires one communication latency overhead (to compute the total force). As a result, we have $\theta(n)$ communication latencies and only $\theta(n^2)$ computations per iteration. This makes it difficult to achieve good results. However, for this script on platforms with small communication latencies, a speedup has been achieved.

7.5 Bar Impact

This benchmark models the oscillations of a metallic bar with one end fixed after an impact has occurred. For given granularity of time tn and space nx, the matrix which represents the position of a piece of bar at time t is computed ($nx = 260$, $nt = 1210$). It takes $\theta(tn \times xn)$ operations to compute the result.

7.6 Linear Interpolation

This simple script computes a piecewise linear interpolation of a data vector defined by the two vectors xd and yd ($nd = 100$, $n = 100000$; where nd is the length of xd and n is the length of x). The program involves $nd - 1$ iterations to find out where each element of the vector x lies (between what elements of vector xd) and interpolates the corresponding element of vector y between these points. This script (if we assume $nd \ll n$), has very good speedup because all operations are elementwise and vectorized. It uses $\theta(n \times nd)$ operations.

7.7 Centroid

This is a toy program which calculates a centroid of a matrix (1024×1024). It computes two means and takes $\theta(n^2)$ time. It is perfectly parallelizable.

7.8 Wing

This is a real application which models the forces on the moving wing. It sets up an $n \times n$ matrix (n is the granularity), and then solves the system of linear equations ($n = 1024$). Setting up the matrix and vector take about $\theta(n^2)$ time, and linear system solving takes $\theta(n^3)$ time. The linear system solving ("\") accounts for the dominant portion of the execution time if n is large enough. The speedup depends on the ScaLAPACK implementation of the linear system solving.

7.9 Sphere

This is a very simple model which calculates over a number of iterations the total force on a sphere flying against the particles and updates the sphere's velocity. Each iteration utilizes $\theta(n)$ of elementwise operations, where n is the granularity (the number of strips into which this sphere is divided, $n = 1000000$). Also, at each iteration, a sum is computed to find the total force. This script is perfectly parallelizable because all calculations are independent, and only one sum is computed (which costs also one communication latency) per iteration.

7.10 Mortgage

This program computes the mortgage over a number of years. It involves setting up the matrix and the calculation of its inverse in $\theta(n^3)$ operations ($n = 12 \times years + 1$, $years = 100$). Also, it uses $\theta(n^2)$ indexing operations (each costs one communication latency) which hurt the performance.

7.11 Rector Modeling

This program calculates the forces and displacement of a membrane under a static pressure. It sets up a matrix in $\theta(n^2)$ operations and solves the least squares problem by using matrix back division in $\theta(n^3)$ time (the matrix size is about 1000×1200). It involves a lot of array slicing operations, which cause the performance to degrade rapidly when the number of nodes increases past one. However, if large enough size is used, the execution time becomes dominated by the calculation of the "\", which scales well.

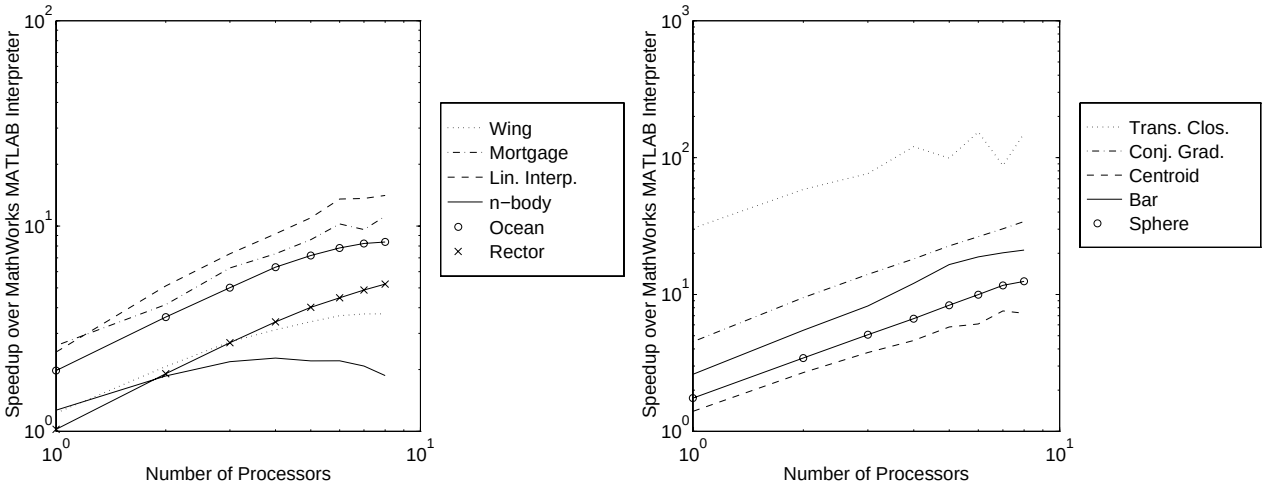

Figure 6: The Performance of Scripts on Sun Enterprise Server 4000

8 The Environment

For benchmarking, I have used a variety of hardware platforms:

- A distributed memory multicomputer – the Meiko CS-2 with 16 nodes. Each node includes one 60 MHz SuperSparc scalar unit, two Fujitsu vector units, and 128 MBytes of memory.
- A symmetrical multiprocessor – the Sun Enterprise Server 4000 with 8 UltraSparc CPU's and 1 Gbyte of memory.
- A cluster of four SparcServer 20 workstations, each a 4 CPU SMP with 128 Mbytes of memory, connected by Ethernet.
- A cluster of ten SparcStation 4 workstations, each with 64 Mbytes of memory, connected by Ethernet.
- A cluster of thirty-two 300 MHz Pentium II PC's, each with 128 Mbytes of memory, connected with fast Ethernet to a Cisco 5505 switch.

9 Results and Explanations

9.1 Methodology

Ten benchmarks have been run on all five parallel platforms, and one benchmark has been run on Sun Enterprise Server 4000. For every data point, seven runs have been made (three runs for the

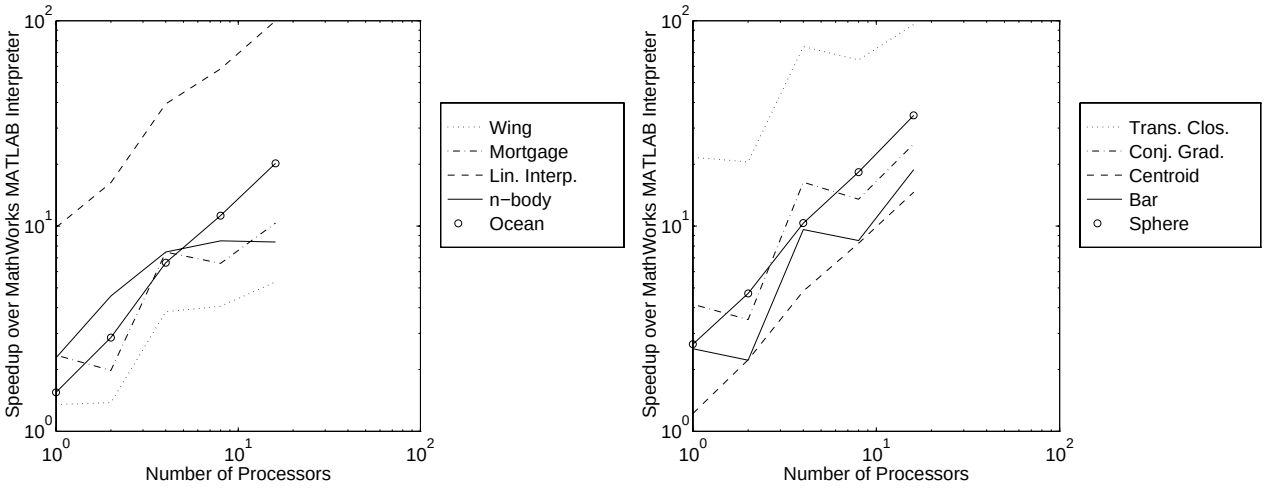

Figure 7: The Performance of Scripts on the Meiko CS-2

rector benchmark), but only the six last runs were used (in order to swap out competing processes and "warm up" a machine). Means of the last six runs were used to compute speedups which are plotted in Figures 6, 7, 8, 9, and 10. Also, tables of standard deviations are included (see Appendices).

9.2 Comparing the Performance on Different Platforms

9.2.1 The Sun Enterprise Server 4000 and the Meiko CS-2

The best platforms for the speedup over the MATLAB interpreter are the Meiko CS-2 and the Sun Enterprise Server 4000 (see Figures 6 and 7). The Meiko CS-2 has very good communication bandwidth of 40 Mbytes/sec, the short message latency of 100 μsec, and relatively slow processor speed. As a result, because the ratio of the communication speed to computation speed is very high here, the results are very good. The Sun Enterprise Server 4000 also has large communication bandwidth between any two nodes of 40 Mbytes/sec and the short message latency of 200 μsec, which have made this platform behave quite nicely for the benchmarks. SMP's suffer from bus saturation, limited memory bandwidth, false cache sharing, and memory module conflicts. The n-body problem is very sensitive to the communication latency and bus saturation; thus, it behaved much worse on the Sun Enterprise Server 4000 compared to the Meiko CS-2. Because the Sun Enterprise Server 4000 has an enormous bus bandwidth (almost 1 Gbyte/sec), most benchmarks scaled well.

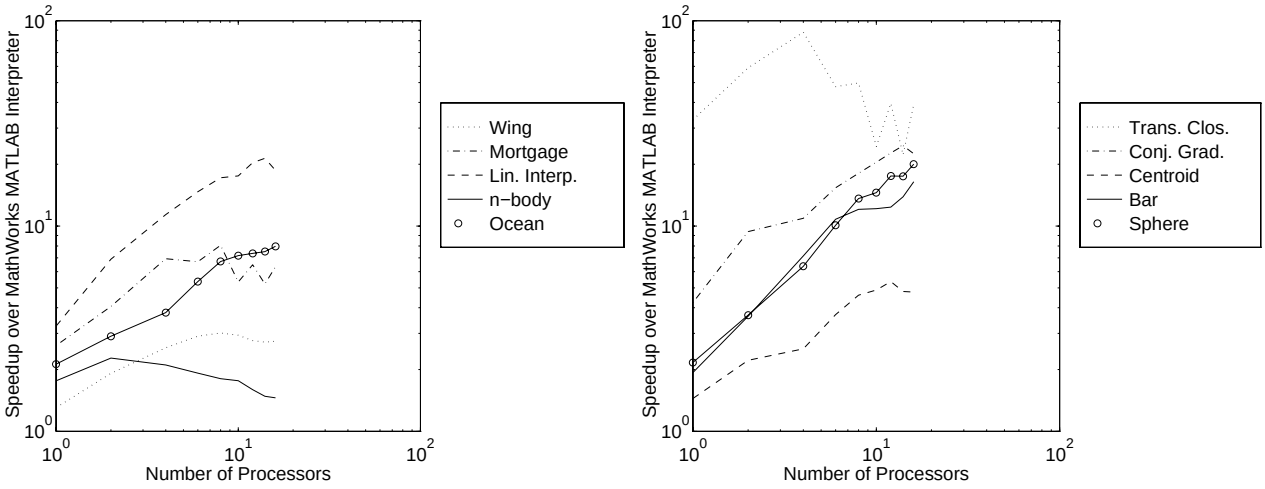

Figure 8: The Performance of Scripts on a Cluster of SparcServer 20 Workstations

9.2.2 A Cluster of Four SparcServer 20 Workstations

As we can see from Figure 8, a cluster of four SparcStation 20 workstations is very sensitive to communication overhead due to very high latency and small bandwidth (1 Mbyte/sec). It also shares problems connected with SMP's described earlier. Thus, the performance here is not as good as on the platforms described earlier. In addition, any change in communication complexity has dramatic effects on the performance here. An interesting anomaly can be observed. When the number of processors becomes 10 or 14, there is a dramatic drop in performance for some benchmarks, such as transitive closure. Later, the performance recovers from this drop. An explanation can be seen very easily from the graphs. Many ScaLAPACK operations are very sensitive to the shape of the virtual processor grid. They frequently benefit from the square virtual processor grid. When the shape of the virtual processor grid becomes very narrow (which is selected automatically or by a command-line parameter), the communication complexity jumps up; as a result, the performance deteriorates. When the grid becomes more square-like, the performance is regained. Because of slow communication, this platform usually does not let scripts scale well.

9.2.3 A Cluster of Ten SparcStation 4 Workstations

If we look at the cluster of ten SparcStation 4 workstations (Figure 9), the results are similar and, sometimes, a little worse than those of a cluster of SparcServer 20 workstations. The reason lies in very small communication bandwidth (about 10 Mbits/sec), high communication latency, and faster processors so that the communication speed to computation speed ratio becomes less

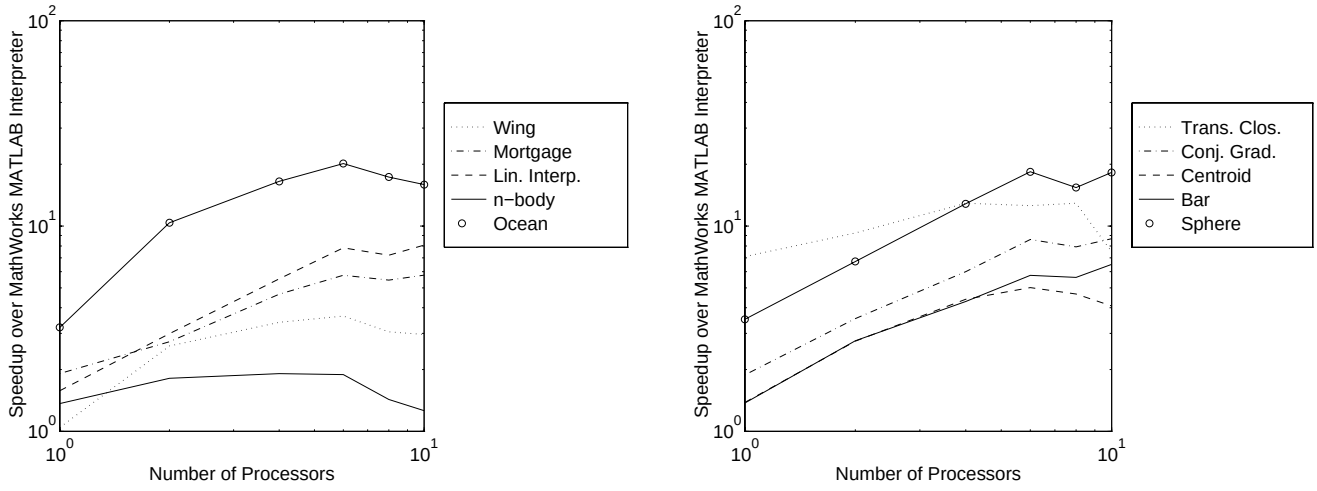

Figure 9: The Performance of Scripts on a Cluster of SparcStation 4 Workstations

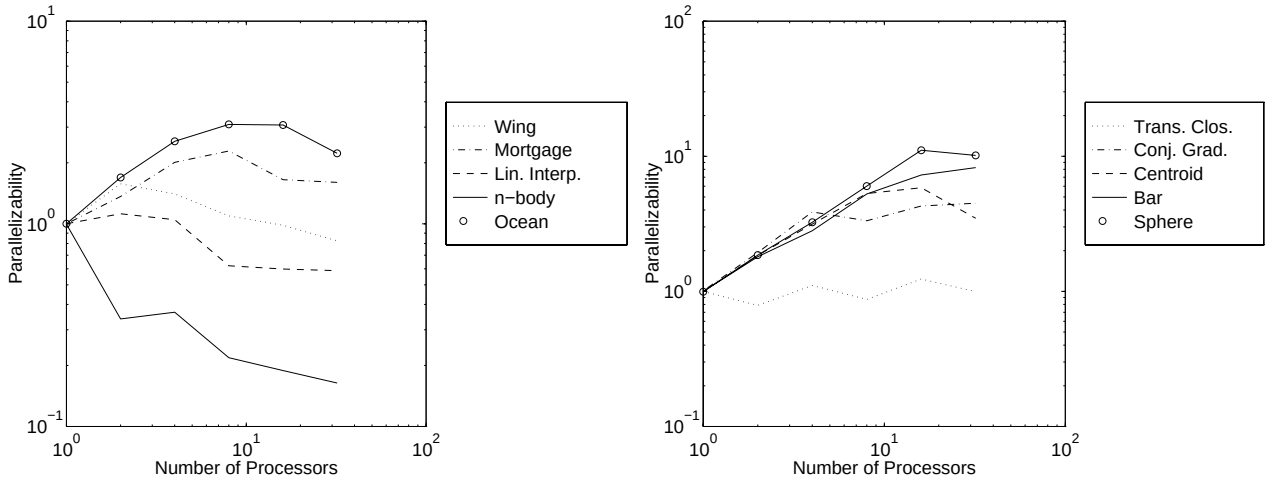

Figure 10: The Performance of Scripts on a Cluster of Pentium II's

favorable for scalability. In addition, the unpredictability of the network, and the fact that the benchmarking has been performed on non-dedicated machines during their normal usage, have contributed to the relatively poor performance.

9.2.4 A Cluster of Pentium II's

The cluster of Pentium II's gave even more disappointing results (Figure 10). It is important to notice that here we measured parallelizability instead of speedup, due to unavailability of the MATLAB interpreter. The low parallelizability resulted from extremely bad communication to processor speed ratio. The extremely high processor speed and low (7-10 MB/sec) communication bandwidth produced their effects, making the communication speed to the computation speed

ratio extremely small.

9.3 Comparing the Performance of Individual Scripts

Let us focus on the performance of individual scripts. We will look at the Sun Enterprise Server 4000 and Meiko CS-2 platforms. Every benchmark achieved one node speedup over the interpreter.

9.3.1 Conjugate Gradient

Conjugate gradient achieved a very profound sequential speedup. It has a fair amount of interpreting overhead for the MATLAB interpreter, so it made a gain almost four times over the interpreter on one node. In addition to this, its heaviest operation, matrix-vector multiplication, is implemented much more efficiently in ScaLAPACK than in the interpreter, giving a significant boost in performance. It is also very scalable, because it uses only matrix and vector operations which scale well.

9.3.2 Transitive Closure

Transitive closure gave very profound speedup on one node. The reason for this is that transitive closure uses the sparse matrix multiplication which is implemented much more efficiently in ScaLAPACK then in the MATLAB interpreter. In addition, it scaled, but not as much as the dense matrix multiplication (see Section 12).

9.3.3 Sphere

This script had a number of elementwise operations and a sum on every iteration which scaled perfectly, and had only a negligible communication overhead. Because the amount of interpreting is small here and all operations are vectorized, speedup on one node was only 2 (on the Sun Enterprise Server 4000).

9.3.4 Ocean

This script also has all its operations vectorized. It scales well due to many matrix and elementwise operations, such as outer products, etc.. Trapz takes a fair amount of the execution time and achieves great speedup. Also its one node performance is good, but it is diminished by elementwise

operations which have no significant one-node speedup because they are already executed very efficiently by the interpreter.

9.3.5 Bar Impact

The bar impact modeling script has elementwise operations and outer products, and, because of described earlier reasons, it scales well and has significant speedup on one node.

9.3.6 Centroid

Centroid computes two means. It scales well because computation of a mean scales very well.

9.3.7 Linear Interpolation

Linear interpolation, because a lot of interpreting is executed here, achieves very good one node speedup, and also scales well because all operations are vectorized.

9.3.8 Wing

The wing script has linear system solving as its major computation, so its performance mostly follows the performance of a "\" operator. One node speedup is very small (because "\" implementation has similar efficiency in the MATLAB interpreter). The speedup is smaller than we would expect because of its small size. Extremely big memory requirements make it difficult to increase the problem size on one node in order to achieve good speedups, but, if the problem size is big enough, the performance will match the performance of the "\" operator.

9.3.9 Mortgage

Mortgage mostly computes the matrix inversion, plus $\theta(n^2)$ of indexing operations. Its performance should match the performance of the matrix inversion with $\theta(n^3)$ time complexity; but the big number of indexing operations hurts the performance. However, it scales well and gives good one-node speedup because its dominant function, inv, scales well and achieves good one-node speedup.

9.3.10 An n-body Problem

Per iteration, an n-body problem uses $\theta(n^2)$ operations, but $\theta(n)$ communication latencies. So, for every latency, only $\theta(n)$ computations are performed. As a result, unless size is enormous (which would take a long time to execute), the latency becomes dominant if the time taken by local computations becomes too small. The Meiko CS-2 gives fair results due to its small communication latency and slow CPU speed. The Sun Enterprise Server 4000 also gives reasonable results due to its small latency. All other platforms have too long communication latencies which overwhelm the computation time. This is clearly seen from the cluster of Pentium II's, whose high speed easily causes the latency to dominate the execution time.

9.3.11 Rector

The rector script achieves reasonable performance on the Sun Enterprise Server 4000 if the problem size is very big. $\theta(n^2)$ of array slicing hurts the performance so much that it was necessary to choose a very large problem size in order for the speedup to be reasonable. With the large problem size that I tested, rector achieves almost linear speedup on the Sun Enterprise Server 4000, but its performance on one node matches the performance of the interpreter.

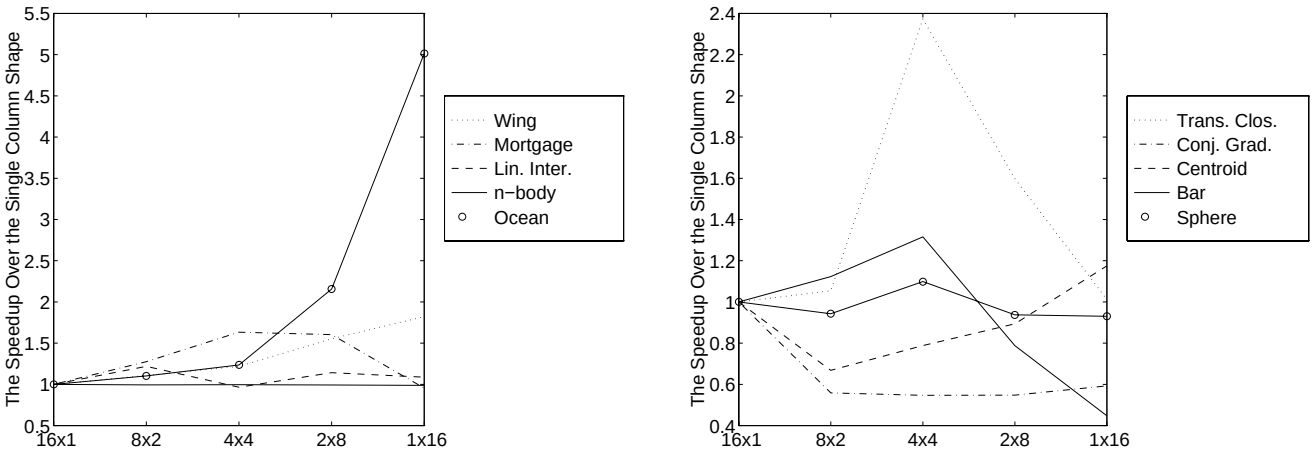

Figure 11: Selecting the Shape of the VPG for the Benchmark Scripts

10 Choosing the Data Distribution Parameters for Scripts

10.1 Selecting the Shape of the Virtual Processor Grid

One machine which is very sensitive to the communication overhead has been utilized (a cluster of four SparcServer 20 workstations) and benchmarked on its 16 nodes, trying all possible shapes of the virtual processor grid (VPG). Here, we are talking about the logical layout of processors; the physical processors are not in the grid, their topology is not considered in project. It can be seen from Figure 11 that the best performance is achieved with the square or row shape of the VPG. Only one benchmark (conjugate gradient) "likes" the VPG in the shape of a single column; its performance changes almost twice when the shape changes from the column to anything else. Barimpact modeling behaves nicely with column and square VPG (a little better for the square shape), but its performance drops abruptly for the row-shaped VPG. Mortgage and transitive closure benefit greatly from the square shape and the performance changes up to 2.4 times (!) when the shape changes from square to a narrow line. An n-body problem, linear interpolation and sphere do not depend much on the shape. Centroid prefers the row shape and dislikes the square shape. The most profound result can be observed on the Ocean benchmark, which prefers the row shape, and the more the shape deviates from the row, the worse the performance becomes, reaching up to 5 times! Wing also prefers the row shape. As observed, the square shape is the best overall, but the row shape is also good for many programs. Only a few programs benefit from

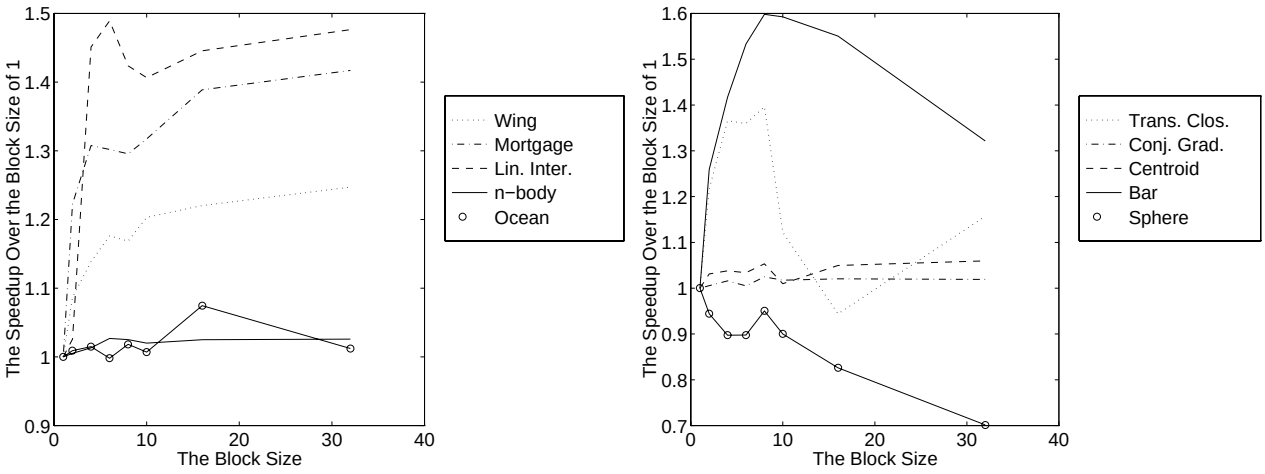

Figure 12: Selecting the Block Size for Benchmark Scripts

the column shape. However, the conclusion is the following: we cannot assume the best shape of the VPG, which is different for every benchmark, but is biased toward the square or row.

Later, you will see how MATLAB operators (or built-in functions) behave and depend on the data distribution parameters. By analyzing a script we can see which functions will dominate, so we can select the data decomposition which is preferred by these dominant functions.

10.2 Selecting the Block Size

The importance of the right block size is not as profound as the shape of the VPG, but the block size makes the performance vary up to 1.6 times. Bar peaks at 8, and goes down for greater size, varying 1.6 times in performance. Ocean's performance peaks at 16; it is not sensitive for sizes less than 10; and it steadily goes down from the block size 16. The n-body, centroid, and conjugate gradient problems are not affected by the block size. Transitive closure's performance goes up sharply and peaks at the block size of 8; but it drops down for the block size of 16, and recovers a little for the block sizes greater then 16. Sphere prefers the block size of 1. The performance of linear interpolation, mortgage, and wing go up sharply until the block size becomes 6, and continue to rise slightly for bigger block sizes. To summarize, the block size depends on application, but usually the best choice would be 8 or 16 for most benchmarks.

11 Choosing the Data Distribution Parameters for the MATLAB Operators and Built-in Functions

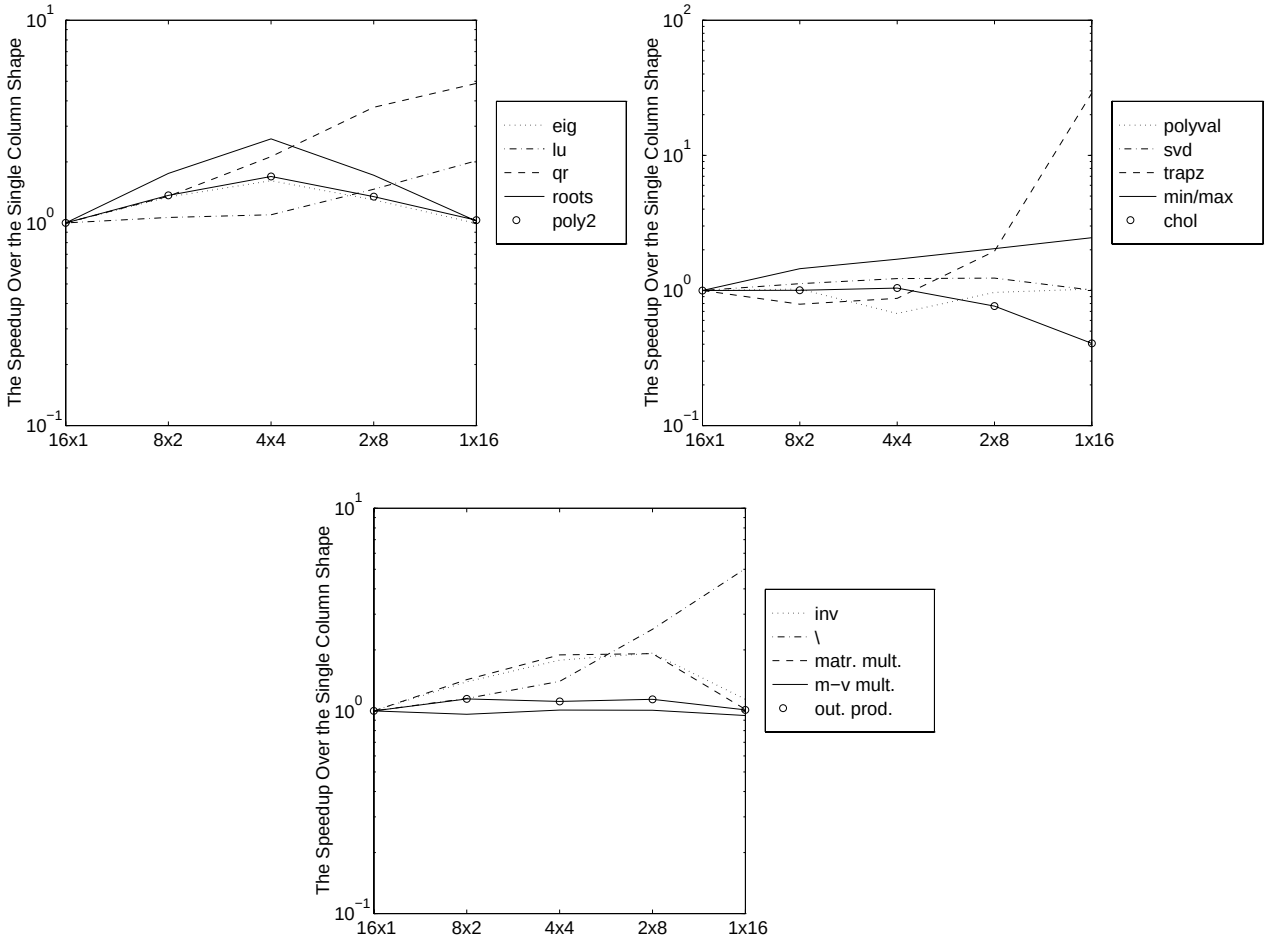

Figure 13: Selecting the Shape of the VPG for MATLAB Operators and Built-in Functions

11.1 Selecting the Shape of the Virtual Processor Grid

The same machine (a cluster of four SparcServer 20 workstations) was used to benchmark on its 16 nodes, trying all possible shapes of the virtual processor grid. It can be seen from Figure 13 that the best performance is achieved with the VPG in the shape of a square or a row. Chol prefers the column or square shape; its performance degrades 2.5 times, if the shape is a row. Qr, trapz, "\", and lu like the row shape. Their performance depends very heavily on the shape to be a row, ranging up to 30 times (trapz). Inv, eig, poly, roots, and matrix multiplication like the square

shape. Svd is almost tolerant to the shape of the VPG. Polyval does not like the square shape. Most other functions not shown on the graphs are tolerant to the shape of the VPG, except few of them which directly use described here functions (det, norm, rank, etc.).

The best shape is usually the square or row.

11.2 Selecting the Block Size

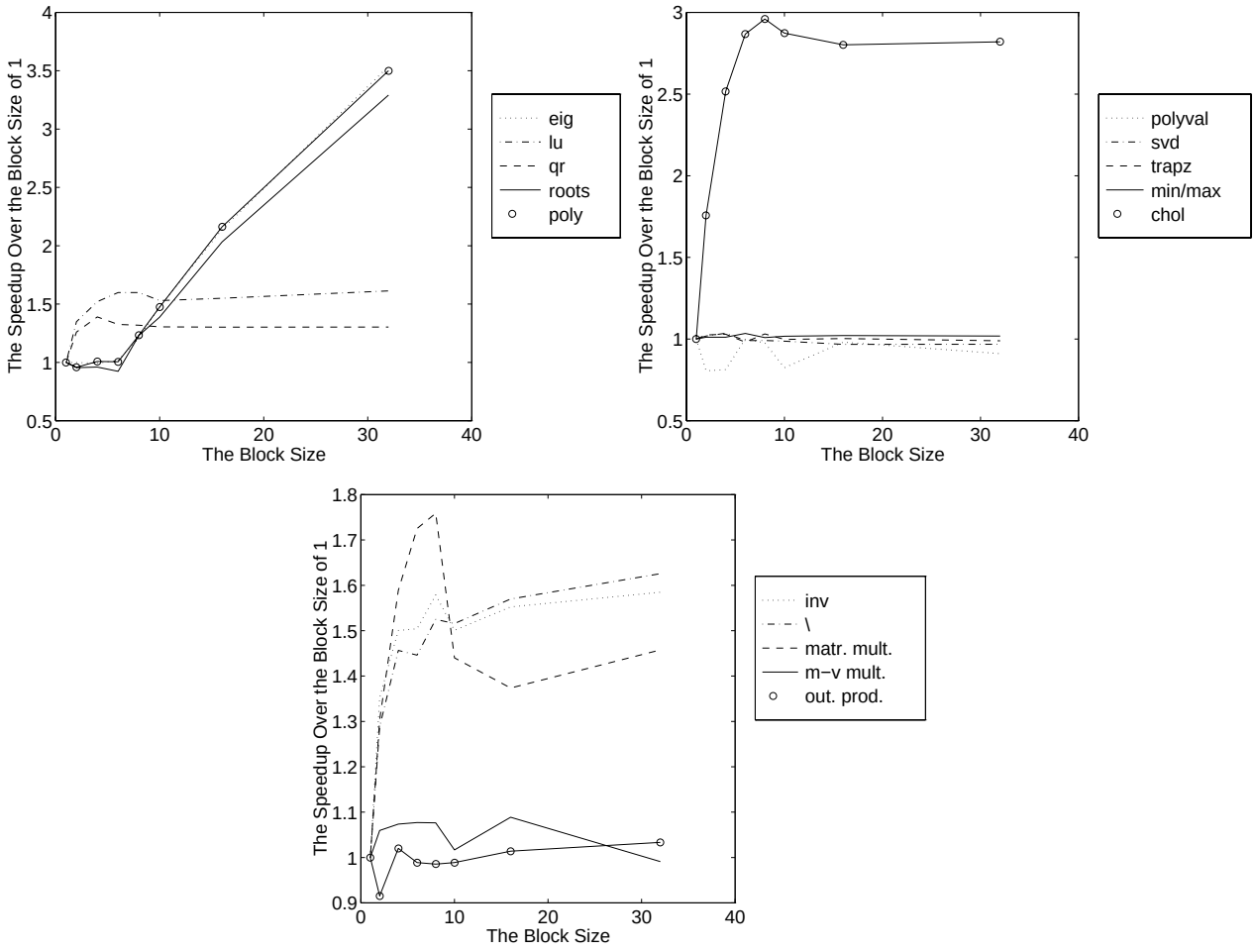

Figure 14: Selecting the Block Size for the MATLAB Operators and Built-in Functions

Similar to tests of the shape of the VPG, the same machine has been selected for the variety of block sizes, ranging from 1 to 32. It can be seen from Figure 14 that the best performance is usually achieved when the block size is large for most functions. Chol gives very sharp performance increase when the block size goes from 1 to 8, staying the same for bigger sizes. Similar but less profound, results can be seen for inv, "\", lu, and qr. Trapz, svd, min/max, and matrix-vector

multiply do not care about the block size. Matrix multiplication is optimal for the block size of 8, dropping slightly for 10, and remaining the same for bigger sizes. Polyval is not very sensitive, but is optimal for the block sizes 1, 6, 8, and those greater than 16.

Eig, roots, and poly prefer the block size as big as possible. These functions require the block size to be at least 6; as a result, in the graph, the results for the block sizes of 1, 2, and 4 are actually the results of runs with the block size of 6.

Outer product is not very sensitive to the block size, but does not like the size of 2 (just about 10 % difference). Basically, the block size is very important and should be chosen carefully. Bigger is usually better, but would be detrimental for narrow matrices, resulting in uneven load.

12 The Performance of Individual MATLAB Operators and Built-in Functions

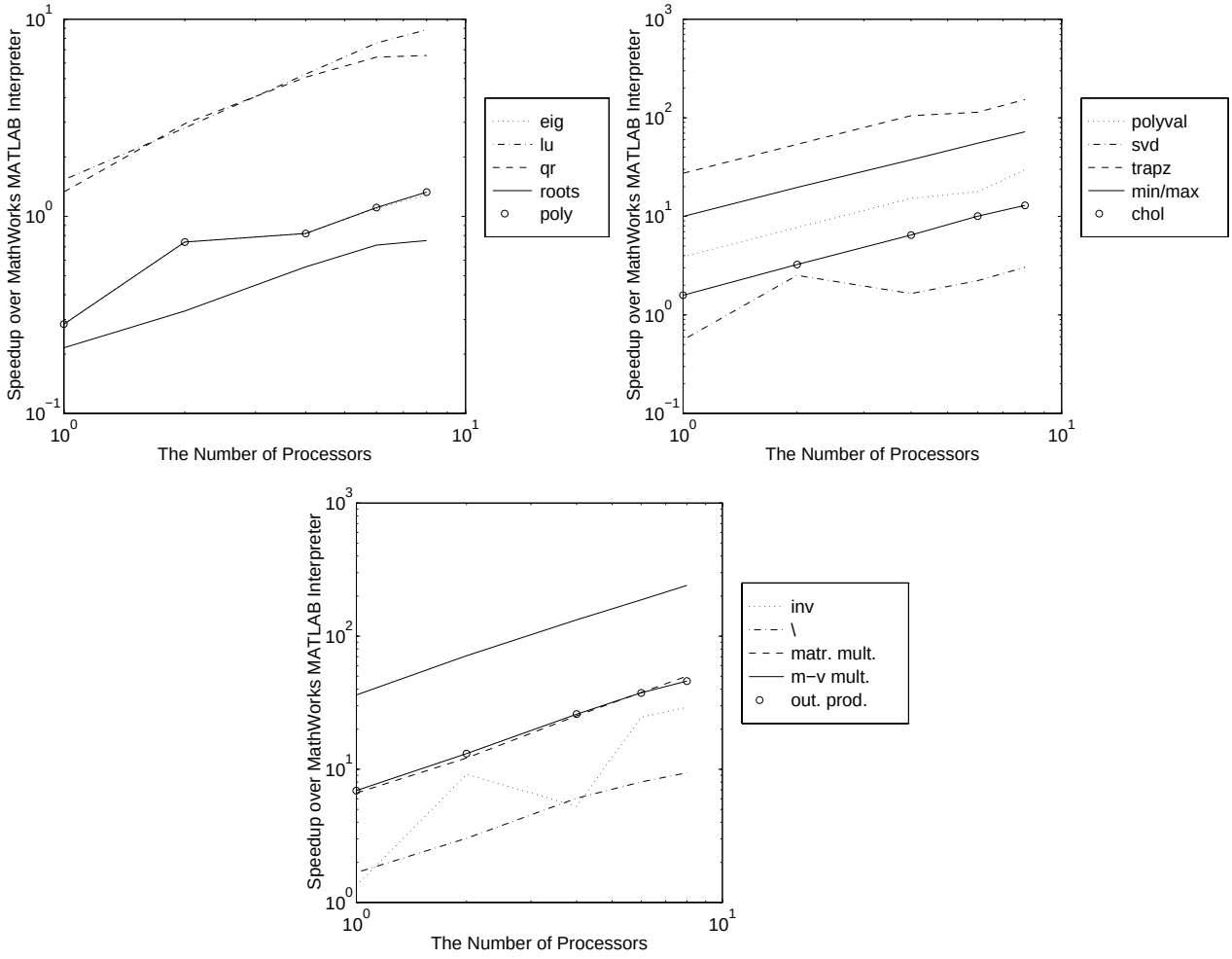

Figure 15: The Performance of MATLAB Operators and Built-in Functions

Let us now look and compare the performance of several MATLAB operators/functions versus their performance under the MATLAB interpreter. The speedup is the speedup of the Otter compiler over the MATLAB interpreter. The performance data have been gathered on the Sun Enterprise Server 4000 (Figure 15). Matrix multiplication ($n = 1024$), matrix-vector multiplication ($n = 8192$), outer product ($n = 8192$), trapz (4096×4096), min/max (6144×6144), and polyval (1000×1000 data points and 100 coefficients) behaved quite well; they achieved good one-node speedup and scaled well. Roots (based on eigenvalue calculation for sparse matrices, $n = 1024$), eigenvalue calculation of the dense matrix ($n = 1024$), and poly (characteristic polynomial calculation, $n =$

1024) scaled well. Inv ($n = 1024$), lu ($n = 1024$), qr ($n = 512$), chol ($n = 2048$), and "\" ($n = 1024$) also scaled well. Svd ($n = 512$) achieved speedup of 5 on 8 processors. The speedup of inv and svd jumped up and down.

In conclusion, MATLAB built-in functions and operators scale well with few exceptions, so results are encouraging. Based on the performance of individual operators and functions, the performance and scalability of whole scripts which use these functions can be predicted.

13 Prediction of a Script's Performance

Sometimes, we can predict a script's performance by just examining its code. First of all, any matrix indexing operations are quite expensive for multiprocessor execution due to communication latencies. If over-used, array slicing can become a bottleneck; and a script which uses array slicing of the same order or bigger than it uses other operations is doomed to be slow. Occasional array slicing is not a problem unless its time dominates. Unless matrix indexing and array slicing are not hidden under scalable operations with higher computational complexity, they can become a serious problem.

Vectorized elementwise operations on matrix's elements, such as matrix addition, will make a script behave quite nicely in terms of scalability, but such operations will not achieve any significant speedup on one node because the interpreter executes them very efficiently.

In order to predict the performance of a script, we need to determine how many operators will be executed, and estimate the problem size of each such operator. Then we can estimate the total time of all "small" operators and the total time of all "big" operators. "Big" operators will include MATLAB operators with big problem sizes. "Small" operators will include all MATLAB operators with small problem sizes. Only "big" operators can scale. If the total time of "small" operators $\ll$ the total time of "big" operators, the script will probably achieve good speedup. Otherwise, we can only benefit from one node performance of the compiled script. By Amdahl's Law, any sequential component will be a limiting factor for the achievable speedup [17]. Hence, if a script contains a noticeable sequential or non-scalable component, the performance will be poor.

Speedup of the compiled script over the interpreted script will be great only if the total fraction of "small" operators and any functions defined as m-files in MATLAB is dominant: in which case the interpreter would have a big overhead due to interpreting of many statements. Functions, defined as m-files, include intepr1, trapz, min, max, etc.. Scripts which spend most of their execution time in the built-in functions defined as m-files with big problem sizes will usually achieve good speedup on one node and will scale well.

Let's now look at a few examples of such analysis. A transitive closure script is a classical example in which the "big" operators dominate (matrix multiplication). It is quite scalable and gives enormous sequential speedup, but only because matrix multiplication is implemented more efficiently in ScaLAPACK than in the interpreter. A similar example is that of conjugate gradient. Its dominant operator is matrix-vector multiplication, which is executed relatively few times.

This makes the script very scalable, and because ScaLAPACK's implementation of matrix-vector multiplication is much faster than the interpreter's, its speedup is also great on one node. Another example is an ocean script. The trapz calculation dominates its execution time. It is a "big" operator and scales well. Because it is defined as an m-file in MATLAB, the speedup on one node is also good.

14 Analyzing Feasibility of Scripts for Parallelization

14.1 An Overview of the Limitations of Otter

Scripts with small execution time, inherently sequential scripts, scripts with a large number of small operations, and scripts with extensive I/O are not good candidates for Otter. Any operation on a single matrix element is detrimental to the performance due to required communications. Extensive array slicing, while good for sequential execution, is detrimental for parallel execution, due to extensive communications. If we keep these ideas in mind, with only small changes in scripts we can achieve dramatic improvements in the performance of compiled scripts.

Our goal will always be to create ways to automate parallel programming. It would be great if the compiler could take any script and produce efficient parallel code. However, as we have seen in many cases, this task is not an easy one. The C* language, which is a parallel extension of C language, allows users to specify the places to parallelize the computations [19]. However, from my own experience, unless there is very careful programming with constant thinking about efficiency of the code, the results will be quite disappointing. The speedup can even become "speed-down." So, it seems very difficult to completely eliminate the user's role here. As a result, our goal changes from full automation of the parallel programming into the goal of minimizing the user's role here as much as possible. Hence, we have to keep in mind the parallel programming style and the rules for how to make scripts more efficiently execute in parallel.

14.2 Time Complexity of Computations

Not every script is worth compiling by Otter. Scripts with small execution time will run much longer if their code is compiled, due to the overhead. Hence, only scripts which run reasonably long are the possible targets for the compiler. Also, the time complexity is very important here. Usually, the greater the time complexity, the better speedup we can get. Matrix operations with time complexity $\theta(n^3)$ are very good to parallelize. $\theta(n^2)$ time complexities are usually more difficult to cope with. Operations with the time complexity of $\theta(n \times \log n)$, like FFT, sometimes are fast enough on sequential machines and are difficult to deal with. Basically, the rule of thumb is to compile the scripts which make a small number of time-consuming function calls. Scripts which make a large number of short-time function calls will not promise good speedup.

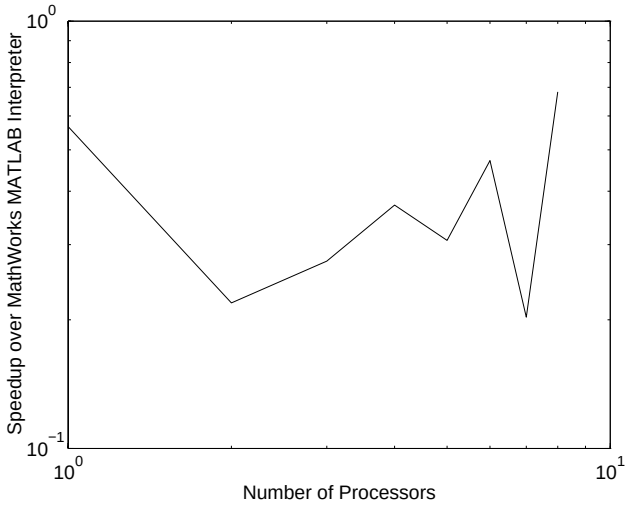

Figure 16: The Performance of the Array Slicing

14.3 Array Slicing

Currently, one of the major limitations of Otter is its inefficient array slicing mechanism. MATLAB provides and encourages the use of extensive array slicing which is quite efficient in sequential implementation. An expression involving any array slicing causes the compiler to generate a temporary matrix, which will hold a slice and make a call to the run-time library to copy the sub-array into this temporary matrix. Array slicing has been benchmarked on the Sun Enterprise Server 4000. The one node performance is satisfactory; but going to two nodes, communication comes into play, so the performance drops sharply. However, with increased numbers of processors, the performance actually goes up due to better utilization of the network's bandwidth (see Figure 16). The local ups and downs are due to the fact that the array slicing benefits from the square processor grid. When the grid becomes too narrow, the performance drops. It recovers when the shape becomes closer to square. However, the performance is still worse than the performance of the MATLAB interpreter due to extensive communication requirements. But, if the size and the number of processors are large enough, better performance can be expected.

Assigning of a matrix to an array slice is done similarly. The compiler will generate a call to the run-time library to copy a matrix into a sub-array of another matrix. The same problems arise here.

14.4 Indexing

Indexing of individual matrix elements involves at least two run-time library calls: `ML__owner` ("do I own this element?"), and `ML__<type>addr` (to get its address). If we read an element to a scalar variable, the compiler also generates an `ML__broadcast` call, which broadcasts this element to a scalar on every node (scalars are replicated). While one node performance is satisfactory, execution on multiple processors will be hindered by high communication latencies, so that any extensive indexing will make scripts run very slowly.

15 Future Work

15.1 The Full Functionality of the Compiler

The commercial MATLAB interpreter offers enormous functionality; therefore, to duplicate this functionality would be a major undertaking. However, by using ScaLAPACK and other available packages, we could implement the whole MATLAB system in reasonable time. I am proposing such an implementation plan. First of all, we use MATLAB, LAPACK, and EISPACK libraries to add the whole set of MATLAB functionality (implemented sequentially) together with already implemented parallel code. In future releases, we will replace sequential functions with parallel ones until the whole functioning parallel MATLAB compiler is created.

15.2 Static Code Analysis and Automatic Selection of the Data Distribution Parameters

Concurrently, we could implement the static code analysis to make decisions about the data distribution parameters as much as possible at compile time to optimize the performance on multiprocessors. In many cases, we can estimate at compile time what fraction of execution time will be spent in every operator or built-in function. Frequently, a user declares several variables, like `N = 100`, and uses these variables in initialization statements and loop bounds, for example, `zeros(N, M)` or `for i = 1 : N`. These variables are, in a sense, constants, because a user never changes them. The compiler can easily propagate the values of such variables, estimate the matrix sizes for each usage of MATLAB operators or built-in functions, and count the number of such operators, which will be executed by counting the number of loop iterations. Then, a compiler can find out what functions (or operators) are dominant in the execution of this script. Based on these data, the compiler could then select the proper data distribution parameters in order to minimize the execution time of dominant operators or functions. For example, an eigenvalue problem prefers large block sizes with a square shape of the VPG, matrix multiplication prefers small block sizes and the square virtual processor grid, trapz prefers the virtual processor grid to be one row, etc..

15.3 Array Slicing Optimizations

As already seen, array slicing executes slowly. In order to optimize array slicing, we could keep the original matrix with information about the array slice it contains. Many operations could consider this special case and perform operations on sub-matrices. Here, we could add a special field in the `MATRIX` data structure which distinguishes an ordinary matrix from a matrix containing a slice. Because ScaLAPACK actually allows us to operate on sub-matrices, we could eliminate this redistribution step in many cases. However, in cases where we cannot operate on the sub-matrices, we could redistribute the data.

For elementwise operations, the compiler could restrict loops only to sub-arrays. However, there is a big problem. In addition to the increased complexity of the compiler and the run-time library, the main assumption would have to be violated: the same matrices are distributed the same way, which allowed us to perform efficiently elementwise operations. It would improve performance dramatically in many cases, but the complexity of the compiler and library would increase tremendously.

15.4 Making an External Interface to the Library

In addition, the run-time library has a very easy and clear interface. We could specify this interface, so that the library could be linked and used from any C programs (similar to what the MATLAB interpreter provides). This would allow for very simple and high level interface to the parallel linear algebra libraries.

15.5 Parallel I/O

Currently, all I/O is handled by one processor, forcing programs to gather the whole matrix onto that node. Improved I/O functions could read data and distribute them at the same time, and could gather data and write/print them at the same time by small pieces. Even further, we could read and parse a file in parallel. First, we could estimate the boundaries in the file for each node. Second, we could read the data in parallel. Third, we could correct the wrong estimations and exchange data if necessary. It is based on the assumption that the file server's bandwidth is much greater than the network bandwidth of individual workstations.

15.6 Replacing ScaLAPACK calls with LAPACK calls for Sequential Execution

If the compiled program runs on a single node, we could call LAPACK subroutines instead of ScaLAPACK subroutines. It could eliminate the overhead of parallel algorithms and, thus, improve the performance on one node.

16 Conclusions

This project has incorporated run-time library support for the parallel MATLAB compiler, which translates MATLAB scripts into SPMD-style C code. In addition, this project has included some limited studies of the parallel performance and the parameters that affect it. Despite the fact that only a subset of MATLAB functionality has been implemented, it has allowed us to compile a few scripts and achieve significant performance increases over the MATLAB interpreter. The 'Otter' system is completely portable, and the run-time library utilizes a number of standard libraries for parallel linear algebra computations and for communications. The performance depends on communication speed to computation speed ratio: the bigger this ratio, the better the parallel performance. The data distribution parameters play a major role here. The choice of these parameters depends on the problem. Because the run-time library allows us to change any parameters just before execution of a compiled script, the parameters can be adjusted in order to optimize the performance. Data distribution parameters have been studied and it has been determined that, usually, the block sizes of 8 or 16 and the square or row shaped virtual processor grid give the best performance. The elementary MATLAB operators/functions usually scale well and give one-node speedup over the MATLAB interpreter. For many scripts, it is possible to predict their performance and scalability by simply examining their code.

References

[1] Blackford, L. S., A. Cleary, J. Choi, J. J. Dongarra, A. Petitet, R. C. Whaley, J. Dammel, I. Dhillo, K. Stanley, D Walker, *LAPACK Working Note 93. Installation Guide for ScaLAPACK*, 1997.

[2] Choi, J.; J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, R. C. Whaley, *LAPACK Working Note 100. A Proposal for Parallel Basic Linear Algebra Subroutines.*, May 1995

[3] Choi, Jayeyong; Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, R. Clint Whaley, *The Design and Implementation of the ScaLAPACK LU, QR, and Choletsky Factorization Routines*, Prentice Hall, September 1994.

[4] 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.

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

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

[7] Dongarra, Jack; R. Clint Whaley, *LAPACK Working Note 94. A User Guide to the BLACS v1.1*, 1997.

[8] Drakenberg, P., Jacobson, P., and K, Kàgström, B., *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.

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

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

[11] Integrated Sensors Home Page. *http://www.sensors.com/isi.*

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

[13] MathTools Home Page. *http://www.mathtools.com*

[14] MathWorks, Inc., *The Student Edition of MATLAB: Version 4 User's Guide*, Prentice-Hall, 1995.

[15] Part-Enander, Eva; Anders Sjoberg, Bo Melin, Pernilla Isaksson, *The MATLAB Handbook*, Addison-Wesley Longman, 1996.

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

[17] Quinn, Michael J.; *Parallel Computing: Theory and Practice.* McGraw-Hill, Inc., 1994

[18] Quinn, M. J., Malishevsky, A., Seelam, N., and Zhao, Y., *Preliminary Results from a Parallel MATLAB Compiler,* in Proceedings of the 12th International Parallel Processing Symposium (to appear).

[19] Thinking Machines Corporation, *Getting Started in C**, Cambridge, Massachusetts, 1993.

[20] 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 Cornell Theory Center, Ithaca, NY (1996).

[21] Wolfe, Michael, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

A Standard Deviations of the Running Time (Percentage of Mean)

A.1 Sun Enterprise Server 4000

Number of processors	1	2	3	4	5	6	7	8
wing	0.1	0.1	0.1	0.3	0.5	0.6	1.4	1.6
mortgage	0.1	0.1	0.2	0.2	0.4	0.8	1.2	1.4
linear	0.3	0.1	0.2	0.2	4.5	0.6	6.4	6.3
n - body	0.1	0.5	0.5	0.6	0.3	1.3	1.2	1.3
ocean	0.6	0.6	0.9	1.0	0.7	2.6	6.1	5.4
closure	0.1	0.1	0.3	0.2	1.4	1.8	3.1	3.9
conjugate	0.5	0.9	1.1	0.2	1.8	1.5	1.2	3.1
centroid	0.5	0.7	0.6	0.3	0.7	5.2	2.2	5.7
barimpact	1.2	0.6	1.2	0.8	0.5	2.0	2.2	4.9
sphere	0.3	0.2	0.6	1.0	0.5	0.3	0.8	10.7
rector	0.0	0.0	0.1	0.3	1.3	0.5	0.8	4.3

A.2 The Meiko CS-2

Number of processors	1	2	4	8	16
wing	0.2	0.2	0.1	0.7	0.4
mortgage	0.0	0.1	0.4	0.4	1.2
linear	0.3	0.5	2.2	4.4	14.3
n - body	0.1	0.1	0.3	0.3	0.2
ocean	0.3	0.4	0.5	1.2	2.3
closure	0.1	0.2	0.5	0.9	2.8
conjugate	0.2	0.1	0.6	0.6	2.6
centroid	0.4	0.4	1.0	1.5	1.9
barimpact	0.1	0.1	0.2	0.3	1.0
sphere	0.1	0.6	1.6	1.1	4.4

A.3 A Cluster of SparcStation 4 Workstations

Number of processors	1	2	4	6	8	10
wing	2.8	0.3	0.7	0.8	1.3	1.9
mortgage	2.4	0.6	0.6	1.1	0.9	0.9
linear	0.6	1.6	3.9	4.0	2.5	1.6
n - body	0.6	0.4	0.1	0.2	2.1	1.6
ocean	15.9	5.3	6.8	5.1	9.6	16.5
closure	2.3	0.5	1.3	1.2	1.0	0.9
conjugate	0.8	0.3	2.8	3.4	4.9	3.2
centroid	1.6	2.6	8.7	15.1	9.0	18.1
barimpact	2.6	0.6	2.4	2.0	2.1	6.3
sphere	0.4	1.4	1.4	0.7	3.9	4.5

A.4 A Cluster of SparcServer 20 Workstations

Number of processors	1	2	4	6	8	10	12	14	16
wing	1.9	8.6	3.4	0.6	0.9	0.6	1.7	1.5	2.8
mortgage	0.3	0.2	1.8	0.4	0.5	0.8	0.6	2.3	4.7
linear	0.1	0.3	1.4	3.0	7.9	28.3	7.9	2.0	42.1
n - body	0.3	1.4	11.8	2.5	0.7	1.3	1.2	1.2	0.7
ocean	0.1	0.4	9.9	2.8	7.3	8.2	3.7	6.7	6.9
closure	3.2	0.3	1.5	0.8	2.2	1.6	1.7	0.7	1.0
conjugate	0.1	0.2	0.2	0.7	0.7	1.1	1.0	1.0	28.7
centroid	0.3	0.8	1.0	2.5	4.9	7.6	8.9	18.4	14.4
barimpact	1.4	0.5	2.1	2.1	4.8	7.7	3.1	6.0	7.0
sphere	0.1	16.3	0.7	2.7	8.8	20.0	14.4	22.0	18.9

A.5 A Cluster of Pentium II's

Number of processors	1	2	4	8	16	32
wing	0.2	0.6	1.4	0.8	1.9	1.8
mortgage	0.2	0.3	0.7	2.2	1.6	1.3
linear	1.4	1.7	1.0	0.7	2.2	1.7
n - body	2.1	1.0	1.0	0.7	0.4	0.3
ocean	1.1	10.1	2.2	5.0	6.1	28.3
closure	0.3	0.2	9.4	6.5	10.8	83.6
conjugate	10.0	6.8	4.3	57.6	25.9	6.1
centroid	0.6	2.6	6.9	3.2	11.9	53.8
barimpact	0.4	0.4	1.7	2.7	6.8	12.9
sphere	1.4	1.1	4.2	14.0	17.9	24.6

B Conjugate Gradient Script

```
% Conjugate Gradient Method for MATLAB Compiler
%
% Author: Yan Zhao (zhao@cs.orst.edu)
%
% Modified by Alexey Malishevsky

%function conjugate()

% initialization
N = 2048;
A = rand(N, N);
% A is positive definate
for i=1:N
  A(i,i) = A(i,i) + 5000.00 + i;
end

X = [1 : N]';

% make b
Epsilon = 0.000001;
b = A * X;

d = zeros(N,1);

% just for testing X
X = zeros(N, 1);

g = -b;

tic

% Start the real work here
for k=1:N

  denom1 = dot(g, g);
  g = A * X - b;

  num1 = dot(g, g);

  if num1 < Epsilon
    break;
```

```
  end

  d = -g + (num1/denom1)*d;

  num2 = dot(d, g);

  denom2 = dot(d, A * d);

  s = -num2/denom2;

  X = X + s * d;

end

toc

% print out iterations
iterations = k

% print out solutions
solution = X;

quit
```

C Transitive Closure Script

```
% Transitive Closure for MATLAB Compiler
%
% Author: Yan Zhao (zhao@cs.orst.edu)
% Date: July 22, 1997

% initialization
N = 512;

A = zeros(N, N);
for ii=1:N
  for j=1:N
    if ii*j < N/2
      A(N-ii,ii+j) = 1;
      A(ii,N-ii-j) = 1;
    end
    if (ii == j)
      A(ii,j) = 1;
    end
  end
end

B = A;

% start timer
tic

% start the real work
ii = N/2;
while ii >= 1
  B = B * B;
  ii = ii / 2;
end

%
B = B > 0;

% stop timer
toc

% print results
```

```
B = B;


quit
```

D Ocean Script

```
% --------------------------------------------------------------------
%  Script:  sisonon.m
%
%  Purpose: Evaluate the nonlinear wave excitation wave force on a submerged
%           sphere using the Morison equation.
%
%  MATLAB script vectorized by Michael J. Quinn
%  Modified by Yan Zhao (zhao@cs.orst.edu) and Alexey Malishevsky
% --------------------------------------------------------------------
load wave.dat                    % Wave staff
load surge0.dat                  % Surge response

% start timer
tic
N = 8192;
M = 95;
x1 = zeros(N,1);
x2 = zeros(N,1);
sur1 = zeros(N,1);
sur2 = zeros(N,1);
fcutoff=1.0;                     % Keep frequencies below this value
fnyquist=8.0;                    % dt=0.0625 sec
dt=0.0625;
xwab=wave(601:8792);             % use 2^13 points out of 12288 points
xwab=xwab-mean(xwab);            % Remove mean from test data

%
surge=surge0(601:8792);
surge=surge-mean(surge);
dtxw=0.0625;                     % Just for misoux.m

tmpvec0 = [0 : (length(xwab) - 1)];

tabi=tmpvec0'*dtxw;              % Just for misoux.m
twab=tmpvec0'*dtxw;

s1 = zeros(1,M);                % make it thinner down to 1/60
for i=0:94
  s1(i+1) = i/65.0;
end
```

```
term = zeros(M,N);
Fd1 = zeros(M,N);
Fi1 = zeros(M,N);

ca=0.5;                          % Added mass coefficient
cm=1+ca;                         % Inertia coefficient
cd=0.6;                          % Drag coefficient
rho=1.94;                        % Density of fluid
T=2.0;                           % Wave period

% add definition of pi
pi = 3.141592653;

w=2*pi/T;
k=0.308878;                      % Wave number
h=9;                             % The water depth
db=(69.75-9)/60;                 % Distance from the bottom of the sphere to
                                 % the bottom of weave tank
sb=18/12;                        % Sphere diameter
s=s1+db;
ds=2*sqrt(s1.*(sb-s1));
prefix1 = w * cosh(k*s) ./ sinh(k*h);
prefix2 = (1.0/2) * rho * cd * ds;
prefix3 = w * cosh(k.*s)/sinh(k*h);
temp1 = rho*pi*cm;
prefix4 = temp1 * (s1 .* (sb - s1));

x1(2:N-1) = xwab(3:N);
x2(2:N-1) = xwab(1:N-2);
sur1(2:N-1) = surge(3:N);
sur2(2:N-1) = surge(1:N-2);

etadot=(x1-x2)/(2*dt);           % Free surface derivative
surgedot=(sur1-sur2)/(2*dt);     % Response velocity
temp2 = ones(length(s1),1) * surgedot';

u = ones(M,N);
u= prefix1'*xwab';

% Particle velocity
term = u - temp2;
tmpP1 = prefix2' * ones(1,N);

Fd1 = tmpP1 .* term .* abs(term);
tmps1 = s1';
```

```
Fd=trapz(tmps1,Fd1);              % Total drag force

udot = prefix3' * etadot';        % Water particle acceleration
Fi1 = (prefix4' * ones(1,N)) .* udot;
Fi=trapz(tmps1,Fi1);              % Total inertia force
Force = Fd+Fi;
toc
tmp = Force(2:N-1);
 check1 = mean(tmp)
 tmp1 = tmp .* tmp;
 check2 = mean(tmp1)
 Y = Force;
 yabi = surge;

quit
```

E An n-body Problem Script

```
% simulation of gravitation movement of a set of objects
% Written by Alexey Malishevsky

N = 5000;
Vx = rand(5000, 1)/1000.0;
Vy = rand(5000, 1)/1000.0;
Vz = rand(5000, 1)/1000.0;
t =  0.0;
dt = 0.001;

% force
Fx = zeros(5000, 1);
Fy = zeros(5000, 1);
Fz = zeros(5000, 1);

% radius-vector
Rx = rand(5000, 1)*10000.0;
Ry = rand(5000, 1)*10000.0;
Rz = rand(5000, 1)*10000.0;

% mass
m = ones(5000, 1) * 1e10;

steps = 1;

% gravitational constant
G = 1e-11;

tic
for j = 1 : steps
 for k = 1 : N
% find the coordinates of radius-vector between particles
   drx = Rx - Rx(k);
   dry = Ry - Ry(k);
   drz = Rz - Rz(k);
% find the distance^2 between k-th particle and all other particles
   r = (drx .* drx + dry .* dry + drz .* drz);
   r(k) = 1.0;

% find the m*M masses, and zero (k, k) pair
   M = m * m(k);
   M(k) = 0.0;

% find the absolute values of gravitational force
   f = G * (M ./ r);
```

```
% find the unit direction vector
   r = sqrt(r);
   drx = drx ./ r;
   dry = dry ./ r;
   drz = drz ./ r;

% find g. forces
   fx = f .* drx;
   fy = f .* dry;
   fz = f .* drz;

%find the resulting force
   fx(k) = 0.0;
   fy(k) = 0.0;
   fz(k) = 0.0;
   Fx(k) = mean(fx) * N;
   Fy(k) = mean(fy) * N;
   Fz(k) = mean(fx) * N;
 end

% find the acceleration
 ax = Fx ./ m;
 ay = Fy ./ m;
 az = Fz ./ m;

% find the velocity
 Vx = Vx + ax * dt;
 Vy = Vy + ay * dt;
 Vz = Vz + az * dt;

% find the radius-vector
 Rx = Rx + Vx * dt;
 Ry = Ry + Vy * dt;
 Rz = Rz + Vz * dt;
 t = t + dt;
end
toc
quit
```

F Barimpact Scripts

F.1 Barimpacrun Script

```
% Program Barimpac
%
% This program computes the longitudinal displacement history
% for an elastic bar fixed at one end and free at the other end.
% The bar is initially at rest when a constant impact load is
% applied to the right end. The resulting motion consists of
% a series of waves traveling at constant velocity along the
% bar. These waves continually reflect back and forth from both
% ends of the bar resulting in motion which is periodic in time.
% For a constant value of x the displacement is a piecewise
% linear function of time. Similarly, for a particular time, the
% displacement is a piecewise linear function of x. The program
% describes the solution by plotting the displacement for two
% positions, and the deflection pattern for two times. The total
% displacement surface u(x,t) for a range of t and x is also shown.

%    by Howard Wilson, U. of Alabama, Spring 1990

% Modified by Alexey Malishevsky

fend=1; a=1; e=1; rho=1; len=1; xmin=0; xmax=1; nx=26*10;
tmin=0; tmax=12; nt=121*10; ntrms=50*10;

tic
[t,x,u]=barimpac_(fend,a,e,rho,len,xmin,xmax,nx, tmin,tmax,nt,ntrms);
toc

check1 = dot(t, t)
check2 = dot(x, x)
check3 = dot(sum(u) ,sum(u))

quit
```

F.2 Barimpac Script

```
function [t,x,u]=barimpac_(fend,a,e,rho,len,xmin,xmax,nx, tmin,tmax,nt,ntrms)
% This function computes the longitudinal displacement in
% an elastic bar which is initially at rest with the left
% end fixed. A constant load suddenly applied to the right end
% causes waves to propogate along the bar and continually
% reflect back and forth from the ends. The resulting time
% response is evaluated from a Fourier series solution.
%
% fend = (end force)   a=area   e = (modulus of elasticity)
% rho = (mass per unit volume)  len = (bar length)
% (xmin,xmax,nx) => evaluate at nx points between xmin and xmax
% (tmin,tmax,nt) => evaluate for nt times between tmin and tmax
% nt = (number of terms summed in the series solution)
% t = (output times)  x = (output positions)
% u = (displacement history stored in an nt by nx array)

% by Howard Wilson, U. of Alabama, Spring 1990

% Modified by Alexey Malishevsky

pi=3.1415926536;
temp1 = [0 : nx - 1];
temp2_ = [0 : nt - 1]';
x = xmin+temp1*((xmax-xmin)/(nx-1));
xol=x/len;
t = tmin+temp2_*((tmax-tmin)/(nt-1));
alpha = sqrt(e/rho);
atl = (alpha/len)*t;
temp3 = [1.0 : ntrms];
beta = pi * temp3 - 0.5;
enddfl = fend*len/(a*e);
gamma = (2*enddfl)*((-1) .^ [1:ntrms])./(beta .* beta);
u = ones(nt,1)*(enddfl*xol);
u = u+((ones(nt,1)*gamma).*cos(atl*beta))*sin(beta'*xol);
```

G Linear Interpolation Script

```
% This function performs piecewise linear interpolation through
% data defined by vectors xd,yd. The components of xd are presumed
% to be in nondecreasing order. Any point where xd(i)==xd(i+1)
% generates a jump discontinuity. For points outside the data range,
% the interpolation gives yd(1) for x < xd(1), and gives yd(nd) for
% x > x(nd), where nd=length(xd).

% Written by H.B.W, Spring 1990

% Modified by Alexey Malishevsky



nd = 100;
n = 100000;
xd = [1.0 : nd];

x = rand(1, n) * 1000;
yd = rand(1, nd) * 1000;

n = length(x);

tic

y=zeros(1, length(x));
nd=length(xd);
y=y+yd(1)*(x<xd(1))+yd(nd)*(x>=xd(nd));
for i=1:nd-1
  xlft=xd(i);
  ylft=yd(i);
  xrht=xd(i+1);
  yrht=yd(i+1);
  dx=xrht-xlft;
  if dx~=0, s=(yrht-ylft)/dx;
    y=y+(x>=xlft).*(x<xrht).*(ylft+s*(x-xlft));
  end
end

toc
```

```
check = mean(y)

quit
```

H Sphere Script

```
% The simulation of the movement of a sphere,
% flying in the gravitational field through the stream of particles.
% Written by Alexey Malishevsky

r=10;
ro=0.1;
n=1000000;
niters = 10;
v = 100.0;
m = 100.0;
pi=3.14159265358;
rings = zeros(1, n);
angle_step = pi/2/n;
angles = [0:angle_step:pi/2];
plate_angles = pi/2 - angles;
vvec = [1.0,1,1];
gvec = [0,0,9.81];
dt = 0.1;

tic

eff = (cos(plate_angles) * 2 * r) .* (sin(angle_step) * 2 * pi * r * sin(angles));

for iter = 1 : niters
 v2 = dot(vvec, vvec);
 force = sum(eff) * ro * v2;
 v2 = sqrt(v2);
 vnorm = vvec / v2;
 a = gvec + (force / m * vnorm);
 vvec = vvec + a * dt;
end

toc

check = vvec

quit
```

I Centroid Script

```
%  calculates the centroid or center of gravity of matrix A
%  c contains [x y] or [(column "index") (row "index")] coordinates
%
% Author(s): R. Johnson
% Revision: 1.0   Date: 1995/11/28
%
% Modified by Alexey Malishevsky

n = 512 * 2;
m = 512 * 2;
A = rand(n, m);
c = rand(2, 1);
rows = n;
cols = m;
x = [1:cols];
y = [1:rows];
y = y';

tic

sumA = sum(sum(A));

% find centroid across columns
c(1) = sum(sum(A.*(ones(rows,1)*x))) / sumA;
% find centroid across rows
c(2) = sum(sum(y*ones(1,cols))) / sumA;
toc
c=c
quit
```

J Mortgage Script

```
% Calculate the mortgage payment
% All right reserved. Can not be translated or
% embedded in any commercial product or package
% with the acknowledgement of the author.
%
% By downloading this m-scripts, you are allowed
% to use the routine for your 'reference' of
% a mortgage payment. The calculation is based
% on a monthly adjustment.
% By using the payment.m, you can calculated the
% amount of principal, interest and cumulative
% payment during a certain period of time.
%
% First version
% Author: F.C. Sze (10/28/96)
% This m-script can also be downloaded from http://sze-pc.ucsd.edu/manager/..
% Q? Forward to dsze@ucsd.edu
%
% Modified by Alexey Malishevsky

tic

years=100;
months=years*12;
rate=0.1;
m=zeros(months+1);
for r=1:months
for c=r:months
if r == c
m(r,c)=1+rate/12;
else
m(r,c)=rate/12;
end
end
m(r,months+1)=-1;
end
for c=1:months
m(months+1,c)=1;
end
x=zeros(months+1,1);
```

```
loan=100000;
x(months+1)=loan;
p=inv(m)*x;
I=ones(months+1,1) * p(months+1)-p;
check = sum(I)
toc

quit
```

K Wing Script

```
% file PRANDTL.M : Calculation of aerodinamic characteristics of straight
% finite wings, using Prandtl - Glauert method and a Chebishev distribution
% of the points on the wing

% (C) by Luigi Tenneriello, 25/11/96
% stnnr109@u6000.csif.unina.it;  euroavia@cds.unina.it
% This is a freeware program: use this program as you want,
% but don't insert it in commercial packages and don't re-sell it!

% Some important remarks:
% 1) This program was developed as an exercise in the Aerospace Engineering
% courses of the University of Naples, Italy; I don't guarantee the
% correctness of the results, but my teacher accepted it...; howewer,
% it has to be (sligthly) modified to include the data of the wing you
% want to study in the "input section" of the program. The results
% will be in the memory at the end of the execution.
%    ----------------------
% 2) Note the presence of the instructions " close('all'); clear; " in the
%    ----------------------
% first line of the program...
% 3) Maybe the significance of the variables is obvious for an aeronautical
% engineering student; howewer, I kept the conventions (and the
% formula too!) from "John D. Anderson Jr.,
%  FUNDAMENTALS OF AERODYNAMICS, Mc Graw Hill Co."
% 4) The Italian readers will be pleased to observe that:
%  "Il programma  anche commentato in Italiano!"
% 5) Note that I wrote this program in 1 hour and an half; do you remember
% the "Columbus egg" tale?
% 6) Thanks to the developers of Matlab!

% INPUT SECTION
% ---------------------------------------------------------------------

% Edited and Vectorized by Alexey Malishevsky

tic
% numero di punti
% no. of points on the wing
n = 1024;
pi_ = 3.1415926536;
```

```
pi = 3.1415926536;

theta0=[0:pi/(n-1):pi];

d1=0.0;
d2=pi;

b=5.0; cr=1.0; ct=1.0; S=0.5*b*(cr+ct); lambda=ct/cr;

y0=-b/2*sin(pi_/2.0 - theta0);

% caratteristiche geometriche dell'ala
% nel mio caso  un' ala trapezia
% geometrical characteristics of the wing,
% e.g. trapezoidal wing in my case
temp_1n = [1:n - 2];
temp_1n2 = temp_1n';
temp1 = zeros(1, 3);
temp2 = zeros(1, 3);
temp1(1) = -b/2.0;
temp1(2) = 0.0;
temp1(3) = b/2.0;
temp2(1) = ct;
temp2(2) = cr;
temp2(3) = ct;
c = interp1(temp1, temp2, y0, 'linear')';
% distribuzione delle corde lungo l' apertura
% distribution of chords on the wing
AR = b*b/S,% Allungamento Alare - Aspect Ratio

Vinf=36; rho=0.125; % velocit / densit dell' aria
% speed and density of the air

Clalpha=2*pi; % in rad^-1
Clalpha=Clalpha*ones(n,1);
% gradiente della retta di portanza di ogni profilo alare
% lift gradient of each wing section

alpha0=0;
% alpha_zero_lift  (in degrees)
% di ogni profilo / of each wing section
alpha0=alpha0*ones(n,1)/180*pi;


alphar=8; % alpha della radice (di riferimento, in gradi)
% angle of attack of the wing root, in degrees
% (it's the reference angle of the wing)
```

```
e=1*ones(n,1); % svergolamento di ogni sezione, in gradi
% twist of each wing section, in degrees


% ---------------------------------------------------------------------
% END OF THE INPUT SECTION

alpha=(alphar-e)/180*pi;
% angolo d' attacco di ogni sezione (in gradi)
% angle of attack of each section (in degrees)
% impostazione del sistema di n-2 equazioni A*An=Anoto e risoluzione
% I build the system A*An=Anoto of n-2 equation, and my PC solve it!
Anoto = (alpha(2:n-1) - alpha0(2:n-1));
c__ = c(2:(n - 1))' *  ones(1, (n - 2));
Clalpha__ = Clalpha(2 : n - 1) * ones(1, (n - 2));
theta0__ = theta0(2 : (n - 1))' * ones(1, n - 2);
Js = ones(n - 2, 1) * temp_1n;
%CC = Clalpha__ - c__;

A = 4 * b * sin(Js .* theta0__) ./ Clalpha__ ./ c__ + ...
 Js .* sin(Js .* theta0__) ./ sin(theta0__);

An = A \ Anoto;

Gamma = zeros(1, n);
% calcolo di Gamma (vorticit lungo l' ala
% calculation of the vorticity Gamma on the wing
for ii=2:n-1
 Gamma(ii) = 2*b*Vinf*sum(An .* (sin(temp_1n * theta0(ii))))';
end;
Gamma(n) = 0;
Gamma(1) = 0;

% calcolo altre caratteristiche aerodinamiche dell' ala
% calculation of the aerodinamical characteristics of the wing
Cl = (2.0 * Gamma) ./ (c / Vinf);
l=rho*Vinf*Gamma; % N/m
CL=An(1)*pi*AR;
alphai = zeros(1, n);
for ii=2:n-1
 temp01=sum(temp_1n2 .* An.*sin(temp_1n2*theta0(ii))./sin(theta0(ii)));
 alphai(ii)=temp01;
end;

temp01 = sum(temp_1n2 .* An .* temp_1n2);
alphai(1) = temp01;
alphai(n) = alphai(1);
```

```
d = l .* alphai;
Cdi=d ./ (0.5*rho*Vinf*Vinf*c);
CDi = pi * AR * sum(temp_1n2 .* (An .* An));
L = CL*.5*rho*Vinf*Vinf*S
Di = CDi*.5*rho*Vinf*Vinf*S

toc

quit
```

L Rector Scripts

L.1 Rectorrun Script

```
nn = 2000;

tmp = [1, 1];
a=tmp(1)/2; b=tmp(2)/2;
tmp = [nn, nn];
nsega = ceil(tmp(1)/2);
nsegb = ceil(tmp(2)/2);
ntrms = ceil(0.6 * nn);
tmp = [25, 25];
nxout=tmp(1); nyout=tmp(2);

tic
y = rector_(a,nsega,b,nsegb,ntrms,nxout,nyout);
toc

quit
```

L.2 Rector Script

```
function y = rector_(a,nsega,b,nsegb,ntrms,nxout,nyout)
%function [cout,phiout,stresout,zout] = rector_(a,nsega,b,nsegb,ntrms,nxout,nyout)
% [c,phi,stres,z]=rector(a,nsega,b,nsegb,ntrms,nxout,nyout)
% This function employs point matching to obtain an approximate
% solution for torsional stresses in a Saint Venant beam of rectang-
% ular cross section. The complex stress function is analytic inside
% the rectangle and has its real part equal to abs(z*z)/2 on the
% boundary. The problem is solved approximately using a polynomial
% stress function which fits the boundary condition in the least
% square sense. The beam is 2*a wide along the x axis and 2*b high
% along the y axis. The shear stresses in the beam are given by the
% complex stress formula:
%
%     (tauzx-i*tauzy)/(mu*alpha) = -i*conj(z)+f'(z)  where
%
%         f(z)=sum( c(j)*z^(2*j-2), j=1:ntrms );

%       by  H. B. Wilson, U. of Alabama, Fall 1989

% First form a vector zbdry of boundary points for point matching.

% Modified by Alexey Malishevsky

eps = 1e-16;
db=b/nsegb;
zbdry=cgrid_(a,a,1,0.0,b-db,nsegb);
ztmp=cgrid_(a,0.0,nsega+1,b,b,1);

temp1 = zbdry;
temp2 = ztmp;
nn = length(temp1) + length(temp2);
n1 = length(temp1);
zbdry=zeros(nn, 1);
zbdry(1:n1) = temp1;
zbdry(n1+1:nn) = temp2;

% neq=length(zbdry); zbdry=[conj(zbdry(neq:-1:2));zbdry];

% Then form the least square equations to impose the boundary
% conditions.

neq=length(zbdry);
amat=ones(neq,ntrms);
```

```
ztmp=zbdry.*zbdry;
bvec = 0.5 * abs(ztmp);

for j=2:ntrms
 temp = amat(:,j-1) .* ztmp;
 amat(:,j) = temp;
end

% Solve the least square equations.
amat=real(amat);

c=amat \ bvec;

% Generate a interior grid to evaluate the solution.

z=cgrid_(-a,a,nxout,-b,b,nyout);
zz=z.*z;

% Evaluate the membrane function on a grid network

phi=real(polyval(flipud(c),zz))-.5*abs(zz);

cc=(2*[1:ntrms]-2)'.*c;

% Evaluate the membrane deflection and the resultant stresses.
% Plot the results.

ysid=imag(cgrid_(a,a,1,-b,b,nyout));
stres=-i*conj(z)+i*polyval(flipud(cc),zz)./(z+eps);

check1 = mean(mean(stres * stres))
check2 = mean(cc)
check3 = mean(mean(phi))
check4 = mean(ysid)

cout = c;
phiout = phi;
stresout = stres;
zout = z;

y = 1;
```

M Compiled C Code of Ocean Script

```
/* Oregon State University MATLAB Compiler */
/* Version unknown --- 16:52:38 Feb 14, 1998 */

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <mpi.h>
#include <matlab.h>

double pi_g;
int i_g;

main (argc, argv)
   int argc;
   char *argv[];
{
   double db_s_r;
   double w_s_r;
   double cd_s_r;
   MATRIX *tmpvec0_m_i = NULL;
   double dt_s_r;
   double fnyquist_s_r;
   double k_s_r;
   MATRIX *s_m_r = NULL;
   MATRIX *surgedot_m_r = NULL;
   MATRIX *s1_m_r = NULL;
   MATRIX *u_m_r = NULL;
   int h_s_i;
   double cm_s_r;
   MATRIX *ds_m_r = NULL;
   MATRIX *Y_m_r = NULL;
   MATRIX *x1_m_r = NULL;
   MATRIX *Fd_m_r = NULL;
   MATRIX *x2_m_r = NULL;
   int M_s_i;
   MATRIX *Fd1_m_r = NULL;
   MATRIX *sur1_m_r = NULL;
   MATRIX *s1_m_i = NULL;
   int N_s_i;
   MATRIX *u_m_i = NULL;
   MATRIX *sur2_m_r = NULL;
   double temp1_s_r;
```

```
double dtxw_s_r;
MATRIX *twab_m_r = NULL;
MATRIX *Fi_m_r = NULL;
double rho_s_r;
MATRIX *Fi1_m_r = NULL;
MATRIX *tabi_m_r = NULL;
MATRIX *x1_m_i = NULL;
MATRIX *tmp_m_r = NULL;
MATRIX *x2_m_i = NULL;
MATRIX *tmp1_m_r = NULL;
MATRIX *tmpP1_m_r = NULL;
MATRIX *Fd1_m_i = NULL;
MATRIX *xwab_m_r = NULL;
MATRIX *sur1_m_i = NULL;
MATRIX *sur2_m_i = NULL;
MATRIX *wave_m_r = NULL;
MATRIX *yabi_m_r = NULL;
double check1_s_r;
MATRIX *tmps1_m_r = NULL;
double check2_s_r;
MATRIX *surge_m_r = NULL;
MATRIX *surge0_m_r = NULL;
MATRIX *Fi1_m_i = NULL;
MATRIX *term_m_r = NULL;
MATRIX *temp2_m_r = NULL;
MATRIX *ML__tmp10 = NULL;
MATRIX *ML__tmp1 = NULL;
MATRIX *udot_m_r = NULL;
MATRIX *ML__tmp11 = NULL;
double ML__tmp2;
MATRIX *ML__tmp12 = NULL;
int ML__tmp3;
MATRIX *ML__tmp13 = NULL;
MATRIX *ML__tmp4 = NULL;
MATRIX *Force_m_r = NULL;
MATRIX *prefix1_m_r = NULL;
MATRIX *term_m_i = NULL;
MATRIX *ML__tmp14 = NULL;
MATRIX *ML__tmp5 = NULL;
MATRIX *prefix2_m_r = NULL;
MATRIX *ML__tmp15 = NULL;
int ML__tmp6;
MATRIX *etadot_m_r = NULL;
MATRIX *prefix3_m_r = NULL;
MATRIX *ML__tmp16 = NULL;
MATRIX *ML__tmp7 = NULL;
MATRIX *prefix4_m_r = NULL;
```

```
MATRIX *ML__tmp17 = NULL;
MATRIX *ML__tmp8 = NULL;
double T_s_r;
double ca_s_r;
double fcutoff_s_r;
MATRIX *ML__tmp9 = NULL;
double sb_s_r;

ML__runtime_init (&argc, &argv);
ML__init(&ML__tmp9, 3, 1, 1);
ML__init(&ML__tmp8, 3, 1, 1);
ML__init(&ML__tmp17, 3, 1, 1);
ML__init(&prefix4_m_r, 3, 1, 1);
ML__init(&ML__tmp7, 3, 1, 1);
ML__init(&ML__tmp16, 3, 1, 1);
ML__init(&prefix3_m_r, 3, 1, 1);
ML__init(&etadot_m_r, 3, 1, 1);
ML__init(&ML__tmp15, 2, 1, 1);
ML__init(&prefix2_m_r, 3, 1, 1);
ML__init(&ML__tmp5, 3, 1, 1);
ML__init(&ML__tmp14, 2, 1, 1);
ML__init(&term_m_i, 2, 1, 1);
ML__init(&prefix1_m_r, 3, 1, 1);
ML__init(&Force_m_r, 3, 1, 1);
ML__init(&ML__tmp4, 2, 1, 1);
ML__init(&ML__tmp13, 3, 1, 1);
ML__init(&ML__tmp12, 3, 1, 1);
ML__init(&ML__tmp11, 2, 1, 1);
ML__init(&udot_m_r, 3, 1, 1);
ML__init(&ML__tmp1, 3, 1, 1);
ML__init(&ML__tmp10, 3, 1, 1);
ML__init(&temp2_m_r, 3, 1, 1);
ML__init(&term_m_r, 3, 1, 1);
ML__init(&Fi1_m_i, 2, 1, 1);
ML__init(&surge0_m_r, 3, 1, 1);
ML__init(&surge_m_r, 3, 1, 1);
ML__init(&tmps1_m_r, 3, 1, 1);
ML__init(&yabi_m_r, 3, 1, 1);
ML__init(&wave_m_r, 3, 1, 1);
ML__init(&sur2_m_i, 2, 1, 1);
ML__init(&sur1_m_i, 2, 1, 1);
ML__init(&xwab_m_r, 3, 1, 1);
ML__init(&Fd1_m_i, 2, 1, 1);
ML__init(&tmpP1_m_r, 3, 1, 1);
ML__init(&tmp1_m_r, 3, 1, 1);
ML__init(&x2_m_i, 2, 1, 1);
ML__init(&tmp_m_r, 3, 1, 1);
```

```
ML__init(&x1_m_i, 2, 1, 1);
ML__init(&tabi_m_r, 3, 1, 1);
ML__init(&Fi1_m_r, 3, 1, 1);
ML__init(&Fi_m_r, 3, 1, 1);
ML__init(&twab_m_r, 3, 1, 1);
ML__init(&sur2_m_r, 3, 1, 1);
ML__init(&u_m_i, 2, 1, 1);
ML__init(&s1_m_i, 2, 1, 1);
ML__init(&sur1_m_r, 3, 1, 1);
ML__init(&Fd1_m_r, 3, 1, 1);
ML__init(&x2_m_r, 3, 1, 1);
ML__init(&Fd_m_r, 3, 1, 1);
ML__init(&x1_m_r, 3, 1, 1);
ML__init(&Y_m_r, 3, 1, 1);
ML__init(&ds_m_r, 3, 1, 1);
ML__init(&u_m_r, 3, 1, 1);
ML__init(&s1_m_r, 3, 1, 1);
ML__init(&surgedot_m_r, 3, 1, 1);
ML__init(&s_m_r, 3, 1, 1);
ML__init(&tmpvec0_m_i, 2, 1, 1);
ML__load(&wave_m_r, "wave.dat");
ML__load(&surge0_m_r, "surge0.dat");
ML__tic();
N_s_i = 8192;
M_s_i = 95;
ML__zeros(x1_m_i, 2, N_s_i, 1);
ML__zeros(x2_m_i, 2, N_s_i, 1);
ML__zeros(sur1_m_i, 2, N_s_i, 1);
ML__zeros(sur2_m_i, 2, N_s_i, 1);
fcutoff_s_r = 1.000000000000e+00;
fnyquist_s_r = 8.000000000000e+00;
dt_s_r = 6.250000000000e-02;
ML__redistribute_1(ML__tmp1, wave_m_r, 601 - 1, 1, 8792 - 1);
ML__matrix_copy(ML__tmp1, xwab_m_r);
ML__vector_mean(xwab_m_r, &ML__tmp2);
ML__matrix_init(&xwab_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, xwab_m_r, 0, 4);
for (ML__tmp3 = ML__local_els(xwab_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--) {
   xwab_m_r->realbase[ML__tmp3] = xwab_m_r->realbase[ML__tmp3] - ML__tmp2;
}
ML__redistribute_1(ML__tmp1, surge0_m_r, 601 - 1, 1, 8792 - 1);
ML__matrix_copy(ML__tmp1, surge_m_r);
ML__vector_mean(surge_m_r, &ML__tmp2);
ML__matrix_init(&surge_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, surge_m_r, 0, 4);
for (ML__tmp3 = ML__local_els(surge_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--) {
   surge_m_r->realbase[ML__tmp3] = surge_m_r->realbase[ML__tmp3] - ML__tmp2
      ;
}
```

```
   dtxw_s_r = 6.250000000000e-02;
   ML__length(xwab_m_r, &ML__tmp3);
   ML__vector_init(tmpvec0_m_i, 0, 0, 1, 0, (ML__tmp3 - 1), 0);
   ML__matrix_transpose(tmpvec0_m_i, ML__tmp4);
   ML__matrix_init(&tabi_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp4, 0, 4);
   for (ML__tmp3 = ML__local_els(tabi_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--) {
      tabi_m_r->realbase[ML__tmp3] = ML__tmp4->intbase[ML__tmp3] * dtxw_s_r;
   }
   ML__matrix_transpose(tmpvec0_m_i, ML__tmp4);
   ML__matrix_init(&twab_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp4, 0, 4);
   for (ML__tmp3 = ML__local_els(twab_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--) {
      twab_m_r->realbase[ML__tmp3] = ML__tmp4->intbase[ML__tmp3] * dtxw_s_r;
   }
   ML__zeros(s1_m_i, 2, 1, M_s_i);
   ML__matrix_copy(s1_m_i, s1_m_r);
   for (i_g = 0; i_g <= 94; i_g++) {
      if (ML__owner(s1_m_r, i_g + 1 - 1, 0 - 1)) {
         *ML__realaddr1(s1_m_r, i_g + 1 - 1) = i_g / 6.500000000000e+01;
}
   }
   ML__zeros(term_m_i, 2, M_s_i, N_s_i);
   ML__zeros(Fd1_m_i, 2, M_s_i, N_s_i);
   ML__zeros(Fi1_m_i, 2, M_s_i, N_s_i);
   ca_s_r = 5.000000000000e-01;
   cm_s_r = 1 + ca_s_r;
   cd_s_r = 6.000000000000e-01;
   rho_s_r = 1.940000000000e+00;
   T_s_r = 2.000000000000e+00;
   pi_g = 3.141592653000e+00;
   w_s_r = 2 * pi_g / T_s_r;
   k_s_r = 3.088780000000e-01;
   h_s_i = 9;
   db_s_r = 1.012500000000e+00;
   sb_s_r = 1.500000000000e+00;
   ML__matrix_init(&s_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, s1_m_r, 0, 4);
   for (ML__tmp3 = ML__local_els(s_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--) {
      s_m_r->realbase[ML__tmp3] = s1_m_r->realbase[ML__tmp3] + db_s_r;
   }
   ML__matrix_init(&ML__tmp5, 3, 0, 0, 0, 0, 0, 0, 0, 0, s1_m_r, 0, 4);
   for (ML__tmp3 = ML__local_els(ML__tmp5) - 1; ML__tmp3 >= 0; ML__tmp3--) {
      ML__tmp5->realbase[ML__tmp3] = s1_m_r->realbase[ML__tmp3] * (sb_s_r -
         s1_m_r->realbase[ML__tmp3]);
   }
   ML__matrix_init(&ds_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp5, 0, 4);
   for (ML__tmp6 = ML__local_els(ds_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
      ds_m_r->realbase[ML__tmp6] = 2 * sqrt(ML__tmp5->realbase[ML__tmp6]);
   }
```

```
ML__matrix_init(&ML__tmp5, 3, 0, 0, 0, 0, 0, 0, 0, 0, s_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(ML__tmp5) - 1; ML__tmp6 >= 0; ML__tmp6--) {
   ML__tmp5->realbase[ML__tmp6] = k_s_r * s_m_r->realbase[ML__tmp6];
}
ML__tmp2 = k_s_r * h_s_i;
ML__matrix_init(&prefix1_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp5, 0, 4);
for (ML__tmp3 = ML__local_els(prefix1_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--
   ) {
   prefix1_m_r->realbase[ML__tmp3] = w_s_r * cosh(ML__tmp5->realbase[
      ML__tmp3]) / sinh(ML__tmp2);
}
ML__matrix_init(&prefix2_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ds_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(prefix2_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--
   ) {
   prefix2_m_r->realbase[ML__tmp6] = 5.000000000000e-01 * rho_s_r * cd_s_r
       * ds_m_r->realbase[ML__tmp6];
}
ML__matrix_init(&ML__tmp5, 3, 0, 0, 0, 0, 0, 0, 0, 0, s_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(ML__tmp5) - 1; ML__tmp6 >= 0; ML__tmp6--) {
   ML__tmp5->realbase[ML__tmp6] = k_s_r * s_m_r->realbase[ML__tmp6];
}
ML__tmp2 = k_s_r * h_s_i;
ML__matrix_init(&prefix3_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp5, 0, 4);
for (ML__tmp3 = ML__local_els(prefix3_m_r) - 1; ML__tmp3 >= 0; ML__tmp3--
   ) {
   prefix3_m_r->realbase[ML__tmp3] = w_s_r * cosh(ML__tmp5->realbase[
      ML__tmp3]) / sinh(ML__tmp2);
}
temp1_s_r = rho_s_r * pi_g * cm_s_r;
ML__matrix_init(&prefix4_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, s1_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(prefix4_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--
   ) {
   prefix4_m_r->realbase[ML__tmp6] = temp1_s_r * (s1_m_r->realbase[ML__tmp6
      ] * (sb_s_r - s1_m_r->realbase[ML__tmp6]));
}
ML__matrix_copy(x1_m_i, x1_m_r);
ML__redistribute_1(ML__tmp8, xwab_m_r, 3 - 1, 1, N_s_i - 1);
ML__matrix_copy(ML__tmp8, ML__tmp7);
ML__shift_1(x1_m_r, ML__tmp7, 2 - 1, 1, N_s_i - 1 - 1);
ML__matrix_copy(x2_m_i, x2_m_r);
ML__redistribute_1(ML__tmp10, xwab_m_r, 1 - 1, 1, N_s_i - 2 - 1);
ML__matrix_copy(ML__tmp10, ML__tmp9);
ML__shift_1(x2_m_r, ML__tmp9, 2 - 1, 1, N_s_i - 1 - 1);
ML__matrix_copy(sur1_m_i, sur1_m_r);
ML__redistribute_1(ML__tmp9, surge_m_r, 3 - 1, 1, N_s_i - 1);
ML__matrix_copy(ML__tmp9, ML__tmp10);
ML__shift_1(sur1_m_r, ML__tmp10, 2 - 1, 1, N_s_i - 1 - 1);
```

```
ML__matrix_copy(sur2_m_i, sur2_m_r);
ML__redistribute_1(ML__tmp7, surge_m_r, 1 - 1, 1, N_s_i - 2 - 1);
ML__matrix_copy(ML__tmp7, ML__tmp8);
ML__shift_1(sur2_m_r, ML__tmp8, 2 - 1, 1, N_s_i - 1 - 1);
ML__matrix_init(&etadot_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, x1_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(etadot_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--
   ) {
   etadot_m_r->realbase[ML__tmp6] = (x1_m_r->realbase[ML__tmp6] -
      x2_m_r->realbase[ML__tmp6]) / (2 * dt_s_r);
}
ML__matrix_init(&surgedot_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, sur1_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(surgedot_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--
   ) {
   surgedot_m_r->realbase[ML__tmp6] = (sur1_m_r->realbase[ML__tmp6] -
      sur2_m_r->realbase[ML__tmp6]) / (2 * dt_s_r);
}
ML__length(s1_m_r, &ML__tmp6);
ML__ones(ML__tmp11, 2, ML__tmp6, 1);
ML__matrix_transpose(surgedot_m_r, ML__tmp5);
ML__matrix_multiply(ML__tmp11, ML__tmp5, temp2_m_r);
ML__ones(u_m_i, 2, M_s_i, N_s_i);
ML__matrix_transpose(prefix1_m_r, ML__tmp12);
ML__matrix_transpose(xwab_m_r, ML__tmp13);
ML__matrix_multiply(ML__tmp12, ML__tmp13, u_m_r);
ML__matrix_init(&term_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, u_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(term_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
   term_m_r->realbase[ML__tmp6] = u_m_r->realbase[ML__tmp6] -
      temp2_m_r->realbase[ML__tmp6];
}
ML__matrix_transpose(prefix2_m_r, ML__tmp12);
ML__ones(ML__tmp14, 2, 1, N_s_i);
ML__matrix_multiply(ML__tmp12, ML__tmp14, tmpP1_m_r);
ML__matrix_init(&Fd1_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, tmpP1_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(Fd1_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
   Fd1_m_r->realbase[ML__tmp6] = tmpP1_m_r->realbase[ML__tmp6] *
      term_m_r->realbase[ML__tmp6] * fabs(term_m_r->realbase[ML__tmp6]);
}
ML__matrix_transpose(s1_m_r, tmps1_m_r);
ML__trapz2(tmps1_m_r, Fd1_m_r, Fd_m_r);
ML__matrix_transpose(prefix3_m_r, ML__tmp12);
ML__matrix_transpose(etadot_m_r, ML__tmp5);
ML__matrix_multiply(ML__tmp12, ML__tmp5, udot_m_r);
ML__matrix_transpose(prefix4_m_r, ML__tmp12);
ML__ones(ML__tmp15, 2, 1, N_s_i);
ML__matrix_multiply(ML__tmp12, ML__tmp15, ML__tmp16);
ML__matrix_init(&Fi1_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, ML__tmp16, 0, 4);
for (ML__tmp6 = ML__local_els(Fi1_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
```

```
      Fi1_m_r->realbase[ML__tmp6] = (ML__tmp16->realbase[ML__tmp6]) *
         udot_m_r->realbase[ML__tmp6];
   }
   ML__trapz2(tmps1_m_r, Fi1_m_r, Fi_m_r);
   ML__matrix_init(&Force_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, Fd_m_r, 0, 4);
   for (ML__tmp6 = ML__local_els(Force_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
      Force_m_r->realbase[ML__tmp6] = Fd_m_r->realbase[ML__tmp6] +
         Fi_m_r->realbase[ML__tmp6];
   }
   ML__toc();
   ML__redistribute_1(ML__tmp17, Force_m_r, 2 - 1, 1, N_s_i - 1 - 1);
   ML__matrix_copy(ML__tmp17, tmp_m_r);
   ML__vector_mean(tmp_m_r, &check1_s_r);
   ML__print_real (check1_s_r, "check1_s_r");
   ML__matrix_init(&tmp1_m_r, 3, 0, 0, 0, 0, 0, 0, 0, 0, tmp_m_r, 0, 4);
for (ML__tmp6 = ML__local_els(tmp1_m_r) - 1; ML__tmp6 >= 0; ML__tmp6--) {
      tmp1_m_r->realbase[ML__tmp6] = tmp_m_r->realbase[ML__tmp6] *
         tmp_m_r->realbase[ML__tmp6];
   }
   ML__vector_mean(tmp1_m_r, &check2_s_r);
   ML__print_real (check2_s_r, "check2_s_r");
   ML__matrix_copy(Force_m_r, Y_m_r);
   ML__matrix_copy(surge_m_r, yabi_m_r);
   ML__free(ML__tmp9);
   ML__free(ML__tmp8);
   ML__free(ML__tmp17);
   ML__free(prefix4_m_r);
   ML__free(ML__tmp7);
   ML__free(ML__tmp16);
   ML__free(prefix3_m_r);
   ML__free(etadot_m_r);
   ML__free(ML__tmp15);
   ML__free(prefix2_m_r);
   ML__free(ML__tmp5);
   ML__free(ML__tmp14);
   ML__free(term_m_i);
   ML__free(prefix1_m_r);
   ML__free(Force_m_r);
   ML__free(ML__tmp4);
   ML__free(ML__tmp13);
   ML__free(ML__tmp12);
   ML__free(ML__tmp11);
   ML__free(udot_m_r);
   ML__free(ML__tmp1);
   ML__free(ML__tmp10);
   ML__free(temp2_m_r);
   ML__free(term_m_r);
```

```
    ML__free(Fi1_m_i);
    ML__free(surge0_m_r);
    ML__free(surge_m_r);
    ML__free(tmps1_m_r);
    ML__free(yabi_m_r);
    ML__free(wave_m_r);
    ML__free(sur2_m_i);
    ML__free(sur1_m_i);
    ML__free(xwab_m_r);
    ML__free(Fd1_m_i);
    ML__free(tmpP1_m_r);
    ML__free(tmp1_m_r);
    ML__free(x2_m_i);
    ML__free(tmp_m_r);
    ML__free(x1_m_i);
    ML__free(tabi_m_r);
    ML__free(Fi1_m_r);
    ML__free(Fi_m_r);
    ML__free(twab_m_r);
    ML__free(sur2_m_r);
    ML__free(u_m_i);
    ML__free(s1_m_i);
    ML__free(sur1_m_r);
    ML__free(Fd1_m_r);
    ML__free(x2_m_r);
    ML__free(Fd_m_r);
    ML__free(x1_m_r);
    ML__free(Y_m_r);
    ML__free(ds_m_r);
    ML__free(u_m_r);
    ML__free(s1_m_r);
    ML__free(surgedot_m_r);
    ML__free(s_m_r);
    ML__free(tmpvec0_m_i);
    ML__runtime_finalize();
}
```