MAT*marks* : A Shared Memory Environment for MATLAB Programming

George Almasi, Calin Cascaval, and David A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
{galmasi,cascaval,padua}@cs.uiuc.edu

February 9, 1999

Abstract

MAT*marks* is an extension of the MATLAB tool that enables shared memory programming on a network of workstations by adding a small set of commands. In this paper we first give a high-level overview of the MAT*marks* system. Then we present the commands we had to add on to MATLAB. Next we dwell on implementation details; finally we present the performance gains we achieved by using the system.

1 Introduction

Many researchers use domain specific languages (DSL) to solve problems in their area of expertise. These languages have several characteristics that make them more convenient than general purpose programming languages. DSLs allow the programmer to focus on the problem at hand rather than deal with memory management or other idiosyncrasies of general purpose languages. However, most DSLs are not designed with efficiency and performance in mind and are not efficiently implemented.

In this paper we describe MAT*marks* , a shared memory extension to the MATLAB [10] environment which allows the user to write programs that run on multiple processors and machines. MATLAB is a matrix manipulation language and environment. Popular among scientists, it is very easy to learn and use, and it has an extensive collection of libraries that implement the most common linear algebra operations.

MAT*marks* provides an environment in which the user can run several MATLAB interpreters in parallel, and a set of primitives that allow the user to program using the SPMD (Single Program Multiple Data) programming model [4].

While interpreted execution is inherently slow, MAT*marks* can be used to write, debug and tune a shared memory parallel program. This program can then be optimized with more conventional methods, such as compilation and vectorization, to obtain near-FORTRAN performance.

MAT*marks* is based on the Treadmarks [1] virtual distributed shared memory system. Treadmarks provides a shared memory view for processes running on a network of workstations. It has a minimal set of primitives that allow users to program using a shared memory paradigm, transparent of the fact that different processes run as distinct processes on a collection of machines. MAT*marks* extends this behavior to MATLAB interpreters.

2 The MAT*marks* Environment

MAT*marks* is a natural extension of MATLAB; it integrates the behavior of the underlying virtual shared memory system, Treadmarks. Like in Treadmarks, the basic tool for enabling parallelism is the ability of the programmer to declare regions of memory as shared, i.e., showing the same value to all participating processes. However, unlike in Treadmarks, declaration of shared variables is entirely dynamic. Different processes involved in the same computation can start sharing a variable at different points in the program. A subset of processes can share a variable, and this subset can change in time.

MAT*marks* allows the programmer to use the SPMD (Single Program Multiple Data) programming paradigm [4]. In this model, the programmer assumes the existence of a master process which executes all the sequential portions of the program, and a set of slave processes that are activated by the master whenever it encounters a parallel region. This logical view is enabled in MAT*marks* by the process identification commands.

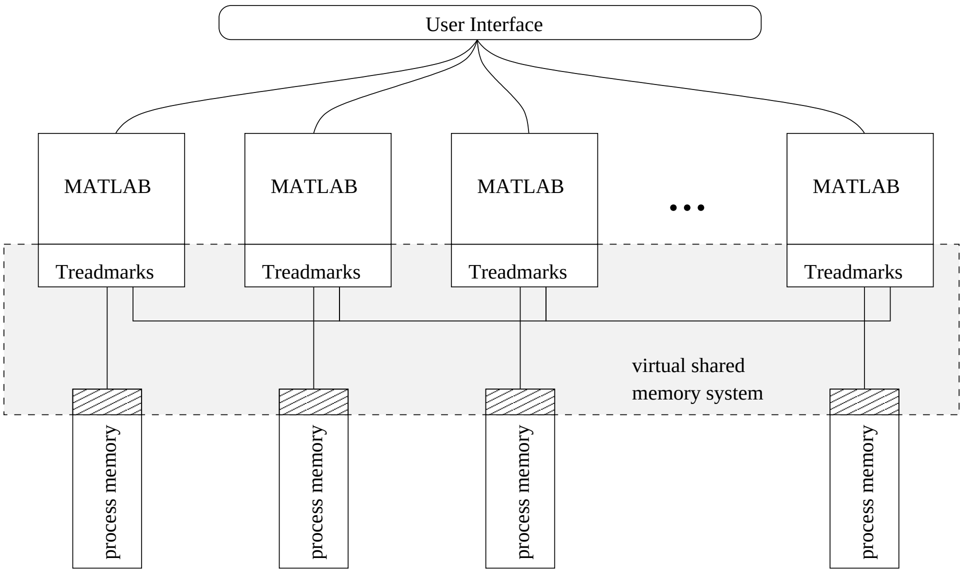

Figure 1: The MAT*marks* System

Figure 1 shows the architecture of the MAT*marks* system. The common user interface ties together a set of MATLAB interpreters, each extended with the Treadmarks library.

The user interface (Figure 2) has two roles. First it starts all the MATLAB interpreters on the desired machines and bootstraps the Treadmarks connections between them. Next, as the system reaches a stable state, it helps distributing the user's input to the member processes and gathers partial results from the machines.

The user interface is available in three distinct flavors. First, a simple command line based interface, used mostly for debugging. There are also two graphical user interfaces (GUIs), one based on the Tk/Tcl toolkit [12] and the other on Java [2].

Both GUIs allow the user to send commands to individual processes or to all processes simultaneously. For example, the GUI in Figure 2 has a text entry window for each process 1, an entry window for sending global commands 2, and a text window 3 to display the results for each process.

The GUI is also responsible for starting up all MATLAB slave processes. The processes are independent until the user presses the `Init` button 4, which connects the processes using the Treadmarks library.

The MATLAB interpreters run on several machines. They are not changed in any way; the MAT*marks* system consists only of dynamic libraries added to the interpreters' search path. One of these libraries contains the Treadmarks system.

Treadmarks uses data replication to implement the illusion of shared memory. Unlike most hardware-based shared memory systems, Treadmarks is not sequential consistent [9]. Instead it implements a *lazy release consistency model* [8] in order to minimize the number of messages exchanged through the network. What this means in practice is that good care must be taken not to allow data races in the programs because unexpected behavior will certainly result. Data races can be avoided using the synchronization primitives provided by MAT*marks* .

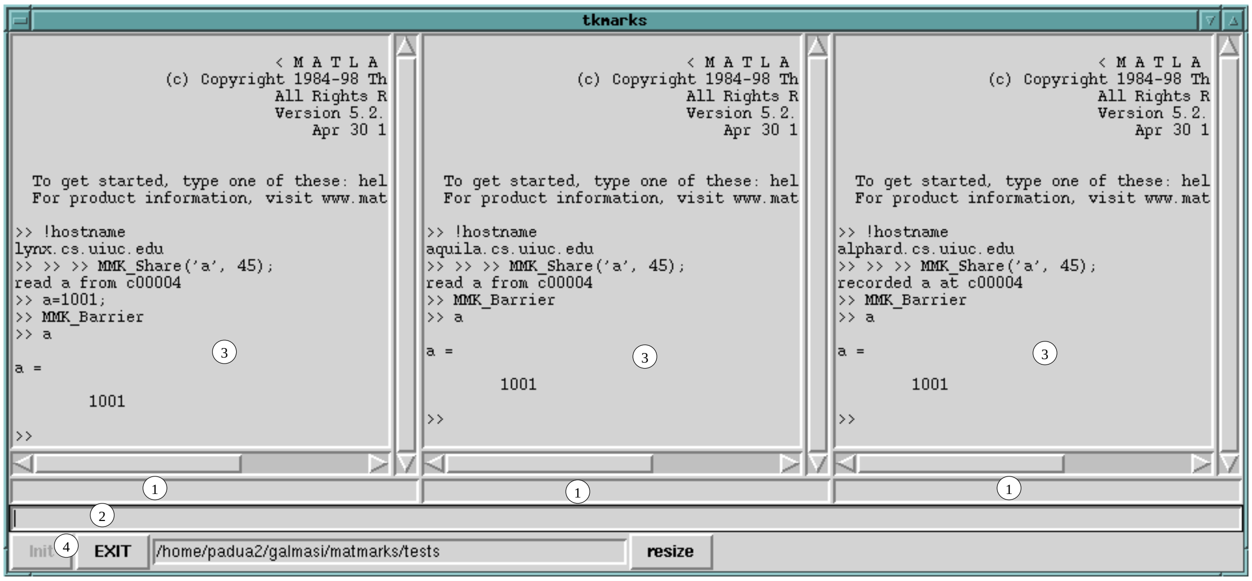

Figure 2: The MAT*marks* Tk/Tcl based User Interface

3 The MAT*marks* Extension Language

To facilitate parallel processing, we extended MATLAB with a set of commands. To keep the system simple, we tried to keep the number of these commands as small as possible. They can be ordered into the following categories:

Initialization and termination commands: `MMK_Init` and `MMK_exit` are used to initialize and terminate the MAT*marks* system. The initialization implies connecting several separate MATLAB interpreters and initializing the underlying TreadMarks layer.

Commands for process synchronization: `MMK_Barrier`, `MMK_Lock` and `MMK_Unlock`. These commands behave exactly like their Treadmarks counterparts [1]. `MMK_Barrier` provides a global point of synchronization, while `MMK_Lock` and `MMK_Unlock` provide the critical section semantics.

Commands for process identification: `MMK_proc_id` and `MMK_nprocs`. Again, these commands behave like the `Tmk_proc_id` and `Tmk_nprocs` variables in Treadmarks; `MMK_nprocs` returns the number of interpreters in the MAT*marks* group of processes; `MMK_proc_id` returns the process number of the current interpreter, numbered from 0 to `MMK_nprocs` -1.

Commands for declaring shared variables: The `MMK_Share` command, when executed by an interpreter, declares a variable to be shared; If any interpreter in the MAT*marks* process group changes the value of a shared variable the changed value will be propagated to all interpreters that have declared it shared.

The following example exposes the behavior of `MMK_Share`:

* interp 1 *	* interp 2 *	* interp 3 *
>MMK_Share ('a', 45);	>MMK_Share ('a', 45);	>MMK_Share ('a', 45);
>a = 1001;	>	>
>MMK_Barrier;	>MMK_Barrier;	MMK_Barrier;
>a	>a	>a
a = 1001	a = 1001	a = 1001

Here, two interpreters share the value of the variable `a`. When the variable is first initialized, the variable in each interpreter contains the value given in the initialization statement, i.e. `MMK_Share`. The value of `a` is synchronized at the first lock or barrier, conforming to the lazy release consistency model implemented by Treadmarks.

MAT*marks* does not attempt to implement scoping. All shared variables have to be at top level. This is a deliberate design choice aimed to keep the system simple.

As mentioned in Section 2, the declaration of shared variables is entirely dynamic, i.e. not all processes need to declare a shared variable at the same time. However, care must be taken to avoid unexpected behavior that might arise from a variable being local in one process and shared in all others. We do not consider this as good parallel programming style, and we would recommend avoiding sharing variables inconsistently across processes.

4 Implementation

This section explains the details of the MAT*marks* implementation. To integrate MATLAB with Treadmarks, we had to

- reconcile Treadmarks' page-based shared memory system with MATLAB's variables;
- maintain a global symbol table for shared variables; and
- find a way for MATLAB to handle variable sharing transparently.

In Treadmarks, a collection of processes runs on a set of workstations. The Treadmarks layer manages the memory to provide a shared view for all the processes. Treadmarks' shared memory system is page based. The unit of sharing is page with the size determined by the operating system (usually 4KB). To avoid excessive copying of pages, Treadmarks uses a mutiple-writer protocol based on page "diffs" - minimizing traffic at additional CPU expense.

To find out when a segment of memory has been referenced, Treadmarks unsets write permission to the shared memory pages, forcing page faults when they are referenced. When the page fault is captured, Treadmarks makes a copy of the original page to be able to build a "diff" later. When the consistency algorithm determines the need for reconciliation, diffs are built for the pages that have been changed. The diffs are used to update the shared pages on different processes.

The shared memory pages occupy the same address in all processes. To initialize shared memory, Treadmarks uses the `Tmk_distribute()` function. This function copies a buffer from a certain address one machine to the same address on all other processes.

Shared memory in MAT*marks* is variable based, and variables in MATLAB are dynamic. Therefore an additional level of indirection is necessary: the addresses of shared variables have to be redistributed to all Treadmarks processes.

The addresses of all shared variables are maintained by MAT*marks* in a symbol table independent of MATLAB itself. To access the shared symbol table, a process has to acquire a lock. The symbol table is redistributed immediately by using the `Tmk_distribute()` function if any changes were made. `Tmk_distribute()` is not subject to the lazy release consistency model, but acts instantly, therefore guaranteeing consistency of the symbol table across processes by the time the lock is released.

When MATLAB allocates a shared variable, MAT*marks* intercepts the call to the memory allocation functions and masquerades as a memory allocator. In reality MAT*marks* checks whether the shared variable in question has already been allocated by another process. If so, the address of the already allocated shared variable is returned. If the variable has not been allocated by anyone else then a new shared memory location is allocated using the `Tmk_alloc()` utility function, and the shared symbol table is modified and redistributed accordingly.

To make all this work transparently through MATLAB, we spoofed the interpreter by redirecting its memory allocation functions. Thus MATLAB is oblivious whether a variable is shared or not; it treats all its variables the same way. When MATLAB references shared variables, the Treadmarks page fault system handles them transparently.

5 Experimental Results

5.1 Benchmarks

All our benchmarks are parallel versions of simple MATLAB programs. Each of them are 100 lines or less in length.

Our first benchmark, MM, is a simple blocked parallel matrix multiplication routine. We chose a matrix size of 60x60 primarily to achieve a running time that could be measured reasonably accurately. Parallelism is achieved by breaking up the result matrix into column strips of appropriate sizes (e.g. 30x60 each for two processors) with each processor responsible to calculate its own strip.

MM2 is a slightly modified version of MM: we replaced scalar MATLAB code with vector code (i.e. vector expressions in the style of FORTRAN 90). Since MATLAB uses optimized BLAS routines to perform vector operations, the speed gains were impressive. To keep execution time in the same order of magnitude we had to increase matrix sizes to 500x500.

We used the MM and MM2 benchmarks to gauge the effect of system load on the performance of MAT*marks* . Synchronization between processes is minimal (in fact it is only necessary to collate the partial results into the final result).

The third benchmark, JACOBI, is an implementation the Jacobi relaxation method, used in this case to solve a two-dimensional elliptic PDE. Starting from the original serial code we set out to achieve parallel speedup with a minimal number of changes in the code. The workspace (a 50x50 matrix) is divided into equal slices, one for each processor. After each relaxation step the processors synchronize (and update their copy of the workspace) at a barrier. We took care to distribute the barriers such that each processor gets an approximately equal chance to "host" a barrier.

We rewrote JACOBI into JACOBI_OPT to gauge the impact of the absolute amount of network traffic on performance. We re-implemented the Jacobi relaxation algorithm by sharing and exchanging only the border zones between the different slices of the workspace. Thus the amount of network traffic in bytes decreases, whereas the amount of messages stays approximately the same.

The last benchmark, SIEVE, is a parallelized implementation of Eratosthenes' sieve for computing primes. In this version of the algorithm first a *reference list* of primes are computed serially by processor 0, and then the sieve uses this list to find other primes. The sieve itself is divided among the processors in such a fashion as to equalize the workload. This entails different array sizes for each processor, since the work needed to establish whether a number is a prime depends on the magnitude of that number.

5.2 Machine Setup

Our machine setup is as follows. We have a four-processor SUN Enterprise server at our disposal, and four single-processor Ultra 5 workstations. The server has a large amount of cache per processor (1MB) and a large amount of RAM (512MB). The workstations have only 64MB of RAM, and only 256KB of L2 cache, but they operate at a slightly higher clock speed. All workstations are connected to the server through a 100Mbit Ethernet connection. Each of the workstations is on the desktop of a member in our group; the server is used by about 30 people.

5.3 Discussion

Figure 3 shows our results for the five benchmarks we just described. Each figure shows a speedup curve for 1 to 8 processors.

Since we ran our experiments on a set of workstations with different capabilities, we encountered large variations in the individual processors' computation times even in the same experiment. This variation shows up as a predictable unevenness in our speedup curves. The processors in positions 1, 2, 6 and 8 were those of the SUN Enterprise server. We can identify two effects from this:

- all speedup curves exhibit upwards "bumps" in positions 2, 6 and 8.
- all speedup curves are generally depressed because the speedup reference was measured on processor 1.

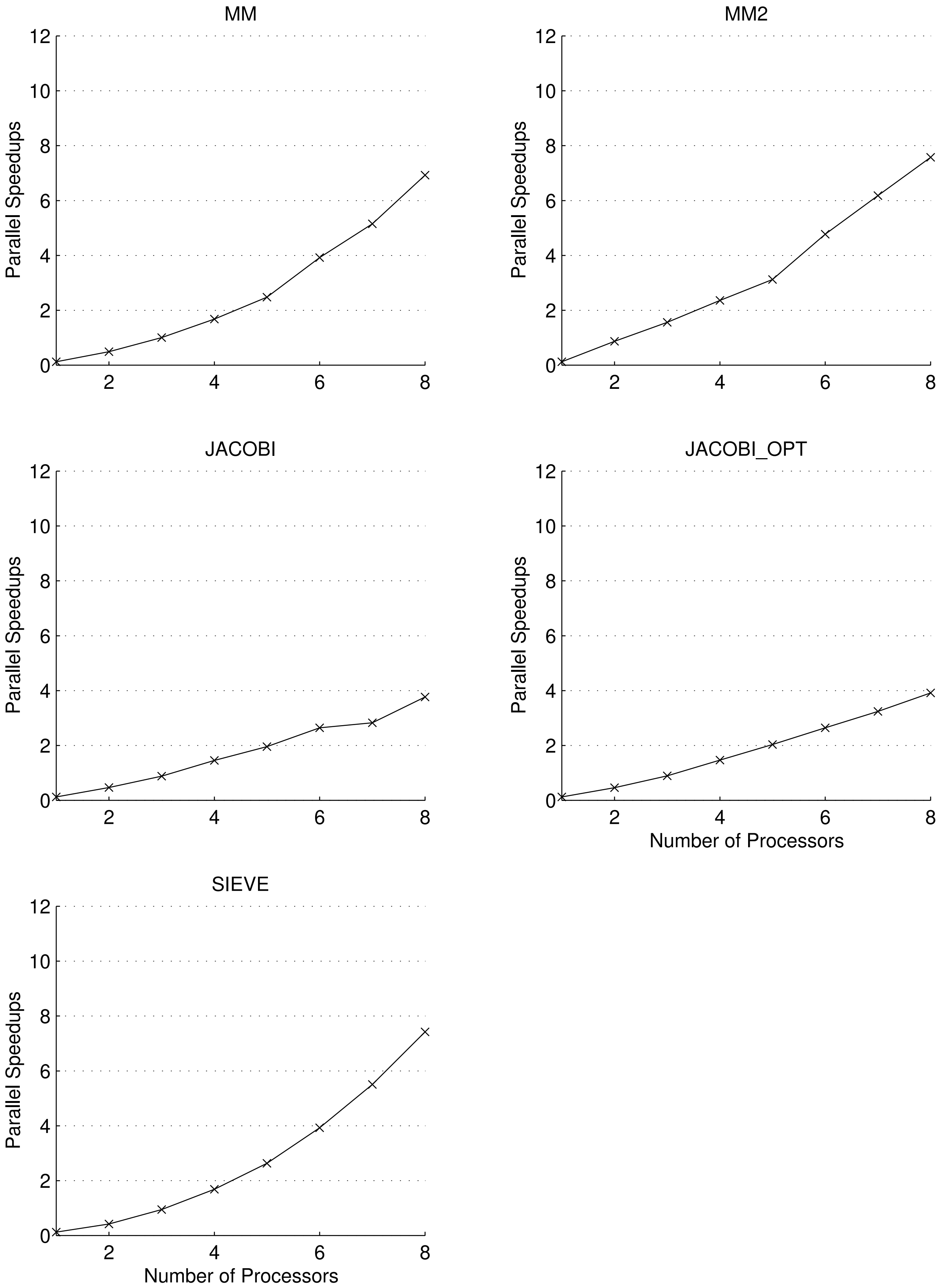

Figure 3: Parallel speedup curves for the five benchmarks on up to 8 processors

The MM benchmark shows a near-linear speedup and a relatively small variance across processors. There were no real surprises in this benchmark: communication requirements were almost inexistent (two barriers). The overall performance is low because of the MATLAB interpreter's overhead, but at least performance scales with added CPU power.

The MM2 benchmark also shows a linear speedup, but there are wild variations in processor performance. We attribute this principally to the fact that the processors had unequal amounts of L2 cache available. The problem involves accessing $8 \times 250000 \times (1 + \frac{2}{nprocs})$ bytes of data per processor (where $nprocs$ is the number of processors on the job). Whether the computation fits into the L2 cache or not depends on the individual processors. The processors with large amounts of cache show consistently better speedups than the rest.

Both JACOBI benchmarks show remarkably similar behavior and speedup. We expected better performance from the optimized version of the benchmark because it transmits a smaller number of bytes over the network; the similar behavior of the benchmarks suggests that it is the number of messages, not their number, which reduces the efficiency of the algorithm. To prove this point we plan to perform two more experiments: first, run a chaotic relaxation version of the JACOBI benchmark; second, run the benchmarks on a network that is slower but has the same latency.

The SIEVE benchmark shows an unexpected behavior of improving efficiency as the number of hosts increases. This is not due to any feature of MAT*marks* but to the algorithm itself, which tends to perform a smaller number of redundant divisibility checks when the workspace is more subdivided.

6 Related Work

As MATLAB has become more popular, many researchers have tried to develop systems that run in parallel. Most of the efforts [5, 13, 3] have focused on translating MATLAB programs into other high level languages, such as Fortran, Fortran 90, C, and C++. Some of these projects used parallelizing compilers to generate parallel codes; others employed already parallelized runtime libraries.

While the translation into other languages and the generation of executables may have its advantages, we consider that it defeats the purpose of MATLAB itself as a simple, rapid prototyping language.

Several other approaches are more similar to MAT*marks* [11, 7] in that they are extending MATLAB with sets of commands to run processes in parallel. However, both approaches cited above are based upon message passing systems, MultiMATLAB on MPI [6], and ParallelToolbox on PVM. We claim that the shared memory programming model used by MAT*marks* is much simpler to use than the one based on message passing. As opposed to the MPI approach, a MATLAB program doesn't need to be entirely rewritten to be parallelized - a much easier iterative approach will do instead.

7 Conclusions and Future Directions

MAT*marks* is simple, portable and reliable. It offers faster execution while preserving the advantage of an interactive environment. The changes to the MATLAB environment are minimal.

Performance results show that linear speedup can be achieved on a moderate number of workstations. While transforming a serial program into a shared memory parallel one is much easier than writing a message-based parallel program, for good performance one needs to be more careful coding a parallel algorithm. More research is needed to pin down the factors that limit parallel speedups (like in the case of the JACOBI benchmarks).

The MAT*marks* approach allows for incremental development of a parallel program, e.g. parallelizing the most time consuming loops first while leaving the rest of the program unchanged.

Previous work has shown that MATLAB execution speed can be improved by one, or even two, orders of magnitude by compilation. The next logical step is therefore to integrate MAT*marks* with a MATLAB compiler, such as FALCON, to obtain even better performance.

Acknowledgements

We would like to thank Stan Kerr for his help with the MATLAB license servers.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Weske, second edition, 1998.

[3] F. Bodin. MENHIR: High performance code generation for MATLAB. `http://www.irisa.fr/caps/PEOPLE/Francois/`.

[4] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, April 1988.

[5] L. A. DeRose and D. A. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings 1996 International Conference on Supercomputing*, pages 309–316, May 1996.

[6] M. P. I. Forum. *A Message-Passing Interface Standard*, May 1994.

[7] J. Hollingsworth, K. Liu, and P. Pauca. Parallel toolbox for matlab. Wake Forest University, 1996. `http://www.mthcsc.wfu.edu/pt/pt.html`.

[8] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, December 1994.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[10] Mathworks Inc. homepage. `www.mathworks.com`.

[11] V. Menon and A. E. Trefethen. MultiMATLAB: Integrating MATLAB with high-performance parallel computing. In *Proceedings of Supercomputing '97*, 1997. `http://simon.cs.cornell.edu/Info/People/lnt/multimatlab.html`.

[12] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[13] Integrated Sensors Inc. RTExpress. `http://www.rtexpress.com`.