Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. 64 (2004) 997–1005

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Short communication

MatlabMPI[☆]

Jeremy Kepner[a,*] and Stan Ahalt[b]

[a] Lincoln Laboratory, MIT 224 Wood Street, Lexington, MA 02420, USA
[b] Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210, USA

Received 2 January 2003; revised 4 March 2004

Abstract

In many projects the true costs of high performance computing are currently dominated by software. Addressing these costs may require shifting to higher level languages such as Matlab. MatlabMPI is a Matlab implementation of the Message Passing Interface (MPI) standard and allows any Matlab program to exploit multiple processors. MatlabMPI currently implements the basic six functions that are the core of the MPI point-to-point communications standard. The key technical innovation of MatlabMPI is that it implements the widely used MPI "look and feel" on top of standard Matlab file I/O, resulting in an extremely compact ($\sim$350 lines of code) and "pure" implementation which runs anywhere Matlab runs, and on any heterogeneous combination of computers. The performance has been tested on both shared and distributed memory parallel computers (e.g. Sun, SGI, HP, IBM, Linux, MacOSX and Windows). MatlabMPI can match the bandwidth of C based MPI at large message sizes. A test image filtering application using MatlabMPI achieved a speedup of $\sim$300 using 304 CPUs and $\sim$15% of the theoretical peak (450 Gigaflops) on an IBM SP2 at the Maui High Performance Computing Center. In addition, this entire parallel benchmark application was implemented in 70 software-lines-of-code, illustrating the high productivity of this approach. MatlabMPI is available for download on the web (www.ll.mit.edu/MatlabMPI).
© 2004 Published by Elsevier Inc.

Keywords: Message passing; High level languages; Parallel Matlab

1. Introduction

MATLAB®[1] is the dominant interpreted programming language for implementing numerical computations and is widely used for algorithm development, simulation, data reduction, testing and system evaluation. The popularity of Matlab is driven by the high productivity that is achieved by users because one line of Matlab code can typically replace 10 lines of C or Fortran code. Many Matlab programs can benefit from faster execution on a parallel computer, but achieving this goal has been a significant challenge. There have been many previous attempts to provide an efficient mechanism for running Matlab programs on parallel computers (see [1,2]) for thorough a survey).

The different parallel Matlab projects can be broken up into four broad categories: message passing [3–8], embarrassingly parallel [9–15], backend servers [16–21], and compiler based [22–28]. Each of these approaches has its performance and functionality tradeoffs. The tradeoff we are most concerned with here is from the software productivity perspective. The dominant mechanism used for obtaining parallelism is to connect the Matlab environment to another technology (usual C/C+ + or Fortran based) that does provide parallelism. The result is that many of these implementations rely on thousands of lines of C/C+ + and Fortran code, which the user has to modify if they wish to extend the functionality. For a typical parallel C/C+ + or Fortran application consisting of tens of thousands of lines of code, this is a reasonable requirement. For a Matlab

[☆] This work is sponsored by the High Performance Computing Modernization Office, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

*Corresponding author.
E-mail addresses: kepner@ll.mit.edu (J. Kepner), sca@ee.eng.ohio-state.edu (S. Ahalt).

[1] MATLAB is a registered trademark of The Mathworks, Inc.

0743-7315/$ - see front matter © 2004 Published by Elsevier Inc.
doi:10.1016/j.jpdc.2004.03.018

program that may only be 1000 lines and written by a user who has chosen not use C/C++ or Fortran, this can be a considerable hurdle.

In the world of parallel computing the Message Passing Interface (MPI) [29] is the de facto standard for implementing programs on multiple processors. MPI defines C and Fortran language functions for doing point-to-point communication in a parallel program. MPI has proven to be an effective model for implementing parallel programs and is used by many of the worlds' most demanding applications (weather modeling, weapons simulation, aircraft design, and signal processing simulation).

MatlabMPI [30–33] consists of a set of Matlab scripts that implements a subset of MPI and allows any Matlab program to be run on a parallel computer. The MPI interface provides a well defined and well understood mechanism for writing parallel programs using point-to-point messaging. The interface revolves around a standard data structure called a communicator, which holds all information about the different machines and their data buffers. This data structure is then used to with all communication requests and allows the underlying transport mechanism to be completely abstracted from the user. The key innovation of MatlabMPI is that it implements this widely used MPI "look and feel" on top of standard Matlab file I/O, resulting in a "pure" Matlab implementation that is exceedingly small (~350 lines of code). Thus, MatlabMPI will run on any combination of computers that Matlab supports.

The next section describes the implementation and functionality provided by MatlabMPI. Section 3 presents results on the bandwidth performance of the library from several parallel computers. Section 4 uses an image processing application to show the scaling performance that can be achieved using MatlabMPI. Section 5 presents our conclusions and plans for future work.

2. Implementation

The central innovation of MatlabMPI is its simplicity. MatlabMPI exploits Matlab's built in file I/O capabilities, which allow any Matlab variable (matrices, arrays, structures, ...) to be written and read by Matlab running on any machine, and eliminates the need to write complicated buffer packing and unpacking routines which would require ~100,000 lines of code to implement. The approach used in MatlabMPI is illustrated in Fig. 1. The sender saves a Matlab variable to a data file and when the file is complete the sender touches a lock file. The receiver continuously checks for the existence of the lock file; when it is exists the receiver reads in the data file and then deletes both the data file and the lock file. An example of a basic send and receive Matlab program is shown below

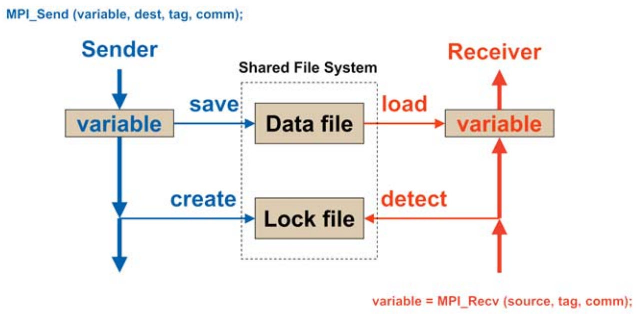

Fig. 1. File-based communication. Sender saves a variable in a Data file, then creates Lock file. Receiver detects Lock file, then loads Data file.

```
MPI_Init;                           % Initialize MPI.
comm = MPI_COMM_WORLD;              % Create communicator.
comm_size = MPI_Comm_size(comm);    % Get size.
my_rank = MPI_Comm_rank(comm);      % Get rank.
source = 0;                         % Set source.
dest = 1;                           % Set destination.
tag = 1;                            % Set message tag.
if(comm_size == 2)                  % Check size.
  if (my_rank == source)            % If source.
    data = 1:10;                    % Create data.
    MPI_Send(dest,tag,comm,data);   % Send data.
  end
  if (my_rank == dest)              % If destination.
    data=MPI_Recv(source,tag,comm); % Receive data.
  end
end
MPI_Finalize;                       % Finalize Matlab MPI.
exit;                               % Exit Matlab
```

The structure of the above program is very similar to MPI programs written in C or Fortran, but with the convenience of a high level language. The first part of the program sets up the MPI world; the second part differentiates the program according to rank and executes the communication; the third part closes down the MPI world and exits Matlab. If the above program were written in a matlab file called `SendReceive.m`, it would be executed by calling the following command from the Matlab prompt:

```
eval(MPI_Run('SendReceive',2,machines));
```

Where the `machines` argument can be any of the following forms:

`machines={};` Run on the local host.
`machines={'machine1' 'machine2'};` Run on a multi processors.
`machines={'machine1 : dir1' 'machine2 : dir2'};` Run on a multi processors and communicate using dir1 and dir2, which must be visible to both machines.

The `MPI_Run` command launches a Matlab script on the specified machines with output redirected to a file. If the rank = 0 process is being run on the local machine, `MPI_Run` returns a string containing the commands to initialize MatlabMPI, which allows MatlabMPI to be invoked deep inside of a Matlab program in a manner similar to fork-join model employed in OpenMP. This mechanism also allows MatlabMPI to be interactive on the rank = 0 process (rank $>$ 0 processes can also be interactive, but this is not the typical mode of operation).

The `SendReceive` example illustrates the basic six MPI functions (plus `MPI_Run`) that have been implemented in MatlabMPI

`MPI_Run` Runs a matlab script in parallel.
`MPI_Init` Initializes MPI.
`MPI_Comm_size` Gets the number of processors in a communicator.
`MPI_Comm_rank` Gets the rank of current processor within a communicator.
`MPI_Send` Sends a message to a processor (non-blocking).
`MPI_Recv` Receives message from a processor (blocking).
`MPI_Finalize` Finalizes MPI.

For convenience, three additional MPI functions have also been implemented along with two utility functions that provide important MatlabMPI functionality, but are outside the MPI specification:

`MPI_Abort` Function to kill all matlab jobs started by MatlabMPI.
`MPI_Bcast` Broadcast a message (blocking).
`MPI_Probe` Returns a list of all incoming messages.
`MatMPI_Save_messages` MatlabMPI function to prevent messages
from being deleted (useful for debugging).
`MatMPI_Delete_all` MatlabMPI function to delete all files created by MatlabMPI.

MatlabMPI handles errors the same as Matlab, however running hundreds of copies does bring up some additional issues. If an error is encountered and the Matlab script has an “exit” statement then all the Matlab processes will die gracefully. If a Matlab job is waiting for a message that never arrives then it needs to be killed with the `MPI_Abort` command. In this situation, MatlabMPI can leave files which need to be cleaned up with the `MatMPI_Delete_all` command.

On shared memory systems, MatlabMPI only requires a single Matlab license since any user is allowed to launch many Matlab sessions. On a distributed memory system, MatlabMPI requires one Matlab license per machine. Because MatlabMPI uses file I/O for communication, there must also be a directory that is visible to every machine (this is usually also required in order to install Matlab). This directory defaults to the directory that the program is launched from.

3. Bandwidth

Many Matlab applications are “embarrassingly” parallel and require minimal performance out of MatlabMPI. These applications exploit coarse grain parallelism and communicate rarely (if at all). Nevertheless, measuring the communication performance is useful for determining which applications are most suitable for MatlabMPI.

MatlabMPI has been run on several Unix platforms. It has been benchmarked and compared to the performance of C MPI on the SGI Origin2000 at Boston University. These results indicate that for large messages ($\sim$1 MByte) MatlabMPI is able to match the performance of C MPI (see Fig. 2). For smaller messages, MatlabMPI is dominated by its latency ($\sim$35 ms), which is significantly larger than C MPI. These results have been reproduced using a SGI Origin2000 and a Hewlett Packard workstation cluster (connected with 100 Mbit ethernet) at Ohio State University (see Fig. 3).

The above bandwidth results were all obtained using two processors engaging in bi-directional sends and receives. Such a test does a good job of testing the individual links on an a multi-processor system. To more broadly test the interconnect the send receive benchmark is run on an eight node (16 cpu) Linux cluster connected with Gigabit ethernet (see Fig. 4). These results are shown for one and 16 cpus. MatlabMPI is able to maintain high bandwidth even when multiple processors are communicating by allowing each processor to have its own receive directory. By cross mounting all the disks in a cluster each node only sees the traffic directed to it, which allows communication contention to be kept to a minimum.

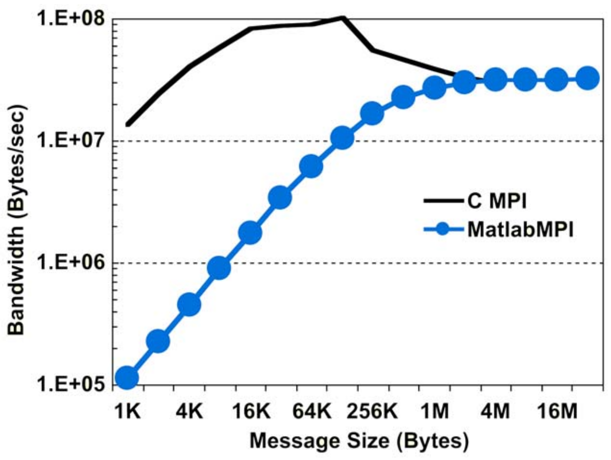

Fig. 2. MatlabMPI vs. MPI Bandwidth. Communication performance as a function message size on the SGI Origin2000. MatlabMPI equals C MPI performance at large message sizes.

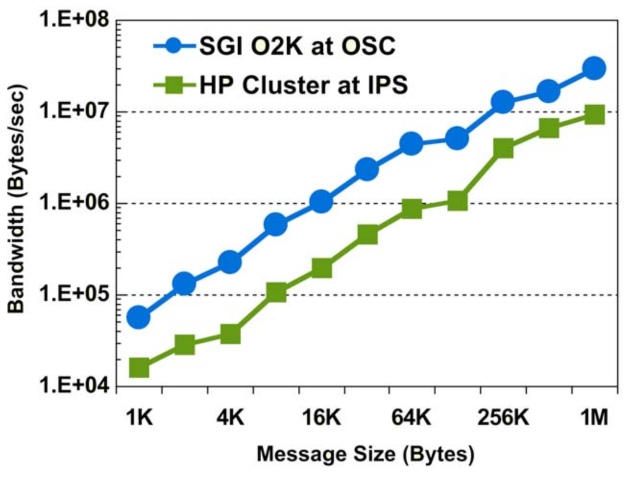

Fig. 3. Bandwidth comparison. Bandwidth measured on Ohio St. SGI Origin2000 and a Hewlett Packard workstation cluster.

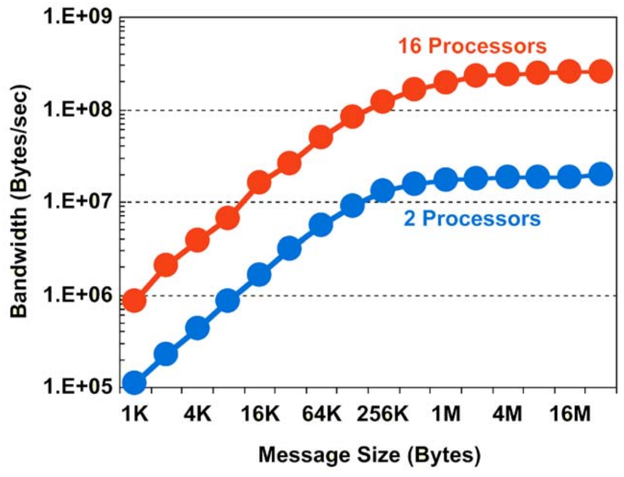

Fig. 4. Bandwidth on a Linux cluster. Communication performance on Linux cluster with Gigabit ethernet. MatlabMPI is able to maintain communication performance even when larger numbers of processor are communicating.

4. Scaling of an image processing kernel

To further test MatlabMPI a simple image filtering application was written. This application abstracts the key computations that are used in many of our DoD sensor processing applications (e.g. wide area Synthetic Aperture Radar). At a given pixel, the convolution is performed by multiplying all pixels in the region surrounding the pixel by a small filter image (e.g. 64×64) and accumulating the result. Typically, an image $I(x,y)$ is acquired by a sensor or extracted from an archive. We wish to convolve or filter this image using a kernel $K(x,y)$ to produce a filtered image $F(x,y)$. Mathematically this is equivalent to

$$F(x,y) = \int\int K(x',y')I(x-x',y-y')\,dx'\,dy'. \quad (1)$$

For convenience it is reasonable to assume the image and the kernel are $M \times M$ and $N \times N$ squares, respectively (this condition can be relaxed later without loss of generality). In the discrete case, the above convolution can be computed by the double sum

$$F(i,j) = \sum_{i'=0}^{N-1}\sum_{j'=0}^{N-1} K(i',j')I(i-i',j-j'), \quad (2)$$

where i and j are pixel indices.

The kernel application executed repeated 2D convolutions on a large image ($1024 \times 128,000$ ~ 2 GBytes). This application was run on a large shared memory parallel computer (the SGI Origin2000 at Boston University) and achieved speedups greater than 64 on 64 processors; showing the classic super-linear speedup (due to better cache usage) that comes from breaking a very large memory problem into many smaller problems (see Fig. 5).

To further test the scalability, the image processing application was run with a constant load per processor (1024×1024 image per processor) on a large shared/distributed memory system (the IBM SP2 at the Maui High Performance Computing Center). The performance of this scaled application was compared to the pure serial implementation, using only the Matlab

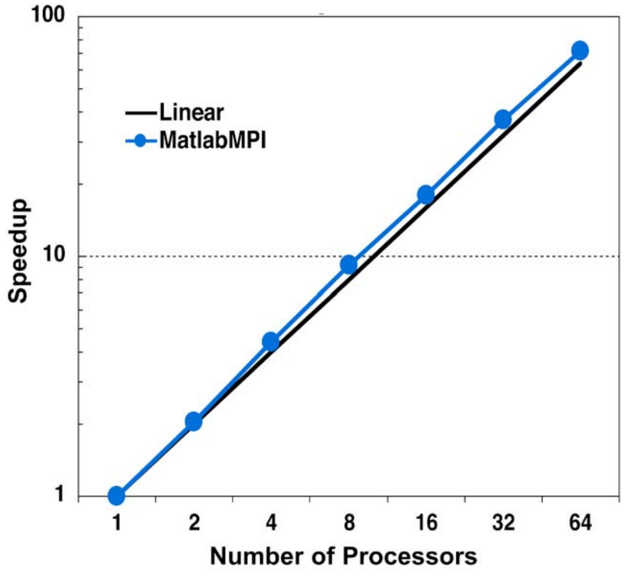

Fig. 5. Shared memory parallel speedup. Speed increase on the SGI Origin2000 of a parallel image filtering application as a function of the number of processors. Application shows "classic" super-linear performance (due to better cache usage) that results when a very large memory problem is broken into multiple small memory problems.

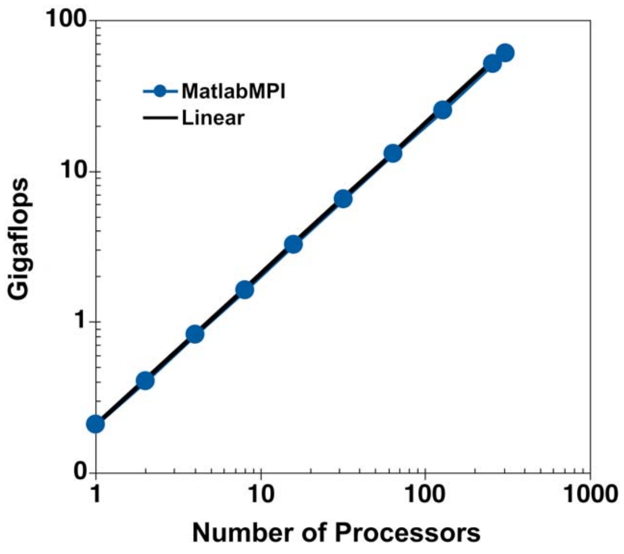

Fig. 6. Shared/distributed parallel speedup. Measured performance on the IBM SP2 of a parallel image filtering application. Application achieves a speedup of ~300 on 304 processors and ~15% of the theoretical peak (450 Gigaflops).

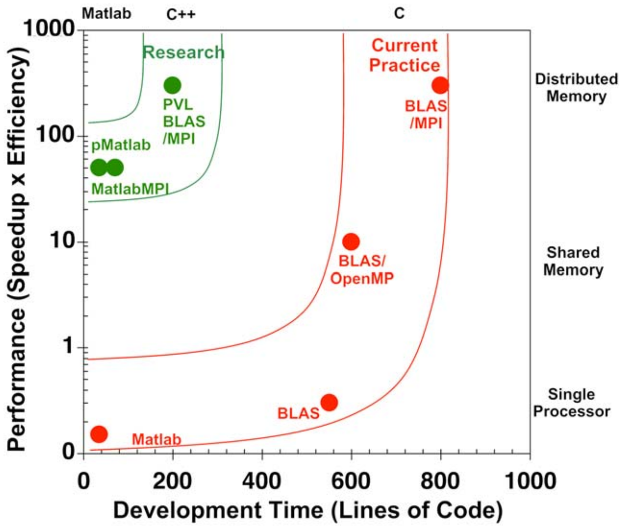

Fig. 7. Performance vs. development time Maximum achieved performance (measured in units of single processor theoretical peak) as a function of development time (measured in lines of code) for different implementations of the same image filtering application. Higher level languages allow the same application to be implemented with fewer lines of code. Increasing the maximum performance generally increases the lines of code. MatlabMPI allows high level languages to run on multiple processors. Parallel Matlab is an estimate of what will be possible using the Parallel Matlab Toolbox.

"conv" routine, which is the fastest algorithm available in this environment. In this test, the application achieved a speedup of ~300 on 304 CPUs as well achieving ~15% of the theoretical peak (450 Gigaflops) of the system (see Fig. 6).

The ultimate goal of running Matlab on parallel computers is to increase programmer productivity and decrease the large software cost of using HPC systems. Fig. 7 plots the software cost (measured in Software Lines of Code or SLOCs) as a function of the maximum achieved performance (measured in units of single processor peak) for the same image filtering application implemented using several different libraries and languages (VSIPL, MPI, OpenMP, using C++, C, and Matlab). These data show that higher level languages require fewer lines to implement the same level of functionality. Obtaining increased peak performance (i.e. exploiting more parallelism) requires more lines of code. MatlabMPI is unique in that it achieves a high peak performance using a small number of lines of code.

5. Conclusions and future work

The use of file I/O as a parallel communication mechanism is not new and is now increasingly feasible with the availability of low cost high speed disks. The extreme example of this approach are the now popular Storage Area Networks (SAN), which combine high speed routers and disks to provide server solutions. Although using file I/O increases the latency of messages it normally will not effect the bandwidth. Furthermore, the use of file I/O has several additional functional advantages which make it easy to implement large buffer sizes, recordable messages, multi-casting, and one-sided messaging. Finally, the MatlabMPI approach is readily applied to any language (e.g. IDL, Python, and Perl).

MatlabMPI demonstrates that the standard approach to writing parallel programs in C and Fortran (i.e., using MPI) is also valid in Matlab, and many users have been able to exploit it [34–37]. In addition, by using Matlab file I/O, it was possible to implement MatlabMPI entirely within the Matlab environment, making it instantly portable to all computers that Matlab runs on. Most potential parallel Matlab applications are trivially parallel and don't require high performance. Nevertheless, MatlabMPI can match C MPI performance on large messages. The simplicity and performance of MatlabMPI makes it a very reasonable choice for programmers that want to speed up their Matlab code on a parallel computer.

MatlabMPI provides the highest productivity parallel computing environment available and can go a long way towards addressing the true costs of high performance computing are currently dominated by software [38,39]. However, because it is a point-to-point messaging library, a significant amount code of must be added to any application in order to do basic parallel operations.

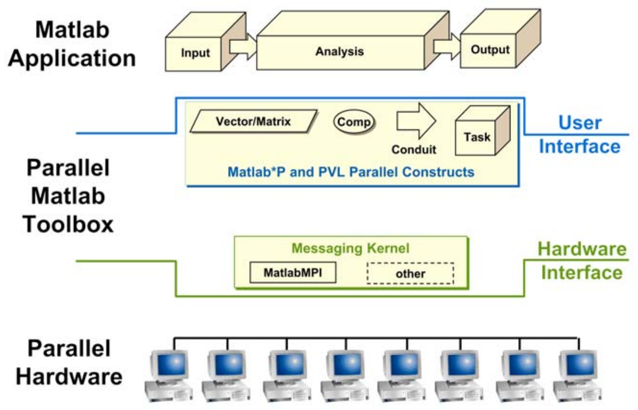

Fig. 8. Future layered architecture. Design of the Parallel Matlab Toolbox which will create distributed data structures and dataflow objects built on top of MatlabMPI (or any other messaging system).

In the test application presented here, the number of lines of Matlab code increased from 35 to 70. While a 70 line parallel program is extremely small, it represents a significant increase over the single processor case. Our future work will aim at creating higher level objects (e.g., distributed matrices) that will eliminate this parallel coding overhead (see Fig. 8). The resulting "Parallel Matlab Toolbox" will be built on top of the MatlabMPI communication layer, and will allow a user to achieve good parallel performance without increasing the number lines of code.

Acknowledgments

The authors would like to thank the sponsorship of the DoD High Performance Computing Modernization Office, and the staff at the supercomputing centers at Boston University, Ohio State University and the Maui High Performance Computing Center. We would also like to thank John Nehrbass for his technical contributions.

Appendix A. Parallel image filtering test application

```
MPI_Init;  % Initialize MPI.
comm = MPI_COMM_WORLD;  % Create communicator.
comm_size = MPI_Comm_size(comm);  % Get size.
my_rank = MPI_Comm_rank(comm);  % Get rank.
% Do a synchronized start.
starter_rank = 0;
delay = 30;  % Seconds
synch_start(comm,starter_rank,delay);
n_image_x = 2.^(10+1)*comm_size;  % Set image size (use powers of 2).
n_image_y = 2.^10;
n_point = 100;  % Number of points to put in each sub-image.
% Set filter size (use powers of 2).
n_filter_x = 2.^5;
n_filter_y = 2.^5;
n_trial = 2;  % Set the number of times to filter.
% Computer number of operations.
total_ops = 2.*n_trial*n_filter_x*n_filter_y*n_image_x*n_image_y;
if(rem(n_image_x,comm_size) ~= 0)
 disp('ERROR: processors need to evenly divide image');
 exit;
end
disp(['my_rank: ',num2str(my_rank)]);  % Print rank.
left = my_rank - 1;  % Set who is source and who is destination.
if (left < 0)
  left = comm_size - 1;
end
right = my_rank + 1;
if (right >= comm_size)
  right = 0;
end
tag = 1;  % Create a unique tag id for this message.
start_time = zeros(n_trial);  % Create timing matrices.
end_time = start_time;
zero_clock = clock;  % Get a zero clock.
n_sub_image_x = n_image_x./comm_size;  % Compute sub_images for each processor.
n_sub_image_y = n_image_y;
```

```
% Create starting image and working images..
sub_image0 = rand(n_sub_image_x,n_sub_image_y).^10;
sub_image = sub_image0;
work_image = zeros(n_sub_image_x+n_filter_x,n_sub_image_y+n_filter_y);
% Create kernel.
x_shape = sin(pi.*(0:(n_filter_x-1))./(n_filter_x-1)).^2;
y_shape = sin(pi.*(0:(n_filter_y-1))./(n_filter_y-1)).^2;
kernel = x_shape.' * y_shape;
% Create box indices.
lboxw = [1,n_filter_x/2,1,n_sub_image_y];
cboxw = [n_filter_x/2+1,n_filter_x/2+n_sub_image_x,1,n_sub_image_y];
rboxw = [n_filter_x/2+n_sub_image_x+1,n_sub_image_x+n_filter_x,1,n_sub_image_y];
lboxi = [1,n_filter_x/2,1,n_sub_image_y];
rboxi = [n_sub_image_x-n_filter_x/2+1,n_sub_image_x,1,n_sub_image_y];
start_time = etime(clock,zero_clock);  % Set start time.
% Loop over each trial.
for i_trial = 1:n_trial
  % Copy center sub_image into work_image.
  work_image(cboxw(1):cboxw(2),cboxw(3):cboxw(4)) = sub_image;
  if (comm_size > 1)
    ltag = 2.*i_trial;    % Create message tag.
    rtag = 2.*i_trial+1;
    % Send left sub-image.
    l_sub_image = sub_image(lboxi(1):lboxi(2),lboxi(3):lboxi(4));
    MPI_Send(  left, ltag, comm, l_sub_image );
    % Receive right padding.
    r_pad = MPI_Recv( right, ltag, comm );
    work_image(rboxw(1):rboxw(2),rboxw(3):rboxw(4)) = r_pad;
    % Send right sub-image.
    r_sub_image = sub_image(rboxi(1):rboxi(2),rboxi(3):rboxi(4));
    MPI_Send( right, rtag, comm, r_sub_image );
    % Receive left padding.
    l_pad = MPI_Recv( left, rtag, comm );
    work_image(lboxw(1):lboxw(2),lboxw(3):lboxw(4)) = l_pad;
  end
  work_image = conv2(work_image,kernel,'same');   % Compute convolution.
  % Extract sub_image.
  sub_image = work_image(cboxw(1):cboxw(2),cboxw(3):cboxw(4));
end
end_time = etime(clock,zero_clock);  % Get end time for the this message.
total_time = end_time - start_time  % Print the results.
% Print compute performance.
gigaflops = total_ops / total_time / 1.e9;
disp(['GigaFlops: ',num2str(gigaflops)]);
MPI_Finalize;  % Finalize Matlab MPI.
exit;
```

References

[1] R. Choy, Interactive Supercomputing Made Practical, Master of Science Thesis, Department of Electrical Engineering and Computer Science, MIT, August 2002.

[2] R. Choy, Parallel Matlab survey, 2004, http://supertech.lcs.mit.edu/~cly/survey.html.

[3] A.E. Trefethen, V.S. Menon, C.C. Chang, G.J. Czajkowski, C. Myers, L.N. Trefethen, MultiMatlab: Matlab on multiple processors. Technical Report 96-239, Cornell Theory Center, 1996, MultiMatlab: www.cs.cornell.edu/Info/People/lnt/multimatlab.html.

[4] J.A. Zollweg, A. Verma, The Cornell Multitask Toolbox, CMTM: www.tc.cornell.edu/Services/Software/CMTM.

[5] S. Pawletta, A. Westphal, T. Pawletta, W. Drewelow, P. Duenow, Distributed and Parallel Application Toolbox (DP Toolbox) for use with Matlab—User's Guide and Reference Manual Version 1.4. Institute of Automatic Control, University of Rostock, 1999, DP-Toolbox: www-at.e-technik.uni-rostock.de/dp.

[6] J. Fernandez, A. Canas, A.F. Doaz, J. Gonzalez, J. Ortega, A. Prieto, Performance of Message-Passing MATLAB Toolboxes, High Performance Computing for Computational Science—VECPAR 2002: Fifth International Conference, Porto, Portugal, June 26–28, 2002, MPITB: atc.ugr.es/javier-bin/mpitb_eng.

[7] G. Almasi, C. Cascaval, D.A. Padua, MATmarks: A Shared Memory Environment for MATLAB Programming, High Performance Distributed Computing Conference, Los Angeles, August 1–11, 1999, MATmarks: polaris.cs.uiuc.edu/matmarks.

[8] V.P. Pauca, J. Hollingsworth, K. Liu, User's Guide for the Parallel Toolbox for MATLAB, Technical Report, Wake Forest University, May, 1995.

[9] T. Krauss, MULTI Toolbox: www.lapsi.eletro.ufrgs.br/Disciplinas/ENG_ELETRICA/CAD-ENG/Matlab/CommSim/COMMSIM%20for%20MATLAB%205.htm.

[10] T. Abrahamsson, Paralize: www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId = 211.

[11] D. Lee, Parallel Matlab Interface, PMI: www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId = 219.

[12] U. Kjems, Parallel Matlab, PLab: bond.imm.dtu.dk/plab/.

[13] L. Andrade, E.S. Manolakos, XIII Research Workshop, Communications and Digital Signal Processing Center, Northeastern University, May 2002 "PARMATLAB: Coarse Parallelization of Matlab Processes in Heterogeneous Networks," Parmatlab: www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId = 217.

[14] E. Heiberg, MATLAB Parallelization Toolkit: hem.passagen.se/einar_heiberg/index.html.

[15] M. Devore, DistributePP: www.essrl.wustl.edu/˜mdd2/DistributePP.

[16] S. Agrawal, J. Dongarra, K. Seymour, S. Vadhiyar, "NetSolve: Past, Present, and Future—A Look at a Grid Enabled Server," in: F. Berman, G. Fox, A. Hey (Eds.), Making the Global Infrastructure a Reality, Wiley Publishing, New York, 2003, Netsolve: icl.cs.utk.edu/netsolve.

[17] B.R. Norris, An Environment for Interactive Parallel Numeric Computing, Ph.D. Thesis, Department of Computer Science, UIUC, 2000, DLab: www.cse.uiuc.edu/˜radenska.

[18] P. Springer, Matpar: parallel extensions for MATLAB, PDPTA proceedings Las Vegas, July 1998, Matpab: www-hpc.jpl.nasa.gov/PS/MATPAR.

[19] G. Morrow, Robert van de Geijn, A parallel linear algebra server for Matlab-like Environments, Supercomputing 1998, PLACKPACK: www.cs.utexas.edu/users/plapack.

[20] Paramat: www.mathworks.com/products/connections/product_main.shtml?prod_id = 12, Alpha-Data.

[21] R. Choy, A. Edelman, Parallel MATLAB: Doing it Right, IEEE Proceedings, 2003, submitted for publication, MATLAB*P: supertech.lcs.mit.edu/˜cly/matlabp.html.

[22] M.J. Quinn, A. Malishevsky, Otter: Bridging the Gap between MATLAB and ScaLAPACK, in: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, July 1998, pp. 114–121.

[23] RTExpress: www.rtexpress.com.

[24] I. Milosavljevic, M. Partridge, R. Calvo, M. Jabri, High performance principal component analysis with ParAL, in: Proceedings of the Ninth Australian Conference on Neural Networks, Brisbane, 1998.

[25] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, D. Padua, FALCON: A MATLAB Interactive Restructuring Compiler, in: Languages and Compilers for Parallel Computing, Springer, Berlin, August 1995, pp. 269–288; (Eighth International Workshop, LCPC'95, Columbus, Ohio.). FALCON: www.csrd.uiuc.edu/falcon/falcon.html.

[26] P. Jacobson, B. Kagstrom, M. Rannar, Algorithm development for distributed memory multicomputers using CONLAB, Sci. Programming 1 (1992) 185–203.

[27] M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary, P. Banerjee, A Library-Based Compiler to Execute MATLAB Programs on a Heterogeneous Platform, ISCA 13th International Conference on Parallel and Distributed Computing Systems, MATCH: www.ece.northwestern.edu/cpdc/Match/Match.html.

[28] S. Chauveau, F. Bodin, MENHIR: An environment for high performance Matlab, in: D. O'Hallaron (Ed.), Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98 Workshop), Vol. 1511, Carnegie Mellon University, May 28, 1998, Springer, Berlin, Menhir: www.irisa.fr/caps/PROJECTS/Menhir.

[29] Message Passing Interface (MPI), www.mpi-forum.org.

[30] J. Kepner, Parallel Programing with MatlabMPI, Fifth High Performance Embedded Computing workshop (HPEC 2001), MIT Lincoln Laboratory, Lexington, MA, September 25–27, 2001.

[31] J. Kepner, J. Kepner, 300x Matlab, Sixth High Performance Embedded Computing workshop (HPEC 2002), MIT Lincoln Laboratory, Lexington, MA, September 25–27, 2002.

[32] J. Kepner, S. Ahalt, Parallel image processing with MatlabMPI, DOD HPCMP User's Group Conference, Bellevue, WA, June 12, 2003.

[33] J. Kepner, N. Travinin, Parallel matlab: the next generation, Seventh High Performance Embedded Computing workshop (HPEC 2003), MIT Lincoln Laboratory, Lexington, MA, September 23–25, 2003.

[34] J. Nehrbass, S. Ahalt, J. Chaves, A. Krishnamurthy, M. Soumekh, Parallel performance of pure MATLAB "M-files" Versus "C-code" as applied to formation of wide-bandwidth and wide-beamwidth SAR Imagery, Seventh High Performance Embedded Computing workshop (HPEC 2003), MIT Lincoln Laboratory, Lexington, MA, September 23–25, 2003.

[35] D. Stein, A parallel implementation of the normal compositional model for hyperspectral analysis based on MatlabMPI, Seventh High Performance Embedded Computing workshop (HPEC 2003), MIT Lincoln Laboratory, Lexington, MA, September 23–25, 2003.

[36] P. Khot, S. Ahalt, J. Chaves, A. Krishnamurthy, J. Nehrbass, A parallel data mining toolbox Using MatlabMPI, Seventh High Performance Embedded Computing workshop (HPEC 2003), MIT Lincoln Laboratory, Lexington, MA, September 23–25, 2003.

[37] G. Landi, E. Loli Piccolomini, F. Zama, A parallel software for the reconstruction of dynamic MRI sequences, Proceedings of 10th European meeting PVM/MPI, 29 September 2 October 2003, Lecture Notes in Computer Science, Vol. 2840, Springer, Berlin, 2003, pp. 511–519.

[38] F.P. Brooks, The Mythical Man Month: Essays on Software Engineering, Addison-Wesley, Reading, MA, 1992.

[39] J. Kepner (Ed.), HPC productivity model synthesis, Internat. J. High Performance Comput. Appl.: Special Issue on HPC Productivity 18, 4, Winter 2004 (November), to appear.

Stan Ahalt has been on the faculty of the Department of Electrical and Computer Engineering at The Ohio State University since 1987. His research interests are in signal processing, data compression, and neural networks, and in the use of high performance computing for signal processing tasks. Dr. Ahalt has published over 100 archival journal articles, conference papers, and book chapters and was an Associate Editor of the IEEE Transactions on Neural Networks. Prior to joining OSU, Dr. Ahalt worked at Bell Telephone Laboratories where he developed industrial data products. He received his BS and MS degrees in engineering from Virginia Tech and his Ph.D. in Electrical and Computer Engineering from Clemson University. He was appointed Executive Director of the Ohio Supercomputer Center (OSC) on July 1, 2003.

Jeremy Kepner is a staff member in the Embedded Digital Systems group. His research has addressed the development of parallel algorithms and tools, and the application of massively parallel computing to a variety of data-intensive problems. He received a B.A. degree in astrophysics from Pomona College, and a Ph.D. degree in astrophysics from Princeton University in 1998, after which he joined Lincoln Laboratory.