A Survey Of Available Matlab Tools For MPI/Parallel Programming[†]

Anirban Sinha
anirbans@cs.ubc.ca

1 INTRODUCTION

Matlab is widely popular software used in many applications like in image processing, in statistical estimations, among machine learning community etc. Problems in these domains are computationally intensive requiring good processing power to solve the mathematical models within a reasonable period of time. Use of parallel computation in these domains can highly benefit research community in solving these problems fast and efficient. Use of parallel tools within matlab can thus help to port the already existing programs written in matlab into their parallel versions and execute them on independent nodes requiring least effort and time. In this write up, we explore the available matlab tools for designing parallel programs and running parallel matlab/mpi programs. We take up three tools currently available for parallel program design and describe each of them in brief.
The rest of this write-up is organized as follows. In section 2 we describe MatlabMPI which is a matlab toolkit for writing MPI look and feel programs. Section 3 describes the distributed Matlab toolbox which also has capabilities for writing parallel Matlab programs. In section 4 we describe the MPI toolbox for Matlab (MPITB) which is similar to MatlabMPI but can be regarded as the Matlab wrapper over LAM/MPI implementation. In our last section, section 5, we summarize our findings and propose future work in this direction.

2 MATLAB MPI

MatlabMPI is set of Matlab scripts that implement a subset of MPI designed by Dr. Jeremy Kepner in the Lincoln Laboratory at Massachusetts Institute of Technology. It implements the commonly known MPI "look and feel" on top of standard Matlab file I/O. This results in an original Matlab implementation that is very small. Matlab MPI runs on any computer where Matlab can be run. It consists of a set of matlab scripts with their corresponding libraries that can be used to design parallel matlab programs. The basic requirement for MatlabMPI is Matlab itself. If used in shared memory systems, it requires a common filesystem visible to all processors. The installation procedure is simple and can be found at the MatlabMPI homepage [1]. We did not find any problems while installing MatlabMPI following the specified procedure. There are a number of interesting sample programs written in matlab provided in “/src” sub directory which can be taken and run using MatlabMPI. The most important advantage of using this tool is that the function interfaces are all similar to MPI interface and thus, the learning curve for anyone who is already familiar with MPI program using LAM or OpenMPI is a lot smoother.
A typical command for running a parallel program in MatlabMPI would be: “*eval(MPI_Run('xbasic', 2,machines));*” where the first argument “*xbasic*” is the name of the matlab script containing the parallel program, second argument is the number of instances of this program to run and the third argument is the list of machines. To run on a local processor, this list should be declared as “*machines = {};*” Otherwise, for running on multiprocessors, the list can be declared as, “*machines = {'machine1' 'machine2'}*”. After executing, these scripts output their

[†] First Uploaded: Monday, March 20, 2006. Last modified: Monday, March 20, 2006, Credits: 05

results on a local text file. Hence, between two independent runs of the same program, "*MatMPI_Delete_all*" must be used to clean up stale output files from the previous run. It is like running a "*lamclean*" on a LAM/MPI system. MatlabMPI also comes with a useful command "*MPI_Abort*" that helps programmers to kill MPI processes that would never complete their execution. It currently works with only Windows XP however.
A list of all available functions can be obtained by typing "*help MatlabMPI*" at matlab command prompt. Details about a specific command can be obtained by typing "*help*" followed by the name of the command. A sample MatlabMPI source code is shown in appendix A.
MatlabMPI has been declared to be compatible with Linux, OSX and Windows based machines running the corresponding versions of Matlab. Although all basic MPI calls are supported, one important weakness of this tool however is that it does not have equivalent matlab scripts for each and every MPI function calls available in MPI documentation.

3 THE DISTRIBUTED COMPUTING TOOLBOX IN MATLAB [§]

The Distributed Computing Toolbox and the MATLAB Distributed Computing Engine enables one to develop distributed MATLAB applications and execute them in a cluster of computers without leaving the Matlab development environment. According to the MathWorks homepage, algorithms that require interdependent tasks use the Message Passing Interface (MPI)-based functions to communicate with each other. The MATLAB Distributed Computing Engine schedules and evaluates tasks on multiple remote MATLAB sessions, reducing execution time compared to running in a single MATLAB session. A single Matlab program is termed as a job. This job is broken down into segments called tasks. The programmer can decide how to break up the large jobs into small sized tasks. Tasks may or may not be identical. The job manager is the part of the engine that coordinates the execution of jobs and the evaluation of their tasks. The job manager distributes the tasks for evaluation to the engine's individual MATLAB sessions called workers. The job manager can be run on any machine on the network. The job manager runs jobs in the order in which they are submitted, unless any jobs in its queue are promoted, demoted, canceled, or destroyed. Each worker is given a task from the running job by the job manager. The workers executes the task and returns the result to the job manager Then they are given another task. When all tasks for a running job have been assigned to workers, the job manager starts running the next job with the next available worker. The Matlab session in which the job and its tasks are defined is called the client session. The relationship between clients, job manager and workers is shown in figure 1.

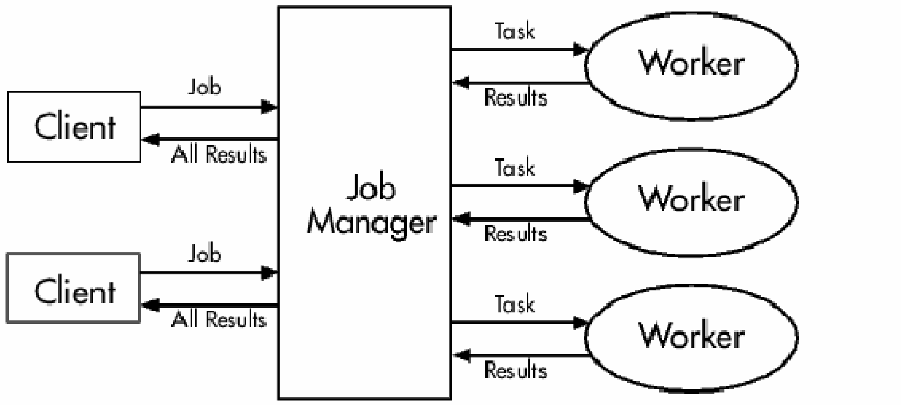

Figure 1: Relationship between client, Job Manager and Worker processes.

[§] Portions of this section are taken directly from Matlab Distributed Computing Toolbox help files.

Every machine that hosts a worker or job manager session must run the Matlab Distributed Computing Engine (MDCE) daemon. The MDCE daemon makes it possible for these processes on different machines to communicate with each other. It is similar to the LAM daemon. However, the MDCE daemon is also responsible for recovering worker and job manager sessions when their host machines crash. If a worker or job manager machine crashes, when MDCE starts up again (usually configured to start at machine boot time), it automatically restarts the job manager and worker sessions to resume their sessions from before the system crash.
The distributed computing toolbox supports heterogeneous nodes, meaning that the nodes can have different platforms.
Figure 2 shows the stages of processing a job. Each job can have several states, namely, pending, queued, running and finished. When we first create a job using the "*createJob*" function, the job is placed in pending queue. When we "*submit*" the job, the job is termed as queued. The job manager executes jobs in the queue in the sequence in which they are submitted, all jobs moving up the queue as the jobs before them are finished. We can change the order of the jobs in the queue with the "*promote*" and "*demote*" functions. When a job reaches the top of the queue, the job manager distributes the job's tasks to worker sessions for evaluation. The job's state becomes running. If more workers are available than necessary for a job's tasks, the job manager begins executing the next job. In this way, there can be more than one running job at a time. When all of a job's tasks have been evaluated, a job is moved to the finished state. At this time, we can retrieve the results from all the tasks in the job with the function "*getAllOutputArguments*". When a job is finished, it remains in the job manager. The job manager keeps all the jobs it has executed, until we restart the job manager in a clean state. Therefore, one can retrieve information from a job at a later time or in another client session, so long as the job manager has not been restarted with the -clean option.

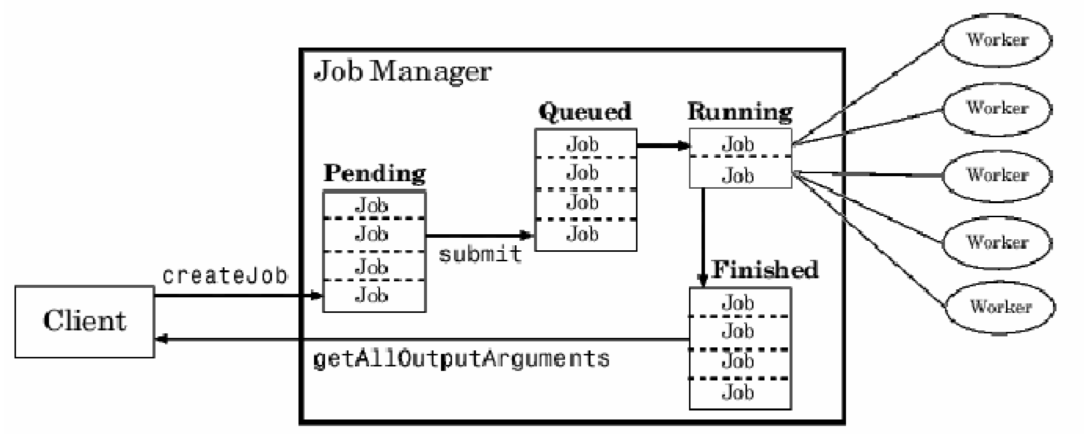

Figure 2: Job processing sequence.

The Distributed Computing Toolbox and Matlab Distributed Computing Engine are supported on Windows, UNIX, and Macintosh platforms. Appendix B gives a basic example from the Matlab help files on how to program a parallel job using this toolbox.

4 MPI TOOLBOX FOR MATLAB (MPITB)

Like MatlabMPI, MPITB [3] is a toolkit that helps to run MPI programs from within Matlab. It is developed by Javier Fernández Baldomero of University of Granada, Computer Science Department, Spain [9]. Unlike MatlabMPI however, MPITB relies on LAM-7 [5] or MPICH [6] in the backend for the parallel operations. Thus, MPITB can be truly called as the Matlab wrapper for MPI. Another interesting aspect of this tool is that it can be used both in Windows [4] and UNIX platform. However, there are extensive documentation for Unix users where as sparse documentation available for Windows users. PC Linux users can use MPITB in order to call MPI library routines from within the Matlab interpreter. A list of available features extracted from MPITB homepage is listed below for reference:

- MEX bindings available for MPI-1.2 calls, except for MPI_Pcontrol, MPI_Op_create, MPI_Op_free, and those related to derived datatypes (MPI_Type_*).
- MEX bindings for some MPI-2.0 calls available: the Info object (MPI_Info_*) and Spawn (MPI_Comm_spawn[_multiple], _Comm_get_parent, _Open/Close_port, _Comm_accept/connect/disconnect, _[Un]Publish/Lookup_name).
- Help files for all interfaced MPI calls available, extracted from the LAM/MPI man pages, which in turn were based on the MPICH man pages
- Source code with error messages/comments in English provided.
- Sample example programs available.

5 CONCLUSION AND FUTURE WORK

In this short survey, we have described three different tools for performing parallel computation using Matlab. These tools are by no means the only tools available for parallel computation in Matlab. A more detailed but an old listing can be found in Baldomero's presentation paper [7]. A somewhat exhaustive list of papers on Matlab MPI toolbox can be found in the MPITB website [8]. We argue that the use of parallel computation in Matlab can be immensely beneficial to researchers in different fields as these provide avenues for fast and efficient computation, especially in modern hardware which has multicore processors. Matlab is immensely powerful for designing complex mathematical models in terms of simple function calls and porting these already existing programs to a parallel design should be a simple exercise.
In future, we plan to take up one specific tool among these three and undertake a detailed analysis and experimentation to evaluate the performance, ease and scalability of the tool.

6 REFERENCES

1. Parallel Programming with MatlabMPI, Dr. Jeremy Kepner, Lincoln Laboratory at Massachusetts Institute of Technology, http://www.ll.mit.edu/MatlabMPI.
2. Matlab Distributed Computing Toolbox Homepage, MathWorks, http://www.mathworks.com/products/distribtb/
3. MPITB homepage, http://atc.ugr.es/javier-bin/mpitb_eng.
4. MPITB Windows version Homepage, http://www.wakun.com/download.aspx
5. LAM/MPI Homepage, http://www.lam-mpi.org/
6. MPICH Homepage, http://www-unix.mcs.anl.gov/mpi/mpich/

7. Message passing under MATLAB, Presentation, Javier Fernández Baldomero, Advanced Simulation Technologies Conference (ASTC), Seattle Washington, April 22-26 2001.
8. MPI Toolbox papers, http://atc.ugr.es/%7Ejavier/investigacion/papers/mpitb_papers.html
9. MathWorks webpage on available MPI toolkits, http://www.mathworks.com/matlabcentral/link_exchange/MATLAB/Parallel_and_Distributed/index.html
10. Javier Fernandez Baldomero, English translated homepage, http://64.233.179.104/translate_c?hl=en&u=http://atc.ugr.es/~javier/&prev=/search%3Fq%3Dhttp://atc.ugr.es/personal.php%26hl%3Den%26lr%3D%26sa%3DG

APPENDIX

A. Sample MatlabMPI Code:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This script implements a basic image convolution
% across multiple processors.
% To run, start Matlab and type:
%
%   eval( MPI_Run('blurimage',2,{}) );
%
% Or, to run a different machine type:
%
%   eval( MPI_Run('blurimage',2,{'machine1' 'machine2'}) );
%
% Output will be piped into to
%
%   MatMPI/blurimage.0.out
%   MatMPI/blurimage.1.out
%   ...
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MatlabMPI
% Dr. Jeremy Kepner
% MIT Lincoln Laboratory
% kepner@ll.mit.edu
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Initialize MPI.
MPI_Init;

% Create communicator.
comm = MPI_COMM_WORLD;

% Modify common directory from default for better performance.
% comm = MatMPI_Comm_dir(comm,'/tmp');
% comm = MatMPI_Comm_dir(comm,'/gigabit/node-a');

% Get size and rank.
comm_size = MPI_Comm_size(comm);
my_rank = MPI_Comm_rank(comm);

% Do a synchronized start.
starter_rank = 0;
delay = 30;  % Seconds
synch_start(comm,starter_rank,delay);

% Set image size (use powers of 2).
% n_image_x = 2.^17;
% n_image_x = 2.^12;
n_image_x = 2.^(10+1)*comm_size;
n_image_y = 2.^10;

% Number of points to put in each sub-image.
n_point = 100;

% Set filter size (use powers of 2).
n_filter_x = 2.^5;
n_filter_y = 2.^5;

% Set the number of times to filter.
```

```
n_trial = 2;

% 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

% Print rank.
disp(['my_rank: ',num2str(my_rank)]);

% Set who is source and who is destination.
left = my_rank - 1;
if (left < 0)
  left = comm_size - 1;
end
right = my_rank + 1;
if (right >= comm_size)
  right = 0;
end


% Create a unique tag id for this message (very important in Matlab MPI!).
tag = 1;

% Create timing matrices.
start_time = zeros(n_trial);
end_time = start_time;

% Get a zero clock.
zero_clock = clock;

% Compute sub_images for each processor.
n_sub_image_x = n_image_x./comm_size;
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];


% Set start time.
start_time = etime(clock,zero_clock);


% 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)
    % Create message tag.
    ltag = 2.*i_trial;
    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

  % Compute convolution.
  work_image = conv2(work_image,kernel,'same');
  % Extract sub_image.
  sub_image = work_image(cboxw(1):cboxw(2),cboxw(3):cboxw(4));
end


% Get end time for the this message.
end_time = etime(clock,zero_clock);

% Print the results.
total_time = end_time - start_time

% Print compute performance.
total_ops;
gigaflops = total_ops / total_time / 1.e9;
disp(['GigaFlops: ',num2str(gigaflops)]);


% Write data to a file.
outfile = ['blurimage.',num2str(my_rank),'.mat'];
%
save(outfile,'start_time','end_time','total_time','kernel','sub_image','work_im
age');
% save(outfile,'start_time','end_time','total_time','kernel');

% Finalize Matlab MPI.
MPI_Finalize;
disp('SUCCESS');

% Don't exist if we are the host.
if (my_rank ~= MatMPI_Host_rank(comm))
  exit;
end
```

B. Matlab Distributed Toolbox Example: Programming a Basic Job

This example illustrates the basic steps in creating and running a job that contains a few simple tasks. Each task performs a sum on an input array.

1. First, we create a job manager. The function "*findResource*" is used to locate a job manager and create the job manager object jm, which represents the job manager in the cluster whose name is *MyJobManager*.
   ```
   jm = findResource('jobmanager','name','MyJobManager');
   ```

2. Next, we create a job. Create job j on the job manager.
   ```
   j = createJob(jm);
   ```

3. Next step is to create tasks. We create three tasks on the job j. Each task evaluates the sum of the array that is passed as an input argument. Here "sum" is the inbuilt matlab function that evaluates the sum along different dimensions of an array.
   ```
   createTask(j, @sum, 1, {[1 1]});
   createTask(j, @sum, 1, {[2 2]});
   createTask(j, @sum, 1, {[3 3]});
   ```

4. Next step is to submit the job to the queue. The job manager moves the job into the queue to be executed when workers are available.
   ```
   submit(j);
   ```

5. Lastly, we retrieve results. We wait for the job to complete, then get the results from all the job's tasks.
   ```
   waitForState(j)
   results = getAllOutputArguments(j)
   results=
           [2]
           [4]
           [6]
   ```