Notes: Parallel MATLAB

Adam Charles*

Last Updated: August 3, 2010

Contents

1 General Idea of Parallelization **2**

2 MATLAB implementation **2**
2.1 Parallel Workspaces and Interactive Pools 3
2.2 Creating and Submitting Jobs 6

3 Using the Neurolab Cluster **8**
3.1 Setup 8
3.2 Configuring MATLAB 8
3.3 Testing the Configuration 10

4 Further information **15**

Appendices **15**

A Useful Functions **15**

*Thanks to Dustin Li and Jeff Bingham fixing the cluster and for the NeuroCluster setup Instructions

1 General Idea of Parallelization

The increase in computer processor speed has reached it's current practical limitations. Faster clock speeds are currently not easily attainable due to a mixture of heat considerations and FET latency in the hardware. Current architectures for High Performance Computing (HPC) now focuses instead on utilizing multiple cores. While processors a few years ago ran single cores at 3GHz+, now the trend is to have processors containing 2 cores at lower speeds (2-3GHz). The goal for computer programs now is to actually utilize these architectures for the benefit of the programmer. The typical linear programming style has to change to focus more on multiple separate tasks being run simultaneously. This involves more intelligent design as the programming aspect now requires two or more schedules to work together. As an example, take a simple program where at each point of a grid a value $f(x, y)$ needs to be simulated. Sequential programming would simulate each $f(x, y)$ one after another, only starting the next simulation once the previous one has finished.

```
for i = 1:N
    for j = 1:M
        F(i,j) = f(x(i), x(j))
    end
end
```

A more intelligent program running on a parallel machine would realize that each $f(x, y)$ can be simulated independently, and thus each core can take one pair of (x, y) and evaluate f, thus saving time. Problems with this level of independence between iterations are known as "embarrassingly parallel" programs. While techniques exist to parallelize less obviously parallel algorithms, the parallelization needs to be figured out separately for each algorithm or class of algorithms as no easy universal rule can be applied. Even with many programs having some degree of parallelization, many more are not significantly parallelizable, to quote an apt proverb: "If you give me 9 women, i can't give you a baby in 1 month." Note however, it would be possible to get nine babies in nine months, averaging out to the desired rate. Thus the first step in parallelizing your program is to correctly distinguish tasks than can be performed independently.

This document will not go into more detail about how to parallelize different programs. Instead this document outlines a tool developed to aid in the programming of parallel tasks once the determination of which tasks are independent is complete. The tool discussed is the MATLAB parallel implementation available in the parallel computing and distributed computing toolboxes. The goal of this document is to familiarize the reader with the options available in these toolboxes for use both on home desktops and high performance computing clusters, as well as to provide enough practical examples to allow the reader to utilize these packages with a reduced learning curve.

2 MATLAB implementation

MATLAB currently implements parallel computing in two different ways. MATLAB utilizes the BASH library to implement linear algebra operations (e.g. a matrix times a matrix) which expands the computations to any additional available cores. This parallelizes many linear algebra operations

automatically. In addition MATLAB includes a parallel computing toolbox toolbox to allow users to take advantage of the multicore architecture found on nearly every desktop, as well as a distributed computing toolbox for computer clusters. While GPGPU computing is available through a third party (Accelereyes' Jacket package: www.accelereyes.com), no official MATLAB GPU toolbox currently exists.

2.1 Parallel Workspaces and Interactive Pools

In order to use parallel computing commands, a MATLAB pool must be opened. MATLAB pools consist of a number of different workers (separate MATLAB instances) linked together by the internal MATLAB scheduler, each performing tasks as assigned. Pools of workers can be handled in one of two ways, either an interactive pool can be opened from an open MATLAB instance, or the pool can be open as part of a job (using `matlabpoolJob`: see Section 2.2).

Opening a MATLAB pool

To start a pool consisting of `mp_size` MATLAB workers in an open instance of MATLAB, the command `matlabpool.m` can be used as in Listing 1 can be used. `matlabpool.m` is a command that opens, closes or checks the size of an existing pool of workers. The code shown first checks to see if a MATLAB pool of the desired size is already open (line 1). If the pool of the desired size is not already open, lines 2-4 check if any pool of workers is open, and closes the pool if it is. The code then opens the desired number of workers in a pool on line 5.

Listing 1: Opening a Matlab Pool

```
1 if matlabpool('size') ~= mp_size
2     if matlabpool('size') == 0
3         matlabpool CLOSE
4     end
5     matlabpool('open', mp_size)
6 end
```

While line 5 can be used independently, opening a pool of workers can take a few seconds, so checking if a pool is already open can optimize automated code. Additionally, errors occur when attempting to close a MATLAB pool when none are open, or to open a MATLAB pool when one is already open. It should be noted that if the only the parallel computing toolbox is available, then a maximum of eight workers can be opened at once. The distributed computing toolbox eliminates this limit, allowing for much larger pools.

Once a MATLAB pool is open, parallel jobs can be run by using the appropriate commands. MATLAB allows varying user control over the specifics of how the parallelization takes place. Depending on the code, the internal MATLAB scheduler will handle much of the interactivity between nodes (data passing, assigning tasks). At one end of the spectrum, the user can specify a very high level command of "perform each of these tasks in parallel" and MATLAB will determine the specifics, such as memory allocation, data transfer etc. At the other end of the spectrum, precise control can be placed over which workers perform which tasks in which order. Even the data passing can be controlled explicitly. Here the basics of the course-control end of the spectrum will be described. More information about finer control can be found in the MATLAB documentation.

Parfor Loops

The easiest code to parallelize is "embarrassingly parallel" for loops. In this type of code, each iteration of a for loop is completely independent of all others, and may be run as its own process. To handle this large class of code (inclusive of parameter sweeps etc.), MATLAB has implemented a `parfor` loop.

`parfor` loops are written using the same syntax as the usual MATLAB `for` loops. Thus instead of typing the syntax in Listing 2, `parfor` is simply substituted as in Listing 3.

Listing 2: `for` loop syntax

```
for ii = ii_start:ii_end
    % Do calculations
end
```

Listing 3: `parfor` loop syntax

```
parfor ii = ii_start:ii_end
    % Do calculations in parallel
end
```

Within the `parfor` loop, each iteration is treated as independent from all others, and the MATLAB built in scheduler portions out each iteration to a worker for computation. The results are then collected and returned appropriately by the scheduler. Since MATLAB handles all of the data copying and organization, a few rules need to be followed to prevent errors:

1. "Slice" all variables. While variable that are uses in their entirety at each iteration can be references with no problem, if different portions of an array are to be used at every iteration, they must be "sliced". Slicing refers to having non-overlapping sets of indexes used in each iteration. Similarly, the outputs must be references with appropriate slicing.

2. Don't use structs. Calling structs within `parfor` loops can cause them to run slower and crash. While this might have been a bug in an earlier version of the toolbox, I personally have not seen this fixed yet.

3. Preallocate all variables. While this should be standard practice for increasing general `for` loop efficiency, `parfor` loops may have iterations finishing non-consecutively, and thus they need to know where to save the data.

With these rules in mind, `parfor` loops are a powerful tool to easily parallelize code with identical, independent tasks, such as parameter sweeping for simulation testing.

Single Program Multiple Data

An alternate method to perform tasks in parallel is to use 'single program multiple data' (`spmd` in MATLAB) programming. `spmd` programming allows a finer control over certain aspects of the process by allowing choice of which worker (referred to as a 'lab') executes the code in what fashion. While `spmd` does allow for different labs to perform different code simultaneously, the namesake stems from the easiest and most simple form of using `spmd`: running the same code on different data

sets. In `spmd`, the same line of code is executed on all labs exactly as it appears. The only difference is that the variables being called in the code may be different on different labs. For example, take the code in Listing 4, which calculates the pseudoinverse of a vector designated `data_vec`.

Listing 4: `spmd` syntax example

```
1 spmd
2     gram_mat = frame_i.'*frame_i;
3     coefs_vec = gram_mat\(frame_i.'*data_vec);
4 end
```

While Listing 4 essentially tells all labs to run lines 2-3, for each lab, a different variable `frame_i` or `data_vec` or both can be saved, thus changing the answer `coefs_vec`. By trying different frames one can, for example, test which frame suits a certain data set best in some metric. To have each lab, for example, test a different frame, each lab must be either initialized correctly. This can be done by referencing each lab directly by using code as in Listing 5.

Listing 5: `spmd` full example

```
1  % Get number of labs
2  nlabs = matlabpool('size');
3
4  % Generate test frames
5  for ii = 1:nlabs
6      frames_to_test{ii} = randn(3,3+ii);
7  end
8
9  % Partition data (frames)
10 for labindex = 1:nlabs
11     frame_i = frames_to_test{labindex};
12     n_samples = 1000;
13 end
14
15 % Test frames
16 spmd
17     data_mat = randn(size(frame_i, 1), n_samples);
18     gram_mat = frame_i.'*frame_i;
19     coefs_mat = gram_mat\(frame_i.'*data_mat);
20     mean_l1 = mean(sum(abs(coefs_mat)));
21 end
22
23 % Collect results
24 for ii = 1:nlabs
25     mean_l1_full(ii) = mean_l1{ii};
26 end
```

Listing 5 shows the extension of 4 into a full test. The second line creates a variable `nlabs` using `matlabpool`, which stores the number of labs (workers) currently open. Lines 5-7 generate the sample frames that will be used in each lab. Note that the sizes no NOT have to match (quite useful!). Lines 10-13 initialize these frames, setting the variable `frame_i` in each lab. This is accomplished by using the indexing variable `labindex`, which cycles through the labs, setting variables for a different lab at every iteration. MATLAB automatically differentiates the `labindex`

index and executes the code appropriately. Note that the number of samples to test (`n_samples`) set here may also differ (as does the size of `frame_i`, but it would be silly to compare different numbers of trials in this situation. When keeping this value the same for each lab, this variable can also be set later, in the `spmd` block, to the same effect. Lines 16 - 21 actually test each frame using the `spmd` command. The testing data, `data_mat` is set randomly for each lab, and the pseudo-inverse is used to find the coefficients of that data in that frame (`coef_mat`. The metric being tested here is the average ℓ_1 norm, $\|\mathbf{x}\|_1 = \sum_i |\mathbf{x}[i]|$,which is associated with sparsity. How MATLAB actually deals with the variables that exist in different labs (the results of the `spmd` block, is to reference them in what is called a 'composite object'. The results of each lab can be referenced by the main script by indexing exactly as a cell array. An example is the last 3 lines (24 - 26) of Listing 5, which collects the results of each lab into one vector, which can be then analyzed by the main script easily.

The mix od using `labindex` and `spmd` mixed with `if-else` statements can be quite a powerful tool in parallelizing more than just identical code, but further development of these concepts is left up to the creativity of the programmer (you).

2.2 Creating and Submitting Jobs

The most general method for performing parallel computation, or computations at all for that matter, on a cluster is to submit jobs to the scheduler. Since MATLAB can interface with many schedulers aside from it's own, the option for opening an interactive MATLAB pool is not always available. Thus submitting a neatly packaged job to the cluster will allow the active scheduler to hand off your program to an isolated set of MATLAB workers as fits with its protocol.

Submitting a job to a cluster can be accomplished in 3 steps, with an optional 4th step of retrieving the data from the completed job. The first step is to create a job. The job has basic descriptions such as the name of the job (necessary for tracking the status and retrieving the results), the type of job (regular, parallel, etc.), and information regarding where to save data within the job if necessary.

There are three basic types of jobs: jobs, parallel jobs and MATLAB pool jobs. Jobs are the basic unit, so to speak, and execute code on one worker for the duration of the job. Parallel jobs utilize multiple cores, but essentially run the same code with different random seeds. This can be useful for simulating the same code with multiple random inputs, for example to test denoising algorithms or adaptive filter schemes on random inputs. MATLAB pool jobs are the most versatile, as they perform any distributed code on a specified number of workers. Thus code using `spmd` or `parfor` loops should be run on MATLAB pool jobs.

To create a job, simply use the appropriate command:

Regular Job	`job_reg = createJob()`
Parallel Job	`job_par = createParallelJob()`
MATLAB Pool Job	`job_pool = createMatlabPoolJob()`

Table 1: Job Creation Functions

The only input the job creation function needs is the scheduler information, which is returned from the command `findResource()`, as shown in Listing 6.

Once the job handle is created, the properties can be edited by either using the `set()` command or by directly setting that . An example of each scheme is given in the example code in Listing

6. For a more complete of job properties to set, please refer to the MATLAB documentation (`doc job` in the command prompt).

The next step is to specify any tasks to be performed within the job. Any number of tasks can be performed in the job. The benefit of stringing multiple tasks within one job is that for most schedulers, a MATLAB worker pool is started for each job, but maintained open until completion of said job. Thus assigning multiple tasks instead of submitting more jobs saves time on the overhead of starting MATLAB. The function to create a task for a job is `createTask()`. The basic inputs into `createTask()` are the job handle for the associated job, the handle to the function (m-file) that the task should run, the number of outputs to return from that function, and the parameters to pass to the function. An example is shown in Listing 6.

Once a Job has been created with all its tasks, submitting the job to the cluster can be accomplished by passing the job handle to the `submit` command: `submit(job_handle)`. The `submit()` function does not need any other inputs.

If the job saves the results internally and does not need to return any results, then the `submit` function is the last function that needs to be used. Otherwise, once the job has been submitted, it needs to be let run to completion for the outputs to be completed. The `waitForState()` functions pauses the script and waits for the data to be ready. `waitForState()` needs as a minimum the job handle to know what job to wait for, and the state for which to wait until. To wait for completion, the corresponding state is `'finished'`. `waitForState()` can also wait for `'running'` or `'queued'` id desired. The third possible option to pass `waitForState()` is a timeout variable. The timeout option gives a maximum time (in seconds) that the script will wait for the job. In this case an output can be defined for `waitForState()`, allowing the script to check if the program completed successfully.

Listing 6 shows an example of creating, running and retrieving the outputs of a MATLAB pool job. In this case, the script `@randtest` takes four inputs, a data matrix, a "basis " (or more technically correct a *frame*), a λ value and a tolerance value. `@randtest` then learns a basis for the data in the data matrix, starting as the input basis, according to the algorithm outlined by Olshausen and Field [1]. There are two outputs, the final basis that was learned, and the time it took to perform the algorithm. Not shown here is the initialization of the basis (essentially `randn()` with appropriate size and normalization of the columns) and the data loading (can be just three dimensional array of natural images available at http://redwood.berkeley.edu/bruno/sparsenet/)

In Listing 6, line 7 finds the scheduler associated with the MATLAB instance. Lines 10-13 set up the job. In this case only the name, and min/max number of workers were specified. Only one task was set up in lines 16-17. The job is let run for a maximum time of one hour and upon successful completion, the outputs are collected at line 34.

Listing 6: Matlab Pool Job Example

```
1
2 %%%%%%%%%%%%%%%%%%%%%%%%%
3 %% Example for creating and submitting a Parallel Job
4
5 %% Get the schedualer
6
7 sched = FindResource();
8
9 % Create the matlabpool job
```

```
job_mpool = createMatlabPoolJob(sched);
job_mpool.Name = 'TestJob';
set(job_mpool,'MinimumNumberOfWorkers',3);
set(job_mpool,'MaximumNumberOfWorkers',3);

% Create tasks - in this case only one
iparam = {data, basis_mat, 0.02, 0.001};
taskLSM(1) = createTask(job_mpool, @randtest, 2, iparam);

% submit the job
submit(job_mpool);

% Wait one hour for the end of the job
timeout_time = 60*60;
return_state = waitForState(job_mpool, 'finished', timeout_time);

% Check if the job finished or timed out
if return_state ~= 1
    % Keep it clean
    destroy(job_mpool)
    % Err out
    error('The job failed!')
else
    % get the outputs
    t_outs = getAllOutputArguments(job_mpool);

    % Keep it clean
    destroy(job_mpool)
end
```

3 Using the Neurolab Cluster

3.1 Setup

The following prerequisites are needed to use MATLAB on the NeuroCluster:

- MATLAB R2009b with the GT license and the Parallel Computing Toolbox installed
- You are hardwired (i.e. via a ethernet cable) to the Neuro network
- You have patience

3.2 Configuring MATLAB

The following steps will configure MATLAB to be able to run jobs on the NeuroCluster:

1. Start your MATLAB client GUI
2. Run the following code (Listing 7 - you may want to put this in your startup.m file):

Listing 7: PCTconfig File

```
disp('Running startup.m');
MyExternalIP=cell2mat(regexp(urlread('http://checkip.dyndns.org'),...
    '(\d+)(\.\d+){3}','match'));
pctconfig('hostname',MyExternalIP);
disp('Startup.m complete.');
```

3. From the Parallel menu choose Manage Configurations.
4. In the Configurations Manager, choose: File-¿New-¿jobmanager
5. Configure the Job manager hostname (brain.neuro.gatech.edu) and Job manager name (NeuroCluster) as shown in Figure 1 (use brain.neuro.gatech.edu instead of vm1.neuro.gatech.edu). Click OK.
6. Choose NeuroCluster as the default configuration in the Configurations Manager (see Figure 2).

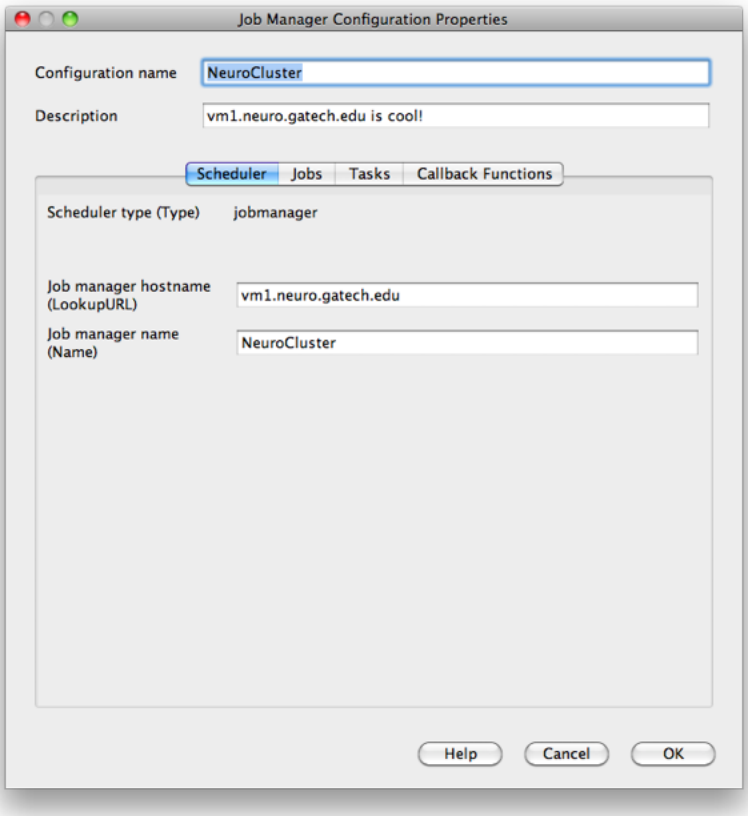

Figure 1: Job Manager Configuration Properties

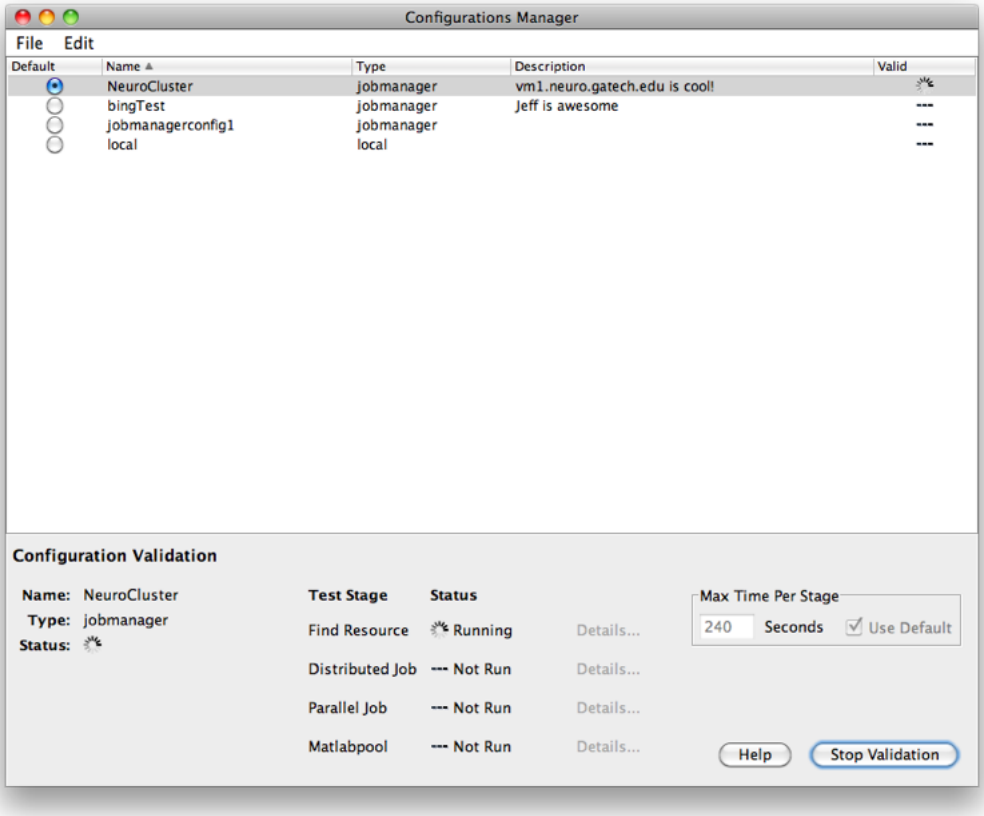

Figure 2: Configurations Manager

3.3 Testing the Configuration

Simple playback

You're almost there. To validate the setup, click the "Start Validation" button in the lower right hand corner of the configurations manager. If everything is setup correctly you should see 4 green checkmarks appear. To view all the labs, run the following script (Listing 8) on the command line:

Listing 8: MATLAB pool test

```
matlabpool 4
spmd
pause(labindex);
disp(labindex);
end
matlabpool close
```

Benchmarking Speedup

Now let's see how the cluster performance scales with additional workers for an embarrassingly parallel job. This code uses `pctdemo_task_blackjack`. Run `dbtype pctdemo_task_blackjack` to see what's happening behind the curtain. In this case, the speedup can be seen in Figure 3 to be fairly linear

Listing 9: `pctdemo_aux_parforbench.m`

```
function S = pctdemo_aux_parforbench(numHands, numPlayers, n)
%PCTDEMO_AUX_PARFORBENCH Use parfor to play blackjack.
%   S = pctdemo_aux_parforbench(numHands, numPlayers, n) plays
%   numHands hands of blackjack numPlayers times, and uses no
%   more than n MATLAB(R) workers for the computations.

%   Copyright 2007-2009 The MathWorks, Inc.
S = zeros(numHands, numPlayers);
parfor (i = 1:numPlayers, n)
    S(:, i) = pctdemo_task_blackjack(numHands, 1);
end
```

Listing 10: `pctdemo_aux_parforbench.m` test script

```
%http://www.mathworks.com/products/parallel-computing/demos.html

%http://www.mathworks.com/products/parallel-computing/demos.html?file=/prod
%ucts/demos/shipping/distcomp/paralleldemo_parfor_bench.html

% modified by Dustin Li, 04/20/2010

poolSize = matlabpool('size');
if poolSize == 0
    error('distcomp:demo:poolClosed', ...
        'This demo needs an open MATLAB pool to run.');
end

% parallelize this task
numHands = 500;
numPlayers = 6;
fprintf('Simulating each player playing %d hands of blackjack.\n', numHands);
t1 = zeros(1, poolSize);
for n = 1:poolSize
    tic;
        pctdemo_aux_parforbench(numHands, n*numPlayers, n);
    t1(n) = toc;
    fprintf('%d workers simulated %d players in %3.2f seconds.\n', ...
            n, n*numPlayers, t1(n));
end

% make sure we aren't be limited by communication overhead
tic;
pctdemo_aux_parforbench(numHands, 10*numPlayers, 1);
clusterToc = toc;
```

```
fprintf('1 cluster workers simulated %d players in %3.2f seconds.\n', ...
    10*numPlayers, clusterToc);

% sequential loop
% tic;
%     parfor z = 0
%         S = zeros(numHands, numPlayers);
%         for i = 1:numPlayers
%             S(:, i) = pctdemo_task_blackjack(numHands, 1);
%         end
%     end
% t1(1) = toc;
% fprintf('Ran in %3.2f seconds using a sequential for-loop.\n', t1(1));

tic;
S = zeros(numHands, 60);
for i = 1:60
    S(:, i) = pctdemo_task_blackjack(numHands, 1);
end
localToc = toc;
fprintf('1 local worker simulated 60 players in %3.2f seconds.\n', localToc);
disp('(Old cluster: 1 worker, 60 players, 11.89 seconds.)');

speedup = (1:poolSize).*t1(1)./t1;
fig = pctdemo_setup_blackjack(1.0);
set(fig, 'Visible', 'on');
ax = axes('parent', fig);
x = plot(ax, 1:poolSize, 1:poolSize, '--', ...
    1:poolSize, speedup, 's', 'MarkerFaceColor', 'b');
t = get(ax, 'XTick');
t(t ~= round(t)) = []; % Remove all non-integer x-axis ticks.
set(ax, 'XTick', t);
legend(x, 'Linear Speedup', 'Measured Speedup', 'Location', 'NorthWest');
xlabel(ax, 'Number of MATLAB workers participating in computations');
ylabel(ax, 'Speedup');

%%
% now plot single thread results
figure(2);
bar([7.37 localToc 11.89]);
set(gca,'XTickLabel',{'New cluster','Local','Old cluster'});
ylabel('Execution time (s)');
title('Single threaded performance');
set(gcf, 'Name', 'Single threaded performance');
```

Listing 11: Example output for test script

```
Simulating each player playing 500 hands of blackjack.
1 workers simulated 6 players in 0.93 seconds.
2 workers simulated 12 players in 0.93 seconds.
3 workers simulated 18 players in 0.95 seconds.
4 workers simulated 24 players in 1.35 seconds.
5 workers simulated 30 players in 1.06 seconds.
6 workers simulated 36 players in 0.97 seconds.
7 workers simulated 42 players in 1.01 seconds.
```

```
 8 workers simulated 48 players in 1.03 seconds.
 9 workers simulated 54 players in 1.05 seconds.
 10 workers simulated 60 players in 1.78 seconds.
 1 cluster workers simulated 60 players in 8.77 seconds.
 1 local worker simulated 60 players in 10.86 seconds.
 (Old cluster: 1 worker, 60 players, 11.89 seconds.)
```

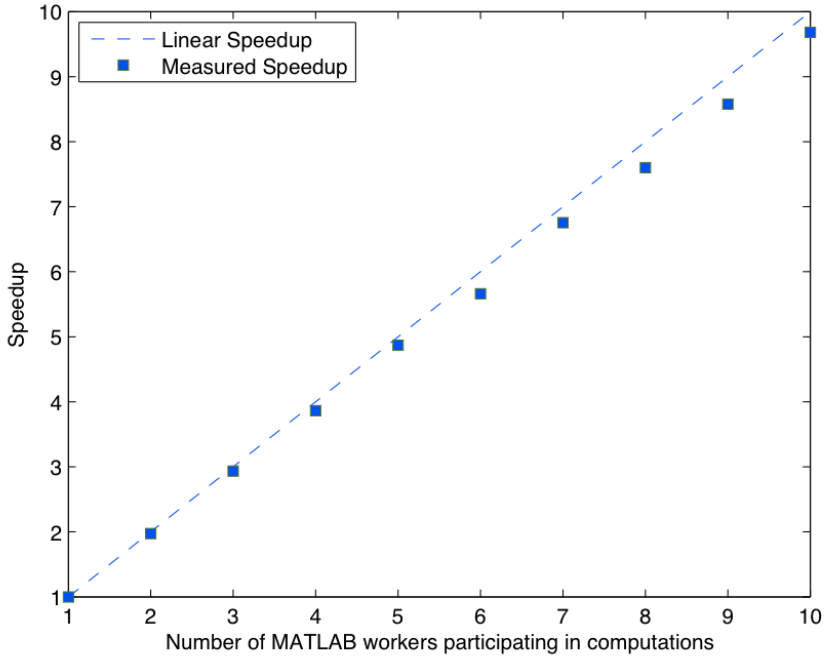

Figure 3: Benchmarking the performance increase

Performance

Compared to the old cluster, sequential performance should also be much higher. The new cluster runs 64-bit Ubuntu Linux on MATLAB r2009b, instead of Mac OS X 10.4 on MATLAB r2008a. This particular code yields a 38% speedup!

Listing 12: Test script on the old cluster

```
>> for n=1
tic;
pctdemo_task_blackjack(5000,1);
toc
end
Elapsed time is 1.984382 seconds.
```

Listing 13: Test script on the new cluster

```
>> for n=1
tic;
pctdemo_task_blackjack(5000,1);
toc
end
Elapsed time is 1.228749 seconds.
```

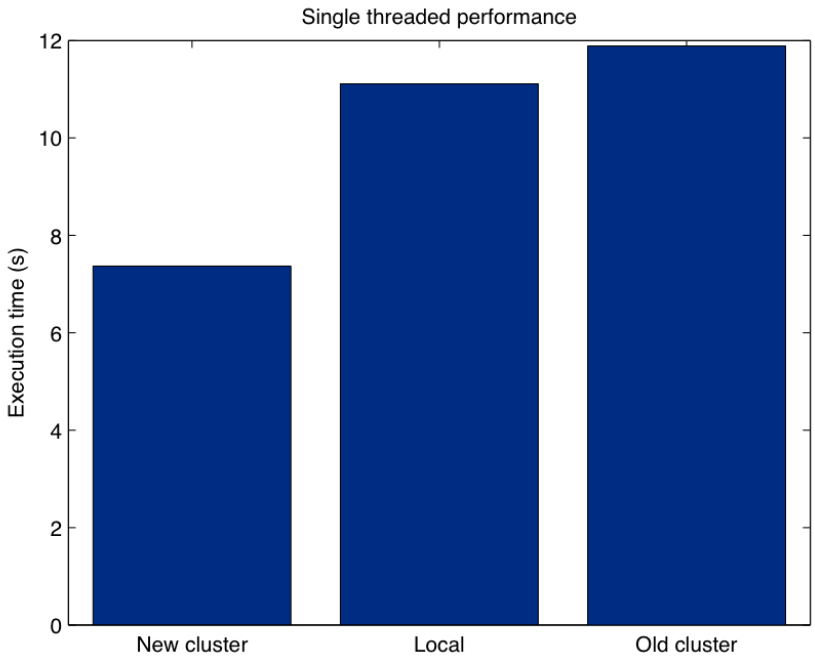

Figure 4: Comparison of speedup with respect to the old MAC OS cluster

Caveats

- At this time, the NeuroCluster is quite finicky with connections outside of the Neuro network (i.e. it's usually not as simple as VPNing into the Neuro net). For external connections, it may be necessary to run this script as a startup file in MATLAB:

Listing 14: Dustin's setup

```
% Dustin's startup.m
% 02/2010
disp('Running startup.m');
MyExternalIP=urlread('http://ip.dustin.li/');
pctconfig('hostname',MyExternalIP);
disp('Startup.m complete.');
```

- There also seems to be a problem if you're behind certain NATs, even if DMZ is set up appropriately. I had a problem with a Netgear RP614 v3 wired to the Neuro network.
- Occasionally, the cluster will get stuck, and not start new jobs even when there are idle nodes. findResource may show that there are finished jobs, pending jobs, but no running jobs. In this case, use admincenter to stop and resume each worker.

4 Further information

See the MATLAB documentation for further information. This is a good place to start:

```
1 doc distcomp
```

Appendices

A Useful Functions

Table 2: Useful MATLAB Functions

Function Name	Brief Description
findResource	Gets detailed information about the scheduler and any jobs/tasks that have been submitted
createJob	Create a job object
createParallelJob	Create a parallel job
createMatlabPoolJob	Create a job that uses a matlab pool
createTask	Create a task within a job
submit	Submit a job to the scheduler
matlabpool	Open, close or check the status of the MATLAB pool

References

[1] B. A. Olshausen and D.A. Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision Research*, 37(23):3311–3325, 1997.