

ACRC How-To: Running MATLAB on BlueCrystal Phase 3

Table of Contents

ACRC How-To: Running MATLAB on BlueCrystal Phase 3.....	1
Introduction.....	1
Available Toolboxes.....	1
Submitting a Batch Job: An Example M-file and Submission Script.....	2
Packaging Your Code using the MATLAB Compiler.....	5
High Performance MATLAB.....	6
Discovering Where Your Code Spends its Time.....	6
Strategies for Accelerating Serial Code.....	7
Using Compiled Code through MEX Files.....	9
A Preamble to Parallelisation: Amdahl's Law.....	9
Multi-threaded Ininsics.....	10
Execute Loop Iterations in Parallel with Parfor.....	11
Summary.....	12

Introduction

This document is not an introduction to MATLAB itself. However, it is an introduction to running MATLAB on a cluster. The intended audience are those who are comfortable running MATLAB but who are new to running their jobs on BlueCrystal.

If you are new to MATLAB, the following web-page provides links to a lot of very good tutorial information: http://www.mathworks.co.uk/academia/student_center/tutorials/launchpad.html

In order to access MATLAB on BlueCrystal phase 3 you will need to load the following module (this command is best added to your .bashrc file):

```
module add apps/matlab-r2013b
```

Available Toolboxes

Since the functionality of MATLAB is influenced by which toolboxes are installed, the following is a handy list of those available of BlueCrystal phase 3, as of the 4th of March 2014:

MATLAB	HDL Verifier	SimElectronics
Simulink	Image Acquisition Toolbox	SimHydraulics
Bioinformatics Toolbox	Image Processing Toolbox	SimMechanics
Communications System Toolbox	Instrument Control Toolbox	SimPowerSystems
Computer Vision System Toolbox	MATLAB Coder	Simscape
Control System Toolbox	MATLAB Compiler	Simulink 3D Animation
Curve Fitting Toolbox	Mapping Toolbox	Simulink Coder
DSP System Toolbox	Model Predictive Control Toolbox	Simulink Control Design
Econometrics Toolbox	Neural Network Toolbox	Simulink Design Optimization
Embedded Coder	Optimization Toolbox	Simulink Verification and Validation
Financial Toolbox	Parallel Computing Toolbox	Stateflow
Fixed-Point Designer	Partial Differential Equation Toolbox	Statistics Toolbox
Fuzzy Logic Toolbox	RF Toolbox	Symbolic Math Toolbox
Global Optimization Toolbox	Robust Control Toolbox	System Identification Toolbox
HDL Coder	Signal Processing Toolbox	Wavelet Toolbox

If you would like to check the current situation, simply type **ver** at the MATLAB prompt. The University of Bristol web page detailing our MATLAB licence is:

<https://www.bris.ac.uk/it-services/locations/zones/zonee/matlab.html>

NB (especially for those used to using the MATLAB cluster in Electrical and Electronic Engineering) **BlueCrystal does not have the MATLAB distributed computing server installed.**

Submitting a Batch Job: An Example M-file and Submission Script

We are all probably most familiar with running MATLAB in interactive mode, perhaps using the GUI interface. This gives us the option of typing commands at the interpreter prompt and also executing sequences of pre-prepared commands contained in an M-file or a MEX-file. **Please do not run computationally intensive MATLAB jobs on BlueCrystal's login nodes.** The system was not designed to work in that way and you will be hindering the work of others if you try.

To run a MATLAB job on a compute cluster, such as BlueCrystal, you will need to create a batch job that can be submitted to the queuing system and subsequently executed when resource on the cluster becomes available. Since this may be when you are not at your computer, the job must be able to run without human intervention. A simple way to achieve this is to encode your task into an M-file.

The quintessential first example for a programming tutorial is, “hello, world.” Let's not buck that trend. Our M-file will be **hello.m**:

```
% A simple M-file
y = 'hello, world'
exit;
```

In order to execute this M-file in batch mode on the cluster, we will need to submit a job to the queuing system. Here is the submission script that I used for the purpose. I named it **matlab_submit**:

```
#!/bin/bash
#! Sample PBS submission script for a MATLAB job

#! Requesting resource (processors and wall-clock time):
#! nodes=1:ppn=1 indicates a single processor.
#! nodes=1:ppn=16 would request a whole node for BCp3.
#! 02:30:00 indicates 02 hours and 30 minutes

#PBS -l nodes=1:ppn=1,walltime=00:10:00

#! change the working directory (default is home directory)
cd $PBS_O_WORKDIR

#! Record some useful job details in the output file
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is $PBS_JOBID
echo This jobs runs on the following nodes:
echo `cat $PBS_NODEFILE | uniq`

#! add the MATLAB module (as per BCp3)
module add apps/matlab-r2013b

#! NB have limited MATLAB to a single thread
options="-nodesktop -noFigureWindows -singleCompThread"

#! Run MATLAB in batch mode
matlab $options -r hello
```

and submitted the job at the command line by typing:

```
qsub matlab_submit
```

Note that I have deliberately included the **-singleCompThread** option, to limit MATLAB to using only one processor. More on this topic in a later section. The **-nodesktop** and **-noFigureWindows** are sensible options when running in a batch environment.

The requested wall-clock time for the job is set to 10 minutes in the above example. Be sure to reset this appropriately if you re-purpose the submission script for your own jobs.

Use **qstat -u <your-username>** to monitor the progress of your job through the queue. A '**Q**' is the 'S' column indicates that the job is queued and is waiting to run. An '**R**' indicates that the job is running, and a '**C**' shows that it has completed.

The output of a job run with the above submission script will collect in a file named, **matlab_submit.o<job-id>**. Any errors will collect in **matlab_submit.e<job-id>**. In my case, the contents of matlab_submit.o208407 are:

```
Running on host node32-021
Time is Tue Mar 4 11:42:40 GMT 2014
Directory is /panfs/panasas01/isys/ggdagw/matlab
PBS job ID is 208407.master.cm.cluster
This jobs runs on the following nodes:
node32-021
Warning: No display specified. You will not be able to display graphics on the screen.
Warning: No window system found. Java option 'MWT' ignored.
```

```
< M A T L A B (R) >
Copyright 1984-2013 The MathWorks, Inc.
R2013b (8.2.0.701) 64-bit (glnxa64)
August 13, 2013
```

```
To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.
```

```
y =
hello, world
```

You can use multiple M-files. For example, if you had a file named **triarea.m**:

```
% function to compute area of a triangle
function a = triarea(b,h)
a = 0.5*(b.* h);
```

You could pass the following **main.m** to MATLAB in the submission script:

```
% calculate some areas of triangles
a1 = triarea(1,5)
a2 = triarea(2,10)
a3 = triarea(3,6)
exit;
```

and, again, the contents of my output file are (without the header this time):

```
a1 =
2.5000

a2 =
10

a3 =
9
```

Packaging Your Code using the MATLAB Compiler

The MATLAB compiler lets you package your code as a standalone application, allowing you to run your code on a machine that does not have MATLAB installed. Packaged code does not require any licenses, so can be useful if you encounter a situation where the toolbox licenses on your system are all used up. Two things to note are that your compiled code will still require the MATLAB compiler runtime (MCR):

<http://www.mathworks.co.uk/products/compiler/mcr/>

and that most, but not all, toolboxes are supported when using the MATLAB compiler. A table detailing support is provided at:

http://www.mathworks.co.uk/products/compiler/supported/compiler_support.html

The MATLAB compiler (**mcc**) accepts quite a complex set of command line arguments, the documentation for which can be hard to find. However, I'll present a fairly general purpose recipe below. Let's compile M-files from the previous example (**main.m**, which contains calls to the function in **triarea.m**):

```
mcc -m -v -w enable -R -singleCompThread main.m
```

Where the options are:

- **-m** generate a standalone application
- **-v** verbose display of compilation steps
- **-w enable** report all warnings
- **-R** pass the following runtime option (**-singleCompThread** in this case)

Other runtime options which you may find useful include **-nodisplay**, **-nojvm**, **-nosplash**, and **-nodesktop**.

A useful site enumerating the list of mcc options is:

<http://www.ling.ohio-state.edu/~kyoon/cowork/splee/HowToConvertM2Exe/mcc-reference.html>

The compilation process produces a wrapper script through which we can launch the standalone application, e.g.:

```
./run_main.sh /cm/shared/apps/Matlab-R2013b
```

The path points to the root directory for the MCR.

It should be noted that the MATLAB compiler does not necessarily improve the performance of your MATLAB jobs. For more information on accelerating your MATLAB code, see the next section.

High Performance MATLAB

It is a mistake to jump straight to parallelisation if you would like faster running MATLAB code. There are several reasons for this. First, it is likely that your serial code can be accelerated, perhaps significantly. If you do not explore this first, you will just be multiplying out inefficiencies. Secondly, parallel code is harder to develop, maintain and deploy, so it should really be your last port of call when searching for additional performance.

Discovering Where Your Code Spends its Time

Your first task when looking for performance gains is to find the portions of your code that consume the majority of the run-time. That way, you can focus your improvements efforts on areas that will yield rewards.

Perhaps the simplest approach to this is to use the MATLAB built-in stopwatch timer. The M-file below uses the **tic** and **toc** functions:

```
% time a portion of code using the
% MATLAB built-in stopwatch timer
tic;
n=1500;
A=rand(n);
B=pinv(A);
toc
exit;
```

After it has run, my output file contains (omitting the header):

```
Elapsed time is 4.326572 seconds.
```

MATLAB also provides a more sophisticated *profiling* tool. Since we will be running our jobs in a batch environment, we cannot use an interactive tool to view the results (since we can't predict when the job will be run). We'll see how we can save the results for viewing at a later date in the example below.

First, let's store the code for a function to convert Cartesian to polar coordinates in an M-file called **cart2plr.m**:

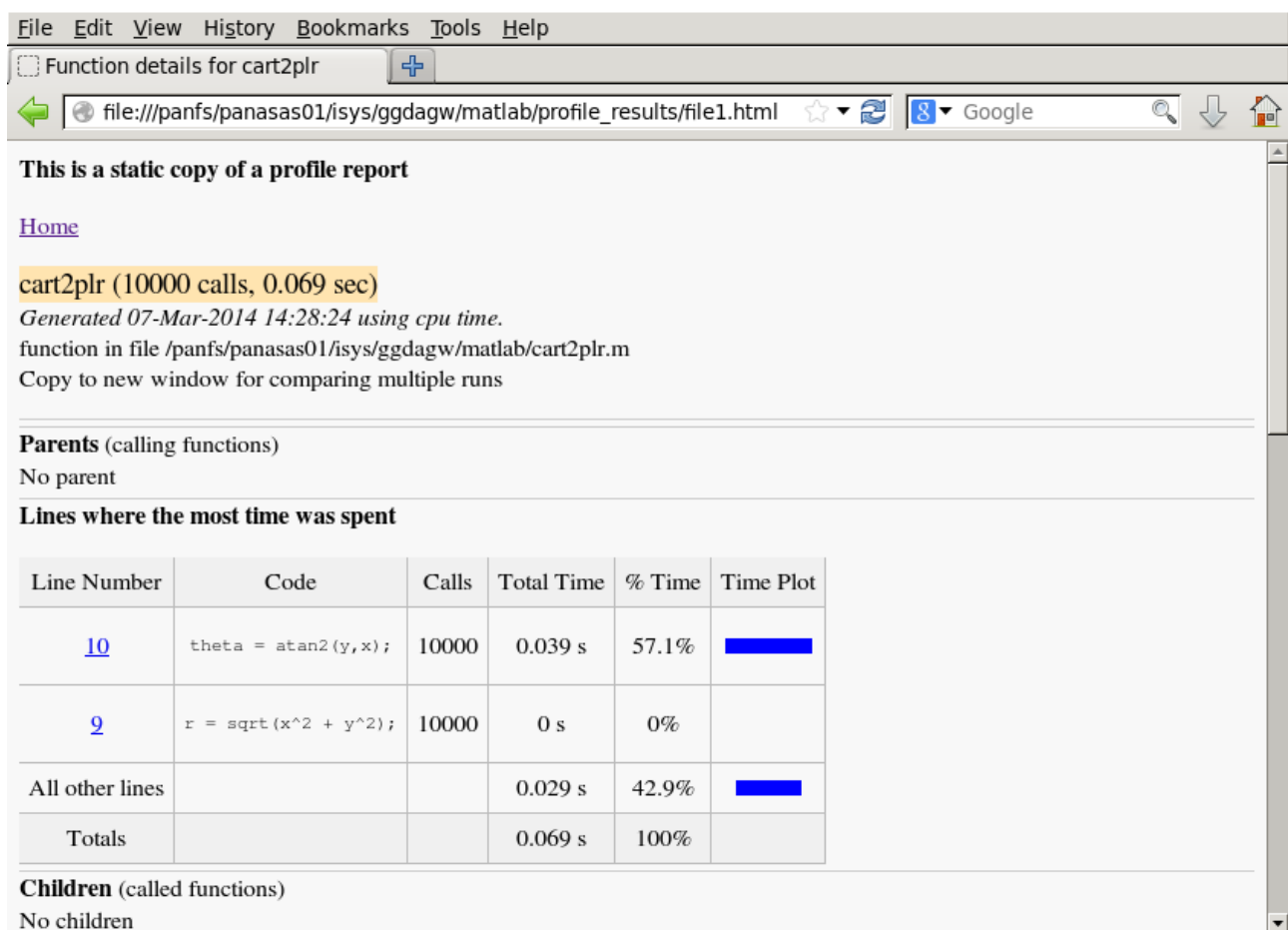
```
function [r,theta] = cart2plr(x,y)
% cart2plr Convert Cartesian coordinates to polar
coordinates
%
% [r,theta] = cart2plr(x,y) computes r and theta with
%
%   r = sqrt(x^2 + y^2);
%   theta = atan2(y,x);

r = sqrt(x^2 + y^2);
theta = atan2(y,x);
```

As before, the control sequence for our job is contained in **main.m**:

```
% Profiling some MATLAB operations
% and saving the results to view later
profile on;
for i=1:10000
    cart2plr(rand(),rand());
end
profile off;
profsave;
exit;
```



The **profsave** function will create a subdirectory called **profile_results** containing a breakdown of where the run-time was spent, in HTML form. When we subsequently view the HTML using a browser:



The screenshot shows a web browser window with the address bar displaying the file path: `file:///panfs/panasas01/isys/ggdagw/matlab/profile_results/file1.html`. The browser's menu bar includes File, Edit, View, History, Bookmarks, Tools, and Help. The page content is titled "Function details for cart2plr" and includes a "Home" link. The main text states: "This is a static copy of a profile report". Below this, it shows the function name and call count: "cart2plr (10000 calls, 0.069 sec)". The generation date and time are "Generated 07-Mar-2014 14:28:24 using cpu time." The function file path is "/panfs/panasas01/isys/ggdagw/matlab/cart2plr.m". A note says "Copy to new window for comparing multiple runs".

Parents (calling functions)
No parent

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
10	<code>theta = atan2(y,x);</code>	10000	0.039 s	57.1%	
9	<code>r = sqrt(x^2 + y^2);</code>	10000	0 s	0%	
All other lines			0.029 s	42.9%	
Totals			0.069 s	100%	

Children (called functions)
No children

we see that, in this case, the trigonometric function took the lion's share of the runtime, and that we would not have gained any performance gain—no matter how hard we tried—if we had attempted to improve the calculation of the root of the sum of the squares.

Strategies for Accelerating Serial Code

It's obviously difficult to prescribe a universal panacea. Undaunted, however, I will outline two strategies that commonly result in performance gains.

Preallocation of vectors

Let's populate an array in two different ways. First, we add to an array of undetermined size in **noprealloc.m**:

```
% incrementally adding values
% to an array
tic;
for i=1:3000,
    for j=1:3000,
        x(i,j)=i+j;
    end
end
toc
```

And secondly, we pre-size the storage and then populate in **prealloc.m**:

```
% adding values to an array
% that was pre-sized
tic;
y=zeros(3000);
for i=1:3000,
    for j=1:3000,
        y(i,j)=i+j;
    end
end
toc
```

The difference in performance is striking:

```
Elapsed time is 14.537684 seconds.
Elapsed time is 0.161670 seconds.
```

Switch from scalar to vector and array operators

In the above example, we used a loop. If you value performance, however, you should avoid loops in your MATLAB code as much as possible.

Given the two arrays—*x* and *y*—from the previous example, let's assume that we now want to multiply all the elements by three. Again, we compare two methods: one which uses a loop and one which does not. First **loop.m**:

```
% multiply by 3 in a loop
for i=1:3000,
    for j=1:3000,
        y(i,j)=y(i,j)*3;
    end
end
toc
```

and secondly, vectorised.m:

```
tic;  
y=y*3;  
toc
```

again, the message is clear:

```
Elapsed time is 0.182269 seconds.  
Elapsed time is 0.009551 seconds.
```

MATLAB contains a number of built-in functions which can save you from writing a loop. Examples include:

- **sum** and **prod**: which compute the sum or product, respectively, of all the elements of vector.
- **cumsum** and **cumprod**: both return a vector and are the cumulative counterparts of '**sum**' and '**prod**'.
- **min** and **max**.
- **any** and **all**: will return true if any or all of the elements of a vector or matrix are true (>0), respectively.
- **find**: returns the indices of a vector that satisfy the given expression. For example, **find(vec > 7)** returns the indices of all elements of vec that are greater than 7.

Using Compiled Code through MEX Files

If you've tried the above strategies and your MATLAB code is still not running fast enough, there is one more step that you should try before thinking about parallel code—using MEX files.

MATLAB allows you to create subroutines in C/C++ or Fortran and call them from MATLAB. If you have identified that you have one or more subroutines in your MATLAB job that are taking the majority of the time, re-writing them and compiling them could give you a useful speed-up. The topic of creating a MEX file is relatively involved, so it is not possible to sensibly provide a small, representative example here. Instead we will link to some very good material on the topic from the Mathworks website:

<http://www.mathworks.co.uk/help/matlab/create-mex-files.html>

If you do not like the prospect of manually re-writing portions of your MATLAB application, you could try out the new MATLAB *coder* tool, which automatically creates C/C++, and optionally MEX files, from your existing MATLAB code. See:

<http://www.mathworks.co.uk/products/matlab-coder/>
for more details.

A Preamble to Parallelisation: Amdahl's Law

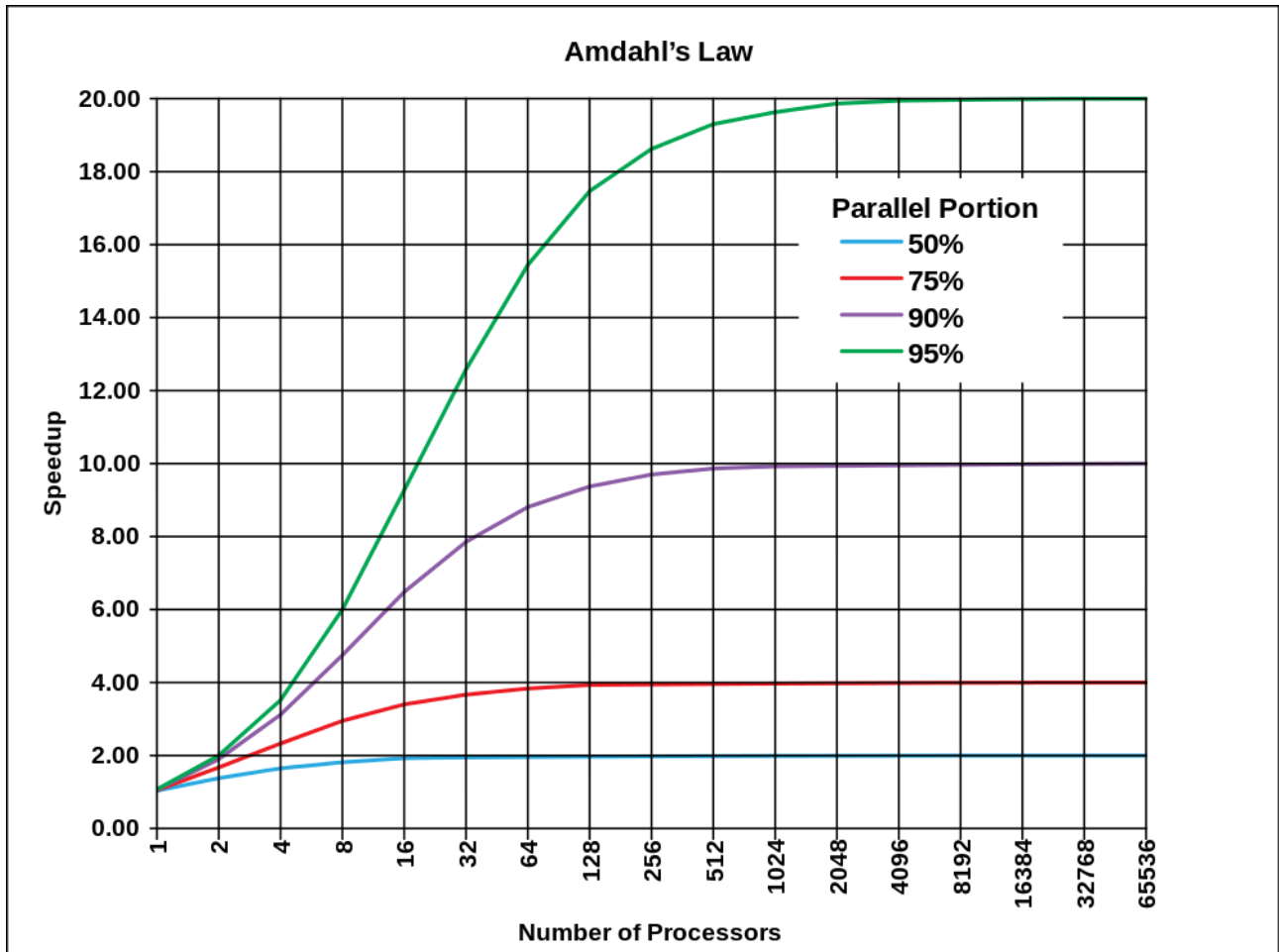
If we want to further accelerate MATLAB code, we should run it in parallel, right? and the more processors that we throw at the problem, the better, yeah?

Not true. Thankfully, Gene Amdahl taught us to think more clearly.

Let's say that we have some code. We estimate that 50% can run in parallel. What's the *best possible* speed-up that we can get on this code? Well the table below tells us:

parallelisable portion of your code	25%	50%	75%	90%	95%
Best possible speed-up	x1.3	x2	x4	x10	x20

Let's also look at how we approach these best possible speed-ups, as we add in more processors (graph courtesy of wikipedia):



The remaining sections describe how you might go about running MATLAB code in parallel. As you read on, however, be sure to ask yourself, “how much of my MATLAB code will run in parallel?” and, “how many processors is it sensible to use for my task?”

Multi-threaded Intrinsics

By far the simplest route to parallelism is to rely on the many MATLAB intrinsic functions that are multi-threaded. The precise list of which functions are multi-threaded changes with each release of MATLAB. However, a flavour is provided at the following URL:

<http://www.mathworks.com/matlabcentral/answers/95958>

It is important to note that all of MATLAB's multi-threaded functions will spread work to all of the processors present in a node. For that reason, you should adjust your submission to request a full node on the cluster. For BlueCrystal phase 3 you should use:

```
#PBS -l nodes=1:ppn=16
```

and to remove `-singleCompThread` from the options which you use to invoke MATLAB:

```
options="-nosplash -nodesktop -noFigureWindows"
```

Execute Loop Iterations in Parallel with Parfor

In this case, we consider the explicit creation of parallel *workers* in order to divide up the work of a loop. We must be careful to note several properties of MATLAB workers:

- They are completely independent and have their own workspaces.
- They can communicate, but cannot access another's workspace.
- There will be (e.g. communication) costs associated with dividing the task among the workers and so a speed-up is **not guaranteed**.

Let's look at our first example of using a parallel loop in MATLAB. Recall our investigation into the use of loops, preallocation of storage and vectorised operations when trying to accelerate some serial MATLAB code. Let's assume that we jumped in too early and assumed that using a parallel loop would *have* to speed things up. Our M-file on this occasion is called **naive-parallel.m**:

```
% adding values to an array  
% that was pre-sized  
% using a parallel for loop  
tic;  
y=zeros(3000);  
parfor i=1:3000,  
    for j=1:3000,  
        y(i,j)=i+j;  
    end  
end  
toc
```

If we call this from a **main.m** which looks like:

```
matlabpool open;  
naive_parallel  
matlabpool close;  
exit;
```

Instead of the elapsed time of ~0.16s for the serial code, we now see:

```
connected to 12 workers.  
Elapsed time is 1.442114 seconds.  
Parallel pool using the 'local' profile is shutting down.
```

Almost an order of magnitude slower thanks to our parallel loop! (And remember that our serial loop ran slower than the vectorised solution.) In order to use **parfor** profitably, we must be savvy. Create loops that:

- have independent iterations, where
- each iteration contains sufficient computational work to offset the cost of setting up and managing the pool of MATLAB workers.

As is often the case, there are many good documents pertaining to this aspect of MATLAB out on the web. For example, a very good introduction to using `parfor` is at:

<http://sc.tamu.edu/shortcourses/SC-matlab/matlab-parallel.pdf>

Also, the file-exchange area of the Mathworks website contains a nice example where using a parallel loop does provide a useful speed-up:

http://www.mathworks.co.uk/matlabcentral/fileexchange/31336-demo-files-for-parallel-computing-with-matlab-on-multicore-desktops-and-gpus-webinar/content/ParallelODE_Example/html/paramSweep.html

When I ran this on BlueCrystal I got the following output:

```
Computing in serial...
Elapsed time is 21.75 seconds.
Computing in parallel...
Starting parallel pool (parpool) using the 'local' profile ... connected to 12 workers.
Elapsed time is 2.65 seconds.
Parallel pool using the 'local' profile is shutting down.
```

The eagle-eyed among you will have noticed that, by default, MATLAB will not open a pool of more than 12 workers.

Summary

The aim of this document was to provide an introduction to running MATLAB jobs in batch-mode on BlueCrystal Phase 3. An example submission script was given including, for example, the **-singleCompThread** flag, along with some very simple example jobs written as M-files.

The topic of *profiling* was introduced, so that you can identify the portions of your MATLAB jobs that are running the slowest (every job has its bottleneck). Examples of the use of the **tic** and **toc** functions and also the MATLAB profiler were given. Common areas where MATLAB jobs can be accelerated were highlighted, including preallocation of storage, vectorisation and the use of MEX files.

Lastly, the topic of parallel MATLAB jobs was covered. Amdahl's law was introduced, so that we can be wise and realistic about what sort of gains parallelism offers. Multi-threaded intrinsics were described along with explicit parallelism using **parfor**—MATLAB's parallel loop construct.