Using the MATLAB Parallel Computing Toolbox on the UB CCR cluster

L. Shawn Matott and Cynthia Cornelius
September, 25, 2014
Outline

• CCR’s resources & how to access them
• Parallel MATLAB resources at CCR
• MATLAB’s Parallel Computing Toolbox:
  – Running on one compute node
  – Running MATLAB interactively
  – parpool and parfor
  – spmd, labindex, and numlabs
  – gplus and gop
• Examples:
  – A distributed “Hello World!” application
  – Monte-Carlo calculation of pi (π)
  – spmd implementation of the Julia set fractal
• Non-interactive MATLAB PCT jobs
• MATLAB Distributed Computing Server (MDCS)
CCR Resources – the numbers

• Compute Nodes
  – Various node configurations (8-, 12-, 16-, and 32-core)
  – Grand total of 8024 processors

• Memory
  – 24 GB to 512 GB RAM per node

• Network Storage
  – 2 GB user home
  – 200 GB to 500 GB group projects space
  – 180 TB high performance scratch space

• Communications
  – 10 Gbps Ethernet
  – 40 Gbps Infiniband
CCR Resources – online help

• Using SLURM:
  www.buffalo.edu/ccr/support/UserGuide/slurm.html

• Cluster information:
  www.buffalo.edu/ccr/support/research_facilities/general_compute.html

• MATLAB on CCR:
  www.buffalo.edu/ccr/support/software-resources/compilers-programming-languages/matlab.html

• Remote Visualization:
  www.buffalo.edu/ccr/support/research_facilities/remote-visualization.html

• MATLAB PCT example (available from rush front-end):
  /util/matlab/example/MyMatlabScript_R2014.m
  /util/matlab/example/slurmMATLAB_R2014
Accessing CCR

- Request account from myccr.ccr.buffalo.edu
- `ssh` to `rush.ccr.buffalo.edu` (front-end)
  - On Windows, use X-Win32
  - On Linux, Mac, use terminal
  - Off-campus, connect to UB VPN first
- CCR nodes use CentOS Linux distribution
  - Must be familiar with Linux command line
- File transfers
  - Use FileZilla or sftp or scp from terminal
- For software with a graphical interface:
  - Use remote visualization (see link on previous slide)
Parallel MATLAB resources at CCR

- **Parallel Computing Toolbox (PCT)**
  - parallel jobs are restricted to run within a single node (like open-mp).
  - unlimited licenses: unlimited # of users and unlimited # jobs
  - Max of 32 cores per job (largest core-count for any CCR node)

- **MATLAB Distributed Computing Server (MDCS)**
  - 1 parallel job can use a maximum of 256 cores
  - jobs run between nodes using MPI (Message Passing Interface)
  - 256-core license.
    - Users share licenses: the sum of all cores being used by all MDCS users (at any given instant) cannot exceed 256.
  - **MDCS is not** covered in today’s workshop
  - See online CCR resources for examples
MATLAB PCT: Running Interactively

• Open a terminal to the rush front-end (ssh -X)
  – Instructions given here:
    www.buffalo.edu/CCR/support/UserGuide/login.html

• Within your home directory, create a new directory for this tutorial.
  mkdir ~/workshop
  cd ~/workshop

• Copy the example scripts (in /util/matlab-scripts) into your new directory
  cp /util/matlab-scripts/* ~/workshop

• When using MATLAB interactively, it is important to request your own compute node to do the computations (i.e. do not run on the front-end).

• Request an (example) interactive job by typing the following:
  fisbatch --partition=debug --nodes=1 --ntasks-per-node=4
MATLAB PCT: Running Interactively

• Once on compute node, load the matlab/R2014a module
  `module load matlab/R2014a`

• Now start MATLAB in “nodesktop” mode
  `matlab -nodisplay`
  – “nodisplay”: disable ALL graphics, including editor and plots
    • Use if MATLAB takes a long time to load and/or you are running over a wireless network (or off-campus)
  – “nodesktop”: don’t load the MATLAB GUI, but still permit access to graphical utilities such as the editor and plots.

• Once presented with a “>>” prompt you will be running an interactive session on the requested cores!
MATLAB PCT: Running Interactively

• It is always a good idea to monitor the performance of your MATLAB code

• `slurmjobvis` is a graphical job monitoring tool
  – Monitors *all cores* on a node, even if not assigned to your job...

• Open a new rush terminal connection and type:
  `squeue -u your_username`

• Note the ID assigned to your job. Now type:
  `slurmjobvis your_job_id &`

• `sjeff` is a non-graphical tool for examining job efficiency
  – Examines only the cores assigned to your job...
  – Combine with “`watch`” to monitor performance over time
  `export STUBL_SJEFF_PCPU=top` (snapshot style of CPU usage)
  `watch sjeff your_job_id` (type CTRL+c to stop monitoring)
An example slurmjobvis

```bash
[lsmatott@rush]$ slurmjobvis 2747609 &
[1] 46729
[lsmatott@rush]$ k16n25s01 has been allocated CPUs 0-11
k16n25s02 has been allocated CPUs 0-11
k16n26s01 has been allocated CPUs 0-11
k16n26s02 has been allocated CPUs 0-11
k16n27s01 has been allocated CPUs 0-11
k16n27s02 has been allocated CPUs 0-11
k16n28s01 has been allocated CPUs 0-11
k16n28s02 has been allocated CPUs 0-11
Cluster: ub-hpc
User: as338
Job: 2747609
Adding node: k16n25s01
Adding node: k16n25s02
Adding node: k16n26s01
Adding node: k16n26s02
Adding node: k16n27s01
Adding node: k16n27s02
Adding node: k16n28s01
Adding node: k16n28s02
```
An example sjeff

```
[lsmatott@rush]$ export STUBL_SJEFF_PCPU=top
[lsmatott@rush]$ watch sjeff 2747609

[lsmatott@rush]$

(CTRL+c)
```

```
Every 2.0s: sjeff 2747609 Wed Sep 10 15:15:07 2014

<table>
<thead>
<tr>
<th>Job_ID</th>
<th>Username</th>
<th>Efficiency</th>
<th>Number_of_CPUs_In_USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2747609</td>
<td>as338</td>
<td>= 99.69 %</td>
<td>(95.70 of 96)</td>
</tr>
</tbody>
</table>
```
MATLAB PCT: parallelism w/out using PCT commands

- A little known fact: common functions such as \texttt{exp}, \texttt{sqrt}, and others may automatically run over multiple cores \textbf{with no change to your code}!

- As you might imagine, this can cause some confusion when comparing the performance of your parallelized code with a (seemingly) non-parallelized code.


- \textbf{Try}: Apply various MATLAB functions to a large matrix. Look in your \texttt{slurmjobvis} (or \texttt{sjeff}) window to determine if the work is being split across multiple cores.
MATLAB PCT: parallelism *without* using PCT commands

- Create a random matrix of size 25000 by 25000.
  ```matlab
g >> A = rand(25000);
```
- Compute the square root of each matrix element.
  ```matlab
g >> B = sqrt(A);
```
- Output from `slurmjobvis`

![Output from slurmjobvis](during matrix creation)

![Output from slurmjobvis](during sqrt operation)
MATLAB PCT: parallelism \textit{w/out} using PCT commands

- Output from `watch sjeff`

```
Every 2.0s: sjeff 2754699
Job_ID  Username  Efficiency  Number_of_CPUs_In_Use
[0;31m2754699  lsmatott  = 24.60 % (0.98 of 4)
```
(during matrix creation)

```
Every 2.0s: sjeff 2754699
Job_ID  Username  Efficiency  Number_of_CPUs_In_Use
2754699  lsmatott  = 54.00 % (2.16 of 4)
```
(during sqrt operation)
MATLAB PCT: parpool (FKA matlabpool)

• Although we requested 4 cores, MATLAB is only expected to execute PCT commands over multiple cores if we tell it to open a corresponding `parpool` of 4 cores (or less, if we desire). In order to open a new pool, any previous pool must first be closed. The basic structure is:

  ```matlab
  >> parpool('local', 4)
  % ( code that may or may not use 4 cpus )
  >> delete(gcp('nocreate'))
  ```

• **Try:** Open a `parpool` on \( x \) number of cores (with \( x \leq 4 \)) and run the following code. How many cores become active in `slurmjobvis`? Close the pool when the `parfor` loop completes.

  ```matlab
  >> clear F
  >> parfor i=1:250000000
     F(i) = sqrt(i);
   end
  ```

  (note: keep typing instructions even though no prompt is given, they will be processed once the "end" statement is entered to finalize the `parfor` loop)
MATLAB PCT: parpool

- Output from “watch sjeff” and “slurmjobvis”
MATLAB PCT: parfor

• Each year, the list of MATLAB functions and constructs that can make use of multiple cores increases. The parfor (parallel for-loop) construct is one such example.

• A parfor loop arbitrarily splits up work between the # of cores specified by the pool. The work on each core is done independently for the entire duration of the loop, and so the elements of the loop must be independent (i.e. the loop must be non-iterative).

• The general parfor structure is:

```
>> parpool('local', ncores)     (request cores)
>> . . .                       (other operations, possibly parallelized)
>> parfor i=start:end          (enter parallelized loop)
     . . .                    (independent loop operations)
     end                      (exit parallelized loop)
>> . . .                       (other operations, possibly parallelized)
>> delete(gcp('nocreate'))     (release cores)
```
MATLAB PCT: spmd

- Code within a `spmd` (single program, multiple data) construct will run on all cores in a pool.
- Each core will simultaneously run an exact replica of the code (i.e. single program). `spmd` uses the MPI (Message Passing Interface) protocol.
- The data produced from each core may be different (i.e. multiple data), depending how one uses the core-identifying variable `labindex` (rank) as well as other `spmd` commands, such as `numlabs` (total cores in a pool).
- **Try:** Do the following for however many cores you would like:

```matlab
>> parpool('local', 4)  % (request 4 cores)
>>   spmd  % (enter spmd region)
   pid=int2str(labindex);  % (identify core)
   np=int2str(numlabs);  % (total cores)
   [ 'Hi from lab ',pid,' of ',np]  % (say hello!)
   end  % (exit spmd region)
>> delete(gcp('nocreate'))  % (release cores)
```
Here’s a for loop parallelized using `spmd`. Each lab starts at it’s `labindex` and increments by `numlabs`, ensuring each entry is processed just once. Vector updates are made to different copies of `V` (one copy is made for each core). So we must gather the results using the `gplus` operation.

```matlab
>> diary on;
>> V=zeros(1,100);
>> parpool('local', 4);
>> spmd
    for i=labindex:numlabs:100
        V(i)=sqrt(i);
    end
    S = gplus(V, 1);
    if(labindex == 1)
        display(V);  
        display(S);
    end
end
>> delete(gcp('nocreate'));
```

`gplus(V, 1)` collects results, store on worker 1

When done, exit MATLAB and check the diary file – note the differences between `V` and `S`. 

(log command outputs to file)
MATLAB PCT: spmd

- Some of the more useful commands that can be used within an spmd statement:
  - `gplus` - global addition across cores.
  - `gop` - general global operation to be done across cores.
  - `labSend` and `labReceive` pass information between cores.
  - `labBarrier` synchronizes cores at a specific point in the code.
  - `labBroadcast` sends data to all cores.

`>> help [command]` (usage information for a command)
The code piMC.m randomly chooses $N$ points in a unit square and then finds the fraction of these points that also lie within an inscribed circle (i.e. with radius of $r=1/2$).

As $N$ increases, the fraction of points located within the circle will approach a limiting value:

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}$$

$$\pi \approx 4 \times \frac{N_{\text{Yellow}}}{N_{\text{Yellow}} + N_{\text{Black}}}$$
spmd example: Monte-Carlo calculation of pi (π)

- The piMC function approximates pi using \(N \times \text{numlabs}\) randomly sampled points within the unit square.
- The following is the contents of \texttt{piMC.m} (\textit{no need to type in at prompt}):

```matlab
function [time]=piMC(N)
    % Monte-Carlo pi calculation
    tic
    n=0;
    for j=1:N
        x=rand; y=rand;
        if (x^2+y^2)<=1, n=n+1; end
    end
    mypi=4*gplus(n)/(numlabs*N); %global sum of pts inside
    error=gop(@max, abs(pi-mypi)); %global max (redundant)
    if (labindex == 1)
        fprintf('pi is about %E\n', mypi);
        fprintf('error is %E\n', error);
    end
    time=toc;
```
spmd example: Monte-Carlo calculation of pi (π)

- Run the piMC example on an interactive compute node:
  ```matlab
  >> diary on;
  >> parpool('local', 4);
  >>   spmd
       time=piMC(250000000);
   end
  >>   display(time{1});  % time{1} → show timing for worker 1
  >> delete(gcp('nocreate'));
  ```

- A simple scale-up study (each experiment samples same number of points):
  ```matlab
  >> for ncpus=1:4
     parpool('local', ncpus);
     spmd
       time=piMC(200000000/ncpus);
     end
     fprintf('ncpus = %d, time = %d\n', ncpus, time{1});
     delete(gcp('nocreate'));
  end
  ```
spmd example: Monte-Carlo calculation of \( \pi \) (**\( \pi \)**)

- Scale-Up Results

```
[lsmatott@d07n33s01 workshop]$ cat diary | grep '^ncpus'
ncpus = 1, time = 2.865519e+01
ncpus = 2, time = 1.465005e+01
ncpus = 3, time = 9.633041e+00
ncpus = 4, time = 7.381599e+00
```

<table>
<thead>
<tr>
<th>ncpus</th>
<th>time (s)</th>
<th>speedup</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28.66</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>14.65</td>
<td>1.96</td>
<td>0.98</td>
</tr>
<tr>
<td>3</td>
<td>9.63</td>
<td>2.97</td>
<td>0.99</td>
</tr>
<tr>
<td>4</td>
<td>7.38</td>
<td>3.88</td>
<td>0.97</td>
</tr>
</tbody>
</table>

\[ \text{speedup}_{ncpus} = \frac{\text{time}_{1cpu}}{\text{time}_{ncpus}} \]

Efficiency = \( \frac{\text{speedup}_{ncpus}}{ncpus} \)
spmd example: Julia set fractal

- The code in Julia.m generates a fractal which is written to a bitmap image (Julia.bmp).
- The main loop of Julia.m is non-iterative, and so the work can be split into independent tasks to run over multiple cores.

Pick a point \( Z_p \) in the complex plane:

\[
Z_0 = Z_p
\]

Iterate according to:

\[
Z_{n+1} = Z_n^2 + C
\]

If \( Z_{n+1} \) diverges, discard point
Else, plot point

(a typical Julia set fractal for a given value of \( C \))
spmd example: Julia set fractal

- The aggregate image is written to Julia.bmp. The image on each core is written to 1.bmp, 2.bmp, ....

```matlab
S = gplus(bmp, 1); % collect results
if(labindex == 1)
    imwrite(S, 'Julia.bmp'); % write image to file
end
imwrite(bmp, [num2str(labindex), '.bmp']);
```

- Try: Run the code within an spmd statement inside a parpool. Use these inputs to the code:

```
>>Julia(288, -0.8, 0.156)
```

- After running the code, display the aggregate image by typing:

```
>>!display Julia.bmp
```

- Look also at the images from each core. Do they look as expected?
MATLAB PCT: submitting a job to the queue

• Use `runjob.m` as a `parpool/spmd` wrapper:

```matlab
%% MATLAB PCT wrapper, use with slurmMATLAB
parpool('local', 12);
spmd
    [time]=piMC(100); % call your code here
end
delete(gcp('nocreate'));
```
MATLAB PCT: submitting a job to the queue

- Edit `slurmMATLAB` SBATCH directives to request desired wall time, partition, and cores (i.e. tasks-per-node).

```bash
#!/bin/bash
#SBATCH --time=1:00:00
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --tasks-per-node=8
#SBATCH --mail-user=your_user_name@buffalo.edu
#SBATCH --mail-type=END
#SBATCH --job-name=MATLAB_R2014
#SBATCH --output=matlab.out
#SBATCH --error=matlab.err
#SBATCH --partition=debug
```
MATLAB PCT: submitting a job to the queue

- Submit job using `sbatch` command from bash prompt:
  
  $ sbatch slurmmATLAB

- Check status in queue using `squeue`:
  
  $ squeue -u your_username

- When the job is running, monitor with `sjeff` or `slurmjobvis`.
- Use `stimes` to get an estimate of when the job will run.
- The `slurmhelp` command lists many other useful SLURM commands and utilities.
Thanks for attending!
(e-mail ccr-help@buffalo.edu for help)