INTRODUCTION TO
MATLAB PARALLEL COMPUTING TOOLBOX

Kadin Tseng
Boston University
Scientific Computing and Visualization
What is the PCT?

- The Parallel Computing Toolbox is a MATLAB tool box.

- This tool box provides parallel utility functions to enable users to run MATLAB operations or procedures in parallel to speed up processing time.

- Tutorial based on R2013a.
Where To Run The PCT ?

- Run on a desktop or laptop
  - MATLAB must be installed on local machine
  - At present, SCC runs MATLAB R2013a, supporting 12 cores.
  - Starting with R2014a, supports 512 cores (theoretically)
  - Must have multi-core to achieve speedup
  - Requires **BU userid** (to access MATLAB, PCT licenses)

- Run on a SCC Cluster node (as a multi-cored desktop)
  - Requires **RCS userid**

- Run on multiple SCC nodes (for up to 64 workers)
  - Requires **RCS userid**
Types of Parallel Jobs?

There are two types of parallel applications.

- Distributed Jobs – task parallel
- Parallel Jobs – data parallel
Distributed Jobs

This type of parallel processing is classified as:

Multiple tasks running independently on multiple workers with no information passed among them. On SCC, a distributed job is a series of single-processor batch jobs. This is also known as task-parallel, or “embarrassingly parallel”, jobs.

Examples of distributed jobs: Monte Carlo simulations, image processing

Parallel utility function: dfeval (recommends to use RCS script)
Parallel Jobs

A parallel job is:

A single task running concurrently on multiple workers that may communicate with each other. On SCC, this results in one batch job with multiple processors. This is also known as a data-parallel job.

Examples of a parallel job include many linear algebra applications: matrix multiply; linear algebraic system of equations solvers; Eigen solvers. Some may run efficiently in parallel and others may not. It depends on the underlying algorithms and operations. This also includes jobs that mix serial and parallel processing.

Parallel utility functions: spmd, drange, parfor, . . .
How much work to parallelize my code?

- Amount of effort depends on code and parallel paradigm used.
- Many MATLAB functions you are familiar with are overloaded to handle data operations based on the variables’ data type – arrays will likely be operated on as vectors while distributed arrays are operated on with multiple workers.
Run Distributed Jobs — *dfeval*

Example: Run a *dfeval* job interactively
Computes 1x4, 3x2 random arrays on 2 cores

```matlab
>> y = dfeval(@rand, {1 3}, {4 2}, 'Configuration', 'local')
```

Submitting task 1
Job output will be written to: /usr1/scv/kadin/Job6_Task1.out
QSUB output: Your job 211908 ("Job6.1") has been submitted

Submitting task 2
Job output will be written to: /usr1/scv/kadin/Job6_Task2.out
QSUB output: Your job 211909 ("Job6.2") has been submitted

```matlab
y =
    [1x4 double]
    [3x2 double]
```

*Job ran in batch or background, output returns to client workspace.*
Run Distributed Jobs – RCS script

For task-parallel applications on the SCC, we strongly recommend the use of an RCS script instead of `dfeval`. This script does not use the PCT. For details, see


*If your application fits the description of a distributed job, you don’t need to know any more beyond this point.*
Run (Data) Parallel Jobs

Two ways to run parallel jobs: `pmode` or `matlabpool`. Use them to turn on/off parallelism and allocate/deallocate resources.

`pmode` is a special mode of application; useful for learning the PCT and parallel program prototyping interactively.

`matlabpool` is the general mode of application; it can be used for interactive and batch processing.
pmode

>> pmode start local 4   % 4 workers on local machine

Above is a MATLAB window on SCC login (or compute) node. A separate Parallel Command Window is spawned. (Similar on Windows)
**pmode**

```
>> pmode start local 4  % use 4 workers in SPMD (parallel) mode
```

PCT terminologies:
- worker = processor
- `labindex`:
  processor number
- `numlabs`:
  Number of processors

- Any command issued at the “P>>” prompt is executed on all workers. Enter “labindex” to query for workers’ ID.
- Use *if* conditional with `labindex` to issue instructions to specific workers, like this: `if labindex==1, numlabs, end`
• Replicate array
  \[
  \text{P} \gg A = \text{magic}(3); \quad \% \text{A is replicated on every worker}
  \]

• Variant array
  \[
  \text{P} \gg A = \text{magic}(3) + \text{labindex} - 1; \quad \% \text{labindex}=1,2,3,4
  \]

\[
\begin{array}{cccc}
\text{LAB 1} & \text{LAB 2} & \text{LAB 3} & \text{LAB 4} \\
|8 \ 1 \ 6|9 \ 2 \ 7|10 \ 3 \ 8|11 \ 4 \ 9| \\
|3 \ 5 \ 7|4 \ 6 \ 8|5 \ 7 \ 9|6 \ 8 \ 10| \\
|4 \ 9 \ 2|5 \ 10 \ 3|6 \ 11 \ 4|7 \ 12 \ 5|
\end{array}
\]

• Private array
  \[
  \text{P} \gg \text{if labindex}==2, A = \text{magic}(3) + \text{labindex} - 1; \text{ end}
  \]

\[
\begin{array}{cccc}
\text{LAB 1} & \text{LAB 2} & \text{LAB 3} & \text{LAB 4} \\
| \quad | \quad | \quad | \\
|9 \ 2 \ 7| \\
\text{undefined} \ |4 \ 6 \ 8|\text{undefined}|\text{undefined}|
\end{array}
\]

\[
\begin{array}{c}
|5 \ 10 \ 3|
\end{array}
\]
Switch from \textit{pmode} to \textit{matlabpool}

If you are running MATLAB \textit{pmode}, exit it.
P>> exit

\begin{itemize}
  \item \textit{MATLAB allows only one parallel environment at a time}.
  \item \textit{pmode} may be started with the keyword \textit{open} or \textit{start}.
  \item \textit{matlabpool} can only be started with the keyword \textit{open}.
\end{itemize}

You can also close \textit{pmode} from the MATLAB window:
>> pmode close

Now, open \textit{matlabpool}
>> matlabpool open local 4 \% omitting 4 default to configured max

\textit{Starting from R2014a, matlabpool will not be needed.}
\texttt{matlabpool} is the essential tool for requesting processors for MATLAB parallel computing. Usage:
\begin{verbatim}
>> matlabpool open \textit{Config Nworkers} \% italic optional item
>> \% Use default value if optional item is omitted
>> \% Config is \textit{local} or \textit{SGE} on SCC
>> \% Nworkers is no. of cores
>> \%
>> \% perform parallel tasks inside matlabpool region
>> \% . . .
>> matlabpool close \% ends \texttt{matlabpool}
\end{verbatim}

Parallel methods within \texttt{matlabpool} region:
\begin{itemize}
  \item \textit{parfor} – parallel for-loop; more frequently used
  \begin{itemize}
    \item Fine-grain \textit{parfor} is thread-based parallelism
  \end{itemize}
  \item \textit{spmd} – single program multiple data parallel region
  \begin{itemize}
    \item Coarse-grain \textit{spmd} is MPI-based parallelism
    \item \textit{pmode} is built on \textit{spmd}
  \end{itemize}
\end{itemize}
Parallel Methods — \textit{parfor}

It is a for-loop that executes on multiple workers in parallel
\begin{itemize}
    \item \textit{parfor} automatically distributes workload via loop indexing
        \begin{itemize}
            \item \textit{parfor} decides which worker does what iterations
            \item \textit{parfor} decides how iterations are divvied up by workers
        \end{itemize}
    \item \textit{parfor} starts and ends on Client (base workspace)
    \item Number of workers requested via \textit{matlabpool}
    \item Without \textit{matlabpool} activation, \textit{parfor} reverts to \textit{for}
    \item Computation for all loop indices must be independent
    \item Results must be independent of loop index execution order
    \item More rules governing \textit{parfor} operations in the PCT doc.
\end{itemize}
Parallel Methods — parfor (2)

```matlab
matlabpool open 2  % my default config is ‘local’
s = 0;       % initialize final sum of sine series
parfor i=1:10
    x(i) = sin(2*pi*i/10); % i=1:5 by 1st worker; i=6:10 by 2nd
    s = s + i;            % not all reduction operations allowed
end
matlabpool close

does not alter the specified variable.
```

**parfor** can do reductions provided that the operator satisfies
associative rule: \( x + (y + z) = (x + y) + z \)  % “*” works too
Subtract (−) and divide (/) operators fail rule — **indeterministic**

```matlab
s=1000; parfor i=1:500, s=s*i; end % correct answer guaranteed
s=1000; parfor i=1:500, s=s/i; end % answer not guaranteed
```

% Example of loop dependency
% matlab editor would warn you this parfor usage is wrong
a = zeros(1,100);
parfor i = 2:100
    a(i) = myfct(a(i-1));
end
Integration Example

• An integration of the cosine function between 0 and $\pi/2$
• Integral $\approx$ sum of areas of rectangles (height $x$ width)
• Several parallel methods will be demonstrated.

$$\int_a^b \cos(x)dx = \sum_{i=1}^{p} \sum_{j=1}^{n} \int_{a_{ij}}^{a_{ij} + h} \cos(x)dx \approx \sum_{i=1}^{p} \left[ \sum_{j=1}^{n} \cos(a_{ij} + \frac{h}{2})h \right]$$

- $a = 0; \ b = pi/2; \ % \ range$
- $m = 8; \ % \ # \ of \ increments$
- $h = (b-a)/m; \ % \ increment$
- $p = numlabs;$
- $n = m/p; \ % \ inc. / \ worker$
- $ai = a + (i-1)*n*h;$
- $aij = ai + (j-1)*h;$
% serial integration (with for-loop)
tic  
    m = 10000;
a = 0;          % lower limit of integration
b = pi/2;       % upper limit of integration
dx = (b - a)/m; % increment length
intSerial = 0;  % initialize intSerial
for i=1:m
    x = a+(i-0.5)*dx;  % mid-point of increment i
    intSerial = intSerial + cos(x)*dx;
end  
toc

\[ dx \rightarrow \]

\[ x(1) = a + \frac{dx}{2} \]  
\[ x(m) = b - \frac{dx}{2} \]
Integration Example — *parfor*

**Parallel integration with *parfor***

```matlab
matlabpool open 4
 tic
 m = 10000;
a = 0;
b = pi/2;
dx = (b - a)/m;  % increment length
intParfor = 0;
parfor i=1:m
    intParfor = intParfor + cos(a+(i-0.5)*dx)*dx;
end
toc
matlabpool close
```
Parallel Methods — *spmd*

In *spmd* parallel environment:

- A MATLAB client process manages multiple workers
- Each worker, or lab, has its own independent memory
- Each worker runs the same program (Single Program) but on its own data (hence Multiple Data)
- Can perform point-to-point communications between workers (*labsend* and *labreceive*)
- Collective communications among workers (*gather*, *gop*)
- Workers can access Client data by simply referencing it
- Client can create/modify/access data on workers
- Two very essential functions within *spmd* region:
  - *numlabs* – returns number of labs available for job
  - *labindex* – returns current lab ID; this is essential to control what to do on specific labs

```
>> spmd, labindex, end  % prints local worker ID (1:numlabs)
```

- *spmd* region is enclosed between *spmd* & *end*
- Arrays created with SPMD are persistent (across regions)
Integration Example — *spmd*

\[
n = 500; \quad \% \text{number of integral increments per lab}
\]
\[
a = 0; \; b = \pi/2; \quad \% \text{lower and upper limits of integration}
\]

\[
\textit{spmd}
\]
\[
deltax = (b - a)/\text{numlabs}; \quad \% \text{integral range length per lab}
\]
\[
ai = a + (\text{labindex} - 1)*\text{deltax}; \quad \% \text{local integral limits, } [ai, bi]
\]
\[
bi = a + \text{labindex}*\text{deltax};
\]
\[
dx = \text{deltax}/n; \quad \% \text{increment length for each rectangle}
\]
\[
x = ai+dx/2:dx:bi-dx/2; \quad \% \text{mid-points of } n \text{ increments per worker}
\]
\[
\text{local_int} = \text{sum} (\cos(x)*dx); \quad \% \text{integral sum for current worker}
\]
\[
\text{intSPMD} = \text{gplus} (\text{local_int},1); \quad \% \text{total sum saved in lab 1 only}
\]

\[
\textit{end} \quad \% \text{spmd}
\]

\[
\text{local_int}{2} \quad \% \text{prints } \text{local_int} \text{ of lab 2 (from Client)}
\]
\[
\text{L} = [\text{local_int}{:}]; \quad \% \text{saves } \text{local_int} \text{ from labs to } L \text{ on Client; needs [ ]}
\]

- Variables defined in *spmd* are class composite — check with *whos*
- Composite array intSPMD is NULL on labs 2:end; set by *gplus*
- \(n=2; \textit{spmd, a=rand(n), end, A=[a{:}];} \quad \% \text{local } a \text{ is 2D; } A \text{ on Client is 2x8}
- \(B = a{1}; \text{ for } j=2:4, B(:,:,j) = a{j}; \textit{end} \quad \% B \text{ is 2x2x4 3D double array}

Integration Example — *drange*

Similar to *parfor* but used within *spmd*

\[
\begin{align*}
n &= 500; \quad \text{% number of integral increments per lab} \\
a &= 0; \ b &= \pi/2; \quad \text{% lower and upper limits of integration} \\
\text{spmd} \\
deltax &= (b - a)/\text{numlabs}; \ % \text{increment length per worker} \\
\text{for } i=\text{drange}(1:\text{numlabs}) \\
\quad ai &= a + (i - 1)\cdot\text{deltax}; \ % \text{local integral limits, \([ai, bi]\)} \\
\quad bi &= a + i\cdot\text{deltax}; \\
\quad dx &= \text{deltax}/n; \ % \text{increment length per rectangle} \\
\quad x &= ai+dx/2:dx:bi-dx/2; \ % \text{mid-points of n increments} \\
\quad \text{local_int} &= \text{sum}(\cos(x)\cdot dx); \\
\text{end} \\
\quad \text{intDrange} &= \text{gplus}(\text{local_int}, 1); \ % \text{send total sum to lab 1} \\
\text{end} \ % \text{spmd} \\
\end{align*}
\]

\[
\begin{align*}
\text{myint} &= \text{intDrange}\{1\}; \ % \text{collect integral from lab 1}
\end{align*}
\]
Integration Example — drange

Similar to parfor but used within spmd

\[ m = 10000; \]  % number of integral increments for all labs
\[ a = 0; b = \pi/2; \]  % lower and upper limits of integration
\[ \text{spmd} \]
\[ dx = (b - a)/m; \]  % increment length per rectangle
\[ \text{local_int} = 0; \]
\[ \text{for } i=\text{drange}(1:m) \]
\[ \text{local_int} = \text{local_int} + \cos(a+(i-0.5)\times dx)\times dx; \]
\[ \text{end} \]
\[ \text{intDrange} = \text{gplus}(\text{local_int}, 1); \% \text{send total sum to lab 1} \]
\[ \text{end } \% \text{spmd} \]

\[ \text{myint} = \text{intDrange}{1}; \]  % send integral to client

Advantage: let drange do work load distribution
Disadvantage: code not vectorized
Array Distributions

- Objective is to distribute data, hence workload, to workers for shorter walkclock time.

- Data distribution utilities:
  - `distributed` – Data distributed from client and access readily on client. Data always distributed along the last dimension
  - `codistributed` – Data distributed within `spmd`
    - Distribute along arbitrary dimension
    - Distribute in parallel
    - Distributed array created directly (locally) on worker
    - No communication required to distribute (from client)
  - `composite` – arrays not created inside spmd region with above commands are in this class
How To Distribute Arrays

Ways to distribute data:
- Partitioning a larger array
  Often, this larger array already exist on Client. Distributing it to labs will take up further memory as well as run time.
- Building from smaller arrays
  Array sections are constructed directly on individual labs resulting in smaller memory footprint.
- Created with MATLAB constructor function (rand, zeros, . . .)
  Supported by distributed and codistributed
  Example. >>> d = distributed.rand(n)  % distributes nxn rand
Array Distribution Usage Examples

matlabpool open local 4
A = magic(8);  % A is on Client; accessible from labs
ad = distributed(A);  % on Client, divide A to 8x2s along last dim
N = size(A,2)/matlabpool('size');  % columns distributed per lab
spmd
    n = size(A,2)/numlabs;  % N is on Client; n is composite
    al = getLocalPart(ad);  % al is local to the current lab
    al(:,1) = 0;  % modify locally on lab; use local index
    ad(:,1+(labindex-1)*N) = 0;  % not local; must use global index
    aa = gcat(al);  % concatenate variant array al to replicated aa
    ab = codistributed(aa);  % co-distribute along last dim (default)
    br = codistributed(A,codistributor1d(1));  % distribute by rows
        % partition from larger array
    b = codistributed.rand(8);  % built from smaller arrays
end
B = [al{:}];  % combine labs’ 8x2 arrays to 8x8 array on Client
matlabpool close
Array Distribution Usage Examples (2)

```
>> whos
    Name      Size            Bytes  Class          Attributes
     A         8x8             512   double
     B         8x8             512   double
      N        1x1              8    double
      aa       1x8           2041   Composite
      ab        8x8           2077   distributed
      ad        8x8           2077   distributed
      al        1x8           2041   Composite
       b        8x8           2077   distributed
      br        8x8           2077   distributed
      n         1x8           2041   Composite
```
Data Parallel Example – Matrix Multiply

Computations of distributed arrays need not be in an spmd region. The operations, however, are governed by *spmd* paradigm.

```matlab
>> maxNumCompThreads(4)
>> n = 3000; A = rand(n); B = rand(n);
>> C = A * B;           % run with 4 threads
>> maxNumCompThreads(1);% set threads to 1
>> C1 = A * B;          % run on single thread
>> matla pool open local 4
>> a = distributed(A);  % distributes A, B along columns on client
>> b = distributed(B);  % a, b on workers; accessible from client
>> c = a * b;           % run on workers; c is distributed
>> matla pool close
```
Additional Ways to Distribute Matrices

More efficient to distribute arrays within `spmd`:
- All workers participate in distribution (*co-distribute*)
- No need to decompose array on client, then copy to individual workers as with *distributed*.

```matlab
>> matlabpool open 4
>> A = rand(3000); B = rand(3000);
>> spmd
    p = rand(n, codistributor1d(1)); % 2 ways to directly create
    q = codistributed.rand(n); % distributes rand array by col.
    s = p * q; % run on workers; s is codistributed
    u = codistributed(A, codistributor1d(1)); % by row
    v = codistributed(B, codistributor1d(2)); % by column
    w = u * v; % run on workers; w is distributed
end
>> matlabpool close
```
A Case Of *parfor* vs *spmd*

% First, consider the serial code
N = 40; B = rand(N); C = rand(N);
p = ones(N,1); v = p; a = rand(N,1);
for i = 1:1000  % iterative loop
    for j=1:N
        vn(j)=foo(v,p(j),a,B(j,:),C(j,:));  % compute new v
    end
    if max(abs(vn-v))< 1e-7  % check for convergence
        fprintf('****** Converged at iteration %5d\n',i);
        break;
    end
    v = vn;  % update v
end
The *parfor* version

```matlab
matlabpool open 4
N = 40; B = rand(N); C = rand(N);
p = ones(N,1); v = p; a = rand(N,1);
for i = 1:1000     % iterative loop
    parfor j=1:N
        vn(j)=foo(v,p(j),a,B(j,:),C(j,:));   % compute new v
    end
    if max(abs(vn-v))< 1e-7   % check for convergence
        fprintf('****** Converged at iteration %5d
',i);
        break;
    end
    v = vn;   % update v
end
matlabpool close
```

*Parfor and other parallel paradigms have high overhead cost to start. Avoid using parfor within an iterative loop.*
**spmd version**

```matlab
matlabpool open 4
N = 40;
spmd  % Single Program Multiple Data parallel paradigm
B=codistributed.rand(N,codistributor('1d',1));
C=codistributed.rand(N,codistributor('1d',1));
p=codistributed.ones(N,1,codistributor());
v = p; % initial guess; v is distributed by virtue of p
for i = 1:1000  % iterative loop
    vn=foo(v,p,a,B,C); % compute new v
    if max(abs(vn-v))< 1e-7  % check for convergence
        printf('****** Converged at iteration %5d\n',i);
        break;
    end
    v = vn; % update v
end  % spmd
matlabpool close
```

*SPMD need be started once. Distributed arrays remain in the worker space until spmd region ends.*
Linear algebraic system Example: $Ax = b$

matlabpool open 4

% serial operations
n = 3000; M = rand(n); x = ones(n,1);
[A, b] = linearSystem(M, x);
u = A\b;  % solves Au = b; u should equal x
clear A b

% parallel operations in spmd
spmd
m = codistributed(M, codistributor('1d',2));  % by column
y = codistributed(x, codistributor('1d',1));  % by row
[A, b] = linearSystem(m, y);
v = A\b;
end
clear A b m y

% parallel operations from client
m = distributed(M); y = distributed(x);
[A, b] = linearSystem(m, y);
w = A\b;

function [A, b] = linearSystem(M, x)
% Returns A and b of linear system Ax = b
A = M + M';  % A is real and symmetric
b = A * x;  % b is the RHS of linear system
Task Parallel vs. Data Parallel

matlabpool open 4
n = 3000; M = rand(n); x = ones(n,1);
% Solves 4 cases of Ax=b sequentially, each with 4 workers
for i=1:4
    m = distributed(M); y = distributed(x*i);
    [A, b] = linearSystem(m, y); % computes with 4 workers
    u = A\b;                  % solves each case with 4 workers
end
clear A b m y
% solves 4 cases of Ax=b concurrently (with parfor)
parfor i=1:4
    [A, b] = linearSystem(M, x*i); % computes on 1 worker
    v = A\b;                           % solves with 1 worker
end
% solves 4 cases of Ax=b concurrently (with drange)
spmd
    for i=drange(1:4)
        [A, b] = linearSystem(M, x*i); % computes on 1 worker
        w = A\b; % 1 worker
    end
end
matlabpool close

function [A, b] = linearSystem(M, x)
% Returns A and b of linear system Ax = b
A = M + M'; % A is real and symmetric
b = A * x;  % b is the RHS of linear system
7. How Do I Parallelize My Code?

1. Profile serial code with *profile*.
2. Identify section of code or function within code using the most CPU time.
3. Look for ways to improve code section performance.
4. See if section is parallelizable and worth parallelizing.
5. If warranted, research and choose a suitable parallel algorithm and parallel paradigm for code section.
6. Parallelize code section with chosen parallel algorithm.
7. To improve performance further, work on the next most CPU-time-intensive section. Repeats steps 2 – 7.
8. Analyze the performance efficiency to know what the sweet-spot is, i.e., given the implementation and platforms on which code is intended, what is the minimum number of workers for speediest turn-around (see the Amdahl’s Law page).
8. How Well Does PCT Scales?

- Task parallel applications generally scale linearly.

- Data parallel applications’ parallel efficiency depend on individual code and algorithm used.

- My personal experience is that the runtime of a reasonably well tuned MATLAB program running on single or multi-processor is an order of magnitude slower than an equivalent C/FORTRAN code.

- Additionally, the PCT’s communication is based on the Ethernet. MPI on SCC uses Infiniband and is faster than Ethernet. This further disadvantaged PCT if your code is communication-bound.
Speedup Ratio and Parallel Efficiency

$S$ is ratio of $T_1$ over $T_N$, elapsed times of 1 and $N$ workers.  
$f$ is fraction of $T_1$ due to code sections not parallelizable.

$$S = \frac{T_1}{T_N} \left( \frac{T_1}{(f + \frac{1-f}{N})T_1} \right) \left( \frac{1}{f} \right) \text{ as } N \to \infty$$

Amdahl’s Law above states that a code with its parallelizable component comprising 90% of total computation time can at best achieve a 10X speedup with lots of workers. A code that is 50% parallelizable speeds up two-fold with lots of workers.

The parallel efficiency is $E = S / N$  
Program that scales linearly ($S = N$) has parallel efficiency 1.  
A task-parallel program is usually more efficient than a data-parallel program. Parallel codes can sometimes achieve super-linear behavior due to efficient cache usage per worker.
Example of Speedup Ratio & Parallel Efficiency
Batch Processing On the SCC

MATLAB provides various ways to submit and run jobs in the background or in batch. RCS recommends a simple and effective method for all PCT batch processing on Katana. *Cut-and-paste the below into a file, say, mbatch*

```bash
#!/bin/csh
# MATLAB script for running serial or parallel background jobs
# For MATLAB applications that contain MATLAB Parallel Computing Toolbox
# parallel operations request (matlabpool or dfeval), a batch job
# will be queued and run in batch (using the SGE configuration)
# Script name: mbatch (you can change it)
# Usage: katana% mbatch <m-file> <output-file>
# <m-file>: name of m-file to be executed, DONOT include .m ($1)
# <output-file>: output file name; may include path ($2)
matlab -nodisplay -r "$1 exit" ! $2 &
```

scc1% chmod +x mbatch        ← give mbatch execute attribute

scc1% mbatch my_mfile myOutput

Do not start MATLAB with `-nojvm`. PCT requires Java.
Compile m-file into Standalone

Running many single- or multi-core jobs concurrently put heavy demand on MATLAB and tool box licenses. This may prevent you and others from running MATLAB jobs whenever or as often as you need.

We strongly recommend that you use MATLAB compiler `mcc` to compile your applications into a standalone executable. This executable runs outside the MATLAB environment and needs no licenses.

http://www.bu.edu/tech/support/research/software-and-programming/common-languages/matlab/standalone/

```
scc1% myExec 200 4  # runs executable myExec at system prompt
scc1% qsub batch_scc # submit batch_scc to run myExec in batch
```
Use ‘local’ Config. on SCC

When you submit a job to an SCC node, it is treated as ‘local.’

1. Request an interactive batch session
   (You need x-win32 or MobaXterm on your client computer)
   ```
   scc1% qsh -pe omp 4
   ```
   Request a node with 4 processors.

2. In the new X-window, run matlab
   ```
   scc1% matlab&
   ```

3. Request workers in MATLAB window
   ```
   >> matlabpool open local 4
   or
   >> pmode start local 4
   ```
   Only need a PCT license; no worker licenses needed.
Communications Among Workers, Client

- Communications between workers and MATLAB client
  pmode: `lab2client, client2lab`
  matlabpool: `A = a{1} % from lab 1 to client`

- Collective communications among workers
  `gather, gop, gplus, gcat`
MPI Point-to-Point Communications

• MPI point-to-point communication among workers

\texttt{labSend} and \texttt{labReceive}

% Example: each lab sends its lab # to lab 1 and sum data on lab 1
\texttt{matlabpool open 4}
\texttt{spmd} % MPI requires spmd
\texttt{a = labindex;} % define \texttt{a} on workers
\texttt{If labindex == 1 \% on worker 1 . . .}
\texttt{\% lab 1 is designated to accumulate sum on \texttt{a}}
\texttt{\hspace{1cm} s = a;} % sum \texttt{s} starts with \texttt{a} on worker 1
\texttt{\hspace{1cm} for k = 2:numlabs \% loop over remaining workers}
\texttt{\hspace{1cm} s = s + labReceive(k);} % receive \texttt{a} from worker \texttt{k}, then add to \texttt{s}
\texttt{\hspace{1cm} end}
\texttt{else \% for all other workers . . .}
\texttt{\hspace{1cm} labSend(a,1);} % send \texttt{a} on workers 2 to 4 to worker 1
\texttt{\hspace{1cm} end}
\texttt{\hspace{1cm} end \% spmd}
\texttt{indexSum = s{1}; % copy \texttt{s} on lab 1 to client}
\texttt{matlabpool close}

\textbf{It is illegal to communicate with itself.}
For some problems, parallel computing can now be performed with GPUs achieving better performance than with CPUs.

GPUs are now much more prevalent than a mere few years ago.

MATLAB GPU utilities are growing.

On the SCC, there are compute nodes equipped with GPUs.

- To request a GPU for interactive use (for debugging, learning)
  scc1% qsh -l gpus=1
- To submit batch job requesting a GPU
  scc1% qsub -l gpus=1 batch_scc

Basic GPU operations:

```matlab
>> n = 3000; % matrix size
>> a = rand(n); % n x n random matrix
>> A = gpuArray(a) % copy a to the GPU
>> B = gpuArray.rand(n) % create random array directly on GPU
>> C = A * B; % matrix multiply on GPU
>> c = gather(C); % bring data back to base workspace
```
maxIterations = 500;
gridSize = 3000;
xlim = [-1, 1];
ylim = [0, 2];
x = gpuArray.linspace( xlim(1), xlim(2), gridSize );
y = gpuArray.linspace( ylim(1), ylim(2), gridSize );
[xGrid,yGrid] = meshgrid( x, y );
z0 = complex(xGrid, yGrid);
count0 = ones( size(z0) );
count = arrayfun( @aSerialFct, z0, count0, maxIterations );
count = gather( count ); % Fetch the data back from the GPU
Useful SCV Info

- **RCS home page**  (www.bu.edu/tech/research/)
- **Resource Applications**  www.bu.edu/tech/accounts/special/research/
- **Help**
  - **System**
    - help@scc.bu.edu, bu.service–now.com
  - **Web-based tutorials**
    http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/
    (MPI, OpenMP, MATLAB, IDL, Graphics tools)
  - **HPC consultations by appointment**
    - Katia Oleinik (koleinik@bu.edu)
    - Yann Tambouret (yannpaul@bu.edu)
    - Kadin Tseng (kadin@bu.edu)