INTRODUCTION TO MATLAB PARALLEL COMPUTING TOOLBOX

Keith Ma ---------------------------------------- keithma@bu.edu
Research Computing Services -------- help@rcs.bu.edu
Boston University ------------------------------------------
Overview

• Goals:
  1. Basic understanding of parallel computing concepts
  2. Familiarity with MATLAB parallel computing tools

• Outline:
  • Parallelism, defined
  • Parallel “speedup” and its limits
  • Types of MATLAB parallelism
    • multi-threaded/implicit, distributed, explicit)
  • Tools: parpool, SPMD, parfor, gpuArray, etc
**Parallel Computing**

**Definition:** The use of two or more processors in combination to solve a single problem.

- Serial performance improvements have slowed, while parallel hardware has become ubiquitous.
- Parallel programs are typically harder to write and debug than serial programs.

Parallel speedup, and its limits (1)

• “Speedup” is a measure of performance improvement

\[
\text{speedup} = \frac{\text{time}_{\text{old}}}{\text{time}_{\text{new}}}
\]

• For a parallel program, we can run with an arbitrary number of cores, \( n \).

• Parallel speedup is a function of the number of cores

\[
\text{speedup}(p) = \frac{\text{time}_{\text{old}}}{\text{time}_{\text{new}}(p)}
\]
Parallel speedup, its limits (2)

- Amdahl’s law: Ideal speedup for a problem of fixed size
- Let: $p =$ number of processors/cores
  $\alpha =$ fraction of the program that is strictly serial
  $T =$ execution time

Then:

$$T(p) = T(1)\left(\alpha + \frac{1}{p} (1 - \alpha)\right)$$

And:

$$S(p) = \frac{T(1)}{T(p)} = \frac{1}{\alpha + \frac{1}{p} (1 - \alpha)}$$

- Think about the limiting cases: $\alpha = 0$, $\alpha = 1$, $p = 1$, $p = \infty$
Parallel speedup, and its limits (3)

- Diminishing returns as more processors are added
- Speedup is limited if $\alpha < 0$
- “Linear speedup” is the best you can do (usually)

*Accelerating MATLAB Performance, Yair Altman, 2015*
Parallel speedup, and its limits (3)

- The program “chokes” if too many cores are added
- Caused by communication cost and overhead, resource contention

![A more realistic Amdahl's law graph]

- Realistic Amdahl's law of parallelization efficiency

*Accelerating MATLAB Performance, Yair Altman, 2015*
Hardware: single core

- No parallelism

- Good luck finding one…
Hardware: multi-core

- Each processor core runs independently
- All cores can access system memory
- Common in desktops, laptops, smartphones, probably toast...
Hardware: multi-core, multi-processor

- Each processor core runs independently
- All cores can access system memory
- Common in workstations and servers (including the SCC here at BU)
Hardware: accelerators

- Accelerator/GPU is an a separate chip with many simple cores.

- GPU memory is separate from system memory.

- Not all GPUs are suitable for research computing tasks (need support for APIs, decent floating-point performance).
Hardware: clusters

- Several independent computers, linked via network
- System memory is distributed (i.e. each core cannot access all cluster memory)
- Bottlenecks: inter-processor and inter-node communications, contention for memory, disk, network bandwidth, etc.
Multithreaded parallelism... one instance of MATLAB automatically generates multiple simultaneous instruction streams. Multiple processors or cores, sharing the memory of a single computer, execute these streams. An example is summing the elements of a matrix.

Distributed computing.

Explicit parallelism.
Three Types of Parallel Computing

*Parallel MATLAB: Multiple Processors and Multiple Cores, Cleve Moler, MathWorks*

- **Multithreaded parallelism.**
- **Distributed computing.**
  multiple instances of MATLAB run multiple independent computations on separate computers, each with its own memory...In most cases, a single program is run many times with different parameters or different random number seeds.
- **Explicit parallelism.**
Three Types of Parallel Computing

**Multithreaded parallelism.**

**Distributed computing.**

**Explicit parallelism…**
several instances of MATLAB run on several processors or computers, often with separate memories, and simultaneously execute a single MATLAB command or M-function. New programming constructs, including parallel loops and distributed arrays, describe the parallelism.
MATLAB parallel computing toolbox (PCT)

- Supports multithreading (adds GPUs), distributed parallelism, and explicit parallelism

- You need...
  - MATLAB and a PCT license
    - Note that licenses are limited at BU – 500 for MATLAB, 45 for PCT
  - Parallel hardware, as discussed above
Distributed Jobs

• Define and run independent jobs
  • No need to parallelize code!
  • Expect linear speedup
  • Each task must be entirely independent

• Approach: define jobs, submit to a scheduler, gather results

• Using MATLAB scheduler – not recommended:

```matlab
c = parcluster;
j = createJob(c);
createTask(j, @sum, 1, {[1 1]});
createTask(j, @sum, 1, {[2 2]});
createTask(j, @sum, 1, {[3 3]});
submit(j);
wait(j);
out = fetchOutputs(j);
```
Distributed Jobs on SCC

• For task-parallel jobs on SCC, use the cluster scheduler (not the tools provided by PCT)

• To define and submit one job:
  qsub matlab -nodisplay -singleCompThread -r "rand(5), exit"

• To define and submit many jobs, use a job array:
  qsub -N myjobs -t 1-10:2 matlab -nodisplay -singleCompThread -r \
      "rand($SGE_TASK_ID), exit"

• Results must be gathered manually, typically by a “cleanup” job that runs after the others have completed:
  qsub -hold_jid myjobs matlab -nodisplay -singleCompThread -r \
      "my_cleanup_function, exit"

• Much more detail on MATLAB batch jobs on the SCC [here](#).
Parallel Jobs

- Split up the work for a single task
  - Must write parallel code
  - Your mileage may vary - some parallel algorithms may run efficiently and others may not
  - Programs may include both parallel and serial sections

- Client:
  - The head MATLAB session – creates workers, distributes work, receives results

- Workers/Labs:
  - independent, headless, MATLAB sessions
  - do not share memory
  - create before you need them, destroy them when you are done

- Modes:
  - pmode: special interactive mode for learning the PCT and development
  - matlabpool/parpool: is the general-use mode for both interactive and batch processing.
Parallel Jobs: matlabpool/parpool

- *matlabpool/parpool* creates workers (a.k.a labs) to do parallel computations.

- **Usage:**

  ```
  parpool(2)
  
  perform parallel tasks
  
  delete(gcp)
  ```

- No access to GUI or graphics (workers are “headless”)

- Parallel method choices that use *parpool* workers:
  - *parfor*: parallel for-loop; can’t mix with *spmd*
  - *spmd*: single program multiple data parallel region
parfor (1)

- Simple: a parallel for-loop

- Work load is distributed evenly and automatically according to loop index. Details are intentionally opaque to user.

- Many additional restrictions as to what can and cannot be done in a parfor loop – this is the price of simplicity

- Data starts on client (base workspace), automatically copied to workers’ workspaces. Output copied back to client when done.

- Basic usage:

  ```matlab
  parpool(2)
  x = zeros(1,10);
  parfor i=1:10
      x(i) = sin(i);
  end
  delete(gcp)
  ```
parfor (2)

- For the parfor loop to work, variables inside the loop must all fall into one of these categories:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>A loop index variable for arrays</td>
</tr>
<tr>
<td>Sliced</td>
<td>An array whose segments are manipulated on different loop iterations</td>
</tr>
<tr>
<td>Broadcast</td>
<td>A variable defined before the loop and is used inside the loop but never modified</td>
</tr>
<tr>
<td>Reduction</td>
<td>Accumulates a value across loop iterations, regardless of iteration order</td>
</tr>
<tr>
<td>Temporary</td>
<td>Variable created inside the loop, but not used outside the loop</td>
</tr>
</tbody>
</table>

```matlab
c = pi; s = 0;
X = rand(1,100);
parfor k = 1 : 100
    a = k; % a - temporary variable; k - loop variable
    s = s + k; % s - reduction variable
    if i <= c % c - broadcast variable
        a = 3*a - 1;
    end
    Y(k) = X(k) + a; % Y - output sliced var; X - input sliced var
end
```
parfor (3): what can’t you do?

- Data dependency: loop iterations are must be independent

```matlab
a = zeros(1,100);
parfor i = 2:100
    a(i) = myfct(a(i-1));
end
```
parfor (4): what can’t you do?

- Data dependency exceptions: “Reduction” operations that combine results from loop iterations in order-independent or entirely predictable ways

```matlab
% computing factorial using parfor
x = 1;
parfor idx = 2:100
    x = x * idx;
end

% reduction assignment with insert operator
d = [];  % Assume d is already initialized
parfor idx = -10 : 10
    d = [d, idx*idx*idx];
end

% reduction assignment of a row vector
v = zeros(1,100);
parfor idx = 1:n
    v = v + (1:100)*idx;
end
```
parfor (4): what can’t you do?

- Data dependency exception: “Reduction” operations that combine results from loop iterations

- Reduction operations must not depend on loop order
  - must satisfy $x \diamond (y \diamond z) = (x \diamond y) \diamond z$
  - Plus (+) and multiply (*) operators pass, subtract (-) and divide (/) may fail.

```matlab
s = 0;
parfor i=1:10
    s = i-s;
end
```
parfor (5): what can’t you do?

• Loop index must be consecutive integers.

parfor i = 1 : 100    % OK
parfor i = -20 : 20   % OK
parfor i = 1 : 2 : 25  % No
parfor i = -7.5 : 7.5 % No
A = [3 7 -2 6 4 -4 9 3 7]; parfor i = find(A > 0) % No

• ...and more! See the documentation
Integration example (1)

Integrate of the \( \cos(x) \) between 0 and \( \pi/2 \) using mid-point rule

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

% 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

\[ x(1) = a + \frac{dx}{2} \quad \text{and} \quad x(m) = b - \frac{dx}{2} \]
Integration example (3): *parfor*

This example performs parallel integration with *parfor*.

```matlab
matlabpool open 4
tic
    m = 10000;
a = 0;
b = pi/2;
dx = (b - a)/m; % increment length
intParfor2 = 0;
parfor i=1:m
    intParfor2 = intParfor2 + cos(a+(i-0.5)*dx)*dx;
end
toc
matlabpool close
```
spmd (1)

spmd = single program multiple data
- Explicitly and/or automatically...
  - divide work and data between workers/labs
  - communicate between workers/labs

- Syntax:

```matlab
% execute on client/master
spmd
% execute on all workers
end
% execute on client/master
```
**spmd (2): Integration example (4)**

- **Example: specifying different behavior for each worker**
  - `numlabs()` --- Return the total number of labs operating in parallel
  - `labindex()` --- Return the ID for this lab

```
parpool(2)
tic  % includes the overhead cost of spmd
spmd
  m = 10000;
  a = 0;
  b = pi/2;
  n = m/numlabs;  % # of increments per lab
  deltax = (b - a)/numlabs;  % length per lab
  ai = a + (labindex - 1)*deltax;  % local integration range
  bi = a + labindex*deltax;
  dx = deltax/n;  % increment length for lab
  x = ai+dx/2:dx:bi-dx/2;
  intSPMD = sum(cos(x)*dx);  % integral sum per worker
  intSPMD = gplus(intSPMD,1);  % global sum over all workers
end  % spmd
toc
delete(gcp)
```
spmd (3): where are the data?

- Memory is not shared by the workers. Data can be shared between workers using explicit MPI-style commands:

```matlab
spmd

if labindex == 1
    A = labBroadcast(1,1:N); % send the sequence 1..N to all
else
    A = labBroadcast(1); % receive data on other workers
end

% get an array chunk on each worker
I = find(A > N*(labindex-1)/numlabs & A <= N*labindex/numlabs);

% shift the data to the right among all workers
to = mod(labindex, numlabs) + 1; % one to the right
from = mod(labindex-2, numlabs) + 1; % one to the left
I = labSendReceive(labTo, labFrom, I);

out = gcat(I, 2, 1); % reconstruct the shifted array on the 1st worker
end
```
spmd (4): where are the data?

- Memory is not shared by the workers. Data can be shared between workers using **special data types: composite, distributed, codistributed**

- **Composite**: variable containing references to unique values on each worker.
  - On the workers, accessed like normal variables
  - On the client, elements on each worker are accessed using cell-array style notation.

- **Distributed/Codistributed**: array is partitioned amongst the workers, each holds some of the data.
  - All elements are accessible to all workers
  - The distinction is subtle: a codistributed array on the workers is accessible on the client as a distributed array, and vice versa
spmd (5): where are the data?

- created **before** *spmd* block: copied to all workers
spmd (6): where are the data?

- created **before** `spmd` block: copied to all workers
- created **in** `spmd` block, then unique to worker, *composite* on client

```
>> spmd
>>     q = magic(labindex + 2);
>> end
>> q
```

```matlab
q =

Lab 1: class = double, size = [3 3]
Lab 2: class = double, size = [4 4]
```

```
>> q{1}
```

```matlab
ans =

          8     1     6
          3     5     7
          4     9     2
```
spmd (7): where are the data?

- created **before** `spmd` block: copied to all workers
- created **in** `spmd` block, then unique to worker, *composite* on client
- created as a **distributed** array before `spmd`: divided up as a **codistributed** array on workers

```matlab
W = ones(6,6);
W = distributed(W); % Distribute to the workers
spmd
    T = W*2; % Calculation performed on workers, in parallel
    % T and W are both **codistributed** arrays here
end
T % View results in client
    % T and W are both **distributed** arrays here
```
spmd (8): where are the data?

- created **before** `spmd` block: copied to all workers
- created **in** `spmd` block, then unique to worker, *composite* on client
- created as a **distributed** array **before** `spmd`: divided up as a **codistributed** array on workers
- created as a **codistributed** array **in** `spmd`: divided up on workers, accessible as **distributed** array on client

```matlab
spmd(2)
    RR = rand(20, codistributor());  % worker stores 20x10 array
end
```
Distributed matrices (1): creation

- There are many ways to create distributed matrices – you have a great deal of control.

\[ A = \mathrm{rand}(3000); \quad B = \mathrm{rand}(3000); \]
\[ \text{spmd} \]
\[ p = \mathrm{rand}(n, \text{codistributor1d}(1)); \quad \% \text{2 ways to directly create} \]
\[ q = \text{codistributed.rand}(n); \quad \% \text{distributed random array} \]
\[ s = p \times q; \quad \% \text{run on workers; } s \text{ is distributed} \]
\[ \% \text{distribute matrix after it is created} \]
\[ u = \text{codistributed}(A, \text{codistributor1d}(1)); \quad \% \text{by row} \]
\[ v = \text{codistributed}(B, \text{codistributor1d}(2)); \quad \% \text{by column} \]
\[ w = u \times v; \quad \% \text{run on workers; } w \text{ is distributed} \]
\[ \text{end} \]
Distributed matrices (2): efficiency

- For matrix-matrix multiply, here are 4 combinations on how to distribute the 2 matrices (by row or column) - some perform better than others.

\[
n = 3000; \ A = \text{rand}(n); \ B = \text{rand}(n);
\]

\[
\begin{align*}
\text{spmd} \\
\ ar &= \text{codistributed}(A, \text{codistributor1d}(1)) \ % \text{distributed by row} \\
\ ac &= \text{codistributed}(A, \text{codistributor1d}(2)) \ % \text{distributed by col} \\
\ br &= \text{codistributed}(B, \text{codistributor1d}(1)) \ % \text{distributed by row} \\
\ bc &= \text{codistributed}(B, \text{codistributor1d}(2)) \ % \text{distributed by col} \\
\ crr &= ar \times br; \quad crc = ar \times bc; \\
\ ccr &= ac \times br; \quad ccc = ac \times bc;
\end{align*}
\]

\[
\text{end}
\]

- Wall-clock times of the four ways to distribute \( A \) and \( B \)

<table>
<thead>
<tr>
<th>C (row x row)</th>
<th>C (row x col)</th>
<th>C (col x row)</th>
<th>C (col x col)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.44</td>
<td>2.22</td>
<td>3.95</td>
<td>3.67</td>
</tr>
</tbody>
</table>
Distributed matrices (3): function overloading

- Some function/operators will recognize distributed array inputs and execute in parallel

matlabpool open 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
a = distributed(A); % distributes A, B from client
b = distributed(B); % a, b on workers; accessible from client
c = a * b; % run on workers; c is distributed
matlabpool close

- Wallclock times for the above (distribution time accounted separately)

<table>
<thead>
<tr>
<th>C1 = A * B (1 thread)</th>
<th>C = A * B (4 threads)</th>
<th>a = distribute(A) b = distribute(B)</th>
<th>c = a * b (4 workers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.06</td>
<td>3.25</td>
<td>2.15</td>
<td>3.89</td>
</tr>
</tbody>
</table>
Linear system example: $Ax = b$

% serial

$n = 3000; M = \text{rand}(n); x = \text{ones}(n,1);$
$[A, b] = \text{linearSystem}(M, x);$
$u = A\backslash b; \quad \% \text{solves } Au = b; u \text{ should equal } x$

clear $A$ $b$

% parallel in \text{spmd}

\text{spmd}

$m = \text{codistributed}(M, \text{codistributor}('1d',2)); \quad \% \text{by column}$
$y = \text{codistributed}(x, \text{codistributor}('1d',1)); \quad \% \text{by row}$
$[A, b] = \text{linearSystem}(m, y);$
$v = A\backslash b;$
\text{end}

clear $A$ $b$ $m$ $y$

% parallel using \text{distributed array overloading}

$m = \text{distributed}(M); y = \text{distributed}(x);$
$[A, b] = \text{linearSystem}(m, y);$
$W = A\backslash b;$

\hspace{1cm} function \[A, b] = \text{linearSystem}(M, x)\]
\hspace{1cm} \% \text{Returns } A \text{ and } b \text{ of linear system } Ax = b$
\hspace{1cm} A = M + M'; \quad \% A \text{ is real and symmetric}$
\hspace{1cm} b = A * x; \quad \% b \text{ is the RHS of linear system}$
Parallel Jobs: pmode

`>> pmode start 4`

- Commands at the “P>>” prompt are executed on all workers
- Use `if` with `labindex` to issue instructions specific to workers
- Memory is NOT shared!

Terminology:
- `worker = lab = processor`
- `labindex = processor id`
- `numlabs = number of processors`
Using GPUs (1)

- For some problems, GPUs achieve better performance than CPUs.
- MATLAB GPU utilities are limited, but growing.

- **Basic GPU operations:**

  ```
  >> 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
  ```

- 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
```
Using GPUs (2): arrayfun

\[
\begin{align*}
\text{maxIterations} & = 500; \\
\text{gridSize} & = 3000; \\
x\text{lim} & = [-1, 1]; \\
y\text{lim} & = [0, 2]; \\
x & = \text{gpuArray.linspace}(x\text{lim}(1), x\text{lim}(2), \text{gridSize}); \\
y & = \text{gpuArray.linspace}(y\text{lim}(1), y\text{lim}(2), \text{gridSize}); \\
[x\text{Grid}, y\text{Grid}] & = \text{meshgrid}(x, y); \\
z0 & = \text{complex}(x\text{Grid}, y\text{Grid}); \\
\text{count0} & = \text{ones}(\text{size}(z0)); \\
\text{count} & = \text{arrayfun}(\text{@SerialFct}, z0, \text{count0}, \text{maxIterations}); \\
\text{count} & = \text{gather}(\text{count}); \% \text{Fetch the data back from the GPU}
\end{align*}
\]