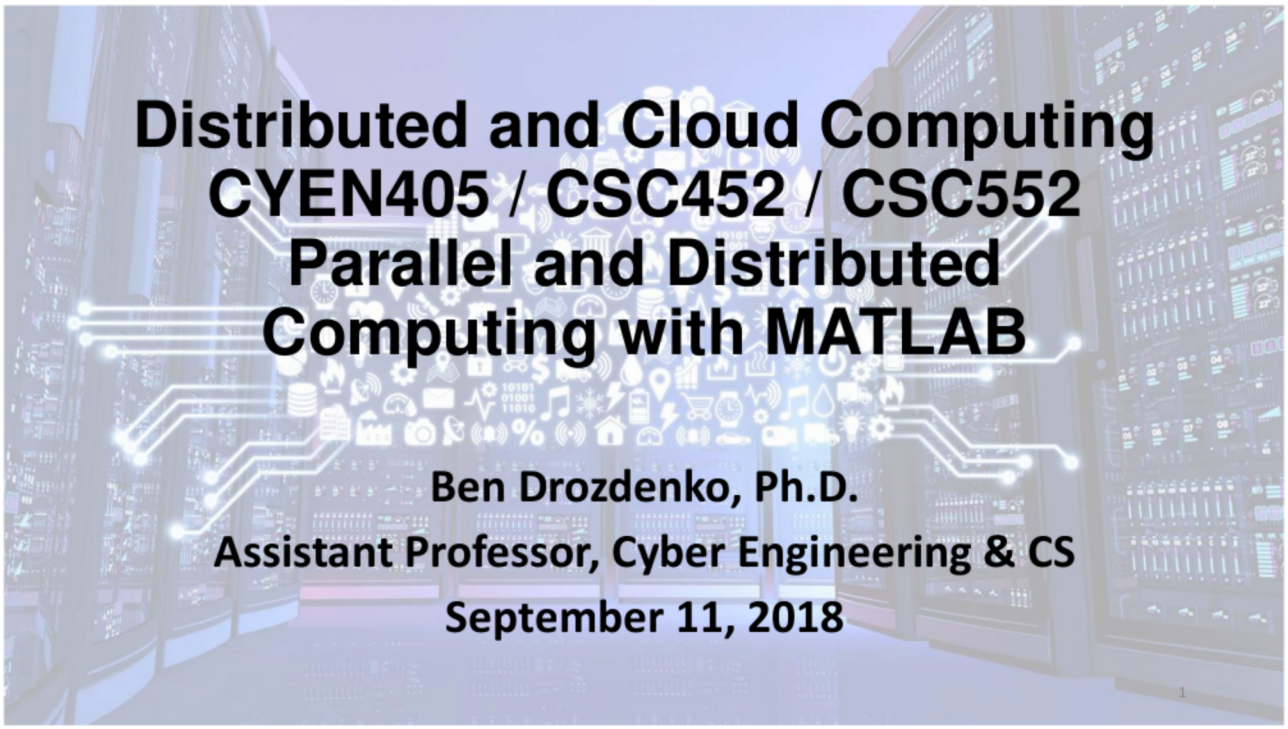


Distributed and Cloud Computing CYEN405 / CSC452 / CSC552 Parallel and Distributed Computing with MATLAB

Ben Drozdenko, Ph.D.
Assistant Professor, Cyber Engineering & CS
September 11, 2018

1

Lecture Outline

- Parallel Computing Toolbox™ (PCT) and MATLAB® Distributed Computing Server™ (MDCS)
  - Starting a Parallel Pool using **`parpool`**
- Task-Parallel Jobs with PCT
  - Embarrassingly Parallel Tasks using **`parfor`**
  - Setting up a Simple Independent Job using **`createJob`**
- Data-Parallel Jobs with PCT
  - Single Program, Multiple Data using **`spmd`**
  - Distributed Datasets and Operations
  - Passing Messages
  - Setting up a Communicating Job

MathWorks® Products

- MATLAB
  - High-level language and interactive environment for numerical computation, visualization, and programming. Language, tools, and built-in math functions to explore multiple approaches and reach a solution faster.
  - Used for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology.
- Parallel Computing Toolbox™ (PCT)
  - Solve computationally-intensive and data-intensive problems using multicore processors, GPUs, and computer clusters.
  - High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—to parallelize MATLAB® applications without CUDA or MPI programming.
- MATLAB® Distributed Computing Server™ (MDCS)
  - Run computationally intensive MATLAB® programs and Simulink® models on computer clusters, clouds, & grids. Develop your program or model on a multicore desktop computer using PCT and then scale up to many computers by running it on MDCS.
  - Server includes a built-in cluster job scheduler and provides support for commonly used third-party schedulers (e.g. Platform LSF, MS Windows HPCS, PBS TORQUE, etc.).
- Source: MathWorks® Product Page: http://www.mathworks.com/products/

Starting Parallel Pool with **`parpool`**

- To get an estimate of the #computational cores on local machine:
  `>> maxNumCompThreads`
- To open a pool of 4 workers, type at the command prompt:
  `>> parpool(4)`
- This command starts 4 new instances of MATLAB, which are available for computations as part of a resource pool.
- In Windows, type Ctrl-Alt-Del to start Task Manager. Select Processes. In total, there are now 4 instances of MATLAB.exe running on the system—4 as part of the pool + original instance.
- To query the current pool:
  `>> p = gcp; p.NumWorkers`
- A parallel pool automatically starts when you execute a parallel language construct that runs on a pool, such as **`parfor`** or **`spmd`**.
- When finished, to close pool, type:
  `>> delete(gcp)`

Lecture Outline (2)

- Parallel Computing Toolbox™ (PCT) and MATLAB® Distributed Computing Server™ (MDCS)
  - Starting a Parallel Pool using **`parpool`**
- Task-Parallel Jobs with PCT
  - Embarrassingly Parallel Tasks using **`parfor`**
  - Setting up a Simple Independent Job using **`createJob`**
- Data-Parallel Jobs with PCT
  - Single Program, Multiple Data using **`spmd`**
  - Distributed Datasets and Operations
  - Passing Messages
  - Setting up a Communicating Job

Task-Parallel Jobs with PCT

- Parallel For Loops with **`parfor`** keyword
  - Each **worker** runs independently
  - Ideally suited for embarrassingly-parallel tasks
- Parallel Jobs using **`createJob`** function
  - Assign specific tasks to run on workers
- Specific rules for usage
  - No communication between loop iterations

Example #1: Birthday Paradox

- What is the probability that in a group of 30 randomly selected individuals, at least two of the individuals will share the same birthday?

Assuming independent events, $p_{bday} = 1 - \frac{30! \cdot \binom{365}{30}}{365^{30}} \approx 70.6\%$

- MATLAB® code for one trial:

```
function match = birthday(groupSize)
bdays = randi(365, groupSize, 1);
bdays = sort(bdays);
match = any(diff(bdays) == 0);
```

- Code to run many trials sequentially ("brute-force algorithm"):

```
function prob = runBirthday(numtrials, groupsize)
matches = false(1,numtrials);
for trial = 1:numtrials
    matches(trial) = birthday(groupsize);
end
prob = sum(matches)/numtrials;
```


Parallel For Loops with **`parfor`**

- Use the keyword **`parfor`** to make any **`for`** loop into a parallel loop that runs the ***independent*** iterations on different workers
- Code to run many trials in parallel:

```
function prob = pRunBirthday(numtrials, groupsize)
matches = false(1,numtrials);
parfor trial = 1:numtrials
    matches(trial) = birthday(groupsize);
end
prob = sum(matches)/numtrials;
```

Time Code using **tic** and **toc**

- With pool still open, time parallel version

```
>> tic; p = pRunBirthday(1e5,30), toc
```

- Close pool, and time sequential version

```
>> delete(gcp);
>> tic; p = runBirthday(1e5,30), toc
```

- On my local machine

N_{trials}	1e5	1e6
Sequential	0.40 sec	4.00 sec
Parallel, $N_{workers}$=4	0.24 sec	1.81 sec
Speedup	1.67X	2.2X

Setup a Parallel Job using **createJob**

- Observe the behavior of a parallel for loop from the following output:

```
>> parfor i=1:10, disp(i); end
```

- Parfor allows for little control over parallel execution. To assign specific tasks to each worker, create an independent job instead:
  1. Connect to a cluster:
     ```
     >> cluster = parcluster('local');
     ```
  2. Create an independent job:
     ```
     >> job = createJob(cluster);
     ```

Setup a Parallel Job using **`createJob`** (cont.)

3. Create many tasks for the job to handle (you could use a `for` loop or a `while` loop for this).

```
>> createTask(job, @runBirthday, 1, {1e5,5});
>> createTask(job, @runBirthday, 1, {1e5,10});
>> createTask(job, @runBirthday, 1, {1e5,15});
>> createTask(job, @runBirthday, 1, {1e5,20});
>> createTask(job, @runBirthday, 1, {1e5,25});
>> createTask(job, @runBirthday, 1, {1e5,30});
```

4. Submit the job. Optionally, wait for it to finish.

```
>> submit(job); wait(job, 'finished');
```

5. Retrieve the results.

```
>> results = fetchOutputs(job);
>> results{end,1}
>> r = cell2mat(results); mean(r)
```

6. When finished, delete job & clear object.

```
>> delete(job); clear job;
```


Example #2: Gene Matching

```
function results = pargenematchsol()
searchSeq = repmat('gattaca', 1, 10);
numTasks = 2;
numBases = 7048095;
cluster = parcluster('local');
job = createJob(cluster);
[startValues, endValues] = splitDataset(numBases, numTasks);
offsetLeft = floor(length(searchSeq)/2);
if mod(length(searchSeq),2) == 0
    offsetRight = offsetLeft - 1;
else
    offsetRight = offsetLeft;
end
startValues(2:end) = startValues(2:end) - offsetLeft;
endValues(1:end-1) = endValues(1:end-1) + offsetRight;
for tasknum = 1:numTasks
    createTask(job, @genematch, 2, {searchSeq, 'gene.txt', ...
        startValues(tasknum), endValues(tasknum)});
end
```

Example #2: Gene Matching (cont.)

```
submit(job); % Submit and Wait for Results
wait(job, 'finished');
results = fetchOutputs(job); % Report the results
results = cell2mat(results); % Return absolute position
[~,idx] = max(results(:,1));
bpm = results(idx,1);
msi = results(idx,2)+startValues(idx)-1;


function [startValues, endValues] = splitDataset(numTotalElements, numTasks)
numPerTask = repmat(floor(numTotalElements/numTasks), 1, numTasks);
leftover = rem(numTotalElements, numTasks);
numPerTask(1:leftover) = numPerTask(1:leftover) + 1;
endValues = cumsum(numPerTask);
startValues = [1 endValues(1:end-1) + 1];
```


Example #2: Gene Matching (cont.)

```
function [bestPctMatch,matchStartIdx]=genematch(searchSeq,file,startIdx,endIdx)
fid = fopen(file, 'rt');
geneSeq = fscanf(fid, '%c');
fclose(fid);
if nargin < 3, startIndex = 1; end
if nargin < 4, endIndex = length(geneSeq); end
[bestPctMatch,matchStartIdx]=findsubstr(geneSeq(startIdx:endIdx),searchSeq);

function [bestPctMatch,matchStartIdx]=findsubstr(baseString,searchString)
bestPctMatch = 0;
matchStartIdx = 0;
for startIdx = 1:(length(baseString)-length(searchString)+1)
    currentSection = baseString(startIdx:startIdx+length(searchString)-1);
    pctMatch = nnz(currentSection==searchString)/length(searchString);
    if pctMatch >= bestPctMatch
        bestPctMatch = pctMatch;
        matchStartIdx = startIdx;
    end
end
```

Lecture Outline (3)

- Parallel Computing Toolbox™ (PCT) and MATLAB® Distributed Computing Server™ (MDCS)
  - Starting a Parallel Pool using **`parpool`**
- Task-Parallel Jobs with PCT
  - Embarrassingly Parallel Tasks using **`parfor`**
  - Setting up a Simple Independent Job using **`createJob`**
- Data-Parallel Jobs with PCT
  - Single Program, Multiple Data using **`spmd`**
  - Distributed Datasets and Operations
  - Passing Messages
  - Setting up a Communicating Job

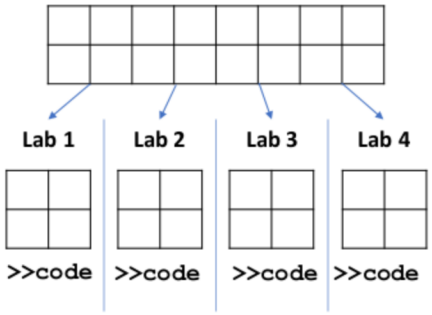
Data-Parallel Jobs with PCT

- Single Program, Multiple Data with **`spmd`**
  - Self-identification using **`labindex`** and **`numlabs`**
  - Types of arrays—replicated, variant, private
  - Composite data type
- Distributed Datasets and Operations
- Passing Messages
  - Practical Considerations
- Setting up Communicating Jobs using **`createCommunicatingJob`**
  - Assign one task to run on all labs
  - A lab can pass messages to other labs

Single Program, Multiple Data: **spmd**

```
>> spmd
>>    labindex
>>    numlabs
>> end
Lab 1:
       1
       4
Lab 2:
       2
       4
Lab 3:
       3
       4
Lab 4:
       4
       4
```

```
>> spmd
>>    code
>> end
```



Types of Arrays on Labs in SPMD

- Replicated Array

```
>> spmd
>>    x = 5;
>> end
```

- Variant Array

```
>> spmd
>>    y = rand;
>> end
```

- Private Array

```
>> spmd
>>    if (labindex==2)
>>        z = 7;
>>    end
>> end
```

Lab 1	x = 5
Lab 2	x = 5
Lab 3	x = 5
Lab 4	x = 5

Lab 1	y = 0.3246
Lab 2	y = 0.2646
Lab 3	y = 0.8847
Lab 4	y = 0.8939

Lab 1	
Lab 2	z = 7
Lab 3	
Lab 4	

Composite Class & Reductions

- From the MATLAB client, all these three types of arrays show up in the workspace as a Composite data type.
```
>> class(y)
```
  - Use the curly braces to extract the contents at any lab index.
```
>> y{3}
```
- Use a global operation to combine the results from all the labs. This performs a Reduction & Broadcast, which turns a Composite variable into a replicated array.
```
>> spmd
>>     ay = gcat(y);      % Global concatenation
>>     sy = gplus(y);     % Global summation
>>     my = gop(@max,y); % Global maximum
>> end
```
  - Or, specify a lab index to perform a Reduction and turn it into a private array:
```
>> spmd
>>     ay1 = gcat(y,1,1);
>>     sy1 = gplus(y,1);
>>     my1 = gop(@max,y,1);
>> end
```


Distributed Datasets and Operations

- Use Distributed Data Type from Client.
```
vars  = load('airportdata');
dlat  = distributed(vars.lat);
dlong = distributed(vars.long);
```
- OR: Use Codistributed Data Type from Labs.
```
spmd
    vars  = load('airportdata');
    clat  = codistributed(vars.lat);
    clong = codistributed(vars.long);
end
```
- Use **gather** function to convert to replicated array.
```
>> allat = gather(dlat);
```
  - Or, specify a lab index to convert to a private array.
```
>> allat1 = gather(dlat, 1);
```
- Also, use **getLocalPart** to convert to a variant array.
```
>> loclat = getLocalPart(dlat);
```

Example #3: Airport Distances

```
spmd
    R = 3963.2;  % in miles
    vars = load('airportdata');
    lat = codistributed(vars.lat);
    long = codistributed(vars.long);
    lat = 90 - lat;
    long = 360 + long;
    x = R * sind(lat) .* cosd(long);
    y = R * sind(lat) .* sind(long);
    z = R * cosd(lat);
    coords = [x y z]';
    dotprod = coords' * coords;
    mag = sqrt(sum(coords.^2));
    angles = min(dotprod ./ (mag' * mag), 1);
    dist = R * acos(angles);    % Arc length
end
```


Passing Messages

- To send a variable ***x*** to another lab:

```
>> labSend(x, dest_labindex);
```

- To receive a variable ***x*** from another lab:

```
>> x = labReceive(src_labindex);
```

- To see whether a lab is ready to receive data:

```
>> isReady = labProbe(dest_labindex);
```

- To broadcast a variable ***x*** to all other labs:

```
>> x = labBroadcast(src_lab,x);
```

- To synchronize all the labs:

```
>> labBarrier;
```

Passing Messages: Practical Considerations

```
spmd
    switch labindex
        case 1
            x1 = labindex * ones(1, 5); % Create local data
            x2 = labReceive(2);         % Receive data from peer
            labSend(x1, 2);             % Send data to peer
            y = x2;                     % Return peer's data
        case 2
            x2 = labindex * ones(1, 5); % Create local data
            x1 = labReceive(1);         % Receive data from peer
            labSend(x2, 1);             % Send data to peer
            y = x1;                     % Return peer's data
    end
end
```

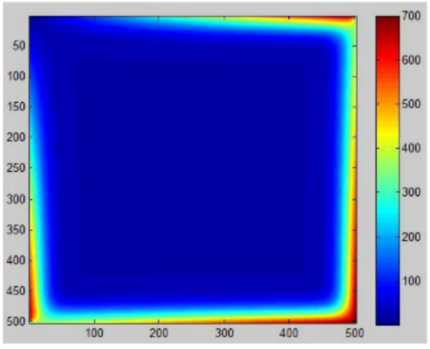
Lab 1 waits to receive

Lab 2 waits to receive

Deadlock! Use **`labSendReceive`** function instead to exchange data between labs.

```
spmd
    switch labindex
        case 1
            x1 = labindex * ones(1, 5);     % Create local data
            x2 = labSendReceive(2, 2, x1); % Exchange data with lab 2
            y = x2;                         % Return peer's data
        case 2
            x2 = labindex * ones(1, 5);     % Create local data
            x1 = labSendReceive(1, 1, x2); % Exchange data with lab 1
            y = x1;                         % Return peer's data
    end
end
```


Example #4: Parallel Heat Equation



- Get a matrix representing the temperature at each point of a 2D square plate of length L & diffusivity c.
- For example, solve for the temperature on a 3m-by-3m copper plate after 40 seconds have elapsed, using 500 time steps of 80 ms each. Thermal diffusivity of copper is 1.13e-4 m^2/s.

Example #4: Sequential Heat Equation

```
function U = heateq(k, n, Ts, L, c)
ms = L / n;
if Ts > (ms^2/2/c), error('Selected time step is too large.'); end
U = initialTempDistrib(n);
north   = 1:n;
south   = 3:(n + 2);
curr    = 2:(n + 1);
east    = 3:(n + 2);
west    = 1:n;
for iter = 1:k
    U(curr, curr) = U(curr, curr) + c * Ts/(ms^2) * (U(north, curr) + ...
        U(south, curr) - 4*U(current, curr) + U(curr, east) + U(curr, west));
end

function U = initialTempDistrib(n)
U = 23*ones(n + 2);
U(1, :) = (1:(n + 2))*700/(n + 2);
U(end, :) = ((1:(n + 2)) + (n + 2))*700/2/(n + 2);
U(:, 1) = (1:(n + 2))*700/(n + 2);
U(:, end) = ((1:(n + 2)) + (n + 2))*700/2/(n + 2);
```


Setting up Communicating Jobs

1. Connect to a cluster.
   ```
   >> cluster = parcluster('local');
   ```
2. Create a communicating job.
   ```
   >> job = createCommunicatingJob(cluster,'Type','SPMD');
   ```
3. Create a repeating task for the job to handle.
   ```
   >> createTask(job,@parheateqn,1,...
        {1e3,500,0.08,3,1.13e-4});
   ```
4. Set a range for the number of workers needed.
   ```
   >> set(job,'NumWorkersRange',[3 3]);
   ```
5. Submit the job. Optionally, wait for it to finish.
   ```
   >> submit(job); wait(job,'finished');
   ```
6. Retrieve the results.
   ```
   >> results = fetchOutputs(job);
   >> U = cell2mat(results');
   >> imagesc(U)
   ```
7. When finished, clean up. Delete job & clear object.
   ```
   >> delete(job); clear job;
   ```

Example #4: Parallel Heat Equation

```
function U = parheateqn(k, n, Ts, L, c)
ms = L / n;
if (Ts>(ms^2/2/c)), error('Selected time step is too large.'); end
Uinit = initialTempDistrib(n);
parts = codistributor1d.defaultPartition(n+2);
numLocalCols = parts(labindex);
leftColInd = sum(parts(1:labindex - 1)) + 1;
rightColInd = leftColInd + numLocalCols - 1;
U = Uinit(:, leftColInd:rightColInd);
if (labindex > 1),       U = [zeros(n+2, 1) U]; end
if (labindex < numlabs), U = [U zeros(n+2, 1)]; end
if (labindex == 1) || (labindex == numlabs)
    numLocalCols = numLocalCols - 1;
end
rightNeighbor = mod(labindex, numlabs) + 1;
leftNeighbor  = mod(labindex - 2, numlabs) + 1;
north     = 1:n;
south     = 3:n + 2;
currRow   = 2:n + 1;
currCol   = 2:numLocalCols + 1;
east      = 3:numLocalCols + 2;
west      = 1:numLocalCols;
```


Example #4: Parallel Heat Equation (cont.)

```
for iter = 1:k
    rightBoundary = labSendReceive(leftNeighbor,rightNeighbor,U(:,2));
    leftBoundary = labSendReceive(rightNeighbor,leftNeighbor,U(:,end-1));
    if (labindex > 1),       U(:, 1) = leftBoundary;    end
    if (labindex < numlabs), U(:, end) = rightBoundary; end
    % Update grid for current iteration
    U(currRow,currCol) = U(currRow,currCol) + ...
        c*Ts/(ms^2)*(U(north,currCol) + U(south,currCol) - ...
        4*U(currRow,currCol) + U(currRow,east) + U(currRow,west));
end
% Combine parts from all labs into a single matrix stored on lab 1
U = gcat(U(currRow, currCol), 2, 1);

function U = initialTempDistrib(n)
U = 23*ones(n + 2);
U(1, :) = (1:(n + 2))*700/(n + 2);
U(end, :) = ((1:(n + 2)) + (n + 2))*700/2/(n + 2);
U(:, 1) = (1:(n + 2))*700/(n + 2);
U(:, end) = ((1:(n + 2)) + (n + 2))*700/2/(n + 2);
```

Summary: Problem Types

	Interactive	Batch
Task-Parallel	parpool **parfor**	createJob createTask
Data-Parallel	parpool **spmd**	createCommunicatingJob createTask