MATLAB HPC Mentors
June 25, 2020

Host: Nicholas Ide
Guest Speakers: Ben Tordoff & Oli Tissot, Parallel Computing Development
Quick Updates

- No meeting in July or August – enjoy the summer, stay healthy!
- If you have issues with recent SSH and MATLAB Parallel Server, let me know.
  - Have been a few reports
  - Members of the community have found temporary solutions
Distributed Arrays: techniques and best practices for handling very large calculations

Ben Tordoff & Oli Tissot, Parallel Computing Development
What are distributed arrays?
- What can you do with distributed arrays?
- Building a distributed array
- Advanced manoeuvres
- Debugging methods and tips
Remote arrays in MATLAB

Rule: take the calculation to where the data is

Normal array – calculation happens in main memory:

\[
x = \text{rand}(\ldots) \\
x_{\text{norm}} = (x - \text{mean}(x)) / \text{std}(x)
\]
Remote arrays in MATLAB

Rule: take the calculation to where the data is

gpuArray – all calculation happens on the GPU:

```matlab
x = gpuArray(...)
x_norm = (x - mean(x)) ./ std(x)
```
distributed – calculation is spread across the memory of a cluster:

```matlab
x = distributed(...)
x_norm = (x - mean(x)) ./ std(x)
```
tall – calculation is performed by stepping through files:

```matlab
x = tall(...)
x_norm = (x - mean(x)) ./ std(x)
```
When should I reach for distributed?

- If your data fits in memory, just use MATLAB normally

- If it won’t fit on one machine, maybe it can be split across the combined memory of a cluster of machines? Use distributed arrays

- If it won’t fit in the combined memory of a cluster of machines then use tall arrays
Using distributed arrays

- Use the memory of multiple machines as though it was your own
- Client sees a “normal” MATLAB variable
- Work happens on cluster

```matlab
>> d = distributed(...);
```
Distributed Algorithms

$$m = \text{mean}(d);$$
• What are distributed arrays?

What can you do with distributed arrays?
• Building a distributed array
• Advanced manoeuvres
• Debugging methods and tips
What can you do with distributed arrays?

Distributed array functions

- Extensive support:
  - Includes most linear algebra
  - Scale up mathematical operations

Run MATLAB functions with distributed arrays
What can you do with distributed arrays?

- >500 functions supported (as of R2020a)
- support for dense and sparse linear algebra
- support for numerics (double, single, logical, etc.)
- support for datetimes, durations, categoricals, tables, …

- focus is on data preparation and large system solve
What can you do with distributed arrays?

Code written for distributed arrays looks like normal MATLAB code

```matlab
% Create / read distributed data
A = distributed(...)

% Same code as for in-memory
b = sum(A,2);
x1 = A\b; % direct solution
x2 = pcg(A, b); % iterative solution

% Bring back from cluster
[x1,x2] = gather(x1, x2);
```
## Processing quite big data

### Multiplication of 2 NxN matrices

$$\text{>> } C = A \ast B$$

<table>
<thead>
<tr>
<th>$N$</th>
<th>1 node, 16 workers</th>
<th>2 nodes, 32 workers</th>
<th>4 nodes, 64 workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>8000</td>
<td>19</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>16000</td>
<td>120</td>
<td>75</td>
<td>50</td>
</tr>
<tr>
<td>20000</td>
<td>225</td>
<td>132</td>
<td>86</td>
</tr>
<tr>
<td>25000</td>
<td>-</td>
<td>243</td>
<td>154</td>
</tr>
<tr>
<td>30000</td>
<td>-</td>
<td>406</td>
<td>248</td>
</tr>
<tr>
<td>35000</td>
<td>-</td>
<td>-</td>
<td>376</td>
</tr>
<tr>
<td>45000</td>
<td>-</td>
<td>-</td>
<td>743</td>
</tr>
<tr>
<td>50000</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Processor: Intel Xeon E5-class v2
16 cores, 60 GB RAM per compute node, 10 Gb Ethernet

![Graph showing execution time for different $N$ values](image)

*Processing quite big data is crucial in various fields, such as machine learning and big data analytics, where the size of the data can affect the performance of algorithms. This image demonstrates how the execution time for multiplying two $N \times N$ matrices changes with the size of the matrices and the number of workers.*
Agenda

• What are distributed arrays?
• What can you do with distributed arrays?

Building a distributed array

• Advanced manoeuvres
• Debugging methods and tips
Building a distributed array

There are four main ways to build a distributed array:

1. Create from in-memory data
2. Build functions
3. Read from datastore
4. Construct from local parts
Building a distributed array: from in-memory

```matlab
>> x = gallery("poisson",10000);
>> dx = distributed(x); % Data sent to workers
```

- All data is sent from client to workers
- Useful for debugging before scaling up
- Useful for data that is close to filling local memory (i.e. can be created but not operated on due to fill-in etc.)
Building a distributed array: build functions

```matlab
>> dx = distributed.ones(1e9,100);
>> dx = distributed.rand(1e9,100);
>> dx = distributed.speye(1e9);
>> dx = distributed.sprand(1e9,1e9,1e-8);
... etc.
```

- No data is sent from client to workers
- Useful for creating test data and examples
Building a distributed array: build from datastore

Datastore:

- Simple interface for data in multiple files/folders
- Presents data a piece at a time
- Access pieces in serial (desktop) or in parallel (cluster)
- Back-ends for tabular text, images, databases and more
- Data always stacked vertically

```matlab
>> ds = datastore("data/*.csv")
>> preview(ds)
    Rows     Cols     Vals
    _____    ____    ____
        1     1        4
        2     1       -1
       10001     1       -1
        :     :        :
        :     :        :
```
Building a distributed array: build from datastore

Read datastore into distributed
- each worker reads its own part of the data
- data is distributed vertically (workers have blocks of rows)

```
>> dt = distributed(ds); % distributed table of [Rows, Cols, Vals]
>> d = sparse(dt.Rows, dt.Cols, dt.Vals);

>> size(d)
ans = 100000000  100000000

>> nnz(d)
ans = 499960000
```
What are distributed arrays?
What can you do with distributed arrays?
Building a distributed array
Advanced manoeuvres
Debugging methods and tips
Writing your own algorithms

SPMD let’s you craft parallel algorithms:

- Inside SPMD distributed -> codistributed
- `getLocalPart` extracts the data for this worker
- Use `gop` to reduce across workers (binary tree)
- Use `gcat` to concatenate results across workers

```
>> d = distributed(...)
>> spmd
    S = sum(getLocalPart(d),"all");  % Sum values on this worker
    totalSum = gop(@plus, S);  % Add results across all workers
end
```
Writing your own algorithms

SPMD let’s you craft parallel algorithms:

- **MPI-like** `labSend`, `labReceive`, `labSendReceive` for low level communication
- `labindex` for working out which worker you are

```MATLAB
>> d = distributed(...)
>> spmd

    mydata = getLocalPart(d);
    % Cycle data to the right (wrapping round on last worker)
    prevWorker = mod(labindex-2, numlabs)+1; % worker to the left
    nextWorker = mod(labindex, numlabs)+1; % worker to the right
    mydata = labSendReceive(nextWorker, prevWorker, mydata)
end
```
Modifying the data distribution

SPMD also gives control over distribution:

- `redistribute` for changing the distribution of an existing array
- `codistributorXX` for creating new (co)distributed arrays

```matlab
>> d = distributed.ones(1e5);  % Default is 1D distribution in dim 2
>> spmd
    % Switch to having whole rows per worker
    d2 = redistribute(d, codistributor1d(1));
    % Use a custom (uneven) distribution of the data
    partition = 1e4 * [1 2 3 4];
    d3 = redistribute(d, codistributor1d(1, partition));
    % Switch to block-cyclic
    d4 = redistribute(d, codistributor2dbc());
end
```
Agenda

• What are distributed arrays?
• What can you do with distributed arrays?
• Building a distributed array
• Advanced manoeuvres

Debugging methods and tips
Debugging

Some rules of thumb:

- Start with “local” pool and small data
  - faster iteration time
  - proves correctness
- Use pre-built algorithms if you can
- If writing your own, prefer higher-level gop, gcat over labSend/Receive
  - It’s easy to create mismatched communications with labSend/Receive!
  - Using unique tags for each communication helps to spot stray messages
  - Keeping computation and communication code separate helps in debugging both
- Run for a few different sizes to understand how duration scales with size
- Think about where your data lives – minimize transmission of files across networks
Debugging

Use parallel profiler (mpiprofiler):
- Shows what code ran, how much data transferred and allows comparison between workers (spot uneven loading)

```
>> mpiprofiler on
>> % Lots of parallel code ...
>> mpiprofview
```
Debugging

Use spmd or parfevalonall to interact with workers and see local state

```
>> spmd, d, end

Lab 1: This worker stores d(:,1:1667).
   LocalPart: [10000x1667 double]
   Codistributor: [1x1 codistributor1d]

Lab 2: This worker stores d(:,1668:3334).
   LocalPart: [10000x1667 double]
   Codistributor: [1x1 codistributor1d]

Lab 3:
   etc.
```
Questions