MATLAB for High Performance Computing

Shaohao Chen
Research Computing Services
Information Services and Technology
Boston University
Why using Matlab for HPC?

- Many Matlab programs run faster on high-performance computing (HPC) clusters than on regular laptops/desktops.
- Boston University (BU) Shared Computing Cluster (SCC) is an HPC cluster with over 11,000 CPU cores and over 250 GPUs.
- MATLAB site license is available to all BU users. (Unlimited on SCC).

- Matlab programs can be exceptionally fast if they are well-designed, and painfully slow if not. It is necessary to optimize MATLAB codes to obtain good performance.
- The code-optimized methods covered in this tutorial can be used not only on HPC clusters but also on regular laptops/desktops.

- The skills you learn today should enable you to solve bigger problems faster using MATLAB.
Outline

✧ Use Matlab on BU SCC
  ✓ Start up
  ✓ Submit Matlab jobs
  ✓ Distributed jobs

✧ Optimize Matlab codes
  ✓ Remove unnecessary work
  ✓ Optimize memory usage
  ✓ Use optimized built-in functions/operators
Access to BU SCC resources

- **Log in to SCC:**
  
  $ ssh -X username@scc1.bu.edu

- **Interactive session:** working interactively on compute nodes.
  
  $ qrsh  # Start an interactive session

  - Any job running for more than 15-minute CPU time on login nodes will be killed.

- **Load Matlab module:**
  
  $ moduleavail | grep matlab  # See all available versions

  $ module load matlab  # Set up environment variables
Open Matlab

$ matlab &

Use VNC to speed up graphical interface.

Refer to: https://www.bu.edu/tech/support/research/system-usage/getting-started/remote-desktop-vnc/
M-file

- An m-file is a simple text file where you can place MATLAB commands.
- Save your works
- Convenient for debugging
- Run directly. Pre-compiling is unnecessary.
Text platform

$ matlab -nodisplay % Work in text interface. Does not display any graph.
$ matlab -nodesktop % Program in text interface. Pop out graphs when necessary.

- Many Linux commands (some prefixed with an exclamation mark) are available within Matlab platform, such as:
  - cd, ls, pwd, !cp, !rm, !mv, !cat, !vim, !diff, and !grep

- Edit M-file and run the program:
  - >> !vim mfilename.m % edit in text window
  - >> edit mfilename.m % create a new or open an existing m-file in graphical window
  - >> open mfilename.m % open an existing m-file in graphical window
  - >> run mfilename.m % run the program
  - >> mfilename % run the program
Submit a batch job to background

- Submit a batch job to background using a script
  
  `$ qsub script.sh`

- Write a batch job script using any Linux editor (such as vim, emacs, gedit, or nano).

- A typical batch script for a Matlab job (Supposed that source codes are saved in an m-file):

  ```bash
 #!/bin/bash -l  # Start a bash script. The option -l is necessary to enable module tool.
  #SBATCH -pe omp 1    # Request 1 CPU core
  #SBATCH -l h_rt=12:00:00  # Request wall time
  #SBATCH -N jobname    # Give a job name
  module load matlab  # Set up environment variables for running matlab
  matlab -nodisplay -singleCompThread -r "addpath /path/to/work; mfile_name"  # Run program
  ```
Exercise 1

◊ Run a Matlab program on BU SCC:

  Request for an interactive session first.

  i) Write a Matlab code in an m-file to print “Hello world” (e.g. use the disp function).

  ii) Run the program interactively in Matlab platform.

  iii) Submit a batch job to run the program in the background.
Standalone executable

◊ Create a standalone executable

$ mcc -mv -o myexe name.m

✓ -mv produces a standalone and shows actions taken
✓ -o myexe specifies the executable name
✓ The name of the m file must be the same as the main function name in it.
✓ A script named run_myexe.sh is automatically created for running jobs in Linux bash shell.

◊ Run the executable (in Linux shell environment)

$ module load mcr  # Set up environment variables for running the executable.
$ mcr ./myexe     # Execute the executable using MATLAB Compiler Runtime (MCR).

✓ There is no considerable performance difference between running a standalone executable and running an m-file directly. The later one is more recommended since it is simpler.
Distributed jobs

- Submit multiple jobs using one command line.
- **Distribute independent jobs:**

  $ qsub -t 1-4 script.sh

  - The batch system sets up the `SGE_TASK_ID` environment variable which can be used to pass the task ID to the program, for example:

    ```
    matlab -nodisplay -singleCompThread -r "rand($SGE_TASK_ID); exit"
    ```

    ```
    matlab -nodisplay -singleCompThread -r "id='$SGE_TASK_ID'; rand(id); exit"
    ```

    ```
    matlab -nodisplay -singleCompThread -r "id=getenv('SGE_TASK_ID'); rand(id); exit"
    ```

- **Distribute dependent jobs:**

  $ qsub -N job1 script1.sh

  $ qsub -N job2 -hold_jid job1 script2.sh

  - Job 2 does not start until job1 ends.
Optimize Matlab codes

- Tools for measuring performance and optimizing codes
- Remove unnecessary work
- Optimize memory usage
- Use built-in functions/operators
Tools for measuring performance and optimizing codes

>> tic  % Start measuring time
>> toc  % End measuring time
>> timeit (function_name)  % Measure time required to run function
>> mlint ('mfile_name')  % Reports potential problems and opportunities for code optimization
>> profile on  % Turn on profile (before the program starts).
>> profile viewer  % View the results in the Profiler window (after the program ends).
Unnecessary work (1): remove redundant operations

◊ Avoid redundant operations in loops

```plaintext
bad
for i=1:N
  x = 10;
  .
  .
end
```

```plaintext
good
x = 10;
for i=1:N
  .
  .
end
```
Unnecessary work (2): reduce overhead costs

..from function calls

```plaintext
function myfunc(i)
    % do stuff
end

for i=1:N
    myfunc(i);
end
```

**bad**

```plaintext
function myfunc2(N)
    for i=1:N
        % do stuff
    end
end

myfunc2(N);
```

**good**

..from loops

```plaintext
for i=1:N
    x(i) = i;
end

for i=1:N
    y(i) = rand();
end
```

**bad**

```plaintext
for i=1:N
    x(i) = i;
    y(i) = rand();
end
```

**good**
Unnecessary work (3): optimize logical tests

Avoid unnecessary logical tests...

...by using short-circuit logical operators

bad

if (i == 1 | j == 2) & k == 5
  % do something
end

...by moving known cases out of loops

bad

for i=1:N
  if i == 1
    % i=1 case
  else
    % i>1 case
  end
end

good

if (i == 1 || j == 2) && k == 5
  % do something
end

good

% i=1 case
for i=2:N
  % i>1 case
end
Unnecessary work (4): reorganize equations

Reorganize equations to use fewer or more efficient operators

Basic operators have different speeds:

- Add: 3–6 cycles
- Multiply: 4–8 cycles
- Divide: 32–45 cycles
- Power, etc: worse

```
c = 4;
for i=1:N
  x(i) = y(i)/c;
  v(i) = x(i) + x(i)^2 + x(i)^3;
  z(i) = log(x(i)) * log(y(i));
end
```

```
s = 1/4;
for i=1:N
  x(i) = y(i)*s;
  v(i) = x(i)*(1+x(i)*(1+x(i)));
  z(i) = log(x(i) + y(i));
end
```
MATLAB improves performance by interpreting a program only once, unless you tell it to forget that work by running “clear all”.

<table>
<thead>
<tr>
<th>Value of Item Type</th>
<th>Variables in scope</th>
<th>Scripts and functions</th>
<th>Class definitions</th>
<th>Persistent variables</th>
<th>MEX functions</th>
<th>Global variables</th>
<th>Import list</th>
<th>Java classes on the dynamic path</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>From command prompt only</td>
</tr>
<tr>
<td>variables</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory (1): preallocate arrays

* Arrays are always allocated in contiguous address space.
* If an array changes size, and runs out of contiguous space, it must be moved. For example,

\[
\begin{align*}
x &= 1; \\
\text{for } i &= 2:4 \\
& \quad x(i) = i; \\
& \text{end}
\end{align*}
\]

* This can be very very bad for performance when variables become large.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x(1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2000</td>
<td>x(1)</td>
</tr>
<tr>
<td>2001</td>
<td>x(2)</td>
</tr>
<tr>
<td>2002</td>
<td>x(1)</td>
</tr>
<tr>
<td>2003</td>
<td>x(2)</td>
</tr>
<tr>
<td>2004</td>
<td>x(3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10004</td>
<td>x(1)</td>
</tr>
<tr>
<td>10005</td>
<td>x(2)</td>
</tr>
<tr>
<td>10006</td>
<td>x(3)</td>
</tr>
<tr>
<td>10007</td>
<td>x(4)</td>
</tr>
</tbody>
</table>
Memory (1): preallocate arrays

◊ Preallocating array to its maximum size prevents intermediate array movement and copying.

\[
A = \text{zeros}(n,m); \quad \% \text{ initialize } A \text{ to 0}
A(n,m) = 0; \quad \% \text{ or touch largest element}
\]

◊ If maximum size is not known, estimate with upper bound. Remove unused memory after.

\[
A = \text{rand}(100,100);
\% \ldots
\% \text{ if final size is } 60 \times 40, \text{ remove unused portion}
A(61:end,:) = []; A(:,41:end) = []; \quad \% \text{ delete}
\]
Memory (2): organize for-loop order

- It is faster to access continuous memory addresses than separated ones.
- Multidimensional arrays are stored in memory along columns (column-major).
- Best if inner-most loop is for array left-most index, etc.

```
bad
n=5000; x = zeros(n);
for i = 1:n    % rows
    for j = 1:n    % columns
        x(i,j) = i+(j-1)*n;
    end
end

good
n=5000; x = zeros(n);
for j = 1:n    % columns
    for i = 1:n    % rows
        x(i,j) = i+(j-1)*n;
    end
end
```
Memory (3): avoid unnecessary variables

- Avoid time needed to allocate and write data to main memory.
- Compute and save array in-place improves performance and reduces memory usage.

```plaintext
bad

x = rand(5000);
y = x.^2;

```

```plaintext
good

x = rand(5000);
x = x.^2;
```

Memory (4): Use sparse matrix

- Sparse matrix: many zero elements. Save only non-zero elements in memory.
- Syntax: \( S = \text{sparse}(i, j, v, m, n) \)

```
n=10    % Matrix size
i=[1, 3, 3, 6, 8, 10];    % Row index for insertion
j=[2, 4, 4, 5, 7, 9];     % Column index for insertion
v=[0.1, 0.3, 0.2, 0.8, 0.6, 0.7]; % Values to be inserted.
S1=\text{sparse}(i, j, v, n, n)    % Create a sparse matrix
S2=\text{sparse}(1:6, 2:7, \text{rand}, n, n) % Create another sparse matrix
S3=S1*S2    % The produce to two sparse matrices is still a sparse matrix
\text{size}(S3) % Output the size of the sparse matrix
```
Exercise 2

◊ Optimize this Matlab code to obtain a better performance.

✓ Hint: Use mlint to report potential problems and opportunities for code optimization.

```matlab
n=5000;
for i=1:n
    for j=1:n
        a=3./2.;
        if (i==1)
            x(i, j) = 5.;
        else
            x(i, j) = i * log(2.) * log(a) + j^2 / 2.;
        end
    end
end
y=x.^2;
```
Matlab built-in functions/operators

Some useful Matlab built-in functions/operators:

- Matrix operations: *, mtimes, inv, eig
- Solve linear equation: mldivide, linsolve, \n- Decomposition: lu, qr
- Optimization: fminsearch, fzero

A full list:

https://www.mathworks.com/help/matlab/functionlist.html#linear-algebra

Built-in functions/operators are optimized and performs well in general.
Matrix multiplication

The built-in operator * has been optimized and has much better performance.

```matlab
n = 500
for j=1:1:n % initialize data
    for i=1:1:n
        A(i,j)=i+j; B(i,j)=2*i-j;
    end
end
C=zeros(n); D=zeros(n);
tic
for i=1:1:n % matrix multiplication by loops
    for j=1:1:n
        C(i,j) = A(i,:)*B(:,j);
    end
end
toc
tic
D = A * B; % matrix multiplication by built-in operator
toc
isequal(C,D) % Check results
```
The backslash operator: solve linear equations

- Solve the linear system $A\times x = b$
  
  $$x = A\backslash b$$
  
  $$x = \text{mldivide}(A,b)$$
  
  $$x = \text{linsolve}(A,b)$$
  
  $$x = \text{linsolve}(A,b,\text{opts})$$

  $$A = \text{triu} (\text{rand}(5,5)); \quad \% \text{random 5*5 up-triangle matrix}$$
  
  $$b = \text{rand}(5,1); \quad \% \text{random column array}$$
  
  $$\text{opts.UT} = \text{true}; \quad \% \text{up-triangle is true}$$
  
  $$x = \text{linsolve}(A,b,\text{opts})$$
  
  $$x = A\backslash b \quad \% \text{What does the backslash operator actually do behind the scene?}$$
Decomposition

◊ **LU decomposition**

\[ [L, U] = \text{lu}(A) \] \% expresses a matrix \( A \) as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The decomposition is often called the LU, or sometimes the LR, decomposition.

\[ [L, U] = \text{lu}(A); \] \% obtain L and U matrices
\[ y = L \backslash b; \]
\[ x = U \backslash y \] \% These 3 lines together are equivalent to \( x = A \backslash b \)

◊ **QR decomposition**

\[ [Q, R] = \text{qr}(A) \] \% expresses a matrix \( A \) into a product \( A = QR \) of an orthogonal matrix \( Q \) and an upper triangular matrix \( R \).

\[ [Q, R] = \text{qr}(A); \] \% obtain Q and R matrices
\[ y = Q^\top \ast b; \]
\[ x = R \backslash y \]
Exercises 3

◊ Exercise 3.1: Create an n*n symmetric matrix A in one of the following ways:
  i) use built-in function \texttt{triu} (up-triangle matrix) and the matrix transpose operation \texttt{'};
  ii) use control flow (\texttt{for, if, else, ...}).

◊ Exercise 3.2: Given a symmetric matrix A from exercise 3.1, solve the linear algebra equation $A\times x = b$ using the following two methods:
  i) use the matrix inverse function \texttt{inv} .
  ii) use the backslash operator \texttt{	extbackslash} .

Compare the computational time and numerical error of the two cases.
Further Information

◊ MathWorks Web:
  ✔ MATLAB documentation: http://www.mathworks.com/help/matlab/

◊ BU Research Computing Services (RCS) Web:
  ✔ MATLAB basics: http://www.bu.edu/tech/support/research/software-and-programming/common-languages/matlab/

◊ A book: Accelerating MATLAB Performance: 1001 tips to speed up MATLAB programs by Yair M. Altman.

◊ RCS help: help@scc.bu.edu, shaohao@bu.edu