UL HPC School 2017

PS8b: Advanced MATLAB (checkpointing and parallel jobs)

UL High Performance Computing (HPC) Team

V. Plugaru

University of Luxembourg (UL), Luxembourg

http://hpc.uni.lu

Latest versions available on Github:

UL HPC tutorials: https://github.com/ULHPC/tutorials

UL HPC School: http://hpc.uni.lu/hpc-school/

PS8b tutorial sources: https://github.com/ULHPC/tutorials/tree/devel/advanced/MATLAB1

Summary

1 **Pre-requisites**

2 Objectives

3 Checkpointing
Example 1 revisited

4 Parallelization
Example 2 revisited

5 Conclusion

Tutorial files

Sample MATLAB scripts used in the tutorial

- download only the scripts:

```
(frontend)$> mkdir $HOME/matlab-tutorial2
(frontend)$> cd $HOME/matlab-tutorial2
(frontend)$> wget
https://raw.github.com/ULHPC/tutorials/devel/advanced/MATLAB2/code/example1.m
(frontend)$> wget
https://raw.github.com/ULHPC/tutorials/devel/advanced/MATLAB2/code/example2.m
(frontend)$> wget
https://raw.github.com/ULHPC/tutorials/devel/advanced/MATLAB2/code/google_finance_data.m
```

- *or* download the full repository and link to the MATLAB tutorial:

```
(frontend)$> git clone https://github.com/ULHPC/tutorials.git
(frontend)$> ln -s tutorials/advanced/MATLAB2/
$HOME/matlab-tutorial2
```

X Window System

In order to see locally the MATLAB graphical interface,
a package providing the X Window System is required:

- on OS X: **XQuartz** `http://xquartz.macosforge.org/landing/`
- on Windows: **VcXsrv** `http://sourceforge.net/projects/vcxsrv/`

Now you will be able to connect with X11 forwarding enabled:

- on Linux & OS X:

  $> `ssh access-gaia.uni.lu` **-X**

- on Windows, with Putty
  Connection → SSH → X11 → **Enable X11 forwarding**

Summary

1 Pre-requisites

2 Objectives

3 Checkpointing
Example 1 revisited

4 Parallelization
Example 2 revisited

5 Conclusion

Objectives of this PS

Better understand the usage of MATLAB on the UL HPC Platform

- application-level checkpointing
  - ↪ using in-built MATLAB functions

Objectives of this PS

Better understand the usage of MATLAB on the UL HPC Platform

- application-level checkpointing
  - ↪ using in-built MATLAB functions

- taking advantage of some parallelization capabilities
  - ↪ use of **parfor**
  - ↪ use of GPU-enabled functions

Objectives of this PS

Better understand the usage of MATLAB on the UL HPC Platform

- application-level checkpointing
  - ↪ using in-built MATLAB functions

- taking advantage of some parallelization capabilities
  - ↪ use of **parfor**
  - ↪ use of GPU-enabled functions

- adapting the parallel code with checkpoint/restart features

Summary

1 Pre-requisites

2 Objectives

3 **Checkpointing**
   Example 1 revisited

4 Parallelization
   Example 2 revisited

5 Conclusion

Checkpointing

What is it?

Technique for adding fault tolerance to your application.
You adapt your code to (regularly) save a snapshot of the environment (workspace), and restart execution from the snapshot in case of failure.

Checkpointing

What is it?

Technique for adding fault tolerance to your application.
You adapt your code to (regularly) save a snapshot of the environment (workspace), and restart execution from the snapshot in case of failure.

Why make the effort to checkpoint?

- because your code may take longer to execute than the maximum walltime allowed
- because losing (precious) hours or days of computation **when** something fails may (should!) not be acceptable

Checkpointing pitfalls

- checkpointing (too) often can be counterproductive
  - ↪ saving state in each loop may take longer than its actual computing time
  - ↪ saving state incrementally can lead to fast exhaustion of your $HOME space
  - ↪ in extreme cases can lead to platform instability – especially if running parallel jobs!

Checkpointing pitfalls

- checkpointing (too) often can be counterproductive
  - ↪ saving state in each loop may take longer than its actual computing time
  - ↪ saving state incrementally can lead to fast exhaustion of your $HOME space
  - ↪ in extreme cases can lead to platform instability – especially if running parallel jobs!

- checkpointing (especially parallel) code can be tricky
- extra-care required if checkpointing simulations involving RNG (e.g. Monte Carlo-based experiments)
- ensure results consistency after you add checkpointing

Checkpointing basics

1. Check that a checkpoint file exists: `exist('save.mat','file')`
2. If it exists, restore workspace data from it: `load('save.mat')`

Checkpointing basics

1. Check that a checkpoint file exists: `exist('save.mat','file')`
2. If it exists, restore workspace data from it: `load('save.mat')`
3. During computing steps, use control variables to direct (re)start of computation

Checkpointing basics

1. Check that a checkpoint file exists: `exist('save.mat','file')`
2. If it exists, restore workspace data from it: `load('save.mat')`
3. During computing steps, use control variables to direct (re)start of computation
4. Every *n* loops, or if execution time (in loop or since startup) is above threshold, checkpoint:
   - ↪ save full workspace state: `save('save.tmp')`
   - ↪ save partial state: `save('save.tmp', 'var1', 'var2')`

Checkpointing basics

1. Check that a checkpoint file exists: `exist('save.mat','file')`
2. If it exists, restore workspace data from it: `load('save.mat')`
3. During computing steps, use control variables to direct (re)start of computation
4. Every *n* loops, or if execution time (in loop or since startup) is above threshold, checkpoint:
   - ↪ save full workspace state: `save('save.tmp')`
   - ↪ save partial state: `save('save.tmp', 'var1', 'var2')`
5. Rename state file to final name: `system('mv save.tmp save.mat')`
   - ↪ this process ensures that in case of failure during checkpointing, next execution doesn't try to restart from incomplete state

When to trigger checkpointing?

- when (loop) execution time is above threshold (e.g. 1h):
  - ↪ use `tic` and `toc` stopwatch functions, remember they can be assigned to variables
  - ↪ use the `clock` function
  - ↪ add some **randomness** to the threshold if you run several instances in parallel!

When to trigger checkpointing?

- when (loop) execution time is above threshold (e.g. 1h):
  - ↪ use `tic` and `toc` stopwatch functions, remember they can be assigned to variables
  - ↪ use the `clock` function
  - ↪ add some **randomness** to the threshold if you run several instances in parallel!
- every *n* loop executions
  - ↪ remember that saving state takes time, depending on workspace size & shared filesystem usage, and
  - ↪ if loops finish fast your code may be slowed down considerably
  - ↪ add some **randomness** to *n* if you run several instances in parallel!

Adding checkpointing to seq. code

example1.m: non-interactive script that shows:

- the use of a stopwatch timer
- how to use an external function (financial data retrieval)
- how to use different plotting methods
- how to export the plots in different graphic formats

Tasks to tackle with checkpointing

- modify the script to download data for Fortune100 companies
- add & test checkpointing to save state after each company's data is downloaded
- more granular downloads - modify download period from 1 year to 1 month, add & test checkpointing to save state after each download

Summary

1 Pre-requisites

2 Objectives

3 Checkpointing
Example 1 revisited

4 **Parallelization**
Example 2 revisited

5 Conclusion

Reference documentation

- **Parallel Computing Toolbox** http://www.mathworks.nl/help/distcomp/index.html
- Parallel for-Loops (parfor)
  http://www.mathworks.nl/help/distcomp/getting-started-with-parfor.html
- GPU Computing http://www.mathworks.nl/discovery/matlab-gpu.html

Accelerate the time to result

Option 1: Split input over several parallel, independent, MATLAB jobs

$\hookrightarrow$ great **if** it's possible (embarrassingly parallel problem)

Accelerate the time to result

Option 1: Split input over several parallel, independent, MATLAB jobs

↪ great **if** it's possible (embarrassingly parallel problem)

Option 2: Use **parfor** to execute loop iterations in parallel

↪ single node **only**

↪ we have 120 & 160 core nodes on which big problems can be tackled

Accelerate the time to result

Option 1: Split input over several parallel, independent, MATLAB jobs

- ↪ great **if** it's possible (embarrassingly parallel problem)

Option 2: Use **parfor** to execute loop iterations in parallel

- ↪ single node **only**
- ↪ we have 120 & 160 core nodes on which big problems can be tackled

Option 3: Use GPU-enabled functions that work on the **gpuArray** data type

- ↪ **require** the code to be run on GPU nodes (subset of Gaia)
- ↪ **great speedup** for some workloads
- ↪ **295** in-built MATLAB functions work on gpuArray

  including discrete Fourier transform, matrix multiplication, left matrix division

Accelerate the time to result

Option 1: Split input over several parallel, independent, MATLAB jobs

↪ great **if** it's possible (embarrassingly parallel problem)

Option 2: Use **parfor** to execute loop iterations in parallel

↪ single node **only**

↪ we have 120 & 160 core nodes on which big problems can be tackled

Option 3: Use GPU-enabled functions that work on the **gpuArray** data type

↪ **require** the code to be run on GPU nodes (subset of Gaia)

↪ **great speedup** for some workloads

↪ **295** in-built MATLAB functions work on gpuArray

including discrete Fourier transform, matrix multiplication, left matrix division

Option 4: MATLAB Distributed Computing Server (MDCS)

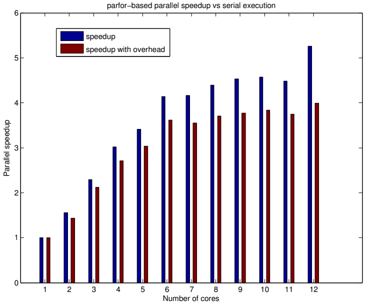
↪ allows multi-node parallel execution

↪ **not yet** part of the UL MATLAB license

Speed up your seq. code

example2.m: non-interactive script that shows:

- the serial execution of time consuming operations
  - ↪ the parallel execution and relative speedup vs serial execution
  - ↪ setting the # of parallel threads through environment variables
  - ↪ GPU-based parallel execution



Speed up your seq. code

example2.m: non-interactive script that shows:

- the serial execution of time consuming operations
  - ↪ the parallel execution and relative speedup vs serial execution
  - ↪ setting the # of parallel threads through environment variables
  - ↪ GPU-based parallel execution

Tasks to tackle

- execute the script on regular vs GPU nodes (with different GPUs)
- increase # of iterations, matrix size
- increase # of workers with/without changing the # of requested cores
- modify the script with other GPU-enabled functions

Summary

1 Pre-requisites

2 Objectives

3 Checkpointing
Example 1 revisited

4 Parallelization
Example 2 revisited

5 Conclusion

What we've seen in this session

- Checkpointing basics
- Specific MATLAB instructions for checkpointing
- Current MATLAB parallelization capabilities on UL HPC Platform

Perspectives

- (incrementally) modify your own MATLAB code for fault tolerance
- parallelize your own tasks using parfor/GPU-enabled instructions

Thank you for your attention...

Questions?

http://hpc.uni.lu

The UL High Performance Computing (HPC) Team
University of Luxembourg, Belval Campus:
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
mail: hpc@uni.lu

1. **Pre-requisites**
2. **Objectives**
3. **Checkpointing**
   Example 1 revisited
4. **Parallelization**
   Example 2 revisited
5. **Conclusion**