Parallel Programming
with MATLAB

Christian Terboven <terboven@rz.rwth-aachen.de>
Frank Graeber <frank.graeber@mathworks.de>
12.12.2013 / Aachen, DE
Stand: 10.10.2013
Version 2.3
Acknowledgements

- This course material has been compiled by Christian Terboven, Center for Computing and Communication, RWTH Aachen University.

- Several parts have been taken from a Hands-on Workshop on Paralleles Rechnen und GPU Unterstützung für MATLAB given by Frank Graeber from The MathWorks at RWTH Aachen University. That material has been developed by The MathWorks.

- Some illustrations have been inspired by Siddharth Samsi’s introduction to Parallel MATLAB given in the Education session at SC11.
Agenda

- Parallelism Everywhere!
- Parallelism in MATLAB
  - matlabpool
  - parfor
  - Data Parallelism
- Configuring MATLAB
- Speedup and Efficiency
- Efficient MATLAB Programming
  - MEX Functions
- Where to find help / more information
Parallelism Everywhere!
The number of transistors on a chip is still doubling every 18 months …

… but the clock speed is no longer increasing that fast!

Instead, we will see many more cores per chip!

Source: Herb Sutter

www.gotw.ca/publications/concurrency-ddj.htm
Finding Concurrency

- **Chances for concurrent execution:**
  - Look for tasks that can be executed simultaneously (task parallelism)
  - Decompose data into distinct chunks to be processed independently (data parallelism)

**Organize by task**
- Task parallelism
- Divide and conquer

**Organize by data decomposition**
- Geometric decomposition
- Recursive data

**Organize by flow of data**
- Pipeline
- Event-based coordination
Divide and conquer

Problem

Subproblem

Subproblem

Subsolution

Subsolution

Subsolution

Solution

split

split

solve

merge

merge
Geometric decomposition

Example: ventricular assist device (VAD)
Pipeline

- Analogy assembly line
- Assign different stages to different PEs

![Pipeline Diagram](diagram.png)
Parallelization Overhead

- **Overhead introduced by the parallelization:**
  - Time to start / end / manage threads / processes
  - Time to send / exchange data
  - Time spent in synchronization of threads / processes

- **Efficient parallelization is about minimizing the overhead introduced by the parallelization itself!**
Load Balancing

- Perfect load balancing:
  - All threads / processes finish at the same time

- Load imbalance:
  - Some threads / processes take longer than others
  - But: All threads / processes have to wait for the slowest thread / process, which is thus limiting the scalability
Parallelism in MATLAB
Solving Big Technical Problems

Problem

Long running
Computationally intensive

You could...
Wait

Problem

Large data set
Reduce size of problem

Solutions

Larger Compute Pool
(e.g. More Processors)

Distribute similar problems to independent processors

Larger Memory Pool
(e.g. More Machines)

Set processors to work in parallel on one big problem
Parallel Computing with MATLAB

- Easily experiment with explicit parallelism on multicore machines
- Rapidly develop parallel applications on local computer
- Take full advantage of desktop power, incl. GPU(s)
- Separate computer cluster not required
Parallelism in MATLAB: Overview

- Programming Parallel Applications

  - Support built into Toolboxes
  - High-level Programming Constructs: (e.g. parfor, batch, distributed)
  - Low-level Programming Constructs: (e.g. Jobs/Tasks, MPI-based)

- Several toolbox fcts. directly leverage the Parallel Computing Tb:
Working with a matlabpool

- Starting a matlabpool with two workers in the *local* configuration

```
>> matlabpool open local 2
Starting matlabpool using the 'local' configuration ... connected to 2 labs.
```

- Determining the size of the pool and closing the pool

```
>> matlabpool size
ans =
    2
```
```
>> matlabpool close
Sending a stop signal to all the labs ... stopped.
```

- There can only be one matlabpool open at a time!
The Mechanics of `parfor` loops

- Distribute loop iterations among the workers in the `matlabpool`
Converting for to parfor

- **Requirements**
  - All loop iterations have to be independent (task + order)

- **Constraints on the loop body**
  - Cannot „introduce“ variables (e.g. eval, load, global, etc.)
  - Cannot contain break or return statements
  - Cannot contain another parfor loop
  - parfor does not provide intermediate results, and quits on error

- **Use the Code-Analyzer to diagnose parfor issues**

parfor
All variables referenced at the top level of the `parfor` loop must be resolved and classified

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Serves as a loop index for arrays.</td>
</tr>
<tr>
<td>Sliced</td>
<td>An array whose segments are operated on by different iterations of the loop.</td>
</tr>
<tr>
<td>Broadcast</td>
<td>A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop.</td>
</tr>
<tr>
<td>Reduction</td>
<td>Accumulates a value across iterations of the loop, regardless of iteration order.</td>
</tr>
<tr>
<td>Temporary</td>
<td>Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop.</td>
</tr>
</tbody>
</table>
**parfor variable classification (2/2)**

```matlab
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor idx = 1:10
    a = idx;
z = z+idx;
b(idx) = r(idx);
    if idx<=c
d = 2*a;
    end
end
```
The Parallel Computing Toolbox provides mechanisms to implement data parallel algorithms through the use of:

- Distributed Arrays: Data is distributed across multiple workers (compute nodes)
- Message Passing: MATLAB workers use message passing to exchange data and program control flow
Using Distributed Arrays

- Assuming sufficient memory on the client, a Distributed Array can be created via the `distributed()` function. Static constructors may be used.
  - 2D matrices are distributed columnwise
  - 3D matrices are distributed pagewise

- **Examples:**
  - `a = rand(1000);`
  - `dist_a = distributed(a);`
  - `dist_b = distributed.ones(4000);`
  - `full_a = gather(a);`
  - Brings all parts of `a` back to variable `full_a`
The `spmd` construct and `getLocalPart()` function

- **SPMD** stands for Single Program Multiple Data. In MATLAB it denotes parallel code.
  - Defines a block of code that runs simultaneously on multiple workers.
- **Syntax:**
  ```
  spmd
      < statements >
  end
  ```
- **The `getLocalPart()` function** can be used to access a worker's local data. Example:
  ```
  a = distributed.ones(1000);
  spmd
      local_a = getLocalPart(a);
      disp(local_a);
  end
  ```
The Parallel Computing Toolbox allows MATLAB workers to communicate with each other via Message Passing. This can be employed for parallel programming.

The PCT provides the following variables:
- `numlabs`: Total number of workers involved in the parallel computation
- `labindex`: Index (Id) of the current worker, ranges from 1 to `numlabs`

The PCT provides the following functions:
- `labSend(data, labTo)`: send data to lab
- `data = labReceive(labFrom)`: receive data from lab
- `data = labSendReceive(labTo, labFrom, data)`: simultaneously send and receive to avoid deadlocks
- `data = labBroadcast(labSender, data)`: send data to all labs
- `labBarrier()`: wait for all barriers (blocks execution)
HelloWorld
First, check whether you are connected to a matlabpool. If not, open one before using the `batch` command.

Additional input parameters:

- **FileDependencies**: Cell array of files required for the job to run
- **PathDependencies**: Full path to directories on the cluster needed for the job

Job output can be retrieved with the `load` command

- `load(job)`: Load all variables
- `load(job, "variable")`: Load only variable
Configuring MATLAB
Scheduling Jobs and Tasks
Configuring MATLAB for our Batch System (1/3)

- Go to **Parallel → Manage Configurations**...

- Select **File → New → ccs**

- **CCS = Compute Cluster Server (= the old product name)**
Choose any configuration name and description you like, e.g. ccs.

Root directory:
```
\\cifs\cluster\software\MATLAB\R2009a
```

Directory for job data:
```
\\cifs\cluster\profile\... or \\cifs\cluster\home\...
```

CCS scheduler: WINHPC-HN
Two configurations available now:

- **local**: Use multi-core capabilities of frontend machine
- **ccs**: Use capabilities of cluster via Batch System

Use parallelization capabilities of MATLAB:
- `matlabpool open configuration size`

Limitations:
- **local**: Up to four processing engines
- Cluster (here: **ccs**): Up to eight processing engines because of (current) license
Demo: Submitting a Batch Job (1/2)

- Use a Configuration matching to our Cluster
  
  ```
  sched = findResource('scheduler', 'configuration', 'ccs_2003')
  ```

- Create and run a Distributed Job
  
  ```
  job = createJob(sched);
  createTask(job, @sum, 1, {[1 1]});
  createTask(job, @sum, 1, {[2 2]});
  createTask(job, @sum, 1, {[3 3]});
  submit(job);
  ```

  ```
  % timeout: 60 sec too small!
  ```

- Gather the job results
  
  ```
  results = getAllOutputArguments(job)
  ```

- Result should look like
  
  ```
  results =
  [2]
  [4]
  [6]
  ```
Demo: Submitting a Batch Job (2/2)

- Use a Configuration matching to our Cluster
  
sched = findResource('scheduler', 'configuration', 'ccs_2003')

- Create and run a Parallel Job
  
  job = createParallelJob(sched);
  createTask(job, @labindex, 1, {});
  set(job, 'MinimumNumberOfWorkers', 3);
  set(job, 'MaximumNumberOfWorkers', 3);
  submit(job);
  waitForState(job, 'finished', 60) % timeout: 60 sec too small!

- Gather the job results
  
  results = getAllOutputArguments(job)

- Result should look like
  
  results =
  
  [1]
  [2]
  [3]
When to Use `parfor` vs. jobs and tasks

**parfor**

- Seamless integration to user’s code
- Several `for` loops throughout the code to convert
- Automatic load balancing

**Jobs and tasks**

- All tasks run
- Query results after each task is finished

- *Try `parfor` first. If it doesn’t apply to your application, create jobs and tasks.*
Not covered in this course: GPU Computing

- Graphics Processing Unit (GPU) originally designed for graphics acceleration, now also used for scientific calculations

- Massively parallel array of integer and floating point processors
  - Typically hundreds of processors per card
  - GPU cores complement CPU cores

- Dedicated (small) high-speed memory

- Parallel Computing Toolbox requires NVIDIA GPUs with Compute Capability 1.3 or greater, including NVIDIA Tesla 10-series and 20-series products. See [tp://www.nvidia.com/object/cuda_gpus.html](http://www.nvidia.com/object/cuda_gpus.html) for a complete listing.
Speedup and Efficiency
Speedup and Efficiency

- Time using 1 CPU: \( T(1) \)
- Time using \( p \) CPUs: \( T(p) \)

- **Speedup** \( S(p) \): \( S(p) = \frac{T(1)}{T(p)} \)
  - Measures how much faster the parallel computation is!

- **Efficiency** \( E(p) \): \( E(p) = \frac{S(p)}{p} \)

**Example:**
- \( T(1) = 6s \), \( T(2) = 4s \)
  - \( S(2) = 1.5 \)
  - \( E(2) = 0.75 \)

- **Ideal case:** \( T(p) = \frac{T(1)}{p} \) \( \Rightarrow S(p) = p \) \( E(p) = 1.0 \)
Amdahl’s Law

- Describes the influence of the serial part onto scalability (without taking any overhead into account).

\[
S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{f \cdot T(1) + (1-f) \cdot T(1)/p} = \frac{1}{f + (1-f)/p}
\]

- \(f\): serial part \(0 \leq f \leq 1\)
- \(T(1)\): time using 1 CPU
- \(T(p)\): time using \(p\) CPUs
- \(S(p)\): speedup; \(S(p) = \frac{T(1)}{T(p)}\)
- \(E(p)\): efficiency; \(E(p) = \frac{S(p)}{p}\)

- It is rather easy to scale to a small number of cores, but any parallelization is limited by the serial part of the program!
If 80% (measured in program runtime) of your work can be parallelized and "just" 20% are still running sequential, then your speedup will be:

- 1 processor: time: 100% speedup: 1
- 2 processors: time: 60% speedup: 1.7
- 4 processors: time: 40% speedup: 2.5
- ∞ processors: time: 20% speedup: 5
After the initial parallelization of a program, you will typically see speedup curves like this:
Efficient MATLAB Programming
Preallocating Big Arrays

- for and while loops that incrementally increase the size of a data structure each time through the loop stress the memory subsystem
  - increasing the already allocated block might not be possibly, so a new allocation has to be performed as well as a copy operation

- use
  - zeros for numeric arrays
  - cell for character arrays


```matlab
x = 0;
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
```

Elapsed time is 0.475303 seconds.

```matlab
x = zeros(1, 1000000);
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
```

Elapsed time is 0.026810 seconds.
If a variable once has been assigned a value to, it implicitly also has received a type. If you later store data of a different type in the same variable, MATLAB needs extra processing time.

- Create a new variable

Do not assign a real value to a variable previously holding a complex value, and vice versa.

Use appropriate logical operators

- & and |: perform logical AND and OR on arrays element by element
- && and ||: perform logical AND and OR on scalar values with short-circuiting
- especially on if and while statements short-circuiting may save evaluations
MATLAB provides a compiler wrapper to enable you to build MEX modules from native C/C++ or Fortran code

- Machine-specific tuning possible
- Fine-grained memory control
- Can contain parallel code, e.g. OpenMP parallelized code
- Can interface accelerated libraries like Intel MKL
Typical function body

```c
#include <matrix.h> // Matlab matrix datatypes
#include <mex.h> // MEX functionality

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    // Code ...
}
```

Code examples

- OpenMP parallelized programm
- Interface to threaded `dpotrf()` (cholesky decomp.) from Intel MKL
How to build a MEX binary with Intel C++ compiler and Intel MKL on the Linux Cluster

- Load the compiler of your choice
  
  ```bash
  $ module load intel/14.0
  ```

- Load MATLAB
  
  ```bash
  $ module load MISC matlab
  ```

- Get the MEX configuration template
  
  - Matlab console: `mex -setup`
  
  - It is stored in `$HOME/.matlab/R2013a/mexopts.sh`
MEX-Functions

- Change the C++ compiler and linker related variables to the corresponding intel equivalents in mexopts.h
  - Change the compiler name to icpc
  - Set the desired tuning and debugging flags
  - Add `/opt/intel/Compiler/14.0/1.106/rwthInk/compiler/lib/intel64/libirc.a` and `/opt/intel/Compiler/14.0/1.106/rwthInk/compiler/lib/intel64/libimf.a` to your C++ libraries
- Compile your files with
  - `$ mex <filelist> -I <includedirectory> -o <modulename>`
Where to find help / more information
Infos on the Web + Contact

- Latest and greatest info on how to use MATLAB on the cluster in the RZ (will be updated soon):

- The RZ service desk, if what is on the web does not work:
  - mail to servicedesk@rz.rwth-aachen.de
Thank you for your attention.

The End