
MATLAB - Parallelization and Performance

Jan Steiner

Zentrum für Informations- und Medientechnik

December 3, 2020

A solid dark blue horizontal bar spanning the width of the slide at the bottom.

A word about Zoom

- Please stay muted unless necessary
- Questions: unmute, say “Question”, remute, I will call you
 - I will probably miss “Raise Hand” button
- Make sure microphone and screen sharing works
 - Might be blocked by OS the first time
- Breakout rooms: groups of 2-3, do exercises collaboratively
 - One person should share screen, rotate through

Round of introductions

- Who are you?
- What is your research about?
- What do you use MATLAB for?
- How much experience do you have with MATLAB?

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
 - Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
 - Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
 - Summary
- Lunch break (approx.)
- Around 12:30-13:00
 - 60 minutes

Outline

- Part 1: Introduction
 - **Background**
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Background

- Lots of MATLAB users at Uni Siegen
 - Sometimes very elaborate setups
 - MATLAB not really built for HPC
 - Rule of thumb: 1-2 orders of magnitude slower than C, Fortran
 - However many features to support cluster use
 - HPC community does not care much about MATLAB
- What advice to give users?

Background: idea

- Idea: approach problem by “simulating” a regular user
- Create own MATLAB application
 - More complex than code examples/exercises
 - Simpler than real scientific codes
- Parallelize application
 - Note pitfalls, tips & tricks
- Optimize application
 - Check against common HPC wisdoms
 - Get a feeling for MATLAB performance

Test code, basics

- Code exists in multiple languages
 - So far Python, MATLAB, Fortran
- 3D panel method (fluid dynamics)
 - Based on APAME (open source), which is based on VSAERO (NASA)
 - Simplified physics compared to modern CFD
- Mix of various parallel techniques
 - Understand techniques
 - Example for teaching purposes

Test code, equation system

- Equation system:

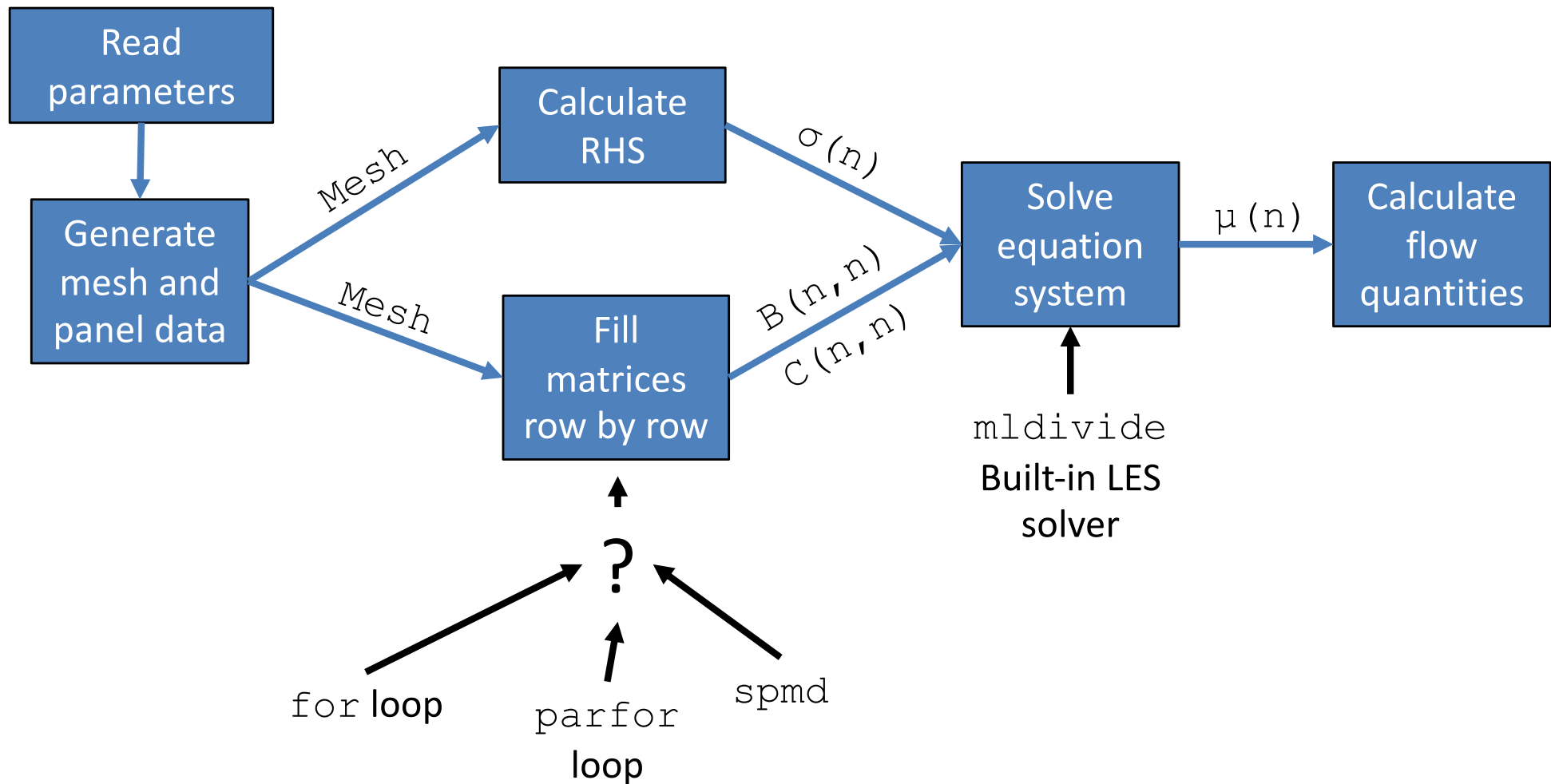
$$\underbrace{\begin{pmatrix} C_{11} & C_{12} & C_{1N} \\ C_{12} & C_{22} & C_{2N} \\ C_{N1} & C_{N2} & C_{NN} \end{pmatrix}}_{Mat} \underbrace{\begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_N \end{pmatrix}}_x = - \underbrace{\begin{pmatrix} B_{11} & B_{12} & B_{1N} \\ B_{21} & B_{22} & B_{2N} \\ B_{N1} & B_{N2} & B_{NN} \end{pmatrix}}_{RHS} \underbrace{\begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_N \end{pmatrix}}_{\text{Known source strengths}}$$

Known dipole influences (points to C matrix)
 Unknown dipole strengths (points to μ vector)
 Known source strengths (points to σ vector)

- Operations:
 - [Generate mesh]
 - Fill 2 matrices
 - Calculate RHS (matrix-vector product)
 - Solve linear equation system
 - [Calculate velocities]

Known source influences (incl. freestream)

Test code, data flowchart



Test code, operations

- Good mix of operations:
 - Built-in `mldivide` for solving equation system
 - Several built-in `vecnorm` function calls
 - Some `for` loops
- Two compute-intensive phases:
 - Fill B and C matrices
 - Solve equation system
- Complexity roughly $O(n^2)$

Test code, data objects

- Data objects
 - Mesh + Paneldata (`mesh` and `panels`)
 - Influence data (`infl`)
 - Information needed to calculate influence of every panel onto one point
 - Lots of vectors (e.g. from every panel corner to target point)
 - Lots of scalars
 - Singularity data (`sing`)
 - B and C matrix
 - Source and dipole strengths
 - RHS vector
 - [Flow data] (velocities, pressure etc.)

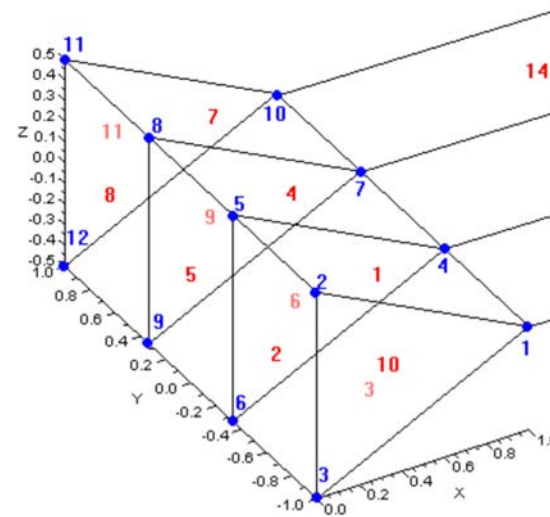
Test code, data requirements

- Data object structure
 - Constructor (allocates arrays)
 - `calc()` method (fills arrays with values)
 - Other methods as needed (called by `calc()` method)
- Total memory needed: $(2 * N^2 + 100 * N) * 8$ Byte
 - More than 50 panels: matrices larger than everything else combined
 - 1000 panels → 16 MB
 - 10000 panels → 1.6 GB
 - 100000 panels → 160 GB

Test cases

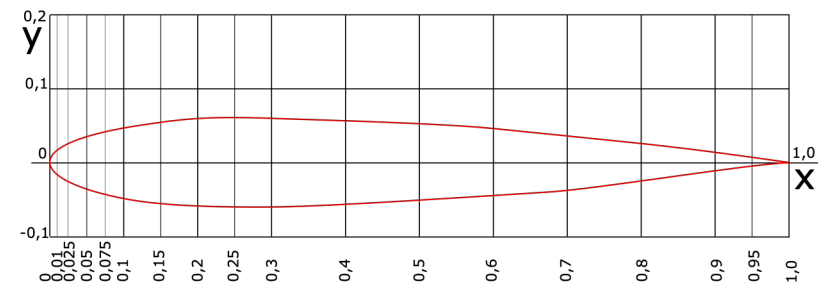
1. APAME test mesh

- Only 12 nodes, 11 panels
- For debugging purposes
- Reference solution exists
- Wake ignored
- Triangles and quads



2. NACA 0012 airfoil

- Simple geometry (one layer of quads)
- Can be mathematically generated
- Easily scalable



Investigation method

- Implement algorithm in Python
- Port to MATLAB (very easy)
- Set up MATLAB profiling
- Investigate serial + built-in multithreading performance
 - What takes how much time?
 - Is scaling logical?
 - Do common tips and tricks apply?
 - What else did I notice?
- Parallelize with different strategies
 - What is easy/hard to understand/implement
- Investigate parallel performance

Outline

- Part 1: Introduction
 - Background
 - **MATLAB at Uni Siegen**
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

MATLAB licensing at Uni Siegen

- MATLAB available to everyone
 - But not free for everyone
- Three types of licenses
 - Employee license
 - Student license
 - Cluster license (Parallel Server)
- No restriction on version

MATLAB licensing: employees

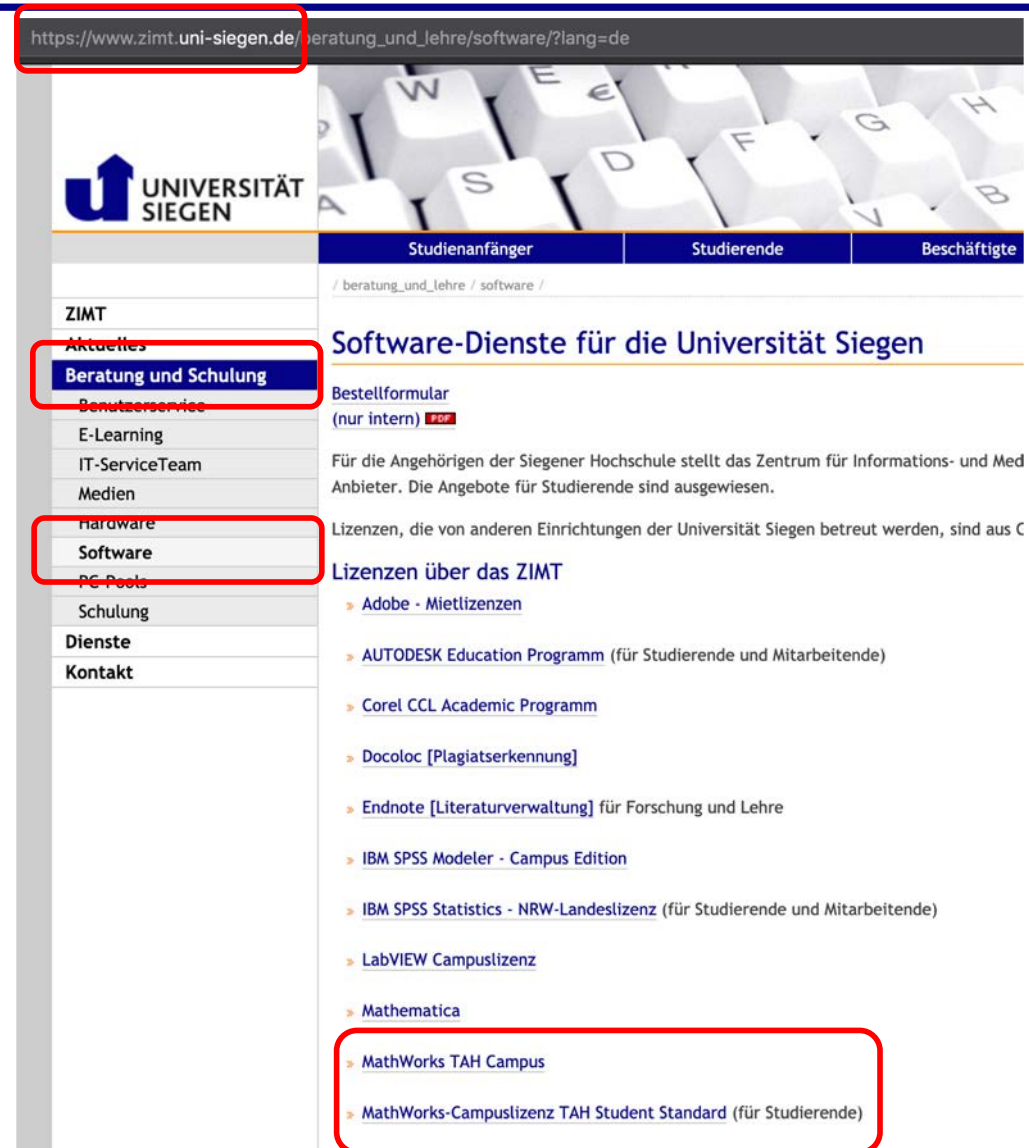
- Available to all institutes/workgroups, managed by Fak. 4
- Large collection of toolboxes (including Parallel Computing)
- May be installed on university or own computer
 - May not be used on cluster outside Uni Siegen
- Cost (recently increased):
 - 1-3 users: €860 each
 - 4-6 users: €640 each
 - 7-10 users: €540 each
 - >10 users: 430 each

MATLAB licensing: cluster

- MATLAB Parallel Server
- Formerly: MATLAB Distributed Computing Server
- 16 workers
- Not limited to one node
 - With multithreading: use up to 16 nodes fully
- Parallel pool in SLURM job
 - Pools in general are part of Parallel Computing Toolbox

Getting MATLAB

- ZIMT website
- Forms to fill out
- Detailed descriptions of license terms
- List of available toolboxes



https://www.zimt.uni-siegen.de/beratung_und_lehre/software/?lang=de

UNIVERSITÄT SIEGEN

Studienanfänger Studierende Beschäftigte

/ beratung_und_lehre / software /

Software-Dienste für die Universität Siegen

Bestellformular
(nur intern) PDF

Für die Angehörigen der Siegener Hochschule stellt das Zentrum für Informations- und Med
Anbieter. Die Angebote für Studierende sind ausgewiesen.

Lizenzen, die von anderen Einrichtungen der Universität Siegen betreut werden, sind aus C

Lizenzen über das ZIMT

- Adobe - Mietlizenzen
- AUTODESK Education Program (für Studierende und Mitarbeitende)
- Corel CCL Academic Programm
- Docoloc [Plagiatserkennung]
- Endnote [Literaturverwaltung] für Forschung und Lehre
- IBM SPSS Modeler - Campus Edition
- IBM SPSS Statistics - NRW-Landeslizenz (für Studierende und Mitarbeitende)
- LabVIEW Campuslizenz
- Mathematica
- MathWorks TAH Campus
- MathWorks-Campuslizenz TAH Student Standard (für Studierende)

MATLAB at Uni Siegen: miscellaneous info

- Free alternative: Octave also installed
 - Mostly MATLAB-compatible syntax
 - Parallel package available
 - Have not looked at this in detail
- MATLAB training courses
 - Mech. Eng. lecture: “Introduction to Computer Science”
 - Contains MATLAB intro segment
 - Many users at ENC
 - Other intro course?
- Getting help:
 - MATLAB documentation free, extensive, also usage tips, tutorials
 - Our cluster website explains integration, downloads

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - **MATLAB parallel features**
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Using MATLAB on the cluster: parallelism

- MATLAB on a cluster only makes sense if one uses parallelism
 - More small computations
 - Larger computations
 - Same computation in shorter time
- MATLAB parallel features:
 - Built-in multithreading
 - Parallel Toolbox
 - MATLAB Parallel Server

MATLAB multithreading

- Built into MATLAB
 - Since at least R2014
- Does not require user action
 - However does allow some control
- You have used it without noticing
- Most built-in functions use it
 - Example: `mldivide()`
- Allows use of a full node (or PC), but only one

MATLAB Parallel Toolbox

- Available as optional first-party toolbox
 - Included in all Uni Siegen licenses
- Introduces concept of a “worker”
 - Multiple MATLAB processes run in background (“workers”), one MATLAB process with GUI in foreground (“client”)
- Concept of a “parallel pool”
 - Group of worker processes
 - Work and data may be distributed among them
- Large arsenal of functions to distribute work over workers

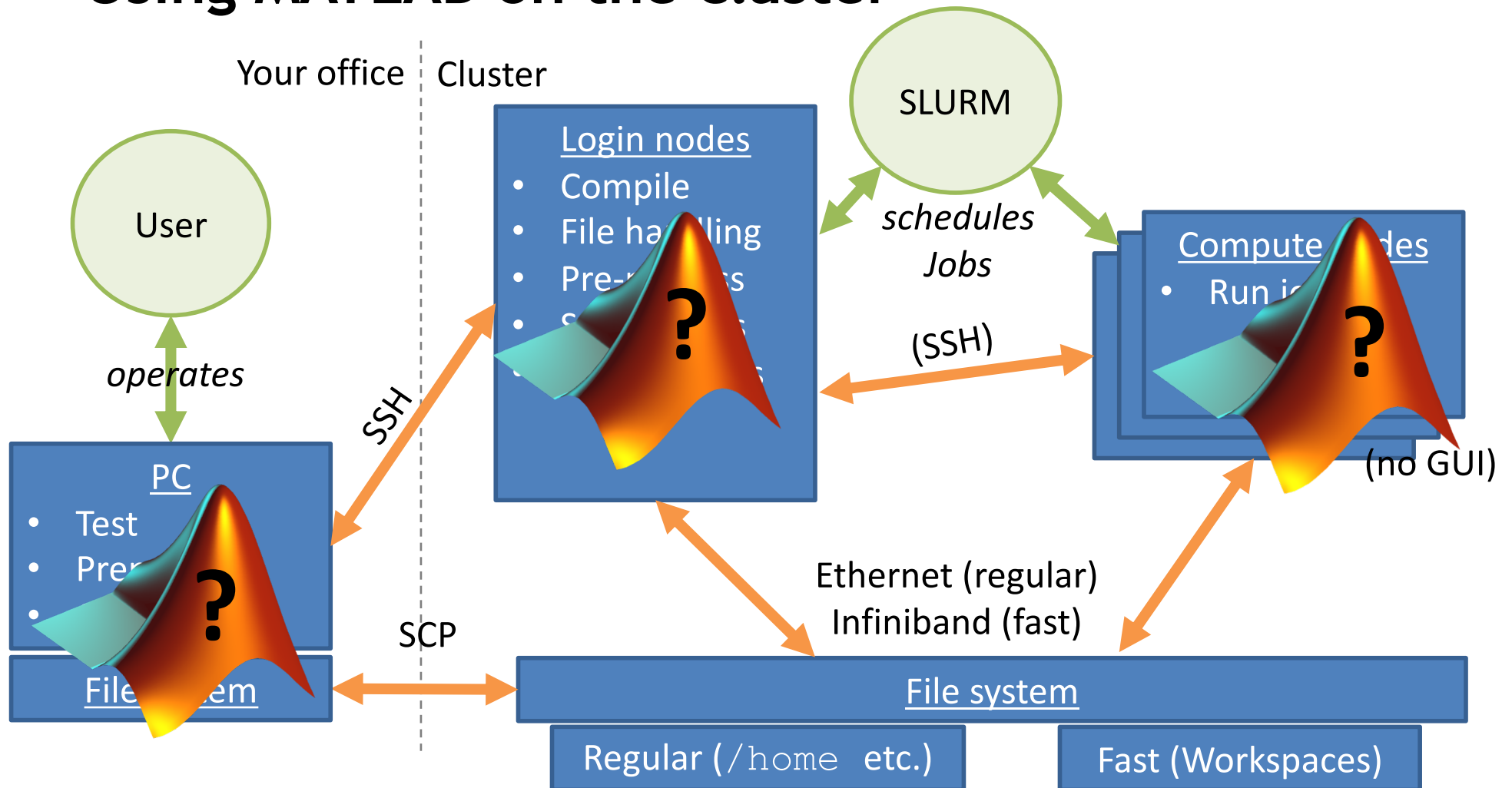
MATLAB Parallel Server

- Server-(i.e. cluster-)side component
 - User does not need additional license/tools etc. beyond Parallel Toolbox
- Allows starting a pool on multiple nodes
 - Can talk to SLURM
- Separate license
 - Uni Siegen has license for 16 workers
 - Theoretically: run job on 16 nodes
- Introduces concept of “cluster profile”
 - “Local” profile identical to running workers on same PC
 - ZIMT provides “HoRUS” profile

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - **Using MATLAB on the cluster**
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Using MATLAB on the cluster



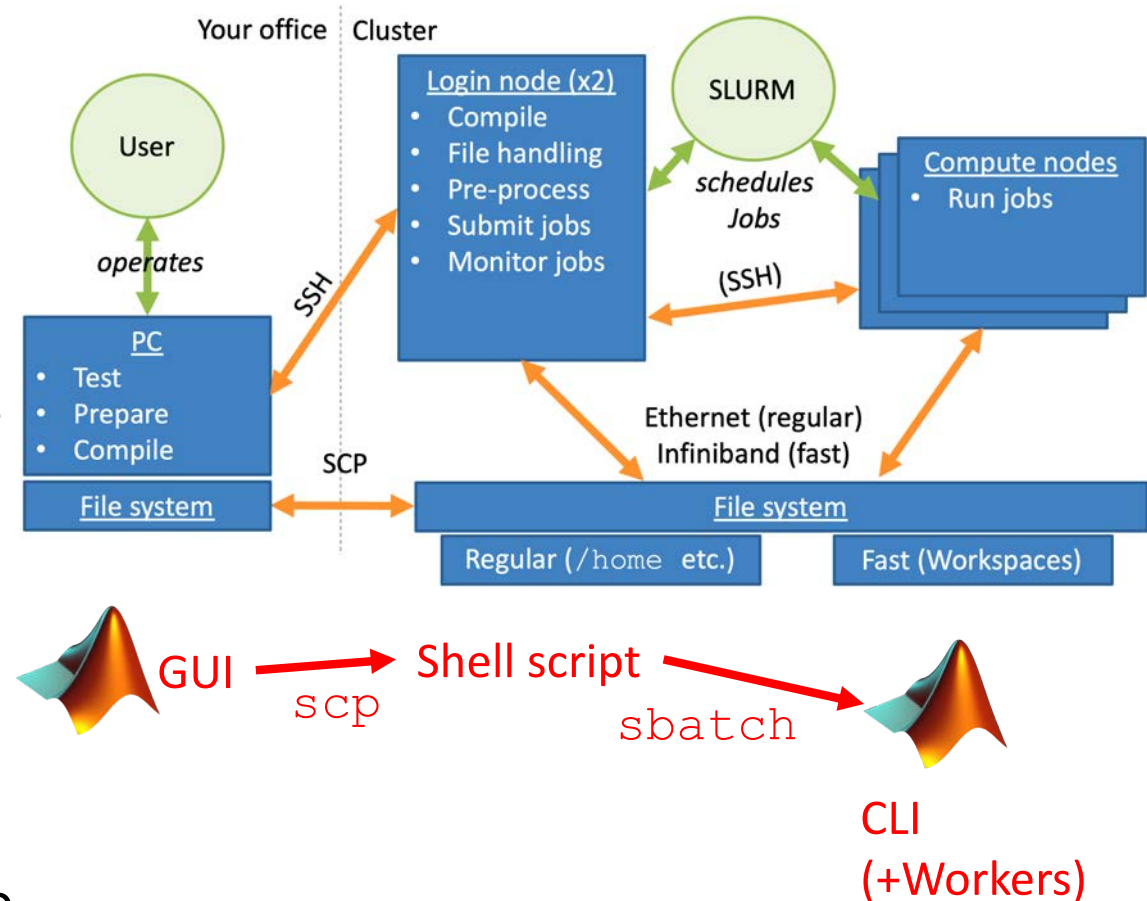
Using MATLAB on the cluster

- Different ways of using MATLAB
- Each has strengths and weaknesses
- Each requires varying levels of setup

Five scenarios on following slides, with advantages and disadvantages

Using MATLAB: Scenario 1a

- You primarily use MATLAB on your PC
 - Can use parallel pools locally
- Your computation is one M-file
- If you need more compute power, switch to cluster
 - Copy M-file to cluster manually
 - Create job script yourself
 - Queue job like any other job



Scenario 1a: comparison

Advantages:

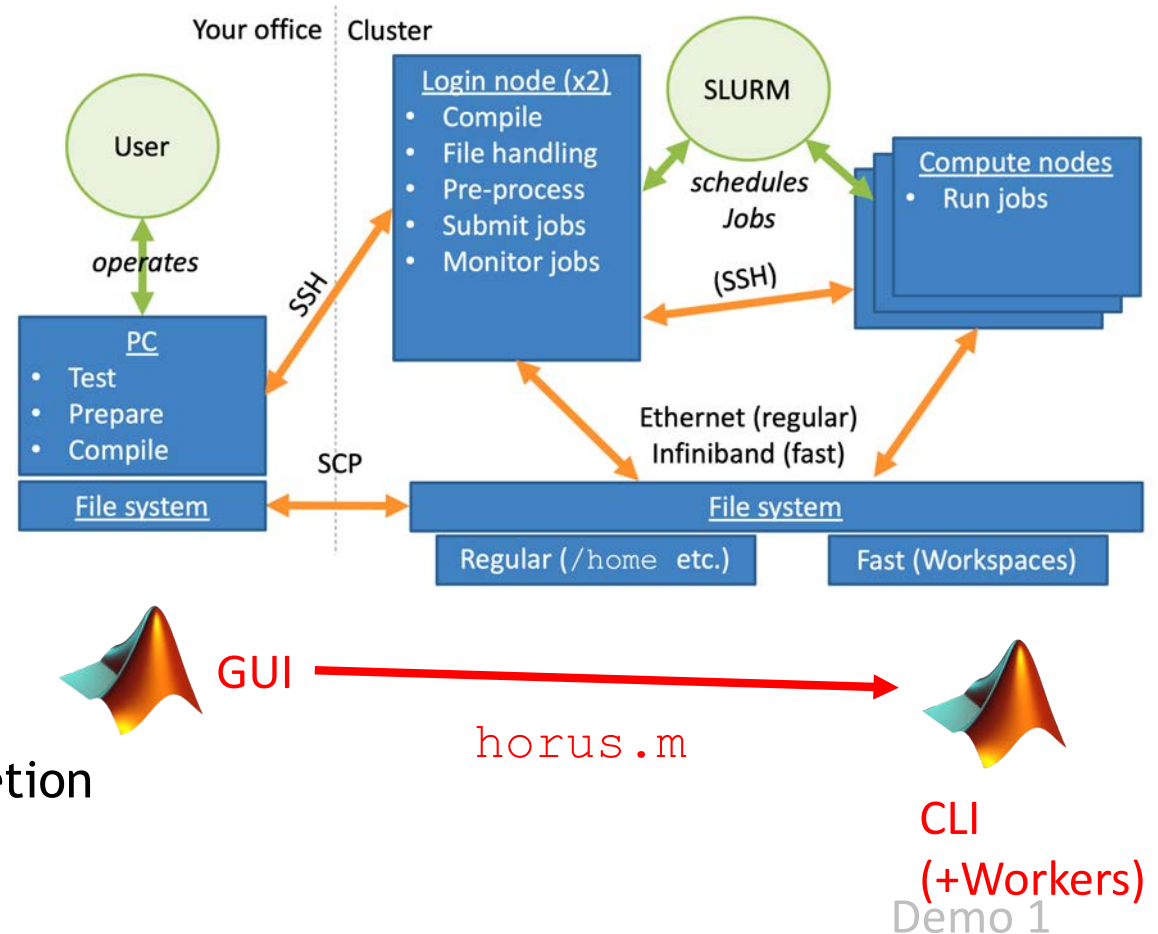
- Very simple job script
- Complete control over SLURM
- Easily allows other operations outside MATLAB
- Local parpool on node(s) still possible...
- Can use own PC for everything except extensive calculations

Disadvantages:

- No interactivity
- Manual file transfer necessary
- Manual job setup and queuing necessary
- ...but parpool needs second job (or only local)
- Heterogenous systems, toolboxes between cluster and PC

Using MATLAB: Scenario 1b

- Like scenario 1a, but partially automated
- Script donated by Prof. Nelles group, modified by me
- Automated steps:
 - Copying files to cluster
 - Generating job script
 - Queuing job
 - Retrieving files after completion
- Built-in documentation



Scenario 1b: comparison

Advantages:

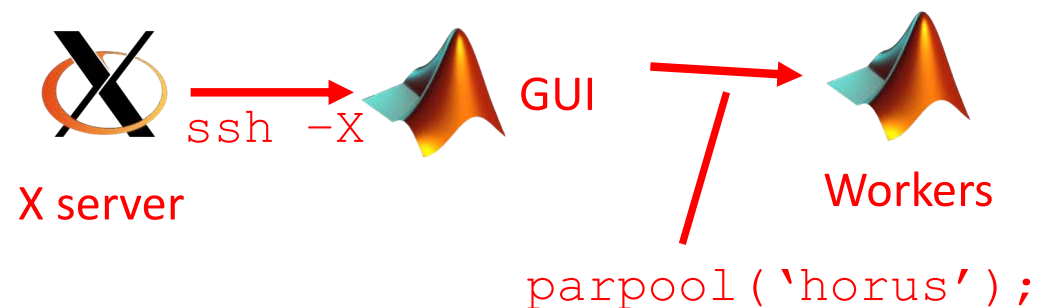
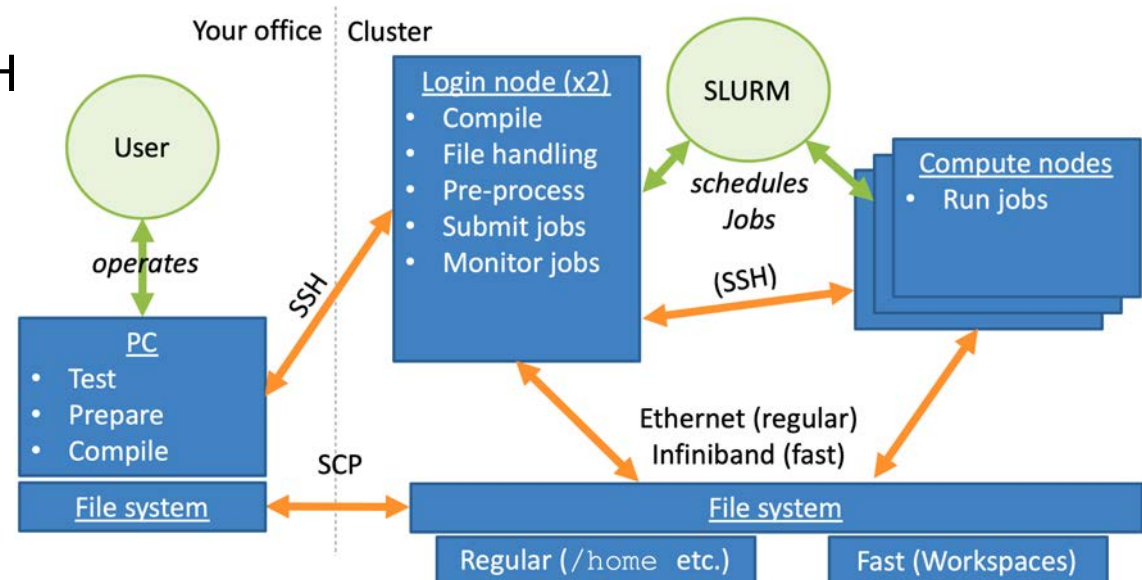
- Highly automated
- Easy to use script
- Handles generation of parpool inside job if necessary
- Handles file synchronization between PC and cluster

Disadvantages:

- Self-built solution (may contain bugs)
- Hard-coded job script, not flexible
- Still no interactivity inside job
- Still heterogeneity between PC and cluster
- Requires password-less login on Windows

Using MATLAB: Scenario 2a

- Connect to cluster via regular SSH
- Use MATLAB GUI on login node
 - Edit scripts, brief test runs
 - **DO NOT RUN ACTUAL COMPUTATION ON FRONT-END**
- Start pool from within MATLAB
 - With horus profile
- Do compute-intensive work with pool (using Parallel Toolbox commands)



Scenario 2a: comparison

Advantages:

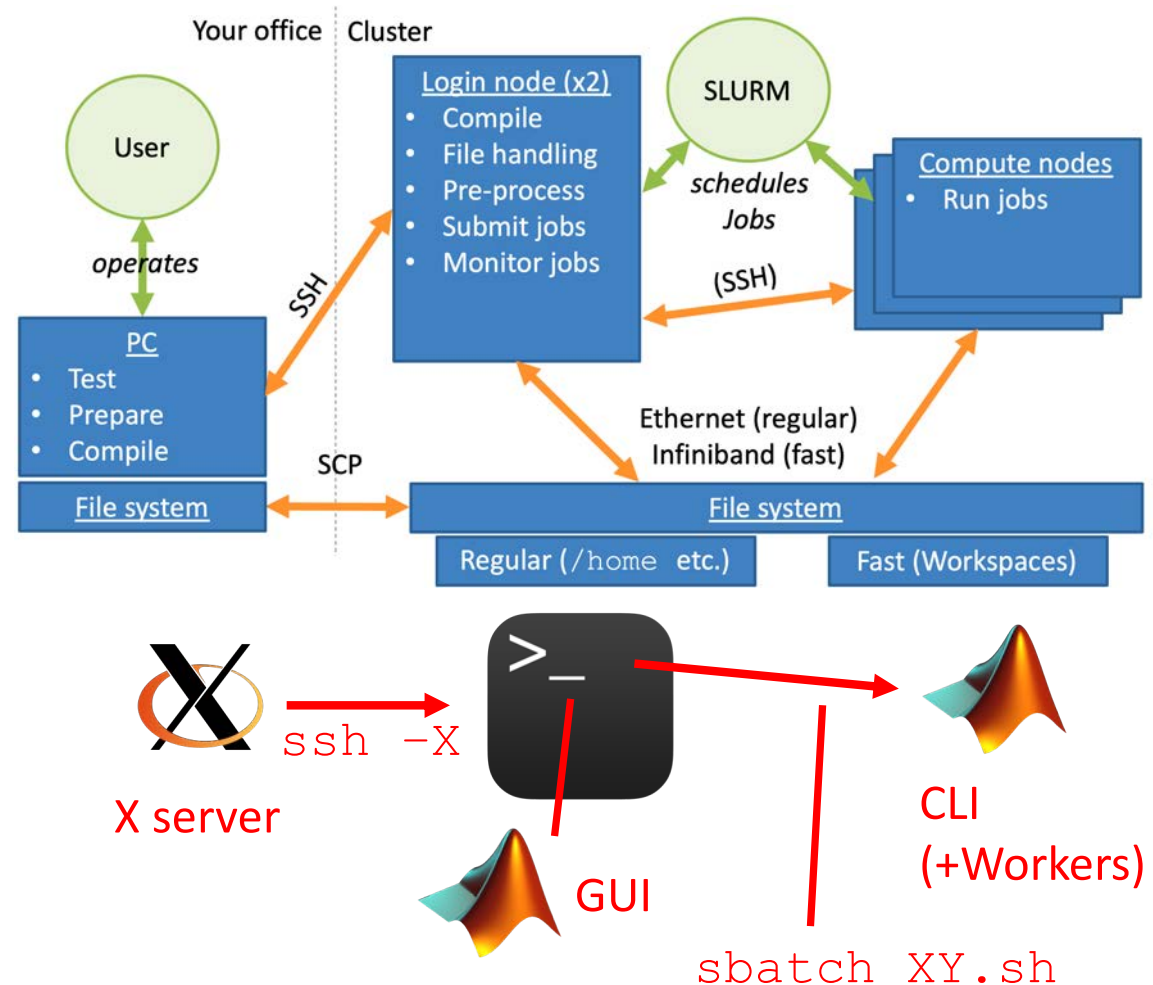
- Needs no MATLAB on PC
- Homogeneous between front-end and back-end
- Full interactivity (once job is running, you can do anything)
- No job script necessary: starting pool will automatically start job

Disadvantages:

- Ugly GUI
 - Does not even relaunch after close
- Have to wait for job to start
 - Cannot run scripts
- Sometimes unstable SSH connection
- Occasional MPI problems in the past
- Control over SLURM (e.g. queue) limited/complicated

Using MATLAB: Scenario 2b

- Connect to cluster and open MATLAB GUI
 - Like 2a
- Do not use MATLAB to launch jobs, only to edit scripts, prepare job
- Launch jobs separately via sbatch
 - Like method 1a



Scenario 2b: comparison

Advantages:

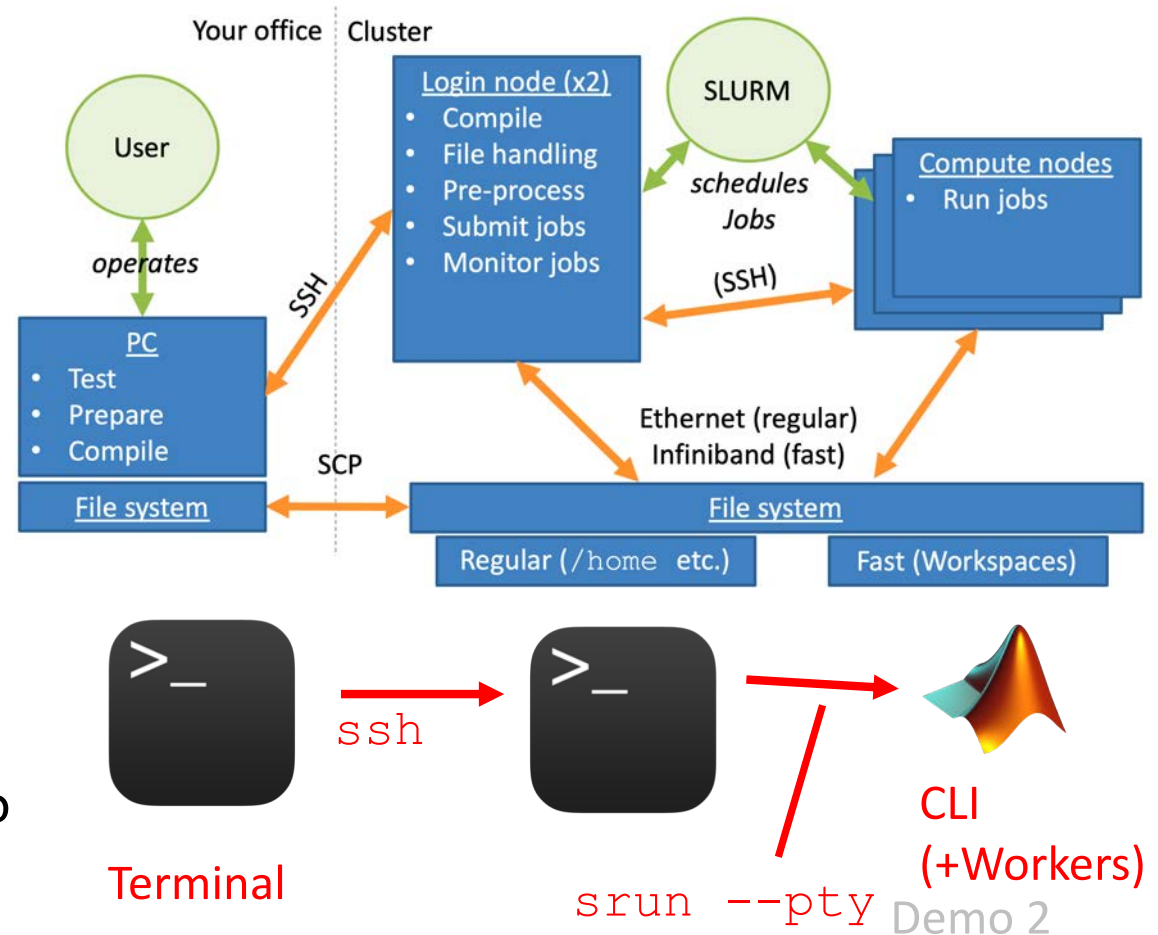
- Combines advantages of 1a and 2a
 - Flexibility
 - Homogeneity
 - Pool optional
- Can quickly look at job output files
- Uses local pool on nodes (no extra wait time)
- Multiple jobs at the same time

Disadvantages:

- Also combines disadvantages
 - Ugly GUI
 - Manual file transfer
 - Manual job setup
 - Worker pool either only local or second job
- Little interactivity with job

Using MATLAB: Scenario 3

- You connect to cluster normally
- Start an interactive job normally
- Start CLI MATLAB inside job
- If needed, start pool inside of job



Scenario 3: comparison

Advantages:

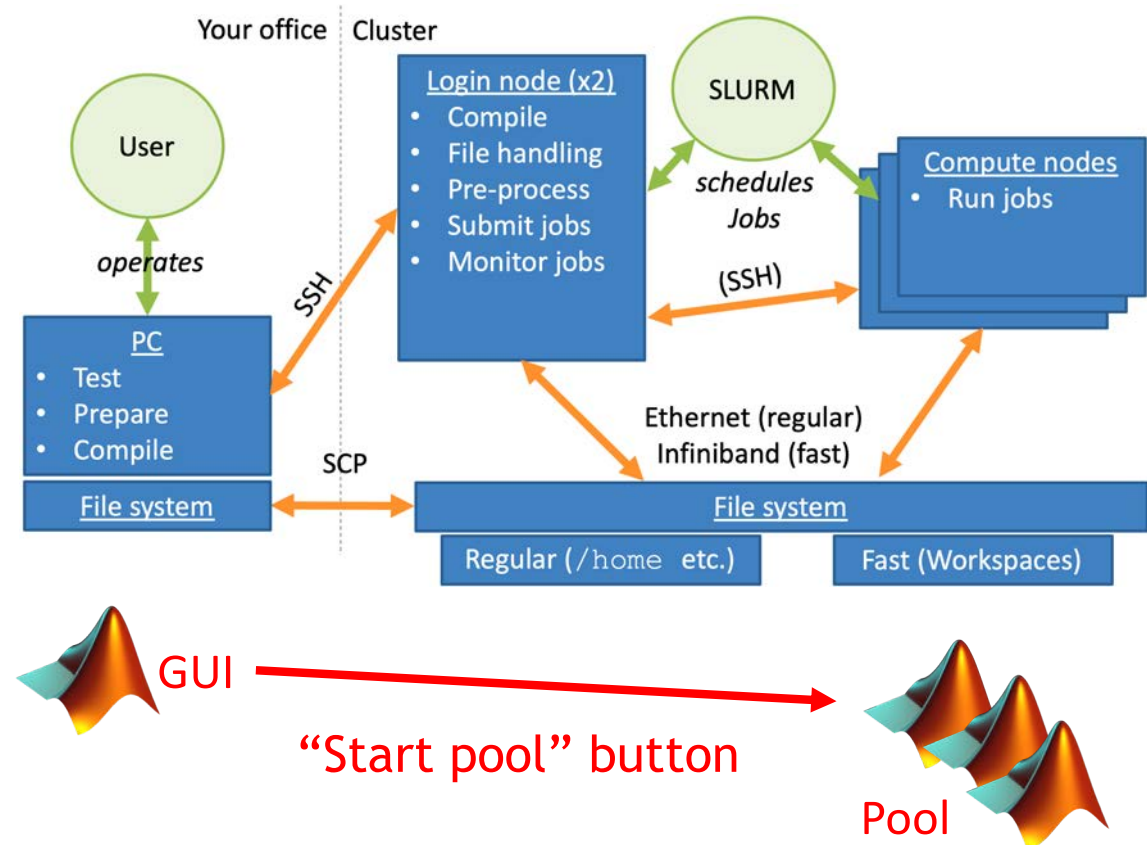
- Compute resources reserved for you
- May still start pool with local profile
- Edit and run job on exact same MATLAB
- CLI MATLAB starts quickly, job continues on MATLAB crash

Disadvantages:

- Only CLI MATLAB available
- Initial job wait time
- Job dies if SSH connection to cluster closes

Not possible

- Cannot start pool on cluster from your PC
- Too much setup
- Only semi-officially supported
- We want you to think about resource usage



Simple basic job script

- Simple job script
 - Scenario 1a, 2b
- Calls single MATLAB M script
- Script may open parallel pool internally
- If local:
 - Can only run on same node
 - Needs different job settings

```
1  #!/bin/bash
2  #SBATCH -p short
3  #SBATCH -t 0:01:00
4  #SBATCH --nodes 1
5  #SBATCH --ntasks-per-node 1
6  #SBATCH --cpus-per-task 6
7  #SBATCH --mem 48000
8
9  matlab -nodisplay -r "demo1"
```

Simple basic job script

Job settings:

- One task (one MATLAB client process)
 - More tasks if workers started
- CPUs per task determines MATLAB threads
 - **However** MATLAB needs to be told this
- MATLAB called with `-nodisplay`

```
1  #!/bin/bash
2  #SBATCH -p short
3  #SBATCH -t 0:01:00
4  #SBATCH --nodes 1
5  #SBATCH --ntasks-per-node 1
6  #SBATCH --cpus-per-task 6
7  #SBATCH --mem 48000
8
9  matlab -nodisplay -r "demo1"
```

Simple basic job script: multithreading

- MATLAB disregards `--cpus-per-task` by default and uses all cores on node
- Threads determined internally by `maxNumCompThreads`
- Can use SLURM environment variable inside MATLAB
 - Either `SLURM_CPUS_ON_NODE` if only MATLAB running, or `SLURM_CPUS_PER_TASK`

Demo 3

```
disp('Default value of maxNumCompThreads:');  
maxNumCompThreads  
  
if ~isempty( getenv('SLURM_CPUS_ON_NODE') )  
    maxNumCompThreads( str2num( getenv('SLURM_CPUS_ON_NODE') ) );  
    disp('Taking value of maxNumCompThreads from SLURM:');  
    maxNumCompThreads  
end
```

Using a pool inside a job

- A job running MATLAB is still only one process
 - Up to 12 cores via threads

→ Start a pool from inside job?

- Option A: use local profile
 - Up to 12 workers
 - MATLAB client also running: 11 workers?
- Option B: use 'horus' profile, accept that second job is started
 - Up to 17 nodes theoretically (only if really needed please)
 - Check resources: how much does MATLAB client need?

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - ***Exercise 1***
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Exercise 1

Login: `schulungXY`
(where XY=01-12)
PW: `Matlab##Dec3`

- Objectives
 - You can start MATLAB on the cluster
 - You understand the differences between the different approaches
- Tasks
 - Connect to cluster
 - Start MATLAB in GUI mode and in CLI mode
 - Start an interactive job with MATLAB
 - Look at the output of `maxNumCompThreads`

Note the following slide (before you begin the exercise)

Exercise 1

- Notes
 - You have been given SLURM and Linux cheat sheets
 - You are allowed and indeed encouraged to use Google and the MATLAB documentation
 - Interactive jobs explained on cluster website:
<https://cluster.uni-siegen.de/benutzung/jobs-einstellen/#interaktiv>
- If bored, get creative
 - Play around with SLURM job settings
 - Open another Linux console with `top`, then run compute-intensive commands in MATLAB and watch CPU usage
 - Look at `horus.m`
 - ...

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - **Parallel pools and cluster profiles**
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

MATLAB Multithreading

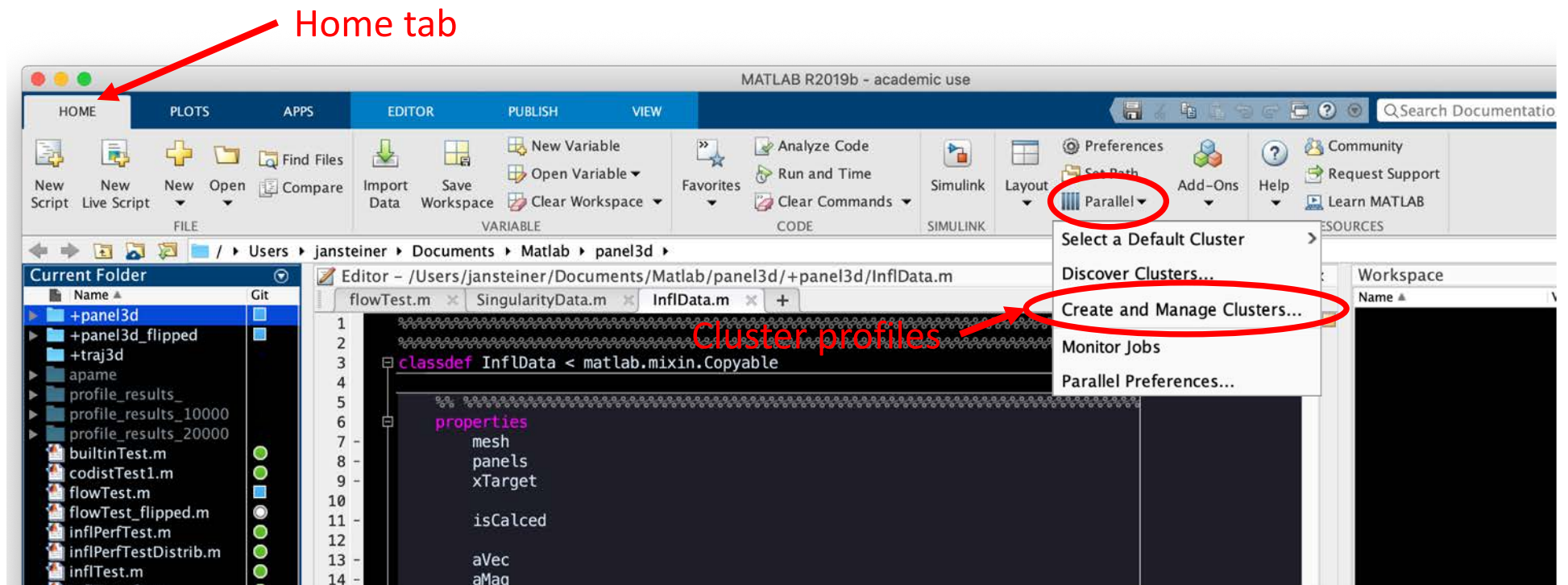
A word on `maxNumCompThreads`:

- Command to control number of threads
 - **Get:** `nThr = maxNumCompThreads`
 - **Set:** `maxNumCompThreads (4)`
 - Will return previous setting
- Cannot be set as MATLAB launch option that I am aware of
 - **Exception:** `-singleCompThread`
- MATLAB documentation: “may be removed in future” (so far not)

Creating clusters and pools

- Controlling parallel pool is done from inside MATLAB
- Either in GUI or programmatically
- Main actions:
 - Add and edit cluster profiles
 - Start and stop parallel pool
 - Send tasks to workers → next section

Cluster profile manager



Cluster profile manager

Import button

Properties see next slides

The screenshot shows the 'Cluster Profile Manager' window. On the left, a list of cluster profiles includes 'GenericProfile1', 'horus', 'HPCServerProfile1', 'local (default)', and 'SlurmProfile1'. A red arrow points to 'local (default)' with the text 'Local profile always there' and 'Can change default'. The main area displays the 'horus' profile details. A red arrow points to the 'Import' button in the top toolbar. Another red arrow points to the 'Properties' button. The 'Validation' tab is active, showing a table of test stages:

Stage	Status	Description
Cluster connection test (parcluster)	Failed	Unable to proceed because 'sinfo' is not on your path. Most likely this is ...
Job test (createJob)	Skipped	Validation skipped due to previous failure.
SPMD job test (createCommunicatingJob)	Skipped	Validation skipped due to previous failure.
Pool job test (createCommunicatingJob)	Skipped	Validation skipped due to previous failure.
Parallel pool test (parpool)	Skipped	Validation skipped due to previous failure.

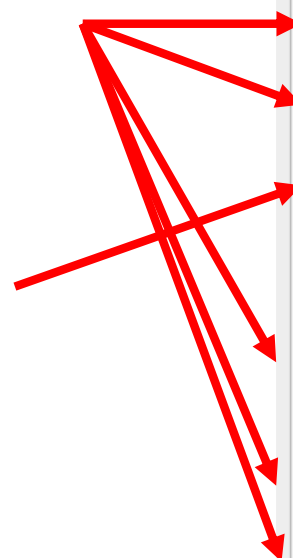
Below the table, it says 'Number of workers to use: Use default'. Under 'STAGE DETAILS', it shows 'Stage started at 16:43:34. Completed in 0 min 2 sec.' and an 'Error Report' with the same message as the failed stage. At the bottom right, the 'Validate' button is circled in red, with the text 'Validate to see if profile works' next to it.

Cluster profile settings

- We have prepared a cluster profile for HoRUS
 - Download from website
 - Import into MATLAB GUI or CLI on cluster
- You might have to adjust some options
 - Some options might need adjusting on a case-by-case basis (e.g. walltime)
- Can be controlled programmatically (`parcluster` object)

Cluster profile settings


- Pre-set and mostly self-explanatory
 - Leave these alone
- Maximum workers in a pool
 - Can be at most 16
 - May still start pool with fewer workers
 - Can generally be left alone



Description of this cluster Description	horus
Folder where job data is stored on the client JobStorageLocation	current working folder (default)
Number of workers available to cluster NumWorkers	16
Number of computational threads to use on each worker NumThreads	1
Root folder of MATLAB installation for workers ClusterMatlabRoot	/cm/shared/mapps/matlab/2019a/
License number (Optional: Used only if this cluster uses online licensing) LicenseNumber	<none>
Cluster uses online licensing RequiresOnlineLicensing	<none>

Cluster profile settings

- Maximum threads per worker
 - Pre-set to 1
 - Depends on your use case
 - Emphasis on built-in functions: 12, but fewer workers
 - Emphasis on pool: 1 thread + 16 workers
 - Full power needed: 16 workers x 12 threads
 - 16-node job → long wait time



Description of this cluster Description	horus
Folder where job data is stored on the client JobStorageLocation	current working folder (default)
Number of workers available to cluster NumWorkers	16
Number of computational threads to use on each worker NumThreads	1
Root folder of MATLAB installation for workers ClusterMatlabRoot	/cm/shared/mapps/matlab/2019a/
License number (Optional: Used only if this cluster uses online licensing) LicenseNumber	<none>
Cluster uses online licensing RequiresOnlineLicensing	<none>

Cluster profile settings

- Basic SLURM options
 - Pre-set to automatically start with correct number of tasks/cores
 - Can be left alone
- Custom SLURM options
 - Whatever you need
 - Especially queue and walltime
 - Unfortunately has to be changed in profile every time
 - Common: `--mem 48000`

ADDITIONAL SLURM PROPERTIES	
Resource arguments for job submission. Use the placeholders '^N^' for the number of workers and '^T^' for NumThreads. ResourceTemplate	--ntasks=^N^ --cpus-per-task=^T^ (default)
Additional command line arguments for job submission SubmitArguments	<none>
Script that cluster runs to start workers CommunicatingJobWrapper	MathWorks provided script (default)

FILES AND FOLDERS	
Automatically send code files to cluster. Data files must be listed below. AutoAttachFiles	true (default)
Manually specify files and folders to copy from client to cluster nodes (One entry per line) AttachedFiles	<input type="text" value="<none>"/>
Manually specify folders to add to the workers' search path (One entry per line) AdditionalPaths	<input type="text" value="<none>"/>

Cluster profile settings

- Leave this alone

- Not sure what these do
 - Can be left alone

ADDITIONAL SLURM PROPERTIES	
Resource arguments for job submission. Use the placeholders '^N^' for the number of workers and '^T^' for NumThreads. ResourceTemplate	--ntasks=^N^ --cpus-per-task=^T^ (default)
Additional command line arguments for job submission SubmitArguments	<none>
Script that cluster runs to start workers CommunicatingJobWrapper	MathWorks provided script (default)

FILES AND FOLDERS	
Automatically send code files to cluster. Data files must be listed below. AutoAttachFiles	true (default)
Manually specify files and folders to copy from client to cluster nodes (One entry per line) AttachedFiles	<input type="text" value="<none>"/>
Manually specify folders to add to the workers' search path (One entry per line) AdditionalPaths	<input type="text" value="<none>"/>

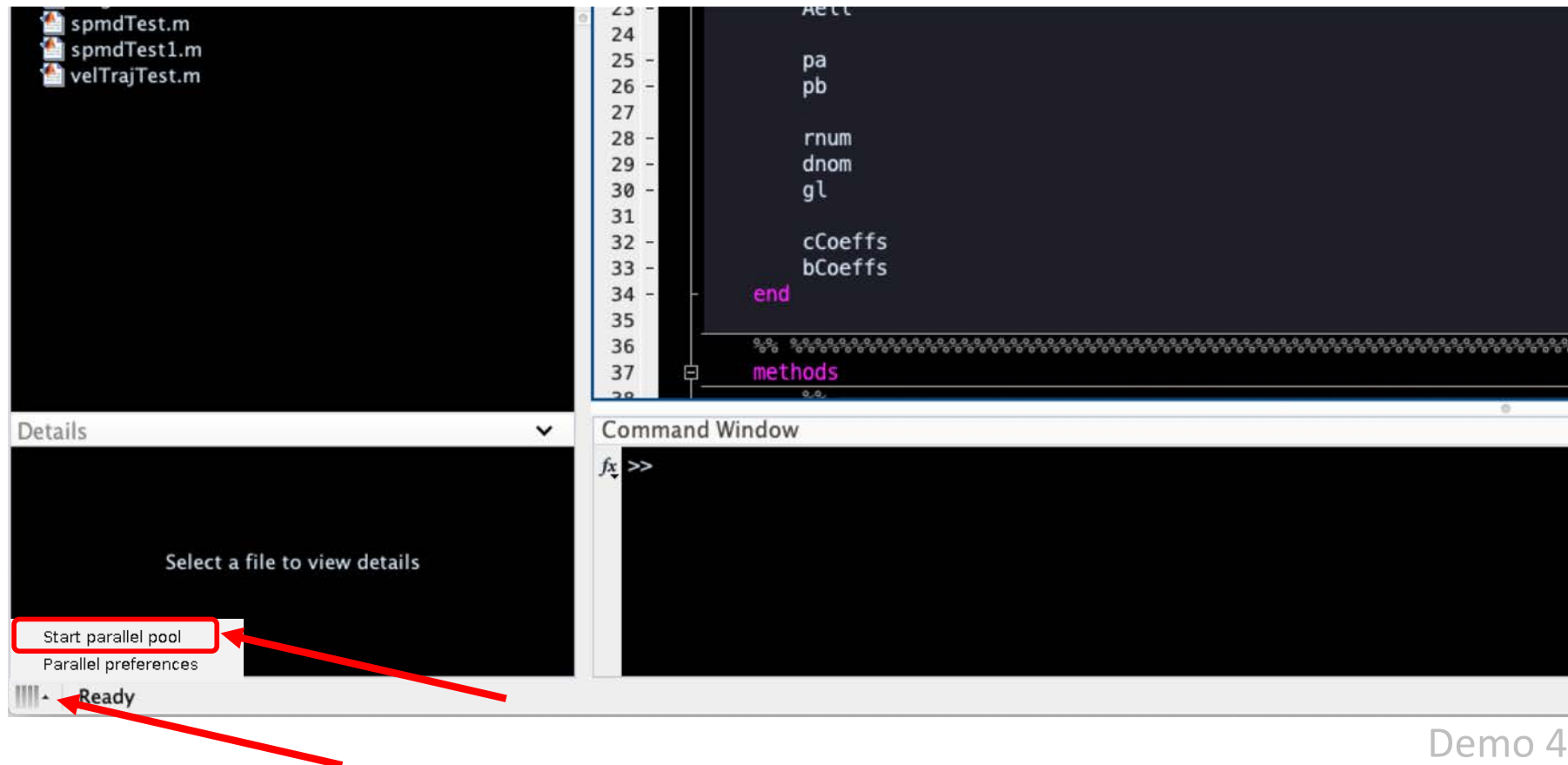
Cluster profile settings

- Limit number of workers
 - Normally not needed
- Not sure how this works exactly
- Might become important when using functionality external to MATLAB (MEX etc.)
 - Otherwise leave alone

WORKERS

Range of number of workers to run job NumWorkersRange	[1 inf] (default)
Return command window output CaptureDiary	false (default)
Manually specify environment variables to copy from client to workers (One entry per line) EnvironmentVariables	<none>

Starting a pool from GUI



MATLAB commands for pool control

- Everything discussed so far can be set with MATLAB commands
- Object-oriented:
 - Parallel pool object
 - Cluster object
- Important functions:
 - `p = parpool()` starts a new pool
 - Optionally specify profile or workers or both:
`p = parpool('horus', 4)`
 - `p = gcp()` → get current pool
 - `c = parcluster('profile')` → get cluster object
 - Edit SLURM settings: `c1.SubmitArguments = <string>`

Demo 5

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Exercise 2

- Objectives
 - You understand the parallel pool settings
 - You have functioning local and Horus cluster profiles
- Tasks
 - Open the cluster profile manager and examine the profile settings
 - Import the horus_m2019 cluster profile (available at:
<https://cluster.uni-siegen.de/anwendungssoftware/matlab/#mdcs>)
 - Launch a pool with the GUI button
 - Launch pools with different settings programmatically

Note the following slide (before you begin the exercise)

Exercise 2

- Notes
 - You should launch jobs in the `short` queue and with short walltimes to await waiting times
- If bored, get creative
 - Examine `parcluster` and `parpool` object member variables
 - Open another Linux console with `top`, then run compute-intensive commands in MATLAB and watch CPU usage
 - Examine the MATLAB “Parallel Preferences”
 - ...

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - **Parallel programming in MATLAB**
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Distributing tasks to workers

- Parallel functions (and associated concepts) in MATLAB
 - Essentially mirrors usual parallel programming approaches
- Send task manually (not covered): `batch()`, `parfeval()`
- Parallel for loop: `parfor`
- Parallel program: `spmd`
- Interactive console on multiple workers: `pmode`
- Many MANY more
 - E.g. Simulink: `parsim/batchsim`

The batch command

- `batch` command sends one task to one worker
- “Non-blocking”: you can keep working on MATLAB client console
- Important concept: “future”
 - Worker completes work
 - Stores results
 - Future object is how to get results back
- Job “diary”: log file etc.

Parallel for loops

- Very straightforward idea:
 - Take a regular `for` loop
 - Distribute loop iterations among workers
 - Multiple iterations get done at the same time
- Simple to apply to existing code in principle...
 - Will even start pool itself (not on HoRUS)
- ...however there are some complications
 - Loop iterations have to be 100% independent, many implications
- MATLAB documentation very helpful here

Parallel for loops

- Basic `parfor` example

```
tic
n = 200;
A = 500;
a = zeros(1,n);
parfor i = 1:n
    a(i) = max(abs(eig(rand(A)))));
end
toc
```

- Will run a few seconds
- Note `tic` and `toc` for time measurement

<https://www.mathworks.com/help/parallel-computing/parfor.html>

Parallel for loops: complications

- Five different types of variables
- MATLAB has to tell which variable is which type
 - Does it exist once or is it copied on each worker?
 - Does it change during the loop?
 - What belongs to which worker?
- Error message if MATLAB cannot tell type
- Overall rule of thumb: the simpler the loop, the fewer problems
- Also useful for explaining parallel concepts in general

Parallel for loops: variable types

1. Loop variable
2. Sliced variable
3. Broadcast variable
4. Reduction variable
5. Temporary variable

```
a = 0;  
c = pi;  
z = 0;  
r = rand(1,10);  
parfor i = 1:10  
    a = i;          ← loop variable  
    z = z+i;        ← reduction variable  
    b(i) = r(i);    ← sliced input variable  
    if i <= c        ← broadcast variable  
        d = 2*a;    ← temporary variable  
    end  
end
```

<https://www.mathworks.com/help/parallel-computing/troubleshoot-variables-in-parfor-loops.html>

Parallel for loops: loop variables

1. Loop variable

- Loop index
- Must be consecutive increasing integer
- Cannot break out of loop early
 - Loop iterations executed in parallel
 - Order not clear
 - What is “early”?
- Cannot be modified

```
parfor i = 0:0.2:1      % not integers
parfor j = 1:2:11       % not consecutive
parfor k = 12:-1:1      % not increasing
```

<https://www.mathworks.com/help/parallel-computing/loop-variable.html>

Parallel for loops: sliced variables

2. Sliced variable

- Exists only once
- Distributed over workers
 - Each worker gets different slice
- May be written to inside loop
- MATLAB has to know how to distribute

```
A = rand(1,10);  
parfor ii = 1:10  
    B(ii) = A(ii);  
end
```

<https://www.mathworks.com/help/parallel-computing/sliced-variable.html>

Parallel for loops: broadcast variables

3. Broadcast variable

- Existing variable before `parfor` block
- Copied to each worker
- Cannot be modified inside `parfor`
- Caution with large variables

```
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;
    z = z+i;
    b(i) = r(i);
    if i <= c
        d = 2*a;
    end
end
```

Diagram illustrating variable types in the `parfor` loop:

- temporary variable**: points to `a = i;`
- reduction variable**: points to `z = z+i;`
- sliced output variable**: points to `b(i) = r(i);`
- loop variable**: points to `i = 1:10`
- sliced input variable**: points to `r(i)`
- broadcast variable**: points to `c` in `if i <= c`

<https://www.mathworks.com/help/parallel-computing/broadcast-variable.html>

Parallel for loops: reduction variables

4. Reduction variable

- “Reduction”: accumulate a value from each parallel task
 - Example: sum up value over each loop iteration
- Must be specific operation
- Cannot be dependent on iteration order

$$X = X + \text{expr}$$

$$X = X - \text{expr}$$

$$X = X .* \text{expr}$$

$$X = X * \text{expr}$$

$$X = X \& \text{expr}$$

$$X = X | \text{expr}$$

$$X = [X, \text{expr}]$$

$$X = [X; \text{expr}]$$

$$X = \min(X, \text{expr})$$

$$X = \max(X, \text{expr})$$

$$X = \text{union}(X, \text{expr})$$

$$X = \text{intersect}(X, \text{expr})$$

<https://www.mathworks.com/help/parallel-computing/reduction-variable.html>

Parallel for loops: temporary variables

5. Temporary variable

- Exists in each loop iteration separately
- Deleted at end of iteration
- If it exists before `parfor`, it is overwritten

```
a = 0;  
z = 0;  
r = rand(1,10);  
parfor i = 1:10  
    a = i;           % Variable a is temporary  
    z = z + i;  
    if i <= 5  
        d = 2*a;     % Variable d is temporary  
    end  
end
```

<https://www.mathworks.com/help/parallel-computing/temporary-variable.html>

Code example: parallelizing a for loop

- LES matrix is filled row by row in loop
 - Rows independent: good candidate for parallelization
- Influence data calculation (all panels on one point)
 - Self-contained operation
- Result is copied into B and C matrices

Code example: parallelizing a for loop

```
% calcMatsFor()
for iTarget=1:nTarget
    xTarget = infl.panels.centers(iTarget,:);
    infl.calc( xTarget );

    cMat(:,iTarget) = infl.cCoeffs;
    bMat(:,iTarget) = infl.bCoeffs;
end
cMat = cMat.';
bMat = bMat.';
```

```
% calcMatsParfor()
parfor (iTarget=1:nTarget,nWorkers)
    xTarget = infl.panels.centers(iTarget,:);
    infl.calc( xTarget );

    cMat(:,iTarget) = infl.cCoeffs;
    bMat(:,iTarget) = infl.bCoeffs;
end
cMat = cMat.';
bMat = bMat.';
```

- Working `parfor` loop really possible with no changes
- Note how matrix is filled by column, then transposed

MATLAB pmode: introduction

- Parallel mode (`pmode`) is a parallel interactive console
 - Start with `pmode start`
 - Will open new window and start pool
- Similar to regular MATLAB console
 - But sends command to each worker
- Also understands constructs like codistributed arrays
- Extremely practical for understanding parallel concepts
 - Also for debugging

Demo 6

MATLAB spmd construct: background

- SPMD: general parallel programming term
 - Single-Program Multiple-Data
- Program is written once, run multiple times in parallel
- Trivial case: each instance does exactly the same thing
 - More common: each does same task for different part of input data
 - However anything is possible
- Often includes communication between instances

MATLAB `spmd` construct: introduction

- MATLAB: `spmd` block
 - Block contents run on workers
- More complicated, but also more flexible than `parfor`
- Variables either copied or distributed over workers
 - Similar to `parfor`, but different mechanisms
- Important tools: `labindex`, `numlabs`
 - Give ID of worker and total number of workers respectively
 - Important to determine which worker works on what

MATLAB spmd: codistributed arrays

- Important concept: codistributed array
 - Array (or n-D matrix), treated mostly like regular array by MATLAB
 - But each part lives on different worker
 - User can say which part lives where
 - Parallel programming term: Partitioned Global Address Space (PGAS)
- Distinction: distributed vs. codistributed array
 - Distributed arrays not covered
- Also common: composite objects
 - Object distributed over workers
 - Not covered here

MATLAB codistributed arrays

- SPMD construct requires moving data to workers manually
- Codistributor object: contains information on how array is distributed
 - Can be along any axis (`codistributor1d`)
 - Can also be block-cyclic (`codistributor2dbc`)
- Is otherwise treated like normal array
 - Makes things easier for user
 - Sometimes tricky to keep track which array is codist or not
- SPMD performance: need to make local copy

Demo 7

MATLAB SPMD code example

- Info about array distribution is retrieved
 - Which rows are on this worker?
- Local copies of part of array
 - Codist-array could be addressed with `(:, :)`
 - But: prohibitively expensive
- Local copies are written back to original array

```
spmd
    codist1 = getCodistributor(cMat);
    iGlob = codist1.globalIndices(1);
    cLocal = getLocalPart( cMat );
    bLocal = getLocalPart( bMat );
    nLocal = size(cLocal,1);

    for iTarget = 1:nLocal
        xTarget = infl.panels.centers(iGlob(iTarget),:);
        infl.calc( xTarget );

        cLocal(iTarget,:) = infl.cCoeffs;
        bLocal(iTarget,:) = infl.bCoeffs;
    end

    cMat(iGlob,:) = cLocal;
    bMat(iGlob,:) = bLocal;
end
```

Second code example: solving the LES

- In addition to filling the LES, solving has to also be parallelized
- Even serial version needs modification (input data could be codistributed)
- Parallel solver needs to be inside spmd block
- Data needs to be scattered, then gathered again

```
% calcDoublets
doublets(:) = gather( mldivide(cCoeffs,rhs) );

% calcDoubletsParallel
spmd
    cCoeffsDist = codistributed(cCoeffs);
    rhsDist = codistributed(rhs);
    solDist = mldivide(cCoeffsDist,rhsDist);
end
doublets(:) = gather( solDist(:) );
```

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Exercise 3

- Objectives
 - You understand the `parfor` construct
 - You understand the `spmd` construct and codistributed arrays
- Tasks
 - Open `pmode` and create some codistributed arrays
 - Implement the `parfor` example (Appendix A) and run it
 - Implement the `spmd` example (Appendix B) and run it

Note the following slide

Exercise 3

- Notes:
 - You can leave out the commented code in the examples for now
- If bored, get creative
 - Add new variables into parfor
 - Add new non-distributed, distributed and codistributed arrays to both examples
 - ...

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - **Profiling basics**
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Serial Performance, profiling

- Never optimize without measurement
- Simple stopwatch: `tic` and `toc`
 - `tic` starts timer, `toc` ends it
 - Can have multiple timers
- MATLAB profiler:
 - Simply use `profile on` and `profile off`
 - How often was each function called, how much time was spent in it
 - Save HTML files with `profsave`
 - Complication for cluster: MATLAB directory is hardcoded in HTML → image files not found

Demo 8

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - **Serial performance**
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- Summary

Serial Performance

- How do I make MATLAB faster?
 - Throw out MATLAB and use a real programming language!
 - Barring that, use built-in functions
- Remember: never optimize without measurement!
- Common performance wisdoms in other languages:
 - Minimize reallocation (e.g. C)
 - Avoid explicit loops (e.g. Python)
 - Use library functions (BLAS, LAPACK)
 - Be aware whether the language is row-major or column-major

Serial Performance, net vs. gross memory

- What is memory use of a Matlab array?
 - Number of entries times 8 byte (double precision float)
 - Plus overhead
 - So what is overhead?
- Gross memory usage: Matlab will not tell you, but OS will
 - Mac OS and Linux: 1.5 - 2 GB when idle
 - Rough approximation

Serial Performance, common wisdoms

- Test: how much is the difference
 1. Minimize allocations, reallocations, copies
 2. Avoid explicit loops
 3. Use built-in functions
- Profile built-in function vs. explicit loops
 - With and without reallocation
- Result:
 - Built-in functions two orders of magnitude faster
 - Reallocation vs. no reallocation makes little difference

Demo 9

Serial Performance, common wisdoms

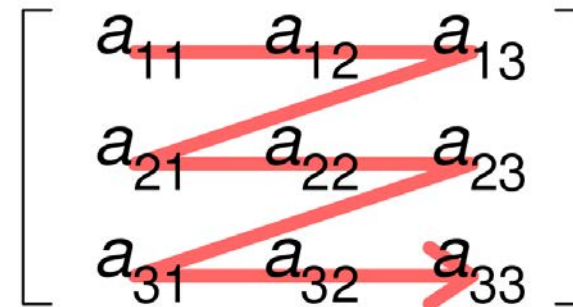
- Minimize allocations, reallocations, copies
 - Difference less than expected
 - That is not a good thing
- Avoid explicit loops
 - Oh boy, yes!
- Use built-in functions
 - Yes (because it's not MATLAB)

Best performance when using obvious way (that is a good thing)

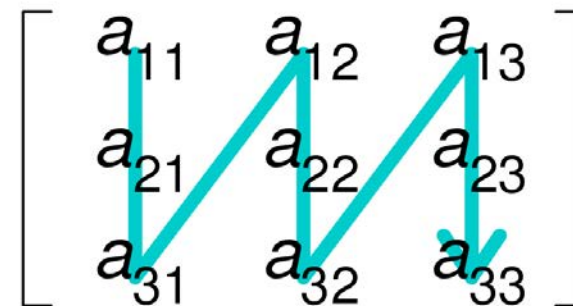
Serial Performance, common wisdoms

- Be aware whether language is row-major or column-major
 - Official: Matlab is column-major (like Fortran)
 - Test: flipped all indices
 - Reality: “right way” is slower
 - Sometimes same time
 - Only exception: `vecnorm()` faster

Row-major order



Column-major order



Serial Performance, other stuff

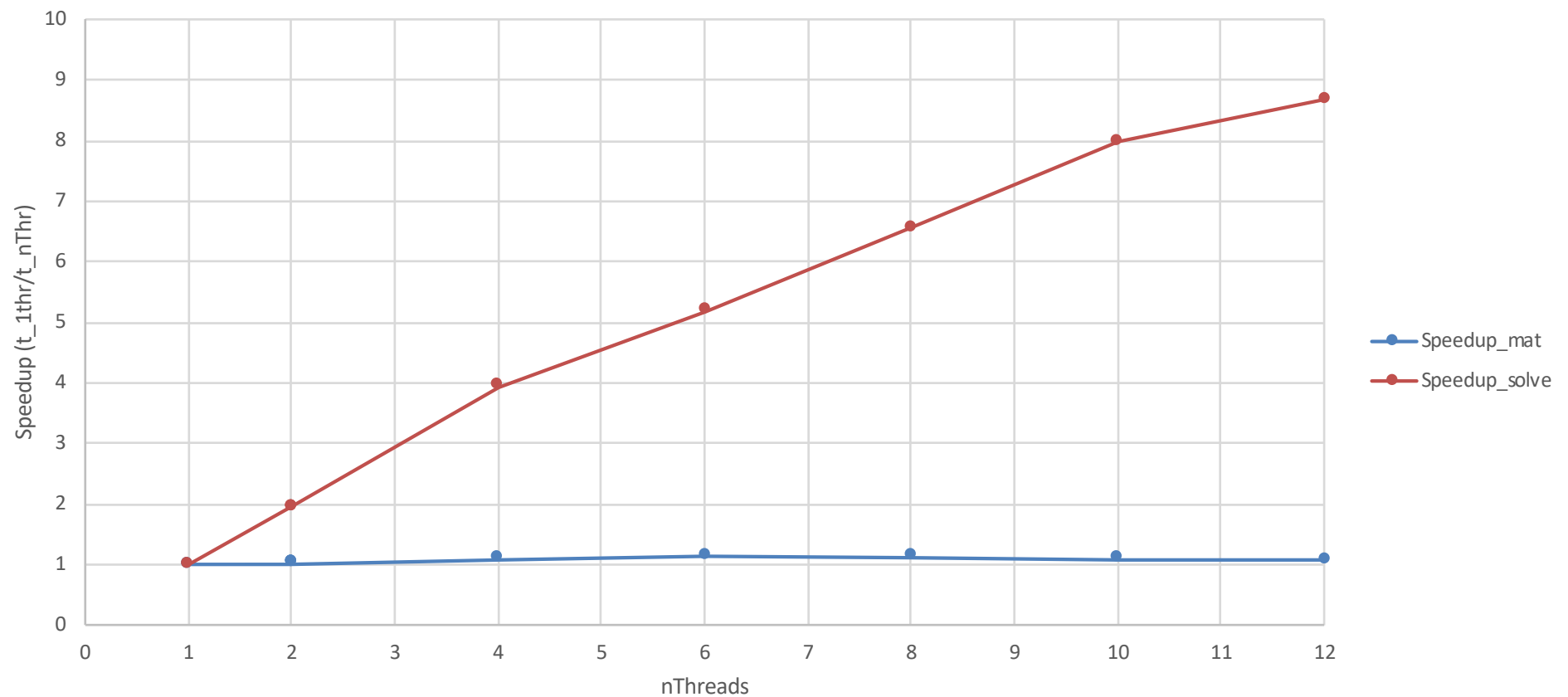
- n threads never means n times the performance
 - Overhead
 - General problem in HPC
- Best friend for debugging: `maxNumCompThreads (n)`
 - Sets number of threads to use
 - Default same as CPUs available

Serial Performance, panel3d Code

- Run panel3d code with no parallelization
- Measured on HoRUS, 1 node
- Varied N
- Varied `maxNumCompThreads`
- Recorded compute time with `tic` and `toc`
- Read CPU and RAM use from `top`

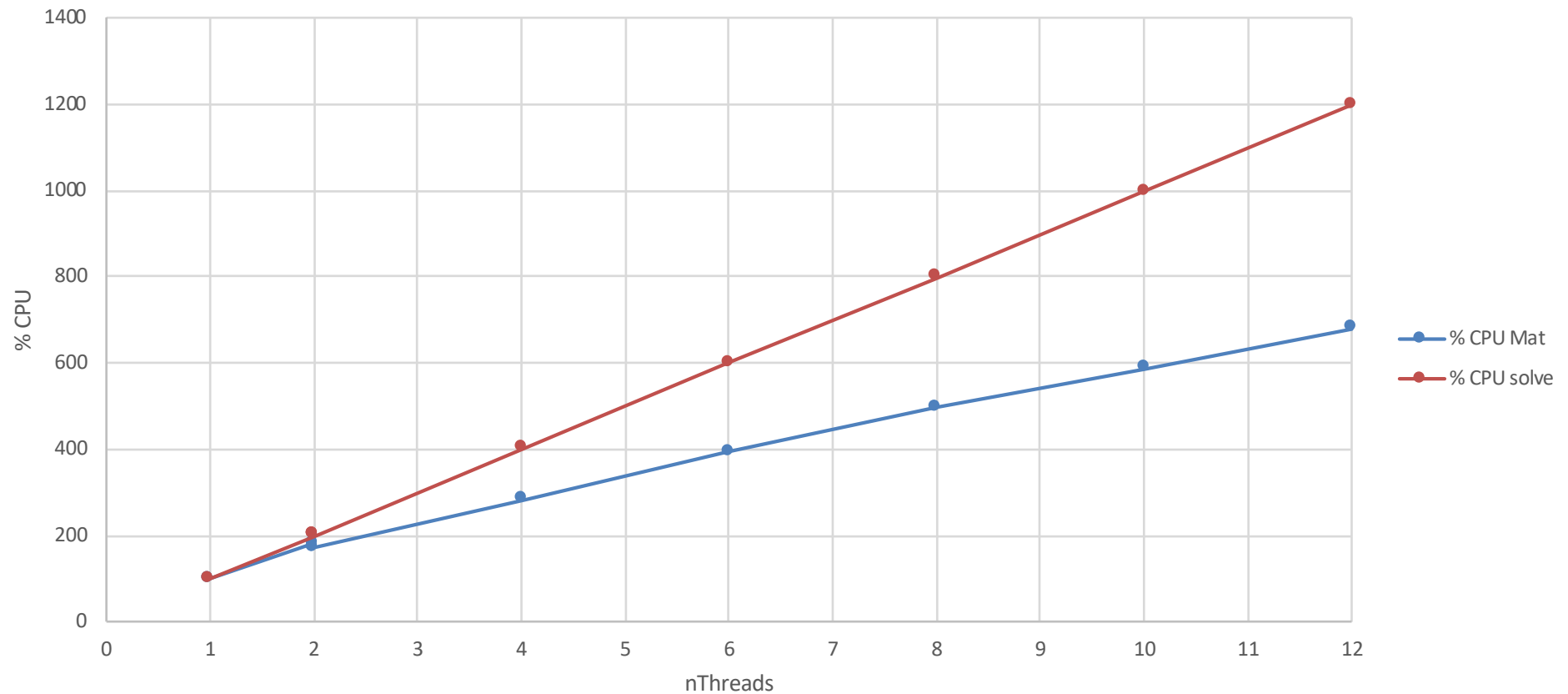
Serial Performance, panel3d Code

Speedup panel3d with MATLAB. N=10000



Serial Performance, panel3d Code

CPU load running panel3d with MATLAB, N=10000



Serial Performance, panel3d Code

- Speed:
 - Building up the matrices: barely scales with threads
 - Best performance at 6 cores (number of cores on 1 socket)
 - Solve LES: scales almost perfectly (ScaLAPACK shipped with MATLAB)
- Memory
 - LES buildup: memory overhead present but reasonable
 - CLI MATLAB has about 650 MB base RAM usage
 - Total measured: 2.4 GB RAM (theoretical 1.6 GB)
 - LES solve: 3.2 GB RAM
 - Algorithm?

Outline

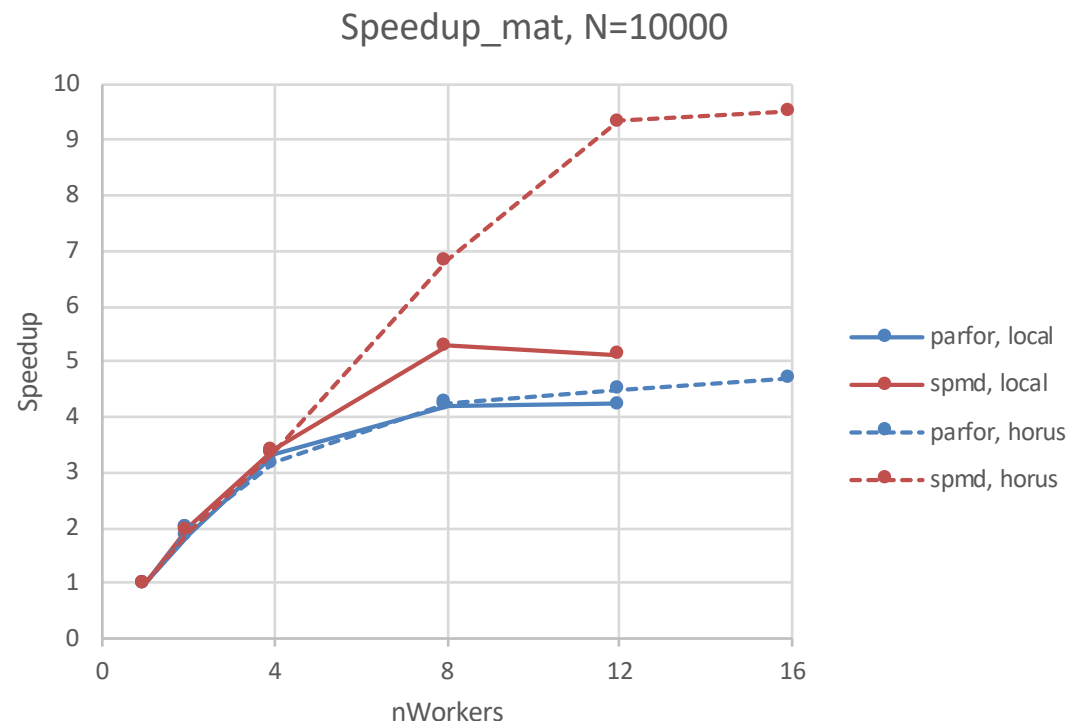
- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - **Parallel performance**
 - *Exercise 4*
 - Mex functions
- Summary

Parallel performance

- Divide work up to workers
 - Building LES
 - Solving LES
- Try different strategies
 - `parfor`
 - `spmd`
 - Local vs. Horus cluster profile
- $N = 10000$, Number of workers 1, 2, 4, 8, 12, 16
- Determine speedup

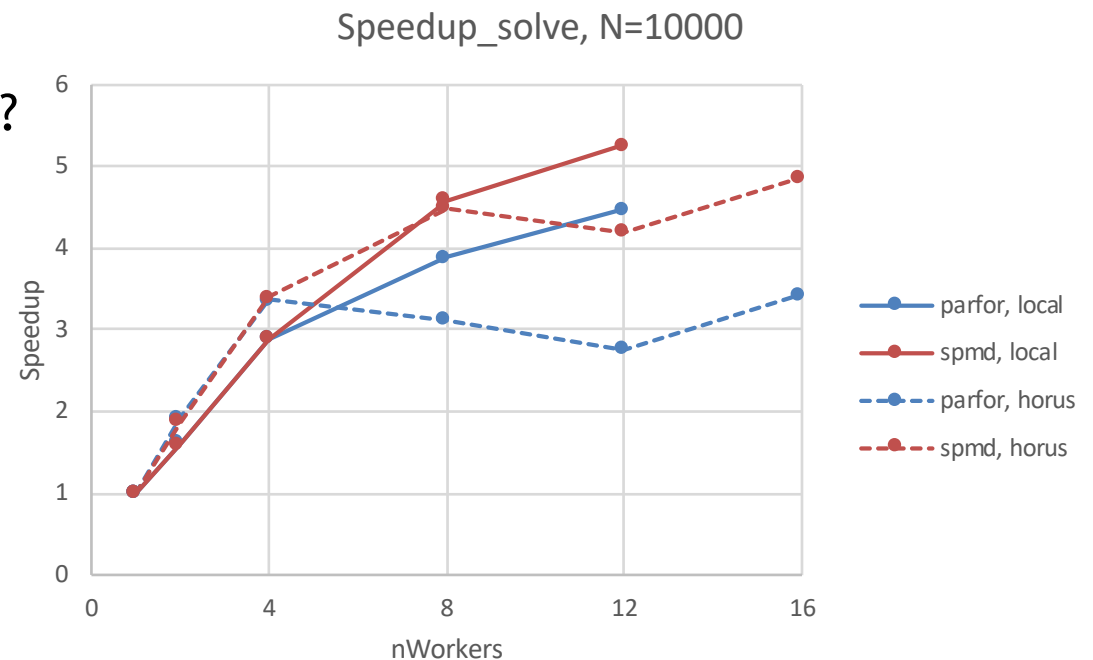
Speedup of matrix building operations

- Speedup has plateau
 - Local profile: much earlier
- Most likely some unparallelizable operations
 - Does not happen with $N = 20000$
- `spmd`-based operation performs better
 - Unclear cause



Speedup of matrix building operations

- Starts to plateau with local profile
 - Unclear cause
 - Communication overhead of alg.?
- Considerably lower performance with `parfor` and Horus profile
 - Strange: this is always an `spmd`, not a `parfor`
- Reason:
 - Automatic array distribution in `parfor`
 - Needs to be redistributed



Parallel Performance

- Both `parfor` and `spmd` accelerate calculation
- All less than n times serial
- `spmd` slightly faster
- Larger matrices possible with `spmd`
 - Likely due to distributed matrices

mpiprofile

- At this point, just a shoutout to `mpiprofile`
- I have not seen such a comfortable tool for parallel performance measurement in any language

Demo 10

spmd Lessons Learned

- Always keep track of whether a variable is distributed
- Assigning will overwrite
 - Assigning like `a(:) = b` will give error when `b` is distributed
- Old Matlab rule about matrix operations rather than loops still holds
 - Setting and accessing entries: prohibitively expensive
 - Better: copy to local variable
- But: basically every Matlab function works with distributed arrays (!)
 - Unsure which matrix distribution is best

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - ***Exercise 4***
 - Mex functions
- Summary

Exercise 4

- Objectives
 - You can use the different profiling methods
 - You can interpret serial and parallel profiler outputs
- Tasks
 - Add the commented profiling commands in the `s_pmd` example (Appendix B) one by one, run the code and examine the output
 - tic and toc
 - profile
 - mpiprofile

Note the following slide (before you begin the exercise)

Exercise 4

- Notes
 - You can use the `return` command inside a script to exit it early
- If bored, get creative
 - Add different functions and operation to the statements
 - Start profiling your own MATLAB code
 - Implement and play around with the built-in function example (Appendix C)
 - ...

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - **Mex functions**
- Summary

What to do about this

- Matrix calculation was tough to optimize
 - Limits of MATLAB reached?
- Idea: delegate matrix calculation function to external Fortran program
- MATLAB has interface for this
- MATLAB extension function: Mex

Mex interface

- MATLAB has loads of interfaces
 - Primarily to C and C++
 - Also Python
 - Also Fortran
 - Both ways (run MATLAB from another language)
- Slightly confusing naming principle
 - Matrix vs mex interface?
 - Call libraries vs. call executables
- We will only look at Fortran mex interface

<https://www.mathworks.com/help/matlab/external-language-interfaces.html>

Mex interface

- To investigate:
 - Fortran or C (or C++) interface more comfortable?
 - How much programming knowledge needed?
 - How well integrated is `mex`?
 - How much better is performance
- What is mex exactly?
 - Interface to system C or Fortran compiler (Linux: gcc)
 - Stand-alone executable (outside MATLAB)
 - Mex interface: list of C and Fortran functions to put in own code

Interfacing Fortran code with MATLAB

- Strategy:
 - Matrix calculation slow and badly parallelized
 - panel3d Fortran version at least twice as fast as fastest MATLAB version
 - Use `calcMats()` function from Fortran
- What needs to be given to Fortran
 - Input: `InflData` object
 - Output: B and C matrices
- Idea: implement this interface in Fortran (code exists)
 - Fortran because matrix syntax similar to MATLAB
 - Future: Also C and/or C++

Interfacing Fortran code: approach

- Mex principle:
 - Single Fortran routine called `mexFunction()` acts as entry/exit point
 - Use as wrapper for existing own code
 - Decode arguments
- What needs to be given to Fortran
 - Input: `InflData` object
 - Output: B and C matrices

Interfacing Fortran code: own code

- Fortran code very similar to MATLAB
- Did not take long to convert
- Parallelization in this case similar to parfor logic
- Row-major vs. column-major more important

```
!$omp parallel do firstprivate(infl, xTarget) shared(pd)
do iTarget = 1,n

    xTarget = pd%centers(:,iTarget)
    call infl%calc(xTarget)

    call calcCCoeffs_obj(xTarget,infl,cMat(:,iTarget) )
    call calcBCoeffs_obj( infl, cMat(:,iTarget), bMat(:,iTarget) )

    if ( mod(iTarget,max(10,n)/10) == 0 ) write(*,*) "    Reached panel", iTarget, &
        | & "on thread", omp_get_thread_num()

end do
!$omp end parallel do

cMat = transpose(cMat)
bMat = transpose(bMat)
```

Hello World mexFunction breakdown

- Single include
- Pointer to left-hand (output) and right-hand-side (input) args
 - Number of args
 - Requires checks for everything
- Pointer data type
- mex-specific library function

```
#include "fintf.h"
subroutine mexFunction(nlhs, plhs, nrhs, prhs)

    implicit none

    mwPointer :: plhs(*), prhs(*)
    integer :: nlhs, nrhs

    call mexPrintf( "Hello MATLAB world, this is Fortran!\n" )

end subroutine mexFunction
```

Decoding and re-encoding arguments

- This function gets one array from args
- Requires 4 steps
- Requires copying array
 - Have not found better way
- Objects: repeat this for every member variable

```
subroutine getDbleMatFromMatlabPtr( ptr, mat )  
!-----  
mwPointer, intent(in) :: ptr  
real(8), dimension(:,,:), allocatable, intent(inout) :: mat  
!-----  
integer :: nrows, ncols  
mwPointer :: dble_ptr  
!-----  
  
nrows = mxGetM( ptr )  
ncols = mxGetN( ptr )  
  
allocate( mat( nrows, ncols ) )  
  
dble_ptr = mxGetDoubles(ptr)  
  
call mxCopyPtrToReal8(dble_ptr, mat )  
  
end subroutine getDbleMatFromMatlabPtr
```

But: works, not complex if you understood it once

Simplifying mex interface

- MATLAB File Exchange: Fortran 95 interface “with extras”
 - <https://www.mathworks.com/matlabcentral/fileexchange/25934-fortran-95-interface-to-matlab-api-with-extras>
 - Community effort, not officially supported
 - Have not tried yet
- Also: pointer interface is C logic
 - Clearly, C support for mex came first
 - More comfortable?
 - Problem: matrix handling in C less comfortable

MATLAB C++ Interfaces

- Multiple C++ APIs, confusing naming
 - C++ mex API: define C++ functions, call them from MATLAB
 - C++ Data API: use MATLAB data structures in C++ code (arrays)
 - C++ Engine API: talk to MATLAB from C++ (call functions)
 - C API: separate, not covered
- Performance not yet tested, therefore only how comfortable is it to use?

MATLAB C++ Interfaces

- Advantages:
 - C++ features very clean and well designed (understood it quickly)
 - Much more comfortable for C++ programmers
 - Object-oriented approach, including exceptions
- Disadvantages:
 - C++ more strict about datatypes (Fortran too by the way)
 - Unclear how much overhead
 - Resulting arrays are not C++-native, limited by implementation
 - No functions like *sin*, *cos*, *sqrt*
 - Everything needs to be a loop

MATLAB C++ Interfaces

Headers come with
MATLAB

Practical
namespaces

```
#include "mex.hpp"
#include "mexAdapter.hpp"

using matlab::engine::MATLABEngine;
using matlab::data::Array;
typedef matlab::data::TypedArray<double> DblArray;
typedef matlab::data::TypedArray<bool> BoolArray;
typedef matlab::data::TypedArray<int64_t> Int32Array;

class MexFunction : public matlab::mex::Function {
public:
    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs)
    {
        std::cout << "Hello MATLAB world, this is C++!" << std::endl;
        const Array& inflIn = inputs[0];
        const DblArray& xTarget = inputs[1];
        const Engine engine = getEngine();

        InflData infl( engine, inflIn );
        infl.calc(xTarget);
    }
};
```

Code comparison

MATLAB

```
function pVec = calcPVectors(centers,xTarget,pVec)
    pVec(:, :) = xTarget - centers;
end
```

Fortran

```
subroutine calcPVectors(centers,xTarget,p)
    !-----
    real(8), dimension(:, :), intent(in) :: centers
    real(8), dimension(NDIMS), intent(in) :: xTarget
    real(8), dimension(size(centers,1),NDIMS), intent(inout) :: p
    !-----
    integer :: iPanel
    !-----

    do iPanel = 1,size(centers,1)
        p(iPanel,:) = xTarget(:) - centers(iPanel,:)
    end do

end subroutine calcPVectors
```

Code comparison

MATLAB

```
function pVec = calcPVectors(centers,xTarget,pVec)
    pVec(:, :) = xTarget - centers;
end
```

C++

```
static void calcPVectors( const DblArray& centers, const DblArray& xTarget, DblArray& pVec)
{
    for (size_t i = 0; i < pVec.getDimensions()[0]; i++)
    {
        for (size_t j = 0; j < pVec.getDimensions()[1]; j++ )
        {
            pVec[i][j] = xTarget[0][j] - centers[i][j];
            std::cout << pVec[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Calling mex

- Simple command line tool
 - Already available on cluster
 - Even `cmake` knows MATLAB and mex (!)
- One call:

```
mex -I../Release_horus ../src/mex/calcMatsMex.F90
```

 - Include directory with own code
 - Needs to be compiled separately
- Easy setup compared to usual C or Fortran compilers
 - Complexity is only in the way interface works

Summary of mex experiments

- MATLAB code can potentially benefit greatly from external code
 - Several times faster
 - Complete arsenal of optimizations (vector registers etc.)
 - May parallelize with OpenMP and MPI
- Fortran interface is extremely awkward
 - Clearly a C relic
- Getting mex to run is however very simple
 - Already installed
 - Minimal includes/library links etc.

Recommendations for mex use

- Minimize interface interaction
 - Few input/output arguments
 - Simple types (arrays)
- Choose interface wisely
 - Which language do you know better?
- Good candidates for delegation to other language
 - Function that takes some arrays, has big loop(s), outputs some other arrays

Outline

- Part 1: Introduction
 - Background
 - MATLAB at Uni Siegen
 - MATLAB parallel features
- Part 2: MATLAB parallelization
 - Using MATLAB on the cluster
 - *Exercise 1*
 - Parallel pools and cluster profiles
 - *Exercise 2*
 - Parallel programming in MATLAB
 - *Exercise 3*
- Part 3: Performance
 - Profiling basics
 - Serial performance
 - Parallel performance
 - *Exercise 4*
 - Mex functions
- **Summary**

Summary

- Wide variety of parallel options
 - Comfortable profiling and debugging (mostly)
- Built-in functions often already optimized
 - Two orders of magnitude when written in another language
- Built-in multithreading
 - Helps a lot
 - “Free” (no additional coding)
- `parfor`
 - Easy to implement
 - Not transparent

Summary

- `spmd`
 - Lots and lots of possibilities
 - Complex
 - Lots of potential for errors
- Scaling good for built-in functions, bad for own functions
- `mex`
 - Initially tricky
 - Ultimately not hard
 - Potentially huge benefit
 - Fortran: code very similar

Recommendations

- Best approach for writing your code:
 1. Use MATLAB for code until it works (rapid prototyping)
 - Focuses on mathematical operations
 - Comfortable debugging
 - Built-in visualization
 2. Once it works, analyze performance
 3. Identify functions where most time is spent
 - Simple in programming terms (likely complex in mathematical terms)
 - Big long loops inside
 4. Delegate these to `mex` function
 - Feasible(!) after initial complexity
 - Fortran: similar to MATLAB syntax, C closer to interface logic

Thank you for your attention

Questions?

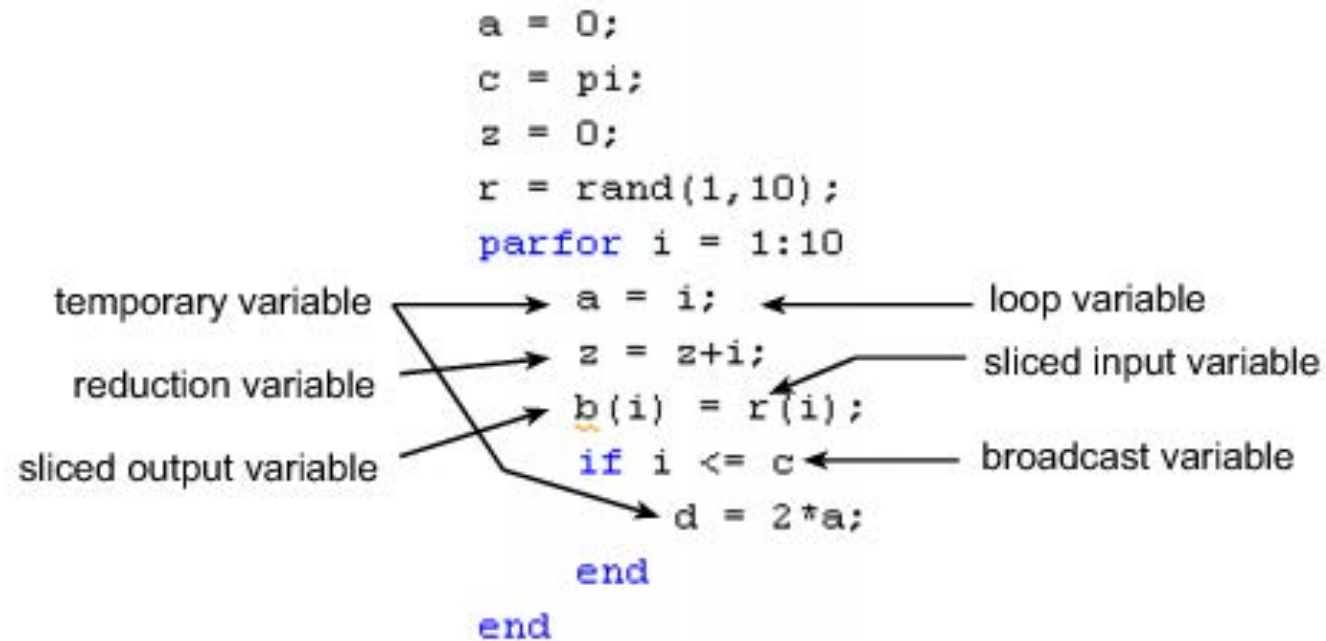
Feedback round

- What were your expectations, and where they fulfilled?
- What was your favorite part about the course?
- What did you dislike or what do you feel can be improved?
- How did you learn about this course?
- What other topics would you like to see in future ZIMT courses?

Appendices: code examples

- Appendix A: parfor example
- Appendix B: SPMD test example
- Appendix C: built-in functions performance example

Appendix A: parfor example



Appendix B: spmd example

```
n = 10000;

% maxNumCompThreads(2);
maxNumCompThreads

As = rand(n);
rs = rand(n,1);

% profile on
% tic
xs = As \ rs;
xs = vecnorm(As,2,1);
% toc
% profile viewer

spmd
%     codist1 = codistributor2dbc(codistributor2dbc.defaultLabGrid, ...
%                               codistributor2dbc.defaultBlockSize, ...
%                               'col');
codist1 = codistributor1d(1);

Ap = rand(n,codist1);
rp = rand(n,1,codist1);

%     mpiprofile on
%     tic
xp = Ap \ rp;
x = vecnorm(A,2,2);
%     toc
%     mpiprofile off
%     mpiprofile viewer;
end
```

Appendix C: built-in functions performance example

```
clear;

n = 20000;
A = rand(n);
b = rand(n,1);
c = zeros(1,n);

%% Without reallocating c
tic
c(:) = A * b;
toc

%% With possible reallocation of c
tic
c = A * b;
toc

%% With newly allocated result
tic
c2 = A * b;
toc

%% Manual matrix multiplication
tic
for i = 1:n
    for j = 1:n
        c(i) = c(i) + A(i,j) * b(j);
    end
end
toc

%% Manual matrix multiplication with reallocation in each step
tic
for i = 1:n
    c3(i) = 0.0;
    for j = 1:n
        c3(i) = c3(i) + A(i,j) * b(j);
    end
end
toc
```