Welcome to this online Hábrók course!

The course will start at 13:00.
In the meantime, please make sure that your audio and video work.

Please use your real name and mute your mic when you are not speaking. If you want to ask something, press the "raise hand" button.

More information about Kaltura Classroom:
https://edusupport.rug.nl/2477457409

Find all course material at:
https://wiki.hpc.rug.nl/habrok/additional_information/course_material

university of groningen / center for information technology

part I

High Performance Computing

Large scale computations and data analysis on the Hábrók cluster (Advanced)

 Fokke Dijkstra

 Cristian Marocico

 Bob Dröge

 Pedro Santos Neves

Introduction

Part 1: methods for submitting many runs

- Bash scripting
- Job arrays
- File systems

Part 2: improve the performance of a single run

- single machine: 1 Core
- single machine: n Cores - OpenMP
- multiple machines - MPI
- Accelerators for extra performance: GPUs

Part I

1. Bash scripting
2. Job arrays
3. File systems

- Bash vs. Python
- Variables
- Script arguments
- Command substitution
- If / else
- Loops
- Arrays
- Useful commands

Bash scripts

Bash	Python (or others)
✓ Easy to start external programs	✗ Some simple tasks take more effort
✓ Direct file management	
✓ Use many unix shell tools	✓ Real programming language
✓ Glueing programs together	✓ Many libraries available
✗ Syntax can get complex quickly	✓ More understandable/readable code
✗ Error prone	

Bash: Shebang

```
#!/bin/bash
```

- This line starts all bash scripts
- It tells the operating system (OS) to use /bin/bash to execute the lines below the shebang

- E.g. for Python one could use:
  `#!/usr/bin/python`

Execute permission

- A new file will not be executable

```
./submit.sh
-bash: ./submit.sh: Permission denied
ls -l submit.sh
-rw------- 1 f111536 f111536 131 30 mei 15:51 submit.sh
```

Linux will not search the current directory for executables

- We can modify the permission bits and add the execute flag using chmod:

```
chmod +x submit.sh
ls -l submit.sh
-rwx--x--x 1 f111536 f111536 131 30 mei 15:51 submit.sh
```

- Note that the `.sh` extension is not obligatory. But a good practice!

Bash: Variables

- Variables hold values that can be used later.
- Can contain information from the system or other tools
- Prefixed by `$`
- Can be surrounded by `{}`, eg. `${HOME}`
  - To distinguish variable name from other text.

Examples of predefined variables:

Variable	Meaning
$HOME	Path to user's home directory
$USER	Username of current user
$PWD	Current working directory
$$	Process id of current script
$?	Exit status of last command run

Setting variables

Set variable:
`variable=value`

Export it to child scripts:
`export VARIABLE=value`

Refer to it using
`$variable or ${variable}`

Case-sensitive!

Example:

```
retries=100

echo $retries
100

echo ${retries}x10
100x10
```

Script arguments

Supply information to a script using arguments

```
./script.sh testfile.txt 20
```

Retrieve the information using:

`$0`	Name of the script
`$1`	First argument, e.g. `testfile.txt`
`$2`	Second argument, e.g. `20`
`$3`	…
`$#`	Number of arguments, e.g. 2

Bash: Command substitution

- Sometimes you want to capture the output of a command
- This can be done using `$( command )`

E.g capture a list of files:

```
myvar=$( ls -1 )
```

Alternative syntax:

```
myvar=`ls -1`
```

Bash: Integer arithmetic

Assign:

```
let a=5+4
```

Increment:

```
let a++
```

Combine values, quotes are to allow spaces

```
let b="$a + 10"
```

Alternative

```
a=$((5+4))
```

Bash: If statements

```
if [ <some test> ]
then
    commands
elif [ <some test> ]
then
    commands
else
    commands
fi
```

Logic expressions

[expression] See "man test" for details

! EXPRESSION	The EXPRESSION is false.
-n STRING	The length of STRING is greater than zero.
-z STRING	The length of STRING is zero (ie it is empty).
STRING1 = STRING2	STRING1 is equal to STRING2
STRING1 != STRING2	STRING1 is not equal to STRING2
INTEGER1 -eq INTEGER2	INTEGER1 is numerically equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is numerically greater than INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is numerically less than INTEGER2
-d FILE	FILE exists and is a directory.
-e FILE	FILE exists.
-r FILE	FILE exists and the read permission is granted.
-s FILE	FILE exists and its size is greater than zero (ie. it is not empty).
-w FILE	FILE exists and the write permission is granted.
-x FILE	FILE exists and the execute permission is granted.

Combine expressions: and/or

```
if [ <some test> ] && [ <some other test> ]
then
    commands
elif [ <some test> ] || [ <some other test> ]
then
    commands
else
    commands
fi
```

Example

```
if [ "$1" = "verbose" ] || [ "$1" = "v" ]
then
   echo "Hello world"
else
   echo "Hi"
fi
```

Output:

```
./verbose.sh
Hi
./verbose.sh verbose
Hello world
./verbose.sh v
Hello world
./verbose.sh vv
Hi
```

Bash: while/until loops

Repeat a task:

```
while [ <some test> ]
do
    commands
done

until [<some test> ]
do
     commands
done
```

While: Example

```
let a=0
while [ $a -lt 10 ]
do
      echo $a
      let a++
done
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Until: Example

```
let a=0
until [ $a -gt 9 ]
do
      echo $a
      let a++
done
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Bash: for

```
for var in <list>
do
    commands
done
```

`list` can come from program output using

`$(  )`

e.g:

`$( seq 1 10 )`

```
p123456@login1:~ seq 1 10
1
2
3
4
5
6
7
8
9
10
```

for: example

```
for file in $( ls -1 *.jpg )
do
     echo $file
done
```

Files:

Paris.jpg

San Francisco.jpg

Squirrel.jpg

Output:

```
Paris.jpg
San
Francisco.jpg
Squirrel.jpg
```

for: example (improved)

```
for file in *.jpg
do
     echo "$file"
done
```

Files:

Paris.jpg

San Francisco.jpg

Squirrel.jpg

Output:

```
Paris.jpg
San Francisco.jpg
Squirrel.jpg
```

Arrays

Define array:

```
ARRAY=(one two three)
echo ${ARRAY[*]}
one two three
```

Refer to specific element, count starts at 0:

```
echo ${ARRAY[2]}
three
```

The Linux toolbox (see man pages)

Command	Description	Example
head	show first lines of a file	head -10 file.txt
tail	show last lines of a file	tail -10 file.txt
cat	dump file contents to terminal	cat file.txt
grep	search for text in a file	grep Alice file.txt
sort	sort lines of text	sort file.txt
uniq	show only unique lines	uniq file.txt
wc	word and line count	wc -l file.txt
dirname	get directory part from file name and path	dirname /home/user/file.txt
basename	get only file name from file name and path	basename /home/user/file.txt

Redirecting input and output

- Commands also produce OUTPUT and ERROR
- These can be redirected to files or other commands
- Send output to file with ”>”:

```
echo ”Hello” > files.txt
```

- Append output to file with “>>”

```
echo ” world!” >> files.txt
```

- Read input from file with ”<”

```
myprog < inputfile.txt
```

- Concatenate commands:

```
cat files.txt | head -5 | sort
```

WARNING!

If the complexity of your bash script increases beyond what has been explained, really consider moving to Python!

Bonus: Configuration files

`.bash_profile`

- Run at login once
- Central environment settings
- Login messages
- Normally loads .bashrc

`.bashrc`

- Run for each bash shell
- Settings that are not inherited from .bash_profile

Job arrays

- Run the same kind of job many times
  - on different data
  - with different parameters
  - with different code
- Same resource requirements for each task
- A variable can be used to distinguish between tasks

```
#SBATCH --time=10:00
#SBATCH --mem=2GB
#SBATCH --job-name=run1
#SBATCH --cpus-per-task=1



python script.py data1
```

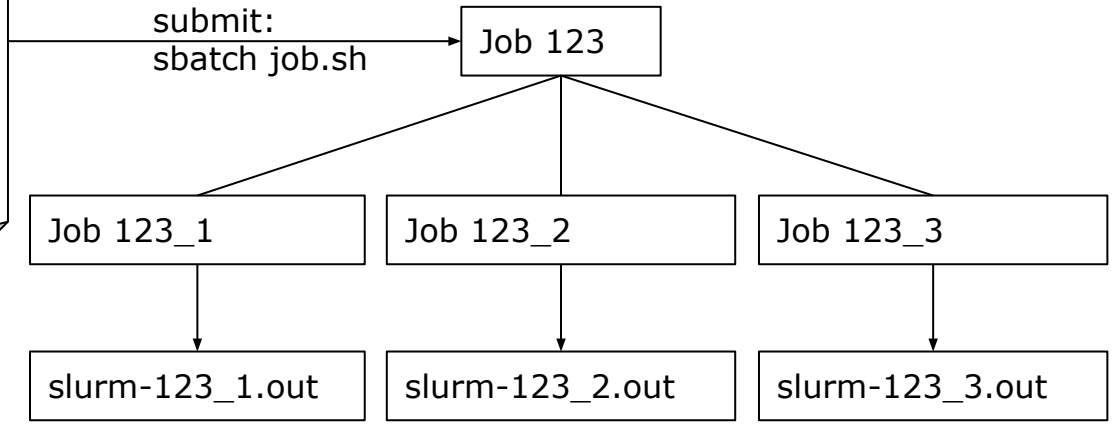
```
#SBATCH --time=10:00
#SBATCH --mem=2GB
#SBATCH --job-name=run2
#SBATCH --cpus-per-task=1



python script.py data2
```

...

```
#SBATCH --time=10:00
#SBATCH --mem=2GB
#SBATCH --job-name=run10
#SBATCH --cpus-per-task=1



python script.py data10
```

Job arrays: overview

```
#SBATCH --array=1-3


python script.py $SLURM_ARRAY_TASK_ID

or

python script_$SLURM_ARRAY_TASK_ID.py
```



Job arrays: ranges

```
--array=min-max[:step]
```

Examples:

```
--array=1-100
```
-> 1, 2, 3, …, 100

```
--array=1-10:2
```
-> 1, 3, 5, 7, 9

Limit number of simultaneously running tasks:

```
--array=1-100%5
```

`$SLURM_ARRAY_TASK_ID` iterates over the range

Max range size: 1001

Job arrays: squeue and scancel

- Get the status of the array:

```
[p123456@login1 array]$ squeue -u p123456
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     1464332_[1-3] regularsh  testjob  p123456 PD       0:00      1 (Resources)
```

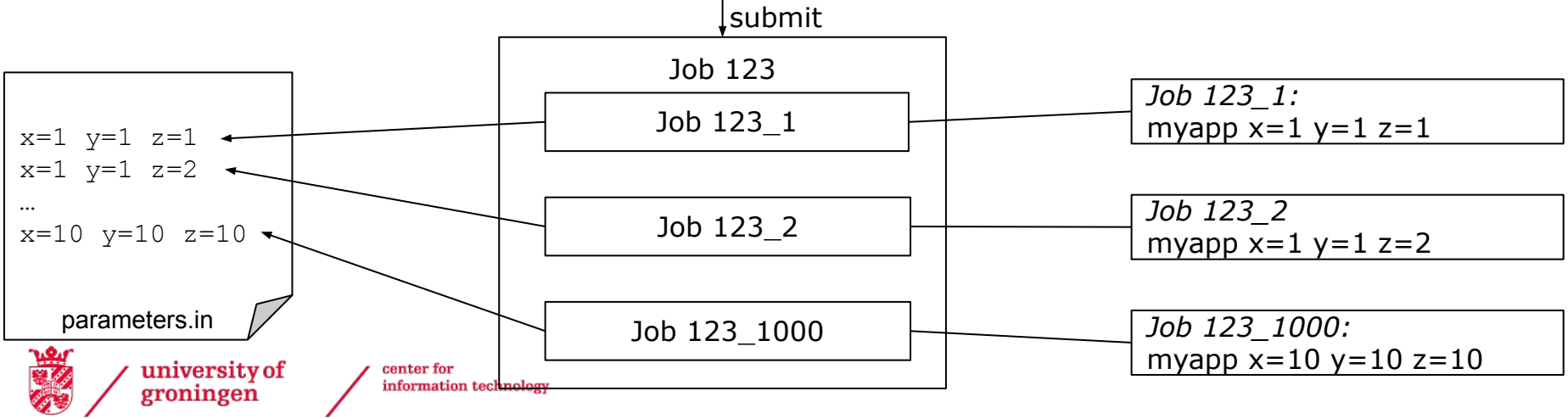
- One task per line:

```
[p123456@login1 array]$ squeue -u p123456 -r
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
         1464332_1 regularsh  testjob  p123456 PD       0:00      1 (Resources)
         1464332_2 regularsh  testjob  p123456 PD       0:00      1 (Resources)
         1464332_3 regularsh  testjob  p123456 PD       0:00      1 (Resources)
```

- Cancel one task of the array: `scancel 1464332_1`
- Cancel the entire array: `scancel 1464332`

Job arrays with parameter file

```
#SBATCH --array=1-1000

INPUTFILE=parameters.in

# get n-th line from $INPUTFILE
ARGS=$(head -n $SLURM_ARRAY_TASK_ID $INPUTFILE | tail -n 1)

myapp $ARGS
```

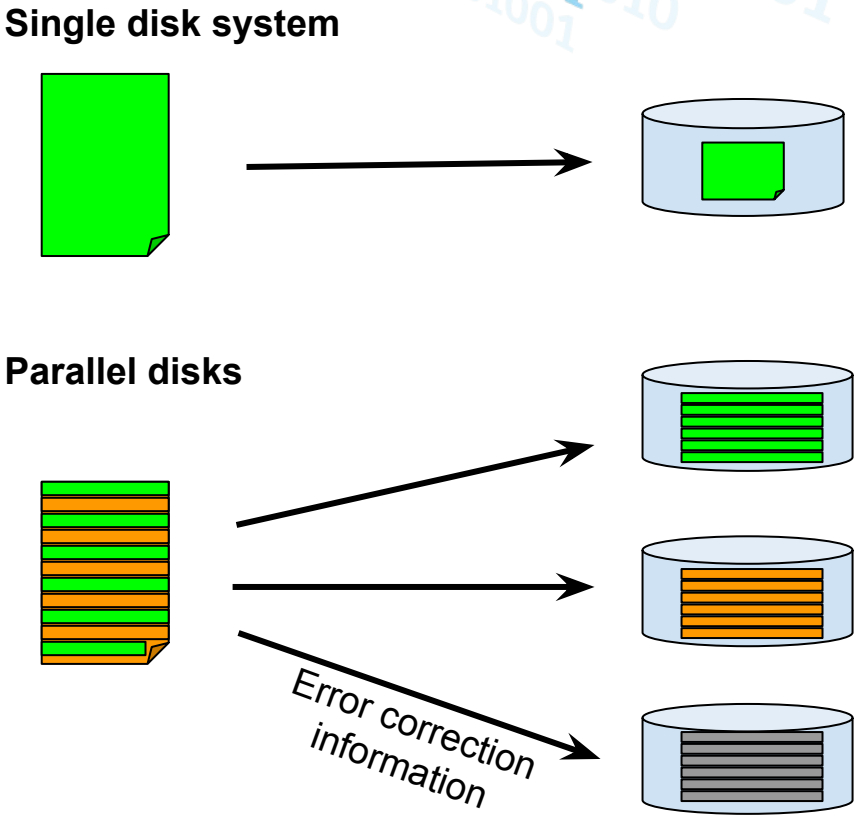


Hábrók file systems

- Hábrók uses a parallel shared file system
- This has advantages and disadvantages
- Some jobs will run well, others won't

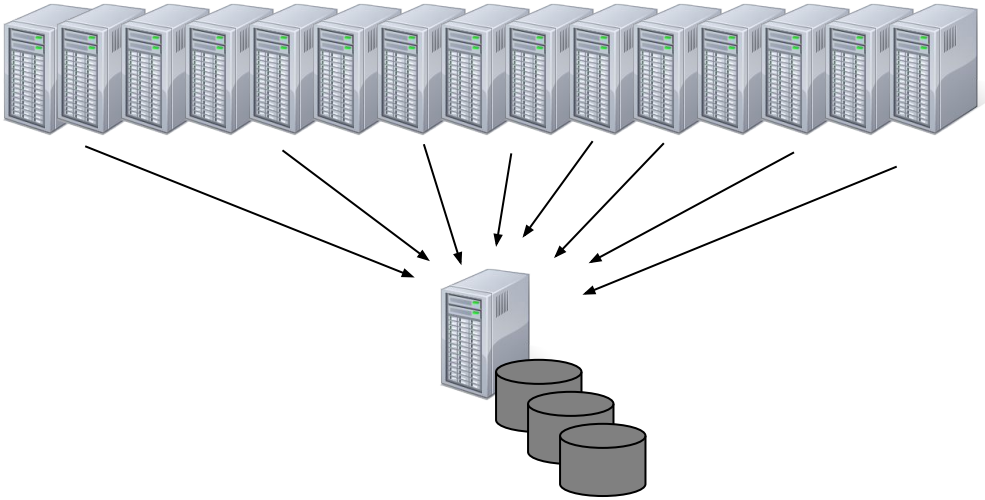
Parallel file systems

- Parallel storage, e.g. RAID system
- Files cut in blocks and distributed over available disks
- Includes error correction blocks
- Works best for large files



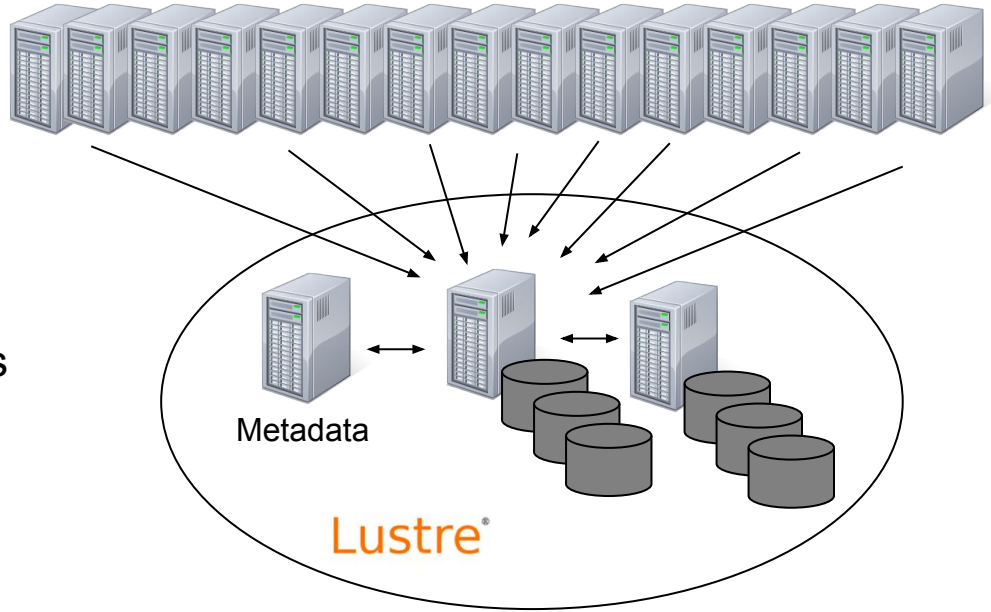
Shared file system

- All computers/nodes access the same storage through a server
- Every node sees the same files
- Synchronization costly!!
  - Changes must be visible everywhere
  - File locking



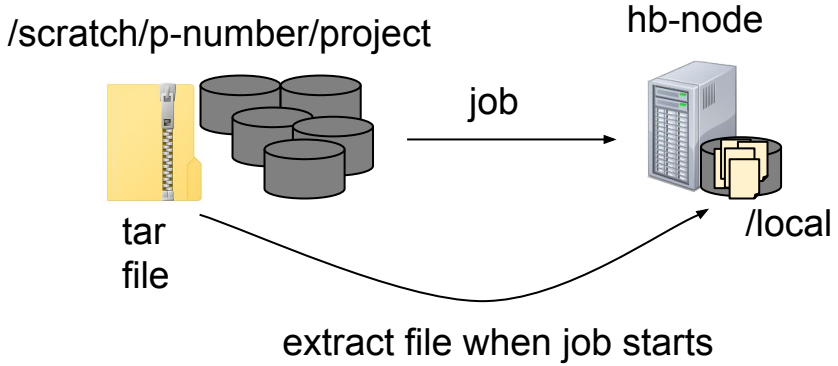
Hábrók Lustre file system

- Parallel file system
  - Multiple disk
  - Multiple storage servers
- Shared file system
  - Metadata centrally stored
- Works very well for > GB files 😀
- Works fine for > MB files 🙂
  - Parallelism through many clients
- Works poorly for kB files 😢
  - Mainly metadata access



When to use which file system?

- Large files (> GB) and medium size files (> MB)
  - Use /scratch
- Many small files (< MB, > 1000)
  - Use /local
    - Single disk, not bothered by metadata overhead
    - Temporary data only
    - Unpack files during job?
    - Solid state (NVMe) drives
      - Very fast
- Long-term storage only
  - Use /projects



Exercises

- Slides and exercises:
  - https://wiki.hpc.rug.nl
  - Hábrók
  - Additional information -> Course material
  - Advanced Hábrók Course

- Use your own Hábrók account
  - If you don't have an account yet, go to https://wiki.hpc.rug.nl, Introduction -> Accounts -> Accounts, access and Policies
    - Follow the instructions to request an account

university of groningen / center for information technology

CIT Academy

Questions?

https://wiki.hpc.rug.nl/habrok

university of groningen / center for information technology

part II

High Performance Computing

Large scale computations and data analysis on the Hábrók cluster (Advanced)

 Fokke Dijkstra

 Cristian Marocico

 Bob Dröge

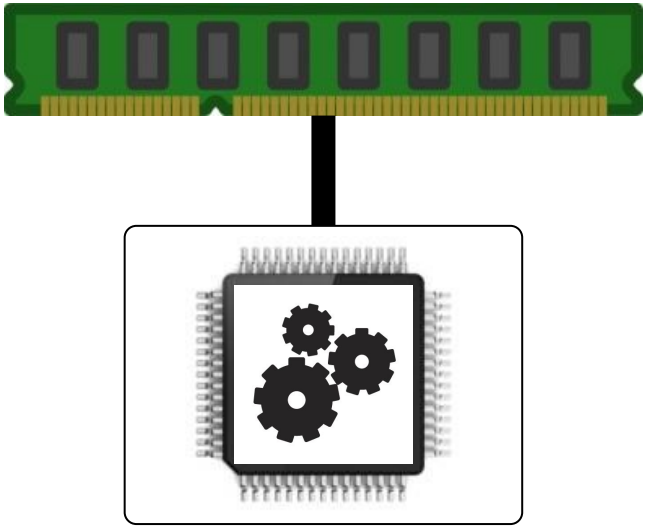
 Pedro Santos Neves

Part II

- Different methods for parallelizing a single run
  - Multiple cores
  - Multiple nodes
- Useful libraries
- Job scripts for parallel applications
- Accelerators / GPUs
- Best practices

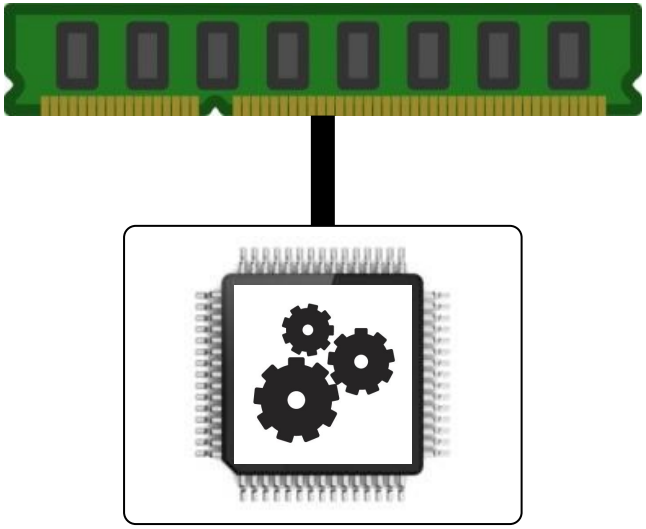
Single-core

- Single-core
- Vector parallelization

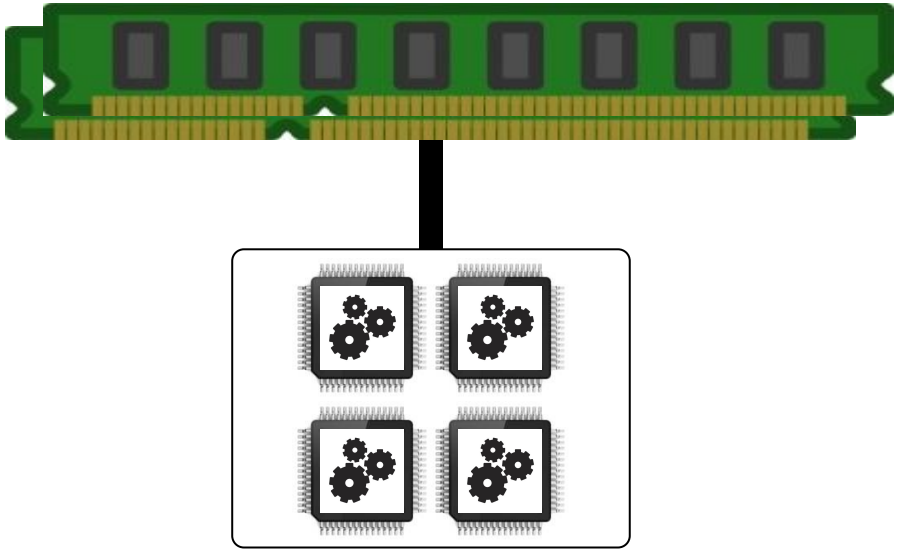


Single-core

Single to multi-core



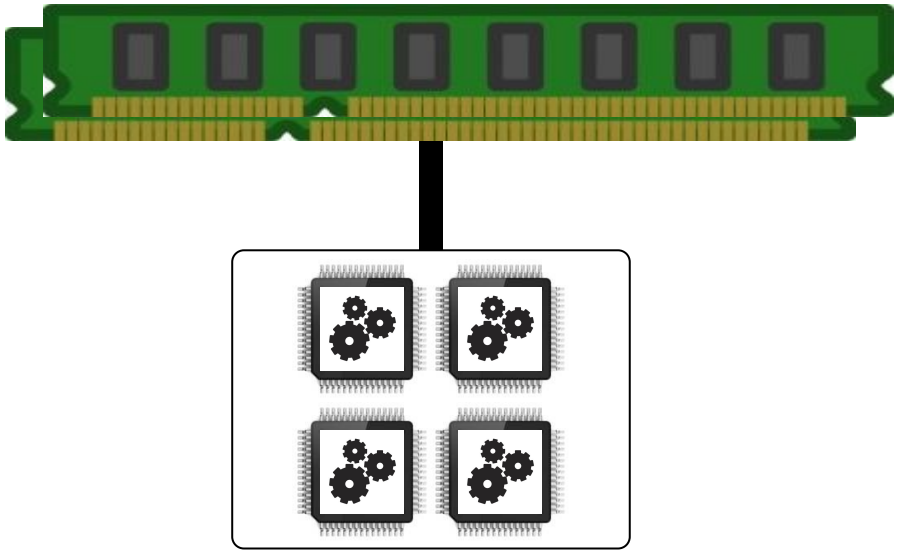
Past



Present

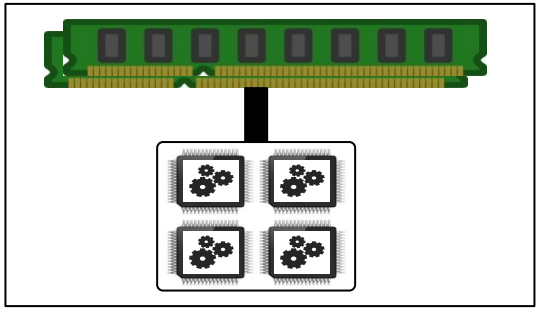
Multi-core

- Multi-core
- Shared memory
- C/C++, Fortran: OpenMP
- Python: multiprocessing
- R: parallel

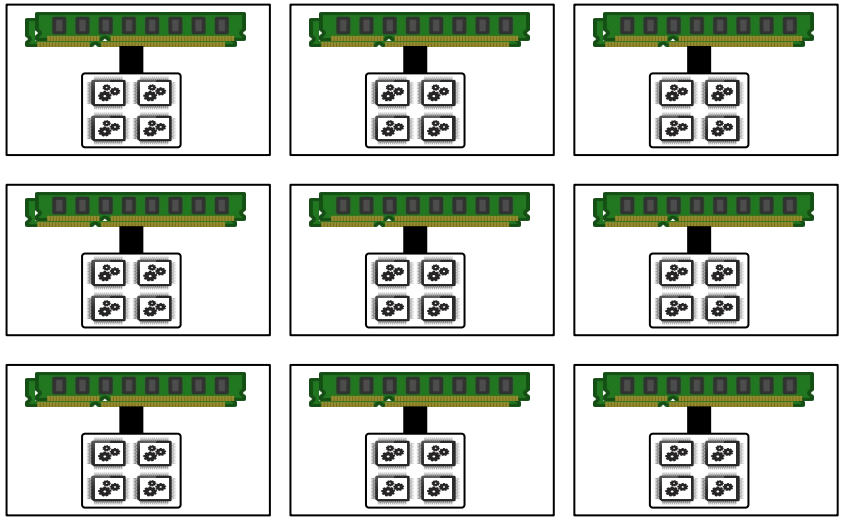


Multi-core

Single to multiple machines (nodes)



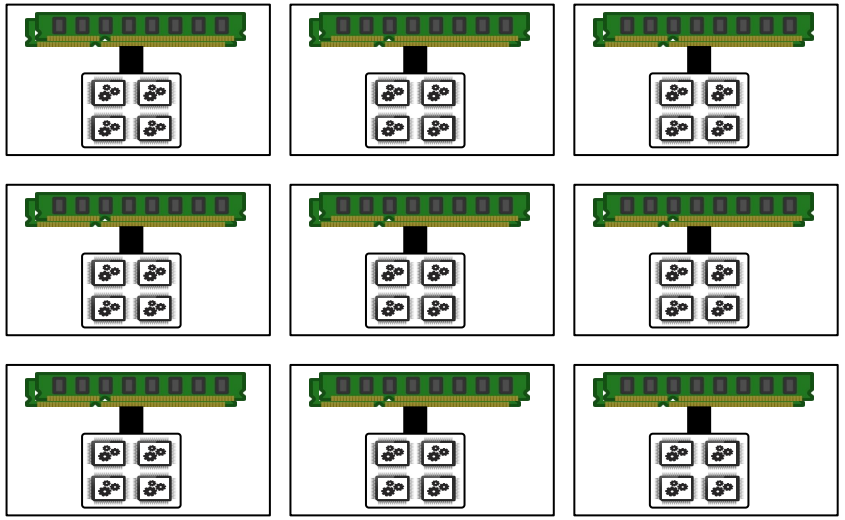
Single machine



Multiple machines

Multiple machines

- Multiple machines
- Distributed memory
- MPI, mpi4py, Rmpi, etc.

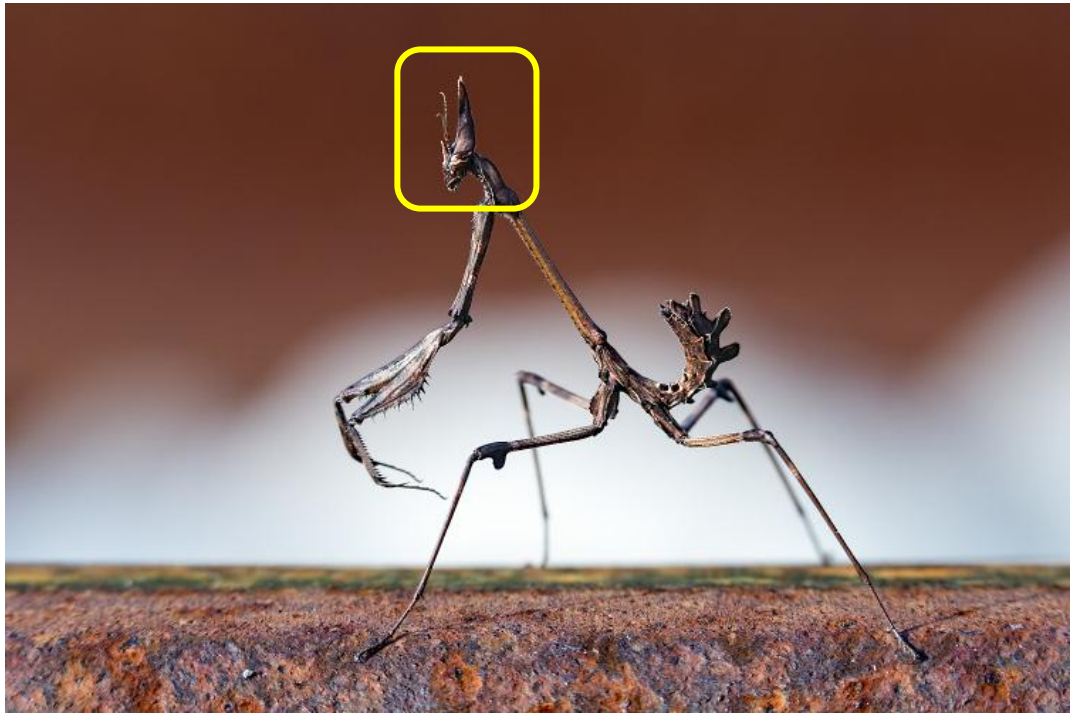


Multiple machines

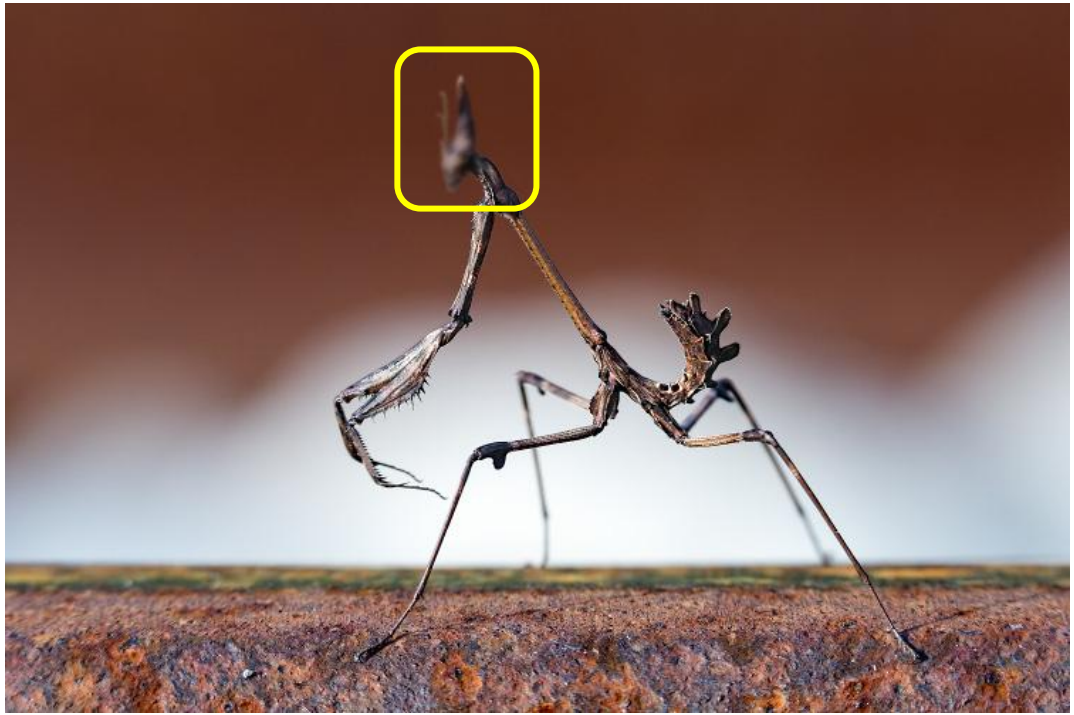
Parallelization: General

- Some applications aren't parallelized…
- Don't run them on multiple cores/nodes!
  - You will be billed for nothing…
- Shared vs. Distributed Memory
- OpenMP, MPI, Hybrids

Problem



Problem



Solution

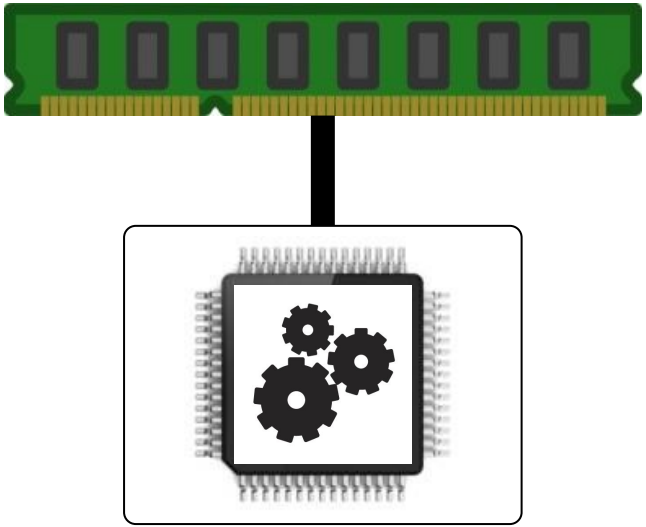
31	24	157	124	0
4	78	65	128	6
9	2	4	5	1
84	241	98	19	116
218	19	81	6	162

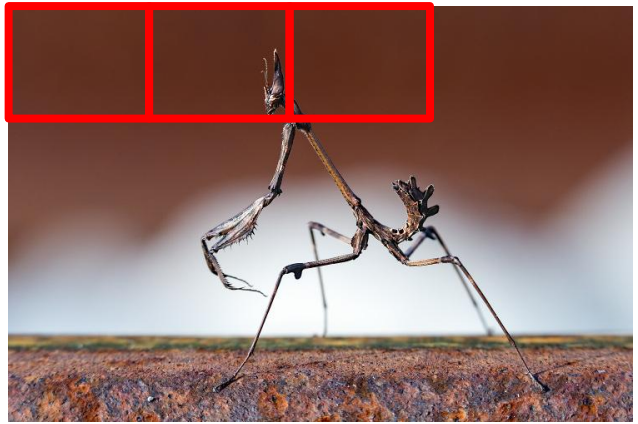
1	2	1
2	4	2
1	2	1

27	58	107	96	32
20	44	71	67	26
40	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .

$$\frac{1*31 + 2*24 + 1*157 + 2*4 + 4*78 + 2*65 + 1*9 + 2*2 + 1*4}{1 + 2 + 1 + 2 + 4 + 2 + 1 + 2 + 1} \sim 44$$

Single-core

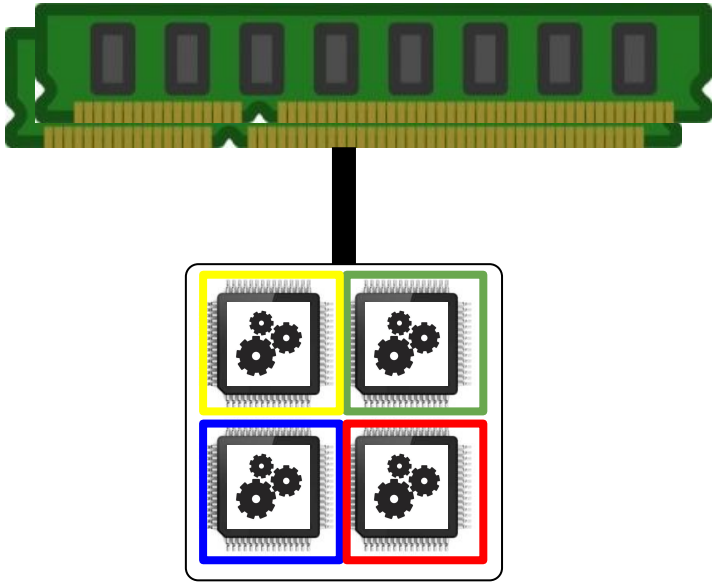




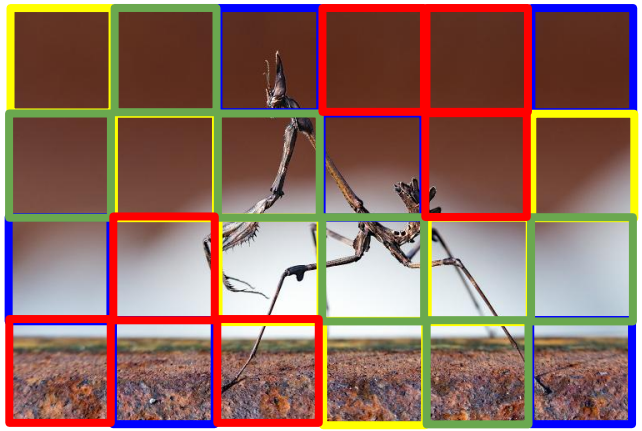
Single-core

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00
#SBATCH --partition=regular

./blur.me mantiss.jpg
```

Multi-core: OpenMP

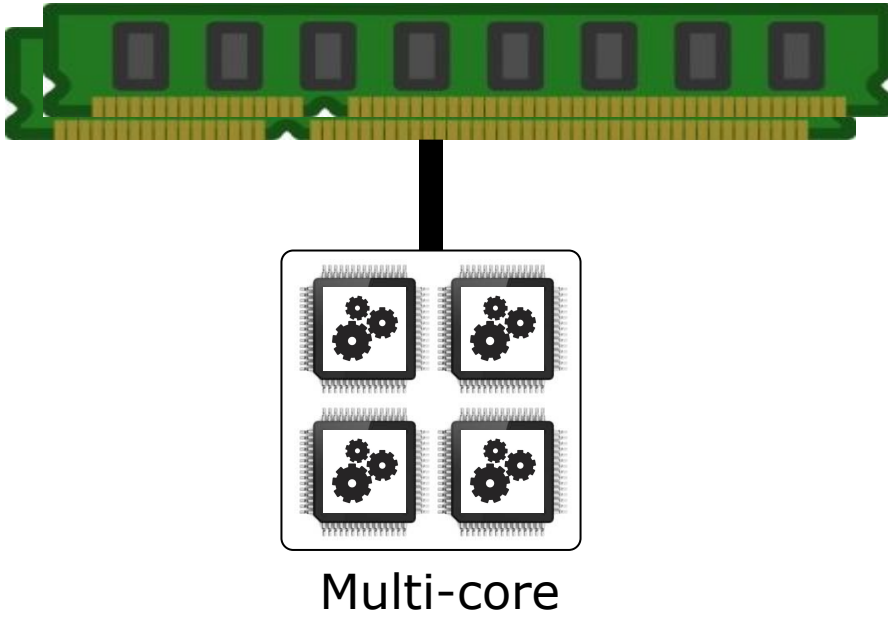


Multi-core

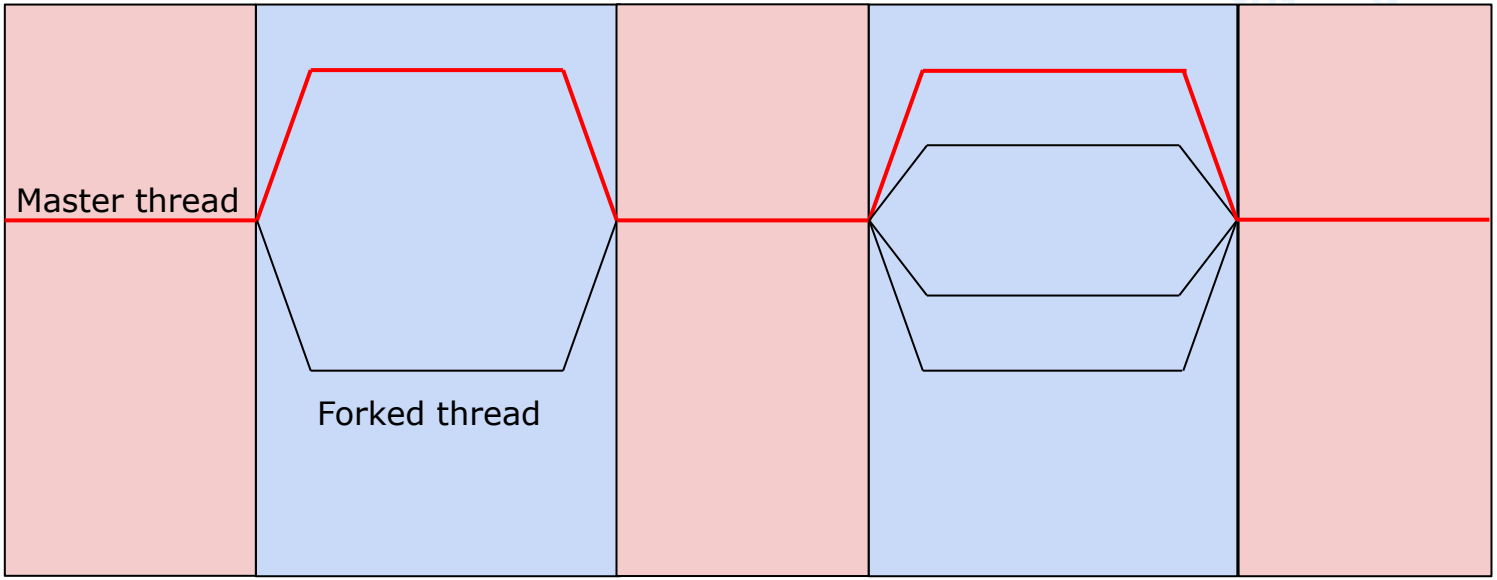


Multi-core: OpenMP

- Open Multi-Processing
- Shared memory API:
  - All cores access same memory
- Compiler directives: #pragma
- Race conditions
- Synchronization: barrier, critical, master, etc.
- Work sharing, etc.



OpenMP: Threads



Sequential example

```
#include <iostream>


int main(void) {


        int ID = 0;
        cout << “Hello from thread “ << ID << endl;

    return 0;
}
```

Parallel example

```
#include <iostream>
#include <omp.h>

int main(void) {
        #pragma omp parallel
        {
                int ID = omp_get_thread_num();
                cout << “Hello from thread “ << ID << endl;
        }
        return 0;
}
```

Threads again

- Threads communicate by sharing variables
- This can lead to race conditions
- Synchronization can protect against data conflicts
  - critical, atomic, barrier, etc.
- It is expensive, better to manage data access

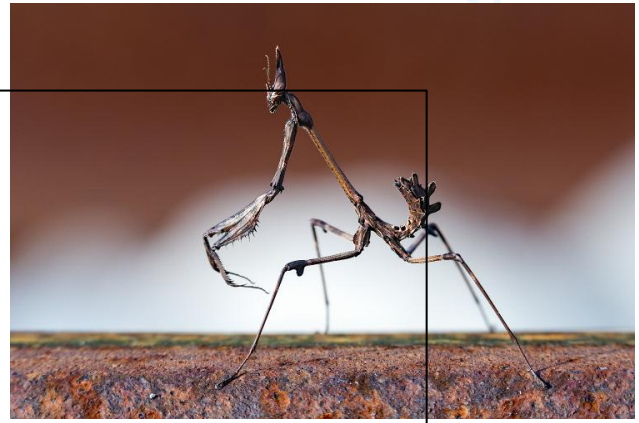
Work Sharing: Loops

```
#include <omp.h>

int main(void) {
        double a[MAX], b[MAX], c[MAX];
        #pragma omp parallel for
        for(i = 0; i < MAX; i++) {
                a[i] = b[i] * c[i];
        }
        return 0;
}
```

Work Sharing: Reduction

```
#include <omp.h>

int main(void) {
        double sum, b[MAX], c[MAX];
        #pragma omp parallel for reduction (+:sum)
        for(i = 0; i < MAX; i++) {
                sum += sqrt(b[i] * c[i]);
        }
        double mean = sum / MAX;
        return 0;
}
```

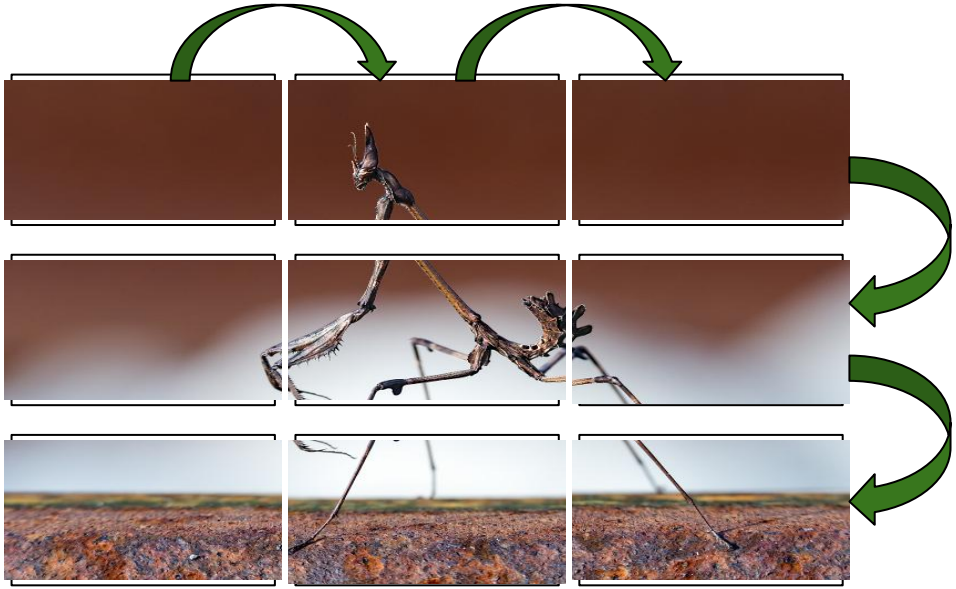
OpenMP: Jobscript

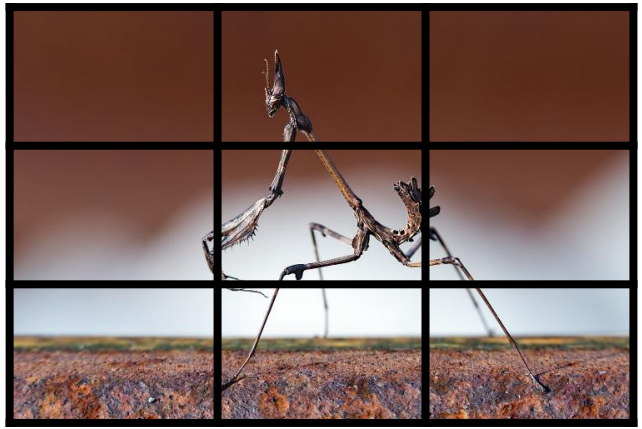


```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=24
#SBATCH --time=00:10:00
#SBATCH --partition=regular

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./blur.me mantiss.jpg
```

- Compile with `-fopenmp` flag (C/C++)

Multiple nodes: MPI





MPI: An extremely short primer

- MPI: Message Passing Interface
- Provides an API and library to manage Message Passing in a distributed memory environment
- `MPI_Init()`, `MPI_Finalize()`
- `MPI_Comm_Size()`, `MPI_Comm_Rank()`
- `MPI_Send()`, `MPI_Recv()`, `MPI_Bcast()`
- MPI Data Types: `MPI_DOUBLE`, `MPI_CHAR`
- And many more ...

MPI: Jobscript

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=24
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00
#SBATCH --partition=regular

srun ./blur.me mantiss.jpg
```

Hybrid OpenMP + MPI: Jobscript

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=24
#SBATCH --time=00:10:00
#SBATCH --partition=regular

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun ./blur.me mantiss.jpg
```

Alternative parallelization options

Python: multiprocessing, mpi4py

R: parallel, Rmpi, etc

Matlab: parfor, built-in multithreading (some functions)

C++: std::thread (since C++11), Boost.thread, Intel TBB, pthreads, etc

Slurm parallel job parameters

`--cpus-per-task`	Multi-core/multi-threading
`--nodes`	No. of nodes
`--ntasks`	Total no. of MPI processes/tasks
`--ntasks-per-node`	No. of tasks per node
`srun/mpirun`	Launch your (MPI) application

Parallelization: Best practices

- Do not use it if the application doesn't support it
- Check that the requested # of CPUs has been used
  - Many applications need to know how many cores they can use / threads they can start (e.g. with a command-line option, $OMP_NUM_THREADS)
- Check the CPU efficiency (job's output file or `jobinfo`)
  - Will also report efficiency hints

Output of jobinfo <jobid>

```
Name                : MyJob
User                : p12345
Partition           : regularshort
Nodes               : node9
Number of Nodes     : 1
Cores               : 4
Number of Tasks     : 1
State               : COMPLETED
Submit              : 2023-09-20T15:31:59
Start               : 2023-09-20T15:32:01
End                 : 2023-09-20T15:33:28
Reserved walltime   : 00:25:00
Used walltime       : 00:01:27
Used CPU time       : 00:01:28 (efficiency: 25.50%)
% User (Computation): 98.74%
% System (I/O)      :  1.26%
Mem reserved        : 8000M
Max Mem (Node/step) : 339.70M (node9, per node)
Full Max Mem usage  : 339.70M
Total Disk Read     : 4.84M
Total Disk Write    : 698.10K
```

$$\frac{\text{Used CPU time}}{\text{Cores x Used walltime}} \times 100\%$$

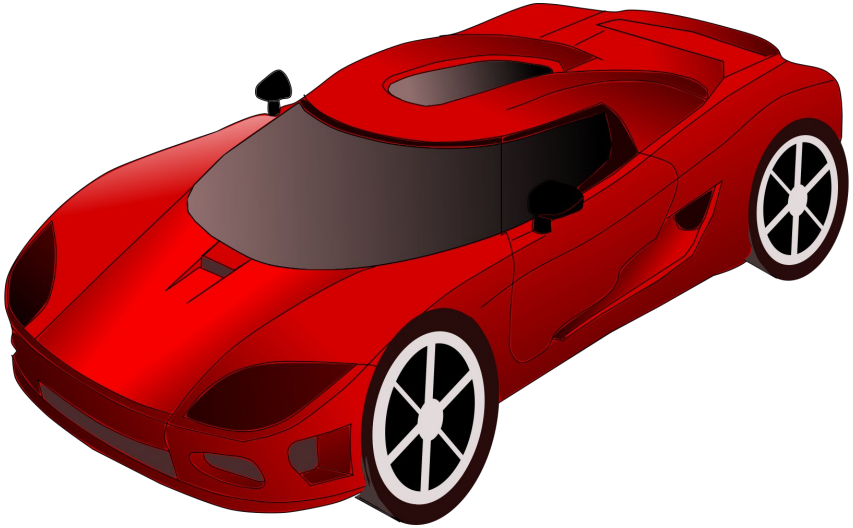
Parallelization: Best practices

- Do not use it if the application doesn't support it
- Check that the requested # of CPUs has been used
  - Many applications need to know how many cores they can use / threads they can start (e.g. with a command-line option, $OMP_NUM_THREADS)
- Check the CPU efficiency (job's output file or `jobinfo`)
  - Keep in mind the law of diminishing returns
  - Experiment until you get things to your liking

Accelerators

- Special add-ons which allow for faster compute
- Simpler cores, but usually many per chip
- Examples:
  - GPUs
  - FPGA
    - Programmable logic
    - Can be tuned for important operations

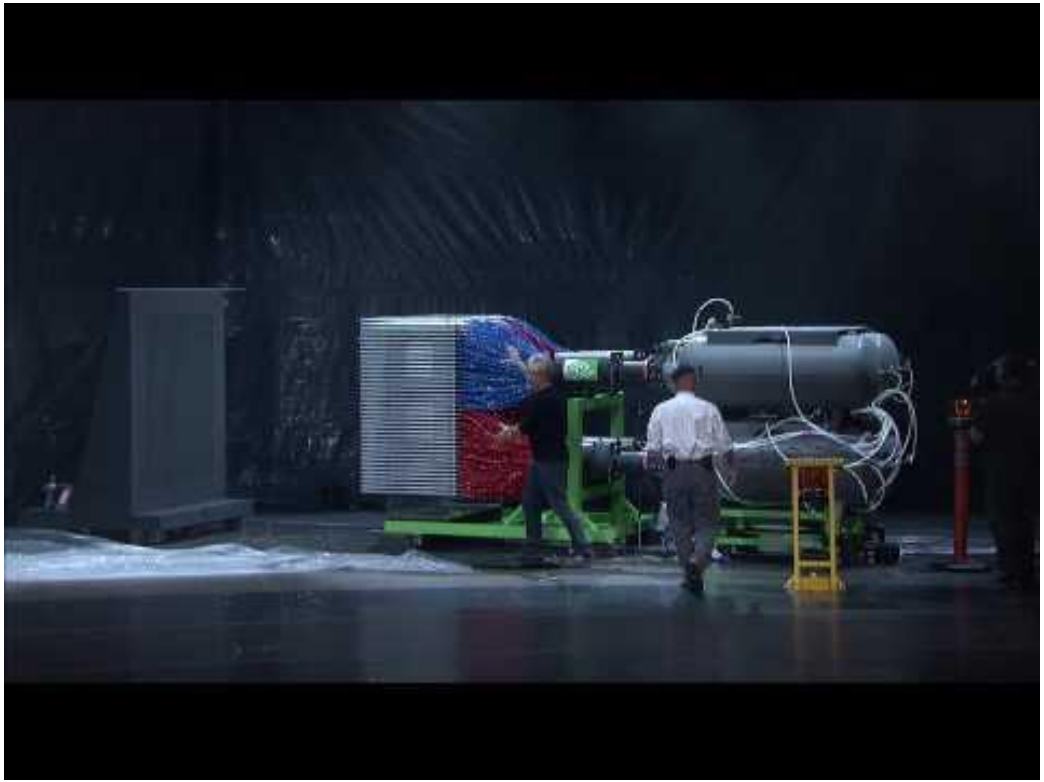
CPU vs GPU (or similar accelerators)



CPU

GPU

Live GPU Demo



GPU libraries / software

C, C++:

- CUDA, OpenCL
- OpenACC

Fortran:

- CUDA, OpenACC

Python:

- CuPy, Numba
- TensorFlow

R:

- gpuR

Java:

- JCuda

Other:

- MATLAB
- Mathematica
- many more...

OpenACC

- Pragma
- Focus on compute intensive parts
  - Loops are prime candidate
- Data management is important for performance
  - Transferring data between host and GPU is costly
  - Prevent unnecessary data movement
- Nvidia compilers support OpenACC
- Nvidia training available

Example in C code:

```
// sum component wise and save result into vector c
#pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
for(i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

Resources and limits GPU nodes

	a100	v100
GPUs per node	4	1/2
Memory per GPU	40 GB	32 GB
CPUs per node	64	8/24
Time	3 days	

Submitting GPU jobs

- Request one or more GPUs per node, specific type is optional:
- `#SBATCH --gpus-per-node=1`
- `#SBATCH --gpus-per-node=v100:1`
- `#SBATCH --gpus-per-node=a100:1` or `2`

Developing and testing GPU tasks/jobs

- Use the interactive GPU nodes: gpu1.hb.hpc.rug.nl & gpu2.hb.hpc.rug.nl
  - Each has one V100 GPU
  - Other than that, similar to the other login/interactive nodes
  - Don't run many tasks simultaneously and/or use a GPU for a very long time

- Interactive jobs on GPU nodes

```
srun --gpus-per-node=1 --time=01:00:00 --pty /bin/bash
```

- `exit`
  - **Interactive jobs currently don't (always) use the software stack built for the allocated nodes**

```
unset SW_STACK_ARCH && module restore
```

Using GPUs: best practices

- Similar to parallelization on the CPU:
  - Do not use it if the application doesn't support it
  - Check that the requested GPU(s) has/have been used
  - Check the GPU utilization (`jobinfo`)

```
Average GPU usage    : 71.0% (a100gpu4)
```

- Low utilizations are often caused by I/O overhead
  - Possible solutions discussed in first part of this course
- Jobs that don't use the GPU in the first couple of hours, are killed by the scheduler

university of groningen / center for information technology

CIT Academy

Questions?

https://wiki.hpc.rug.nl/habrok

Exercises

- Slides and exercises:
  - https://wiki.hpc.rug.nl
  - Hábrók
  - Additional information -> Course material
  - Advanced Hábrók Course

- Use your own Hábrók account
  - If you don't have an account yet, ask us
- Wrap-up at 17:00

Wrap-up

- Summary

- If you haven't finished the exercises, you work on them in your own time
  - The GPU node reservation for the last exercise will not be available
- Questions / feedback?
- Any unclear topics which need some more explanation?

- Feel free to contact us at:
  - hpc@rug.nl
  - Virtual walk-in session: https://wiki.hpc.rug.nl/habrok/introduction/support

university of groningen / center for information technology

CIT Academy

Thank you for attending this course!

https://wiki.hpc.rug.nl/habrok