university of groningen

center for information technology

CIT Academy

Advanced Peregrine Course

Introduction

Single run:

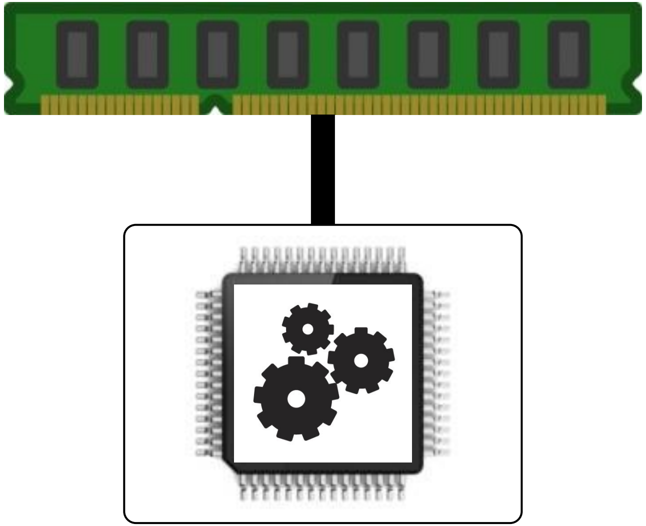
- single machine: 1 Core
- single machine: 1 Core+GPU
- single machine: n Cores - OpenMP
- multiple machines - MPI

Multiple runs:

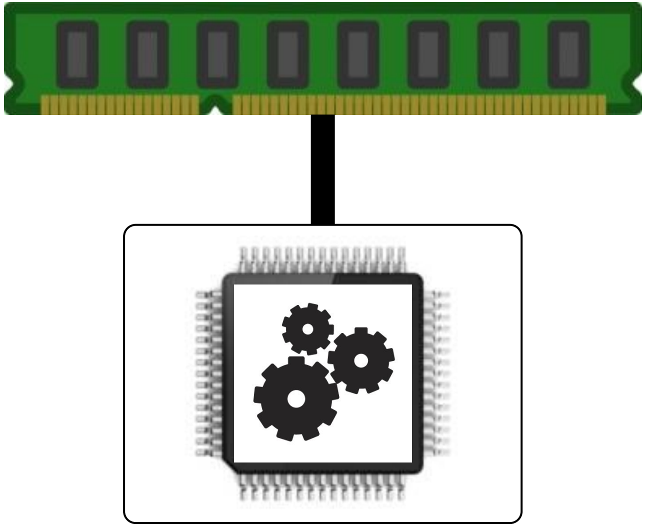
- job arrays, bash scripting, etc.

Single-core

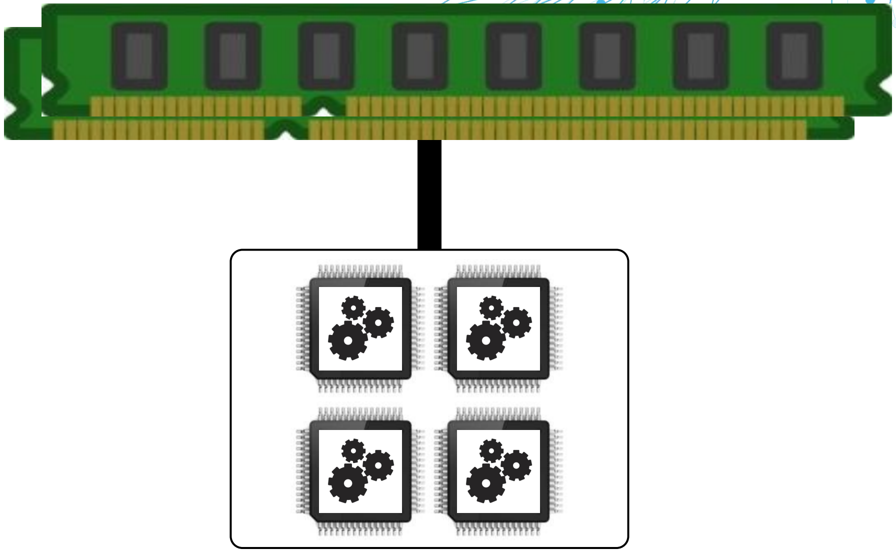
- Single-core
- Vector parallelization



Single to multi-core



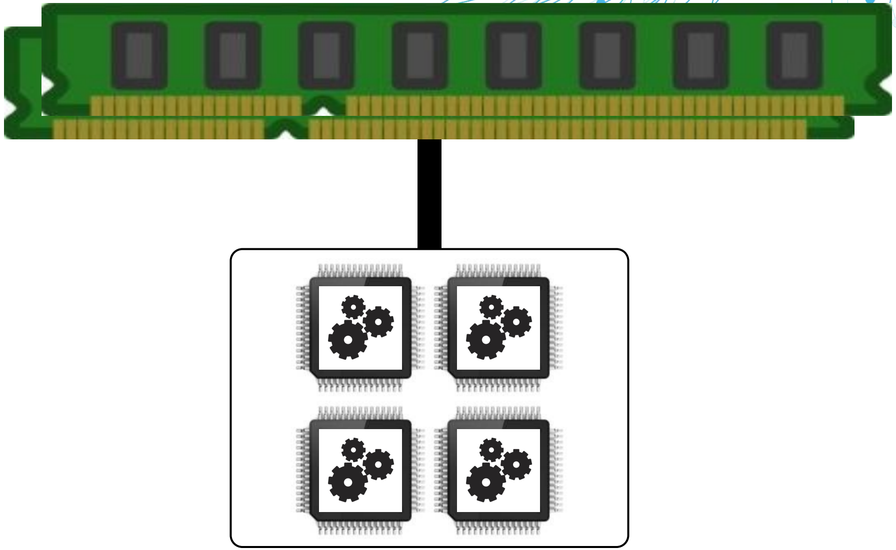
Past



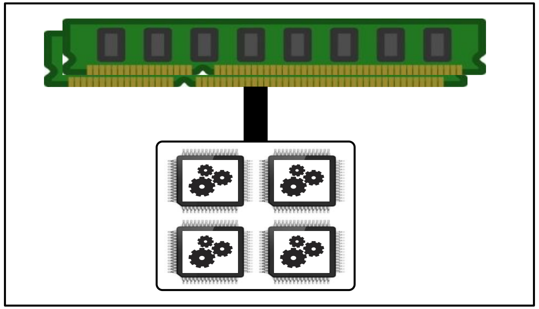
Present

Multi-core

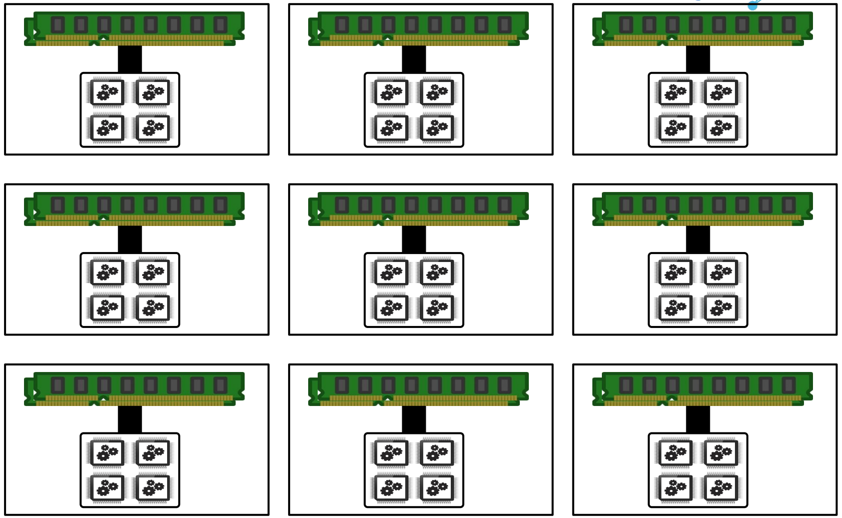
- Multi-core
- Shared memory
- C/C++, Fortran: OpenMP
- Python: multiprocessing
- R: parallel



Single to multiple machines



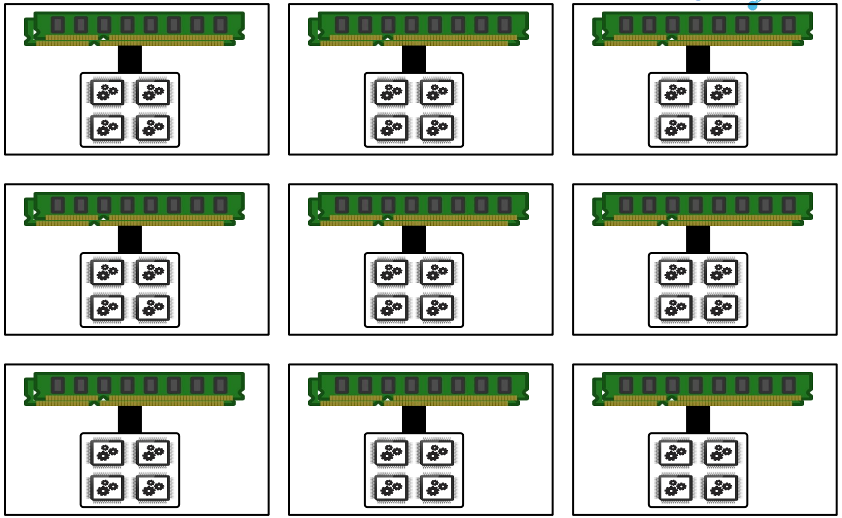
Single machine



Multiple machines

Multiple machines

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

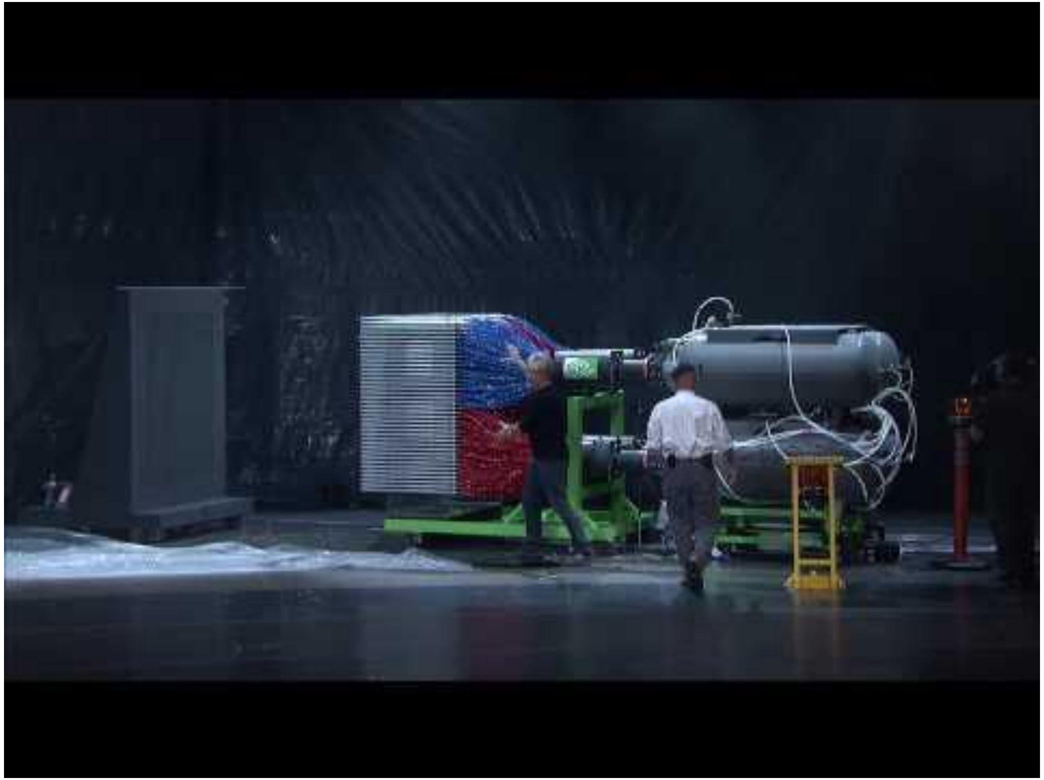


Multiple machines

Accelerators

- Special add-ons which allow for faster compute
- Simpler cores
- Examples:
  - GPUs
  - FPGA
    - Programmable logic
    - Can be tuned for important operations

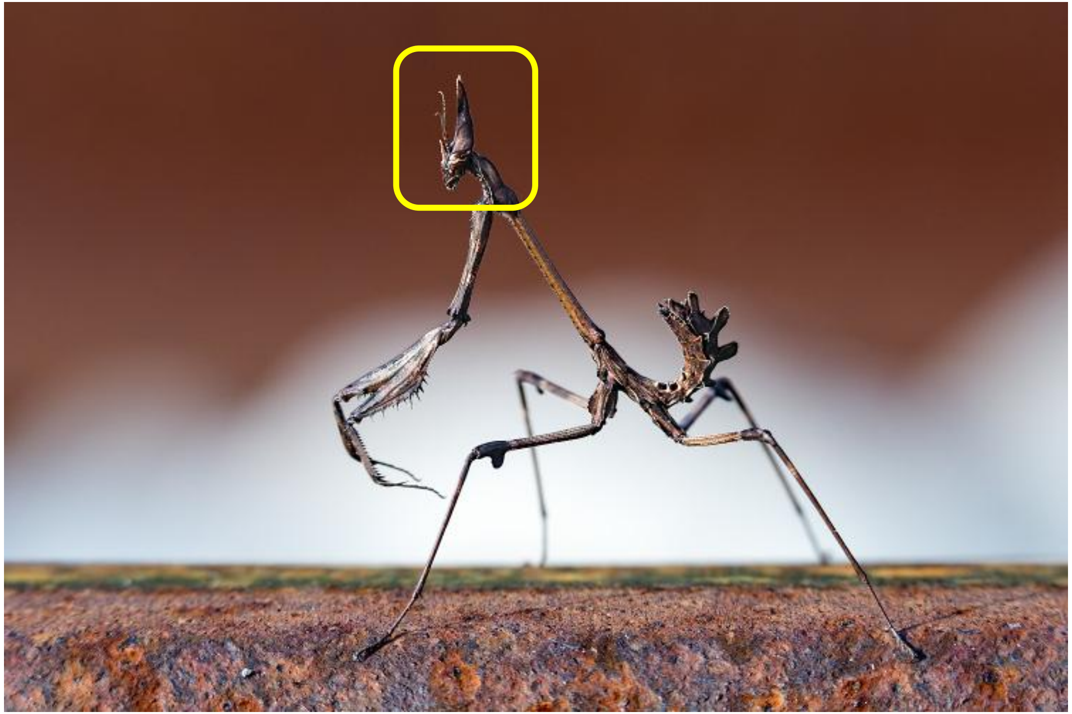
Live GPU Demo



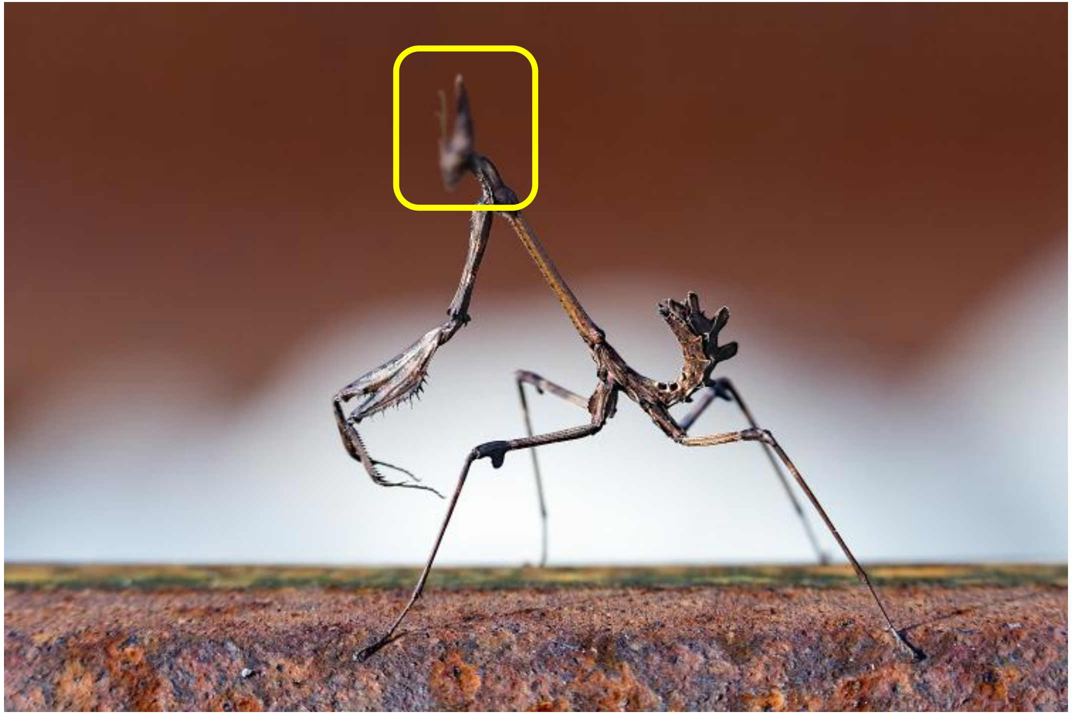
Parallelization: General

- 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

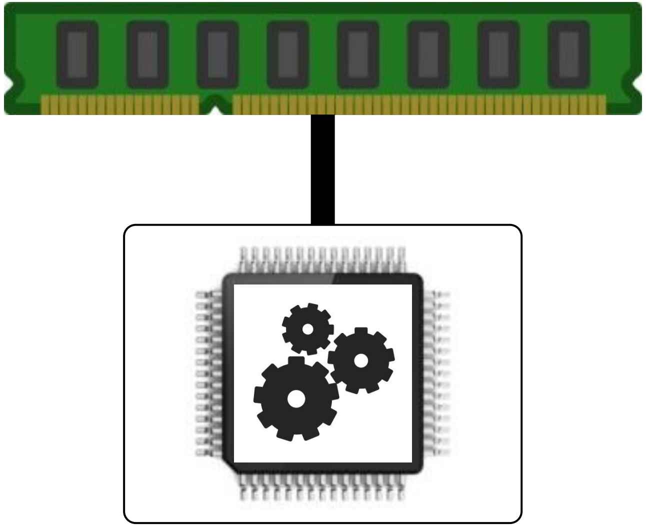
31	24	157	124	0
4	78	65	128	6
9	2	4	5	1
84	241	98	19	116
218	19	81	6	162

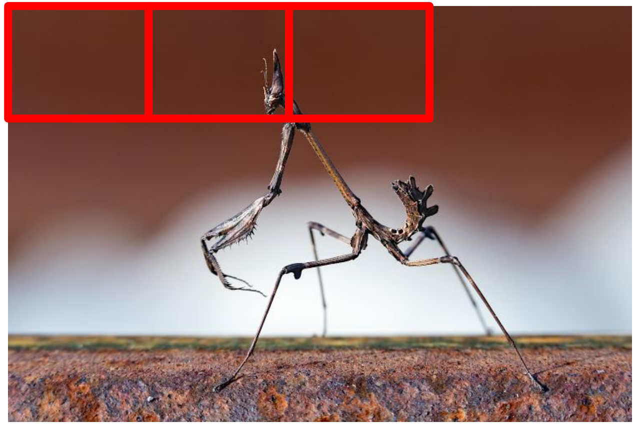
1	2	1
2	4	2
1	2	1

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



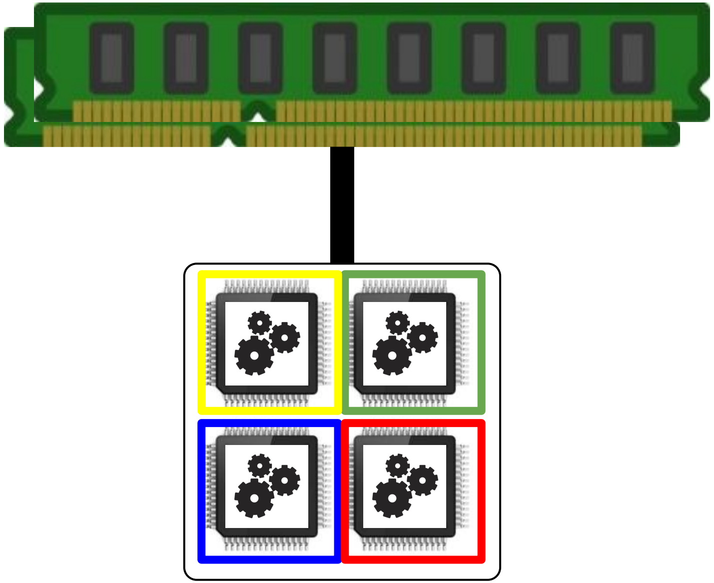


Single-core

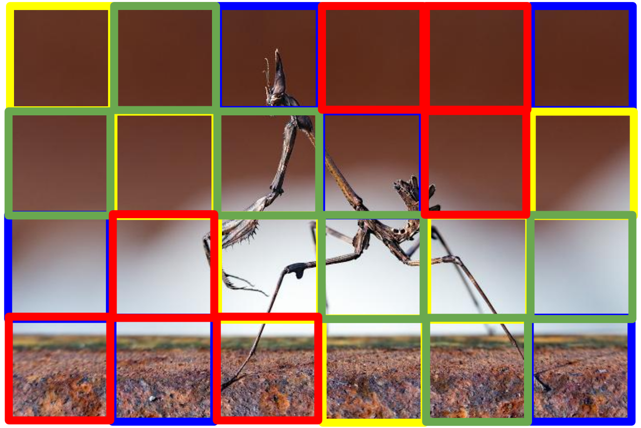
```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00
#SBATCH --partition=short

./blur.me mantiss.jpg
```

Multi-core: OpenMP

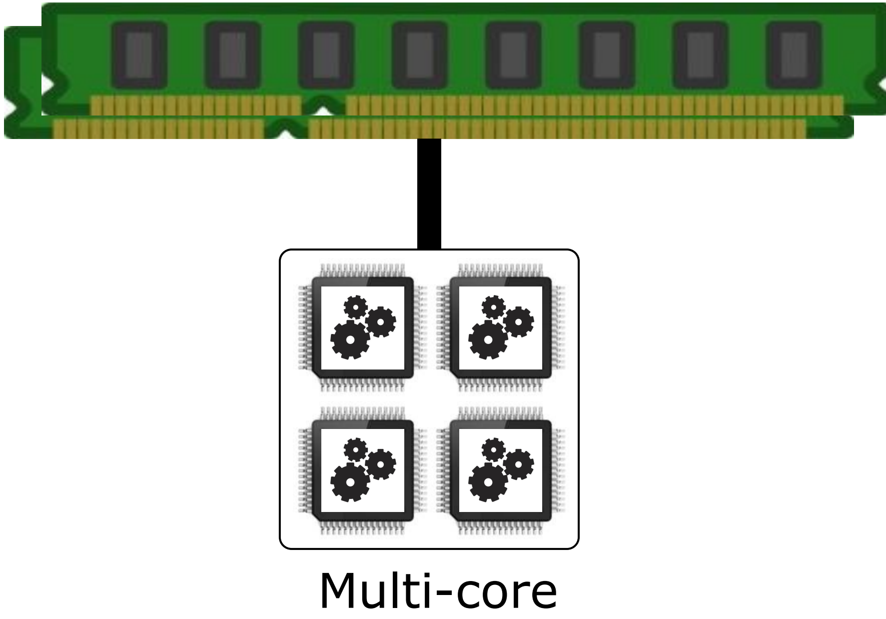


Multi-core

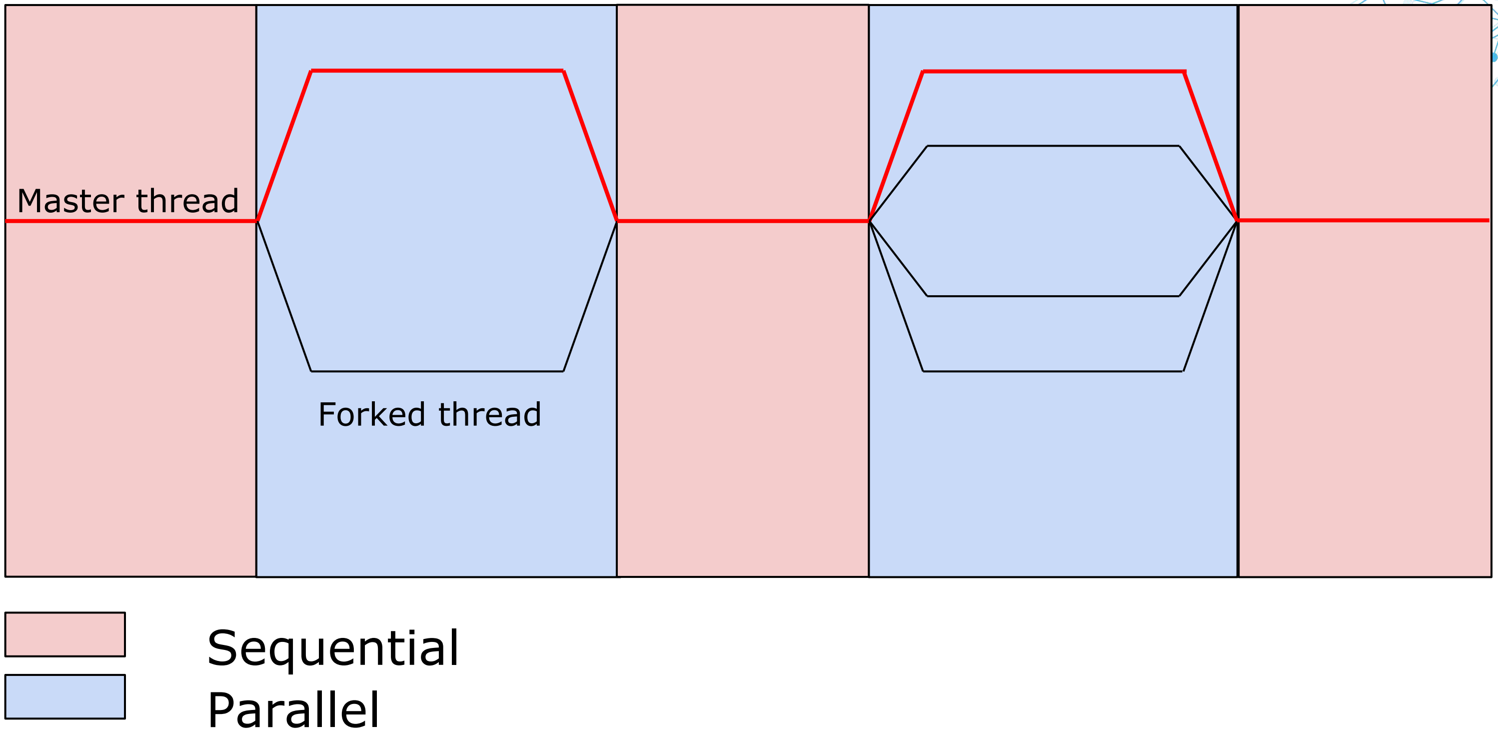


Multi-core: OpenMP

- 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];

	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=short

./blur.me mantiss.jpg
```

- Compile with -fopenmp flag (C/C++)
- OMP_NUM_THREADS, etc.

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 --ntasks=4
#SBATCH --cpus-per-task=6
#SBATCH --time=00:10:00
#SBATCH --partition=short

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

Alternative 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

MPI: Slurm parameters

--ntasks	No. of MPI processes
--cpus-per-task	Multi-core/multi-threading
--nodes	No. of nodes
--ntasks-per-node	No. of tasks per node
srun/mpirun	

Parallelization: Best practices

- Do not use it if the application doesn’t support it
- Check that the requested # of CPUs has been used
- Check the CPU efficiency (`jobinfo`)

Output of `jobinfo <jobid>`

```
Name                : MyJob
User                : p123456
Partition           : regular
Nodes               : pg-node001
Cores               : 4
State               : COMPLETED
Submit              : 2018-30-23T25:23:47
Start               : 2018-30-23T25:36:37
End                 : 2018-30-23T25:42:09
Reserved walltime   : 08:00:00
Used walltime       : 00:05:32
Used CPU time       : 00:19:30 (efficiency: 88.17%)
% User (Computation): 99.87%
% System (I/O)      :  0.13%
Mem reserved        : 8000M/node
Max Mem used        : 452.00M (pg-node001)
Max Disk Write      : 20.48K (pg-node001)
Max Disk Read       : 819.20K (pg-node001)
```

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

Parallelization: Best practices

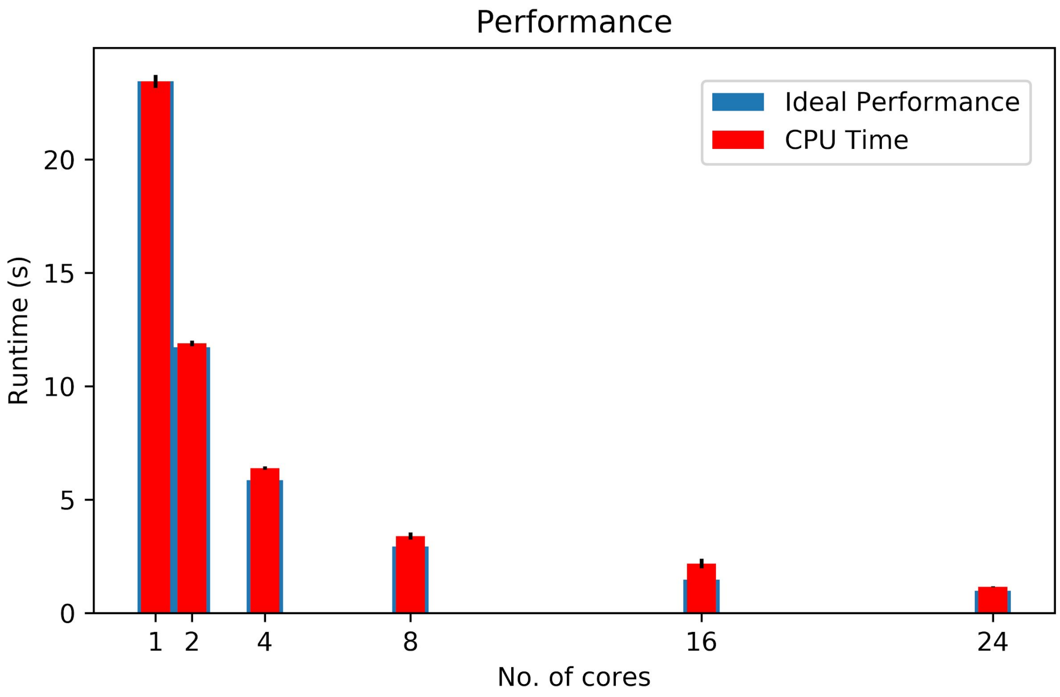
- Do not use it if the application doesn’t support it
- Check that the requested # of CPUs has been used
- Check the CPU efficiency (`jobinfo`)
  - Keep in mind the law of diminishing returns
  - Experiment until you get things to your liking

Exercises

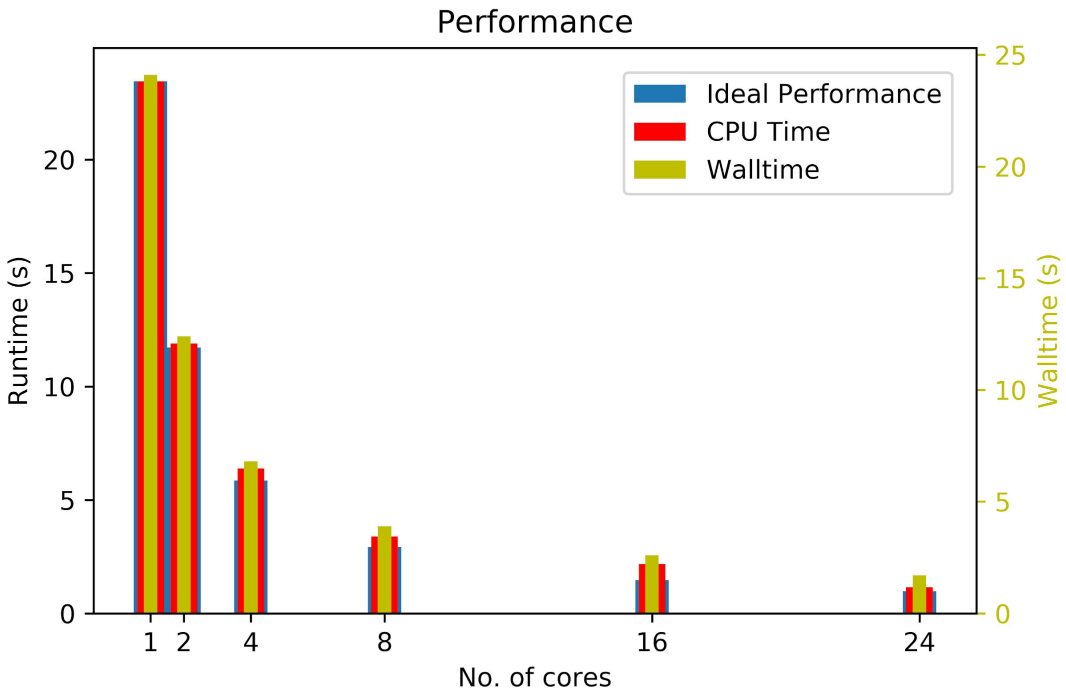
- Hostname: `peregrine.hpc.rug.nl`
- Username: `see handouts`
- Password: see handouts

- Slides, go to:
  - https://redmine.hpc.rug.nl
  - Peregrine
  - Wiki
  - Course material

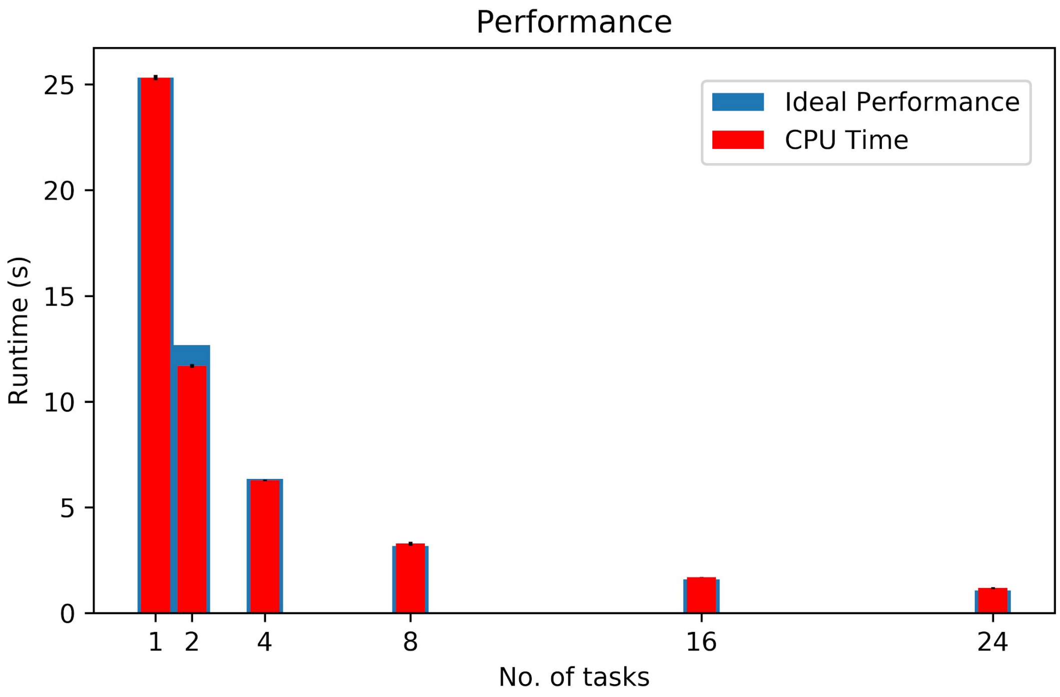
Exercises: Discussion - OpenMP



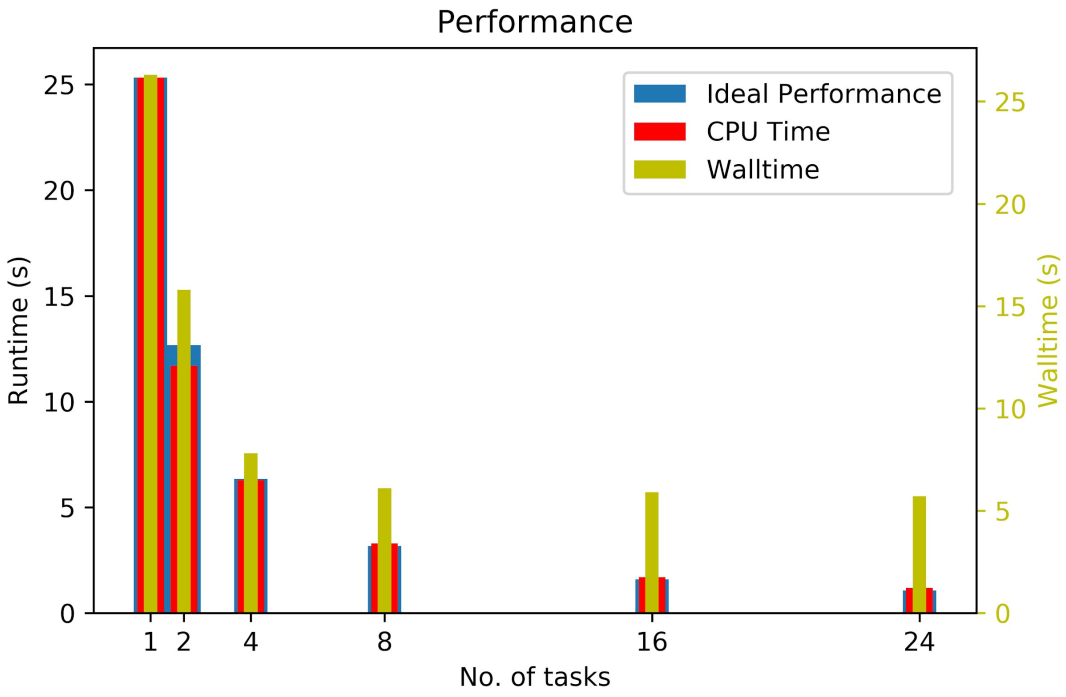
Exercises: Discussion - OpenMP



Exercises: Discussion - MPI



Exercises: Discussion - MPI



Part II

1. Bash scripting
2. Job arrays
3. Pilot jobs

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

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	✓ Easier to understand 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------ 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 by 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

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.

Example

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

beware of the spaces

Output:
./verbose.sh
Hi

./verbose.sh verbose
Hello world

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)

```
f111536@peregrine:~ 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
```

Arrays

Define array:

```
ARRAY=(een twee drie)
echo ${ARRAY[*]}
een twee drie
```

Refer to specific element, count starts at 0:

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

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 load .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 script1.py data1
```

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



python script2.py data2
```

...

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



python script10.py data10
```

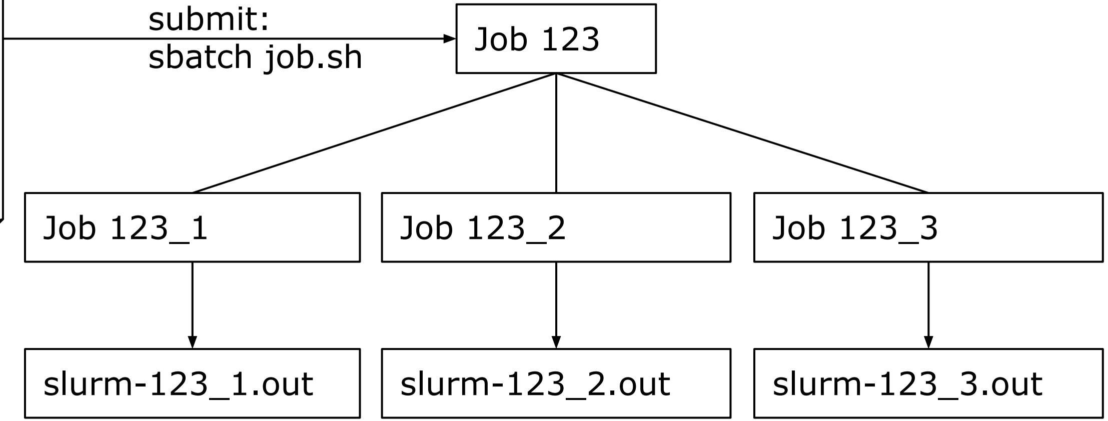
Job arrays: overview

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

- One task per line:

```
[p123456@peregrine array]$ squeue -u p123456 -r
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
         1464332_1     short  testjob  p123456 PD       0:00      1 (Resources)
         1464332_2     short  testjob  p123456 PD       0:00      1 (Resources)
         1464332_3     short  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 $INPUTFILE
ARGS=$(cat $INPUTFILE | head -n $SLURM_ARRAY_TASK_ID | tail -n 1)

myapp $ARGS
```

submit

Job 123

Job 123_1

Job 123_2

Job 123_1000

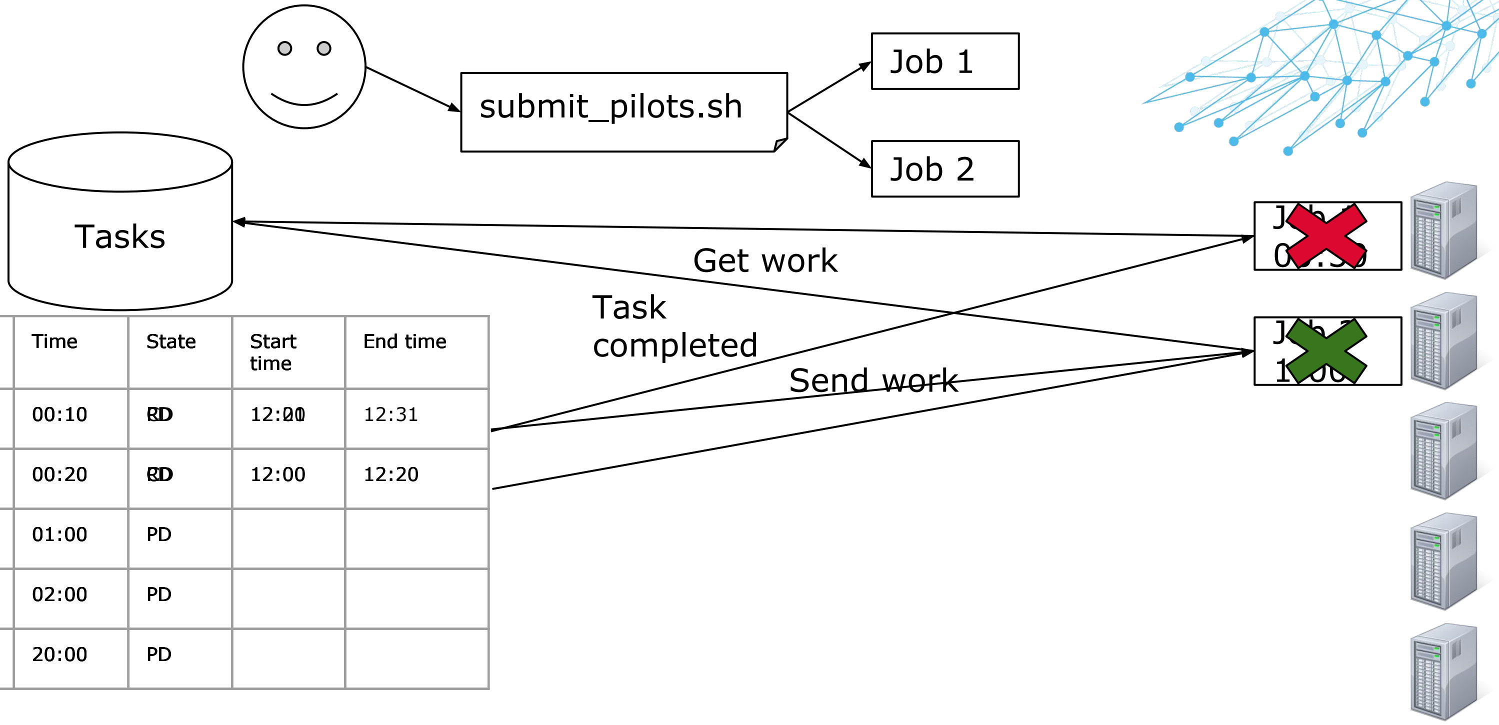
```
x=1  y=1  z=1
x=1  y=1  z=2
…
x=10  y=10  z=10
```

Job 123_1:
myapp x=1 y=1 z=1

Job 123_2
myapp x=1 y=1 z=2

Job 123_1000:
myapp x=10 y=10 z=10

Pilot jobs



ID	Parameters	Time	State	Start time	End time
1	n=10	00:10	CD	12:21	12:31
2	n=20	00:20	CD	12:00	12:20
3	n=50	01:00	PD		
4	n=100	02:00	PD		
5	n=1000	20:00	PD		

Pilot jobs

- Decouple workload from job script
- Job tasks:
  - Request resources
  - Set up the environment
  - While time and work available:
    - Request workload (e.g. from database)
    - Fetch data
    - Do the work
    - Store results
    - Confirm task completion
  - Exit
- Use heartbeat / timeout to cope with pilot issues

Pilot jobs: pros and cons

- ✓ Less error-prone for job initialization problems
- ✓ Failed tasks will automatically run again
- ✓ Clear administration of (non-)completed tasks
- ✓ Tasks can run (almost) anywhere

- ✗ Requires extra work to get started
- ✗ Finding optimal choices for the number of pilot jobs to submit and their resource specifications

Pilot jobs: more information

- Picas / ToPos
  http://docs.surfsaralabs.nl/projects/grid/en/latest/Pages/Practices/pilot_jobs.html

- DIRAC
  http://dirac.readthedocs.io/en/latest/index.html

- BigJob
  http://saga-project.github.io/BigJob/

High Performance Computing

Questions?

Exercises

- Hostname: `peregrine.hpc.rug.nl`
- Username: `see handouts`
- Password: see handouts

- Slides, go to:
  - https://redmine.hpc.rug.nl
  - Peregrine
  - Wiki
  - Course material