Consortium des Equipements
de Calcul Intensif
en Fédération Wallonie-Bruxelles

Introduction to Parallel Computing

damien.francois@uclouvain.be
November 2023

Agenda

1. General concepts, definitions, challenges
2. Hardware for parallel computing
3. Programming models
4. User tools

1.

General concepts

Why parallel? (simplified)

Speed up – Solve a problem faster
→ more processing power
(a.k.a. strong scaling)

Scale up – Solve a larger problem
→ more memory and network capacity
(a.k.a. weak scaling)

Scale out – Solve many problems
→ more storage capacity

Also: energy consumption is a cubic function of clock frequency so using 2 compute units is 8 times cheaper than using one unit with double the frequency

Parallelization involves:

- *decomposition* of the work
  - **distributing instructions** to processors
  - **distributing data** to memories

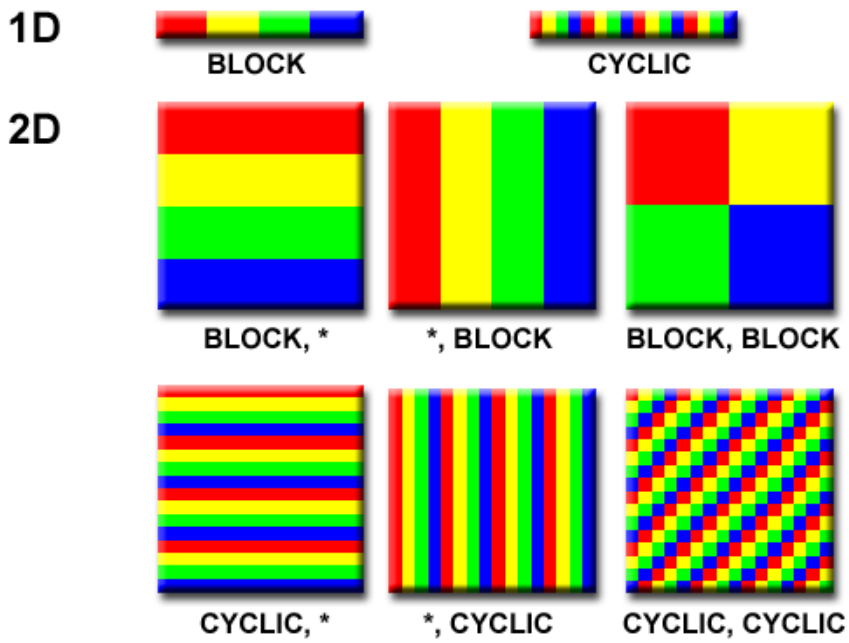
- *collaboration* of the workers
  - **synchronization** of the distributed work
  - **communication** of data

Decomposition of the work

- **Operation decomposition** : task-level parallelism
  - Multiple programs (functional decomposition)
  - Multiple instances of the same program (SPMD)
- **Data decomposition** : data-level parallelism

Decomposition of the work

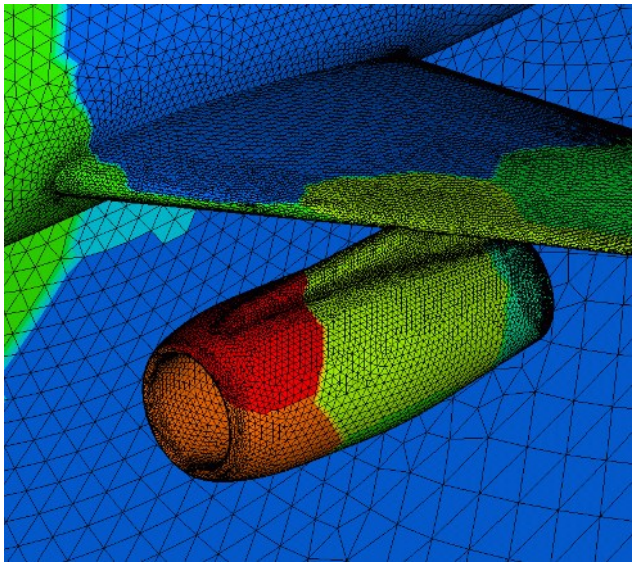
- **Operation decomposition** : task-level parallelism
- **Data decomposition** : data-level parallelism
  - **Block, cyclic**



https://nyu-cds.github.io/python-mpi/04-decomposition/

Decomposition of the work

- **Operation decomposition** : task-level parallelism
- **Data decomposition** : data-level parallelism
  - **Domain decomposition** : decomposition of work and data is done in a higher model, e.g. in the reality



https://fun3d.larc.nasa.gov/example-54.html

Collaboration of the workers

- **Synchronization of the workers**
  - **Synchronous** (SIMD) at the processor level ; the same processor instruction for each worker at any time ; (instruction level)
  - **Fine-grained** parallelism : subtasks communicate many times per second (typically at the loop level)
  - **Coarse-grained** parallelism : they do not communicate many times per second (typically function-call level)
  - **Embarrassingly parallel** : they rarely or never have to communicate (asynchronous)

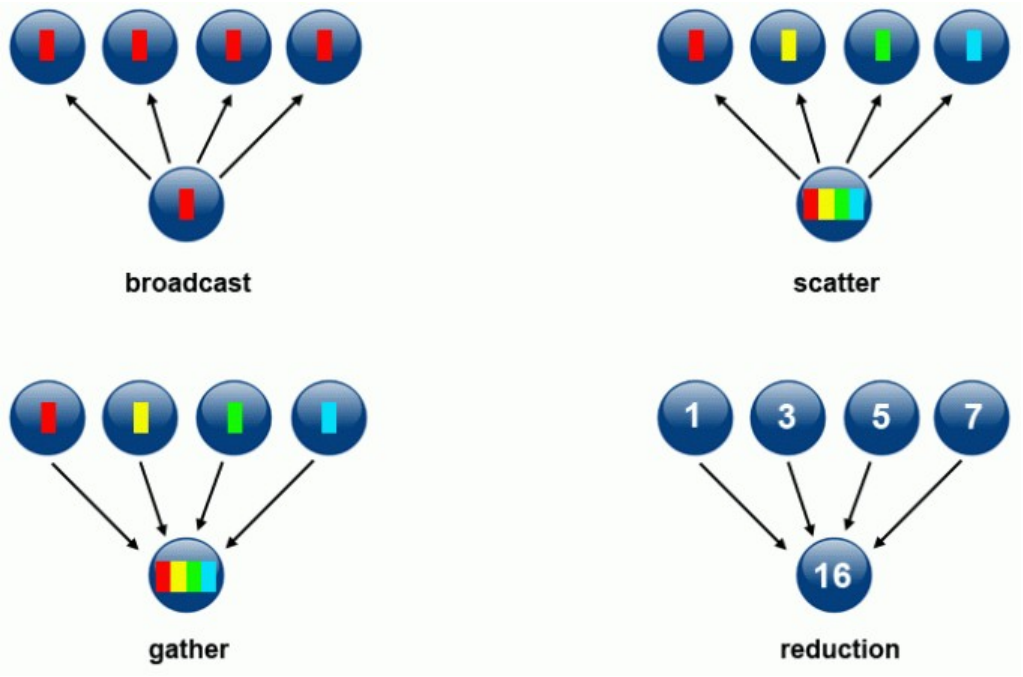
high

communicating / computing

low

Collaboration of the workers

- **Communication between workers**
  - Point to point
  - Broadcast
  - Scatter
  - Gather
  - Reduction



https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Does it work?
Speedup, Efficiency, Scalability

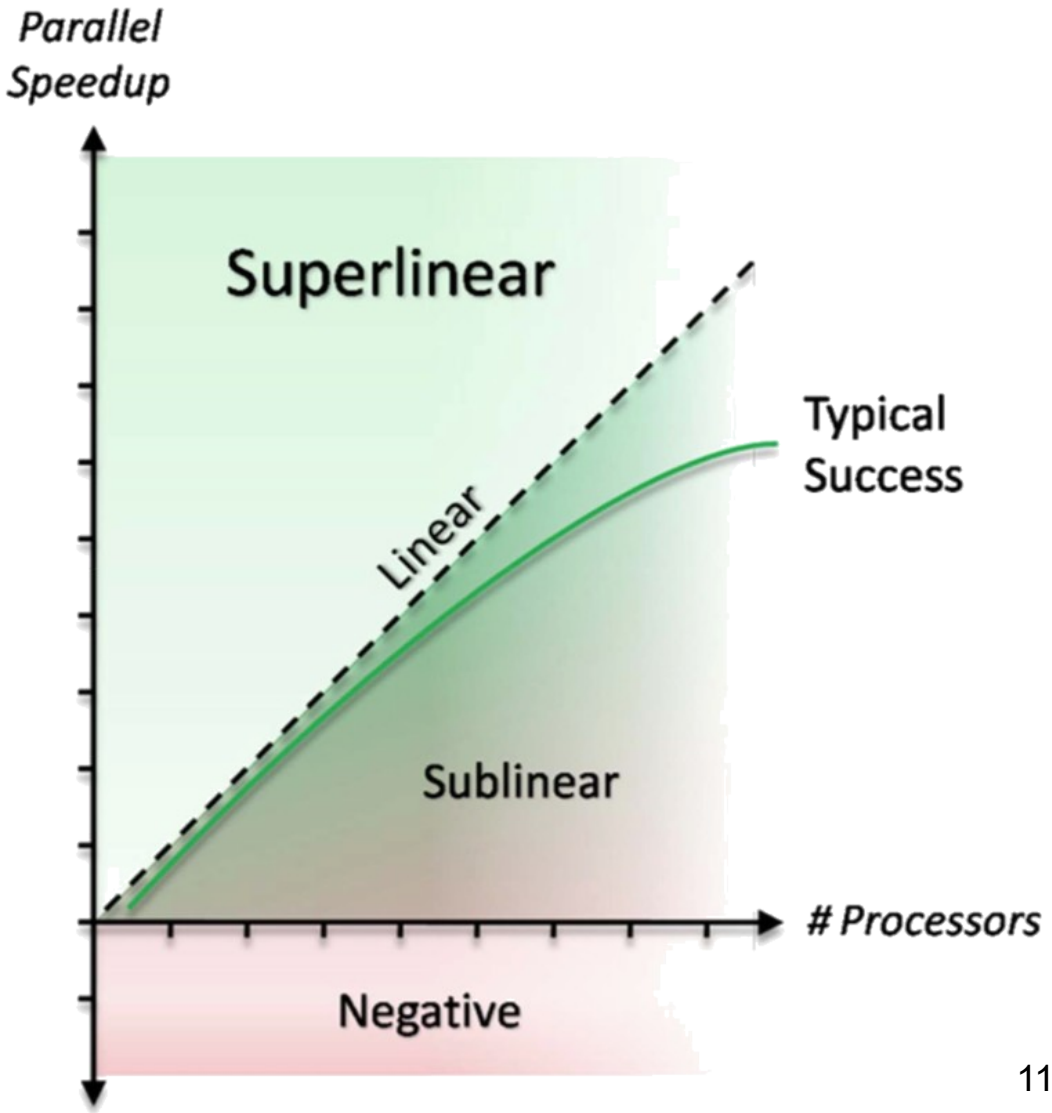
Time for serial operations

Speedup $$S = \frac{T_S}{T_P}$$

Time for parallel operations

Efficiency $$E = \frac{S}{p} = \frac{T_S}{pT_p}$$

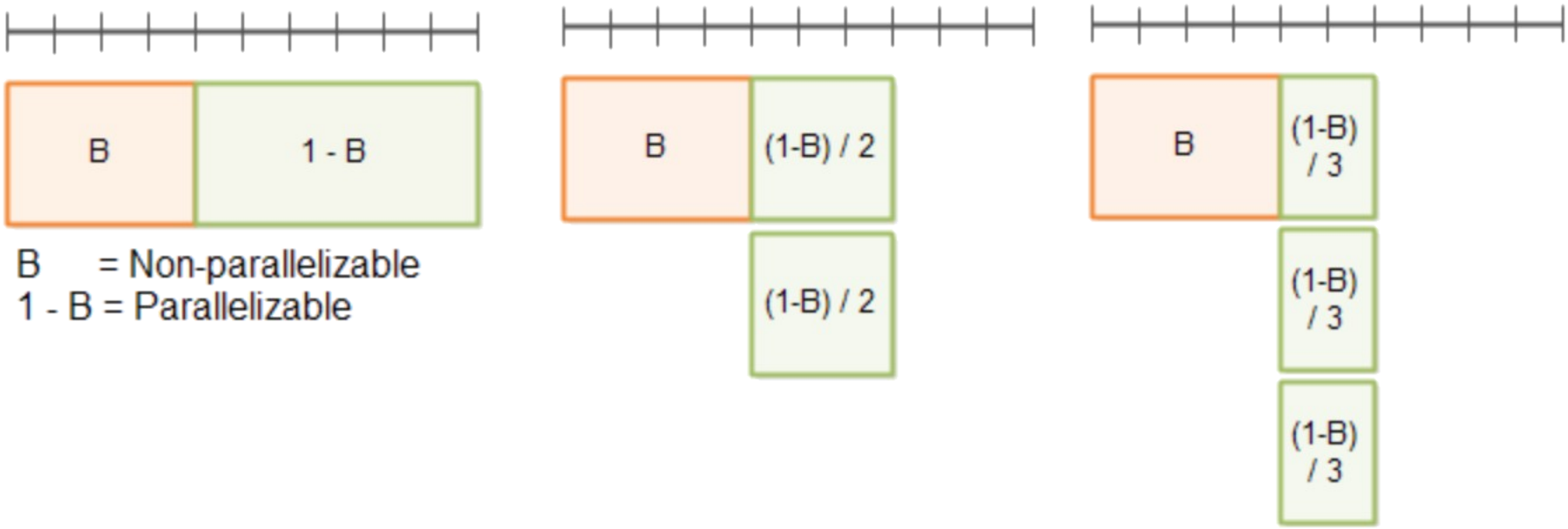
Number of processors



https://www.cs.uky.edu/~jzhang/CS621/chapter7.pdf

Challenge 1: Amdahl's Law

Not all the work can be decomposed



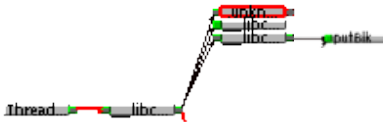
In parallel computing, Amdahl's law is mainly used
to predict the theoretical maximum speedup
for programs using multiple processors.

http://tutorials.jenkov.com/java-concurrency/amdahls-law.html

Challenge 2: Parallel overhead

Collaboration means communication and extra work

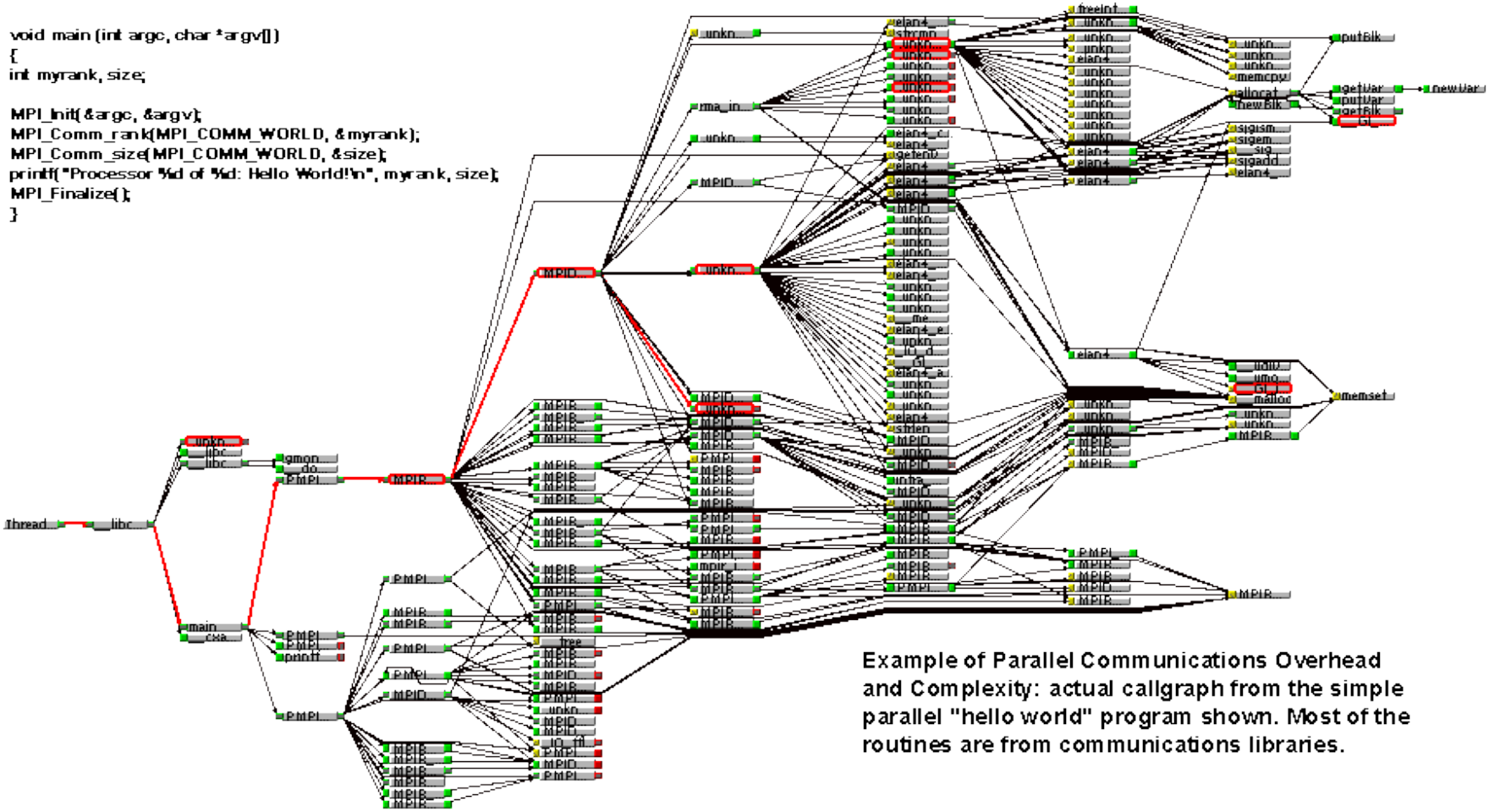
```
void main (int argc, char *argv[])
{

printf("Processor %d of %d: Hello World!\n",

}
```



https://computing.llnl.gov/tutorials/parallel_comp/images/helloWorldParallelCallgraph.gif

Challenge 2: Parallel overhead

Collaboration means communication and extra work

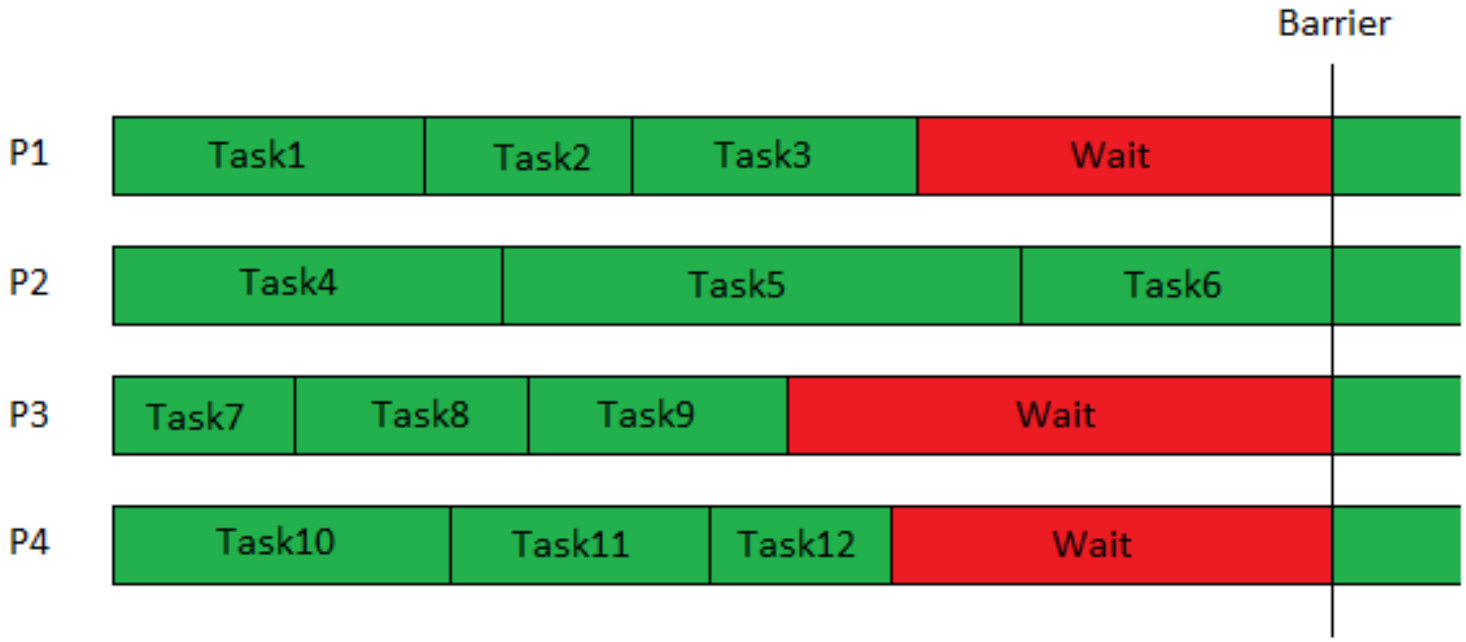


Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

https://computing.llnl.gov/tutorials/parallel_comp/images/helloWorldParallelCallgraph.gif

Challenge 3: Load imbalance

Parallelization is efficient only if every worker has the same amount of work



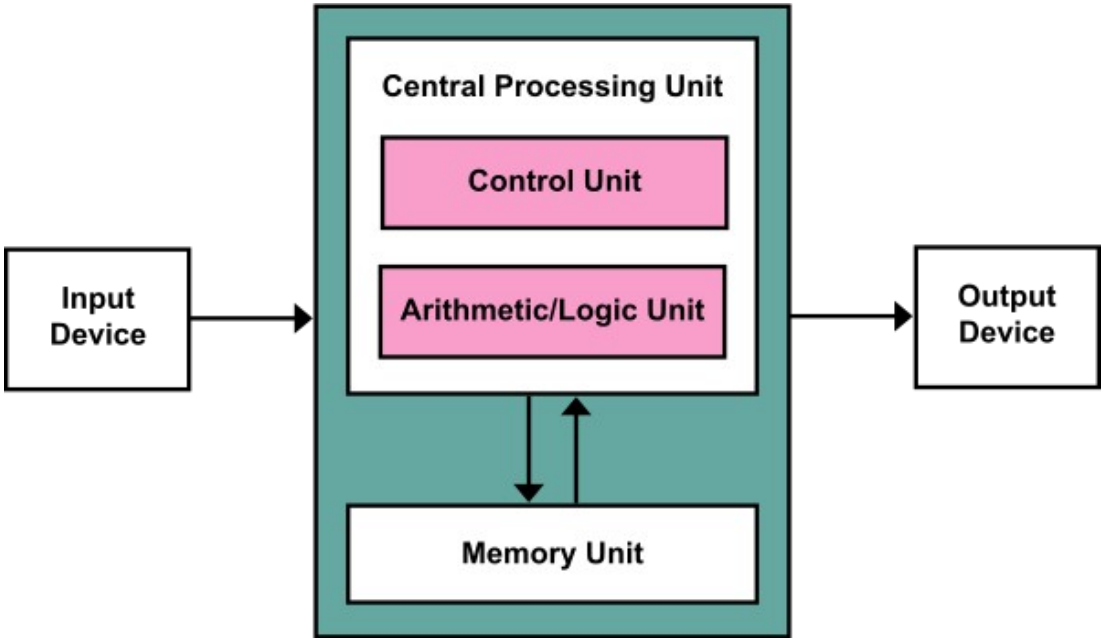
https://hpc-wiki.info/hpc/Load_Balancing

2.

Hardware for parallel computing

Von Neumann (serial) architecture

An abstract view of early computers



https://en.wikipedia.org/wiki/Von_Neumann_architecture

Parallelism at the CPU (core) level

- Instruction-level parallelism (**ILP**)
  - Instruction pipelining
  - Out-of-order execution
  - Speculative execution
  - ...
- Single Instruction Multiple Data (**SIMD**)

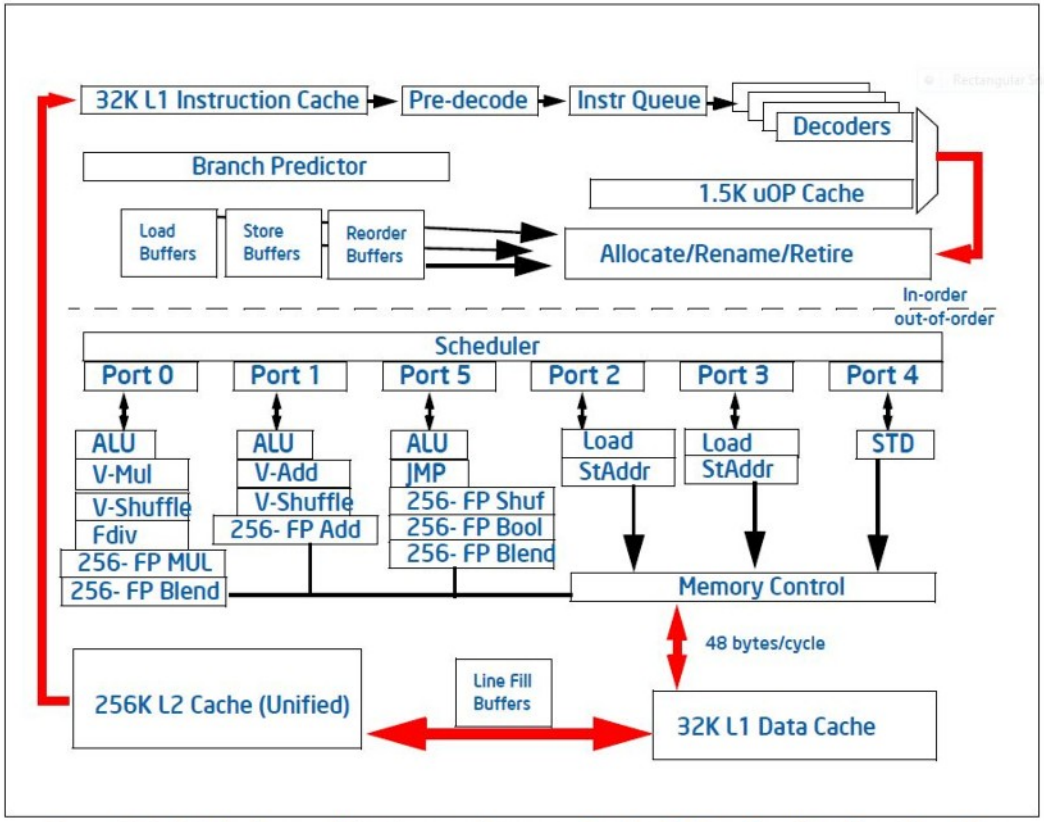
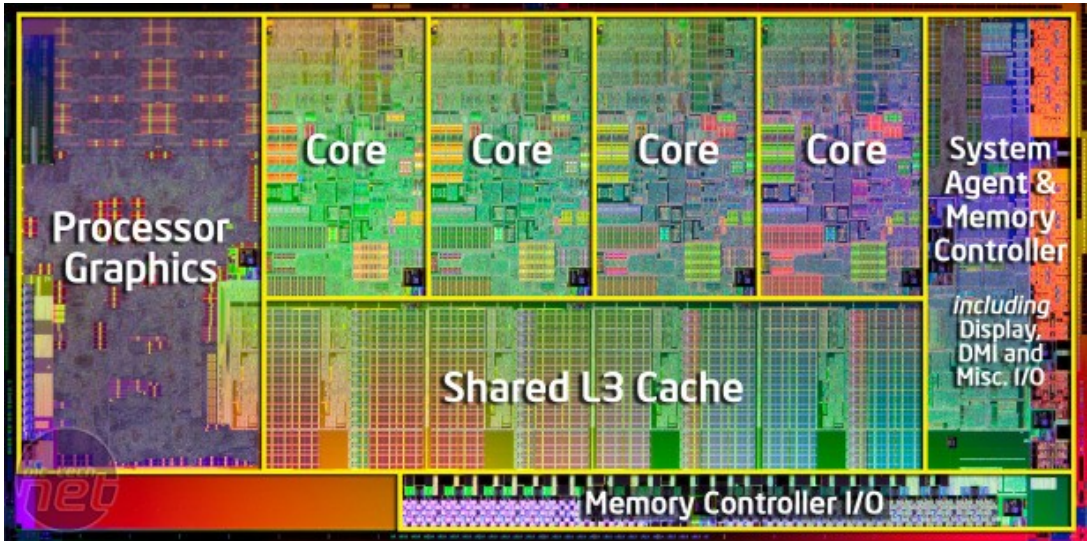


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

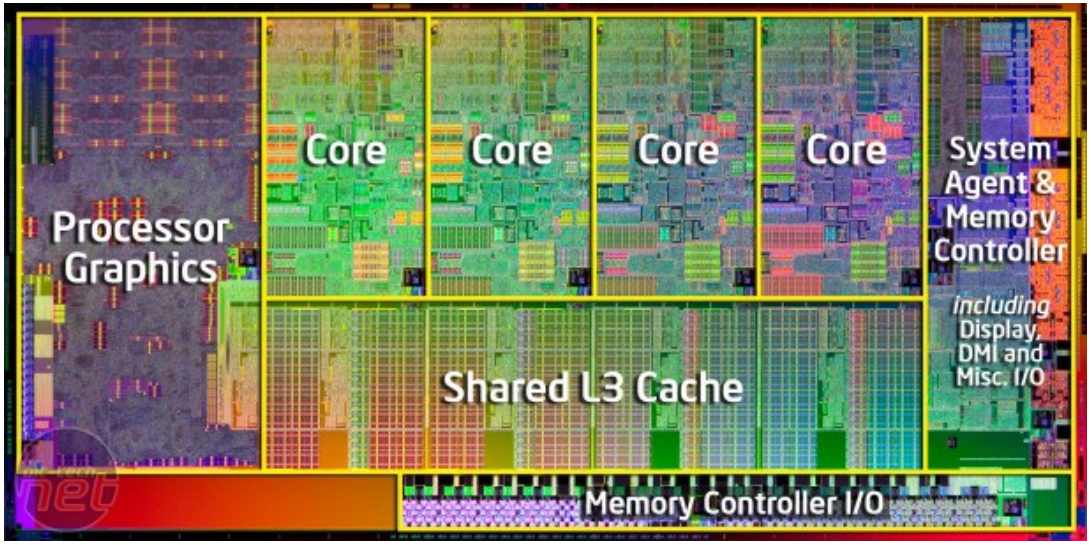
Parallelism at the chip (socket) level

- **Multicore** parallelism

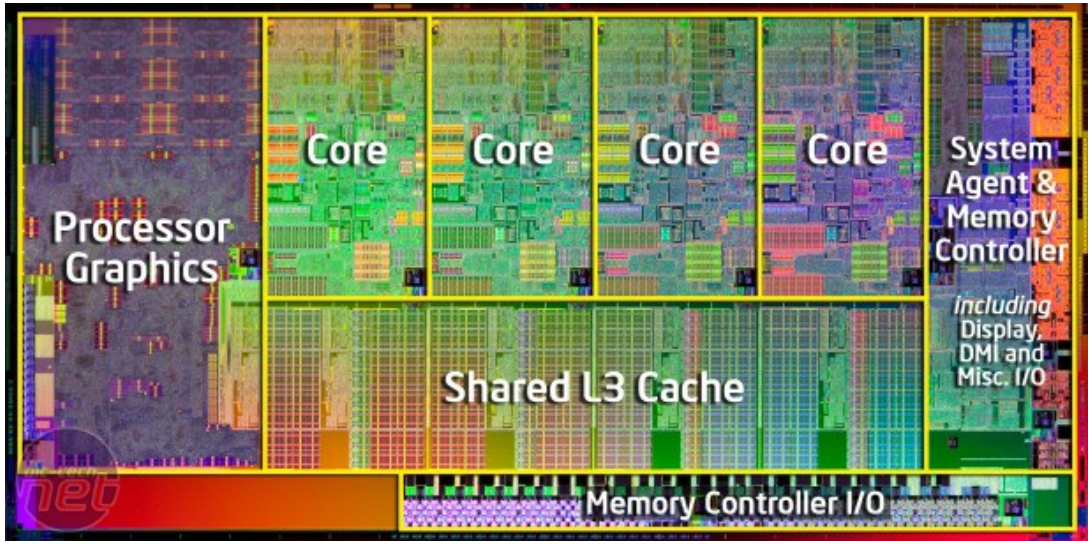


Parallelism at the computer level

- Multi-socket parallelism
  - SMP
  - NUMA



- Accelerators



Parallelism at the data center level

Multi-node parallelism

Parallelism at the data center level

Cluster computing

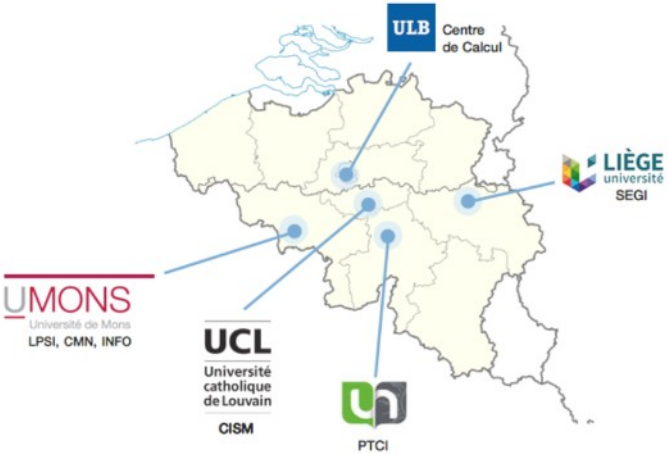
C.E.C.I

Consortium des Équipements de Calcul Intensif

6 clusters, 10k cores, 1 login, 1 home directory

About

CÉCI is the 'Consortium des Équipements de Calcul Intensif'; a consortium of high-performance computing centers of UCL, ULB, ULg, UMons, and UNamur. Read more.



The common storage is functional!

Have you tried it yet? More info...

Latest News

SATURDAY, 23 SEPTEMBER 2017

A CECI user pictured in the ULiège news!

The ULiège website published a story (in French) about the work of Denis Baurain and his collaborators on the Tier-1 cluster Zenobe that lead to a publication in Nature Ecology & Evolution.

TUESDAY, 01 AUGUST 2017

Ariel Lozano is the new CÉCI logisticien

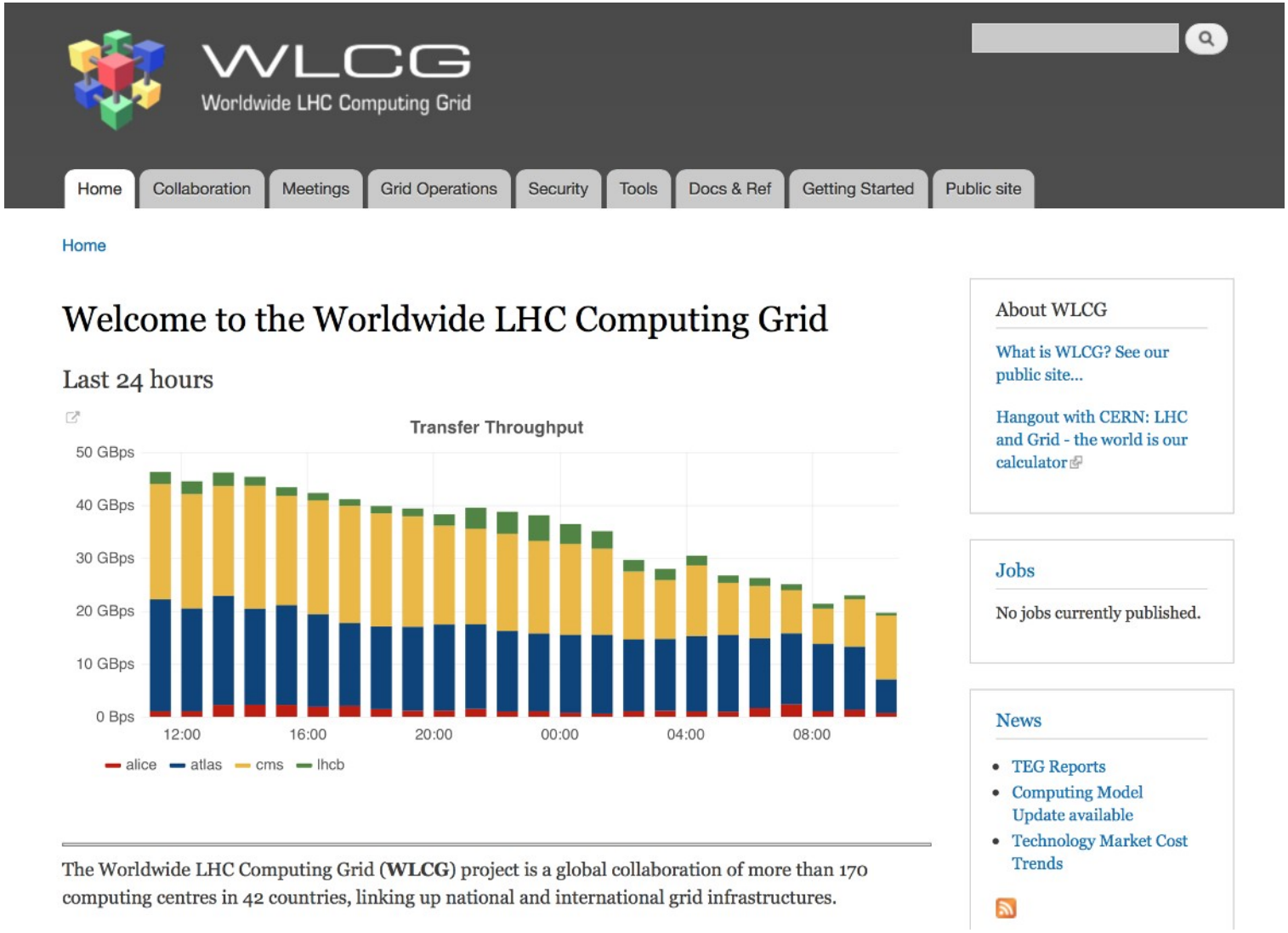
We are happy to announce the hire of a new CECI logisticien: Ariel Lozano. Welcome Ariel!

Parallelism at the data center level

Cloud computing “someone else’s cluster”

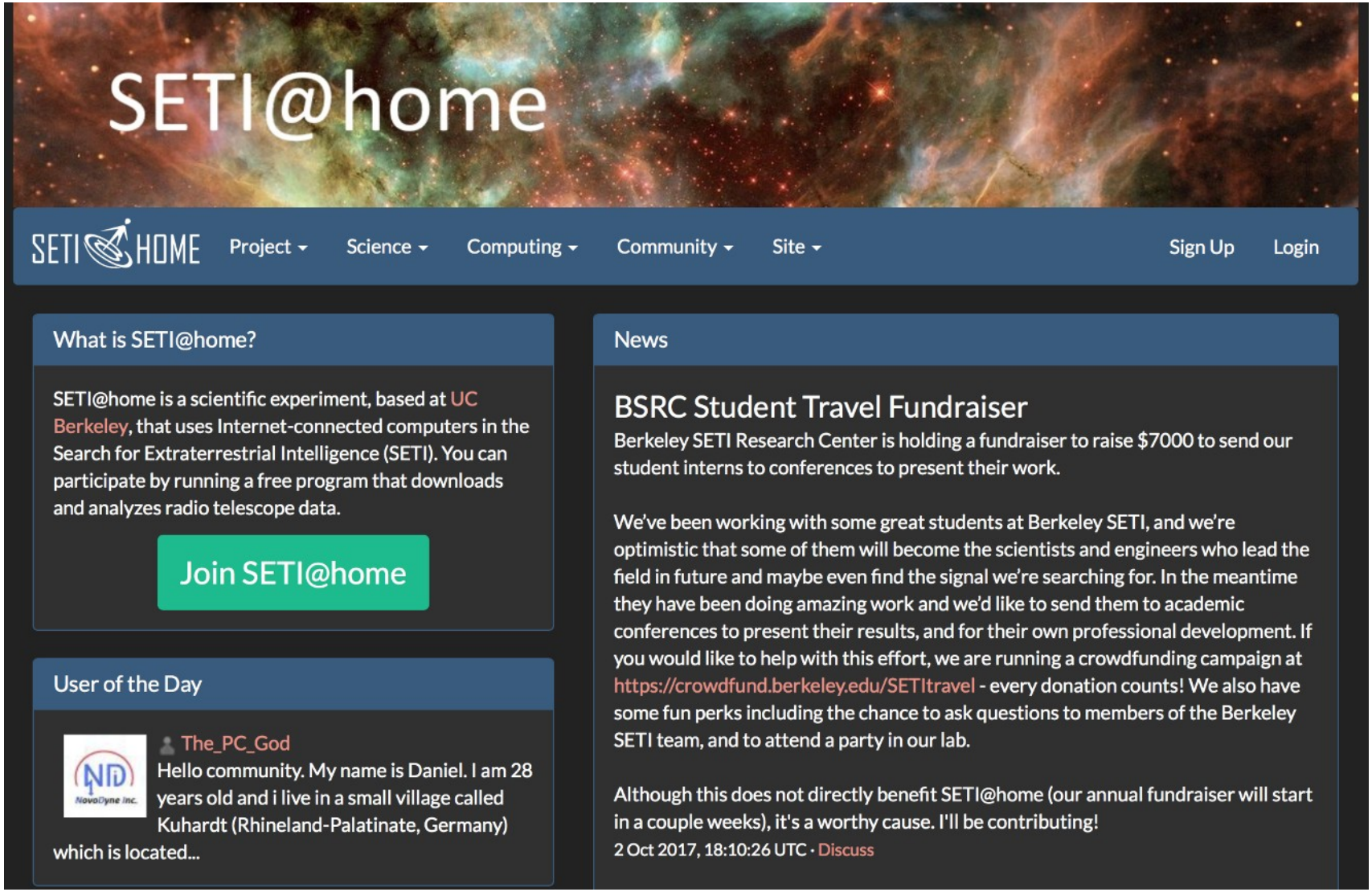
Parallelism at the world level

Grid computing – “cluster of clusters”



Parallelism at the world level

Distributed computing – “no unused cycle”



3.

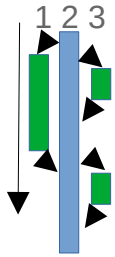
Programming paradigms and programming models

Is parallization automagic?

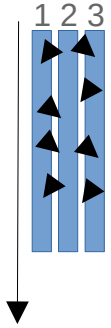
- **ILP**: yes, by the CPU and/or the compiler
- **SIMD**: mostly, by the compiler, but it can be fooled by your code (→ *pragmas*)
- **Intra-node**: can be if the library/software you use is designed for it (→ **_NUM_THREADS env vars, be aware of NUMA placement issues*)
- **GPUs**: can be if the library/software you use is designed for it (→ *CUDA_VISIBLE_DEVICES env vars*, *be aware of GPU/CPU memory transfers*)
- **Inter-node**: never automagic. You will at least need to explicitly start processes (→ *be aware of the network capabilities)*

Main parallel programming paradigms

- **Task-farming:**
  - Master program distributes work to worker programs (*leader/follower*); or
  - Worker programs pick up tasks from pool (*work stealing*).

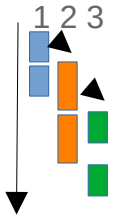


- **SPMD** (Single program multiple data)
  - A single program that contains both the logic for distributing work and computing
  - Multiple instances are started and “linked” together
  - Instances are identified with a distinct index



Other parallel programming paradigms

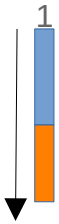
- **MPMD (Multiple program multiple data)**
- **Pipelining** : workers take care of a subtask in the processing chain and pass the intermediate result to the next worker



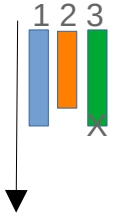
- **Divide and Conquer** :
  - workers are spawned at need and report their result to the parent

  - Speculative parallelism : workers are spawned and result possibly discarded

```
CPU1: If (very_long_computation())
CPU1: then
CPU1:     do A
CPU1: else
CPU1:     do B
```




```
CPU1: res=very_long_computation()
CPU2: do A
CPU3: do B
CPU1: if (res) discard B else discard A
```



Programming models

- Single computer:
  - **CPUs**: PThreads, *OpenMP*, TBB, OpenCL, ...
  - **Accelerators**: *CUDA*, OpenCL, *OpenAcc/OpenMP*, SYCL, Hipp, ROCm, ...
- Multi-computer:
  - **Clusters**:
    - Message passing: *MPI*, PVM
    - PGAS: CoArray Fortran, UPC, Global Arrays
  - **Clouds:** MapReduce, Spark RDD
  - **Distributed computing**: BOINC

4.

User tools

that GNU/Linux offers

4.1. Parallelized utilities

Some tools have a parallelized counterpart, or parallel options. Examples:

serial		parallel
gzip	⟷	pigz
grep	⟷	ripgrep, singrep
ssh	⟷	clustershell
sort	⟷	sort --parallel
scp	⟷	bbcp
bc	⟷	bcx
...		...

4.2. Job control & Parallel processes in Bash

Consider the following example program

```
dfr@hmem00:~/parcomp $ cat lower.sh
#!/bin/bash
#
# Usage:
#    ./lower.sh [input_file [output_file]]
#
# Make ACTG chars lower case with extra processing.
#
# If output_file is not defined, stdout is used
# If input_file and output_file are not defined, stdin and stdout are used.

while read line; do
sleep 1
echo $line | tr ACTG actg >> ${2-/dev/stdout}
done < ${1-/dev/stdin}

dfr@hmem00:~/parcomp $ cat d.txt
G
C
A
G
dfr@hmem00:~/parcomp $ ./lower.sh d.txt
g
c
a
g
dfr@hmem00:~/parcomp $
```

It is written in Bash and just transforms some upper case letters to lower case

4.2. Job control & Parallel processes in Bash

Run the program twice

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # Foreground: commands end with ';'
dfr@hmem00:~/parcomp $ time { ./lower.sh d1.txt r1.txt ; ./lower.sh d1.txt r2.txt ; };

real    0m8.033s
user    0m0.004s
sys     0m0.019s
dfr@hmem00:~/parcomp $ # Background, in parallel: commands end with '&' and 'wait' necessary
dfr@hmem00:~/parcomp $ time { ./lower.sh d2.txt r1.txt & ./lower.sh d2.txt r2.txt & wait ; };
[1] 49722
[2] 49723
[1]-  Done                    ./lower.sh d2.txt r1.txt
[2]+  Done                    ./lower.sh d2.txt r2.txt

real    0m4.011s
user    0m0.004s
sys     0m0.005s
dfr@hmem00:~/parcomp $
```


https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html

4.2. Job control & Parallel processes in Bash

Run the program twice and measure the time it takes

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # Foreground: commands end with ';'
dfr@hmem00:~/parcomp $ time { ./lower.sh d1.txt r1.txt ; ./lower.sh d1.txt r2.txt ; };

real    0m8.033s
user    0m0.004s
sys     0m0.019s
dfr@hmem00:~/parcomp $ # Background, in parallel: commands end with '&' and 'wait' necessary
dfr@hmem00:~/parcomp $ time { ./lower.sh d2.txt r1.txt & ./lower.sh d2.txt r2.txt & wait ; };
[1] 49722
[2] 49723
[1]-  Done                    ./lower.sh d2.txt r1.txt
[2]+  Done                    ./lower.sh d2.txt r2.txt

real    0m4.011s
user    0m0.004s
sys     0m0.005s
dfr@hmem00:~/parcomp $
```


https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html

4.2. Job control & Parallel processes in Bash

Run the program twice and measure the time it takes

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # Foreground: commands end with ';'
dfr@hmem00:~/parcomp $ time { ./lower.sh d1.txt r1.txt ; ./lower.sh d1.txt r2.txt ; };

real    0m8.033s
user    0m0.004s
sys     0m0.019s
dfr@hmem00:~/parcomp $ # Background, in parallel: commands end with '&' and 'wait' necessary
dfr@hmem00:~/parcomp $ time { ./lower.sh d2.txt r1.txt & ./lower.sh d2.txt r2.txt & wait ; };
[1] 49722
[2] 49723
[1]-  Done                    ./lower.sh d2.txt r1.txt
[2]+  Done                    ./lower.sh d2.txt r2.txt

real    0m4.011s
user    0m0.004s
sys     0m0.005s
dfr@hmem00:~/parcomp $
```


https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html

4.2. Job control & Parallel processes in Bash

Run the program twice “in the background” and measure the time

dfr@hmem00 — bash

```
dfr@hmem00:~/parcomp $ # Foreground: commands end with ';'
dfr@hmem00:~/parcomp $ time { ./lower.sh d1.txt r1.txt ; ./lower.sh d1.txt r2.txt ; };

real    0m8.033s
user    0m0.004s
sys     0m0.019s
dfr@hmem00:~/parcomp $ # Background, in parallel: commands end with '&' and 'wait' necessary
dfr@hmem00:~/parcomp $ time { ./lower.sh d2.txt r1.txt & ./lower.sh d2.txt r2.txt & wait ; };
[1] 49722
[2] 49723
[1]-  Done                    ./lower.sh d2.txt r1.txt
[2]+  Done                    ./lower.sh d2.txt r2.txt

real    0m4.011s
user    0m0.004s
sys     0m0.005s
dfr@hmem00:~/parcomp $
```


https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html

4.2. Job control & Parallel processes in Bash

Parallel for loop in Bash:

```
for i in {1..10}; do
  command1
  command2
done
```

```
for i in {1..10}; do
(
  command1
  command2
) &
done; wait
```

`(...) &` : creates a sub-shell with all commands in the bloc and start it in the background
`wait` : barrier to synchronize all sub-shells

4.3.1. One program and many files

The xargs command distributes data from stdin to program

```
dfr@hmem00:~/parcomp $ ls d?.txt
d1.txt  d2.txt  d3.txt  d4.txt
dfr@hmem00:~/parcomp $ ls d?.txt | xargs -n 1 echo "File: "
File:  d1.txt
File:  d2.txt
File:  d3.txt
File:  d4.txt
dfr@hmem00:~/parcomp $ time { ls d?.txt | xargs -n 1 ./lower.sh > /dev/null ; }

real    0m16.041s
user    0m0.010s
sys     0m0.006s
dfr@hmem00:~/parcomp $ time { ls d?.txt | xargs -n 1 -P 4 ./lower.sh > /dev/null ; }

real    0m4.014s
user    0m0.008s
sys     0m0.016s
dfr@hmem00:~/parcomp $
```

Equivalent to
./lower.sh d1.txt ;
./lower.sh d2.txt ;
./lower.sh d3.txt ;
./lower.sh d3.txt ;

Equivalent to
./lower.sh d1.txt &
./lower.sh d2.txt &
./lower.sh d3.txt &
./lower.sh d3.txt &
wait

4.3.2. Several programs and one file

Using UNIX pipes for pipelining operations

./upper.sh waits for ./lower.sh to finish
Note the intermediate file

dfr@hmem00 — bash

```
dfr@hmem00:~/parcomp $ # Using an intermediay file
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmp.txt ; ./upper.sh tmp.txt res.txt ; }

real    0m8.033s
user    0m0.005s
sys     0m0.017s
dfr@hmem00:~/parcomp $ # Using pipes (as our programs can handle stdin and stdout)
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt | ./upper.sh > res.txt ; }

real    0m5.014s
user    0m0.006s
sys     0m0.009s
dfr@hmem00:~/parcomp $ mkfifo tmpfifo
dfr@hmem00:~/parcomp $ ls -l tmpfifo
prw-rw-r-- 1 dfr dfr 0 Oct  7 10:27 tmpfifo
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmpfifo & ./upper.sh tmpfifo res.txt ; }
[1] 65343
[1]+  Done                    ./lower.sh d.txt tmpfifo

real    0m5.013s
user    0m0.002s
sys     0m0.007s
dfr@hmem00:~/parcomp $
```

4.3.2. Several programs and one file

Using UNIX pipes for pipelining operations

./upper.sh waits for ./lower.sh to finish
Note the intermediate file

dfr@hmem00 — bash

```
dfr@hmem00:~/parcomp $ # Using an intermediay file
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmp.txt ; ./upper.sh tmp.txt res.txt ; }

real    0m8.033s
user    0m0.005s
sys     0m0.017s
dfr@hmem00:~/parcomp $ # Using pipes (as our programs can handle stdin and stdout)
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt | ./upper.sh > res.txt ; }

real    0m5.014s
user    0m0.006s
sys     0m0.009s
dfr@hmem00:~/parcomp $ mkfifo tmpfifo
dfr@hmem00:~/parcomp $ ls -l tmpfifo
prw-rw-r-- 1 dfr dfr 0 Oct  7 10:27 tmpfifo
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmpfifo & ./upper.sh tmpfifo res.txt ; }
[1] 65343
[1]+  Done                    ./lower.sh d.txt tmpfifo

real    0m5.013s
user    0m0.002s
sys     0m0.007s
dfr@hmem00:~/parcomp $
```

4.3.2. Several programs and one file

Using UNIX fifos for pipelining operations

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # Using an intermediay file
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmp.txt ; ./upper.sh tmp.txt res.txt ; }

real    0m8.033s
user    0m0.005s
sys     0m0.017s
dfr@hmem00:~/parcomp $
dfr@hmem00:~/parcomp $

dfr@hmem00:~/parcomp $ mkfifo tmpfifo
dfr@hmem00:~/parcomp $ ls -l tmpfifo
prw-rw-r-- 1 dfr dfr 0 Oct  7 10:27 tmpfifo
dfr@hmem00:~/parcomp $ time { ./lower.sh d.txt tmpfifo & ./upper.sh tmpfifo res.txt ; }
[1] 65343
[1]+  Done                    ./lower.sh d.txt tmpfifo

real    0m5.013s
user    0m0.002s
sys     0m0.007s
dfr@hmem00:~/parcomp $
```

A FIFO file is a “fake” file to which a process can write at the end and another can read at the beginning

./upper.sh starts reading as soon as ./lower.sh starts writing

4.3.3. One program and one large file

The split command distributes data from stdin to program

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # One process to process the whole file
dfr@hmem00:~/parcomp $ time { cat d.txt | ./lower.sh > res.txt

real    0m4.014s
user    0m0.003s
sys     0m0.009s
dfr@hmem00:~/parcomp $ # Four processes handling one line in round robin fashion
dfr@hmem00:~/parcomp $ time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh >res.txt ; }

real    0m1.011s
user    0m0.009s
sys     0m0.021s
dfr@hmem00:~/parcomp $ !! & top -u dfr -bn1 | grep lower
time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh >res.txt ; } & top -u dfr -bn1 | gr
ep lower
[1] 12817
12822 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12823 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12824 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12825 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
dfr@hmem00:~/parcomp $
real    0m1.011s
user    0m0.011s
sys     0m0.019s

[1]+  Done                    time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh > res
.txt; }
dfr@hmem00:~/parcomp $
```

Split the file and start 4 processes

Need recent version of Coreutils/8.22-goolf-1.4.10

4.3.3. One program and one large file

The split command distributes data from stdin to program

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ # One process to process the whole file
dfr@hmem00:~/parcomp $ time { cat d.txt | ./lower.sh > res.txt

real    0m4.014s
user    0m0.003s
sys     0m0.009s
dfr@hmem00:~/parcomp $ # Four processes handling one line in round robin fashion
dfr@hmem00:~/parcomp $ time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh >res.txt ; }

real    0m1.011s
user    0m0.009s
sys     0m0.021s
dfr@hmem00:~/parcomp $ !! & top -u dfr -bn1 | grep lower
time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh >res.txt ; } & top -u dfr -bn1 | gr
ep lower
[1] 12817
12822 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12823 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12824 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
12825 dfr       20   0  103m 1252 1052 S  0.0  0.0   0:00.00 lower.sh
dfr@hmem00:~/parcomp $
real    0m1.011s
user    0m0.011s
sys     0m0.019s

[1]+  Done                    time { cat d.txt | split --unbuffered --number r/4 --filter ./lower.sh > res
.txt; }
dfr@hmem00:~/parcomp $
```

Split the file and start 4 processes

Need recent version of Coreutils/8.22-goolf-1.4.10

4.3.4. Several programs and many files

A Makefile describes dependencies and is executed with ‘make’

dfr@hmem00 — bash

```
# Sample Makefile to process each file with
# lower.sh then upper.sh
#
all: d1.res d2.res d3.res d4.res

# Build intermediary files
%.tmp: %.txt
    ./lower.sh $< $@


# Build final result
%.res: %.tmp
    ./upper.sh $< $@
```

This means: build a .tmp file from a similarly-named .txt file using the ./lower program

This means: build a .res file from a similarly-named .tmp file using the ./upper.sh program

"Makefile" 14L, 219C written 14,0-1 All

https://www.gnu.org/software/make/manual/html_node/index.html

4.3.4. Several programs and many files

The ‘make’ command can operate in parallel

dfr@hmem00 — bash

```
dfr@hmem00:~/parcomp $ time make
./lower.sh d1.txt d1.tmp
./upper.sh d1.tmp d1.res
./lower.sh d2.txt d2.tmp
./upper.sh d2.tmp d2.res
./lower.sh d3.txt d3.tmp
./upper.sh d3.tmp d3.res
./lower.sh d4.txt d4.tmp
./upper.sh d4.tmp d4.res
rm d1.tmp d2.tmp d4.tmp d3.tmp

real    0m32.260s
user    0m0.028s
sys     0m0.099s
dfr@hmem00:~/parcomp $ rm *res
dfr@hmem00:~/parcomp $ time make -j 4
./lower.sh d1.txt d1.tmp
./lower.sh d2.txt d2.tmp
./lower.sh d3.txt d3.tmp
./lower.sh d4.txt d4.tmp
./upper.sh d1.tmp d1.res
./upper.sh d2.tmp d2.res
./upper.sh d4.tmp d4.res
./upper.sh d3.tmp d3.res
rm d1.tmp d2.tmp d4.tmp d3.tmp

real    0m8.163s
user    0m0.025s
```

Summary

- You have either
  - one very large file to process
    - with one program: split
    - with several programs: fifo (or pipes)
  - many files to process
    - with one program xargs
    - with many programs make

4.4. GNU Parallel

GNU Parallel

GNU **parallel** is a shell tool for executing jobs in parallel using one or more computers. A job can be a single command or a small script that has to be run for each of the lines in the input. The typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe. GNU **parallel** can then split the input and pipe it into commands in parallel.

If you use xargs and tee today you will find GNU **parallel** very easy to use as GNU **parallel** is written to have the same options as xargs. If you write loops in shell, you will find GNU **parallel** may be able to replace most of the loops and make them run faster by running several jobs in parallel.

GNU **parallel** makes sure output from the commands is the same output as you would get had you run the commands sequentially. This makes it possible to use output from GNU **parallel** as input for other programs.

GNUparallel

For people who live life in the parallel lane.

For each line of input GNU **parallel** will execute *command* with the line as arguments. If no *command* is given, the line of input is executed. Several lines will be run in parallel. GNU **parallel** can often be used as a substitute for **xargs** or **cat | bash**.

More complicated to use but very powerful
Might not be available everywhere

4.4. GNU Parallel

- Syntax: parallel *command* ::: *argument list*

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ parallel echo ::: 1 2 3 4
1
2
3
4
dfr@hmem00:~/parcomp $ parallel echo ::: {1..10}
1
2
3
4
5
6
7
8
9
10
dfr@hmem00:~/parcomp $ time parallel sleep ::: {1..10}

real    0m11.200s
user    0m0.206s
sys     0m0.129s
dfr@hmem00:~/parcomp $ parallel echo ::: d?.txt
d1.txt
d2.txt
d3.txt
d4.txt
dfr@hmem00:~/parcomp $
```

4.4. GNU Parallel

- Syntax: {} as argument placeholder.

```
d1.txt
d2.txt
d3.txt
d4.txt
dfr@hmem00:~/parcomp $ parallel echo {} {.}.res ::: d?.txt
d1.txt d1.res
d2.txt d2.res
d3.txt d3.res
d4.txt d4.res
dfr@hmem00:~/parcomp $ parallel echo {} ::: ../parcomp/d?.txt
../parcomp/d1.txt
../parcomp/d2.txt
../parcomp/d3.txt
../parcomp/d4.txt
dfr@hmem00:~/parcomp $ parallel echo {/} ::: ../parcomp/d?.txt
d1.txt
d2.txt
d3.txt
d4.txt
dfr@hmem00:~/parcomp $
dfr@hmem00:~/parcomp $
dfr@hmem00:~/parcomp $
```

4.4. GNU Parallel

- Multiple parameters and --xapply

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ parallel echo ::: 1 2 3 4 ::: A B
1 A
1 B
2 A
2 B
3 A
3 B
4 A
4 B
dfr@hmem00:~/parcomp $ parallel --xapply echo ::: 1 2 3 4 ::: A B C D
1 A
2 B
3 C
4 D
dfr@hmem00:~/parcomp $ parallel  echo {1} and {2} ::: 1 2 3 4 ::: A B C D
1 and A
1 and B
1 and C
1 and D
2 and A
2 and B
2 and C
2 and D
3 and A
3 and B
3 and C
3 and D
4 and A
```

4.4. GNU Parallel

- When arguments are in a file : use :::: (4x ':')

```
dfr@hmem00 — bash
dfr@hmem00:~/parcomp $ cat experiments.csv
Number,Letter
1,A
2,B
3,B
3,A
4,C
5,C
5,A
dfr@hmem00:~/parcomp $ parallel --colsep ',' --header '\n' echo {Number} {Letter} :::: experiments.csv
1 A
2 B
3 B
3 A
4 C
5 C
5 A
dfr@hmem00:~/parcomp $
```

4.4. GNU Parallel

A terminal for every job

Using **--tmux** GNU **parallel** can start a terminal for every job run:

```
seq 10 20 | parallel --tmux 'echo start {}; sleep {}; echo done {}'
```

This will tell you to run something similar to:

```
tmux -S /tmp/tmsrPr00 attach
```

Using normal **tmux** keystrokes (CTRL-b n or CTRL-b p) you can cycle between windows of the running jobs. When a job is finished it will pause for 10 seconds before closing the window.

https://www.gnu.org/software/parallel/parallel_tutorial.html

4.4. GNU Parallel

Interactivity

GNU **parallel** can ask the user if a command should be run using **--interactive**:

```
parallel --interactive echo ::: 1 2 3
```

Output:

```
echo 1 ?...y
echo 2 ?...n
1
echo 3 ?...y
3
```


https://www.gnu.org/software/parallel/parallel_tutorial.html

4.4. GNU Parallel

Timing

Some jobs do heavy I/O when they start. To avoid a thundering herd GNU **parallel** can delay starting new jobs. **--delay** *X* will make sure there is at least *X* seconds between each start:

```
parallel --delay 2.5 echo Starting {}\;date ::: 1 2 3
```

Output:

```
Starting 1
Thu Aug 15 16:24:33 CEST 2013
Starting 2
Thu Aug 15 16:24:35 CEST 2013
Starting 3
Thu Aug 15 16:24:38 CEST 2013
```


https://www.gnu.org/software/parallel/parallel_tutorial.html

4.4. GNU Parallel

Progress information

Based on the runtime of completed jobs GNU **parallel** can estimate the total runtime:

```
parallel --eta sleep ::: 1 3 2 2 1 3 3 2 1
```

Output:

```
Computers / CPU cores / Max jobs to run
1:local / 2 / 2

Computer:jobs running/jobs completed/%of started jobs/
  Average seconds to complete
ETA: 2s 0left 1.11avg  local:0/9/100%/1.1s
```


https://www.gnu.org/software/parallel/parallel_tutorial.html

4.4. GNU Parallel

With a joblog GNU **parallel** can be stopped and later pickup where it left off. It it important that the input of the completed jobs is unchanged.

```
parallel --joblog /tmp/log exit  ::: 1 2 3 0
cat /tmp/log
parallel --resume --joblog /tmp/log exit  ::: 1 2 3 0 0 0
cat /tmp/log
```

Output:

```
Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0

Seq Host Starttime      Runtime Send Receive Exitval Signal Command
1   :    1376580069.544 0.008   0    0       1       0      exit 1
2   :    1376580069.552 0.009   0    0       2       0      exit 2
3   :    1376580069.560 0.012   0    0       3       0      exit 3
4   :    1376580069.571 0.005   0    0       0       0      exit 0
5   :    1376580070.028 0.009   0    0       0       0      exit 0
6   :    1376580070.038 0.007   0    0       0       0      exit 0
```


https://www.gnu.org/software/parallel/parallel_tutorial.html

Other interesting options

`--pipe`	Split a file
`-S`	Use remote servers through SSH
`-j n`	Run n jobs in parallel
`-k`	Keep same order
`--delay n`	Ensure there are n seconds between each start
`--timeout n`	Kill task after n seconds if still running

Author asks to be cited: O. Tange (2011): *GNU Parallel - The Command-Line Power Tool*, The USENIX Magazine, February 2011:42-47.

Homework

Reproduce the examples from the previous slides
with ./lower and ./upper.sh

using GNU Parallel

Solutions

- One program and many files

```
$ time parallel -k ./lower.sh {} > res.txt  ::: d?.txt
```

- One program and one large file

```
$ time cat d.txt | parallel -k -N1 --pipe ./lower.sh {} > res.txt
```

- Several programs and several files

```
$ time { parallel ./lower.sh {} {.}.tmp ::: d?.txt ; \
> parallel  ./upper.sh {} {.}.res ::: d?.tmp ; }
```

Summary

1. General concepts, definitions, challenges
2. Hardware for parallel computing
3. Programming models
4. User tools