Consortium des Equipements
de Calcul Intensif
en Fédération Wallonie-Bruxelles

Introduction to Parallel Computing

damien.francois@uclouvain.be
November 2024

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

Does it work?
Speedup, Efficiency, Scalability

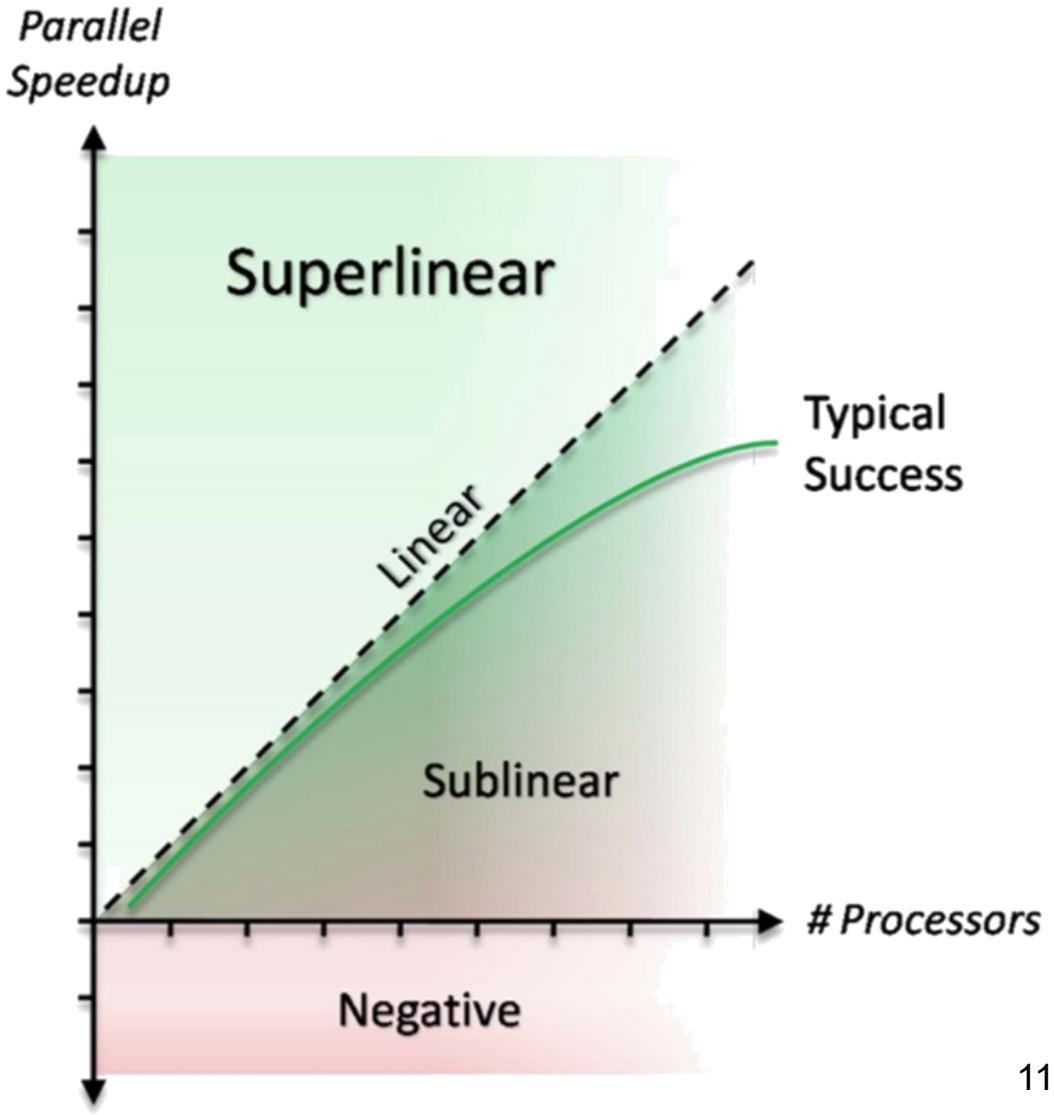
Time for serial operations

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

Time for parallel operations

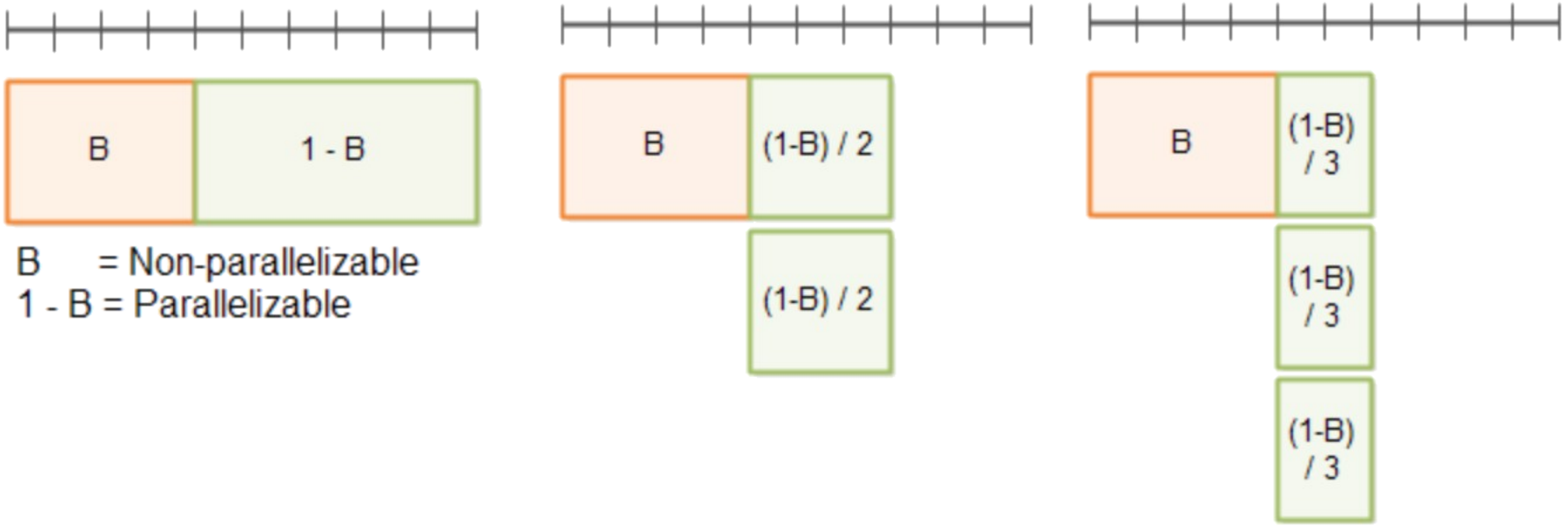
Efficiency $E = \frac{S}{p} = \frac{T_S}{pT_p}$

Number of processors



Challenge 1: Amdahl's Law

Not all the work can be decomposed



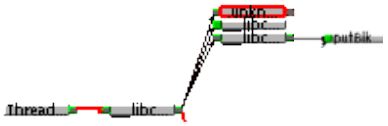
In parallel computing, Amdahl's law is mainly used
to predict the theoretical maximum speedup
for programs using multiple processors.

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

Challenge 2: Parallel overhead

Collaboration means communication and extra work

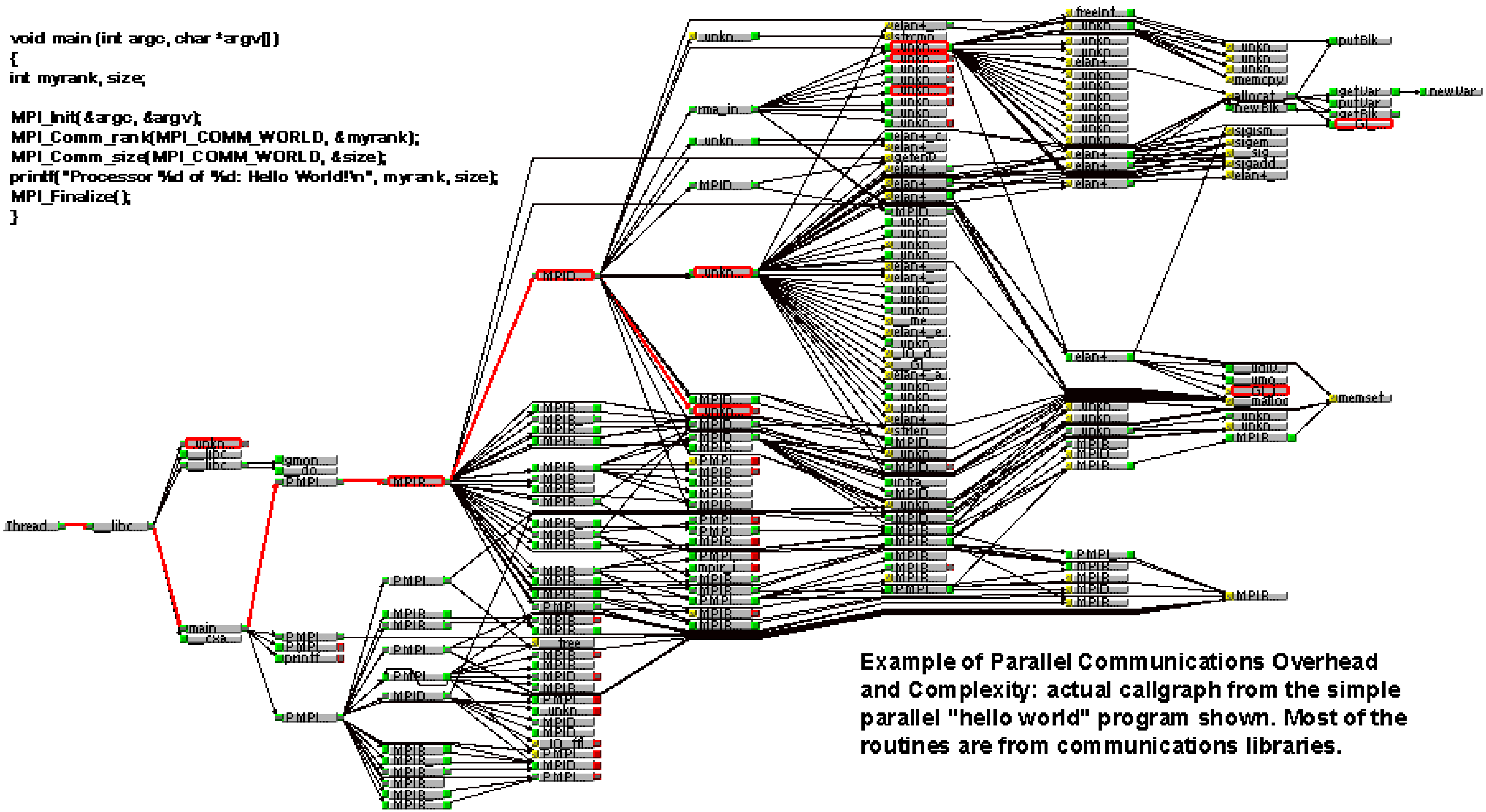
```
void main (int argc, char *argv[])
{



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



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

Challenge 3: Load imbalance

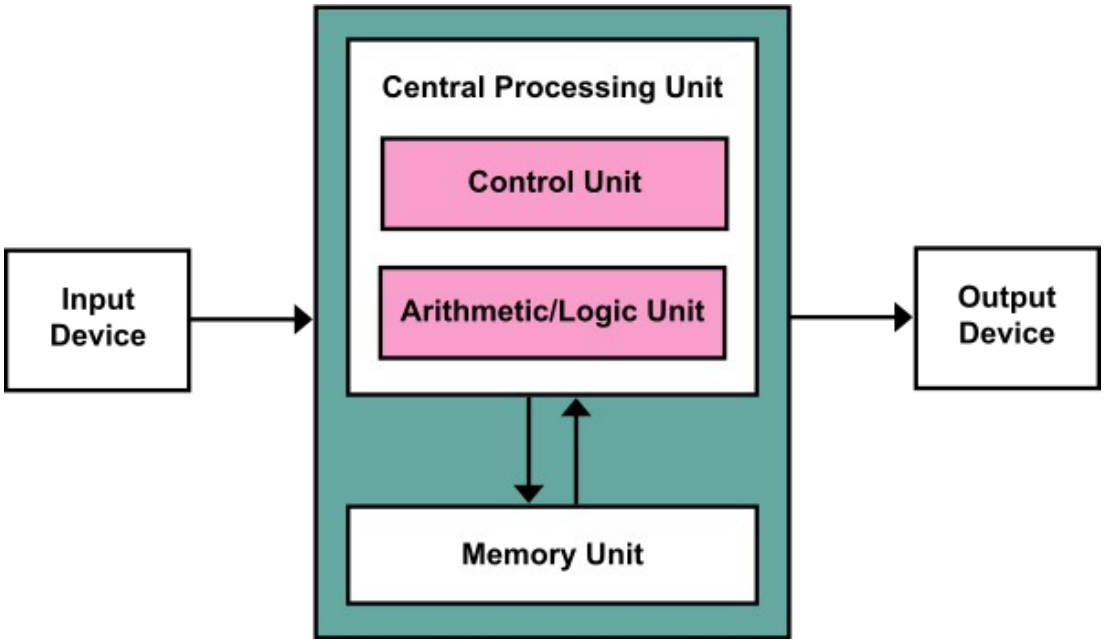
Parallelization is efficient only if every worker has the same amount of work

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

An abstract view of modern CPUs

- Instruction-level parallelism (**ILP**)
  - Instruction pipelining
  - Out-of-order execution
  - Speculative execution
  - …
- Simultaneous multithreading (**SMT**)
- Single Instruction Multiple Data (**SIMD**)

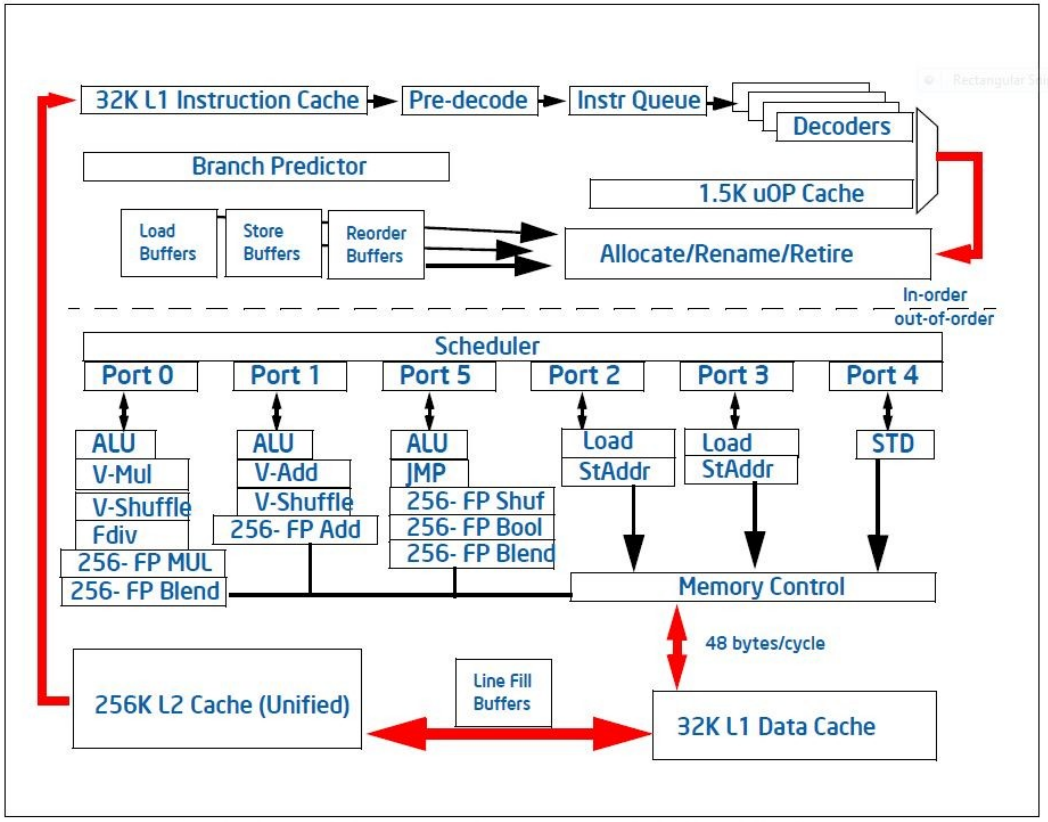
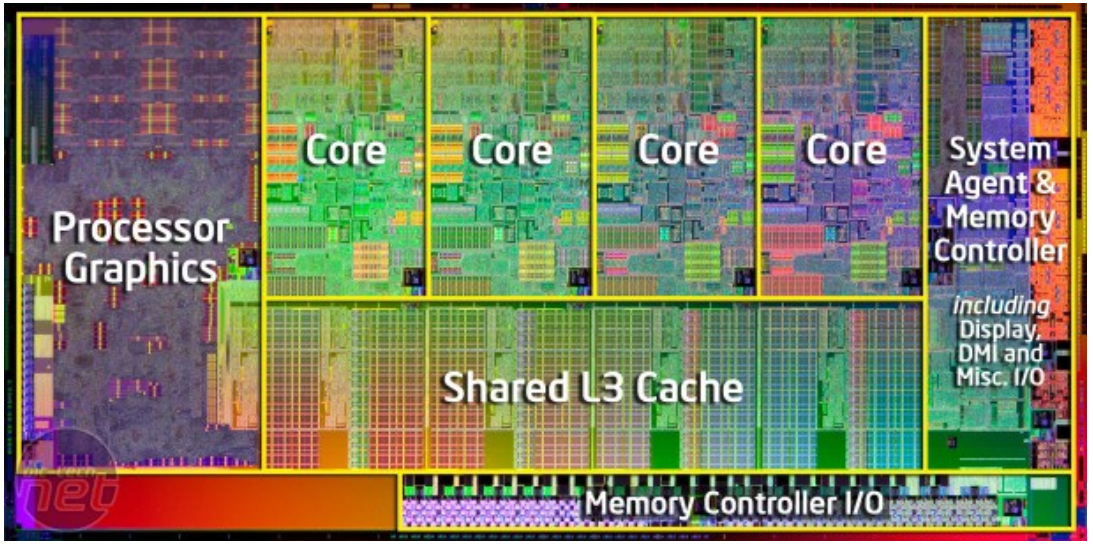


Figure 2-1. Intel microarchitecture code name Sandy Bridge Pipeline Functionality

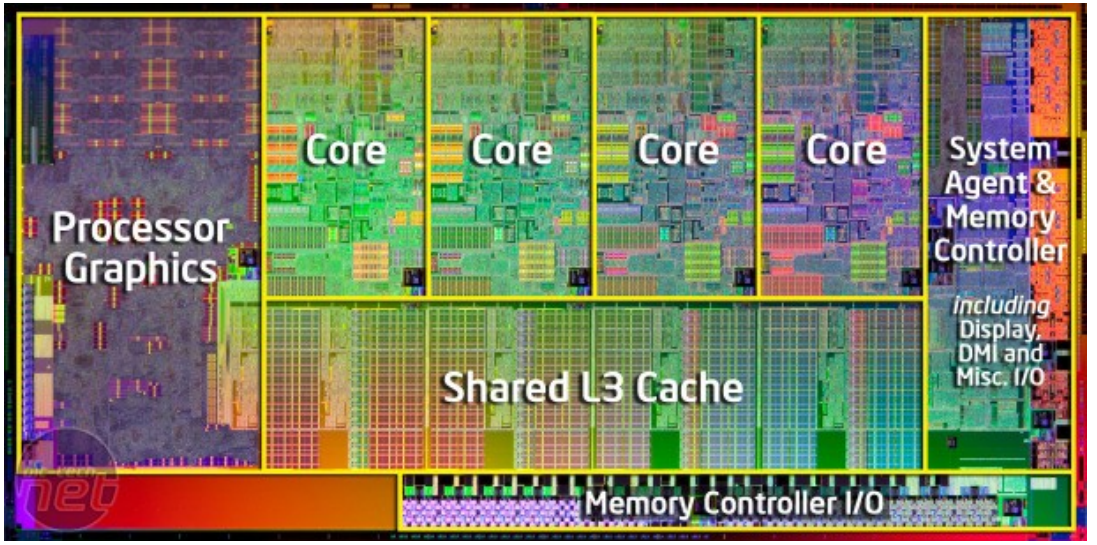
Parallelism at the chip (socket) level

- **Multicore** parallelism

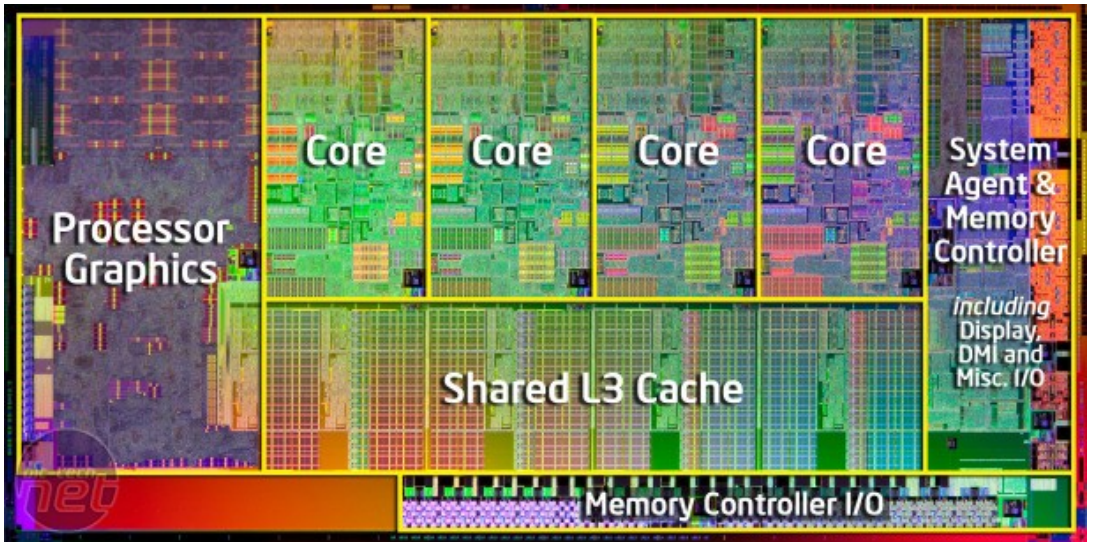


Parallelism at the computer level

- Multi-socket parallelism
  - SMP
  - NUMA



- Accelerators



Parallelism at the data center level

Multi-node parallelism

Parallelism at the data center level

Cluster computing

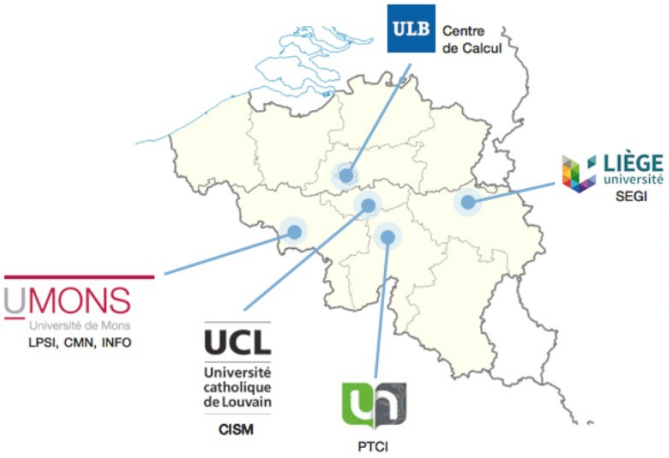
C.E.C.I

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

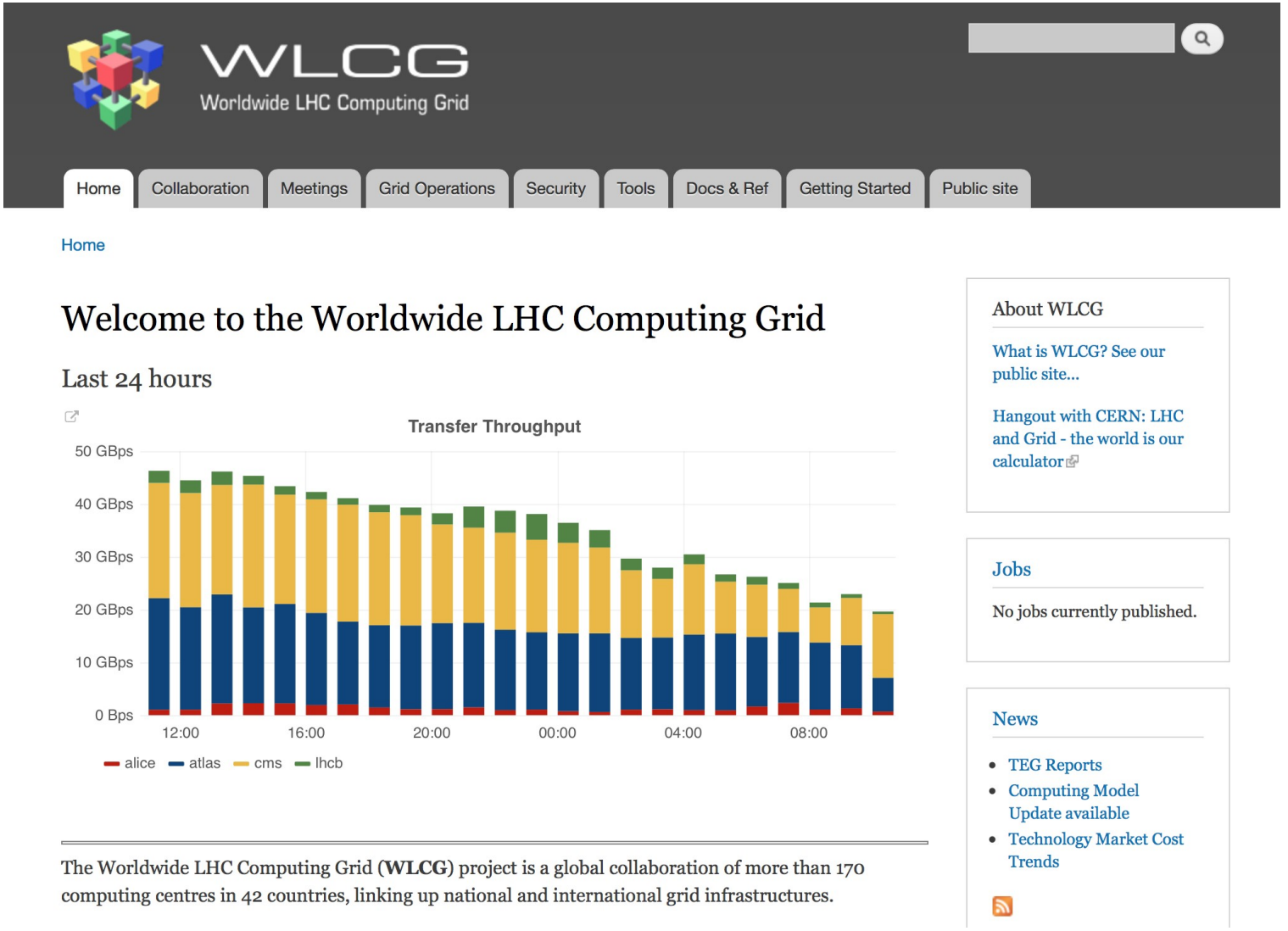
We are happy to announce the hire of a new CECI logisticien: Ariel Lozano. Welcome Ariel!

Parallelism at the data center level

Cloud computing “someone else’s cluster”

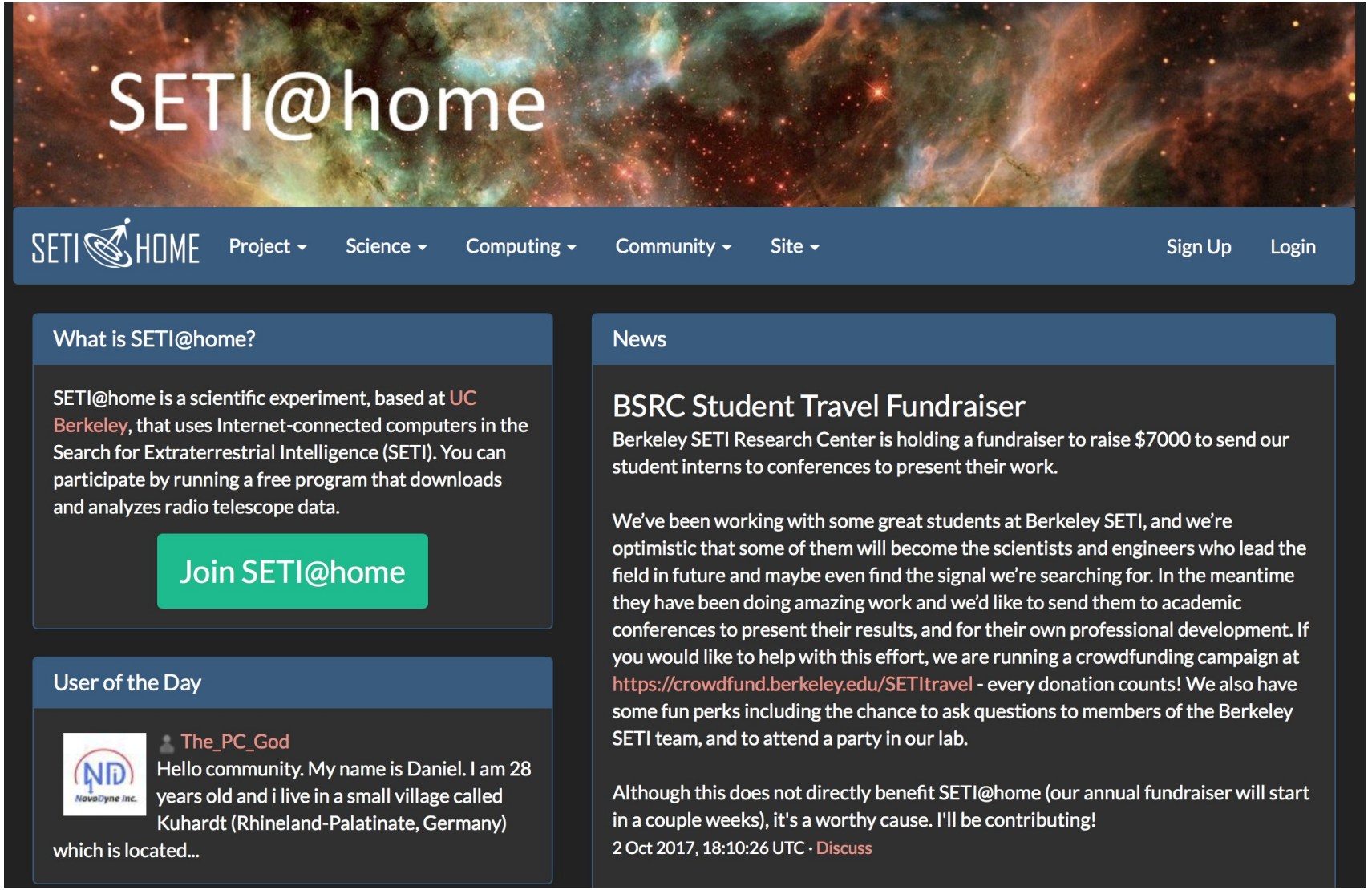
Parallelism at the world level

Grid computing – “cluster of clusters”



Parallelism at the world level

Distributed computing – “no unused cycle”



3.

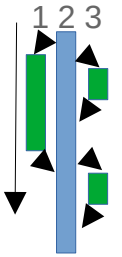
Programming paradigms
(logic of work sharing and organizing)
Programming models
(software libraries and APIs)

Is parallization automagic?

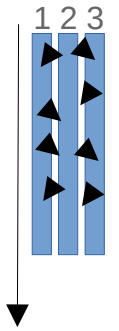
- **ILP**: yes, by the CPU and/or the compiler
- **SIMD**: mostly, by the compiler, but it can be fooled by your code (→ *pragmas*)
- **Intra-node**: can be if the library/software you use is designed for it (→ **_NUM_THREADS env vars, be aware of NUMA placement issues*)
- **GPUs**: can be if the library/software 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*).

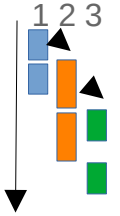


- **SPMD** (Single program multiple data)
  - A single program that contains both the logic for distributing work and computing
  - Multiple instances are started and “linked” together
  - Instances are identified with a distinct index



Other parallel programming paradigms

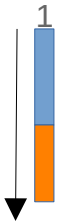
- **MPMD (Multiple program multiple data)**
- **Pipelining** : workers take care of a subtask in the processing chain and pass the intermediate result to the next worker



- **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

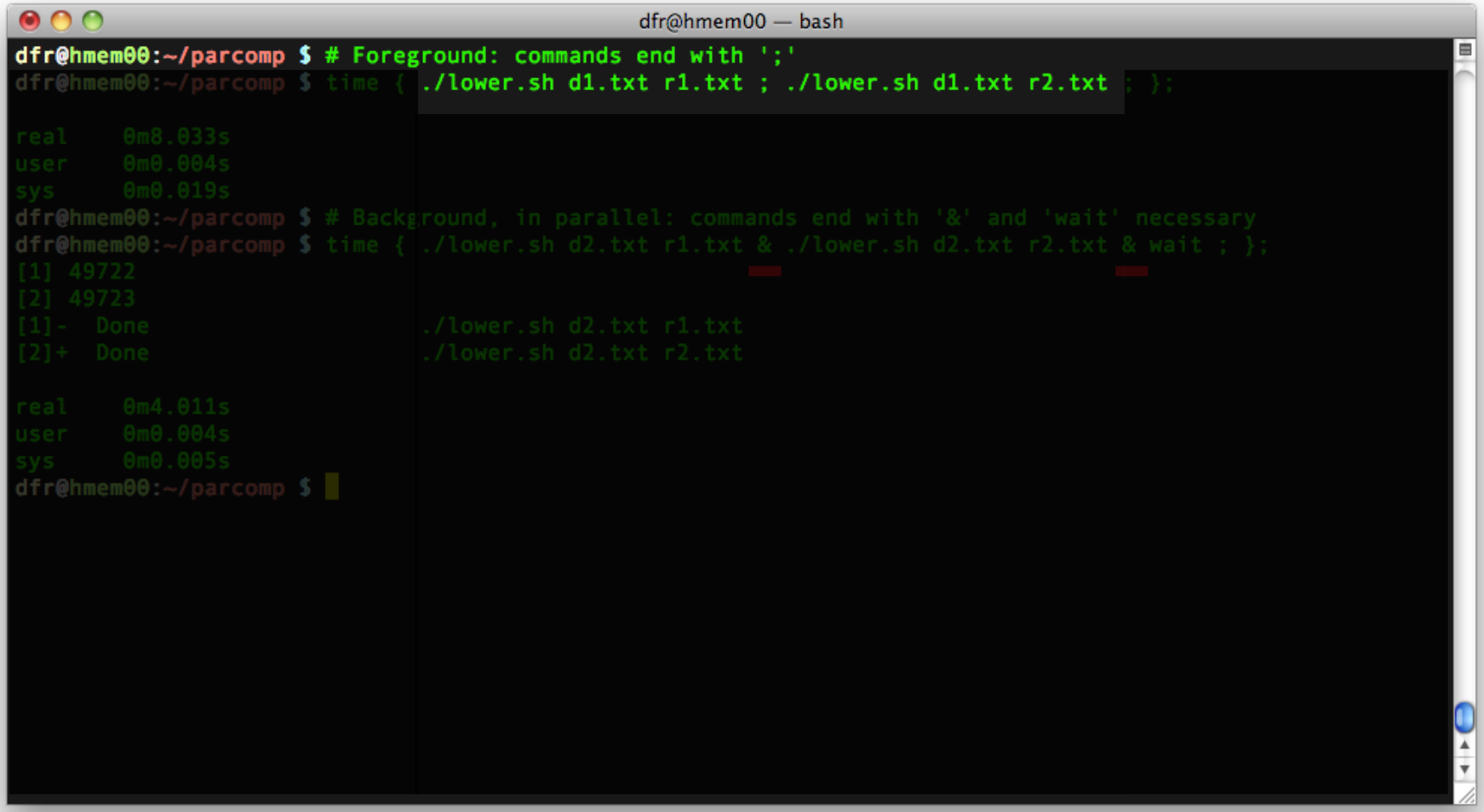
dfr@hmem00 — bash

```
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

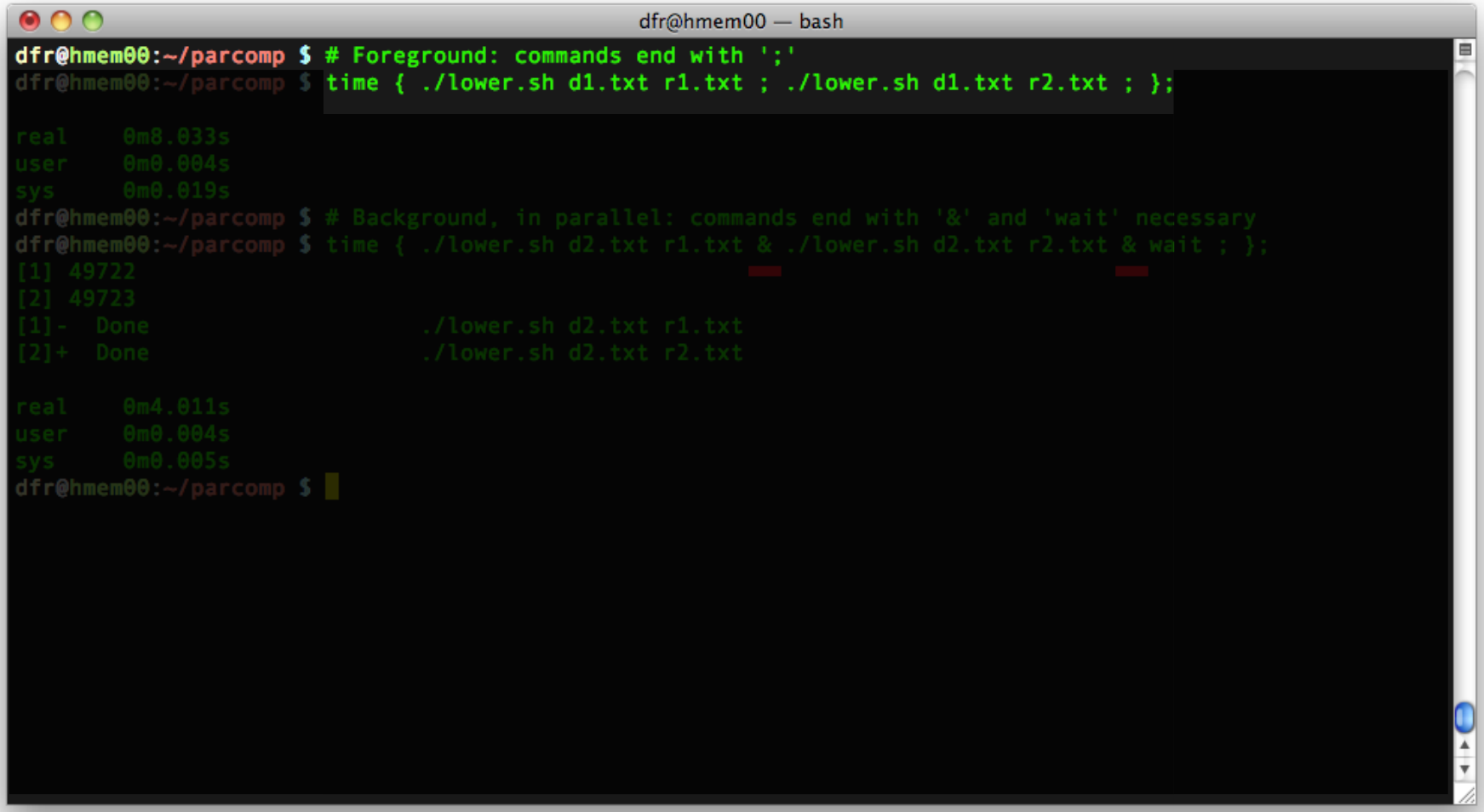
Run the program twice



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

4.2. Job control & Parallel processes in Bash

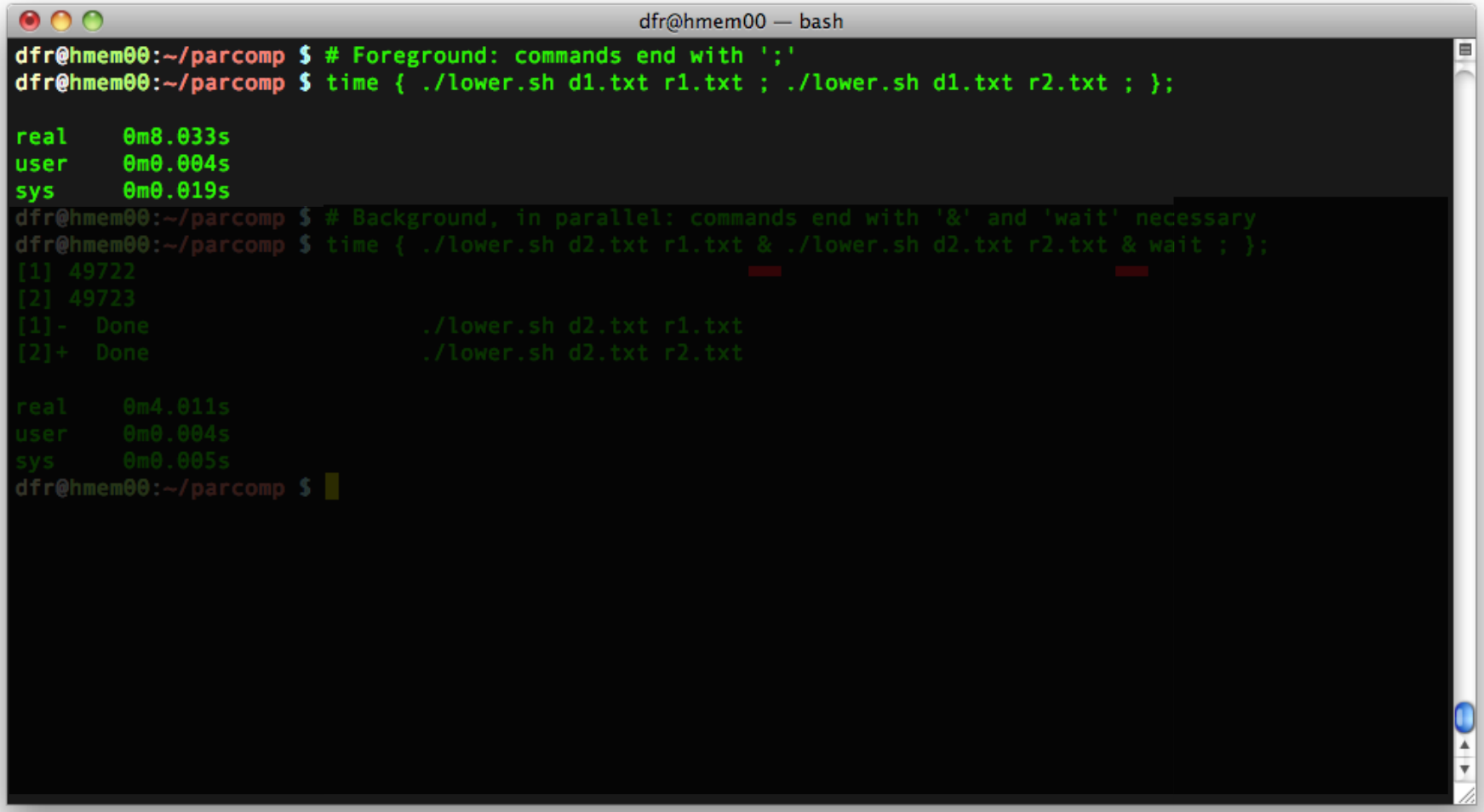
Run the program twice and measure the time it takes



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

4.2. Job control & Parallel processes in Bash

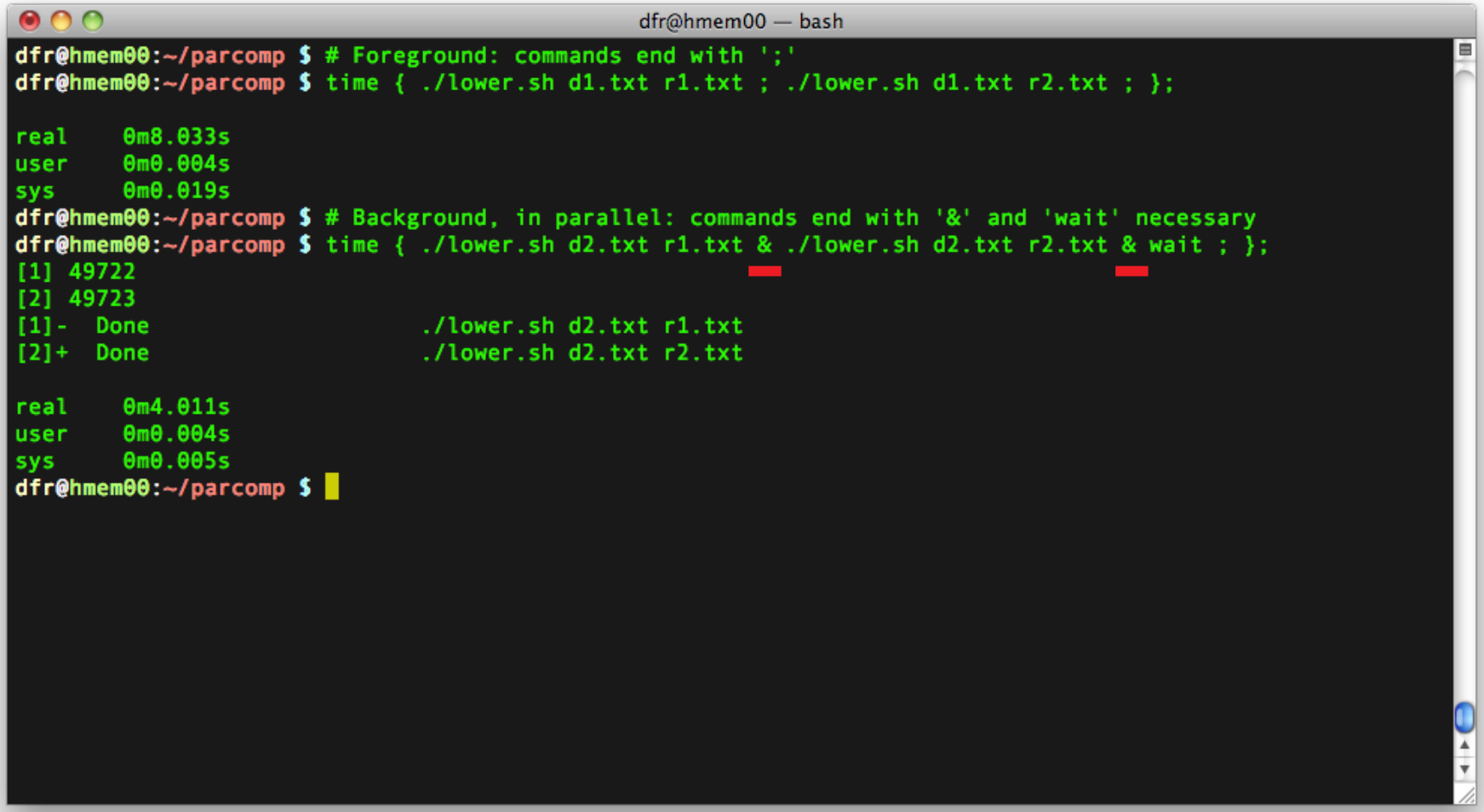
Run the program twice and measure the time it takes



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



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:

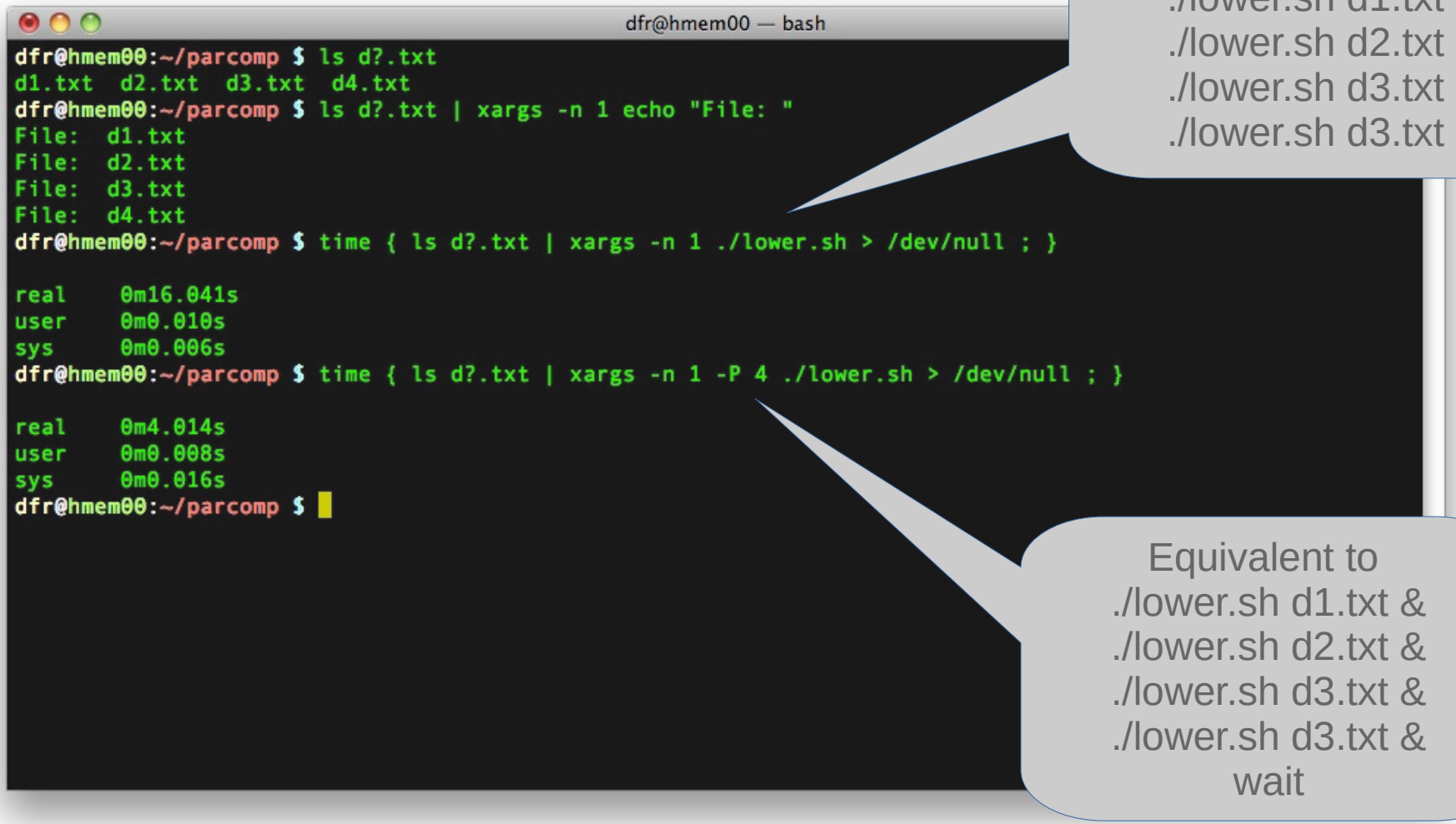
```
for i in {1..10}; do
  command1
  command2
done
```

```
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

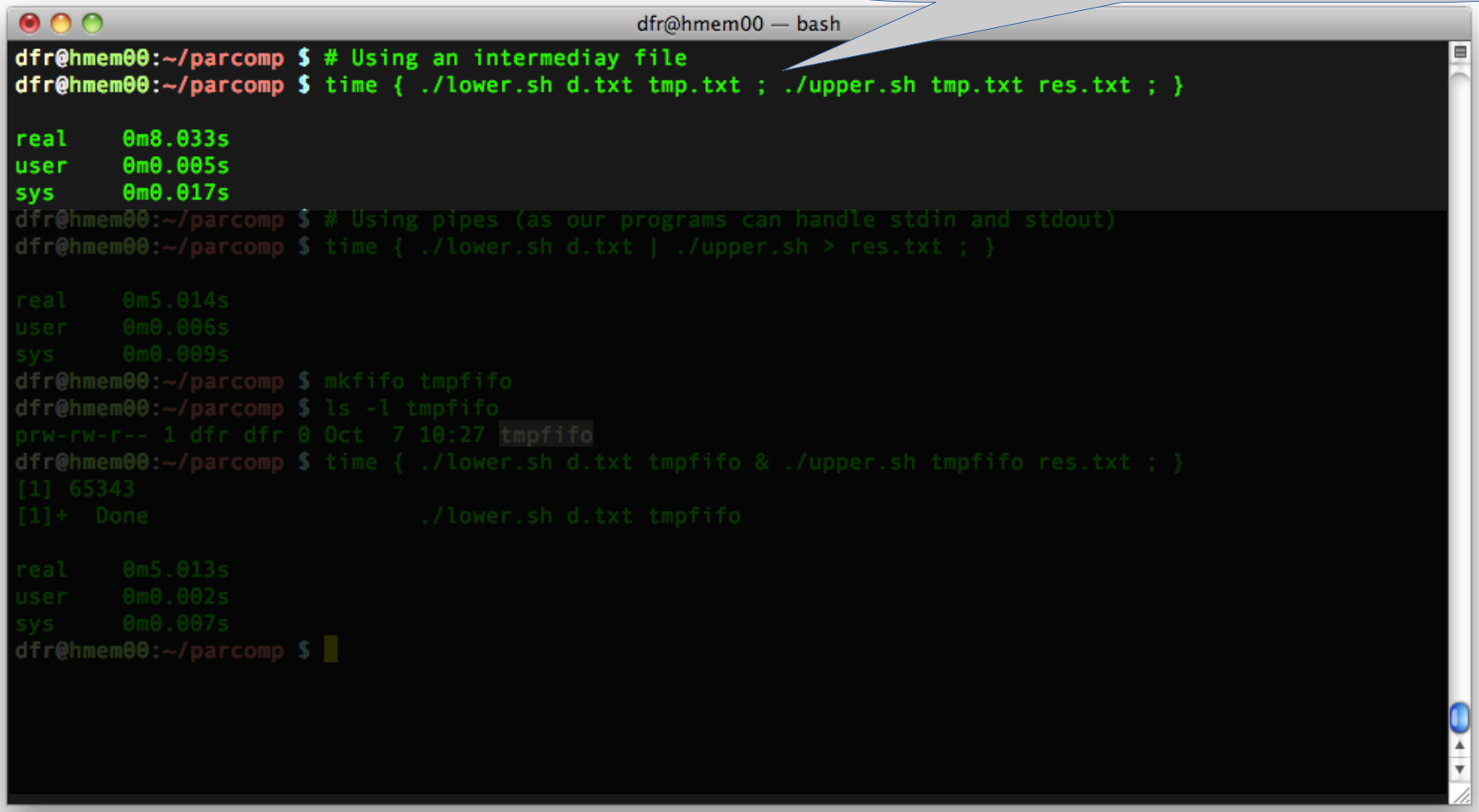
The xargs command distributes data from stdin to program



4.3.2. Several programs and one file

Using UNIX pipes for pipelining operations

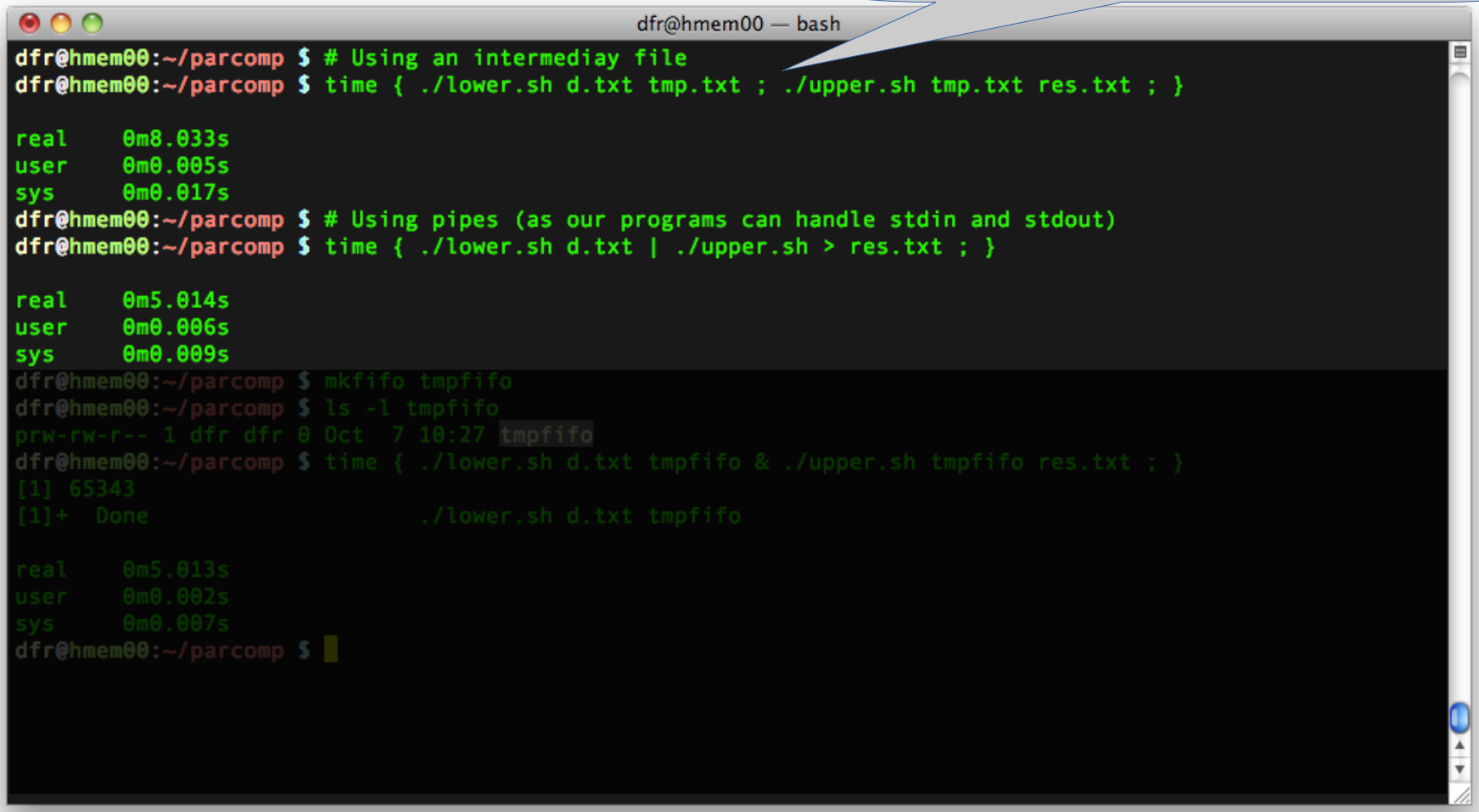
./upper.sh waits for ./lower.sh to finish
Note the intermediate file



4.3.2. Several programs and one file

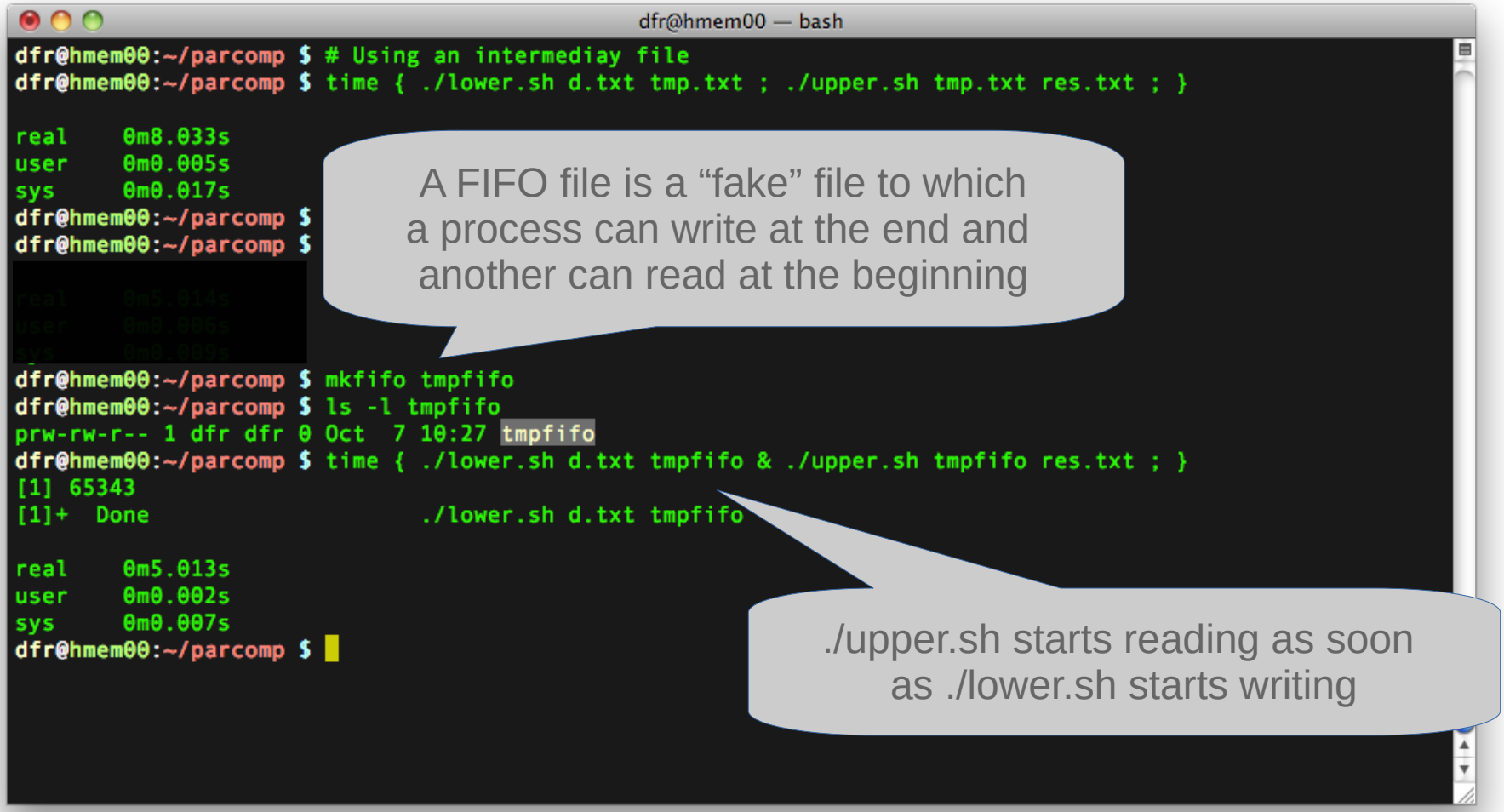
Using UNIX pipes for pipelining operations

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



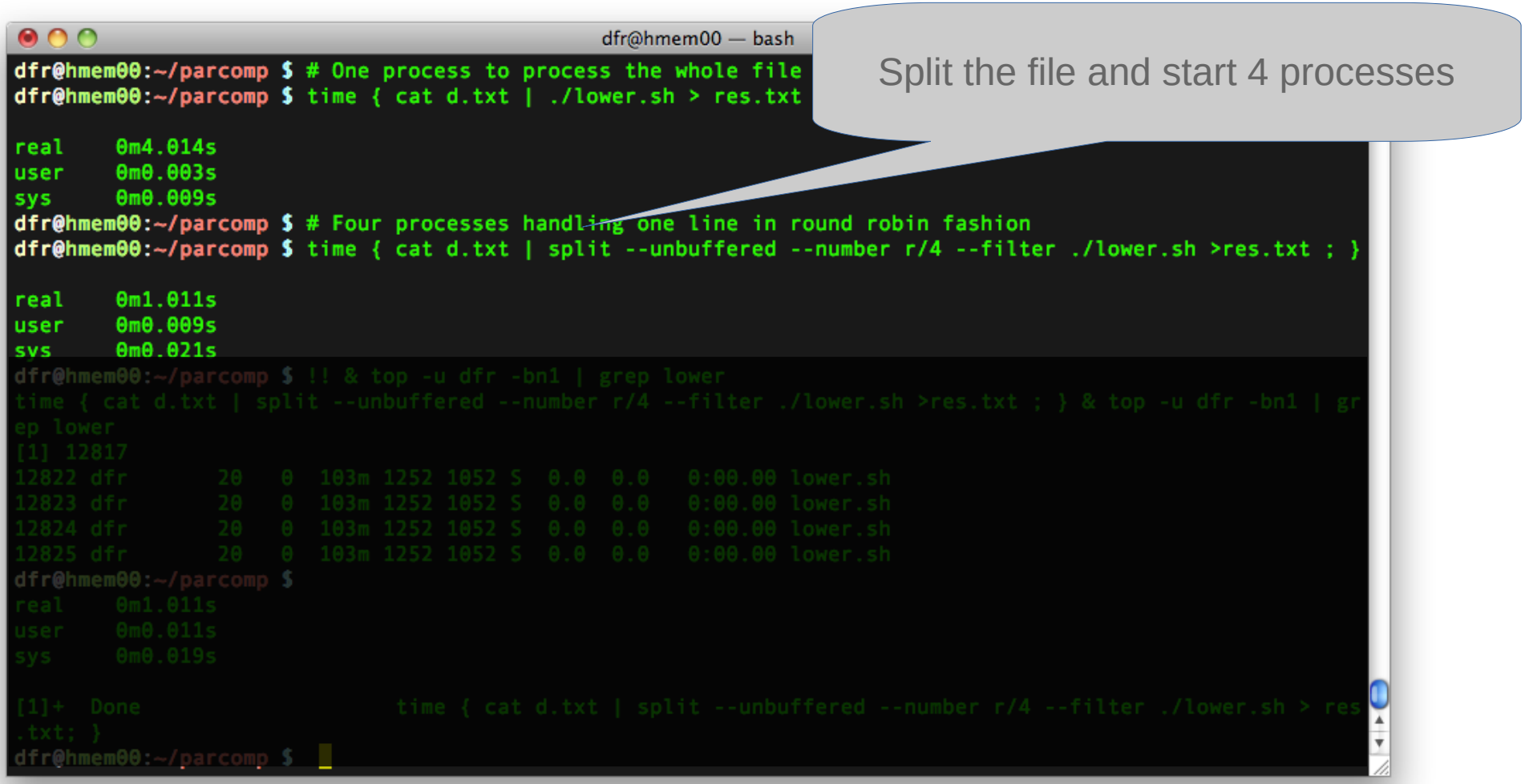
4.3.2. Several programs and one file

Using UNIX fifos for pipelining operations



4.3.3. One program and one large file

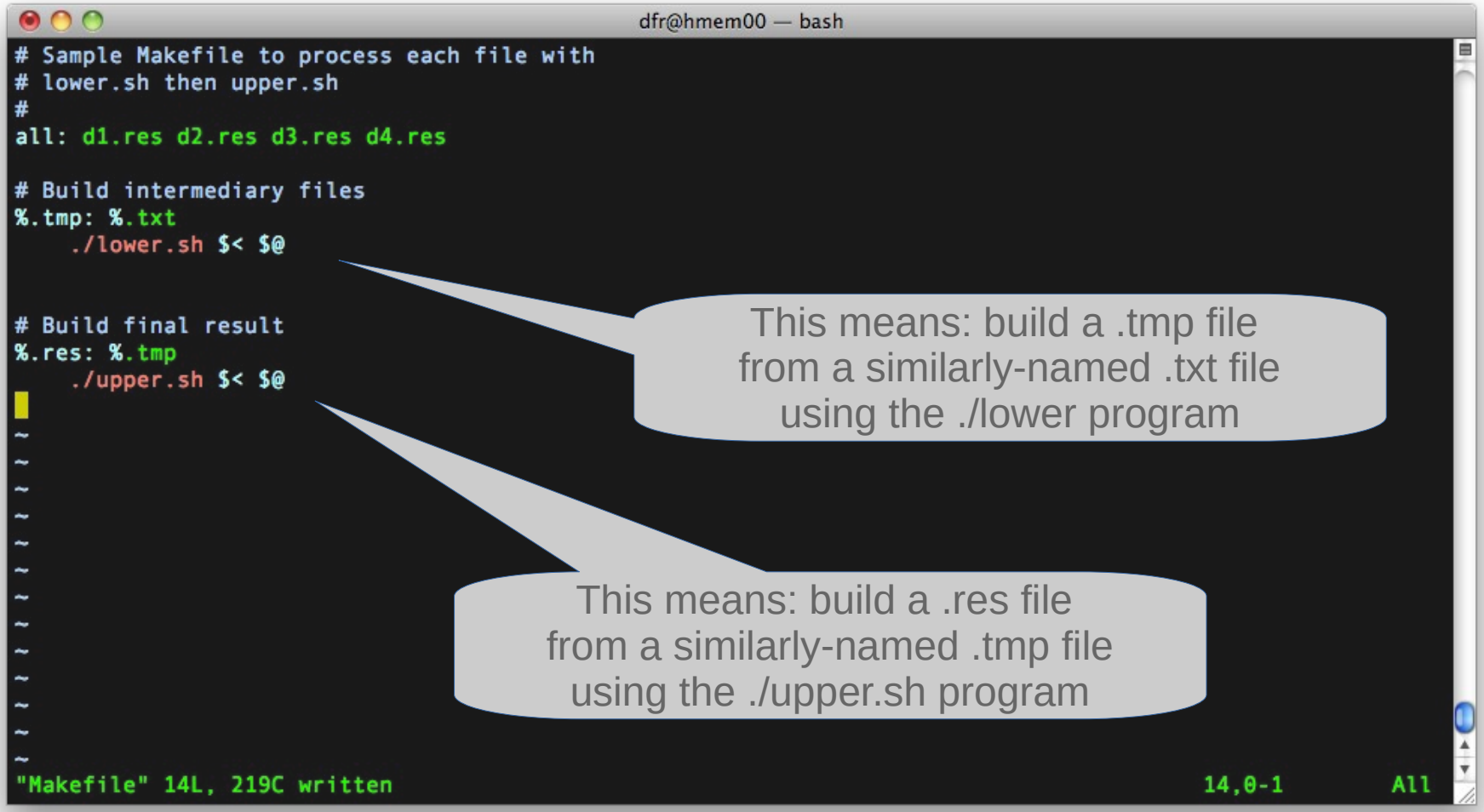
The split command distributes data from stdin to program



Need recent version of Coreutils/8.22-goolf-1.4.10

4.3.4. Several programs and many files

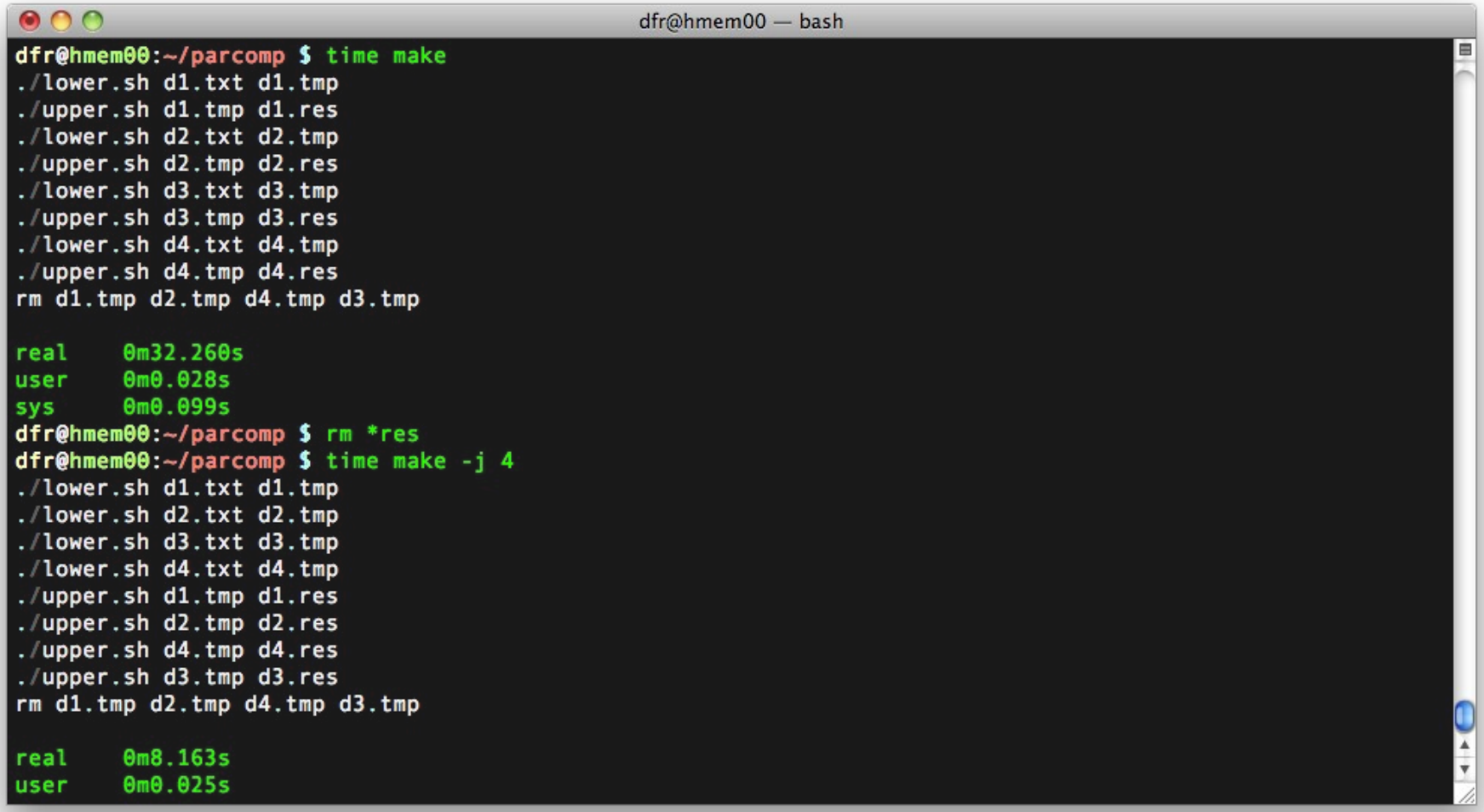
A Makefile describes dependencies and is executed with ‘make’



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



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.

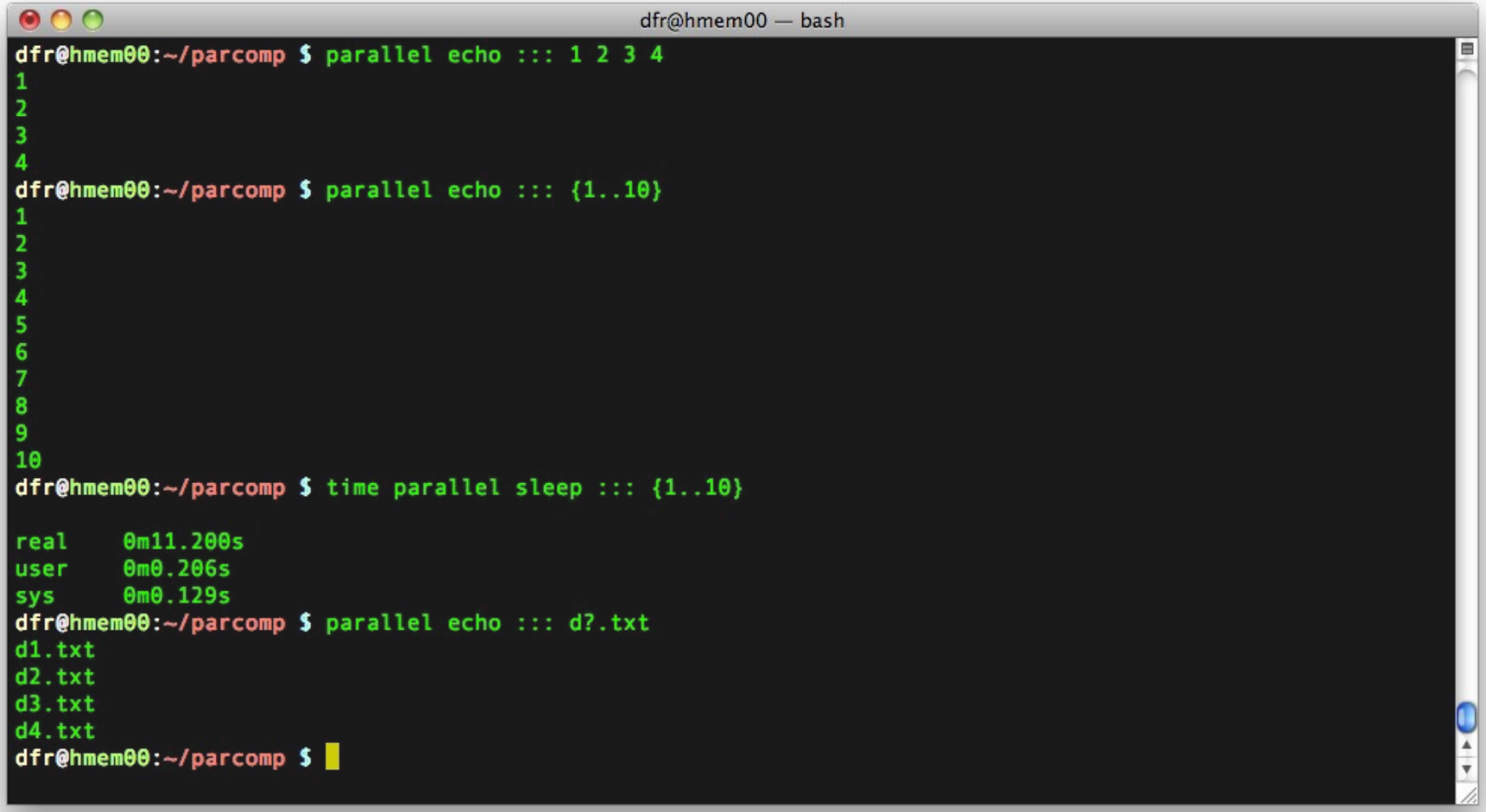
For people who live life in the parallel lane.

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 but you can install it

4.4. GNU Parallel

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



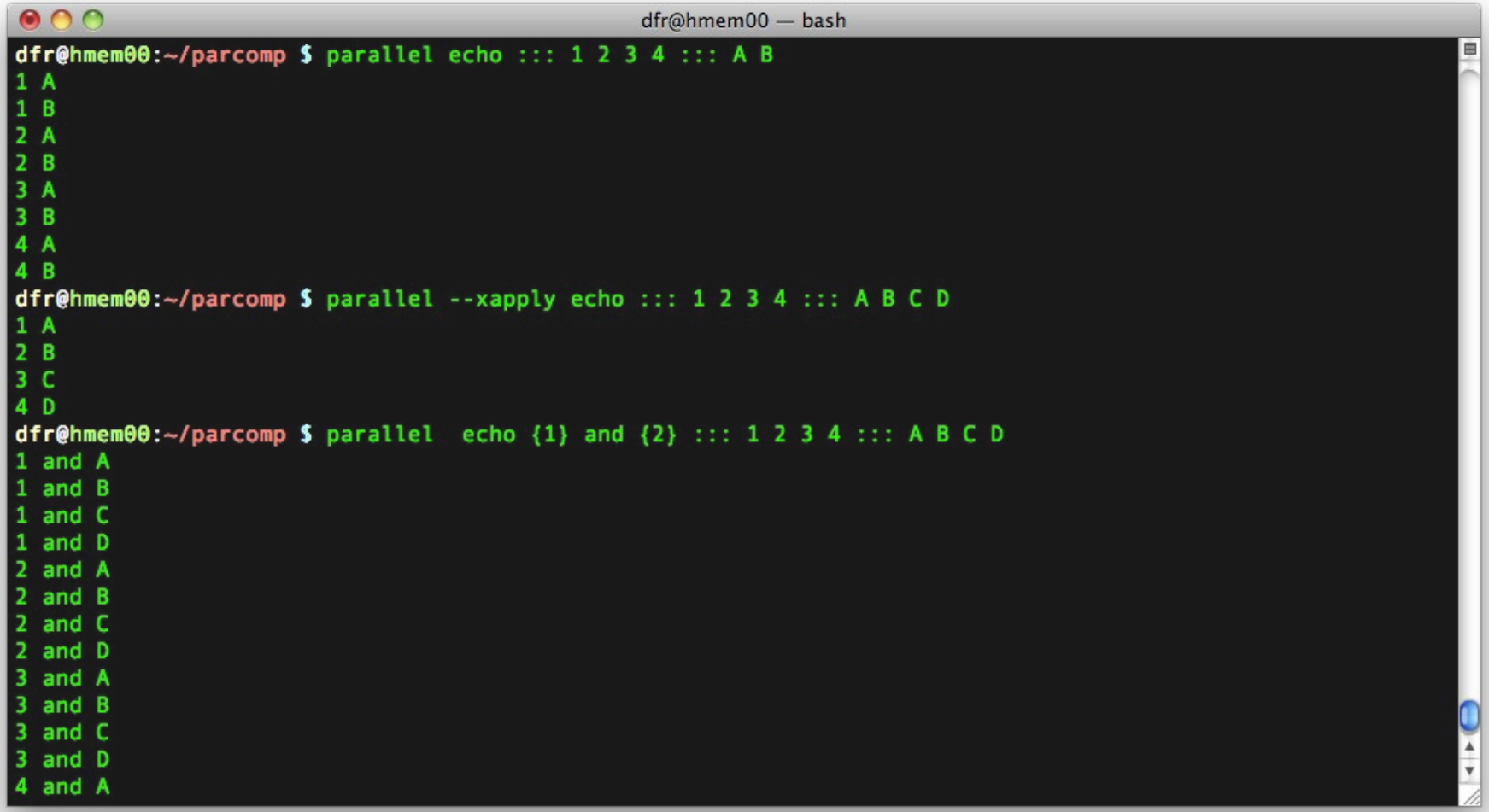
4.4. GNU Parallel

- 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 $
```

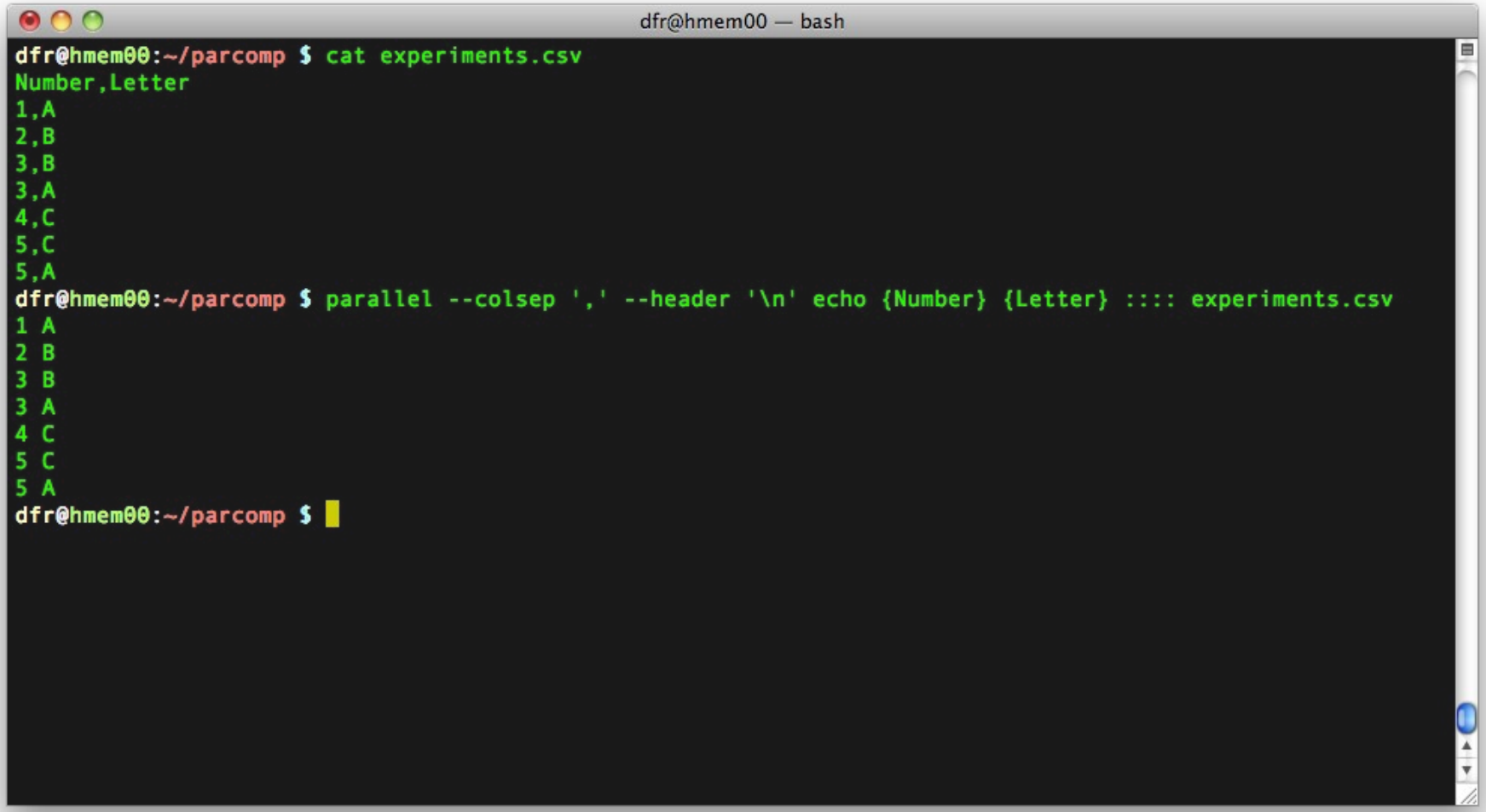
4.4. GNU Parallel

- Multiple parameters and --xapply



4.4. GNU Parallel

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



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

4.4. GNU Parallel

- Split a file with --pipe

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
```

- Multiple programs and one file
  (not suitable for that)
- 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