Communication cost model and collective communication operations

Laura Grigori

EPFL and PSI

September 17, 2024

EPFL

PAUL SCHERRER INSTITUT PSI

Plan

Abstract model of a parallel machine

MPI collective communication
- Main MPI collectives
- Example of implementation in MPI
- Cost of collectives on P procs

Introduction to using the EPFL cluster
- Clusters
- Remote access and file transfer
- Submitting a job

Abstract model of a parallel machine

- Consider a simple model of a parallel machine formed by a collection of homogeneous processors connected through a fast network
- γ: time to compute one flop
- α: interprocessor latency, time to send one word between two processors
- β: inverse of the interprocessor bandwidth

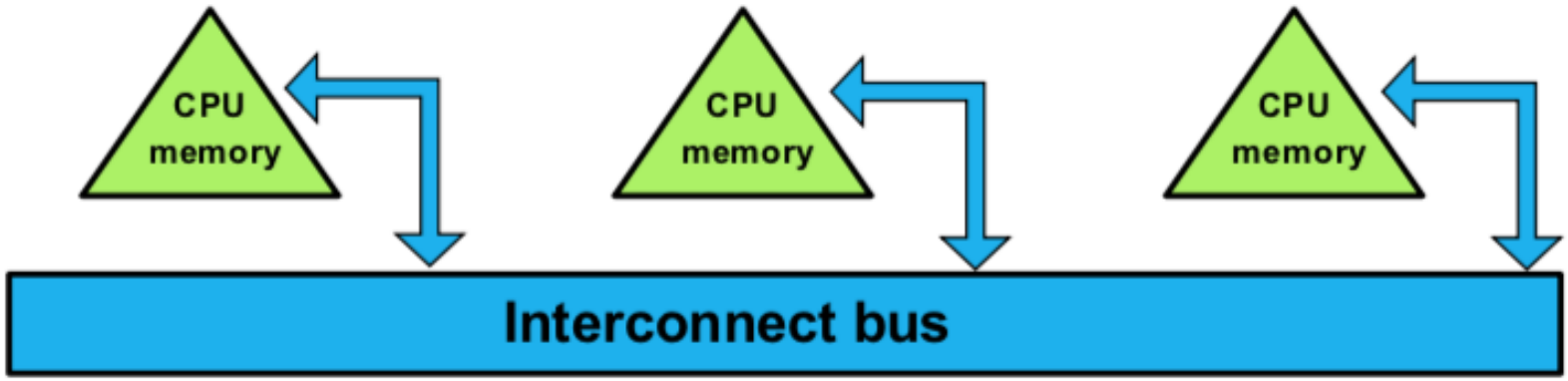


Figure: Abstract model of a parallel machine

Simplifying assumptions of the abstract model

- Time required to compute one flop per processor is constant
  - Model ignores the memory hierarchy of each processor
- Communication cost is independent of:
  - topology of the interconnect network
  - physical distance between processors
  - ignores network contention
- At a given time, a processor can send and can receive a message
- Any subset of disjoint pair of processors can communicate simultaneously and the links in the network are assumed to be bidirectional
  - Communication cost of exchanging a message of n words between a pair of processors is estimated as

$$\alpha + n\beta$$

Abstract model of a parallel machine

- Time of a parallel algorithm estimated with $\alpha - \beta - \gamma$ model:

$$T = \gamma \cdot \#\ \text{flops} + \beta \cdot \#\ \text{words} + \alpha \cdot \#\ \text{messages}, \tag{1}$$

where

- $\#flops$: computation on the critical path of the algorithm,
- $\#words$ volume of communication,
- $\#messages$ number of messages exchanged on the critical path of the parallel algorithm

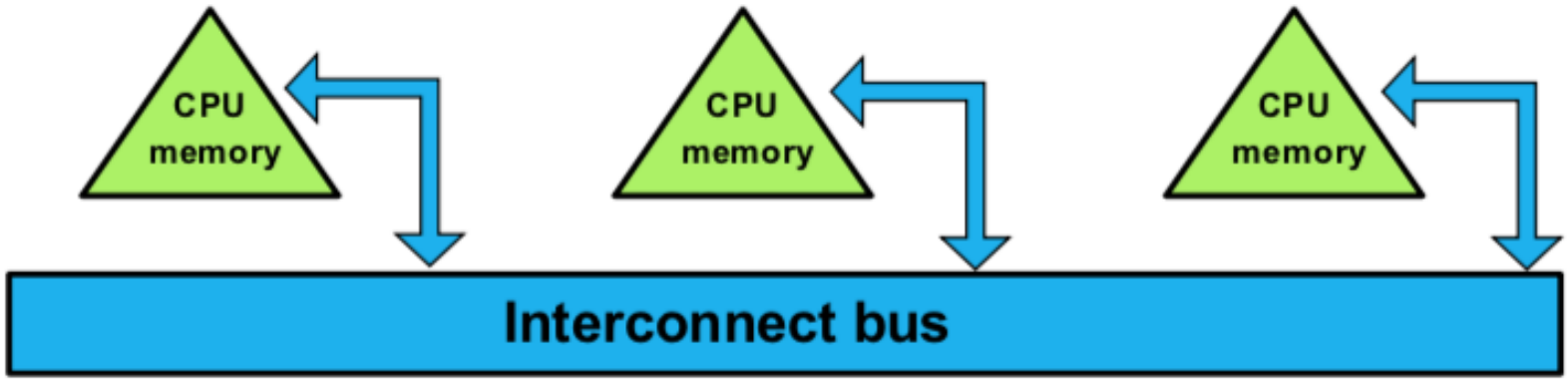


Figure: abstract model of a parallel machine

Plan

Abstract model of a parallel machine

MPI collective communication
- Main MPI collectives
- Example of implementation in MPI
- Cost of collectives on P procs

Introduction to using the EPFL cluster
- Clusters
- Remote access and file transfer
- Submitting a job

MPI collective communication

- Many routines available: we will discuss the most used ones
- Collectives involve all processors in a specified communicator
- The default communicator when program starts is MPI_COMM_WORLD
- Some routines specify a root processor:
  Broadcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv
- Other routines (All versions) deliver results to all participating processes:
  Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv
- V versions allow the chunks to have variable sizes.
- Allreduce, Reduce, Reduce_scatter, and Scan take both built-in and user-defined combiner functions

Routines that specify a root processor

More details about 4 routines:

- broadcast, scatter and gather, reduce

Broadcast

A root broadcasts n words to all P processors

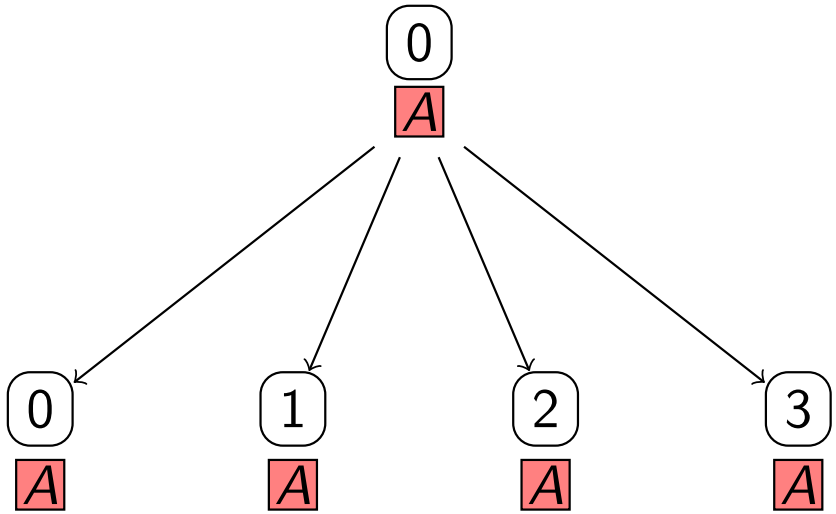


Figure: Broadcast

Scatter and Gather

A root scatters n words, each processor receives n/P words

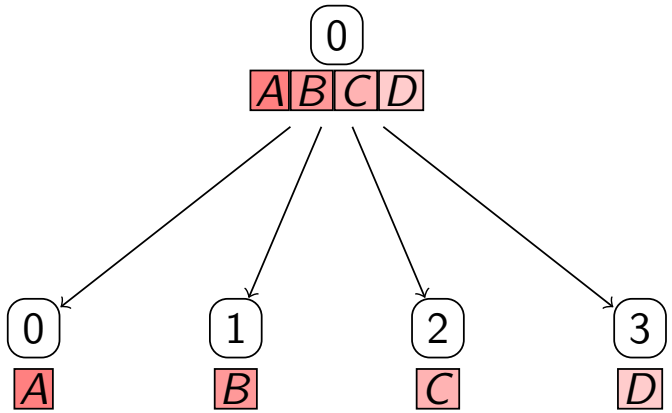


Figure: Scatter

Each processor sends n/P words, which are gathered on root

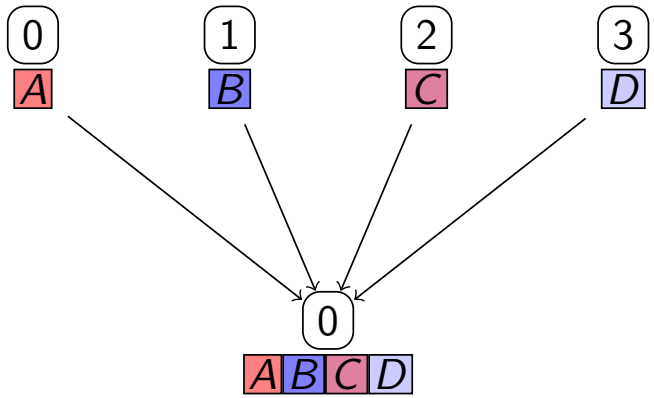


Figure: Gather

Reduce

Reduction on n words from each processor, result returned on root

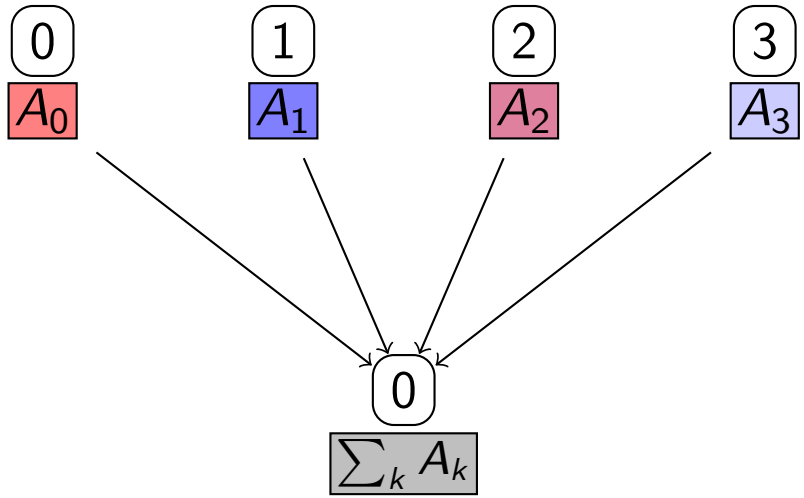


Figure: Reduce

Routines that do not specify a root processor

More details about 4 routines that return results on all processors:

- Reduce_scatter, Allgather, Allreduce, Alltoall

Reduce_scatter

Reduction on n words from each processor, result scattered on all processors

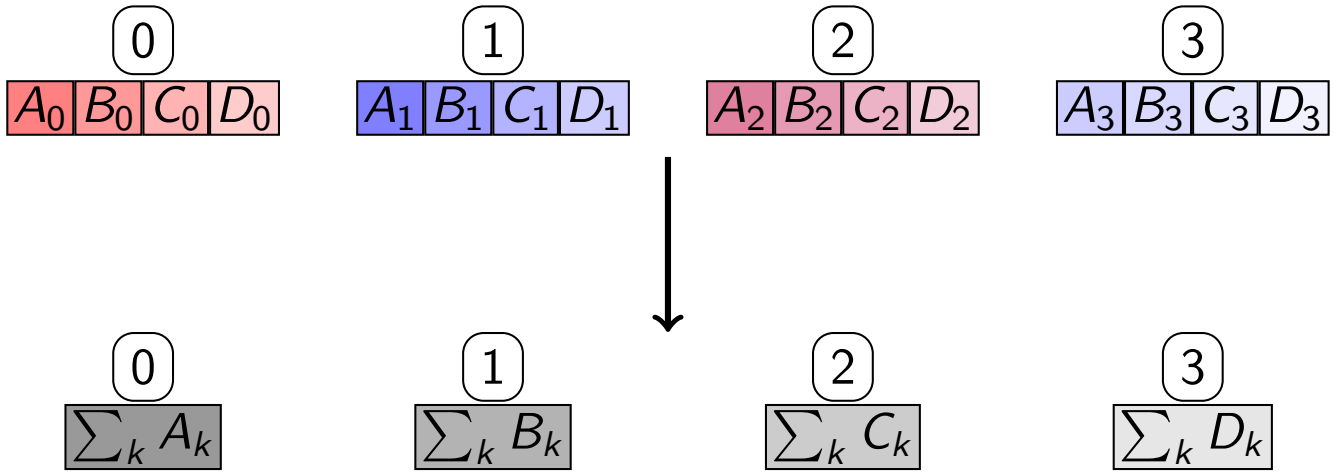


Figure: Reduce_scatter

Allgather

Each processor sends n/P words, which are gathered on all processors

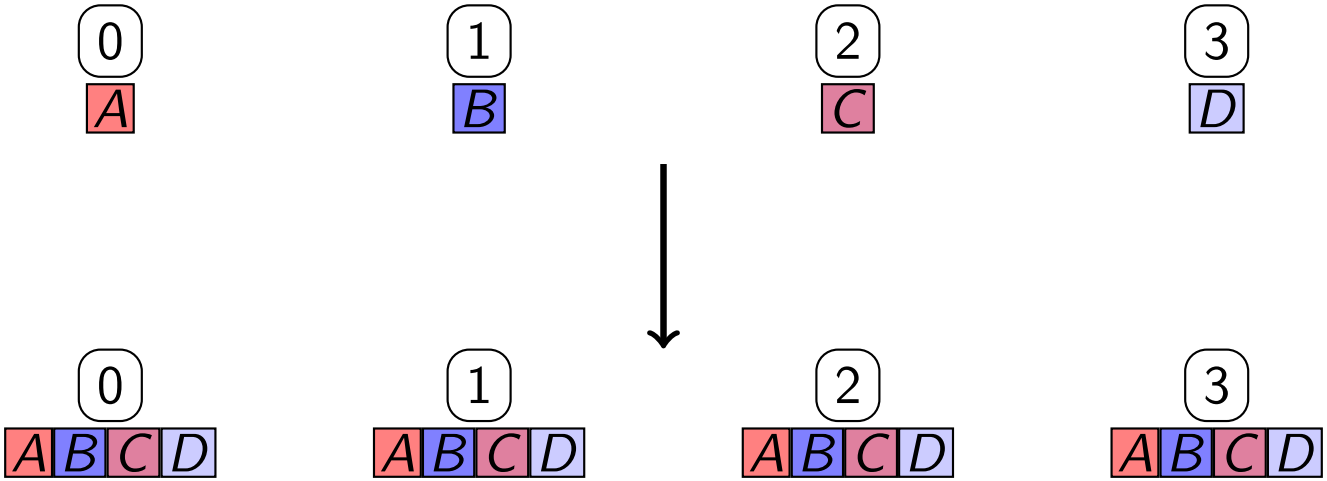


Figure: Allgather

Allreduce

Reduction on n words from each processor, result returned on all processors

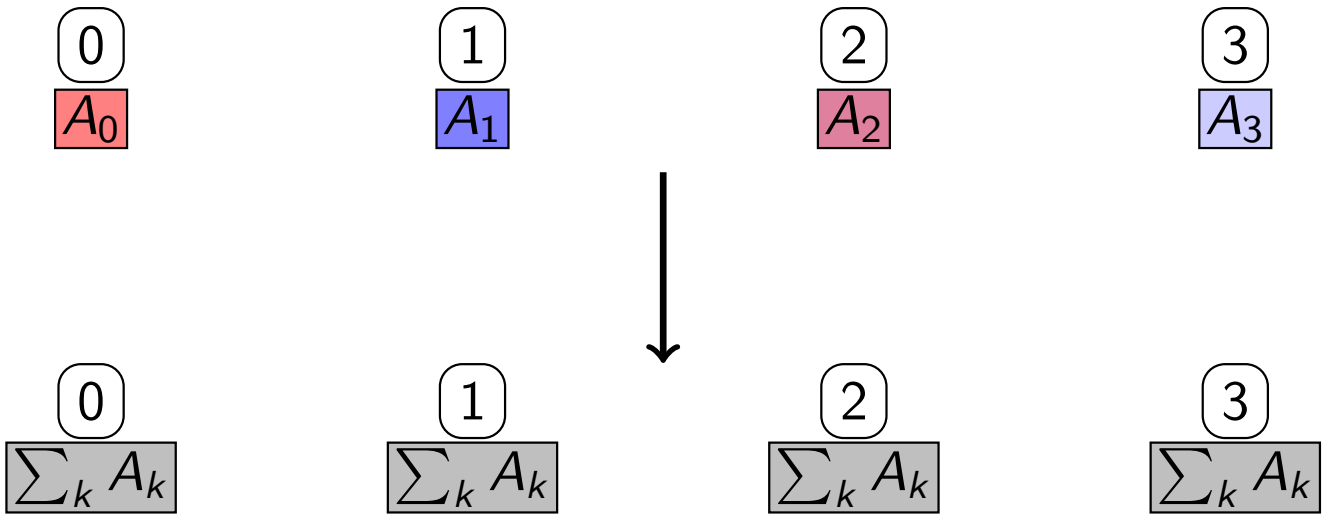


Figure: Allreduce

Alltoall

Each processor sends different n/P words to every other processor

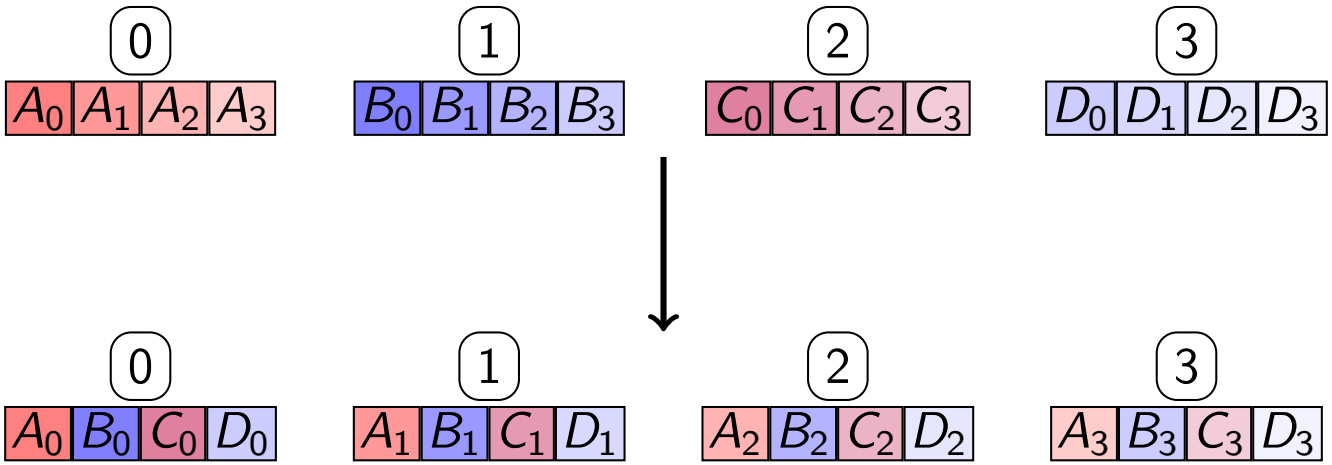


Figure: Alltoall

Call of collective routines

- All collective operations must be called by all processes in the communicator
- For example, MPI_Broadcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
- "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive
- In our examples, Processor 0 is the root

MPI Built-in Collective Computation Operations

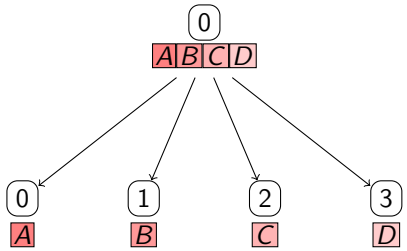
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Binary and
MPI_BOR	Binary or
MPI_BXOR	Binary exclusive or
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Scatter: implementation in MPI

A root scatters n words, each processor receives n/P words

Recursive halving:

First step, root 0 sends 2nd half of data to $\frac{P}{2}$;
continue recursively with 0 and $\frac{P}{2}$ as new roots.



$$\alpha \cdot \log_2 P + \beta \cdot n\frac{P-1}{P} \approx \alpha \cdot \log_2 P + \beta \cdot n$$

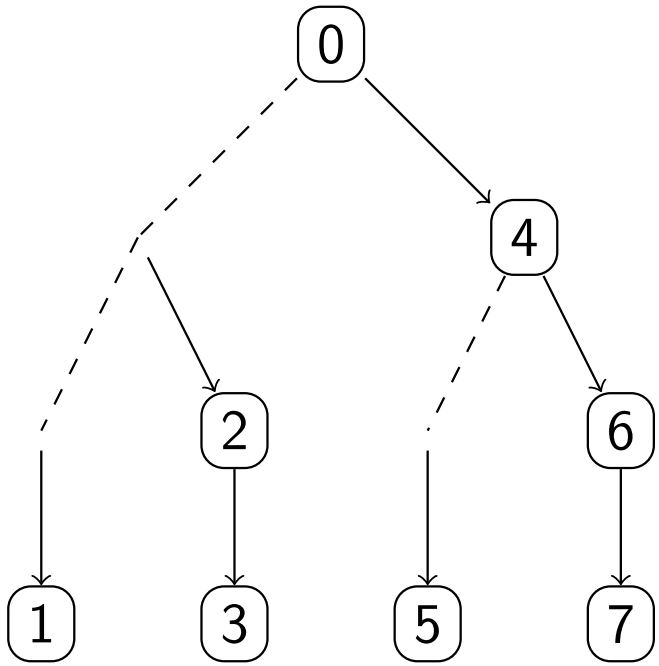
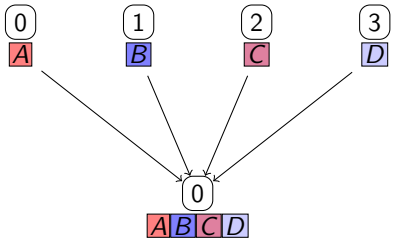


Figure: Binomial tree communication scheme

Gather: implementation in MPI

Each processor sends n/P words, which are gathered on root

Recursive doubling (oposite of scatter):
First step, even procs receive n/P data from odd procs;
continue recursively on even procs.



$$\alpha \cdot \log_2 P + \beta \cdot n$$

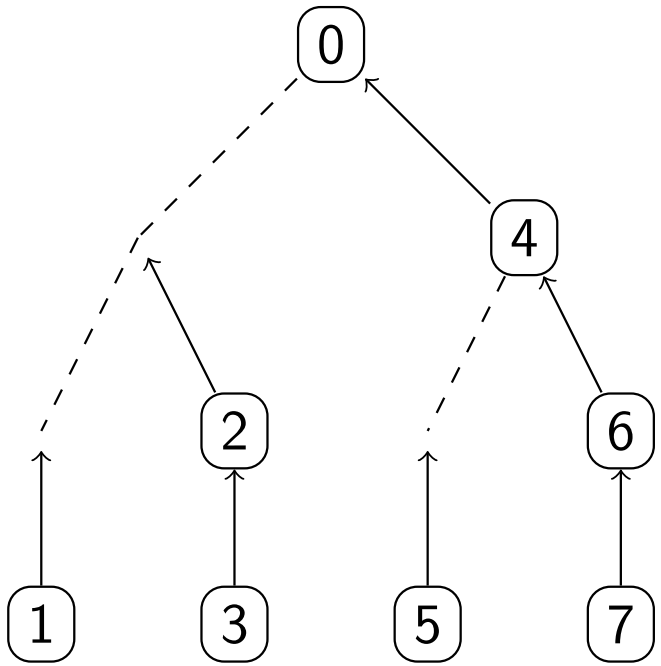


Figure: Binomial tree communication scheme

Allgather: implementation in MPI

Each processor sends n/P words, which are gathered on all processors

Recursive doubling algorithm with butterfly scheme:

- At time t, process i exchanges (sends/receives) all its current data (its original data plus anything received until then) with process $i + 2^t$. In the first step, i is even, and the pattern continues accordingly in subsequent steps.
- Exchange $\frac{n}{P}, \frac{2n}{P}, \ldots$ up to $2^{\log_2 P - 1}\frac{n}{P}$ data in the last step.

$$\alpha \cdot \log_2 P + \beta \cdot n\frac{P-1}{P} \approx \alpha \cdot \log_2 P + \beta \cdot n$$

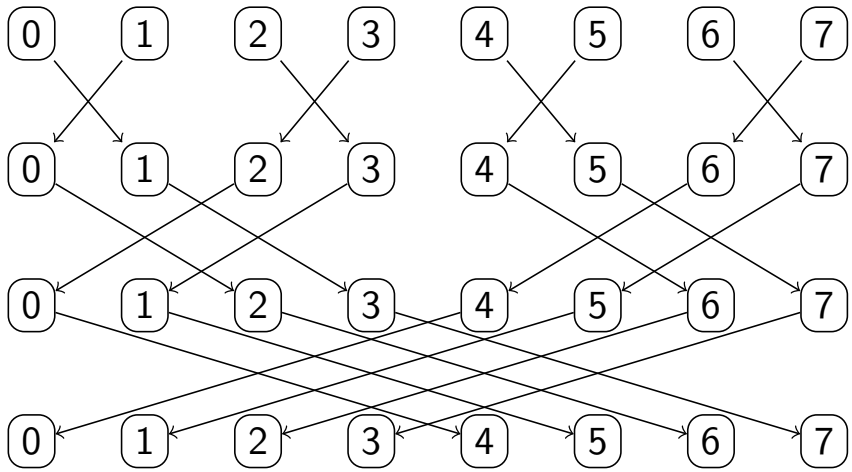


Figure: Butterfly communication scheme

Reduce_scatter: implementation in MPI

Reduction on n words from each processor, result scattered on all processors

- First step: each proc i s.t. $0 \leq i < P/2$ exchanges $n/2$ words with processor $i + P/2$
- All processors compute the reduction operation between the $n/2$ words they owned originally and the received data
- Proceed recursively on each half of processors simultaneously

$$\alpha \cdot \log_2 P + \beta \cdot n + \gamma \cdot n$$

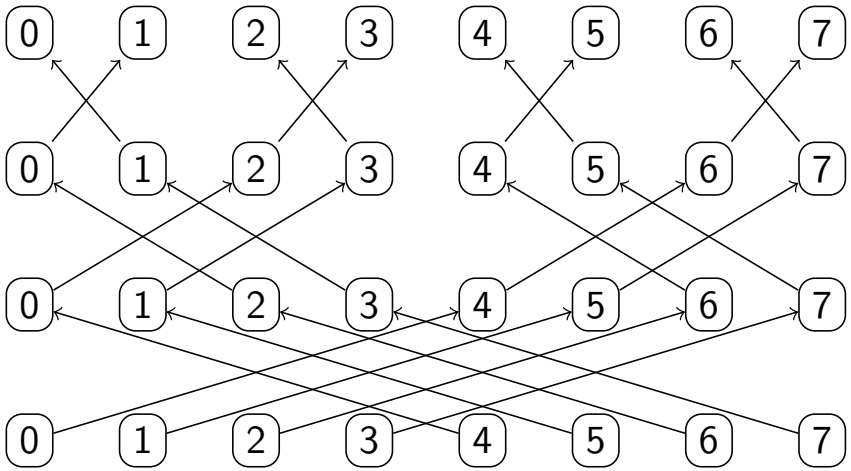


Figure: Butterfly communication scheme

Implementation of collectives in MPI: Broadcast

Broadcast based on Scatter/Allgather

- Scatter the n words among processors using binomial tree
  - Shrink message size from n to n/P
- Allgather n/P words among processors using recursive doubling and butterfly communication scheme
  - Grow message size from n/P to n

$$\alpha \cdot 2 \log_2 P + \beta \cdot 2n$$

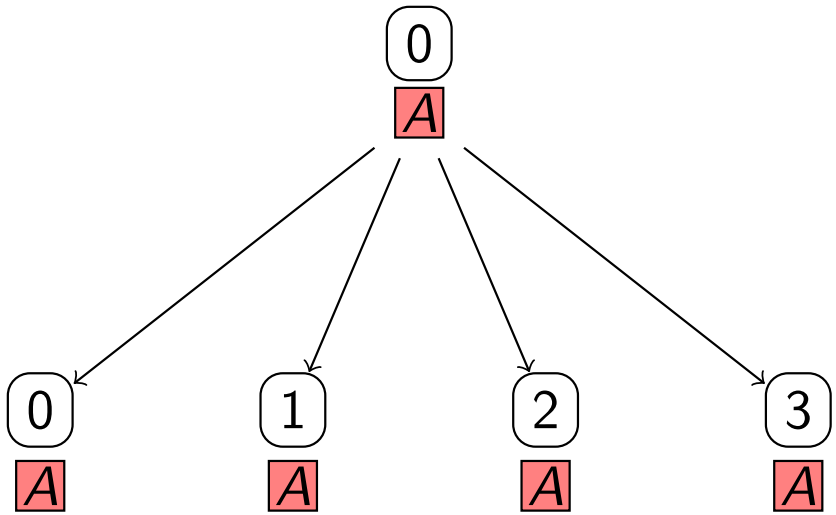


Figure: Broadcast

Other collectives in MPI

Reduce: reduction on n words from each processor, result returned on root
Reduce_scatter followed by a Gather

$$\alpha \cdot 2 \log_2 P + \beta \cdot 2n + \gamma \cdot n$$

Allreduce: reduction on n words from each processor, result returned on all processors
Reduce scatter followed by an Allgather (both implemented with the butterfly scheme)

$$\alpha \cdot 2 \log_2 P + \beta \cdot 2n + \gamma \cdot n$$

Alltoall: each process sends different $\frac{n}{P}$ data to every other process
Based on a butterfly algorithm:

$$\alpha \cdot \log_2 P + \beta \cdot \frac{n}{2} \log_2 P$$

Cost of collectives on P procs

Routine	Description and cost of efficient algorithm	
Scatter	a root scatters n words, each processor receives n/P words	
	$\alpha \cdot \log_2 \min(n, P) + \beta \cdot n$	
Gather	each processor sends n/P words, which are gathered on root	
	$\alpha \cdot \log_2 \min(n, P) + \beta \cdot n$	
Reduce_scatter	reduction on n words from each processor, result scattered on all processors	
	$\alpha \cdot \log_2 P + \beta \cdot n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2 P + \beta \cdot (n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Allgather	each processor sends n/P words, which are gathered on all processors	
	$\alpha \cdot \log_2 P + \beta \cdot n$	when $n \geq P$
	$\alpha \cdot \log_2 P + \beta \cdot (n + \log_2(P/n))$	otherwise
Reduce	reduction on n words from each processor, result returned on root	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot \log_2(Pn) + \beta \cdot (2n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Broadcast	a root broadcasts n words to all processors	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n$	when $n \geq P$,
	$\alpha \cdot \log_2(Pn) + \beta \cdot (2n + \log_2(P/n))$	otherwise
Allreduce	reduction on n words from each processor, result returned on all processors	
	$\alpha \cdot 2\log_2 P + \beta \cdot 2n + \gamma \cdot n$	when $n \geq P$,
	$\alpha \cdot 2\log_2 P + \beta \cdot 2(n + \log_2(P/n)) + \gamma \cdot (n + \log_2(P/n))$	otherwise
Alltoall	each processor sends different n/P words to every other processor	
	$\alpha \cdot \log_2 P + \beta \cdot \frac{n}{2} \log_2 P$	

MPI_Comm_split

- It is possible to create communicators for subsets of processors

```
int MPI_Comm_split(MPI_Comm comm,
    int color,
    int key,
    MPI_Comm *newcomm)
```

MPI's internal Algorithm:

- Use MPI_Allgather to get the color and key from each process
- Count the number of processes with the same color; create a communicator with that many processes. If this process has MPI_UNDEFINED as the color, create a process with a single member.
- Use key to order the ranks

Color: controls assignment to new communicator
Key: controls rank assignment within new communicator

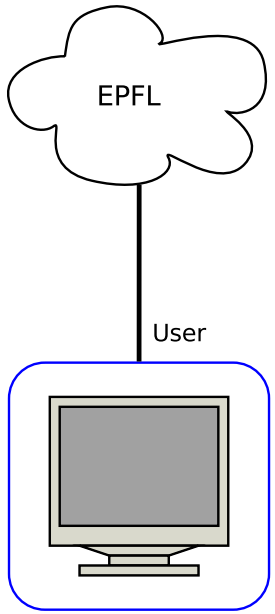
Synchronization

MPI_Barrier(comm)

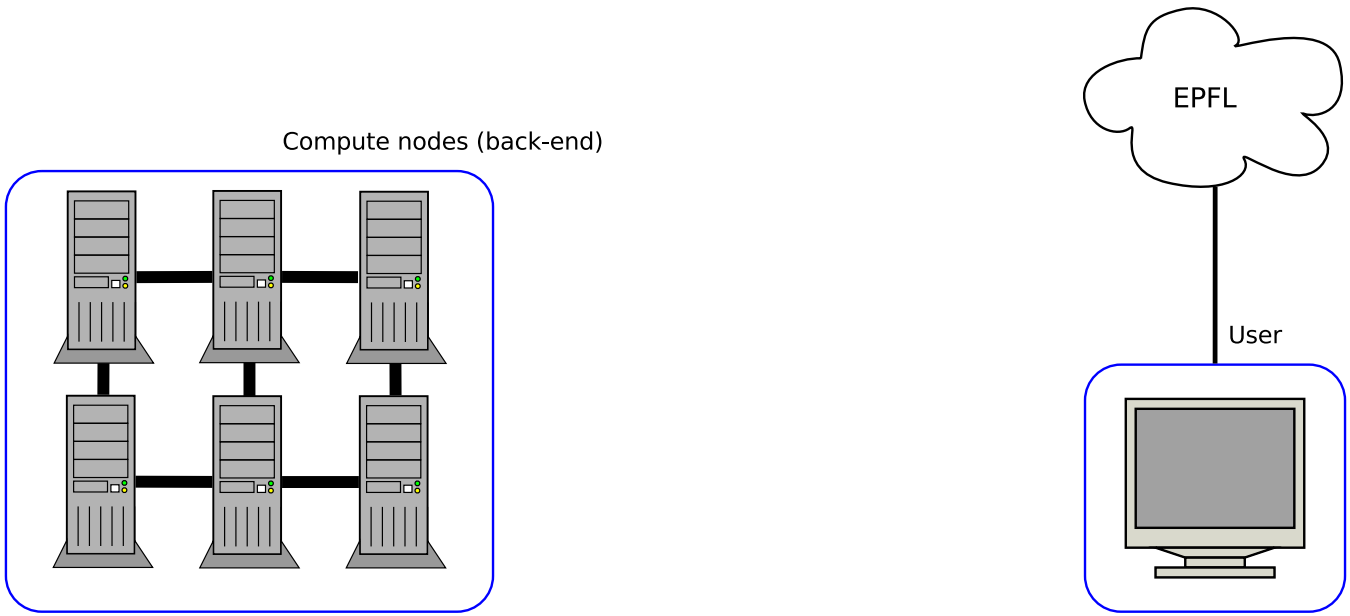
- Blocks until all processes in the group of the communicator comm call it.
- Almost never required in a parallel program
- Occasionally useful in measuring performance and load balancing

Plan

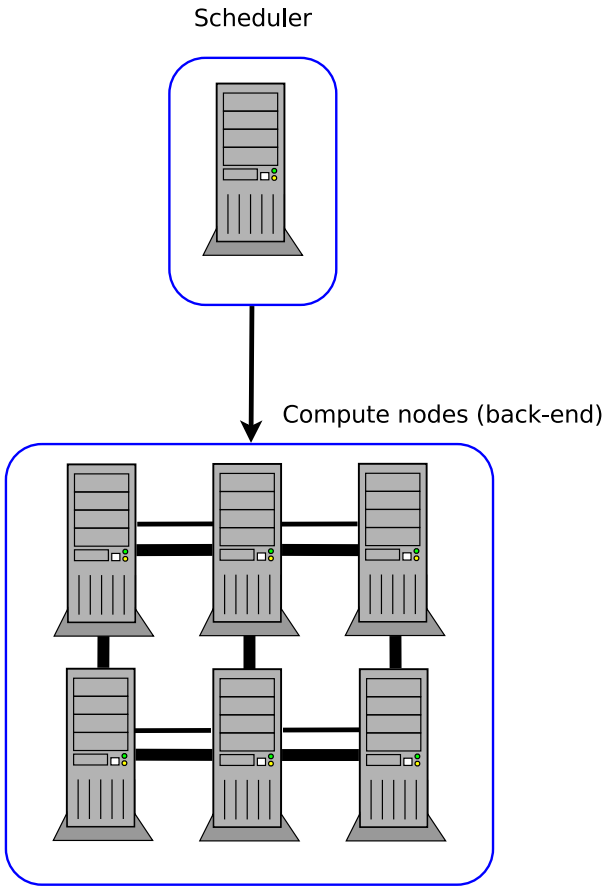
What is a cluster?

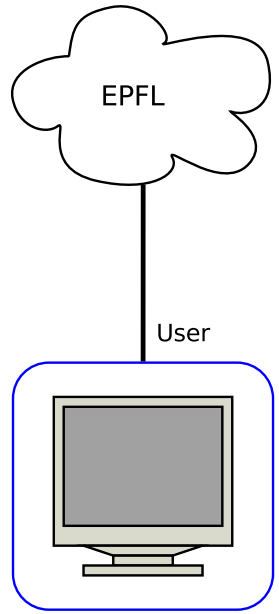


What is a cluster?

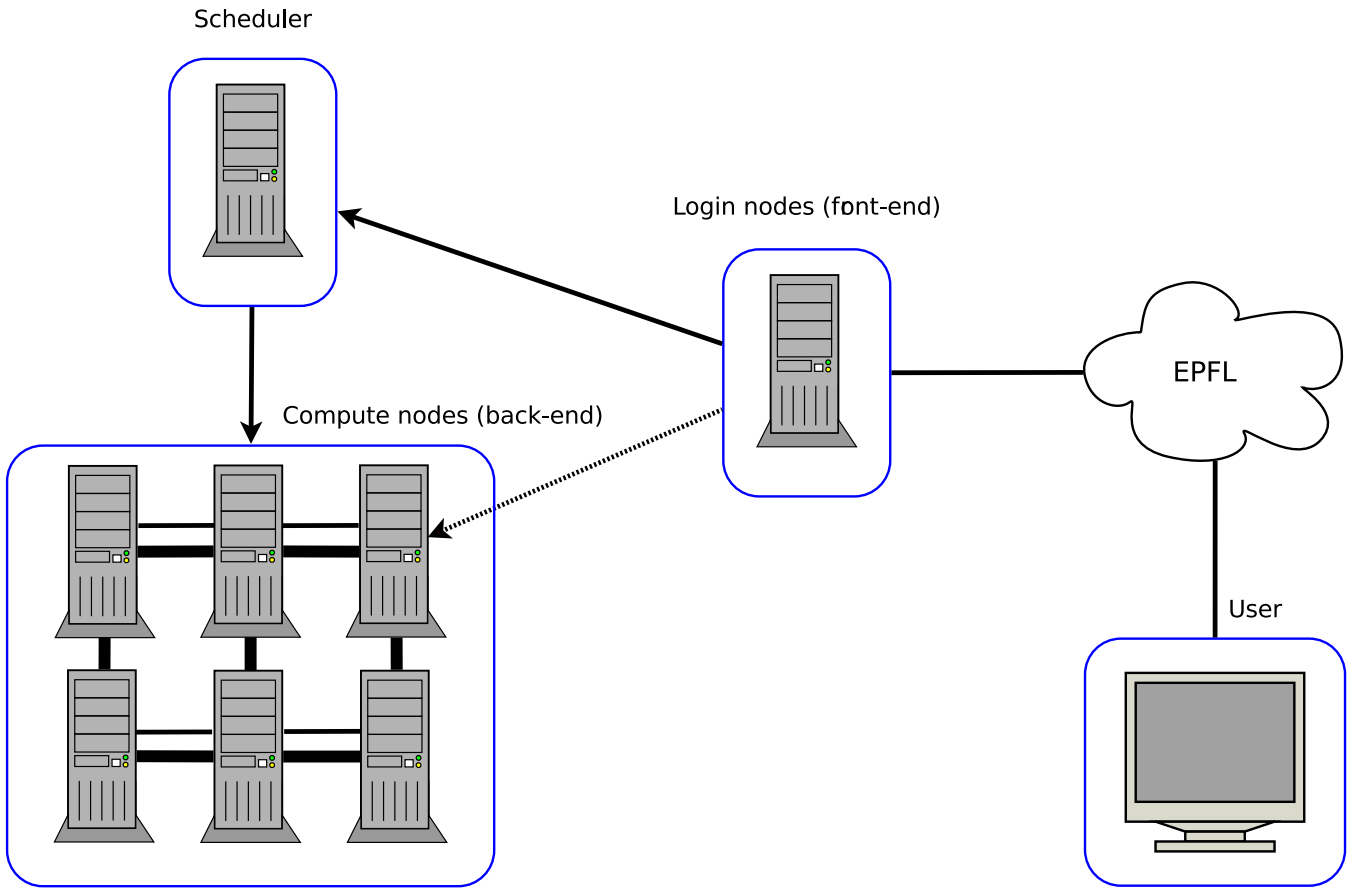


What is a cluster?

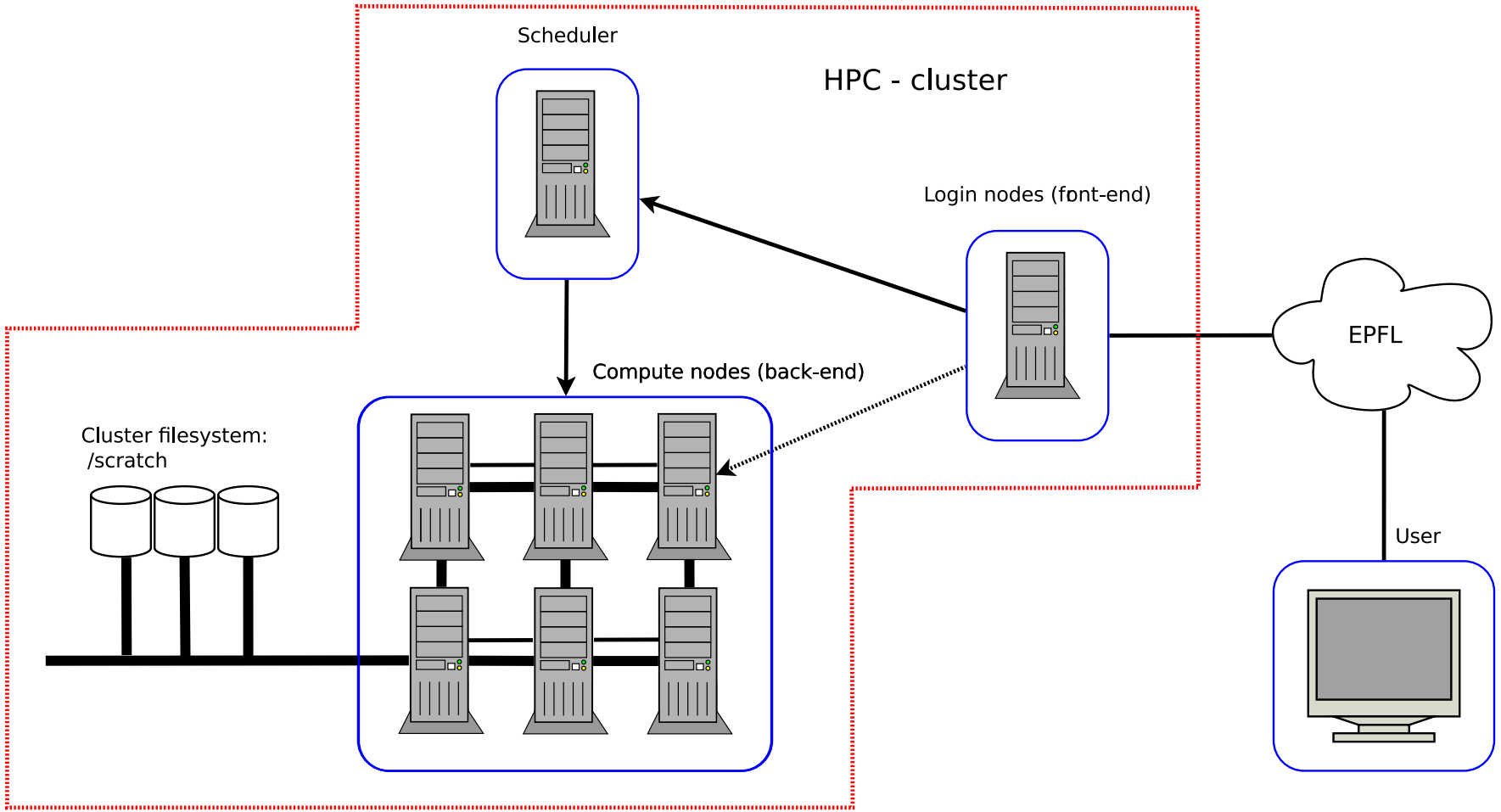




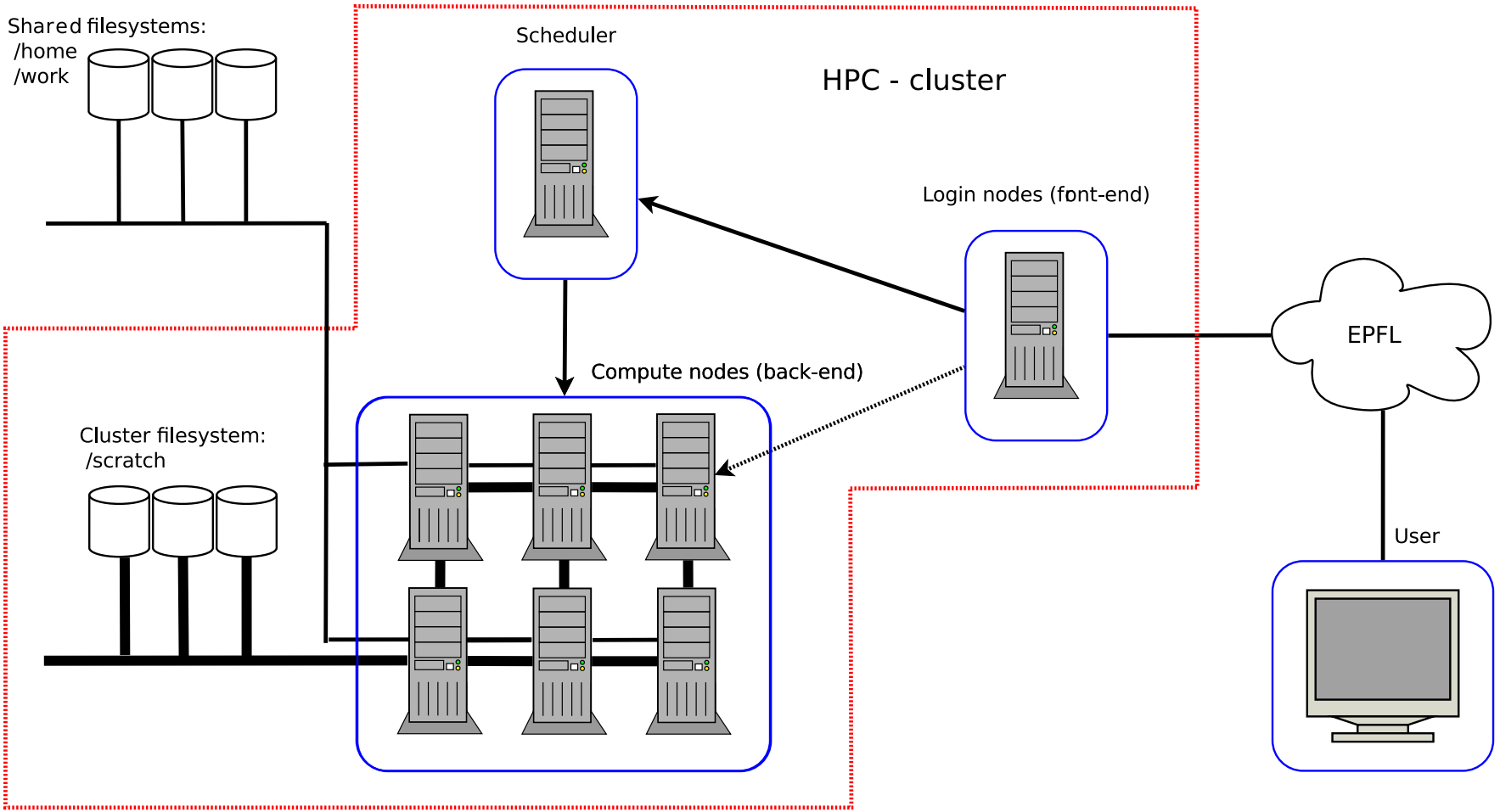
What is a cluster?



What is a cluster?



What is a cluster?



EPFL cluster used in this class

Login	**Nodes**	**Cores**	**RAM**	**Network**
hostname	#	# x GHz	GB	Gbit/s
`helvetios.epfl.ch`	287	2 x 18 x2.3	192	100 (IB)

EPFL clusters storage

- The simulation data is written on the storage systems. At SCITAS:
  - **/home**: store source files, input data, small files
  - **/work**: collaboration space for a group
  - **/scratch**: temporary huge result files
- Please, note that only **/home** and **/work** have backups!
- **/scratch** data can be erased at any moment!

Connecting to remote machines

First step

- Connect to a remote cluster to get a shell
- SSH: Secure SHell

How to use

```
$ ssh -l <username>  <hostname>
$ ssh <username>@<hostname>
```

For windows users

Just install **git**, and use **git bash**

Connecting to remote machines

First step

- Connect to a remote cluster to get a shell
- SSH: Secure SHell

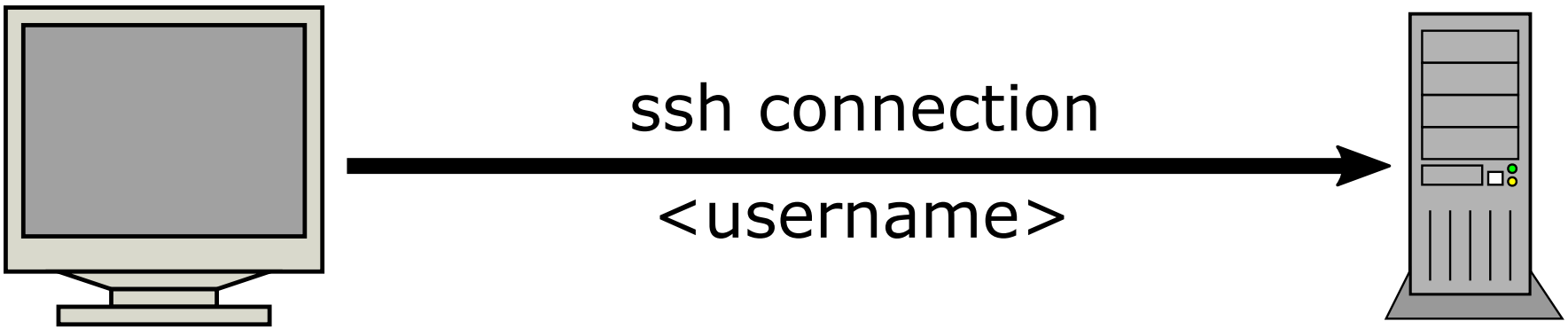
How to use

```
$ ssh -l <username>  <hostname>
$ ssh <username>@<hostname>
```

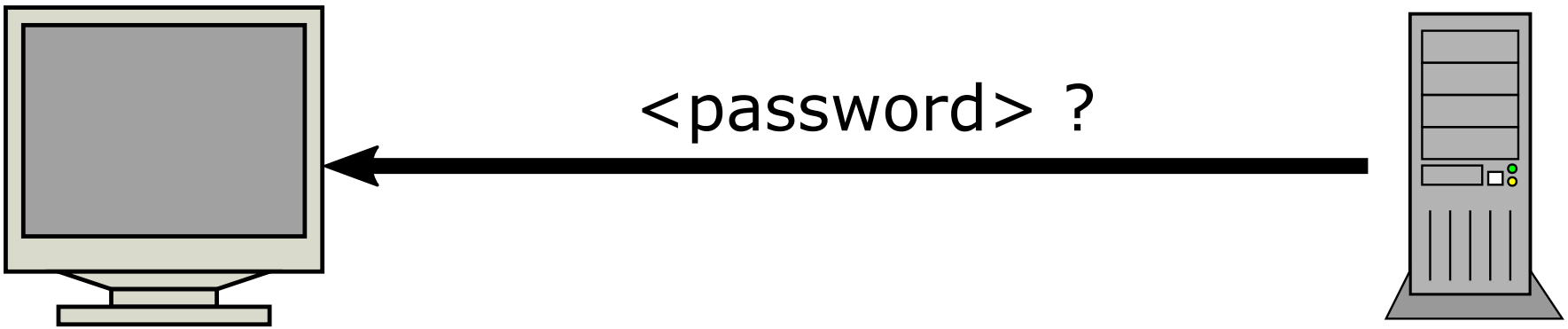
For windows users

Just install **git**, and use **git bash**

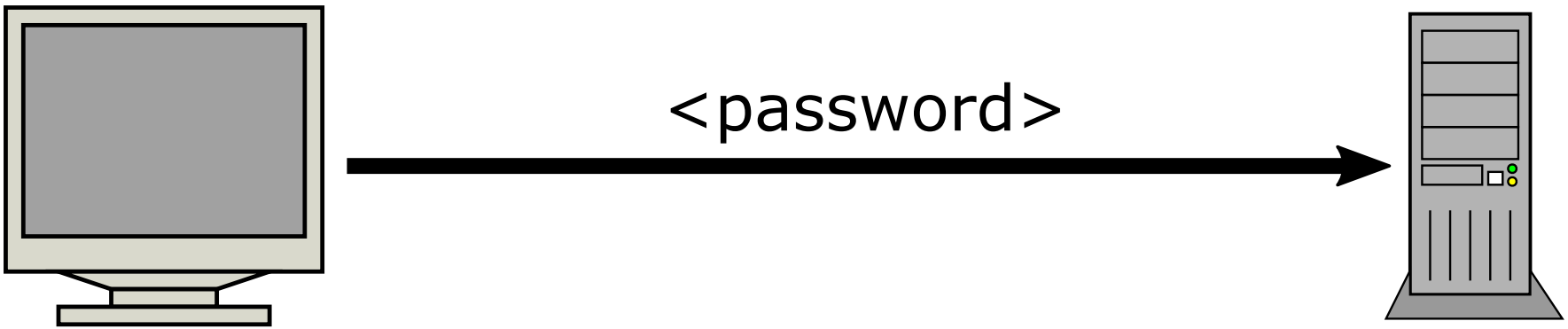
Getting a remote shell



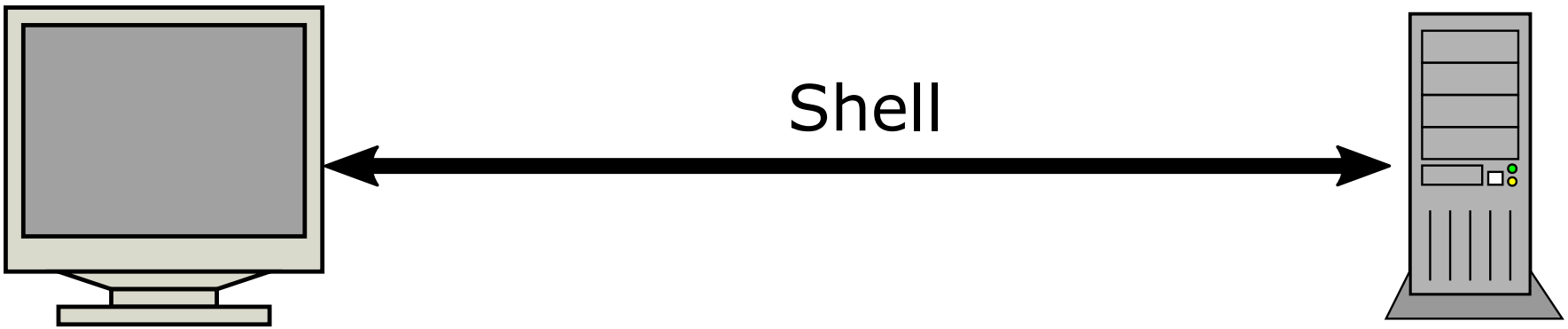
Getting a remote shell



Getting a remote shell



Getting a remote shell



Simple connection

To connect to the front node of a cluster

```
$ ssh -l jdoe  helvetios.epfl.ch
$ ssh jdoe@helvetios.epfl.ch
```

Front nodes

- **helvetios** *[CPU (OpenMP/MPI)]*

- Connect to helvetios front node
- Check the different folders **/home /scratch**

Using **scp**

How to use **scp**/**pscp.exe**

Send data to remote machine:

```
$ scp [-r] <local_path>  <username>@<remote>:<remote_path>
```

Retrieve data from remote machine:

```
$ scp [-r] <username>@<remote>:<remote_path> <local_path>
```

Note: *It is always easier to "send to" and "receive from" the clusters since they have a fix ip/name*

- Copy a file from your machine to the cluster.
- Modify the file (e.g., using **vim**, **nano**) and retrieve it from the cluster onto your machine

Using modules

What are modules

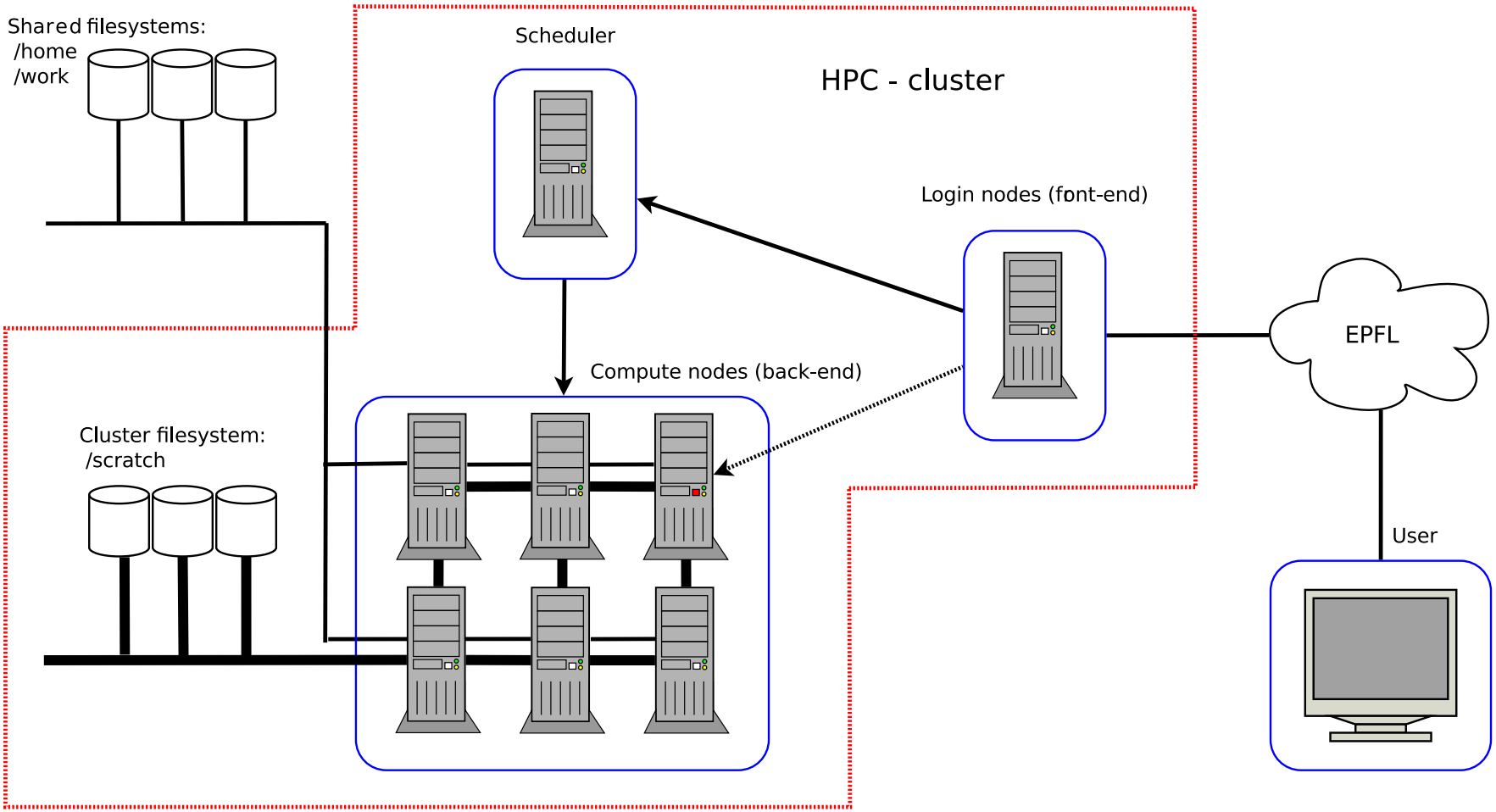
- A way to dynamically modify the environment
- The contain configurations to use an application/a library

How to use them

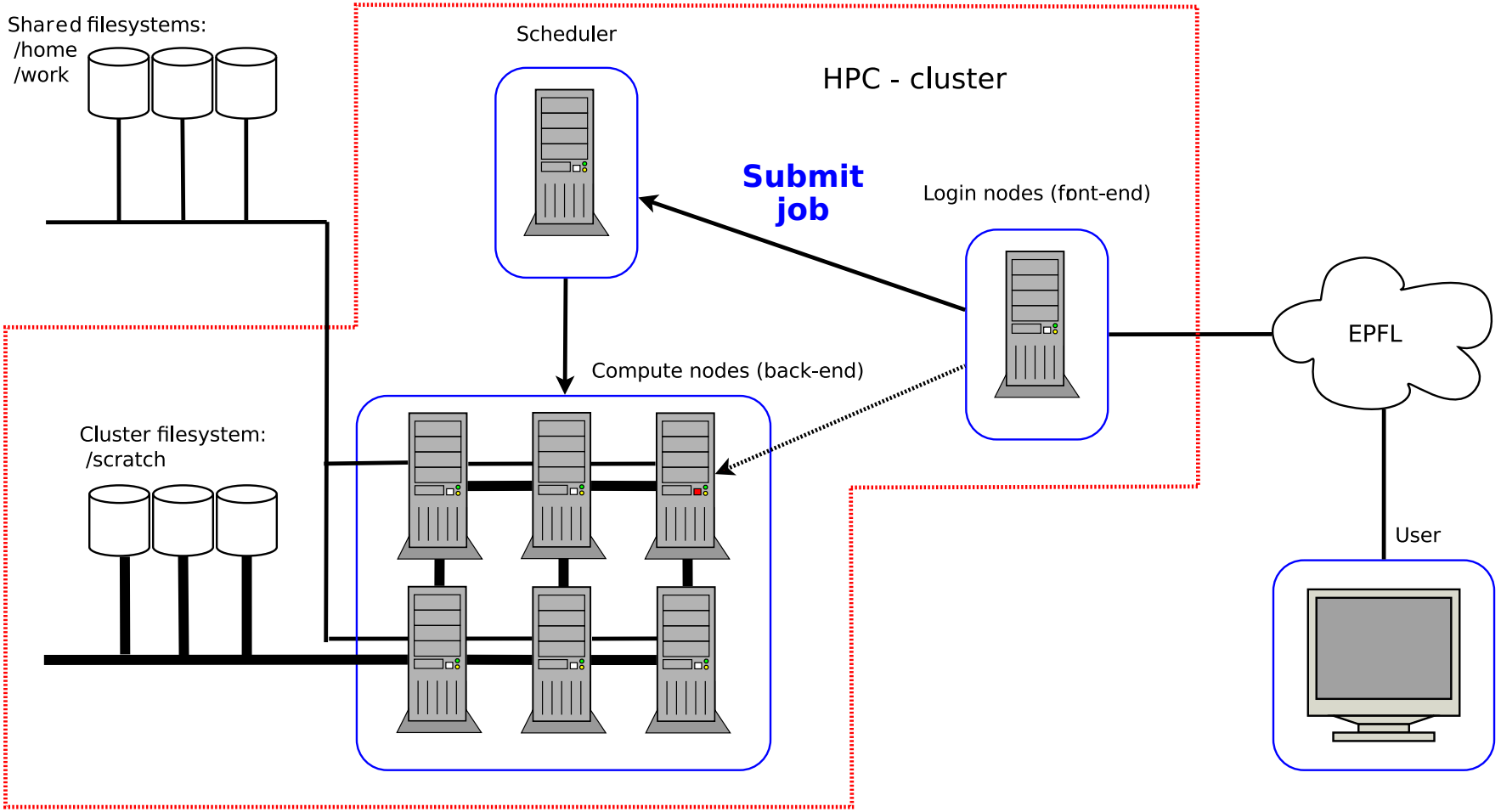
- **module avail** list all possible modules
- **module load module_name** load a module
- **module unload module_name** unload a module
- **module purge** unload all the modules
- **module list** list all loaded modules

SLURM

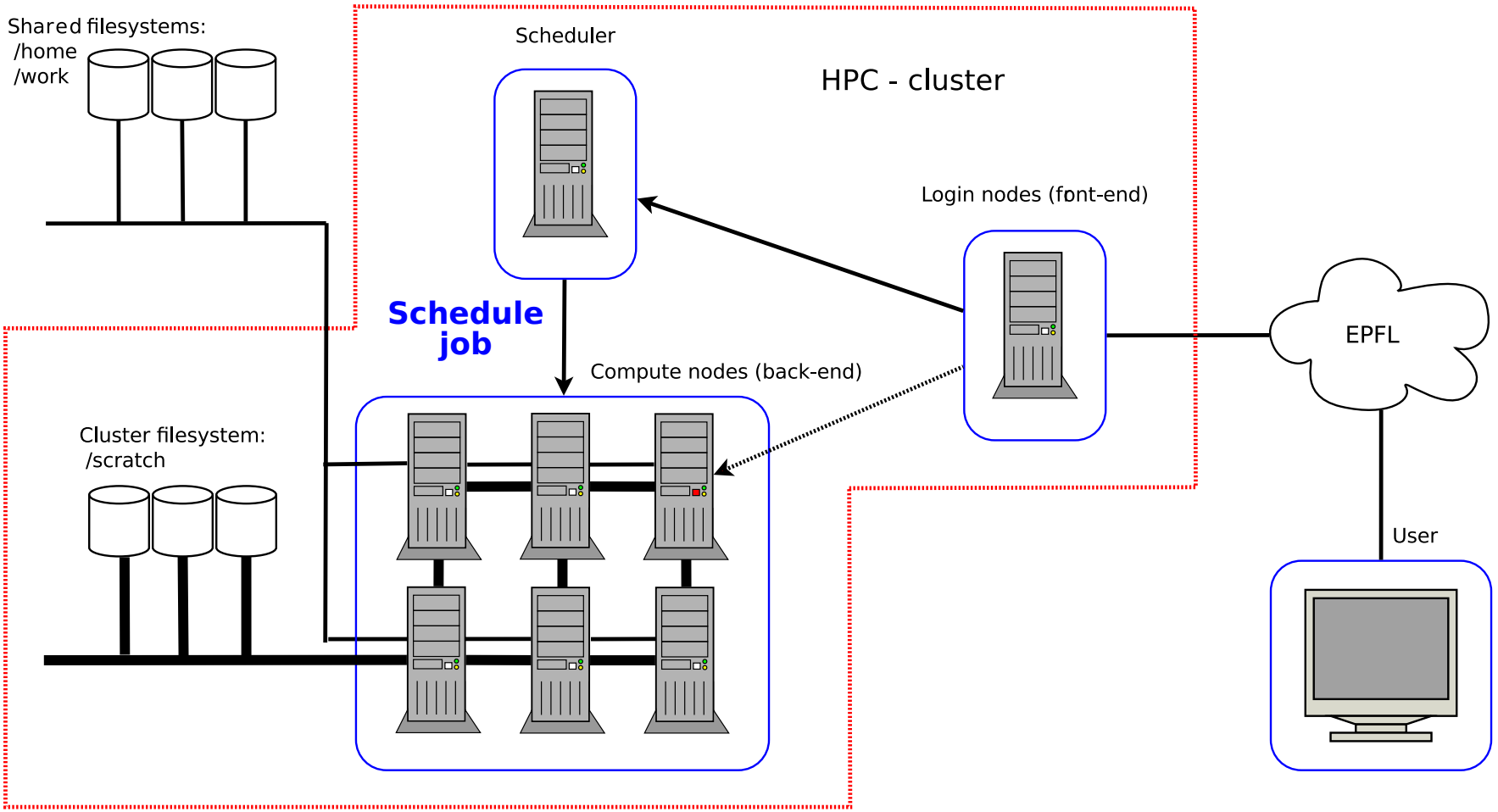
Scheduler


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
User

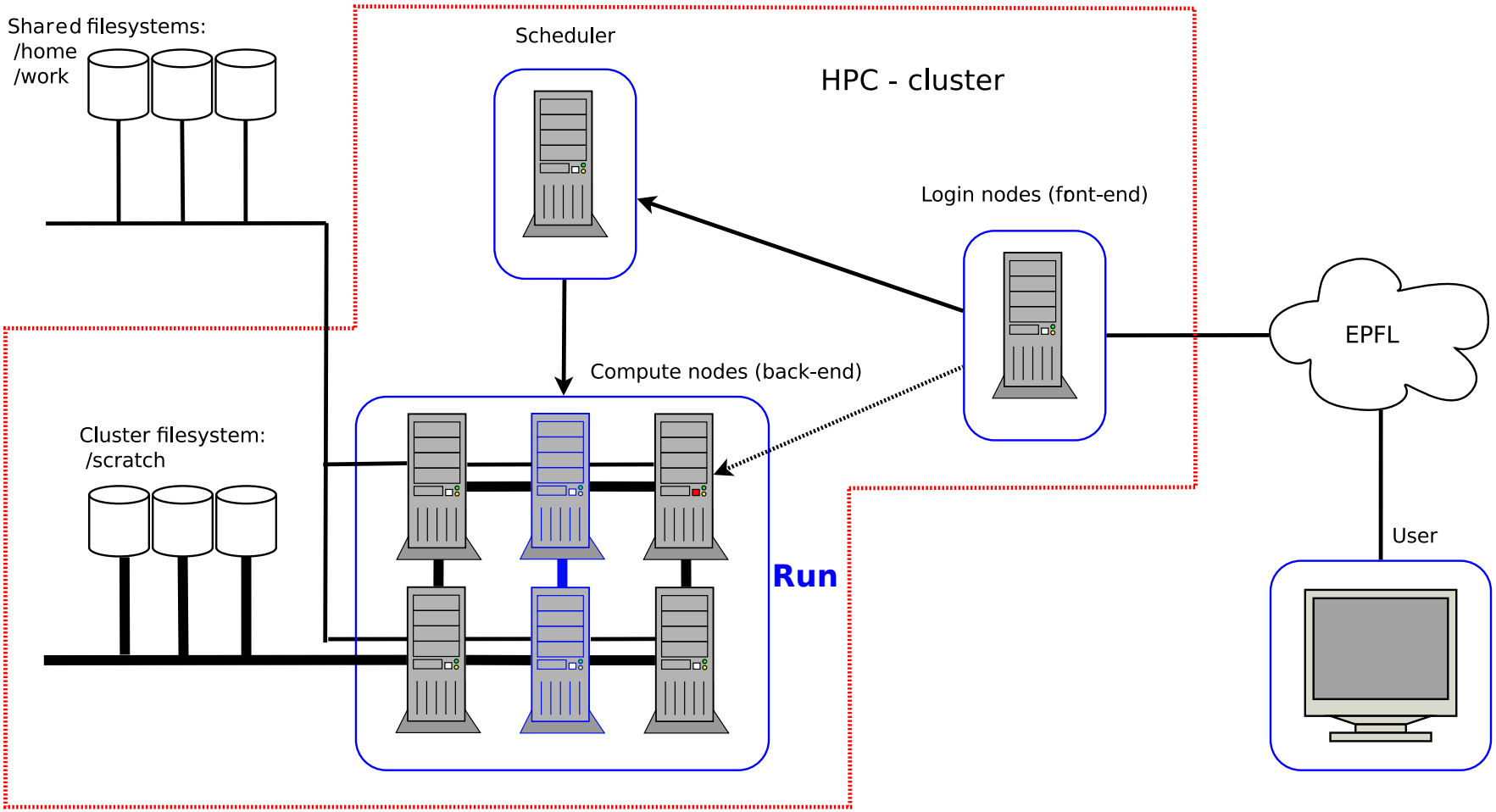
Scheduler


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Submit
job
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
User

Scheduler


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
Schedule
job
Compute nodes (back-end)
EPFL
Cluster filesystem:
/scratch
User

Scheduler


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
Run
User

SLURM

What is SLURM

- Simple Linux Utility for Resource Management
- Job scheduler

Basic commands

- **sbatch** submit a job to the queue
- **salloc** allocates resources
- **squeue** visualize the state of the queue

SLURM: common options

SLURM options

- **-A** / **--account=account_name** name of your SLURM account
  For this class you are in `math-505` Further details on moodle
- **-u** / **--user=username_name** defines the user

SLURM: common options

SLURM options

- **-t** / **--time=HH:MM:SS** set a limit on the total run time of the job
- **-N** / **--nodes=N** request that a minimum of **N** nodes be allocated to the job
- **-n** / **--tasks=n** dvise SLURM that this job will launch a maximum of **n**, in the MPI sense
- **-c** / **--cpus-per-task=ncpus** advises SLURM that job will require **ncpus** per task
- **--ntasks-per-node=ntasks** number of tasks per node
- **--mem=size[units]** defines the quantity of memory per node requested

Need more help? Have a look at the
`https://slurm.schedmd.com/sbatch.html`

SLURM: common options

Or you can put everything in a file: srun to execute a code with 38 MPI ranks over two nodes, 1 thread per rank, 7000 MB of RAM per node, so in total the job gets 14'000 MB, 20 minutes for the job, parallel QOS

{mysimulation.job}

```
#!/bin/bash -l
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=19
#SBATCH --cpus-per-task=1
#SBATCH --mem=7000
#SBATCH --time=20:00
#SBATCH --qos=parallel
#SBATCH --account=math-505
module load gcc openmpi python py-mpi4py
srun ./my_python_script.py
```

and submit the job

```
$ sbatch mysimulation.job
```

To continue

during exercise sessions !
To simplify the execution and understanding of results:

- Allocate one MPI process per core
- use 1 thread per MPI process (rank) - no multithreading
- MPI will use explicite communication, independently if the processes are run on a same node or not

Acknowledgement

- Introduction to using the EPFL cluster: slides from P. Antolin, N. Richart, E. Lanti, V. Keller's lecture notes