LUND UNIVERSITY

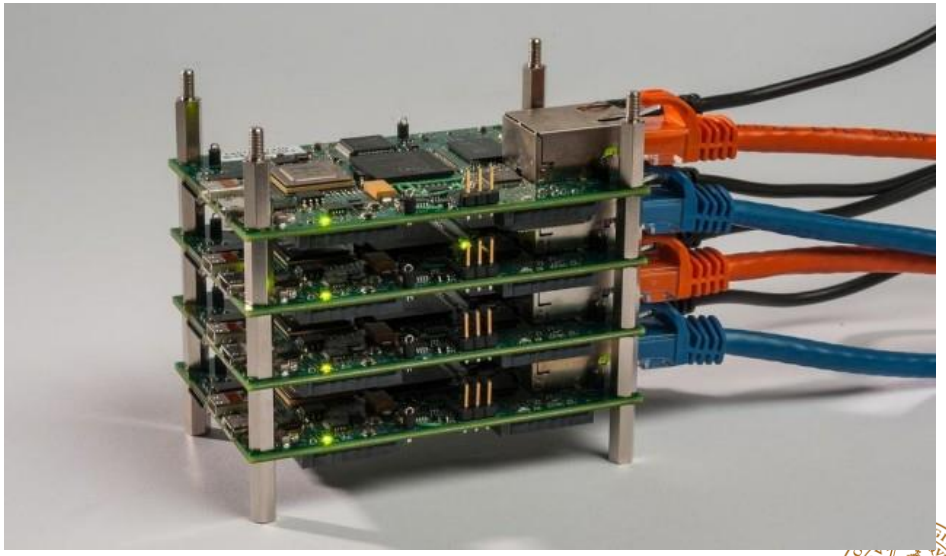
Active Learning: Lunarc/Iridium, batch systems

COMPUTE RESEARCH SCHOOL COURSE NTF004F

Basic concepts of parallelism

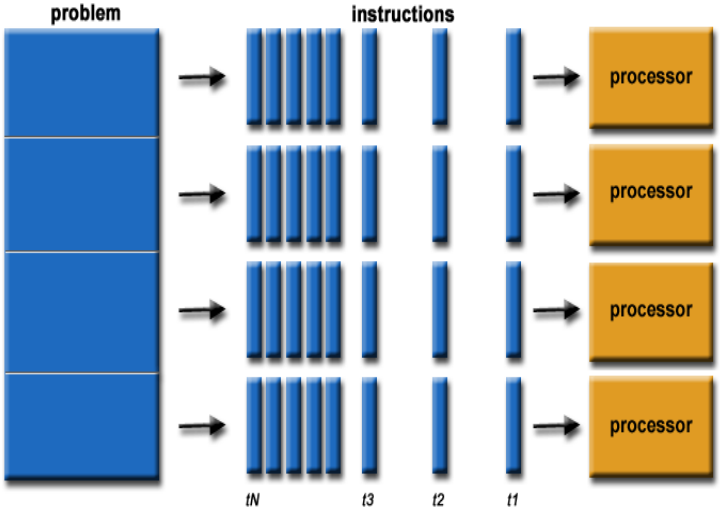
Outline

- Multi-core computing in a nutshell
  - motivation, terminology, difficulties
- Handy tools for remote sessions
  - ssh, screen
- Riding on the clusters
  - Batch system basics
- Task farming
  - Scaling experiment



What is parallel computing?

- Traditional computing: serial execution of a single stream of instructions on a single processing element
- **Parallel computing**: simultaneous execution of stream(s) of instructions on multiple **processing elements**
  - **Non-sequential** execution of a computational task
  - (part of) the problem solved by **simultaneous** subtasks (processes)
  - Relies on the assumption that problems can be divided (decomposed) into smaller ideally independent ones that can be solved **parallel**



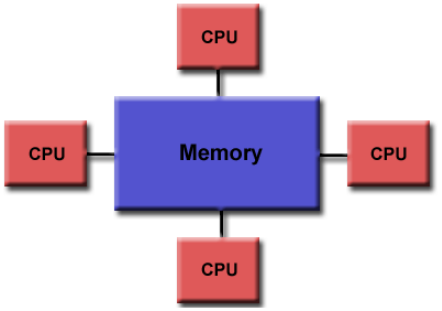
What is parallel computing (cont.)?

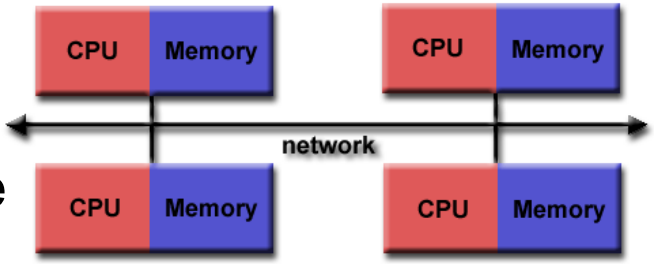
- **Parallelism levels** (”distance” among the processing elements):
  - Bit and Instruction level: inside the processors (e.g. 64 bits processor can execute 2 32 bits operations)
  - Multicore/multi cpu level: inside the same chip/computer. The processing elements share the memory, system bus and OS.
  - Network-connected computers: clusters, distributed computing. Each processing element has its own memory space, OS, application software and data
    - » Huge difference depending on the interconnects: e.g. High Performance Computing (supercomputers) vs. High Throughput Computing (seti@home)

Some classifications

SMP vs. MPP (or the **shared memory** vs. **distributed memory** debate):

- SMP: Symmetric Multi Processors system: shared memory approach
  - ”single box” machines, OpenMP programming family
- MPP: Massively Parallel Processors system: distributed memory, network-connected CPUs
  - ”clusters”, MPI programming family (message passing)
- SMPs are easier to program but scale worse than the MPPs





Why parallel computing?

- It is cool
- Sometimes the problem does not fit into a single box: you need more resources than you can get from a single computer
- To obtain at least 10 times more power than is available on your desktop
- To get exceptional performance from computers
- To be couple of years ahead of what is possible by the current (hardware) technology
- The frequency scaling approach to increase performance does not work any longer (power consumption issues):
  - The new approach is to stuff more and more processing units into machines, introducing parallelism everywhere

Measuring performance gain: the Speedup

- **In an ideal scenario** a program running on **P processing elements** would execute **P times faster**..., giving us a linear speadup
- **Speedup S(n,P):** ratio of execution time of the program on a single processor (T_1) and execution time of the parallel version of the program on P processors (T_P):
  - » In practice, the performance gain depends on the way the problem was divided among the processing elements and the system characteristics.

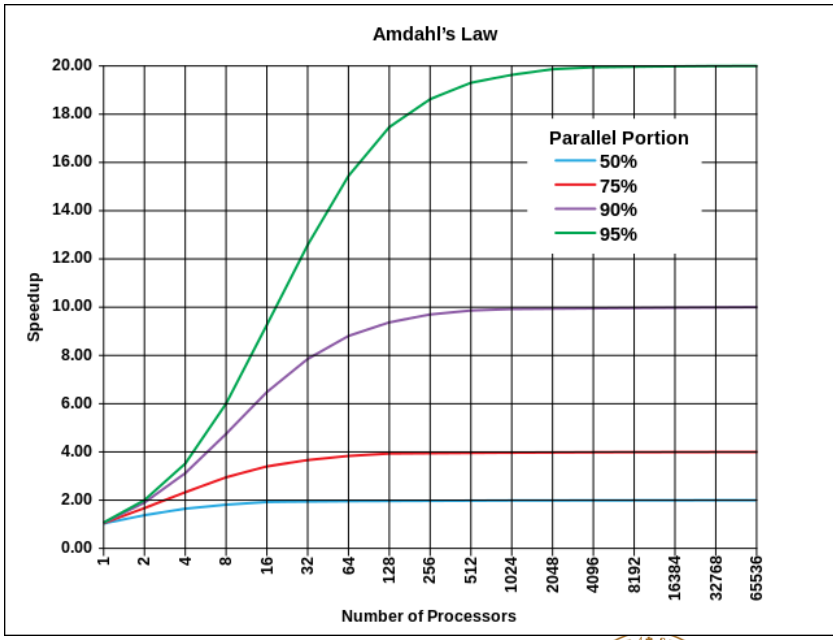
$$S(n,P) = \frac{T(n,1)}{T(n,P)}$$

`n denotes the problem size.`
`T denotes the execution time.`

- **Amdahl's law**: gives an upper estimate for maximum **theoretical speedup** and states that it is limited by the non-parallelized part of the code:

$$S(n,P) \leq \frac{1}{\alpha + (1-\alpha)/P} \leq \frac{1}{\alpha}$$

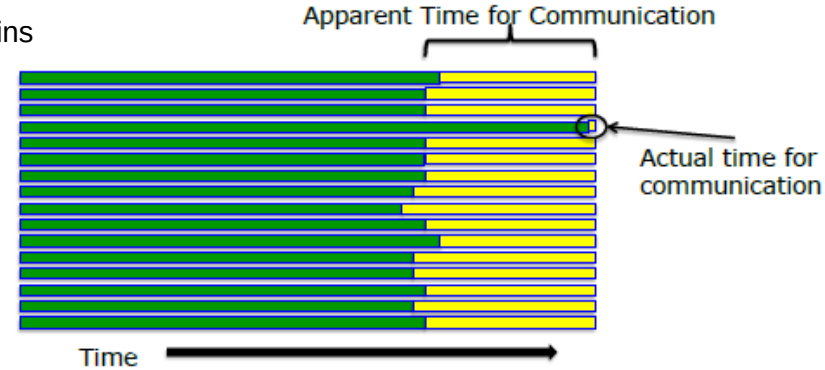
  - alpha is the sequential fraction of the program
  - e.g. if 10% of the code is non-parallizable, then the maximum speedup is limited by 10, independent of the number of used processors (!)



The dark side ....

”the bearing of a child takes nine months, no matter how many women are assigned”

- Not everything is suitable for parallelization
- Complexity increases as more and more communication is involved:
  - embarrasingly paralell -> coarse-grained -> fine-grained problem domains
- Parallel computing opens up new set of problems:
  - Communication overheads
  - Concurrency problems
  - Synchronization delays
  - Race conditions and dead locks

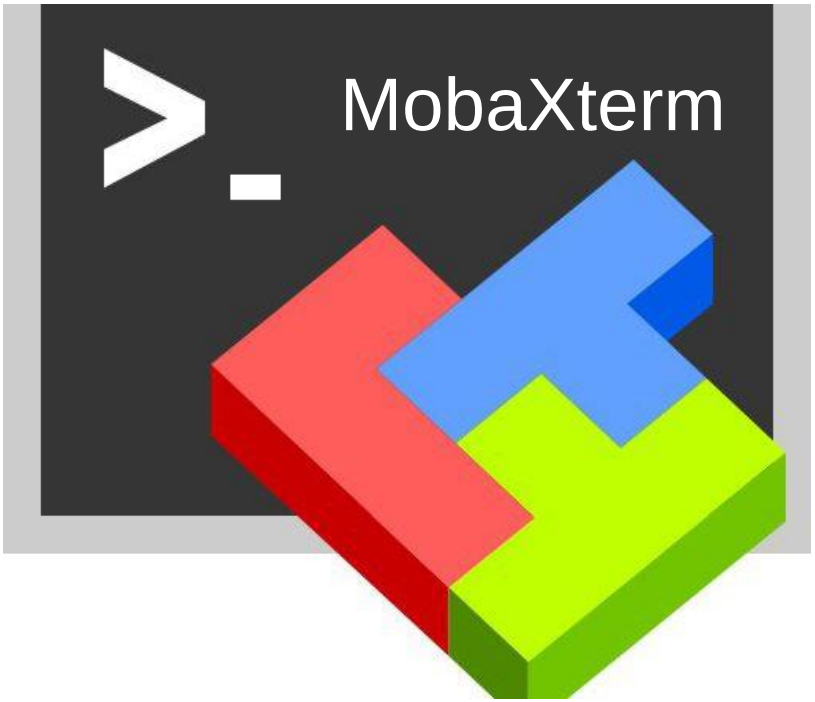


- Nobody wants to debug a parallel code...
- Developing & deploying a parallel code usually consume more time than the expected speedup
- A practical advice for parallelization:
  - Unless you have an embarrasingly parallel problem, forget it
  - If you are stubborn, then at least use an available parallel (numerical) library and start with the profiling (understanding) of your program
  - Wait for the holy grail of computational science: automatic parallelization by compilers

LUND UNIVERSITY

Now comes active learning ☺

Accessing remote computers



LUND UNIVERSITY

Task 1: Howto avoid loosing all your work on a remote computer

IMAGINE that:

- You are being logged on a remote computer
- In the middle of a long task (e.g. compilation, download, etc..)
- Then, suddenly the network connection dies
- or you'd like to go home and continue the same work from your home

Is there a way to avoid loosing all your work? How can one disconnect & "session" without the need to restart everything from scratch?

TODO: try to use the **screen** utility to keep your remote session alive. Launch screen and start working in several screen session. Then, imitate a network failure and with the help of screen resume your work on the remote machine.

screen allows you to

- Keep a session active even through network disruptions
- Disconnect and re-connect to a sessions from multiple locations (computers)
- Run a long remote running process without maintaining an active remote login session

Task 1: helpdesk

Use the Linux **screen** utility to manage remote screen sessions, connect, reconnect to active session, survive a network failure ☺

- Screen is started from the command line just like any other command
  - **[iridium ~]$: screen**
  - You can create new “windows” inside screen, ctr+a c then rotate, switch between windows with ctrl+a n
- Listing your screens:
  - **[iridium ~]$: screen -list**
- Disconnecting from your active session, screen (your task keeps running!):
  - **[iridium ~]$: screen –d** or **ctrl+a d**
- Re-connecting to an active screen session (re-attach to screen):
  - **[iridium ~]$: screen –r**

– Terminating, logging out of screen

  - type **exit** from inside all your active screen sessions

– Using screen to log your activity:

  - **[iridium ~]$: screen –L** or **ctrl+a H** turns on/off logging during a screen session

LUND UNIVERSITY

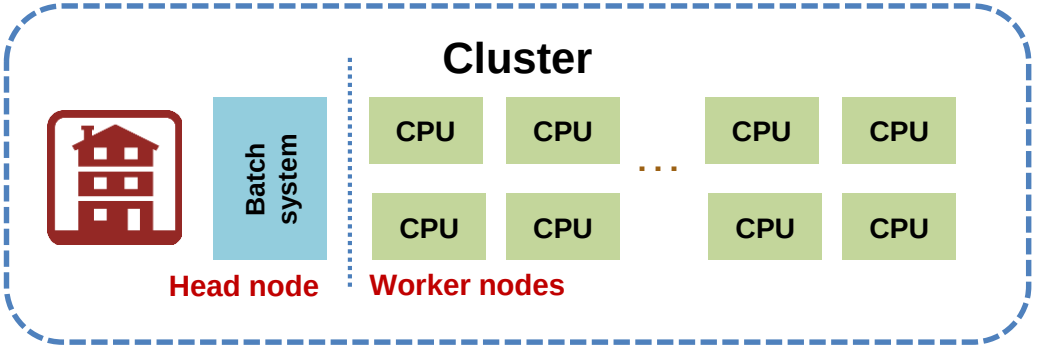
Task 2: Understand the layout of a cluster: Head node vs. Worker node

- Head (login) node: pptest-iridium.lunarc.lu.se
- Worker nodes: n3, n4, ..., n12

TODO: try to use various linux commands to discover the structure of the iridium cluster:

Hint: you can „login" to a node by using the *interactive* command

- is there shared file system?
- are the nodes identical (cpus, memory)?
- how about network connectivity?
- how about user accounts?
- how about the operating system, available software?
- how about running processes, cpu-load?



Task 3: Understand the basic concept of „resource schedulling” by using SLURM

- There is no infinite size cluster ☺
  - Always limited number of cores, memory, walltime
  - Workload Management System, Scheduller, Batch System: SLURM

TODO: get greedy and eat all the cake! Here is the „menu”:

- slurm.schedmd.com/quickstart.html
- lunarc-documentation.readthedocs.io/en/latest/batch_system/#first-example-for-a-job-submission

Hint: grab a piece of cake by creating a so-called job on the cluster

  - Find out howto create (submit) a job. Hint: use the *sbatch* command
  - Find out the state of the cake. Hint: use *sinfo, squeue*
  - Find out how to control the size of a piece of cake? (number of nodes, cores, memory, etc..)
  - Find out how to put back a piece of cake (cancel job)
  - Find out how to obtain status of the job, receive notification

LUND UNIVERSITY

Task 3: helpdesk

- List SLURM queues (partitions)
  - **> sinfo**
- Create file **myscript** (use provided examples)
- Submit simple jobs and check their status:
  - **> sbatch myscript**
  - **> cat slurm-<jobid>.out**
  - **> squeue**
  - **> scontrol show job <jobid>**
- Repeat with multi core/node jobs
  - **sbatch –N4 myscript**
  - **sbatch –n6 myscript**
  - In a multi-core advanced example, pay attention how jobs are distributed across nodes and cores
- Receive notification on state changes using the directives within the scripts (not enabled on iridium):
  - #SBATCH --mail-user=fred@institute.se
  - #SBATCH --mail-type=END (BEGIN, END, FAIL, REQUEUE, ALL)

Simple `myscript`:

```
#!/bin/sh
#SBATCH -J “simple job”
#SBATCH --time=1
echo “we are on the node”
hostname
who
sleep 2m
```

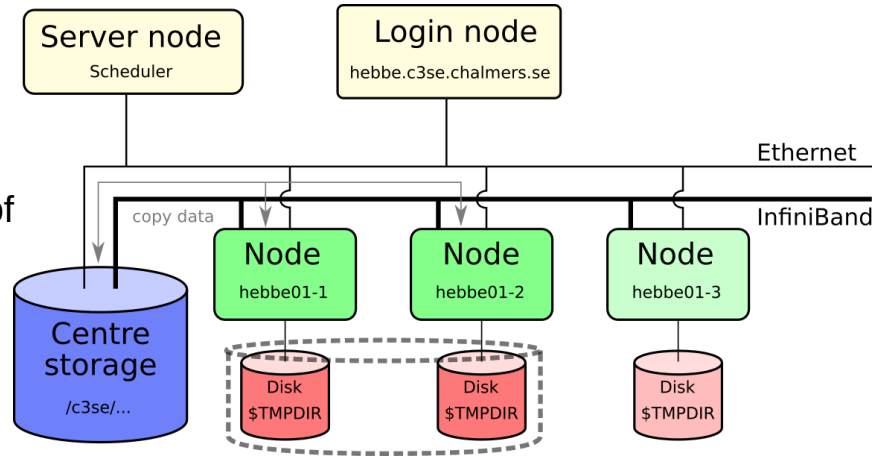
Multicore/node `myscript`:

```
#!/bin/sh
#SBATCH -J “multi job”
#SBATCH --time=1
srun hostname |sort
sleep 5m
```

Task 4: Understand the concept of work directory, input & output of batch jobs

Unless it is a trivial „hello world” exercise, most of the real-life jobs process some input data and produces output data. It is very important to understand where all that data is located relative to the worker node.

TODO: Execute a non-trivial task as a batch job submitted to SLURM that either reads in some data from a file and/or generates some output. As a second example, use the sysbench toolkit submitted as a batch job to measure IO performance on various filesystems.



- Find out what happens to the standard output and standard error of the „executable”
- Find out what is the execution directory on the Worker Node
- Find out where the output files are placed.
- Use special variables understood by SLURM to place your job into a specific directory (SLURM_SUBMIT_DIR, TMPDIR)
- Execute the *sysbench --test=fileio* testkit as part of a batch job to measure disk performance
  - Info and manual about sysbench: *wiki.gentoo.org/wiki/Sysbench*

LUND UNIVERSITY

Task4: helpdesk

Sysbench `myscript`:

```
#!/bin/sh
#SBATCH -J “ioperf”
#SBATCH --time=10

echo “we are on the node”
hostname
who
sleep 2m
```

Stdout/Stderr `script`:

```
#!/bin/sh
#SBATCH -J "ioperf"
#SBATCH --time=4
#SBATCH -o ioperf_%j.out
#SBATCH -e ioperf_%j.err

echo "we are on the node and testing io"
hostname
cd $TMPDIR
mkdir testdir
cd testdir
pwd

sysbench --test=fileio --file-total-size=4G prepare
sysbench --test=fileio --file-total-size=4G --file-test-mode=rndrw --max-time=120 --max-requests=0 run
sysbench --test=fileio --file-total-size=4G cleanup
```

LUND UNIVERSITY

Task farming

Implementing trivial parallelism with a master –worker system

- With a help of a master script you are going to **execute X number of subtasks on Y number of processing units**
- The master script (master.sh) takes care of launching (new) subtasks as soon as a processing element becomes available
- The worker.sh script imitates a payload execution that corresponds to a subtask

Steps:

1. Copy the scripts to a new directory on pp-test-iridium
2. Set the problem size (*NB_of_subtasks*) and the number of processing elements (*#SBATCH -n*) in the *master.sh*, the payload size (i.e. How long a subtask runs, PAYLOAD) in the *worker.sh*
3. Launch the taskfarm (*sbatch master.sh*), monitor the execution of the subtasks (*squeue –j <jobid> -s*) and finally check how much time the taskfarm processing required (check the output files of the subtasks and the slurm job)
4. Repeat the taskfarming with modified parameters, What is the speedup?