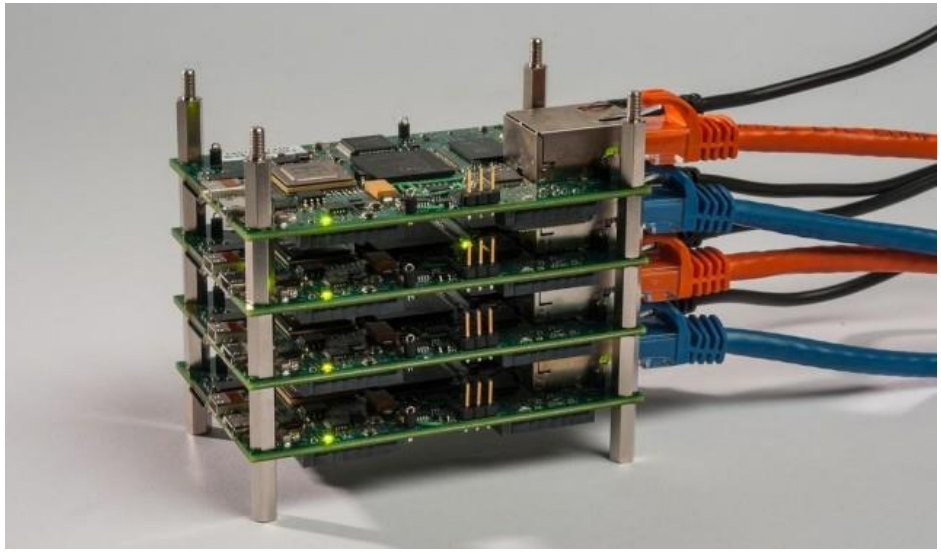
Introduction to Programming and Computing for Scientists

Tutorial-7a: Parallel (multi-cpu) Computing

Outline

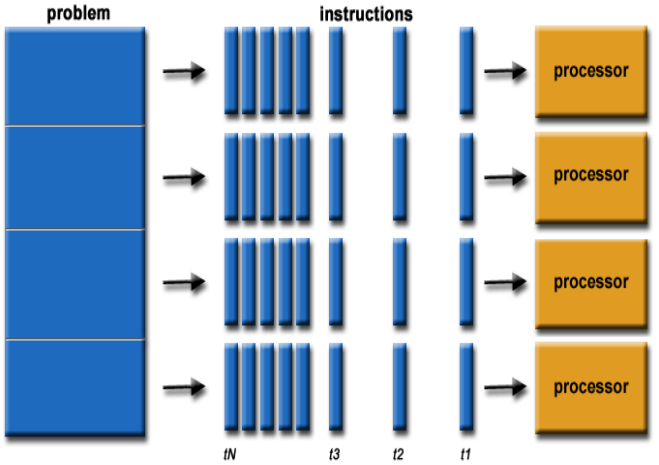
- Parallel computing in a nutshell:
  - motivation, terminology, solutions
- Howto ride on ”big iron”:
  - The practical basics of working with batch systems
- Multi-task jobs



http://arstechnica.com/information-technology/2013/07/creating-a-99-parallel-computing-machine-is-just-as-hard-as-it-sounds

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-sequantial** 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?

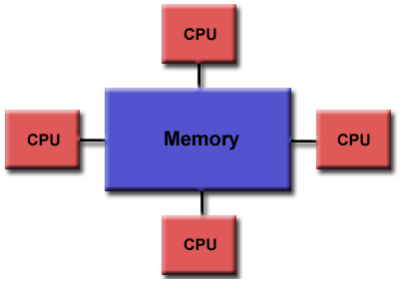
- **Parallelism levels** (”distance” among the processing elements):
  - Bit and Instruction level: inside the processors (e.g. 64 bits processor can execute 2 32 bits operations)
  - Multicore/multi cpu level: inside the same chip/computer. The processing elements share the memory, system bus and OS.
  - 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

Flynn's taxonomy:

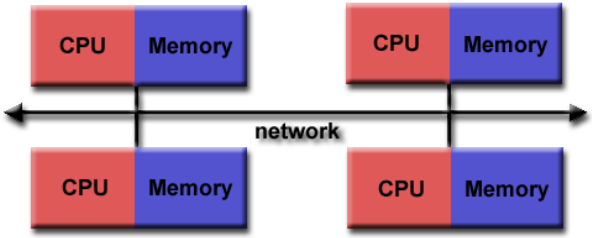
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

- SISD: sequential "normal" programs
- MIMD: most of the parallel programs
- SIMD: data chewing by the same algorithm
- *MISD: rarely exists*



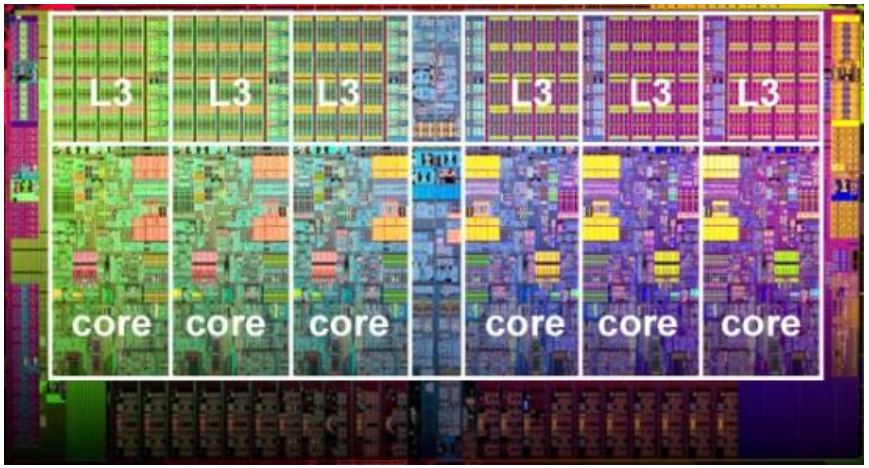
SMP vs. MPP (or the **shared memory** vs. **distributed memory** debate):

- SMP: Symmetric Multi Processors system: shared memory approach
  - "single box" machines, OpenMP programming family
- MPP: Massively Parallel Processors system: distributed mem 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.

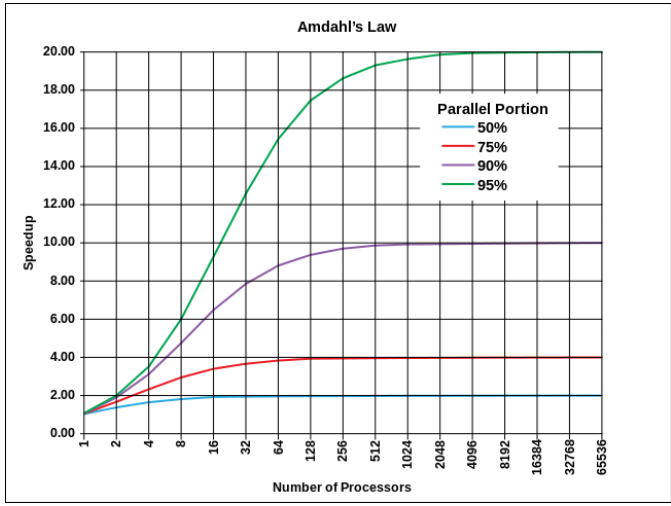
$$S(n,P) = \frac{T(n,1)}{T(n,P)}$$

`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 (!)

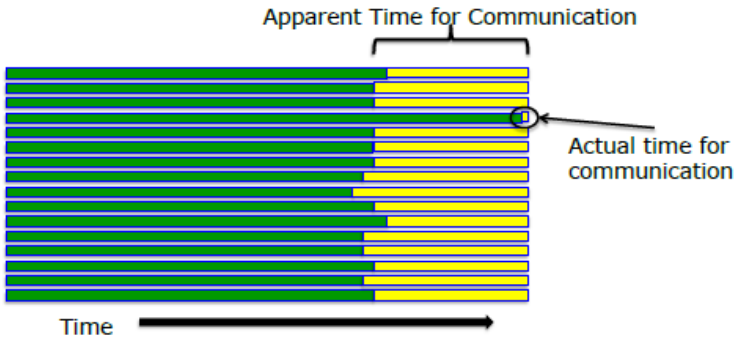


source: wikipedia

The dark side ....

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

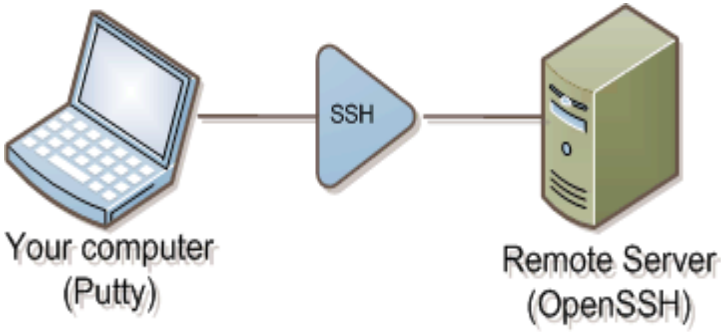
- Not everything is suitable for parallelization
- Complexity increases as more and more communication is involved:
  - embarrasingly paralell -> course-grained -> fine-grained problem domains
- Parallel computing opens up new set of problems:
  - Communication overheads
  - Concurrency problems
  - 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 ☺

Accessing remote computers

- Secure Shell (SSH) is a secure way of accessing remote computers, executing commands remotely or moving data between computers.
  - All network traffic is encrypted
  - The de-facto protocol for remote login & computer access
  - SSH clients are available on non-linux platforms too (*putty and winscp on windows*)
  - SSH servers are listening to incoming connections on the standard TCP 22 port
  - Login is done with username/passwd or using keypairs (advanced topic)

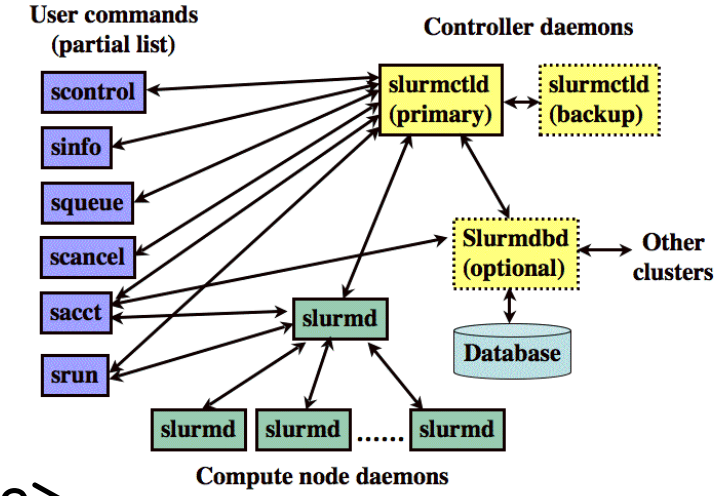


- Various Windows and Linux clients exist

Exercise 1: working with a cluster

Goal: understand basic concepts of cluster, Workload Management system, queue, jobs

- Cluster: *Iridium* cluster at LUNARC
- Frontend: **pptest-iridium.lunarc.lu.se**
- Batch system: SLURM



Tasks:

- Log in to the cluster:
  - `> ssh -X <username>@<clustername>`
- Copy files to the cluster:
  - **scp localfile user@machine:remote_dir**

- Look around on the front-end (e.g. inspect CPU and memory details):
  - `cat /proc/cpuinfo; cat /proc/meminfo; top`
  - `who, pwd`
- Check man pages for SLURM commands:
  - `sbatch, sinfo, squeue, scontrol, scancel`

Exercise 2: simple jobs

- 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

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
```

Exercise 3: task farming

- 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. Download, 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) 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?

Further reading

- Introduction Parallel computing (by Lawrence Livermore National Laboratory)
  - https://computing.llnl.gov/tutorials/parallel_comp/
  - most of the images are taken from this tutorial
- SLURM:
  - https://computing.llnl.gov/linux/slurm/quickstart.html