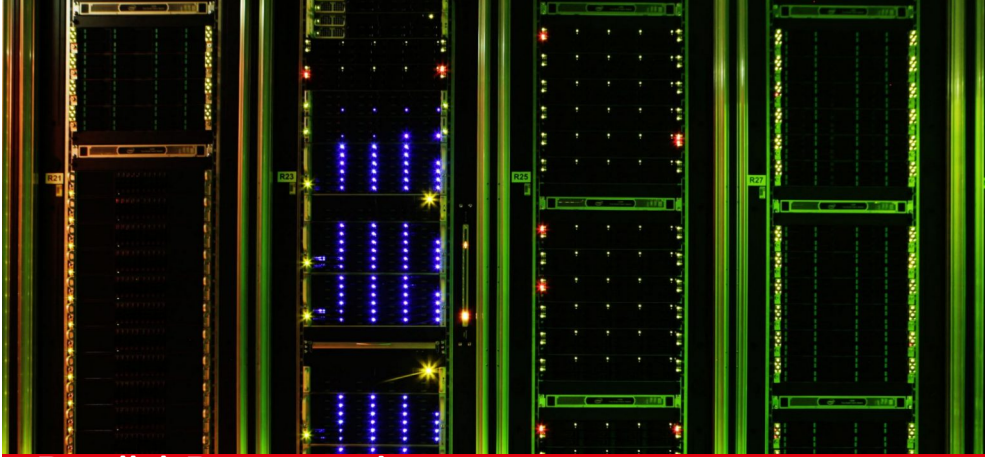


Parallel Programming

Single-core optimization, MPI, OpenMP, and hybrid programming

Nicolas Richart
Emmanuel Lanti

Course based on V. Keller's lecture notes

27th of October - 1st of November 2024

Administration

Course organization

- This parallel programming course amounts to two full weeks of work (70 hours)
- It is organized in two parts:
  - A week of "theoretical" lectures completed by practical exercises
  - A personal project realized within the two following weeks
- An oral evaluation of your project (15' + 5') will conclude the course
- If passed, you'll get 3 ECTS

A few remarks

- During the course, we'll primarily use C++ for the examples, but you can also use C or Fortran
- Do not hesitate to interrupt if you have questions
- Exercises are important! Do not hesitate to play with them, change the parameters, see what happens, make the code crash, try to understand why, etc.

Lecture and exercises

- We tried to build this course with as much exercises as possible
- We often use exercises during the lectures to illustrate and understand concepts we presented
- To easily differentiate between theory and exercises, there are two templates
- This is a theory slide!

Lecture and exercises

- This is an exercise slide!

Basic concepts

Goal of this section

- Cluster access using `ssh`
- Data transfers using `scp` and/or `rsync`
- Software modules on the cluster
- Code compilation
- Debugging

Connection to the cluster

- Secure shell or better known as SSH

```
$> man ssh
    ssh (SSH client) is a program for logging into a remote machine and
    for executing commands on a remote machine. It is intended to
    provide secure encrypted communications [...]. X11 connections [...]
    can also be forwarded over the secure channel.
```

- Basic usage

```
$> ssh <user>@<host>
```

<user> is your GASPAR username, <host> is any of {helvetios,izar,jed}.epfl.ch

- Example

```
$> ssh jdoe@helvetios.epfl.ch
password: *****
```

Connection to the cluster

Optional step

- The alternative to password is to use a cryptographic key pair

```
$> ssh-keygen -b 4096 -t rsa
[Follow the instructions]
$> ssh-copy-id jdoe@helvetios.epfl.ch
```

- `ssh-keygen` generates a public/private key pair
- By default, they are found in `~/.ssh`
  - `id_rsa.pub` is the public key and can be shared with anyone (`ssh-copy-id` copies it to the remote)
  - `id_rsa` is the private key and it is **SECRET**!

Data transfer

- We are working remotely and need to get your data back locally
- There are two main commands:

```
$> man scp
    scp copies files between hosts on a network. It uses ssh for data transfer, and
    ↪  uses the same authentication and provides the same security as ssh.
```

```
$> man rsync
    Rsync is a fast and extraordinarily versatile file copying tool. It can copy
    locally, to/from another host over any remote shell, or to/from a remote rsync
    daemon. [...] It is famous for its delta-transfer algorithm, which reduces the
    amount of data sent over the network by sending only the differences between the
    source files and the existing files in the destination. Rsync is widely used for
    backups and mirroring and as an improved copy command for everyday use.
```

- Similar usage pattern. The path on a remote host is written `hostname:/path/to/file`. For example

```
$> scp jdoe@helvetios.epfl.ch:src/myCode/file.c src/
```

Software modules

A tool to organize your environment

- HPC clusters are particular because many software and versions of them are installed alongside
- We need a tool to make the software easily available to everyone
- The main tools used today are `environment-modules` and `Lmod`
- At SCITAS, we chose `Lmod` (we'll see later why)

Software modules

A tool to organize your environment

- HPC clusters are particular because many software and versions of them are installed alongside
- We need a tool to make the software easily available to everyone
- The main tools used today are `environment-modules` and `Lmod`
- At SCITAS, we chose `Lmod` (we'll see later why)

- Those tools package different software and their configurations into *modules*
- When you need to use a software, you need to *load* the corresponding module
- Examples of (made-up) modules:
  - `intel-19.0.2`: provides Intel compiler version 19.0.2
  - `data-analysis`: provides tools for data-analysis such as Python with different packages, and Matlab

- Lmod is called using the `module` command followed by an action:
  - `avail`: print a list of available modules
  - `load/unload <modules>`: load/unload the `<modules>`
  - `purge`: unload all modules
  - `swap <module1> <module2>`: swap `<module1>` for `<module2>`
  - `list`: print a list of currently loaded modules
  - `spider <module>`: print all possible versions of `<module>`
  - `show`: print the module configuration
  - `save/restore <name>`: save/restore current module collection under `<name>`
  - `help`: print help
- Many of those commands are also available in `environment-modules`

- Lmod supports a hierarchical software stack
- When you switch a module, it will automatically reload the ones depending on it
- You need to load a compiler, and an MPI and BLAS implementation to have access to all modules

Compilation

0100101110101001010...

- A computer only understands ON and OFF states (1 and 0)
- It would be very inconvenient for us to code in binary
- We therefore use different levels of abstraction (languages), *e.g.* C, C++, Fortran
- We need a translator!

- Translation is made by a compiler in 4 steps

  Preprocessing Format source code to make it ready for compilation (remove comments, execute preprocessing directives such as `#include`, etc.)

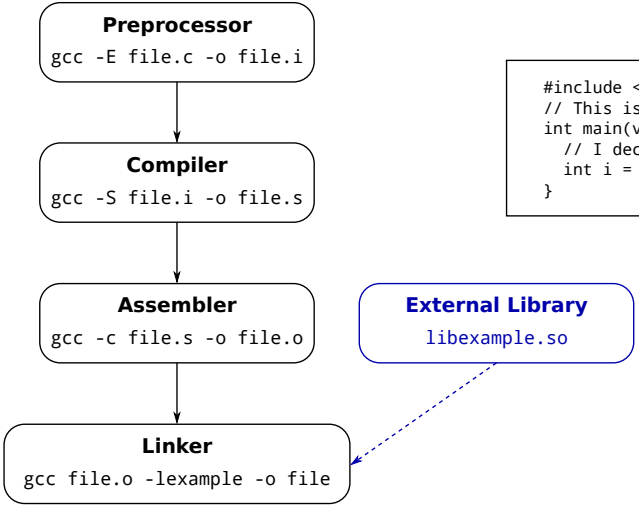
  Compiling Translate the source code (C, C++, Fortran, etc) into assembly, a very basic CPU-dependent language

  Assembly Translate the assembly into machine code and store it in object files

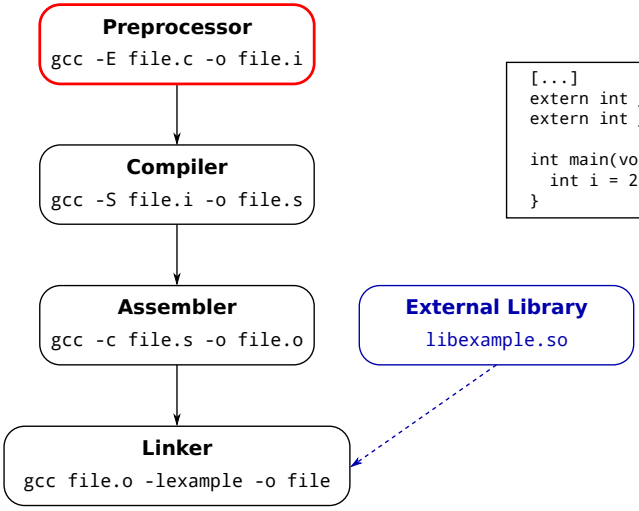
  Linking Link all the object files into one executable

- In practice, the first three steps are combined together and simply called "compiling"

The four compilation steps (visually)




```
#include <stdio.h>
// This is my main function
int main(void) {
  // I declare i to be equal to 2
  int i = 2;
}
```

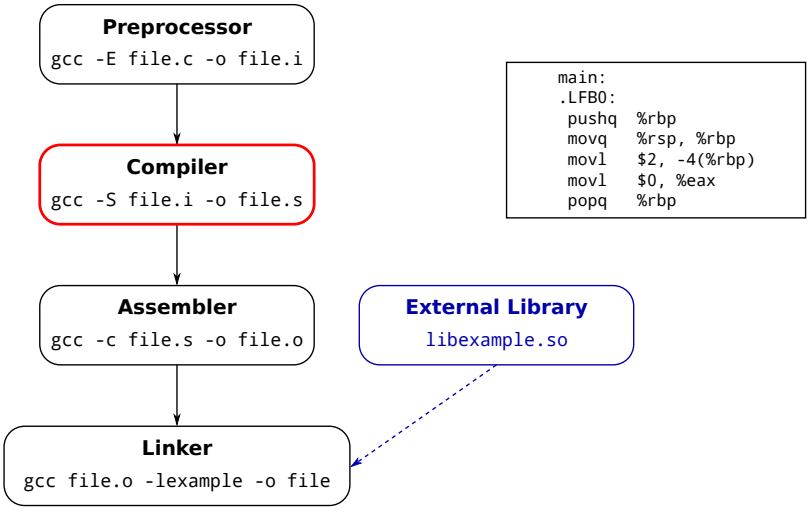



```
[...]
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);

int main(void) {
  int i = 2;
}
```

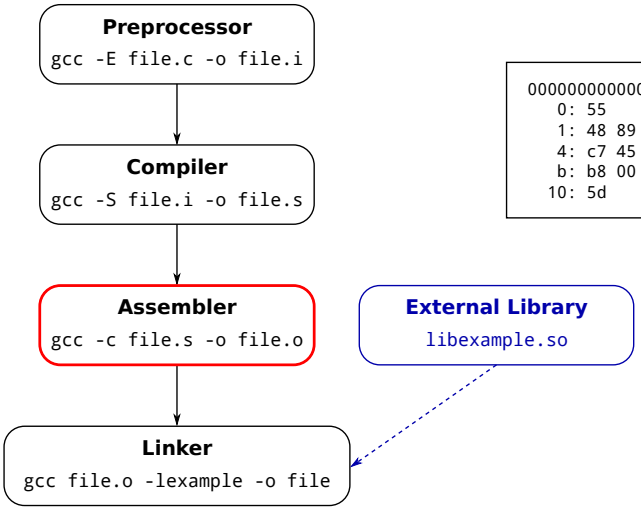
Compilation

The four compilation steps (visually)

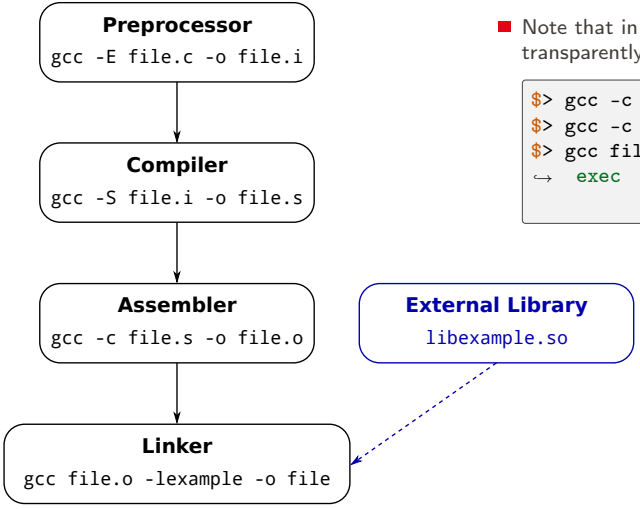


Compilation

The four compilation steps (visually)




```
0000000000000000 <main>:
   0: 55
   1: 48 89 e5
   4: c7 45 fc 02 00 00 00
   b: b8 00 00 00 00
  10: 5d
```



- Note that in reality, everything is done transparently

```
$> gcc -c file_1.c
$> gcc -c file_2.c
$> gcc file_1.o file_2.o -lexample -o
↪  exec
```

- Why bother debugging?
  - Studies[1] show $\sim$ 20 bugs/kloc in industry codes
  - You don't want to find a bug when on a deadline
- Only optimize a correct code

- There are different types of bugs:

Syntax error	A code keyword is misspelled, *e.g.* `dobule` instead of `double`. The code doesn't compile and the compiler tells you where is the error.
Runtime error	Division by 0 (fpe), out of bound access (seg. fault), etc. The code compiles fine, but will (most likely) crash at runtime.
Logical errors	Mistake that leads to an incorrect or unexpected behavior. You want to compute a distance from a velocity and a time, but you use an acceleration instead.

- Logical errors are clearly the most dangerous! The compiler doesn't complain and your code runs. You need to test it!

[1] Code Complete, S. McConnell

- Write tests (unit tests, application tests)!
- Write tests!
- Ask the compiler to complain (`-g -Wall -Wextra`)
- Use debuggers (gdb, TotalView, Alinea DDT)
- Use memory checkers (Valgrind, TotalView, Intel Inspector, `-fsanitize=address`)
- Don't use print statements (Heisenbug)

- In the **debugging** folder, make the executable.
- Execute the **./write** executable
- Run the code with **gdb**
  `$ gdb ./write`
- Run the code in gdb with `run` in gdb, it should stop at the line where the segfault happens.
- You can print the value of the variables with `print`
  `(gdb) print i`
  `(gdb) print data`
- At this point you should see the bug

- Execute the **./read** executable
- It might run fine but there is a bug.
- Run the code with **valgrind**
  `$ valgrind ./read`
- You can also compile with special sanitize options (this works only with gcc and clang)
  `$ CXXFLAGS=-fsanitize=address make`
  In this case the bound check is always done at execution.

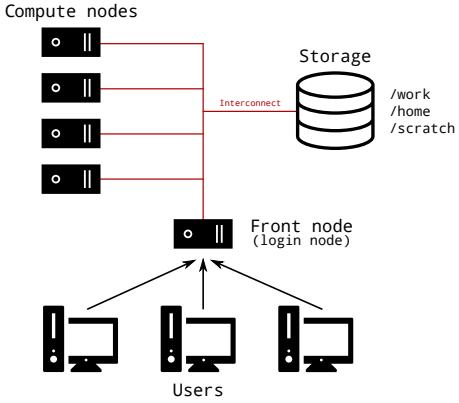
Cluster Architecture

Cluster Architecture

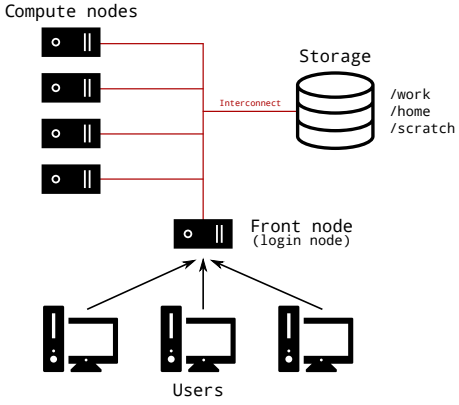
- The goal of this section is to understand what's under the cluster's hood
- In order to take full advantage of your computer, you have to understand how it works, what are the limits, etc.
- We'll go from the cluster level down to the core level

- An HPC cluster is composed of
  - Login node(s)
  - Compute nodes
  - Storage system
  - High performance interconnect
- 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

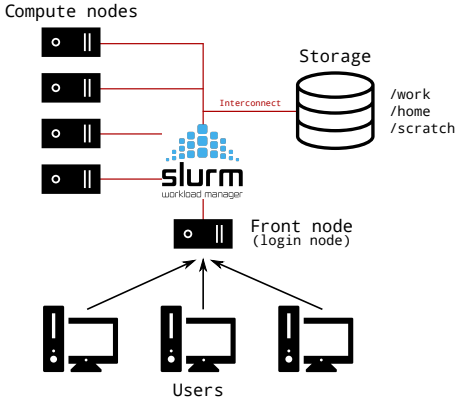
  Please, note that only `/home` and `/work` have backups! `/scratch` data can be erased at any moment!



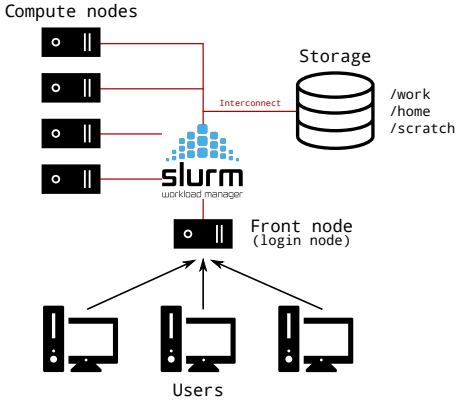
- Users do not run their calculations directly on the compute nodes
- A *scheduler* is used to ensure fair resource usage



- Users do not run their calculations directly on the compute nodes
- A *scheduler* is used to ensure fair resource usage
- At SCITAS, we use the Slurm scheduler



- Users do not run their calculations directly on the compute nodes
- A *scheduler* is used to ensure fair resource usage
- At SCITAS, we use the Slurm scheduler
- You submit your simulation and the resources you need to Slurm
- Slurm stores it into a queue and assigns it a starting time depending on many parameters
- Your job may not start right away and it is normal!



To submit a job

```
$> srun -A phys-743 ./my_program
```

-A / --account=<account> : name of your Slurm account
-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 / --ntasks=<n> : advise Slurm that this job will launch a maximum of *n* tasks
-c / --cpus-per-task=<ncpus> : advise Slurm that job will require **ncpus** per task
--mem=<size[units]> : specify the memory required per node

Need more help? Have a look at the documentation

Or you can put everything in a file called, *e.g.* my_simulation.job

```
#!/bin/bash -l
#SBATCH --account=phys-743
#SBATCH --time=01:10:00
#SBATCH --nodes=2
#SBATCH --ntasks=56

srun ./my_program
```

and submit the job with

```
$> sbatch my_simulation.job
```

To list all your jobs

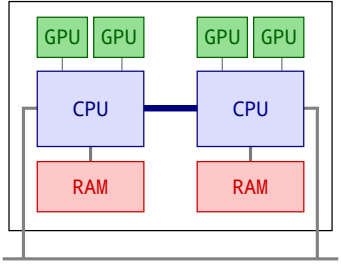
```
$> squeue -u <username>
$> squeue --me
```

To cancel a simulation

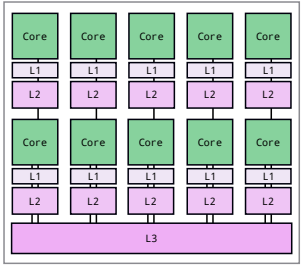
```
$> scancel <jobid>
```

The <**jobid**> can be found using **squeue**

- The compute node is the basic building bloc of a cluster
- It is composed of one or more CPU with RAM (memory) and eventually one or more accelerator, *e.g.* GPUs
- All the nodes are connected together with an interconnect



- The CPU is the "brain" of the node
- CPUs work in clock cycles; they are the "heart beat" of the CPU
- It is composed of cores and different levels of memories called caches
- There are usually three levels of cache called L1, L2, and L3



Event	Latency	Scaled	Capacity
1 CPU cycle	0.1 ns	1 s	–
L1 cache access	1 ns	10 s	kB
L2 cache access	1 ns	10 s	MB
L3 cache access	10 ns	1 min	MB
RAM access	100 ns	10 min	GB
Solid-state disk access	100 μs	10 days	TB
Hard-disk drive access	1–10 ms	1–12 months	TB

Let's go back to Frontier

- Second most powerful HPC cluster in the world according to the Top500 June 2024 list
- It is composed of 9472 compute nodes (74 racks/cabinets containing 64 blades holding 2 nodes each)
- Around 145 km of interconnect cables
- Power consumption of 22'786 kW
- Equivalent consumption as a city with ∼30'000 inhabitants
- In Lausanne, running Frontier would cost ∼110'000 CHF/d only for electricity!

Cluster Architecture

Summary of SCITAS' clusters

Helvetios

- CPU cluster
- 287 nodes each with
  - 2 Intel Xeon Gold 6140 @2.3 GHz with 18 cores each
  - 192 GiB of DDR3 RAM

Jed

- CPU cluster
- 419 nodes, 2 Intel Ice Lake Platinum with 36 cores each
  - 375 nodes with 512 GiB of DDR3 RAM
  - 42 nodes with 1 TiB of DDR3 RAM
  - 2 nodes with 2 TiB of DDR3 RAM

Izar

- CPU + GPU cluster
- 64 nodes each with
  - 2 Intel Xeon Gold 6230 @2.1 GHz with 20 cores each
  - 2 NVIDIA V100 PCIe 32 GiB GPUs
  - 192 GiB of DDR4 RAM
- 2 nodes each with
  - 2 Intel Skylake @2.1 GHz with 20 cores each
  - 4 NVIDIA V100 SMX2 32 GiB GPUs
  - 192 GiB of DDR4 RAM

Kuma

- CPU + GPU cluster
- 84 nodes each with
  - 2 AMD EPYC 9334 @2.7 GHz with 32 cores each
  - 4 NVIDIA H100 94 GiB GPUs
  - 384 GiB of RAM
- 20 nodes each with
  - 2 AMD EPYC 9334 @2.7 GHz with 32 cores each
  - 8 NVIDIA L40S 48 GiB GPUs
  - 384 GiB of RAM

Performance measurement

Goal of this section

- Key concepts to quantify performance
  - Metrics
  - Using a profiler
  - Scalings, speedup, efficiency
- Roofline model

Performance metrics

- How can we quantify performance?
- We need to define a means to measure it
- We will focus on the most interesting metrics for HPC

Performance metrics

- How can we quantify performance?
- We need to define a means to measure it
- We will focus on the most interesting metrics for HPC

- The first that comes in mind is *time*, *e.g.* time-to-solution
- Derived metrics: speedup and efficiency

Performance metrics

- How can we quantify performance?
- We need to define a means to measure it
- We will focus on the most interesting metrics for HPC

- The first that comes in mind is *time*, *e.g.* time-to-solution
- Derived metrics: speedup and efficiency

- Scientific codes do computations on floating point numbers
- A second metric is the number of *floating-point operations per second* (FLOP/s)

Performance metrics

- How can we quantify performance?
- We need to define a means to measure it
- We will focus on the most interesting metrics for HPC

- The first that comes in mind is *time*, *e.g.* time-to-solution
- Derived metrics: speedup and efficiency

- Scientific codes do computations on floating point numbers
- A second metric is the number of *floating-point operations per second* (FLOP/s)

- Finally, the *memory bandwidth* indicates how much data does your code transfers per unit of time

Profiling

A tool to measure various timings

- Where is my application spending most of its time?
  - (bad) measure time "by hand" using timings and prints
  - (good) use a tool made for this, *e.g.* Intel Amplifier, Score-P, gprof
- There are two types of profiling techniques
  - Sampling: you stop the code every now and then and check in which function you are
  - Code instrumentation: instructions are added at compile time to trigger measurements
- In addition to timings, profilers give you a lot more information on
  - Memory usage
  - Hardware counters
  - CPU activity
  - MPI communications
  - etc.

- For the purpose of this exercise, we will use MiniFE
  - 3D implicit finite-elements on an unstructured mesh
  - C++ mini application
  - `https://github.com/Mantevo/miniFE`
  - You don't need to understand what the code does!
- We will use Intel VTune, part of the OneAPI Base toolkit (free)

- Download miniFE
- Compile the basic version found in `ref/src`
- Profile the code using the hotspot analysis
- Open Intel VTune and select your timings
- Play around and find the 5 most time-consuming functions

- Download miniFE

```
$> git clone https://github.com/Mantevo/miniFE.git
$> cd miniFE
```

- Compile the basic version found in **ref/src**
  - You will need to load a compiler and an MPI library

```
$> module load intel intel-mpi intel-vtune
```

  - Change the `Makefile` to set `CXX=mpiicpc` and `CC=mpiicc` and compile

```
$> make
```

  - Make sure to compile your code with `-g -O3`

- Profile the code using

```
$> srun -n 1 amplxe-cl -collect hotspots -r prof_results -- ./miniFE.x -nx 128 -ny
↪  128 -nz 128
```

- This will profile for the "hotspots" and store the timings in `prof_results`
- You can have more info on the types of analysis with

```
$> amplxe-cl -h collect
```

- Open Intel VTune and select your timings

```
$> amplxe-gui prof_results/prof_results.amplxe
```

- Play around and find the 5 most time-consuming functions

- 50.0% of the time spent in matrix/vector multiplications
- 12.5% of time spent imposing boundary conditions
- etc.
- Does the problem size influence the timings?

- This time, we profile a problem of size $(16, 16, 16)$
- 13.6% of the time is spent opening libraries
- 13.6% of the time is spent initializing MPI
- etc.
- Depending on the problem size, different parts of the code will dominate

- Profile a code without bugs!
- Choose the right problem size (representative of your simulations)
- Focus on the functions taking the most time first
- If the profile is not explicit, try refactoring into smaller functions
  - Some profilers, *e.g.* ScoreP, let you define custom regions

Speedup and efficiency

- Two important metrics are derived from timings
- Compare timings with n processes, T_n, against the reference timing, T_{ref}

Speedup

$$S(n) = \frac{T_{\text{ref}}}{T_n}$$

Efficiency

$$E(n) = \frac{S(n)}{n}$$

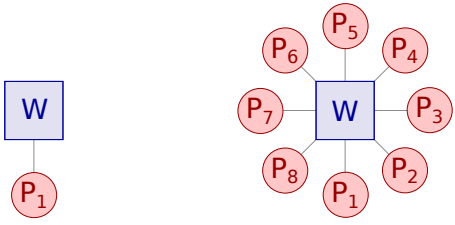
- We want $S(n)$ as close to n and $E(n)$ as close to 1 (100%) as possible

Strong scaling

- Scalings are a way to assess how well a program performs when adding computational resources
- Strong scaling: add resources, keep total amount of work constant

$$S(n) = \frac{T_1}{T_n}, \qquad E(n) = \frac{S(n)}{n} = \frac{T_1}{nT_n}$$

- Strong scaling is an indication on how much profitable it is to add resources to solve your problem

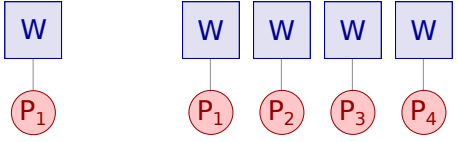


Weak scaling

- Weak scaling: add resources and maintain amount of work per resource constant

$$S(n) = \frac{nT_1}{T_n}, \qquad E(n) = \frac{S(n)}{n} = \frac{T_1}{T_n}$$

- Weak scalings are an indication on how well your code will perform on a bigger machine (and with a bigger problem)
- These scalings are always required for a proposal
  - For strong scalings the metric is speedup (how do I improve performance)
  - For weak scalings the metric is efficiency (how well performance is kept)



Amdahl's law

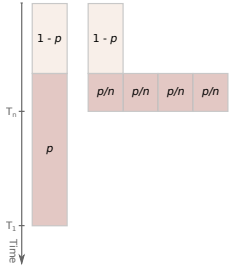
- Amdahl's law gives you an upper bound to the achievable speedup for a fixed problem size
- By definition it is a strong scaling analysis

Amdahl's law

- Amdahl's law gives you an upper bound to the achievable speedup for a fixed problem size
- By definition it is a strong scaling analysis

- Assume a fraction p of your code is (perfectly) parallel and timing with 1 process is T_1
- Timing with n processes is

$$T_n = (1-p)T_1 + \frac{p}{n}T_1 = \left[(1-p) + \frac{p}{n}\right] T_1$$



Amdahl's law

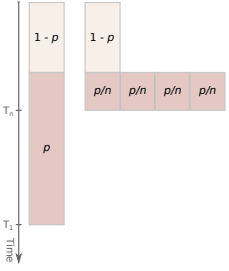
- Amdahl's law gives you an upper bound to the achievable speedup for a fixed problem size
- By definition it is a strong scaling analysis

- Assume a fraction p of your code is (perfectly) parallel and timing with 1 process is T_1
- Timing with n processes is

$$T_n = (1-p)T_1 + \frac{p}{n}T_1 = \left[(1-p) + \frac{p}{n}\right] T_1$$

- Speedup becomes

$$S(n) = \frac{T_1}{T_n} = \frac{1}{(1-p) + \frac{p}{n}}$$



Amdahl's law

- Amdahl's law gives you an upper bound to the achievable speedup for a fixed problem size
- By definition it is a strong scaling analysis

- Assume a fraction p of your code is (perfectly) parallel and timing with 1 process is T_1
- Timing with n processes is

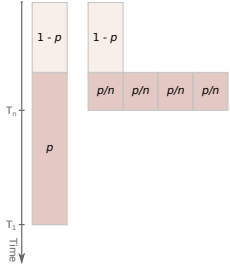
$$T_n = (1-p)T_1 + \frac{p}{n}T_1 = \left[(1-p) + \frac{p}{n}\right] T_1$$

- Speedup becomes

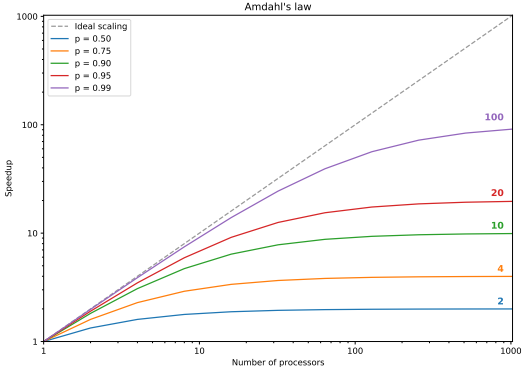
$$S(n) = \frac{T_1}{T_n} = \frac{1}{(1-p) + \frac{p}{n}}$$

- In the limit of infinite resources

$$\lim_{n\to\infty} S(n) = \frac{1}{1-p}$$

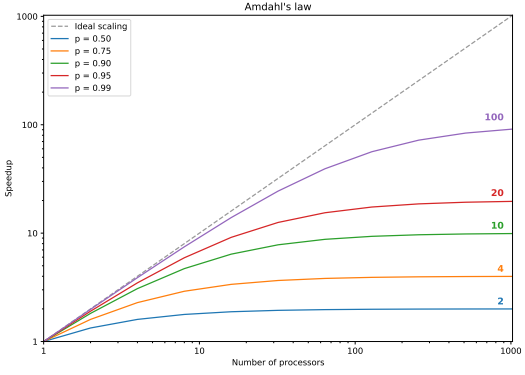


Amdahl's law



- Limited by the serial part (very sensitive)!
- Does this mean we cannot exploit large HPC machines?

Amdahl's law



- Limited by the serial part (very sensitive)!
- Does this mean we cannot exploit large HPC machines?
- No, in general with more resources, we simulate larger systems $\Rightarrow$ weak scaling (see Gustafson law)

FLOP/s and memory bandwidth

- FLOPs are floating point operations, *e.g.* $+, -, \times, \div$
- Can be evaluated by hand, dividing the number of operations by the running time

- Memory bandwidth measures the amount of data transferred by unit of time [B/s, KiB/s, MiB/s, GiB/s, ...]
- Can be measured by hand dividing the amount of data transferred by the running time

- In both cases, generally use tools such as PAPI, Tau, likwid, Intel Amplxe, STREAM, etc.

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!

- Each iteration has 2 FLOP (1 add and 1 mul) and there are `N = 1e8` iterations
- Our code $2 \cdot 10^8$ FLOP$/174.25 \cdot 10^{-3}$ s $= 0.001$ TFLOP/s
- Our hardware can achieve a theoretical peak performance of 1.16 TFLOP/s...

- Assume Intel Xeon Gold 6132 (Gacrux)

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- My code runs in 174.25 ms. It is amazingly fast!

- Each iteration has 2 FLOP (1 add and 1 mul) and there are `N = 1e8` iterations
- Our code $2 \cdot 10^8$ FLOP$/174.25 \cdot 10^{-3}$ s $= 0.001$ TFLOP/s
- Our hardware can achieve a theoretical peak performance of 1.16 TFLOP/s...

- Each iteration has 3 memory operations (2 loads and 1 store)
- Our code 2.23 GiB$/174.25 \cdot 10^{-3}$ s $= 12.82$ GiB/s
- Our hardware can achieve a theoretical memory bandwidth of 125 GiB/s...

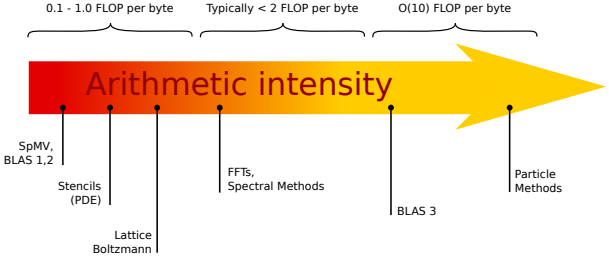
Roofline model

- How well am I exploiting the hardware resources?
- The roofline model is a performance model allowing to have an estimate to this question

Roofline model

- How well am I exploiting the hardware resources?
- The roofline model is a performance model allowing to have an estimate to this question

- Key concept: the arithmetic intensity, *AI*, of an algorithm is # FLOP/B of data transferred
- It measures data reuse



Roofline model

How to find arithmetic intensity

- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- There are 2 operations (1 add and 1 mul)
- Three 8-byte memory operations (2 loads and 1 store)
- The AI is then $2/24 = 1/12$

Roofline model

How to find arithmetic intensity

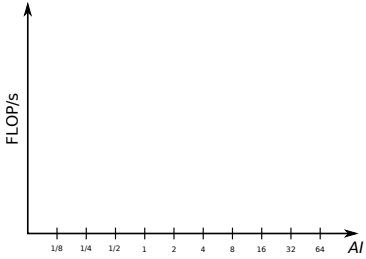
- For very simple algorithms, you can compute the AI
- Let's take back the DAXPY example

optimization/daxpy.cc

```
1 for (int i = 0; i < N; ++i) {
2   c[i] = a[i] + alpha * b[i];
3 }
```

- There are 2 operations (1 add and 1 mul)
- Three 8-byte memory operations (2 loads and 1 store)
- The AI is then $2/24 = 1/12$
- For more complex algorithms, use a tool, *e.g.* Intel Advisor

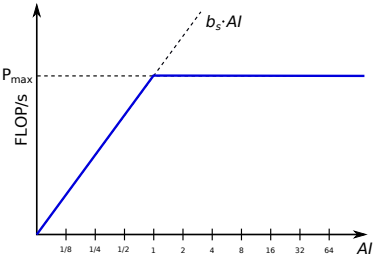
- Roofline model is plotted on **log-log scale**
  - x-axis is the *AI*
  - y-axis is FLOP/s



- Roofline model is plotted on **log-log scale**
  - x-axis is the AI
  - y-axis is FLOP/s
- The hardware limits are defined by

$$P = \min(P_{\max}, b_s \cdot AI)$$

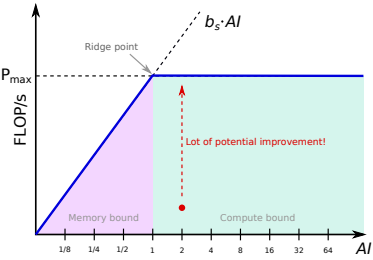
  - $P_{\max}$ is the CPU peak FLOP/s
  - AI is the intensity
  - b_s is the memory BW



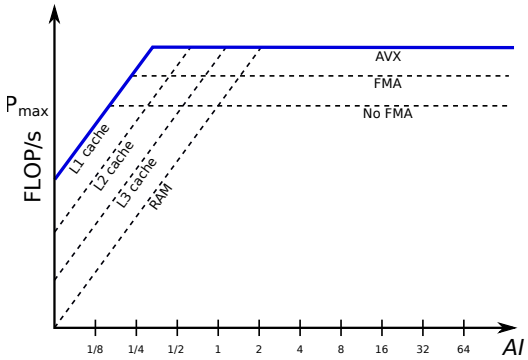
- Roofline model is plotted on **log-log scale**
  - x-axis is the AI
  - y-axis is FLOP/s
- The hardware limits are defined by

$$P = \min(P_{\max}, b_s \cdot AI)$$

  - $P_{\max}$ is the CPU peak FLOP/s
  - AI is the intensity
  - b_s is the memory BW



- Refinements can be made to the Roofline model
- Adding a memory hierarchy with caches
- Adding different levels of DLP (Data-Level parallelism)
- They give you hint on what to optimize for

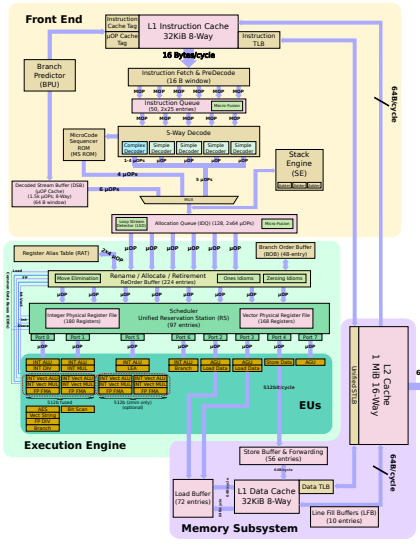


- Theoretical peak performance

  $P_{\max} =$ Number of FP ports (ILP)
  $\times$ flops/cycles (*e.g.* 2 for FMA)
  $\times$ vector size (DLP)
  $\times$ frequency (in GHz)
  $\times$ number of cores (TLP)

- Example: Intel Xeon Gold 6132

  $P_{\max} =$ 2 (ports)
  $\times$ 2 FLOP/c (2 for FMA)
  $\times \frac{512\,\text{bit (AVX512)}}{64\,\text{bit (double)}}$
  $\times$ 2.3 GHz
  $\times$ 14 (cores)
  $=$1.16 TFLOP/s



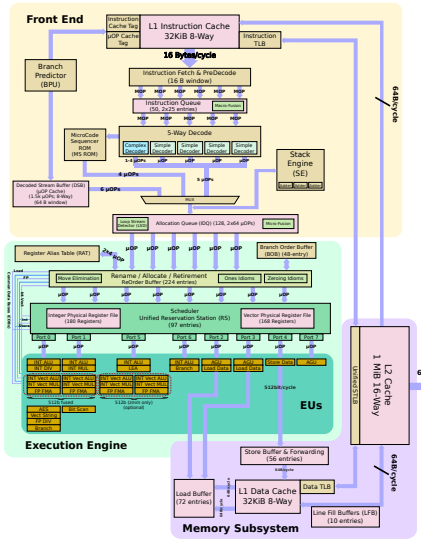
- Theoretical peak performance

  $P_{\max} =$ Number of FP ports (ILP)
  $\times$ flops/cycles (*e.g.* 2 for FMA)
  $\times$ vector size (DLP)
  $\times$ frequency (in GHz)
  $\times$ number of cores (TLP)

- Example: Intel Xeon Gold 6132

  $P_{\max} =$ 2 (ports)
  $\times$ 2 FLOP/c (2 for FMA)
  $\times \frac{512\,\text{bit (AVX512)}}{64\,\text{bit (double)}}$
  $\times$ 2.3 GHz
  $\times$ 14 (cores)
  $=$1.16 TFLOP/s

- Or use a software that estimates it



- Theoretical memory bandwidth of the memory

$$\begin{aligned} \mathrm{BW}_{\mathbf{max}} = \ & \text{Number of transfers per second} \\ & \times \text{Bus width} \\ & \times \text{Number of interfaces} \end{aligned}$$

- In general, we suppose that RAM matches CPU bandwidth (found on the CPU spec. list)
- Example: Intel Xeon Gold 6132

$$\begin{aligned} \mathrm{BW}_{\mathbf{max}} = \ & 2666\,\mathrm{MT/s}\ (\text{DDR4 2666}) \\ & \times 8\,\mathrm{B/T}\ (\text{64bit bus}) \\ & \times 6 \end{aligned}$$

  - 19.86 GiB/s for 1 channel
  - Maximum of 119.18 GiB/s

- Theoretical memory bandwidth of the memory

$$\begin{aligned} \mathrm{BW}_{\mathbf{max}} = \ & \text{Number of transfers per second} \\ & \times \text{Bus width} \\ & \times \text{Number of interfaces} \end{aligned}$$

- In general, we suppose that RAM matches CPU bandwidth (found on the CPU spec. list)
- Example: Intel Xeon Gold 6132

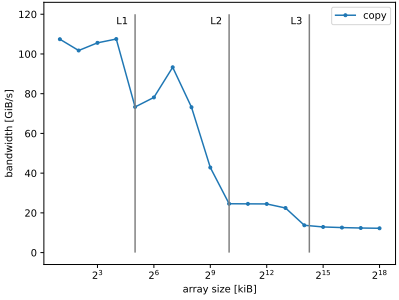
$$\begin{aligned} \mathrm{BW}_{\mathbf{max}} = \ & 2666\,\mathrm{MT/s}\ (\text{DDR4 2666}) \\ & \times 8\,\mathrm{B/T}\ (\text{64bit bus}) \\ & \times 6 \end{aligned}$$

  - 19.86 GiB/s for 1 channel
  - Maximum of 119.18 GiB/s
- Or use a software that estimates it
- A corollary from "theoretical" is that it is not achievable in practice!

Roofline model

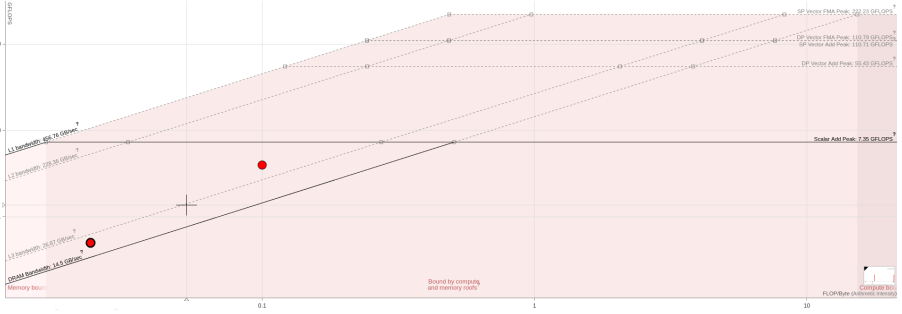
How to measure the actual values

- Peak performance measurement
  - Using a compute bound kernel
  - Using dgemm:
    1 core: 98.0 GFLOP/s
    14 cores: 965.0 GFLOP/s
- Bandwidth measurement
  - Using a memory bound kernel
  - Using stream (triad):
    1 core: 12.7 GiB/s
    6 core: 70.1 GiB/s
    9 core: 82.7 GiB/s



Roofline model

Intel Amplifier



Optimization

- We now have a pretty good idea of which part of the code to optimize
- Different options are possible (by order of complexity)
  1. Compiler and linker flags
  2. Optimized external libraries
  3. Handmade optimization (loop reordering, better data access, etc.)
  4. Algorithmic changes

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations
  - `-Ofast`: activate more aggressive options, *e.g.* `-ffast-math` (but can produce wrong results in some particular cases)

- Compilers have a set of optimizations they can do (if possible)
- You can find a list of options for GNU compilers on their doc
- Common options are:
  - `-O0`, `-O1`, `-O2`, `-O3`: from almost no optimizations to most optimizations
  - `-Ofast`: activate more aggressive options, *e.g.* `-ffast-math` (but can produce wrong results in some particular cases)
- Test your program with different options (`-O3` does not necessarily leads to faster programs)
- Note that the more optimization the longer the compilation time

- Do not re-invent the wheel!
- A lot of optimized libraries exist with different purposes (solvers, data structures, I/O, etc.). A few examples:
  - Solvers: PETSc, MUMPS, LAPACK, scaLAPACK, PARDISO, etc.
  - I/O: HDF5, ADIOS, etc.
  - Math libraries: FFTW, BLAS, etc.

- Sometimes, we cannot rely on compiler options or libraries and we must optimize "by hand"
- Usually, the goal is to rewrite the code in such a way that the compiler can optimize it
- Start by having a correct program before trying to optimize
- "Premature optimization is the root of all evil", D. Knuth

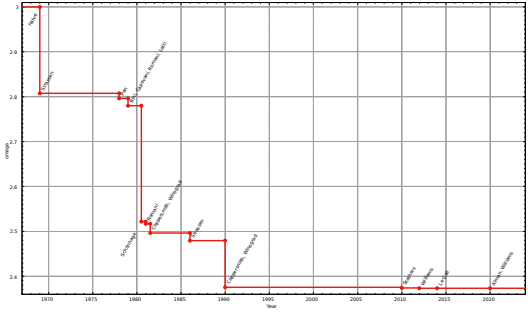
Pareto principle

The 80/20 rule

- General principle that states that 80% of the effect comes from 20% of causes
- Applies in many domains and especially in optimization
- 80% of the time is spent in 20% of your code
- Concentrate on those 20% and don't arbitrarily optimize

- Example of matrix/matrix multiplication. Graph shows complexity ($\mathcal{O}(n^\omega)$) for different algorithms

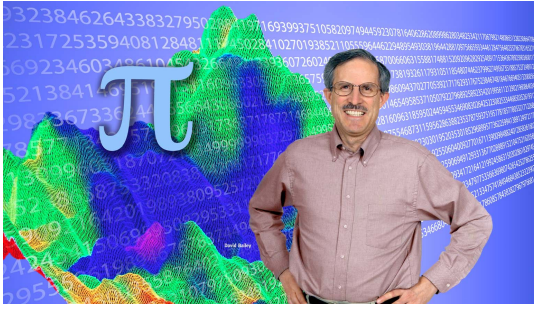
- Example of matrix/matrix multiplication. Graph shows complexity ($\mathcal{O}(n^\omega)$) for different algorithms

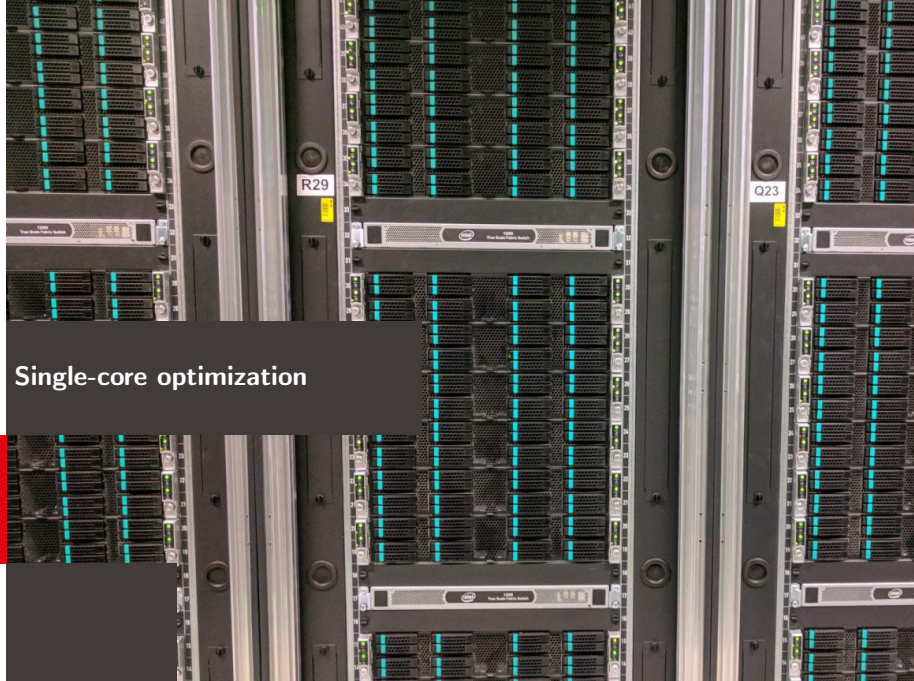


- Only when your code has *no bugs* and is *optimized*
- Are your ready to parallelize?
  1. Is it worth to parallelize my code? Does my algorithm scale?
  2. Performance prediction?
  3. Profiling?
  4. Bottelnecks?
  5. Which parallel paradigm should I use? What is the target architecture (SMP, cluster, GPU, hybrid, etc)?

In 1991, David H. Bailey published a famous paper: Twelve ways to fool the masses when giving performance results on parallel computers

6: Compare your results against scalar, unoptimized code on Crays.





Single-core optimization

- Better grasp how programming can influence performance
- We first review some basic optimization principles to keep in mind
- Deeper understanding of the working principles of the CPU
  - How data transfers are handled
  - Concept of vectorization

- Often, very simple changes to the code lead to significant performance improvements
- The following may seem trivial, but you would be surprised how often they could be used in scientific codes
- The main problem is that we often make a one-to-one mapping between the equations and the algorithm
- Note that a recent compiler will most likely do the work for you for small pieces of code.

Do less work

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = (alpha + sin(x)) * b[i];
3 }
```

```
1 double tmp = alpha + sin(x);
2 for (int i = 0; i < N; ++i) {
3   a[i] = tmp * b[i];
4 }
```

- Constant term is re-computed at every iteration of the loop
- Can be taken out of the loop and computed once

Avoid branches

```
1  for (i = 0; i < N; ++i) {
2    for (j = 0; j < N; ++j) {
3      if (j >= i) {
4        sign = 1.0;
5      } else {
6        sign = -1.0;
7      }
8      b[j] = sign * a[i][j];
9    }
10 }
```

```
1  for (i = 0; i < N; ++i) {
2    for (j = i; j < N; ++j) {
3      b[j] = a[i][j];
4    }
5    for (j = 0; j < i; ++j) {
6      b[j] =  -a[i][j];
7    }
8  }
```

- Avoid conditional branches in loops
- They can often be written differently or taken out of the loop

- To better understand the concepts behind caching, let's take the example of a librarian
- The first customer enters and asks for a book. The librarian goes into the huge storeroom and returns with the book when he finds it
- After some time, the client returns the book and the librarian puts it back into the storeroom
- A second customer enters and asks for the same book...
- This workflow can take a lot of time depending on how much customers want to read the same book

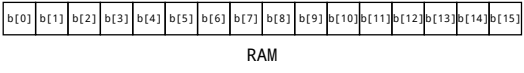
- Our librarian is a bit lazy, but clever. Since a lot of customers ask for the same book, he decides to put a small shelf behind his desk to temporarily store the books he retrieves.
- This way he can quickly grab the book instead of going to the storeroom.
- When a customer asks for a book, he will first look on his shelf. If he finds the book, it's a *cache hit* and he returns it to the customer. If not, it's a *cache miss* and he must go back in the storeroom.
- This is a very clever system, especially if there is *temporal locality*, i.e. if the customers often ask for the same books.
- Can he do better ?

- Oftentimes, our librarian see that people taking one book will go back and ask for the sequels of the book
- He decides to change a bit his workflow. Now, when he goes into the storeroom to retrieve a book, he comes back with a few of them, all on the same shelf
- This way, when the customer brings back a book and asks for the sequel, it is already present on the librarian shelf
- This workflow works well when there is *spatial locality*, i.e. when you ask for a book there is a significant chance that you will read the sequel

- Now, what is the link between our librarian and the CPU? They work in a similar fashion!
- When a load instruction is issued the L1 cache logic checks if data is already present. If yes, this is a *cache hit* and data can be retrieved very quickly. If no, this is a *cache miss* and the next memory levels are checked.
- If the data is nowhere to be found, then it is loaded from the main memory
- As for our librarian, not only the required data is loaded for each cache miss, but a whole *cache line*

- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

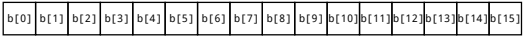


- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

Load b[0]:
Cache miss
Fetch cache line from RAM

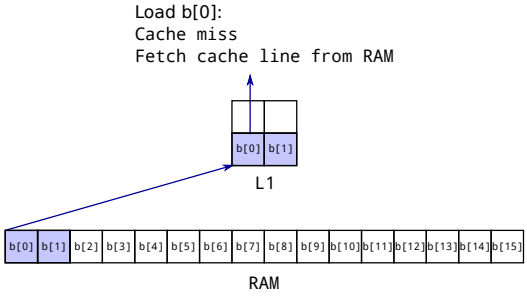
L1



RAM

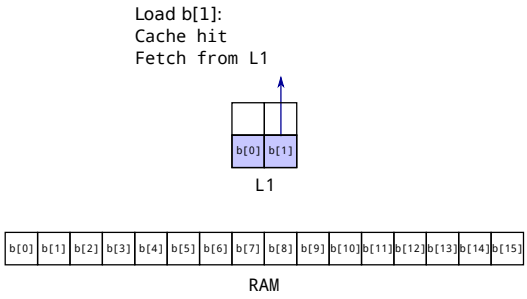
- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```



- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

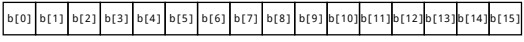


- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```

Load b[2]:
Cache miss
Fetch cache line from RAM

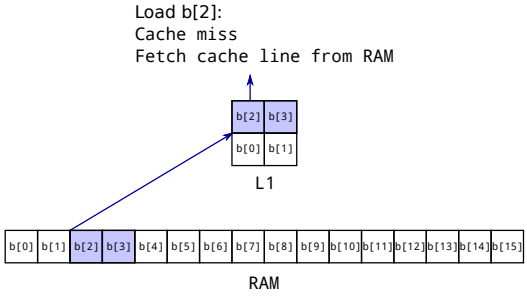
L1



RAM

- Simple vector/scalar multiplication
- Focus on data loading (**b[i]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

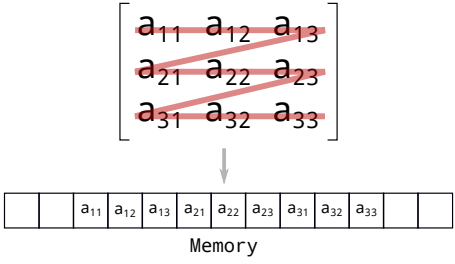
```
1 for (int i = 0; i < N; ++i) {
2   a[i] = alpha * b[i];
3 }
```



- How do we store ND arrays into memory?
- Memory is a linear storage. Arrays are stored contiguously, one element after the other.
- We have to choose a convention. Row major (C/C++) or column major (Fortran).
- Row major means that elements are stored contiguously according to the last index of the array. In column-major order, they are stored according to the first index.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

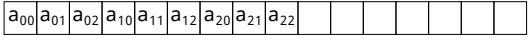
- How do we store ND arrays into memory?
- Memory is a linear storage. Arrays are stored contiguously, one element after the other.
- We have to choose a convention. Row major (C/C++) or column major (Fortran).
- Row major means that elements are stored contiguously according to the last index of the array. In column-major order, they are stored according to the first index.



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

L1



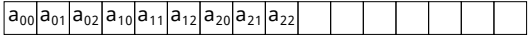
RAM

- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

Load a[0][0]:
Cache miss
Fetch cache line from RAM

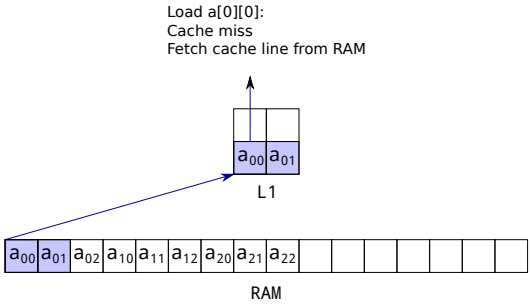
L1



RAM

- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

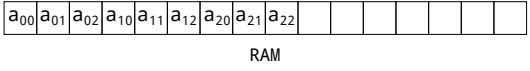
```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

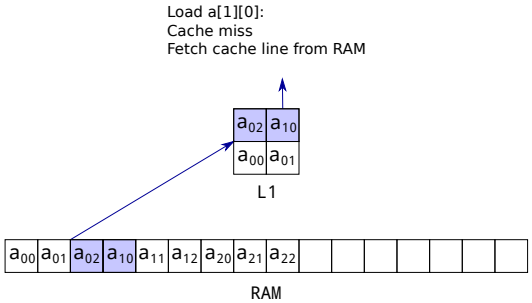
```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

Load a[1][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

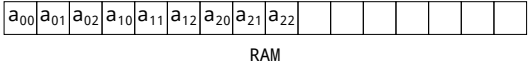
```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

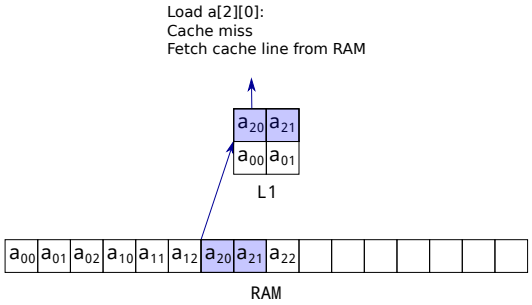
```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

Load a[2][0]:
Cache miss
Fetch cache line from RAM



- Focus on data loading (**a[i][j]**)
- Assume only one level of cache with a cache line of two doubles (16 bytes)

```
1 for (int j = 0; j < N; ++j) {
2   for (int i = 0; i < N; ++i) {
3     c[i] += a[i][j] * b[j];
4   }
5 }
```

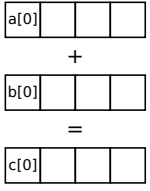


- Caches are small, but very fast memories
- Their purpose is to alleviate long latency and limited bandwidth of the RAM
- Data is fetched by group, called cache line, and stored into the different levels of cache
- In order to fully exploit caches, data in caches must be re-used as much as possible

- Avoid random memory accesses that case many cache misses and prefer contiguous access
- Be careful of the data types you use and how they are mapped onto memory

- Modern CPUs can apply the same operation to multiple data
- Special registers `xmm`, `ymm` and `zmm` holding 2, 4 or 8 doubles

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i]
}
```



- Modern CPUs can apply the same operation to multiple data
- Special registers `xmm`, `ymm` and `zmm` holding 2, 4 or 8 doubles

```
for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i]
}
```