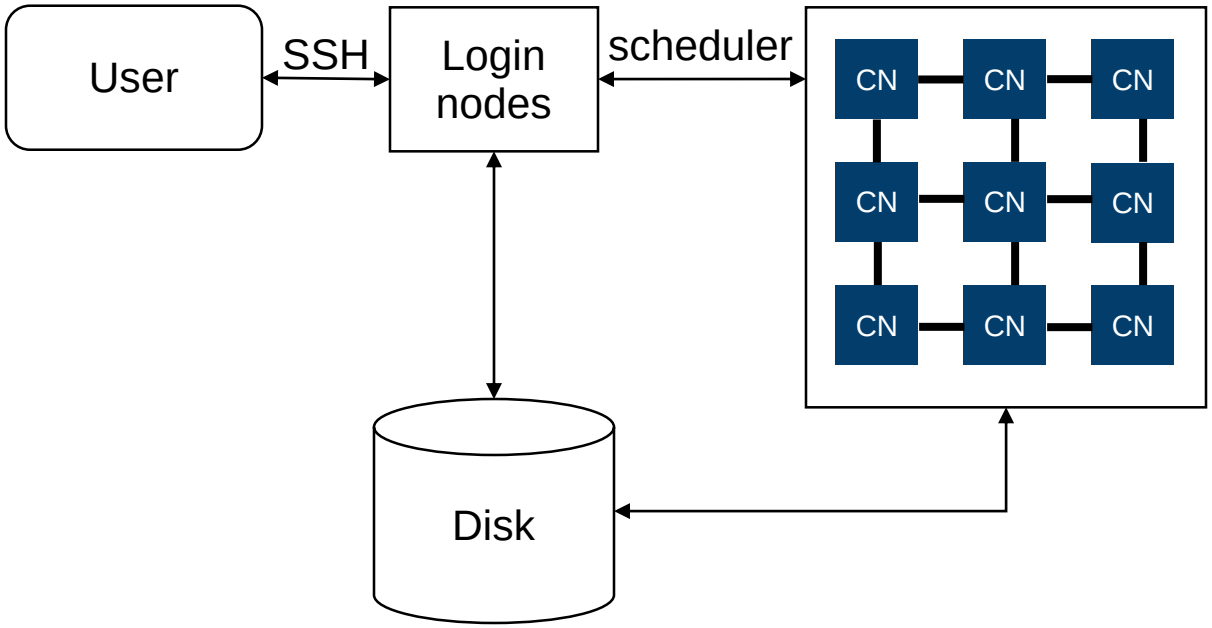
SINGLE NODE OPTIMISATION

PERFORMANCE AND TOOLS

10.07.2019 I ILYA ZHUKOV

HIGH-PERFORMANCE COMPUTER

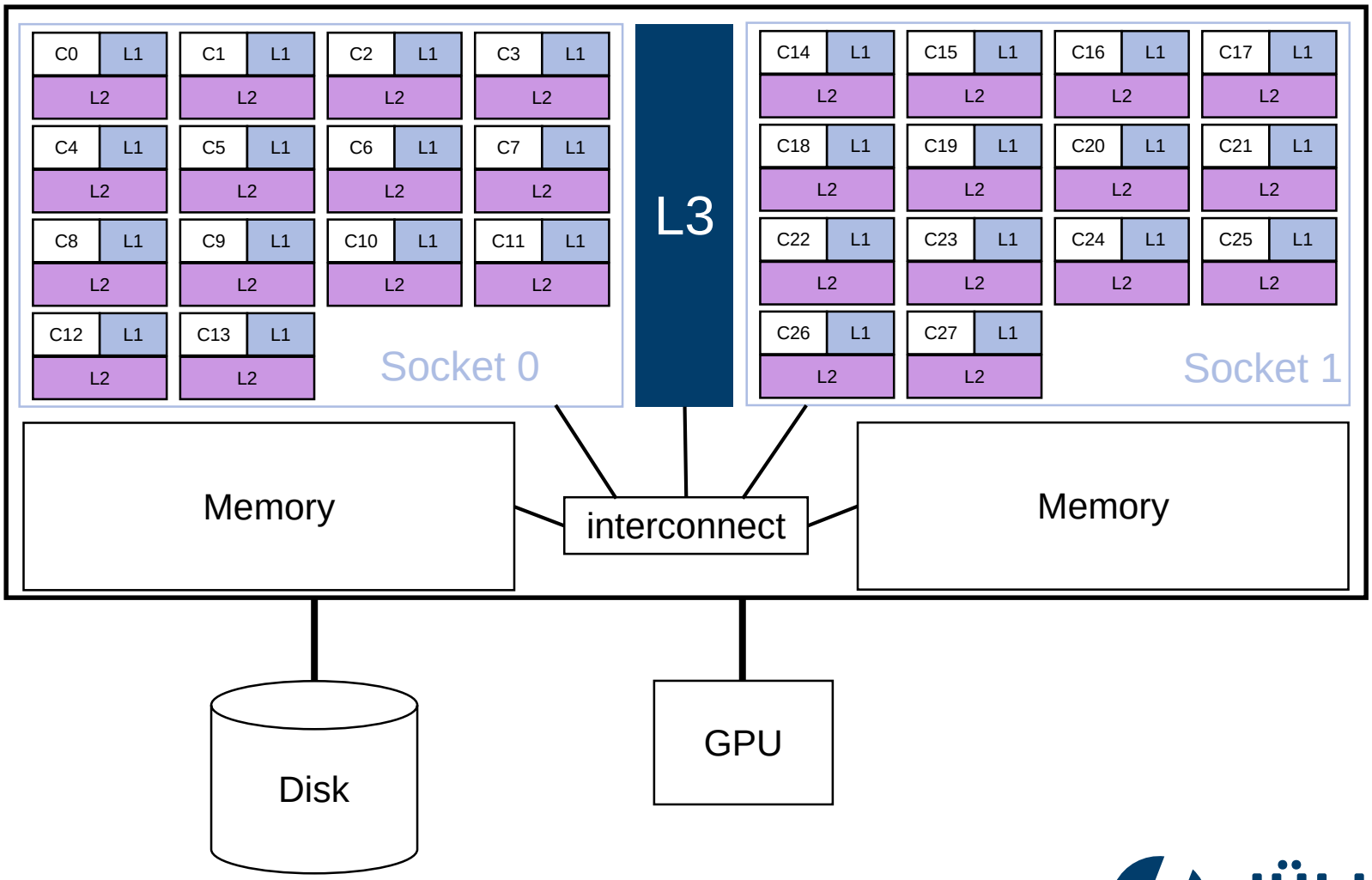
HPC building blocks



- Hardware
  - Login and compute nodes
  - Network
  - Disk
- Software
  - OS
  - Scheduler
  - Compilers
  - Libraries

NODES

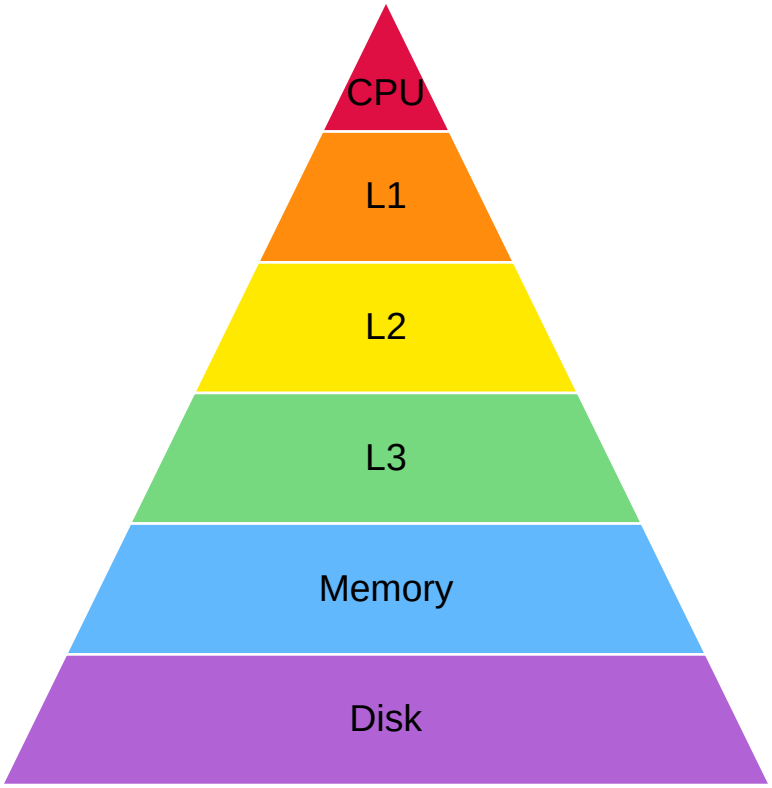
Intel® Xeon® Processor E5-2695 v3



JÜLICH Forschungszentrum

MEMORY HIERARCHY

Expensive
Fast
Small



Cheap
Slow
Large

RSM node on Bridges

Type	Size	Latency
L1	32KB	~ 4 cycles
L2	256KB	~ 11 cycles
L3	35MB	~ 60 cycles
RAM	128GB	~ 62cycles+100ns
Node storage	2x4TB	~ 1000 cycles

Locality

- Programs tend to use data and instructions with addresses near or equal to those they have used recently
- Temporal locality: recently referenced items are likely to be referenced again in the near future
- Spatial locality: Items with nearby addresses tend to be referenced close together in time

PARALLEL EXECUTION

Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
- **Simultaneous Multi-Threading (SMT)**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

HEALTH CHECK PROTOCOL

- Does my code produce correct results?
- Does my code have performance problems?
  - Which function in my code consumes the most wall clock time?
  - Does my application scale as expected?
  - Does my program suffer from load imbalance?
  - Is there a disproportionate time spent in communication or synchronization?
  - Is my application limited by resource bounds? (CPU, memory, I/O)
- What causes performance problem?
- How can I improve?

PREPARATION

Task 0

- Login to Bridges

  ssh -X -p 22 <user>@bridges.psc.edu

- Copy exercises

  cp /home/zhukov/ihpcss19/tutorial/mm.tar.gz $HOME

- Extract tarball
  - **tar -xvf mm.tar.gz**
- Investigate which CPU do you use on your laptop and Bridges?
  - Possible tools/utilities
    - **/home/zhukov/ihpcss18/tools/likwid/gcc_openmpi/bin/likwid-topology**
    - **lstopo-no-graphics, lstopo***
    - **hwloc-ls**
    - **cat /proc/cpuinfo**

    *not available on Bridges

```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU stepping:   2
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                2
Cores per socket:       14
Threads per core:       1
```

MATRIX-MATRIX MULTIPLICATION I

Problem description

- Let's consider simple problem

$$\begin{pmatrix} a_{0,0} & \cdots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,l-1} \end{pmatrix} \begin{pmatrix} b_{0,0} & \cdots & b_{0,m-1} \\ \vdots & \ddots & \vdots \\ b_{l-1,0} & \cdots & b_{l-1,m-1} \end{pmatrix} = \begin{pmatrix} c_{0,0} & \cdots & c_{0,m-1} \\ \vdots & \ddots & \vdots \\ c_{n-1,0} & \cdots & c_{n-1,m-1} \end{pmatrix}$$

where **A** is a n x l matrix and **B** is a l x m matrix and **C** computed as follows

$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$

- For simplicity reason consider square matrices, i.e. $n=l=m$
- For correctness checking assume B is an identity matrix, i.e. **AB=A**
- Example

$$\begin{pmatrix} 1 & 9 & 8 \\ 4 & -2 & 3 \\ 14 & 42 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 9 & 8 \\ 4 & -2 & 3 \\ 14 & 42 & 0 \end{pmatrix}$$

DO NOT REINVENT A WHEEL! USE LIBS!

Task 1 matrix-matrix multiplication with library

- Load GNU Scientific Library
  module load GSL
- Use GNU Scientific Library for matrix-matrix multiplication
  - **https://www.gnu.org/software/gsl/**
  - See **01_mm_libs/tasks/README**
  - To build: **make**
  - To run: **./mm <matrix_size>**
- Implement time measurement of actual computation with **gettimeofday**
  - **http://man7.org/linux/man-pages/man2/gettimeofday.2.html**
- Measure computation time for matrices sizes 1024, 2048, 4096 and note results

Wallclock time of matrix-matrix multiplication on Bridges

Library	1024	2048	4096
GSL	0.49	6.35	64.48
MKL serial	0.05	0.39	3.09

INCORRECT RESULTS?!

Use GDB!

- GDB: The GNU Project Debugger
  - **https://www.gnu.org/software/gdb/**
- Typical workflow
  - Compile application with -g flag
  - **gdb --args <executable> <list_of_args>**

GDB cheat sheet	
run	Starts application with given arguments
list	Shows the current or given source context. <filename>:<function>, <filename>:<line_number>
break	Create breakpoint L : function name, line LN , or FILE:LN
next	Go to next instruction (source line) but don‘t dive into functions.
step	Go to next instruction (source line), diving into function.
continue	Continue normal execution.
print	Print content of variable/memory location/register.
set var <variable_name>=<value>	Change the content of a variable to the given value.
thread <thread #>	Switch to <thread #>
info threads	Info about existing threads
bt	Print backtrace of all stack frames
<ENTER>	Execute the previously executed command again
quit	Exit from the debugger

Note: GDB is not perfect beyond single process

Alternatives: TotalView, DDT

SOMETHING WRONG WITH MEMORY?!

Use Valgrind!

- Valgrind - instrumentation framework for building dynamic analysis tools
  - **http://valgrind.org/**
- Typical workflow
  - **valgrind --tool=<tool> <executable>**
  - Frequently used tools
    - **memcheck** – memory leaks
    - **cachegrind** – cache usage profiling
    - **massif** – heap memory usage profiling
    - **callgrind** – call graph tracing
    - **drd** – data race condition detection
    - **helgrind** – deadlock/livelock detection

Memory leaks analysis with Valgrind

```
==13754==
==13754== HEAP SUMMARY:
==13754==     in use at exit: 2,097,152 bytes in 1 blocks
==13754==   total heap usage: 3 allocs, 2 frees, 6,291,456 bytes allocated
==13754==
==13754== LEAK SUMMARY:
==13754==    definitely lost: 2,097,152 bytes in 1 blocks
==13754==    indirectly lost: 0 bytes in 0 blocks
==13754==      possibly lost: 0 bytes in 0 blocks
==13754==    still reachable: 0 bytes in 0 blocks
==13754==         suppressed: 0 bytes in 0 blocks
```

WHICH IS TIME CONSUMING ROUTINE?

Use gprof!

- gprof – profiling tool
  - Part of binutils
  - Uses sampling and instrumentation
  - **https://sourceware.org/binutils/docs/gprof/**
- Typical workflow
  - compile/link with **-pg** option
  - Set output file (by default gmon.out)
    - **export GMON_OUT_PREFIX=<gprof_output_file>**
  - To see profile and callpath
    - **gprof <executable> <gprof_output_file>**
  - To see only profile
    - **gprof -p -b <executable> <gprof_output_file>**
  - To see only callpath
    - **gprof -q -b <executable> <gprof_output_file>**

Flat profile with gprof

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
100.18	44.87	44.87	1	44.87	44.87	compute
0.09	44.91	0.04	1	0.04	0.04	assign
0.00	44.91	0.00	1	0.00	0.00	allocate
0.00	44.91	0.00	1	0.00	0.00	check_results
0.00	44.91	0.00	1	0.00	0.00	free_memory

Callpath with gprof

index	% time	self	children	called	name
[1]	100.0	0.00	44.91		main [1]
		44.87	0.00	1/1	compute [2]
		0.04	0.00	1/1	assign [3]
		0.00	0.00	1/1	allocate [4]
		0.00	0.00	1/1	check_results [5]
		0.00	0.00	1/1	free_memory [6]

		44.87	0.00	1/1	main [1]
[2]	99.9	44.87	0.00	1	compute [2]

		0.04	0.00	1/1	main [1]
[3]	0.1	0.04	0.00	1	assign [3]

		0.00	0.00	1/1	main [1]
[4]	0.0	0.00	0.00	1	allocate [4]

		0.00	0.00	1/1	main [1]
[5]	0.0	0.00	0.00	1	check_results [5]

		0.00	0.00	1/1	main [1]
[6]	0.0	0.00	0.00	1	free_memory [6]

Note: gprof is not reliable for multithreaded applications

Alternatives: Score-P, TAU, Extrae, Vtune, HPCToolkit

I WANT TO KNOW EVEN MORE!

Use perf!

- perf - Linux profiling with performance counters
  - **https://perf.wiki.kernel.org/index.php/Tutorial**
- Typical workflow
  - See available metrics
    - **perf list**
  - Collect metrics
    - Typical way
      - **perf stat -e <metrics>,<metric>,... <executable> <list_of_args>**
    - Detailed mode
      - **perf stat -d <executable> <list_of_args>**
  - Collect profile
    - **perf record <executable> <list_of_args>**
  - Visualize profile
    - **perf report**

Performance counters with perf

```
perf stat -e instructions,ref-cycles ./mm 1024
Square matrix multiplication AxB with size: 1024 Repetitions: 1
Memory allocation: 0.00s
Set matrix values: 0.01s
# 0 matrix-matrix multiplication: 2.87s
AVG time: 2.87s MIN time: 2.87s
Program terminated SUCCESSFULLY
Free memory: 0.00s

 Performance counter stats for './mm 1024':

    10.833.024.194      instructions:u
     7.164.501.650      cycles:u

       2,888547659 seconds time elapsed
```

Alternatives: PAPI, likwid

MATRIX-MATRIX MULTIPLICATION

Typical implementation. Only for learning!!!

- The typical implementation for square matrices uses three nested loops

```
for ( i = 0; i < size; i++ )
  for ( j = 0; j < size; j++ )
    for ( k = 0; k < size; k++ )
      c[i * size + j] += a[i * size + k] * b[k * size + j];
```

NAÏVE IMPLEMENTATION

Task 2 matrix-matrix multiplication

- See **02_mm_naive/tasks/README**
- Find error with **gdb**
- Find and fix memory leak with **Valgrind**
- Profile application with **gprof**
- Measure execution time for matrices sizes 1024, 2048 and note results
  - Compare results with GSL implementation

Wallclock time of matrix-matrix multiplication on Bridges

	1024	2048	4096
GSL	0.49	6.35	64.48
MKL serial	0.05	0.39	3.09
Naïve	3.15	52.56	794.26

MATRIX-MATRIX MULTIPLICATION

Typical implementation. Only for learning!!!

- The typical implementation for square matrices uses three nested loops

```
for ( i = 0; i < matrix_size; i++ )
  for ( j = 0; j < matrix_size; j++ )
    for ( k = 0; k < matrix_size; k++ )
      c[i * matrix_size + j] += a[i * matrix_size + k] * b[k * matrix_size + j];
```

MATRIX-MATRIX MULTIPLICATION

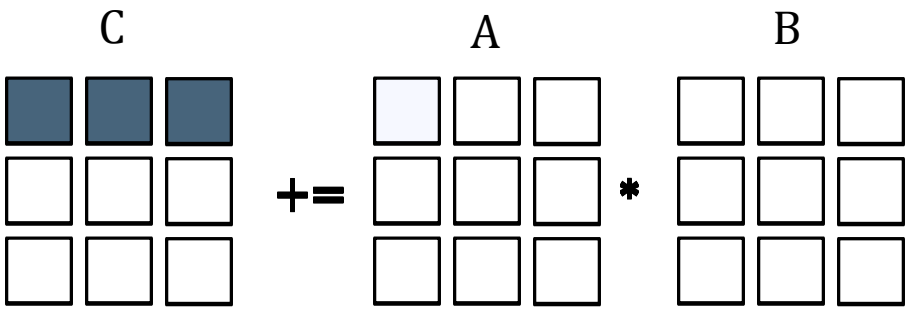
```
perf stat -d ./mm 1024 1
Square matrix multiplication AxB with size: 1024 Repetitions: 1
Memory allocation: 0.00s
Set matrix values: 0.01s
# 0 matrix-matrix multiplication: 3.02s
AVG time: 3.02s MIN time: 3.02s
Program terminated SUCCESSFULLY
Free memory: 0.00s

 Performance counter stats for './mm 1024 1':

       3036,334190      task-clock:u (msec)       #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 K/sec
                 0      cpu-migrations:u          #    0,000 K/sec
               932      page-faults:u             #    0,307 K/sec
     8.680.514.276      cycles:u                  #    2,859 GHz                      (49,95%)
     9.749.381.160      instructions:u            #    1,12  insn per cycle           (62,46%)
     1.093.489.871      branches:u                #  360,135 M/sec                    (62,48%)
         1.090.744      branch-misses:u           #    0,10% of all branches          (62,52%)
     2.169.019.648      L1-dcache-loads:u         #  714,355 M/sec                    (62,47%)
     1.113.839.847      L1-dcache-load-misses:u   #   51,35% of all L1-dcache hits    (25,00%)
     1.073.773.926      LLC-loads:u               #  353,642 M/sec                    (24,97%)
            71.971      LLC-load-misses:u         #    0,01% of all LL-cache hits     (37,45%)

       3,037077084 seconds time elapsed
```

Ideal IPC for some CPUs: https://en.wikipedia.org/wiki/Instructions_per_cycle

MATRIX-MATRIX MULTIPLICATION

- Naïve implementation

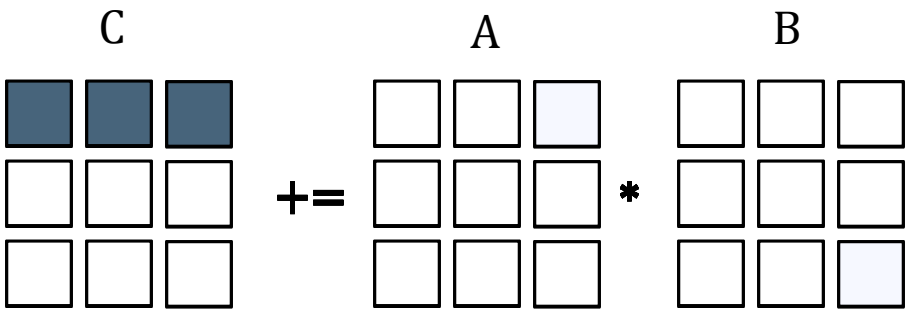
```
for ( i = 0; i < matrix_size; i++ )
  for ( j = 0; j < matrix_size; j++ )
    for ( k = 0; k < matrix_size; k++ )
      c[i * size + j] += a[i * matrix_size + k] * b[k * matrix_size + j];
```



MATRIX-MATRIX MULTIPLICATION

- Interchange loop to improve data usage

```
for ( i = 0; i < matrix_size; i++ )
  for ( k = 0; k < matrix_size; k++ )
    for ( j = 0; j < matrix_size; j++ )
      c[i * matrix_size + j] += a[i * matrix_size + k] * b[k * matrix_size + j];
```



LOOP INTERCHANGE

Task 3 loop interchange

- See **03_mm_reorder/tasks/README**
- Make loop interchange
- Did **perf** metrics improve?
- Measure execution time for matrices sizes 1024, 2048, 4096 and compare results

Wallclock time of matrix-matrix multiplication on Bridges

	1024	2048	4096
GSL	0.49	6.35	64.48
MKL serial	0.05	0.39	3.09
Naïve	3.15	52.56	794.26
Reorder	0.90	8.57	71.19

Can we do better?

- Cache blocking (see **04_mm_cache_block**)
- Allocate aligned memory (see **05_mm_mem_align**)

LOOP INTERCHANGE

```
perf stat -d ./mm 1024 1
Square matrix multiplication AxB with size: 1024 Repetitions: 1
Memory allocation: 0.00s
Set matrix values: 0.03s
# 0 matrix-matrix multiplication: 1.10s
AVG time: 1.10s MIN time: 1.10s
Program terminated SUCCESSFULLY
Free memory: 0.00s

 Performance counter stats for './mm 1024 1':

      1140,660331      task-clock:u (msec)       #    0,999 CPUs utilized
                0      context-switches:u        #    0,000 K/sec
                0      cpu-migrations:u          #    0,000 K/sec
              671      page-faults:u             #    0,588 K/sec
    2.964.541.975      cycles:u                  #    2,599 GHz                      (49,81%)
   10.839.832.875      instructions:u            #    3,66  insn per cycle           (62,42%)
    1.095.139.976      branches:u                #  960,093 M/sec                    (62,51%)
        1.091.065      branch-misses:u           #    0,10% of all branches          (62,66%)
    3.251.898.378      L1-dcache-loads:u         # 2850,891 M/sec                    (62,38%)
      135.735.582      L1-dcache-load-misses:u   #    4,17% of all L1-dcache hits    (24,89%)
        6.006.914      LLC-loads:u               #    5,266 M/sec                    (24,89%)
           56.727      LLC-load-misses:u         #    0,94% of all LL-cache hits     (37,34%)

      1,141555620 seconds time elapsed
```

PARALLEL EXECUTION

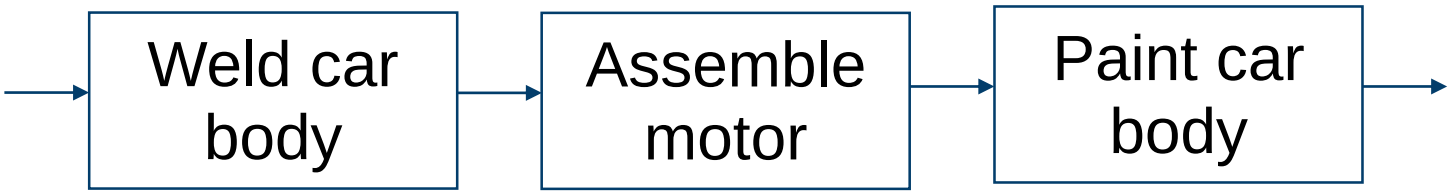
Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
- **Simultaneous Multi-Threading (SMT)**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

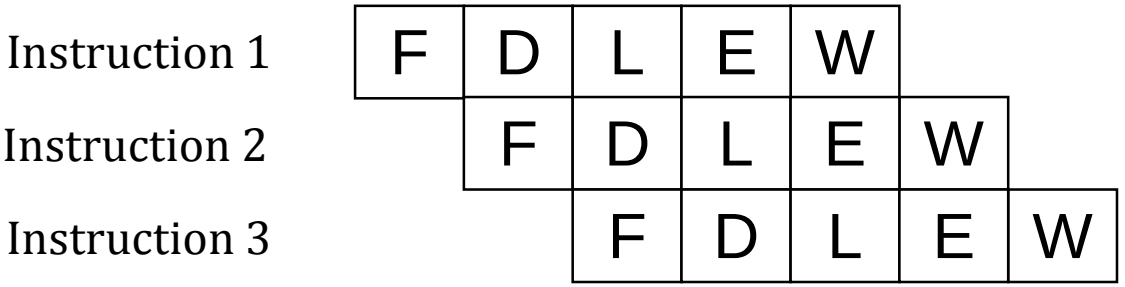
INSTRUCTION LEVEL PARALLELISM (ILP)

- ILP - parallelism among instructions from small code areas which are independent of one another, e.g. overlapping of instructions in a pipeline

Ford's assembly line (pipeline)



Modern processor pipeline (simplified)



F – fetch instruction
D – decode instruction
L – load operation
E – execute instruction
W – write result

PIPELINING: LOOP UNROLLING

- Let's take current version of matrix-matrix

```
for ( i = 0; i < matrix_size; i++ )
  for ( k = 0; k < matrix_size; k++ )
    for ( j = 0; j < matrix_size; j++ )
      c[i * matrix_size + j] += a[i * matrix_size + k] * b[k * matrix_size + j];
```

- Apply loop unrolling

```
for ( i = 0; i < matrix_size; i++ )
  for ( k = 0; k < matrix_size; k++ )
    tmp = a[i * matrix_size + k];
    for ( j = 0; j < matrix_size; j+=2 ) {
      c[i * matrix_size + j ] += tmp * b[k * matrix_size + j];
      c[i * matrix_size + j + 1] += tmp * b[k * matrix_size + j + 1];
    }
```

Note: better leave unrolling to compiler!

LOOP UNROLLING

Task 6 loop unrolling

- See **06_mm_loop_unroll/tasks/README**
- Try compiler options for unrolling. Did it improve runtime?
- Make loop unrolling
- Did **perf** metrics improve?
- Measure execution time for matrices sizes 1024, 2048, 4096 and compare results

Wallclock time of matrix-matrix multiplication on Bridges

	1024	2048	4096
GSL	0.49	6.35	64.48
MKL serial	0.05	0.39	3.09
Naïve	3.15	52.56	794.26
Reorder	0.90	8.57	71.19
Unroll	0.60	6.17	60.08

LOOP UNROLLING

```
perf stat -d ./mm 1024
Square matrix multiplication AxB with size: 1024 Repetitions: 1
Memory allocation: 0.00s
Set matrix values: 0.03s
# 0 matrix-matrix multiplication: 0.73s
AVG time: 0.73s MIN time: 0.73s
Program terminated SUCCESSFULLY
Free memory: 0.00s

 Performance counter stats for './mm 1024':

        762,635794      task-clock:u (msec)       #    0,999 CPUs utilized
                 0      context-switches:u        #    0,000 K/sec
                 0      cpu-migrations:u          #    0,000 K/sec
               671      page-faults:u             #    0,880 K/sec
     2.006.516.947      cycles:u                  #    2,631 GHz                      (49,95%)
     8.130.398.227      instructions:u            #    4,05  insn per cycle           (62,53%)
       554.523.479      branches:u                #  727,114 M/sec                    (62,64%)
         1.095.062      branch-misses:u           #    0,20% of all branches          (62,64%)
     2.185.664.447      L1-dcache-loads:u         # 2865,935 M/sec                    (62,25%)
       135.823.947      L1-dcache-load-misses:u   #    6,21% of all L1-dcache hits    (24,91%)
         2.814.570      LLC-loads:u               #    3,691 M/sec                    (24,91%)
            19.911      LLC-load-misses:u         #    0,71% of all LL-cache hits     (37,37%)

       0,763548697 seconds time elapsed
```

INSTRUCTION LEVEL PARALLELISM (ILP)

Best practice and tools

- Consider small chunk of code
- Try fully utilize pipelines as much as possible
  - Organize you data to avoid cache misses, e.g. AoS vs SoA
  - Avoid data hazards, e.g. data dependencies, branches etc.
- Identify your limits with Roofline model
- Consider IPC/CPI for performance measurements
- Play with compiler options, e.g. fast math, unrolling, etc.
- Use tools
  - **Perf, PAPI, Likwid** for hardware counters
  - **Valgrind** for memory accesses
  - **Intel Vtune, gprof** for profiles and analytics
  - **Intel Advisor** for roofline model

PARALLEL EXECUTION

Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
- **Simultaneous Multi-Threading (SMT)**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

VECTORISATION / SIMD

- Vector instructions exploit data level parallelism by operating on data items in parallel
  - E.g. vector multiplication

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} * \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

- In many cases compiler can automatically vectorise
- User can provide hints to compiler what should/can be vectorised
  - Simplify memory accesses, use pragmas

VECTORISATION

Task 7 SIMD

- See **07_mm_simd/tasks/README**
- Verify with compiler if the code was autovectorised
- Help compiler to autovectorise
- Did **perf** metrics improve?
- Measure execution time for matrices sizes 1024, 2048, 4096 and compare results

Wallclock time of matrix-matrix multiplication on Bridges

	1024	2048	4096
GSL	0.49	6.35	64.48
MKL serial	0.05	0.39	3.09
Naïve	3.15	52.56	794.26
Reorder	0.90	8.57	71.19
Unroll	0.60	6.17	60.08
Vectorise	0.39	4.22	53.10

VECTORISATION

```
perf stat -d ./mm 1024 1
Square matrix multiplication AxB with size: 1024 Repetitions: 1
Memory allocation: 0.00s
Set matrix values: 0.02s
# 0 matrix-matrix multiplication: 0.39s
AVG time: 0.39s MIN time: 0.39s
Program terminated SUCCESSFULLY
Free memory: 0.00s

 Performance counter stats for './mm 1024 1':

        408,457277      task-clock:u (msec)       #    0,998 CPUs utilized
                0      context-switches:u        #    0,000 K/sec
                0      cpu-migrations:u          #    0,000 K/sec
              675      page-faults:u             #    0,002 M/sec
    1.113.721.776      cycles:u                  #    2,727 GHz                      (49,93%)
    1.717.787.873      instructions:u            #    1,54  insn per cycle           (62,57%)
      293.350.812      branches:u                #  718,192 M/sec                    (62,57%)
        1.096.038      branch-misses:u           #    0,37% of all branches          (62,56%)
      568.390.665      L1-dcache-loads:u         # 1391,555 M/sec                    (62,06%)
      135.947.200      L1-dcache-load-misses:u   #   23,92% of all L1-dcache hits    (24,95%)
       54.911.235      LLC-loads:u               #  134,436 M/sec                    (24,96%)
            9.061      LLC-load-misses:u         #    0,02% of all LL-cache hits     (37,44%)

      0,409289193 seconds time elapsed
```

VECTORISATION

Best practice and tools

- Consider biggest loop
- Investigate with compiler options if it was vectorised
  - Avoid data dependencies
  - Avoid function calls from the loop
  - Use aligned memory if possible (see **05_mm_mem_align**)
  - Organize you data to simplify SIMD, e.g. AoS vs SoA
  - Force SIMD if possible, **#pragma omp simd**
- Consider your limits with Roofline model
- Use tools
  - **Perf, PAPI, Likwid** for hardware counters
  - **Intel Advisor, MAQAO** for detailed analytics

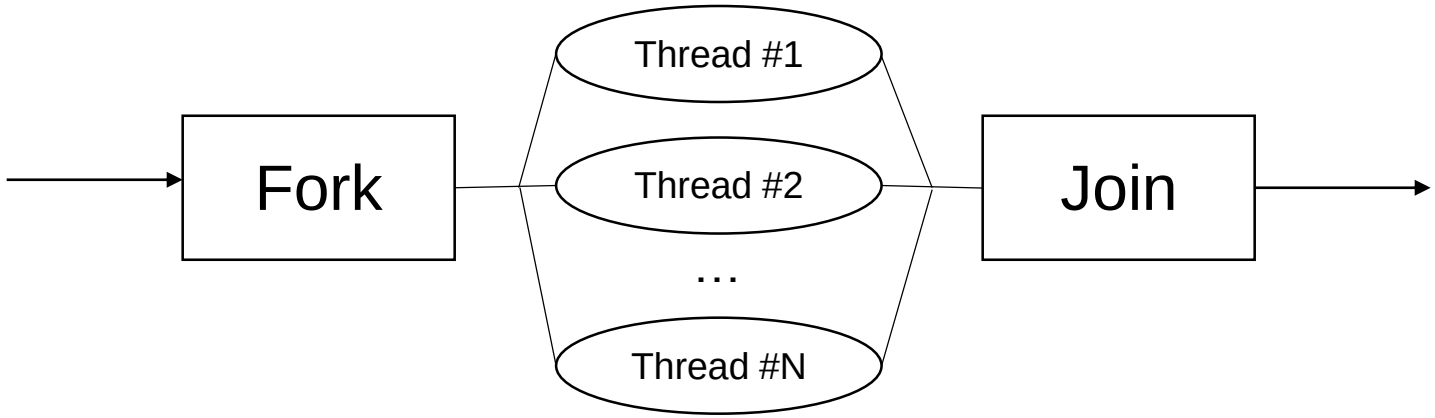
PARALLEL EXECUTION

Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
- **Simultaneous Multi-Threading (SMT)**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

THREAD LEVEL PARALLELISM (TLP)

- **Thread** is a smallest unit of processing that can be scheduled by operating system
- Each thread can be assigned to particular core (**pinning/binding**)
- Typical execution model:



- Popular Application Programming Interfaces (API):
  - POSIX Threads
  - OpenMP

PERFORMANCE METRICS

- A typical program has two categories of components
  - Inherently sequential sections: can't be run in parallel
  - Potentially parallel sections
- **Speedup**
  - typically S(N) < P

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

- **Parallel efficiency**
  - typically E(N) < 1

$$E(N,P) = \frac{S(N,P)}{P}$$

Where N is the size of the problem and P the number of processes

AMDAHL'S LAW

- Assumption
  - total problem size stays the same as the number of processors increases (**strong scaling**)
  - α is a completely serial fraction
  - parallel part is 100% efficient
- Parallel runtime

$$T(N,P) = \alpha T(N,1) + \frac{(1-\alpha)T(N,1)}{P}$$

- Parallel speedup

$$S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}}$$

- Our code is fundamentally limited by the serial fraction
  - α=0, S=P
  - α=0.1, max speedup is 10, e.g. S(N,10)=5.26, S(N,1000)=9.91

GUSTAFSON'S LAW

- Assumption
  - the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same (**weak scaling**)
  - α is a completely serial fraction
  - parallel part is 100% efficient
- Runtime on single process $T(N,1) = \alpha T(N,1) + (1-\alpha)PT(N,1)$
- Parallel runtime $T(N,P) = \alpha T(N,1) + (1-\alpha)T(N,1)$
- Parallel speedup $S(N,P) = \frac{T(N,1)}{T(N,P)} = \alpha + (1-\alpha)P$

- Limitation by the serial fraction becomes less
  - α=0, S=P
  - α=0.1, e.g. S(N,10)=9.10, S(N,1000)=900.10

OPENMP PARALLELISATION

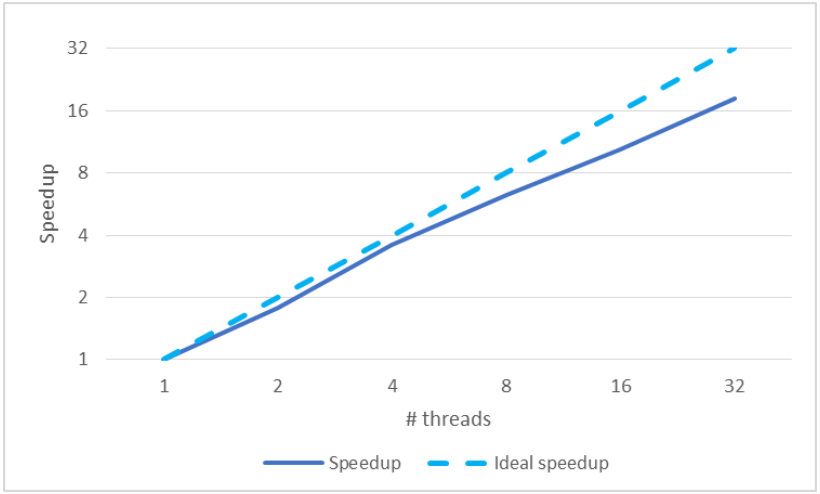
Task 8 OpenMP parallelisation

- See **08_mm_omp/tasks/README**
- Request allocation in interactive session
  interact -R performance -p RM -N 1 -t 1:00:00
- Apply OpenMP parallelisation
- Apply strong and weak scaling
- Compute speedup and efficiency and plot results

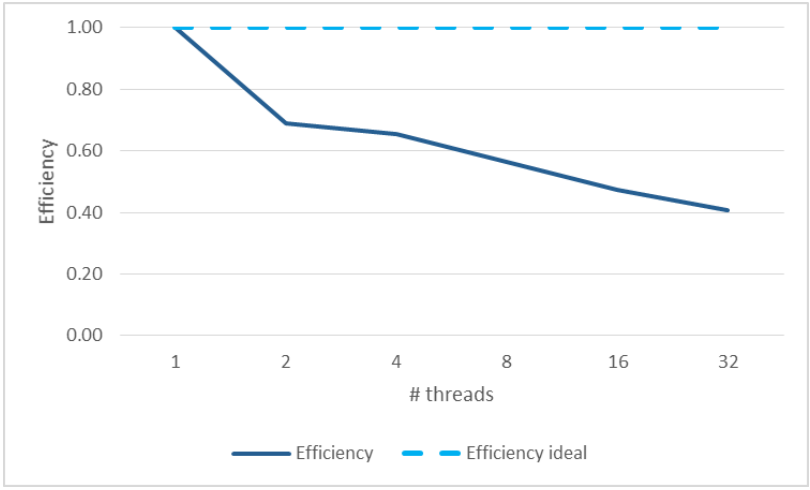
Compare OpenMP and MKL threaded (4096x4096)

- 28 OpenMP threads 4.35s
- MKL 0.16s (see **10_mm_mkl_thread**)

Strong scaling



Weak scaling



THREAD LEVEL PARALLELISM (TLP)

Best practice and tools

- Check threads binding
- Think about memory locality (see **09_mm_omp_numa**)
- Try fully utilize threads across program life time
- Avoid unnecessary synchronization
- Balance work across threads, e.g. scheduling policies
- Use tools
  - **Likwid**, **hwlock** for correct bindings
  - **ThreadSanitizer** for data races and deadlocks detection
  - **Perf**, **PAPI** for hardware counters
  - **Intel Vtune**, **Score-P**, **Cube**, **Scalasca** for detailed analytics

PARALLEL EXECUTION

Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
- **Simultaneous Multi-Threading (SMT)** **NOT SUPPORTED ON BRIDGES**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

SIMULTANEOUS MULTI-THREADING (SMT)

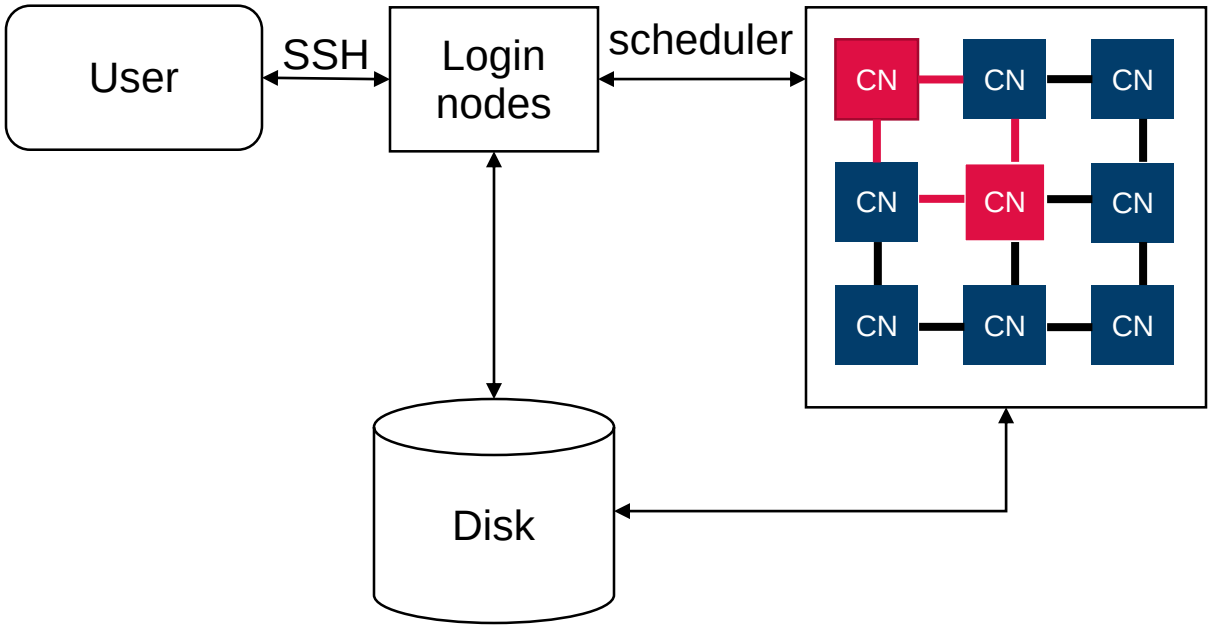
- **SMT** (aka Hyper-threading, hardware threading) architecture allows to execute simultaneously more than one thread per core
- TLP and ILP are exploited simultaneously
- Additional architectural requirements
  - Dynamic management of resources
  - Duplication of resources for each thread
  - Capability for instructions from multiple threads to commit
- OS sees SMT as multiple logical processors

PARALLEL EXECUTION

Take advantage of all possible levels of parallelism

- **Instruction Level Parallelism (ILP)**
  - Multiple instruction pipelines
- **Vector/SIMD instructions**
  - Single Instruction performing same operation on Multiple Data
- **Multiple cores per processor**
  - Processors with multiple cores
- **Simultaneous Multi-Threading (SMT)**
  - Concurrent execution of multiple instruction streams within same processor core
- **Multiple processors**
  - Multiple processors per nodes, multiple nodes per system

MULTIPLE PROCESSORS



- Communication over network is required
- Typical programming interface
  - MPI
  - SHMEM

- See naïve implementation **11_mm_omp_mpi**

MULTIPLE PROCESSORS

Best practice and tools

- Check MPI/threads binding
- Think about domain decomposition
- Think about communication patterns
- Avoid unnecessary synchronization/communication
- Balance work across MPIs/threads
- Use tools
  - **Perf**, **PAPI** for hardware counters
  - **Score-P**, **Cube**, **Scalasca** for detailed analytics
  - **Vampir** for trace visualisation

LITERATURE

- David A. Patterson and John L. Hennessy. **Computer Architecture: a Quantitative Approach.** Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
- Hager, Georg, and Gerhard Wellein. **Introduction to High Performance Computing for Scientists and Engineers**. Boca Raton, Fla.: CRC Press, 2011.
- Anderson, Matthew, C. Gordon Bell, Maciej Brodowicz, and Thomas Sterling. **High Performance Computing: Modern Systems and Practices.** Cambridge, MA: Elsevier Morgan Kaufmann Publishers, 2018.