Introduction to High Performance Computing

Lecture 1

Jakub Gałecki

Logistics

Classes: 10 x 3h

Integrated lecture + computer lab

Instructors:

- Jakub Gałecki
- Stanisław Gepner
- Grzegorz Gruszczyński

Materials

Contact info

Grading

Participants will be graded based on their final projects

Topics: chosen by participants, accepted by the instructors

Projects should be related to the participants' field of research/interest

Results should be computed on a cluster

Ideally MPI + chosen method of intra-node parallelism

Projects will be presented during the final class

Obligatory attendance

Motivation

Examples of HPC applications:

- Weather prediction
- Reasearch on the human brain
- Epidemiological modeling
- Genome research
- Nuclear research
- Machine learning & AI
- Aircraft design

Key indicators:

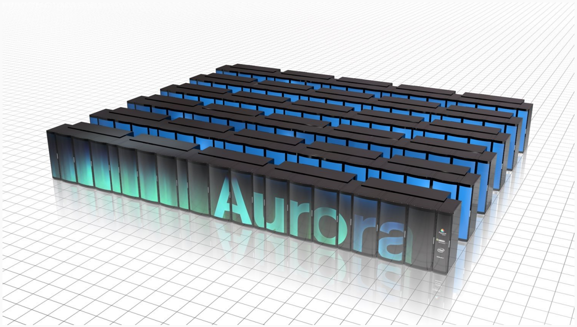
- Flops
- Flops/Watt

What does a modern supercomputer look like?

Network of small machines – each node roughly within an order of magnitude of our desktops

Example: Single node of Aurora (2023)

- CPU: 2x Intel®Xeon®Sapphire Rapids (56 cores each)
- GPU: 6x Intel®Ponte Vecchio

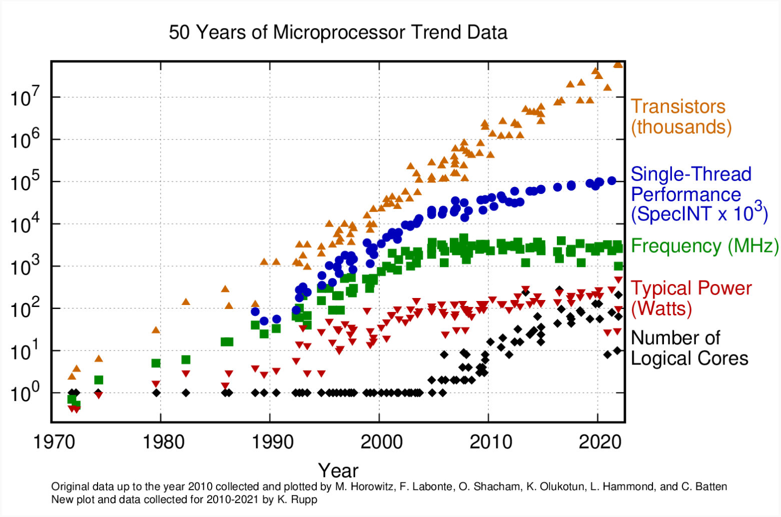


Largest supercomputers in the world

	Name	Country	Total Cores	Rmax [PFlop/s]	Power [kW]
1	Frontier	USA	8,730,112	1,102	21,100
2	Fugaku	Japan	7,630,848	442.01	29,899
3	LUMI	Finland	2,220,288	309.1	6.016
4	Leonardo	Italy	1,463,616	174.7	5,610
5	Summit	USA	2,414,592	148.6	10,096
6	Sierra	USA	1,572,480	94.64	7,438
7	Sunway TaihuLight	China	10,649,600	93.01	15,371
8	Perlmutter	USA	706,304	64.59	2,528
9	Selene	USA	555,520	63.46	2,646
10	Tianhe-2A	China	4,981,760	61.44	18,482
113	Athena	Poland	41,616	5.05	146.6

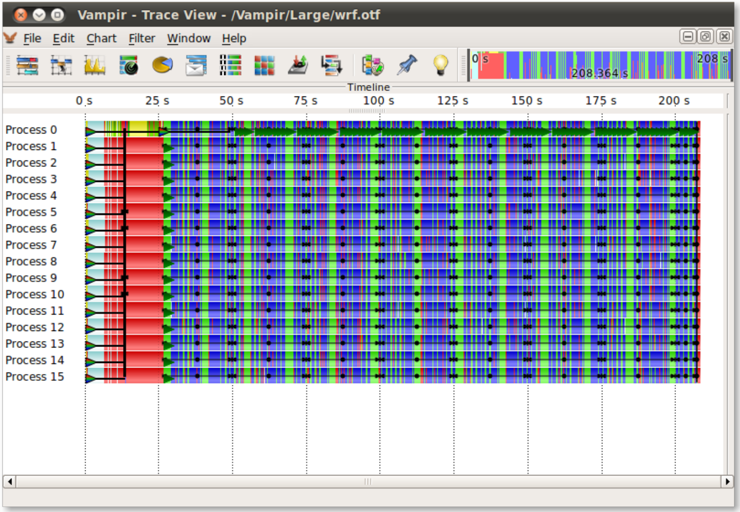
Data as of November 2022. Source: `https://www.top500.org/lists/top500/2022/11/`

Why many small nodes, and not a single large one?



Source: `https://github.com/karlrupp/microprocessor-trend-data`

Example workload



What are we going to study?

Our goal is to learn to write performant and scalable code.

- How to best make use of a single processor
- How to efficiently exchange information between nodes
- How to make use of accelerators (GPUs)

The plan

1. Introduction. The Google Benchmark library [JG]
2. Memory bound – what's the deal with memory? [JG]
3. ILP, SIMD – how to get the most out of a single core [JG]
4. Multithreading – how to make use of the entire CPU [SG]
5. Multithreading cont'd [SG]
6. MPI – communication between nodes [SG]
7. Managing the HPC software stack. Job management [JG]
8. GPUs – when CPUs can't quite cut it [GG?]
9. GPUs cont'd [GG?]
10. Project presentations [participants]

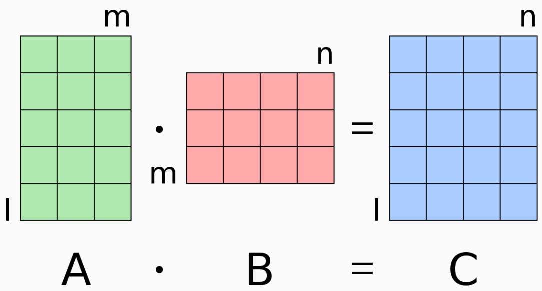
Case study: matrix-matrix multiplication

Classic example: Product of two $n \times n$ matrices

Simplifying assumption: $n = 2^k$

Time complexity: $O\left(n^3\right)$ (exactly $2n^3$)

Problem size: $k = 12 \Rightarrow 2n^3 \approx 137.4 \cdot 10^9$ [Flops]



Platform

CPU model:	2× Intel®Xeon®E5-2650 v3
Clock speed:	2.3GHz
Cores:	2 × 10 (+ hyperthreading)
Vector extensions:	SSE, AVX, AVX2 (256b FMA)
Flop/cycle/core:	16 (DP)
L1 cache:	private: 32kB I, 32kB D
L2 cache:	private: 256kB U
L3 cache:	shared: 25.6MB U
Rpeak:	$2.3 \cdot 2 \cdot 10 \cdot 16 = 736$ [GFlops (DP)]

More about calculating Rpeak:
Dolbeau, R. Theoretical peak FLOPS per instruction set: a tutorial. *J Supercomput* 74, 1341–1377 (2018).
`https://doi.org/10.1007/s11227-017-2177-5`

What can we do?

What do we have control over?

- Programming language
- Compiler flags
- Algorithm
- Implementation

What don’t we have control over?

- The hardware
- Other processes

Let’s brainstorm how to approach this problem...

Attempt 1: Python

```python
import sys, numpy
from time import *

n = 128
A = numpy.random.rand(n, n)
B = numpy.random.rand(n, n)
C = numpy.zeros([n, n])
start = time()
for i in range(n):
  for j in range(n):
    for k in range(n):
      C[i][j] += A[i][k] * B[k][j]
end = time()
print(end - start, 'seconds')
```

Results

Attempt	Walltime	Speedup r.	Speedup a.	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$

Not exactly peak performance...

Attempt 2: C++

Matrices stored in column major order:

`A[i][j]` $\rightarrow$ `A[i + j * N]`

Compiler: gcc 11.2

```
for (ptrdiff_t r = 0; r < N; ++r)
  for (ptrdiff_t c = 0; c < N; ++c)
    for (ptrdiff_t k = 0; k < N; ++k)
      C[r + c * N] += A[r + k * N] * B[k + c * N];
```

Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$

Still not great...

- Why is C++ faster than Python?
- What else can we change?

Let’s turn on optimization

Compiler flags:

- `-O3`
- `-march=native`
- `-mtune=native`

Optimization – the compiler is no longer bound by the letter of our code

It’s now possible for the compiler to emit the FMA instructions available on Haswell

Compiler flags

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02

Loop interchange

Does the order of the loops matter? Let's try to following...

```
for (ptrdiff_t c = 0; c < N; ++c)
  for (ptrdiff_t k = 0; k < N; ++k)
    for (ptrdiff_t r = 0; r < N; ++r)
      C[r + c * N] += A[r + k * N] * B[k + c * N];
```

Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02
Loop interchange	58.8s	15.75	1,279.4	2.34	0.32

What's going on here?

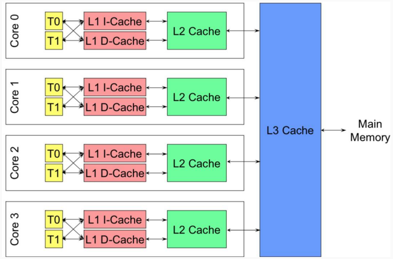
CPU cache

CPUs have caches – small, fast to access memory

CPUs detect sequential memory access and try to prefetch data before it is needed

Memory access patterns which promote good cache utilization:

- Spatial locality: addresses which are close together
- Temporal locality: same address within a short time



Access patterns

Initial:

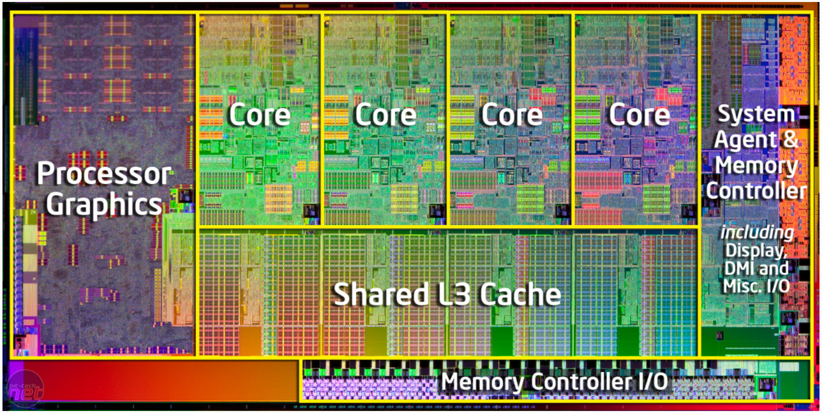
$$\begin{bmatrix} \bullet & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & \dots \\ & & & \\ & & & \\ & & & \end{bmatrix} \times \begin{bmatrix} 1 & & & \\ 2 & & & \\ 3 & & & \\ \vdots & & & \end{bmatrix}$$

With loop interchange:

$$\begin{bmatrix} 1 & & & \\ 2 & & & \\ 3 & & & \\ \vdots & & & \end{bmatrix} = \begin{bmatrix} 1 & & & \\ 2 & & & \\ 3 & & & \\ \vdots & & & \end{bmatrix} \times \begin{bmatrix} \bullet & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Multithreading

Let's start using the entire die...



Multithreaded code

Using Intel's TBB library:

```
using namespace oneapi::tbb;
parallel_for(blocked_range< ptrdiff_t >{0, N},
             [&](const blocked_range< ptrdiff_t >& range) {
    for (auto c = range.begin(); c != range.end(); ++c)
      for (ptrdiff_t k = 0; k < N; ++k)
        for (ptrdiff_t r = 0; r < N; ++r)
          C[r + c * N] += A[r + k * N] * B[k + c * N];
  }
);
```

Rule of thumb: parallelize the outer loop first

Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02
Loop interchange	58.8s	15.75	1,279.4	2.34	0.32
Parallelized	2.44s	24.1	30,832	56.31	7.65

24x speedup? But we only have 20 cores...

How can we explain this?

Can we do better?

We’re making use of all the available hardware facilities…

- Cache friendly memory access
- Vectorization (thanks to the compiler)
- Multithreading

…but we’re still a long way from from Rpeak.

So what can we improve?

Tiling

Idea: improve temporal locality by using a blocked approach. Instead of multiplying whole matrices, break the product down into blocks (tiles). Keep tiles in cache and use them multiple times before moving on to the next one.

```
parallel_for(blocked_range< ptrdiff_t >{0, N / tile_size},
             [&](const blocked_range< ptrdiff_t >& crange) {
  for (ptrdiff_t ct = crange.begin() * tile_size;
       ct < crange.end() * tile_size; ct += tile_size)
    for (ptrdiff_t rt = 0; rt < N; rt += tile_size)
      for (ptrdiff_t kt = 0; kt < N; kt += tile_size)
        for (ptrdiff_t c = 0; c < tile_size; ++c)
          for (ptrdiff_t k = 0; k < tile_size; ++k)
            for (ptrdiff_t r = 0; r < tile_size; ++r)
              C[(r + rt) + (c + ct) * N] +=
              A[(r + rt) + (k + kt) * N] * B[(k + kt) + (c + ct) * N];
  }
);
```

What is the optimal tile size?

Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02
Loop interchange	58.8s	15.75	1,279.4	2.34	0.32
Parallelized	2.44s	24.1	30,832	56.31	7.65
Tiling	1.48s	1.65	50,831	92.84	12.61

We're getting somewhere...

Improved tiling

Our tiling strategy can be further improved:

- Pack and transpose tiles of **A** – improved spatial locality
- Nested tiling

Our implementation may be tweaked a bit further, but we shall leave it be for now. During the subsequent lectures, we will explore a more sophisticated blocked approach (the Goto algorithm). The listener is welcome to play around with the current code and try to squeeze out a bit more performance.

Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02
Loop interchange	58.8s	15.75	1,279.4	2.34	0.32
Parallelized	2.44s	24.1	30,832	56.31	7.65
Tiling	1.48s	1.65	50,831	92.84	12.61
Tiling + transposition	0.869s	1.70	86,570	158.111	21.48
Tiling x2 + transposition	0.774s	1.12	97,196	177.52	24.12

This seems pretty decent, right?

Real world approach

In the real world, faced with a project requiring matrix-matrix multiplication, what would we actually do?

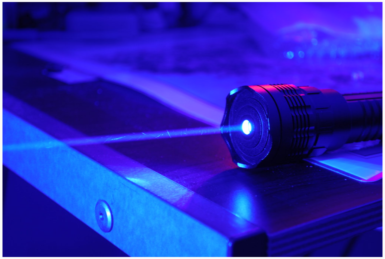
Results

Attempt	Walltime	Speedup rel	Speedup abs	GFlops	% Rpeak
Python	20h 54min	–	1	$1.83 \cdot 10^{-3}$	$0.248 \cdot 10^{-3}$
C++	2h 52min	7.29	7.29	0.013	$1.81 \cdot 10^{-3}$
`-O3 -march=native`	15min 26s	11.15	81.25	0.148	0.02
Loop interchange	58.8s	15.75	1,279.4	2.34	0.32
Parallelized	2.44s	24.1	30,832	56.31	7.65
Tiling	1.48s	1.65	50,831	92.84	12.61
Tiling + transposition	0.869s	1.70	86,570	158.111	21.48
Tiling x2 + transposition	0.774s	1.12	97,196	177.52	24.12
Intel MKL	0.264s	2.93	284,960	520.45	70.71

Speedup

Going from our naïve Python implementation to MKL is like going from

to



Key takeaways

→ Modern hardware is vastly complex

→ Support all claims with measurements

→ Rely on experts

Q&A