# Introduction to High Performance Computing

Lecture 1

Jakub Gałecki

Classes: 10 x 3h

Integrated lecture + computer lab

Instructor: Jakub Gałecki, `jakub.galecki@pw.edu.pl`



Materials

Participants will be graded based on attendance and 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: $-1/2$ grade per absence

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

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 | El Capitan | USA | 11,039,616 | 1,742.00 | 29,581 |
| 2 | Frontier | USA | 9,066,176 | 1,353.00 | 24,607 |
| 3 | Aurora | USA | 9,264,128 | 1,012.00 | 38,698 |
| 4 | Eagle | USA | 2,073,600 | 561.20 | - |
| 5 | HPC6 | Italy | 3,143,520 | 477.90 | 8,461 |
| 6 | Fugaku | Japan | 7,630,848 | 442.01 | 29,899 |
| 7 | Alps | Switzerland | 2,121,600 | 434.90 | 7,124 |
| 8 | LUMI | Finland | 2,752,704 | 379.70 | 7,107 |
| 9 | Leonardo | Italy | 1,824,768 | 241.20 | 7,494 |
| 10 | Tuolumne | USA | 1,161,216 | 208.10 | 3,387 |
| 69 | Helios GPU | Poland | 89,760 | 19,140.00 | 316.88 |

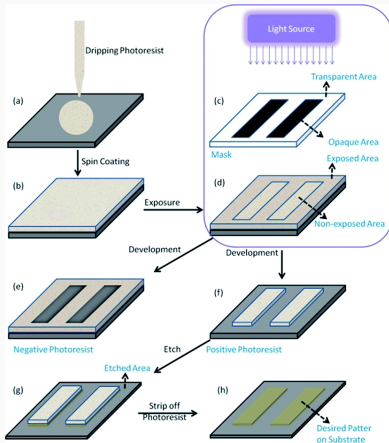# Why many small nodes, and not a single large one?



50 Years of Microprocessor Trend Data
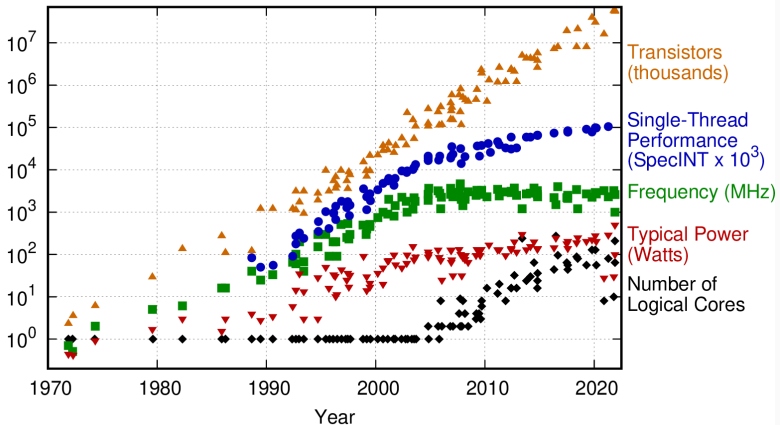
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
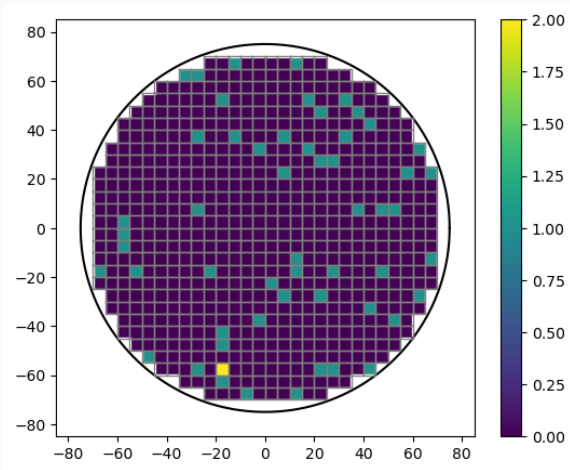New plot and data collected for 2010-2021 by K. Rupp

Dennard scaling:

- As transistors are scaled down in size, their power density remains constant
- Breakdown c.a. 2006 – the power wall
- Cause: leakage current

Moore's law:

- Number of transistors per chip doubles approximately every 2 years
- Still going (?)

# Manufacturing defects
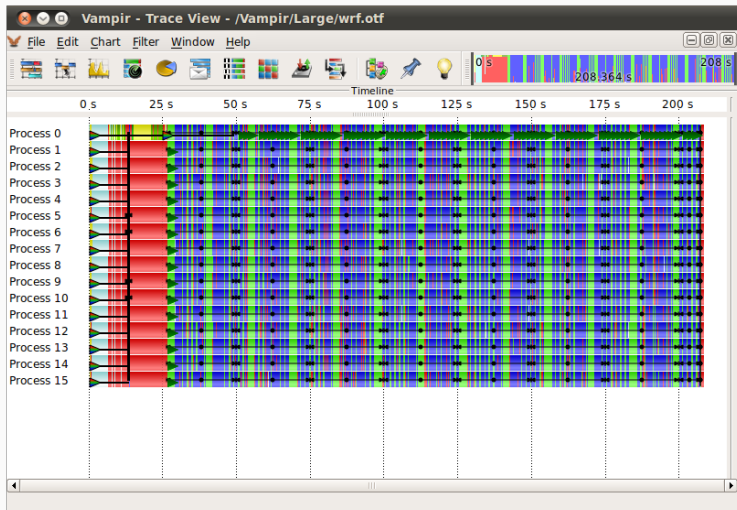
Our goal is to learn to write performant and scalable software which takes full advantage of the available hardware.

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

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

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]



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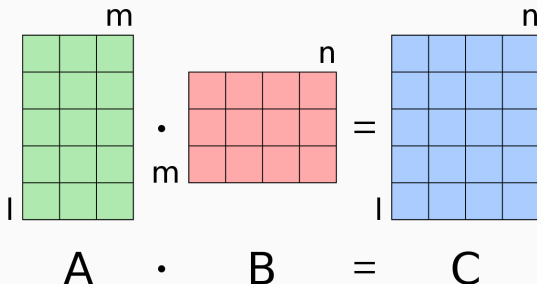
| | |
|---|---|
| CPU model: | 2× Intel®Xeon®E5-2650 v3 |
| Clock speed: | 2.3GHz |
| Cores: | $2 \times 10$ (+ hyperthreading) |
| Vector extensions: | SSE, AVX, AVX2 (256b FMA) |
| Flop/cycle/core: | 16 (DP) |
| L1 cache: | private: 32kB I, 32kB D |
| L2 cache: | private: 256kB U |
| L3 cache: | shared: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

14

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...

```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')
```

| 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...

Matrices stored in column major order:
A[i][j] → A[i + j * N]

Compiler: gcc 11.2

```cpp
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];
```

| 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?

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

| 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 |

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

```cpp
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];
```

| 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
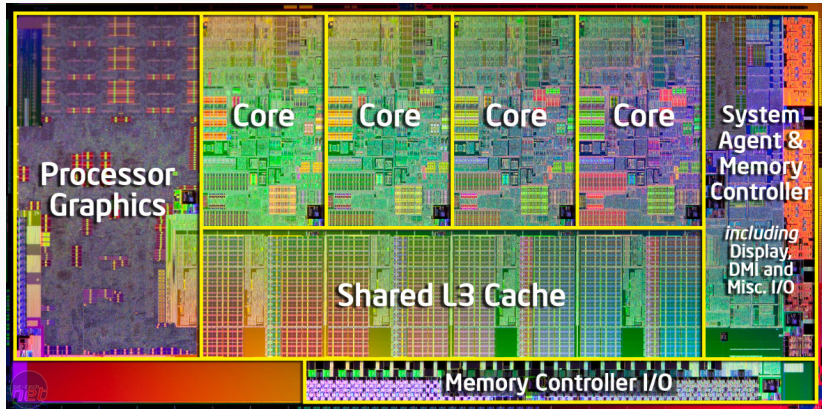- Temporal locality: same address within a short time
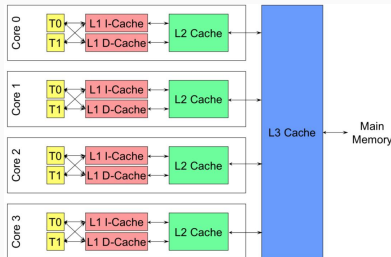
Initial:

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

With loop interchange:

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

Let's start using the entire die…

Using Intel's TBB library:

```cpp
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

| 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?

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?

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?

| 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…

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.

| 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?

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

| 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 |

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



to

→ Modern hardware is vastly complex

→ Support all claims with measurements

→ Rely on experts