High Performance Computing

Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Course Information

- Schedule
  - Wednesday 9:00 – 11:00
  - Thursday 9:00 – 11:00
  - tentative schedule on the eLearning portal
- Office hours
  - Thursday 11:00 – 12:30, after the lecture on the course Teams
- 48 hours of lab / 6 CFU

- Prof. Biagio Cosenza
  - http://www.cosenza.eu
  - bcosenza@unisa.it (subject: "HPC Course")
- Teaching Assistants
  - Majid Salimibeni
  - Muhammad Tanveer
- Language: English
  - You can write me in IT, DE, ES

Course Objectives

- Goal: Be able to program high performance computing systems
  - Programming models for parallel systems
    - vectorial instruction, multi-threading, heterogenous computing, GPU programming
  - Parallel programming patterns
    - rewrite program to exploit parallelism
  - Optimization
    - code and compiler optimization, profiling and tools
  - Applications
- Focus on Lab and Exercise sessions
  - Each lecture has a ~1h of theory part followed by programming exercises

Teaching Materials and Books

- Slide notes
- Publications
- Books
  - *McColl, Robison, Reinders*. Structured Parallel Programming
  - *Mattson, He, Koniges*. The OpenMP Core
  - Herlihy, Shavit. The Art of Multiprocessor Programming
  - *Kirk, Hwu*. Programming Massively Parallel Processors
- Code examples
- Teaching material will be available in the course eLearning portal

Course Grading and Examination

- Group project (code + report), either
  - Reproducibility (1 student): replicate a scientific result published in a top conference
  - Exploration (2-3 students): to investigate a problem on a specific target architecture
  - Either pick a project from our list or propose your idea
    - try to relate your project to the lectures
- Considering bonus points for attendance
- Written examination
  - Exam dates will be shown in the eLearning portal
    - Pre-appello: tbd (June)
    - Primo appello: tbd (June)
    - Secondo appello: tbd (July)
    - Appello Settembre: tbd (September)

Counseling Center

Centro di Counseling Psicologico

Contacts

- 089 965099
- counseling@unisa.it
- https://web.unisa.it/vivere-il-campus/servizi/salute-e-assistenza-sanitaria
  https://web.unisa.it/en/campus-life/services/health-care

Introduction to High Performance Computing

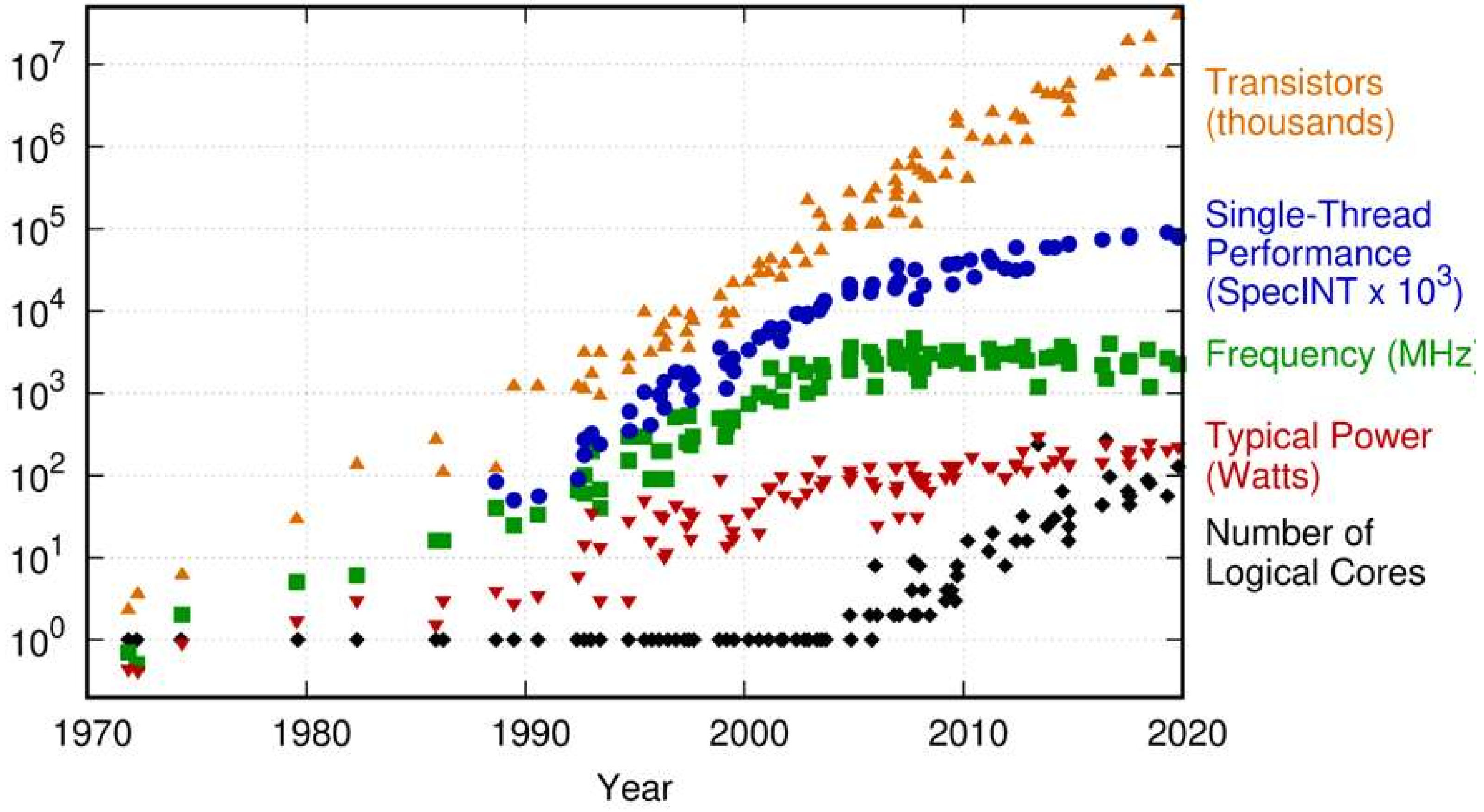
High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

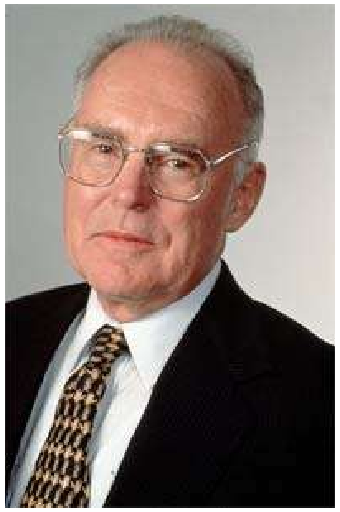
- Motivation for parallelism
  - The three walls, Moore’s law and Dennard's scaling
  - Flynn’s Taxonomy
- High Performance Computing systems and applications
- Programming parallel computing systems
  - Amdahl's and Gustafson Law
- Lab Matrix multiplication

How did we get here?



Moore’s Law

The number of transistors in a dense integrated circuit doubles about every two years.


Gordon Moore

- Moore's law is an observation and projection of a historical trend
  - rather than a law of physics, it is an empirical relationship linked to gains from experience in production
- Moore’s law is about transistors, not performance

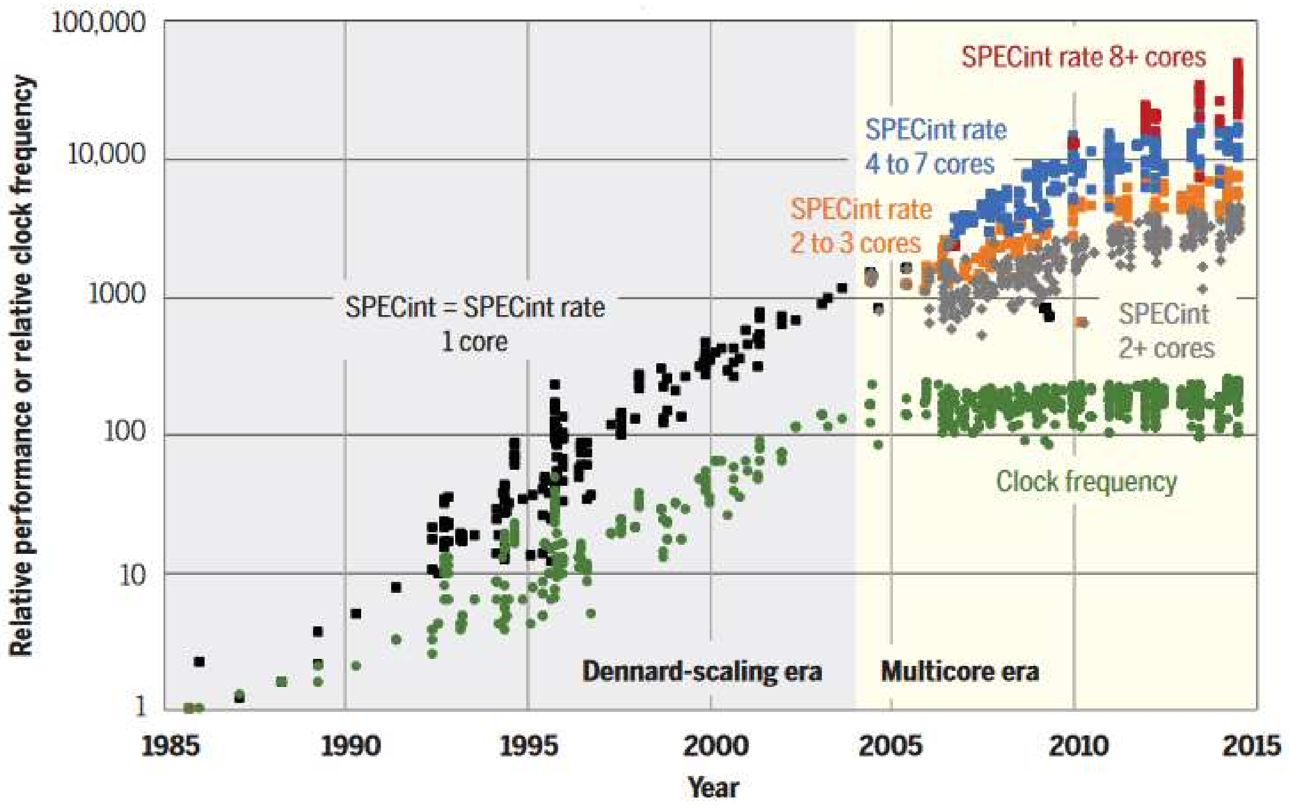
Dennard Scaling and the Power Wall

Voltage and current should be proportional to the linear dimensions of a transistor.
Thus, as transistors shrank, so did necessary voltage and current; power is proportional to the area of the transistor.

Robert Dennard

- Moore's law + Dennard Scaling = Koomey's law
  - performance per watt grows even faster, doubling about every 18 months
- Unfortunately , Dennard's scaling ignored the leakage current and threshold voltage, which establish a baseline of power per transistor
  - transistors get smaller, power density increases because these don't scale with size
  - Power Wall: practically, processor frequency is limited to around 4 GHz since 2006
  - Dark Silicon: the amount of circuitry that cannot be powered-on at the nominal operating voltage for a given thermal design power (TDP) constraint

End of Dennard-scaling era



SPECint (largely serial) performance, SPECint-rate (parallel) performance, and clock-frequency scaling for microprocessors from 1985 to 2015, normalized to the Intel 80386 DX microprocessor in 1985

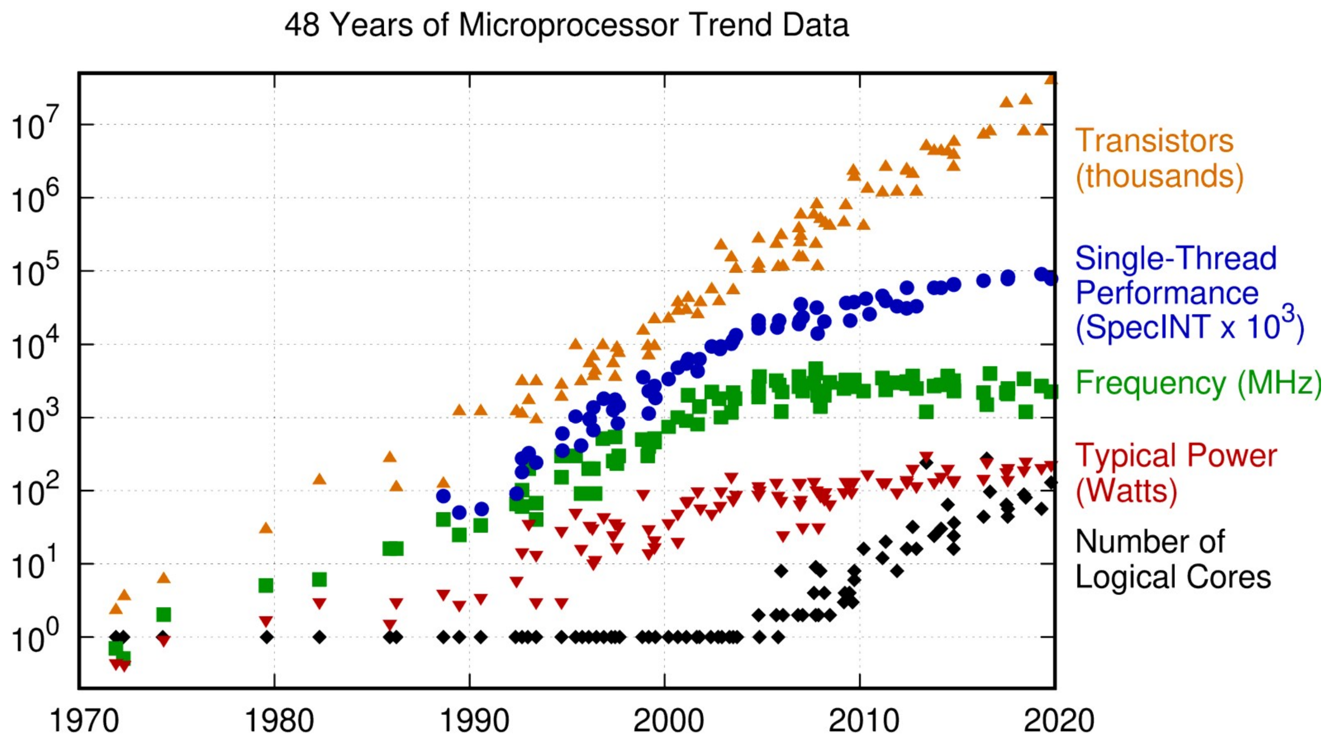
The Power Wall

- Old Conventional Wisdom (CW): Power is free, but transistors are expensive
  - New CW Power is expensive, but transistors are "free"
    - That is, we can put more transistors on a chip than we have the power to turn on
- Old CW: If you worry about power, the only concern is dynamic power
  - New CW For desktops and servers, static power due to leakage can be 40% of total power

Asanovíc, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, Yelick
The Landscape of Parallel Computing Research: A View from Berkeley
EECS Technical Report 2006

The Power Wall

- End of processor's clock rate scaling
- Future gains must be derived from parallelism
- Questions
  - What kind of parallelism?
  - Why not ILP?



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 2010-2019 by K. Rupp

The ILP Wall

- Old CW: We can reveal more instruction-level parallelism (ILP) via compilers and architecture innovation
  - Pipelining within instruction and between instructions, superpipelining
  - Multiple functional units, multi-issue execution
  - Out-of-order execution
  - Superscalar processing
  - Speculation
  - Very Long Instruction Word (VLIW)
  - Hardware multithreading (hyperthreading)
- New CW is the "ILP wall": There are diminishing returns on finding more ILP

J. Hennessy and D. Patterson: Computer Architecture: A Quantitative Approach 4th edition, Morgan Kauffman, San Francisco, 2007

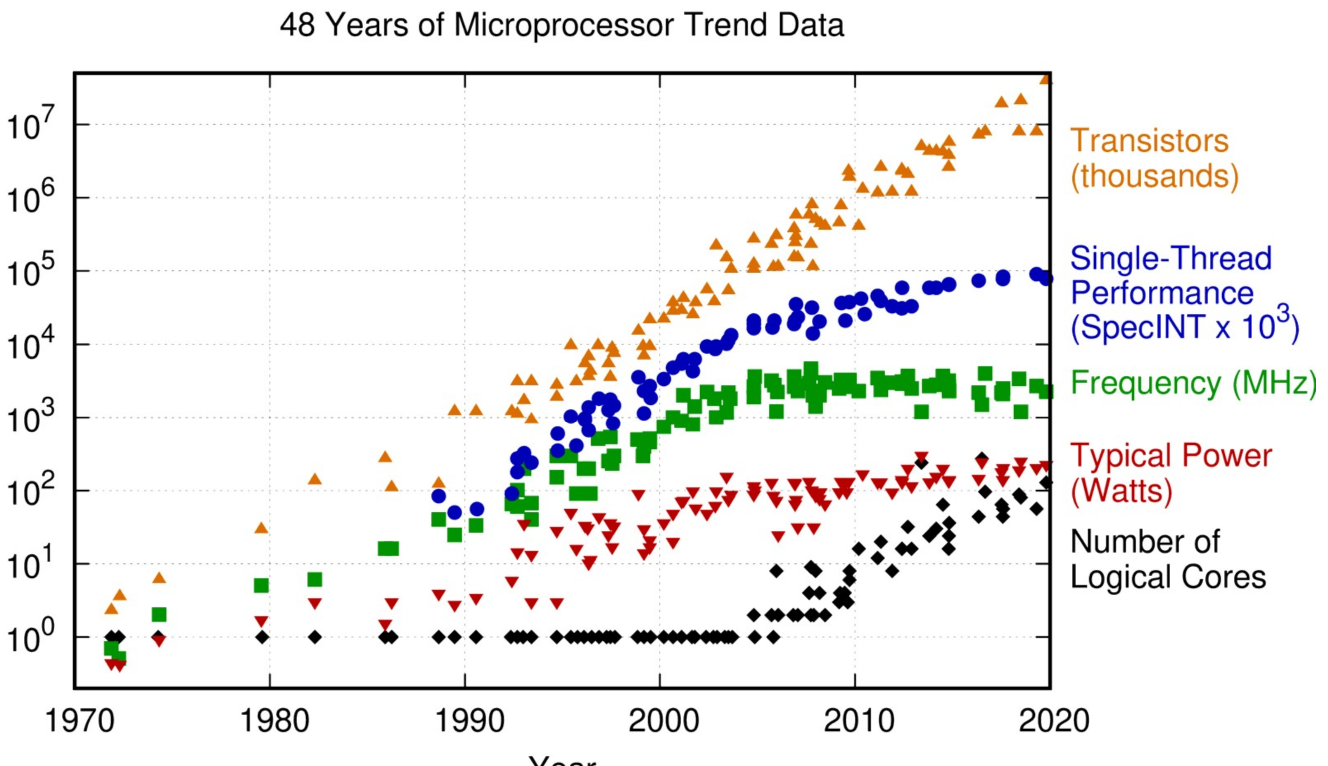
The Memory Wall

- Old CW: Multiply is slow, but load and store is fast
- New CW is the "Memory wall" [Wulf and McKee 1995]
  - load and store is slow, but multiply is fast
  - modern microprocessors can take 200 clocks to access Dynamic Random-Access Memory (DRAM), but even floating-point multiplies may take only four clock cycles

William A. Wulf, Sally A. McKee: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23(1): 20-24 (1995)

Towards Parallel Architectures

- There are limits to "automatic" improvement of scalar performance
  - three walls: ILP, power, memory
  - future gains must be derived from parallelism
- Explicit parallel mechanisms and explicit parallel programming are required for performance scaling



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 2010-2019 by K. Rupp

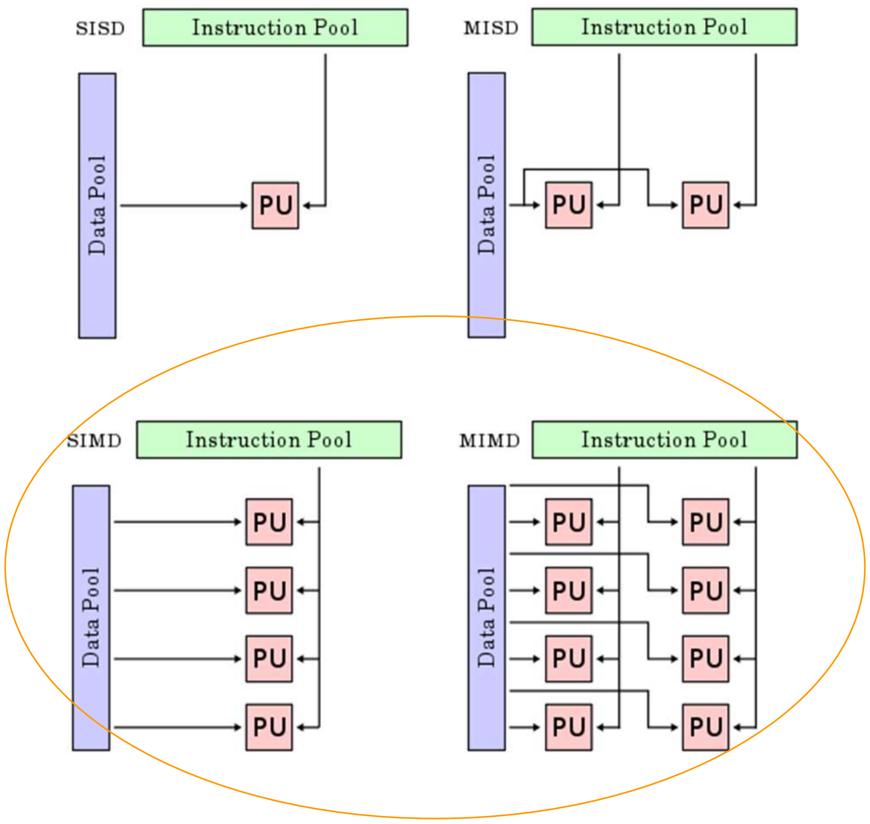
Towards Parallel Architectures: New Conventional Wisdoms

- Old CW: Uniprocessor performance doubles every 18 months
  - New CW Power Wall + Memory Wall + ILP Wall = Brick Wall
  - The doubling of uniprocessor performance may now take 5 years
- Old CW: Don't bother parallelizing your application, as you can just wait a little while and run it on a much faster sequential computer
  - New CW It will be a very long wait for a faster sequential computer
- Old CW: Increasing clock frequency is the primary method of improving processor performance
  - New CW Increasing parallelism is the primary method of improving processor performance
- Old CW: Less than linear scaling for a multiprocessor application is failure
  - New CW Given the switch to parallel computing, any speedup via parallelism is a success

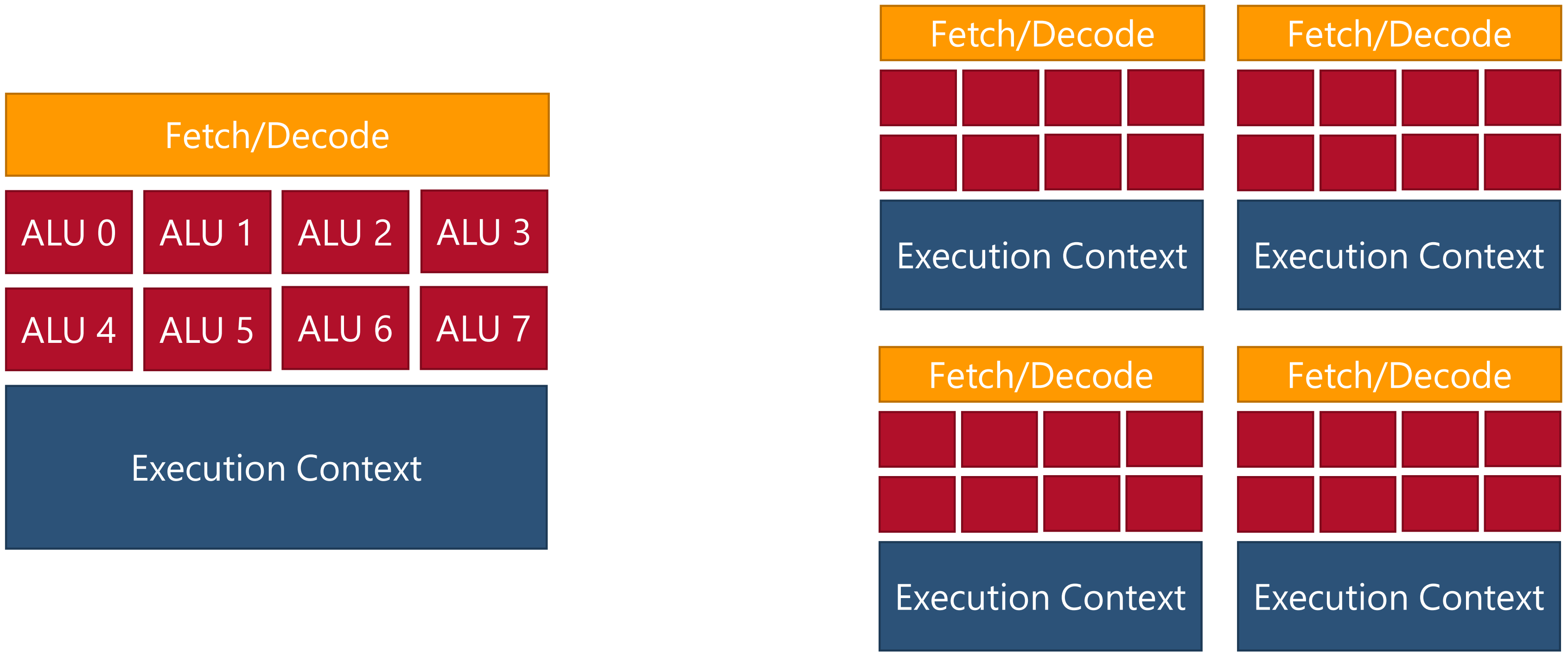
Asanovíc, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, Yelick
The Landscape of Parallel Computing Research: A View from Berkeley. EECS Technical Report 2006

Parallel Hardware: Flynn's Taxonomy

- Single instruction stream, single data stream (SISD)
  - no parallelism, traditional uniprocessor machines
- Single instruction stream, multiple data streams (SIMD)
  - a single instruction operates on multiple different data streams
    - e.g., CPU vectorial instructions (AVX, SVE) and GPU's SIMT model
- Multiple instruction streams, single data stream (MISD)
  - Multiple instructions operate on one data stream
  - uncommon architecture which is generally used for fault tolerance
- Multiple instruction streams, multiple data streams (MIMD)
  - Multiple autonomous processors simultaneously executing different instructions on different data
    - e.g., multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space



From SIMD to Multicore Parallelism



Parallelism in Hardware

- Instruction-Level Parallelism (ILP)
- Data parallelism
  - Increase amount of data to be operated on at same time
- Processor parallelism
  - Increase number of processors
- Memory system parallelism
  - Increase number of memory units
  - Increase bandwidth to memory
- Communication parallelism
  - Increase amount of interconnection between elements
  - Increase communication bandwidth

Heterogeneous Computing

- Heterogeneous Computing: a computing systems that use more than one kind of processor or cores
  - typically, different ISA, e.g. a CPU-GPU system
  - alternatively: same ISA but different core frequency, e.g. ARM big.LITTLE processor
- Also: collaborative, hybrid, co-operative or synergistic execution, coprocessing, divide and conquer approach, etc.
- Why?
  - Hardware specialization
    - exploit domain-specific hardware: e.g., tensor units, graphics support, crypto
  - Program-specific features
  - Optimized for performance, energy, accuracy or other objectives

Impact of Hardware Complexity

- Computer architectures are becoming complex
  - parallelism (SIMD, multi-core, massively parallel processors, accelerators) and heterogeneity
- Harder to program: need to explicitly express details about the hardware
- Harder to tune: require software tuning

We also think that autotuners should take on a larger, or at least complementary, role to compilers in translating parallel programs

Asanovíc, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, Yelick
The Landscape of Parallel Computing Research: A View from Berkeley
EECS Technical Report 2006

High Performance Computing Systems

Fugaku at RIKEN **#1**

158,976 nodes / 7,299,072 cores
Manufacturer: Fujitsu
CPU: A64FX 48C (48+8 cores) 2.2GHz
with SVE vectorial instruction
Network: TOFU interconnect D
Rmax: 415 PFlop/s

HPC5 at ENI S.p.A. **#6**

1820 nodes / 669,760 cores
Manufacturer: Dell EMC
CPU: Intel Xeon Gold 6252 24C 2.1GHz
Accelerators: 4x NVIDIA Tesla V100 GPUs
Network: Mellanox HDR Infiniband
Rmax: 35 PFlop/s

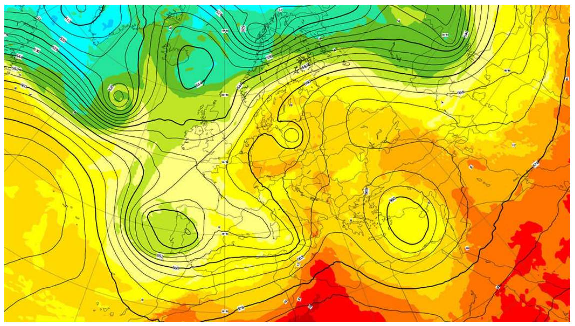
Marconi-100 at CINECA **#9**

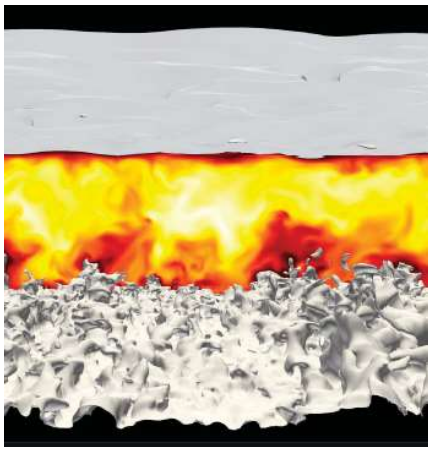
980 nodes / 347,776 cores
Manufacturer: IBM
CPU: 2x16 cores IBM POWER9 AC922 at 3.1 GHz
Accelerators: 4x NVIDIA Volta V100 GPUs
Network: Mellanox IB EDR DragonFly++
Rmax: 22 PFlop/s

High Performance Computing Applications

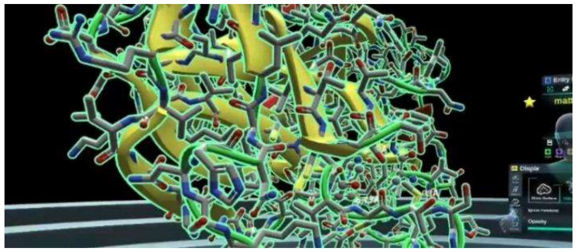
Identify oil reservoirs / ENI spa

Deep Learning / CINECA


Weather forecast / ECMWF


Naval design / CINECA

Aerodynamic simulation
Dallara Automobili


Drug discovery
DOMPE spa

"(...) The geophysical and seismic information we collect from all over the world is sent to HPC5 for processing. Using this data, the system develops extremely in-depth subsoil models, and on the basis of these, we can determine what is hidden many kilometres below the surface: indeed, this is how we found Zohr, the largest gas field ever discovered in the Mediterranean." Source ENI: https://www.eni.com/en-IT/operations/green-data-center-hpc5.html

How do we program these machines?

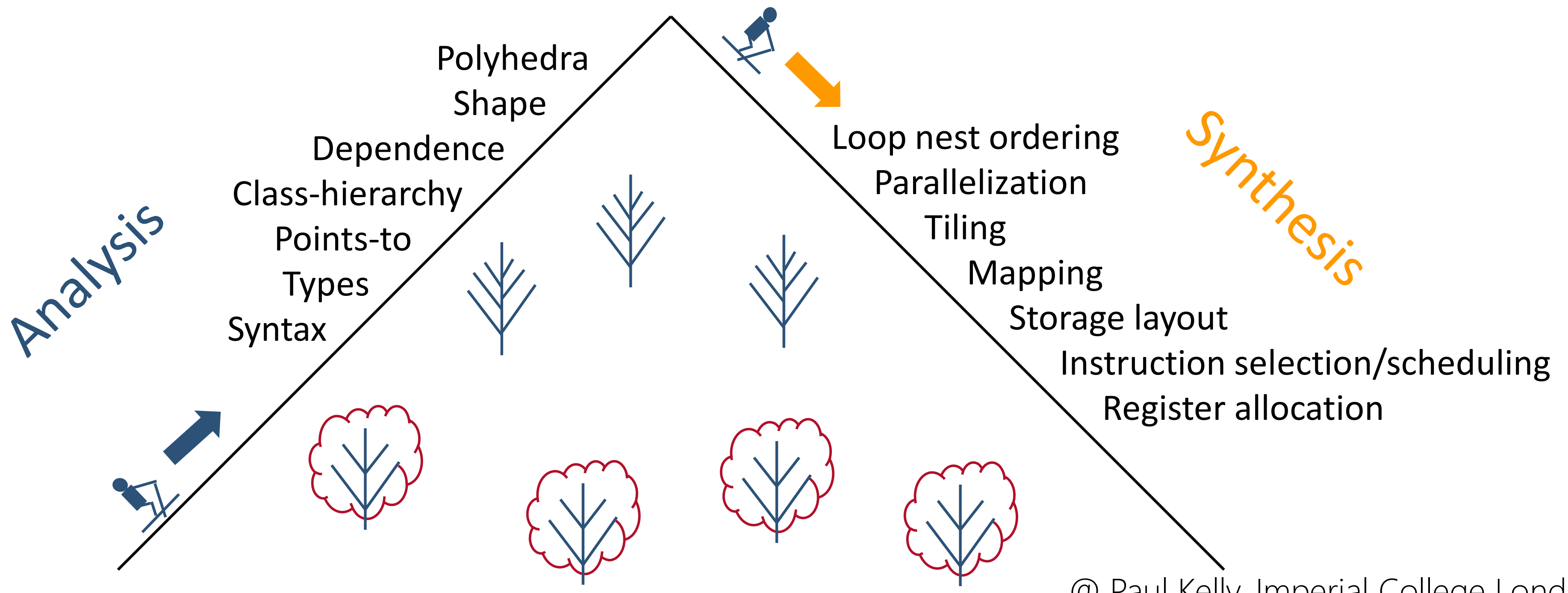
Kawasaki 500 Mach III (1969)

- One of the hardest-to-drive motorbikes in history
  - 65 CV on 185 kg
  - engine was placed in the back
  - too much power for the time
    - no electric control
- Also known as “the flying coffin”
- Only expert bikers could drive it

Programming Models

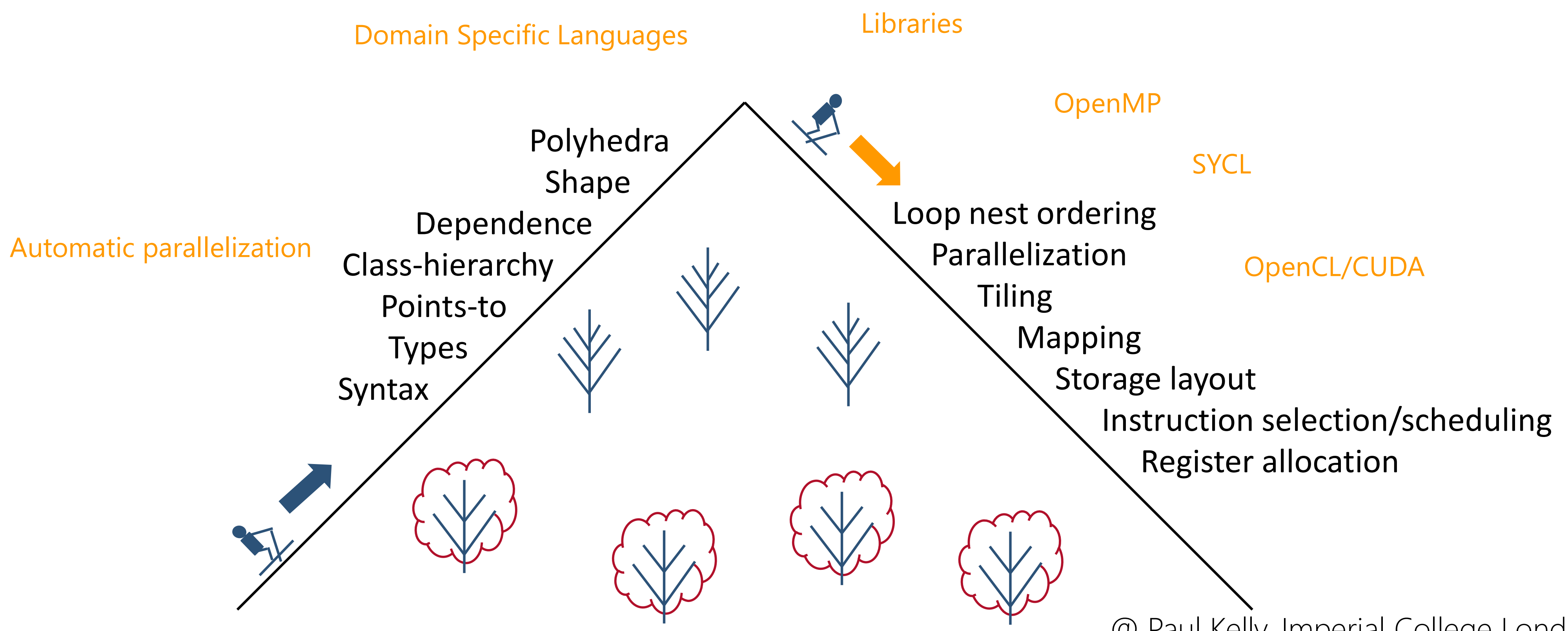
- Programming models includes
  - programming languages
  - compilers
  - libraries
  - software ecosystems
- A programming model is about abstractions
- Important questions
  1. How do we exploit hardware parallelism through a programming models?
  2. How do we express parallelism through a programming models?

Programming Models – What a compilers does



@ Paul Kelly, Imperial College London

Programming Models for Parallel Architectures



@ Paul Kelly, Imperial College London

Parallel Programming Models Examples

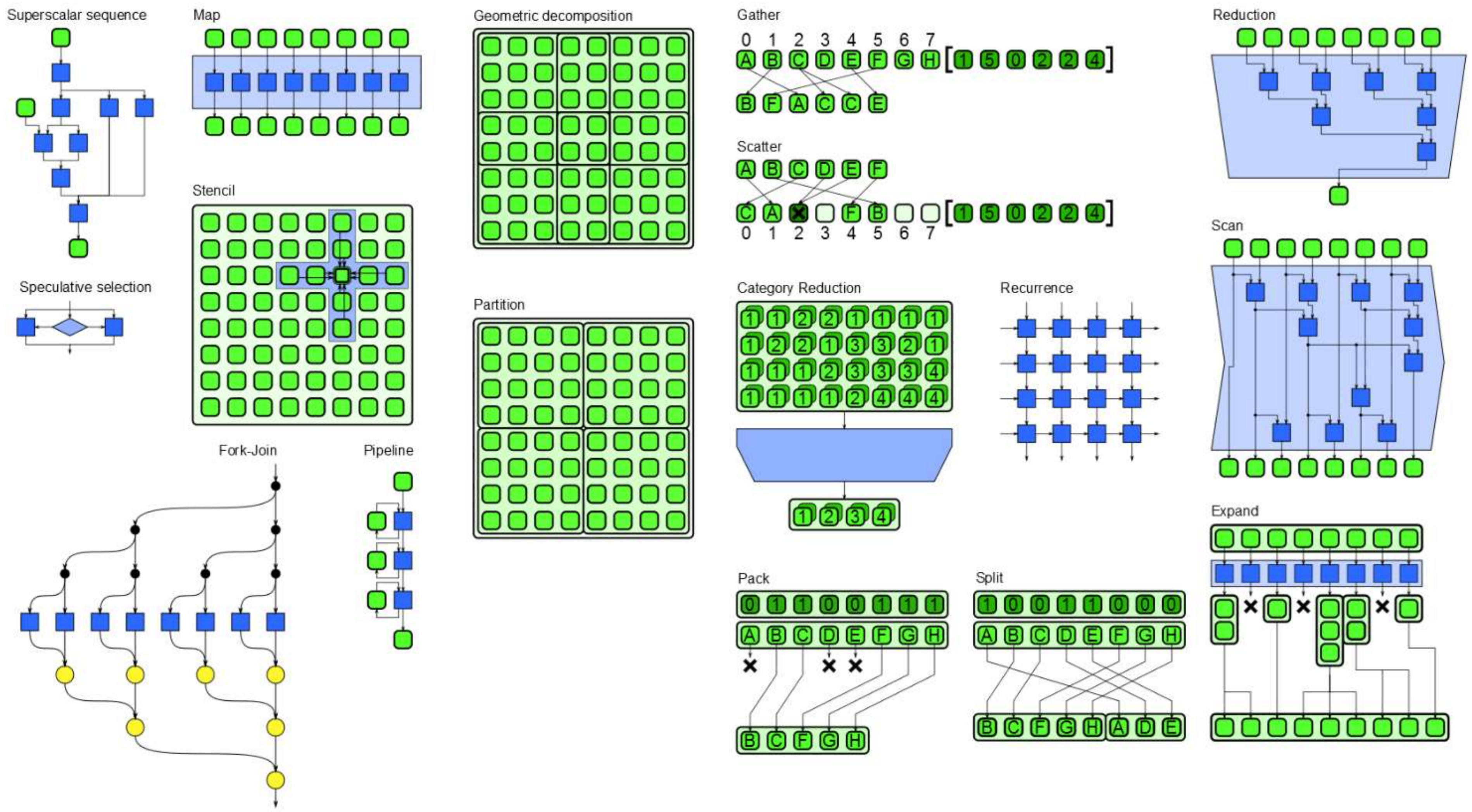
- C/C++ sequential code
- SIMD intrinsic
- OpenMP pragmas
- Cilk
- OpenCL, SYCL
- CUDA, ROCm HIP
- Vulkan
- MPI
- Runtime
  - High Performance ParalleX (HPX)
  - OmpSs
  - Celerity
  - Legion
- DSL
  - SQL, OpenGL Shading Languages
- Libraries
  - BLAS, FFTW
  - GROMACS, PLASMA

Research Questions in Programming Models

- Is it possible to improve existing programming languages with more advanced analyses?
  - Can we reach higher performance with existing programming languages?
- Can programming models be extended to provide high-level programmability and high-performance?
- Can programming models exploit domain-specificity to both improve programmability and performance?
  - Domain Specific Languages, libraries
- How can we automatically tune existing code transformation frameworks to reach (portable) high-performance with little efforts by the programmers?
  - Autotuning

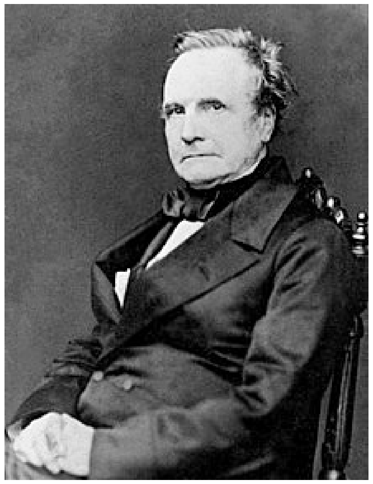
Structured Programming with Patterns

- Patterns are "best practices" for solving specific problems
  - can be used to organize your code, leading to algorithms that are more scalable and maintainable
  - a pattern supports a particular "algorithmic structure" with an efficient implementation
  - good parallel programming models support a set of useful parallel patterns with low-overhead implementations
- Most popular patterns [McCool, Robison, Reinders]
  - superscalar sequence, map, recurrence, scan, reduce, pack/expand, fork/join, pipeline, partition, segmentation, stencil, search/match, gather


Superscalar sequence
Map
Geometric decomposition
Gather
0 1 2 3 4 5 6 7
A B C D E F G H
1 5 0 2 2 4
B F A C C E
Scatter
A B C D E F
C A F B
0 1 2 3 4 5 6 7
1 5 0 2 2 4
Reduction
Stencil
Speculative selection
Partition
Category Reduction
1 1 2 2 1 1 1 1
1 2 2 1 3 3 2 1
1 1 1 2 3 3 3 4
1 1 1 1 2 4 4 4
1 2 3 4
Recurrence
Scan
Fork-Join
Pipeline
Expand
Pack
0 1 1 0 0 1 1 1
A B C D E F G H
B C F G H
Split
1 0 0 1 1 0 0 0
A B C D E F G H
B C F G H A D E

Performance

The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.



- Performance is the main motivation for parallelism

Charles Babbage

Parallel Performance

- What is parallel performance?
- Performance is the main motivation for parallelism
  - Parallel performance versus sequential performance
  - If the "performance" is not better, parallelism is not necessary
- Parallel processing includes techniques and technologies necessary to compute in parallel
  - hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools
- Parallelism must deliver performance
  - How? How well?

Performance Scaling

- If each processor is rated at `k` MFLOPS and there are `p` processors, should we see `k*p` MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?
- Several causes affect performance
  - each must be understood separately
  - but they interact with each other in complex ways
  - solution to one problem may create another
  - one problem may mask another
- Performance scaling with regard to number of processor and problem size

Embarrassingly Parallel Computations

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - in a truly embarrassingly parallel computation, there is no interaction between separate processes
  - in a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes `T` time sequentially, there is the potential to achieve `T/P` time running in parallel with `P` processors
  - What would cause this not to be the case always?

Scalability

- Scalability: ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- How do you evaluate scalability?
- Comparative evaluation
  - Is scalability linear? If double the number of processors, what to expect?
- Evaluation metrics
  - Sequential runtime (T_{seq}) is a function of problem size and architecture
  - Parallel runtime (T_{par}) is a function of problem size and parallel architecture, # processors used in the execution
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?

Performance Metrics

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processors
- *S(p)* (S_p) is the speedup: $S(p) = \frac{T_1}{T_p}$
  - T_1 is sometime
- *E(p)* (E_p) is the parallel efficiency: $E(p) = \frac{S_p}{p}$
  - usually expressed as %, e.g.,100% efficiency
- *Cost(p)* (C_p) is the cost: $Cost(p) = p\, T_p$

- A parallel algorithm is cost-optimal or cost efficient if the parallel runtime times the number of processors is comparable to the running time of the best sequential algorithm
  - $T_p\, p = T_1$, which implies $C_p = T_1$ *and* $E_p = 100\%$

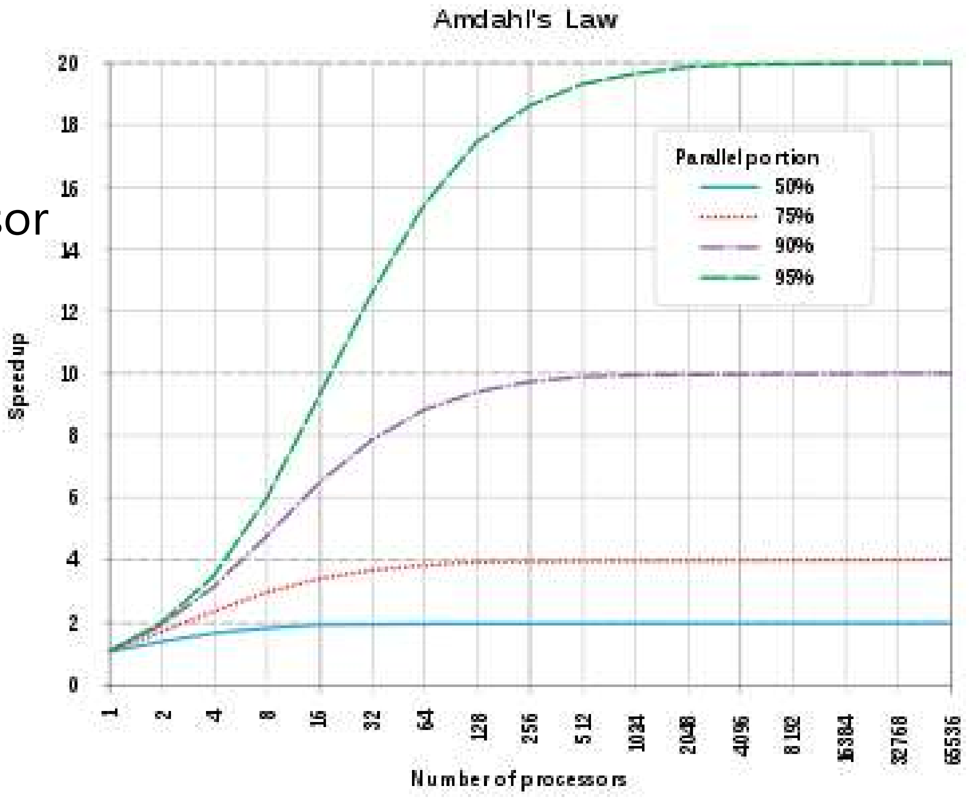
Amdahl's Law: Fixed-Size Speedup

- Let f be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let T_1 be the execution time on 1 processor
- Let T_p be the execution time on p processors
- S_p is the speedup

$$S_p = T_1 / T_p$$
$$= T_1 / (fT_1 + (1-f)T_1/p))$$
$$= 1/(f + (1-f)/p))$$

- As $p \rightarrow \infty$

$$S_p = 1/f$$

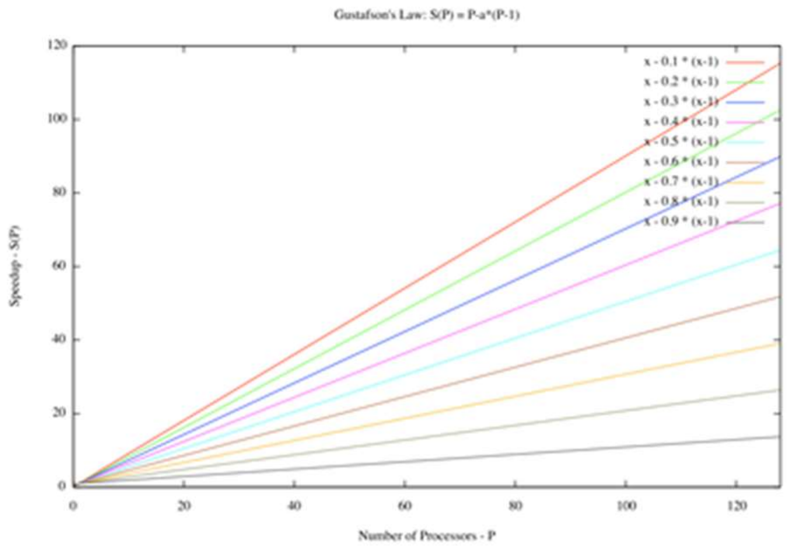


Issues with Amdahl's Law

- Scalability: Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Amdahl's Law apply?
  - When the problem size is fixed
  - Strong scaling ($p \to \infty$, $S_p = S_\infty \to 1 / f$)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not by the # processors
  - Perfect (Amdahl's) efficiency is hard to achieve

Gustafson-Barsis' Law: Scaled Speedup

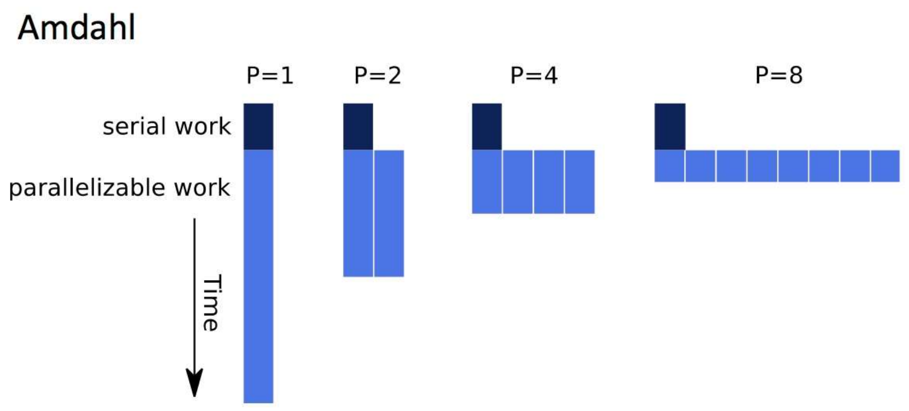
- Often interested in larger problems when scaling
  - How big of a problem can be run (HPC Linpack)
  - Constrain problem size by parallel time
- Assume parallel time is kept constant
  - $T_p = C = (f + (1-f)) * C$
  - f_{seq} is the fraction of T_p spent in sequential execution
  - f_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
  - Let $C = 1$, then $T_s = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par}$
- What is the speedup in this case?
  - $S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par}$

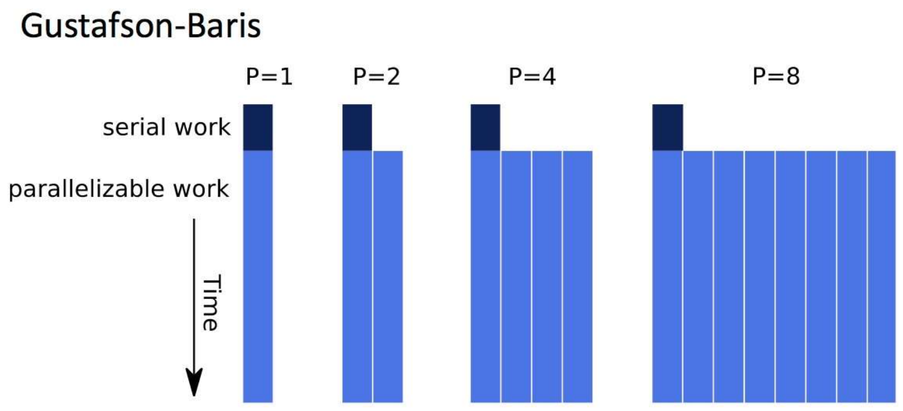


Gustafson-Barsis' Law and Scalability

- Scalability: Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Gustafson's Law apply?
  - When the problem size can increase as the number of processors increases
  - Weak scaling ($S_p = 1 + (p-1)f_{par}$)
  - Speedup function includes the number of processors!
  - Can maintain or increase parallel efficiency as the problem scales

Amdahl vs Gustafson-Baris





Strong vs Weak Scaling

- High performance computing, there are two common notions of scalability
- Strong scaling
  - defined as how the solution time varies with the number of processors for a fixed total problem size
  - Amdahl's approach
  - harder to obtain
- Weak scaling
  - defined as how the solution time varies with the number of processors for a fixed problem size per processor
  - Gustafson-Baris’ approach

Lab Setup

- Install a C compiler
  - Instruction for Ubuntu 20.04.1.LTS
  - GCC

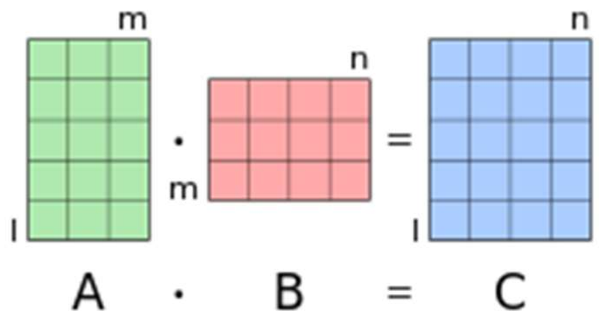
```
> sudo apt install gcc
```

  - LLVM

```
> sudo apt install libllvm10
```

Lab: Matrix Multiplication

- Basic linear algebra tool
  - Input: two matrices A with size `lxm` and B with size `mxn`
  - Output: a matrix C with size `lxn`
  - C has the number of rows of the first and the number of columns of the second matrix



Lab: Matrix Multiplication

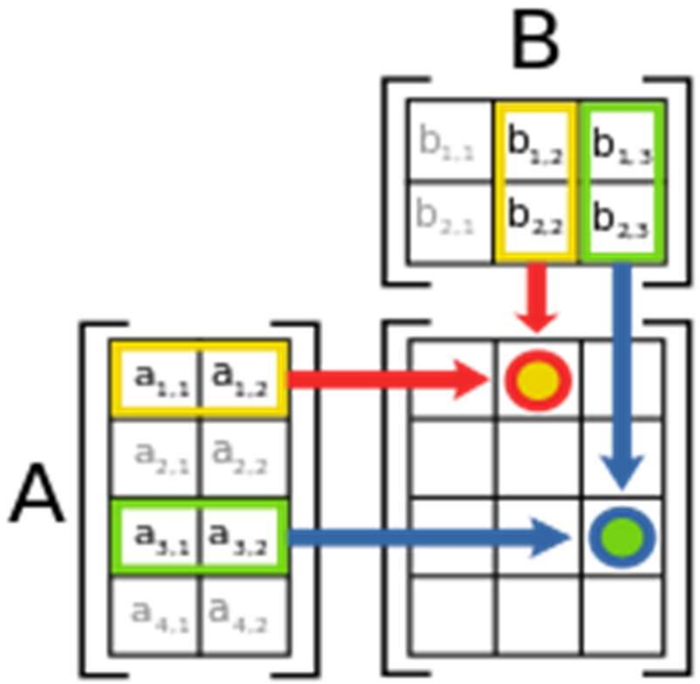
- Given $A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$ and $B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$
- We calculate the matrix $C = AB$ as follow:

$$C = \begin{bmatrix} a_{1,1}b_{1,1} + a_{2,1}b_{1,2} & a_{1,2}b_{1,1} + a_{2,2}b_{1,2} \\ a_{1,1}b_{2,1} + a_{2,1}b_{2,2} & a_{1,2}b_{2,1} + a_{2,2}b_{2,2} \end{bmatrix}$$

Visually:

$$\begin{array}{cc} & \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \\ \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & \begin{bmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{bmatrix} \end{array}$$

Lab: Matrix Multiplication Example



Lab Assignment

- Write a matrix multiplication in C or C++
  - assume l=m=n and power of two size
- Report on timing for different input sizes