Parallel algorithms
Introduction

Paolo Bientinesi

Umeå Universitet
pauldj@cs.umu.se

AQTIVATE workshop
28–29 November 2023

HPAC High Performance and Automatic Computing

UMEÅ UNIVERSITY

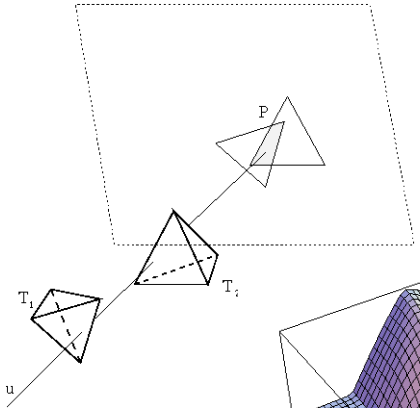
HPC2N

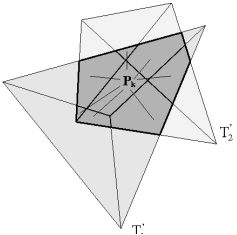
About me

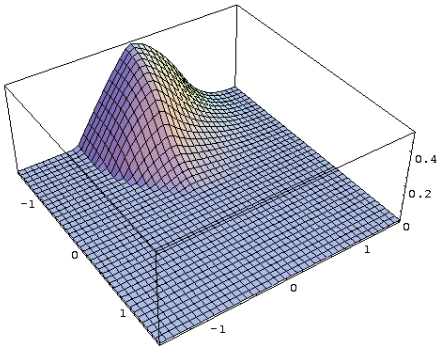
Italy – Tuscany

Italy – University of Pisa

Computational Geometry







USA – UT Austin, TX

- Symmetric eigenproblem $AX = X\Lambda$
- FLAME project
  Automatic generation of algorithms

Algorithm $LU = A$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right), U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$

where $A_{TL}, L_{TL}, U_{TL},$ are 0×0

While $m(A_{TL}) \leq m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & u_{01} & U_{02} \\ \hline 0 & \upsilon_{11} & u_{12}^T \\ \hline 0 & 0 & U_{22} \end{array}\right)$$

where $\alpha_{11}, 1, \upsilon_{11}$ are scalars

$$\upsilon_{11} := \alpha_{11} - l_{10}^T u_{01}$$
$$u_{12}^T := a_{12}^T - l_{10}^T U_{02}$$
$$l_{21} := (a_{21} - L_{20} u_{01})/\upsilon_{11}$$

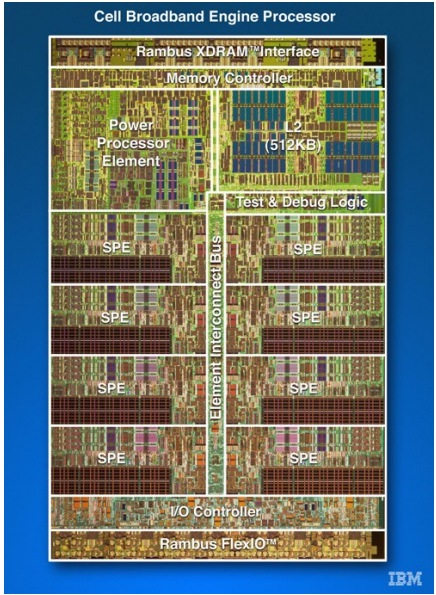
Continue with

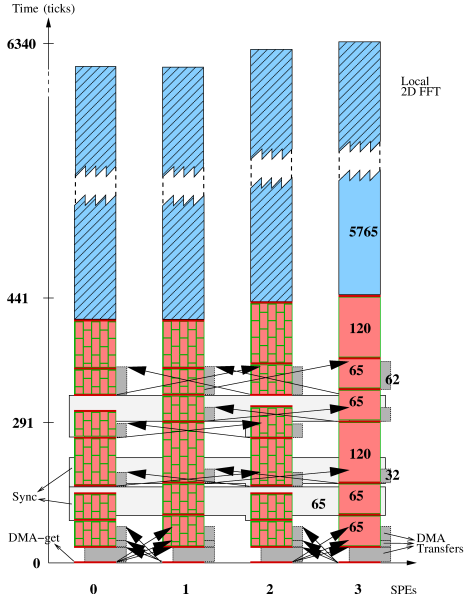
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & 1 & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right), \cdots$$

endwhile

USA – Duke, NC

- Cell
- FFTs





Germany – RWTH Aachen University

High-Performance &
Automatic Computing

Modern
Inverse
Problems

Sweden – Umeå University

- Matrix & tensor operations
- HPC
- Computer music

HPC2N

High-Performance Computing Center North

Motivation

Why Parallel Computing?

- Time: Sequentially, the problem cannot be solved fast enough
  - Real-time constraints, accuracy, ...

Motivation

Why Parallel Computing?

- Time: Sequentially, the problem cannot be solved fast enough
  - Real-time constraints, accuracy, ...
- Space: The problem cannot be solved on a single node
  - Large datasets, large memory requirement.

Motivation

Why Parallel Computing?

- Time: Sequentially, the problem cannot be solved fast enough
  - Real-time constraints, accuracy, ...
- Space: The problem cannot be solved on a single node
  - Large datasets, large memory requirement.
- My computer is parallel, why not to take advantage of it?

The Top500 List – Top500.org

- List of the 500 fastests supercomputers in the world
- Twice per year (ISC, June, Germany. SC, November, USA)
- Computers ranked based on the LINPACK benchmark
  - Solution of linear system of equations: $Ax = b$
  - Result measured in FLOPS/s[1] (in double precision)
- Established in 1993: 60 GFlop/s
- June 2016: 122,000,000 GFlop/s
- June 2023: 1,194,000,000 GFlop/s

[1] Floating Point Operations per Second

Examples of supercomputers

Fugaku
Rank #1: June 2020 – Nov. 2021

Source: top500.org

Fugaku

- Site: Riken Center for Computational Science in Kobe, Japan
- 158,976 nodes
  - Fujitsu A64FX CPU (48+4 core)
- 7,630,848 cores
- 32 GiB/node RAM
- 442,010 PFlop/s (537,212 PFlop/s)
- 29,899 kW

Examples of supercomputers

Sunway TaihuLight
Rank #1: June 2017 – Nov. 2017

Source: top500.org

Sunway TaihuLight

- Site: National Supercomputing Center in Wuxi, China
- 40,960 SW26010 processors, each
  - 256 processing cores
  - 4 auxiliary cores
- 10,649,600 cores
- 1,310,720 GB RAM
- 93,014.6 TFlop/s (125,435.9 TFlop/s)
- 15,371 kW

So...

is parallel programming only for supercomputers?

So...

is parallel programming only for supercomputers?

Not at all!!

Parallel Computers are everywhere!

Source: hp.com

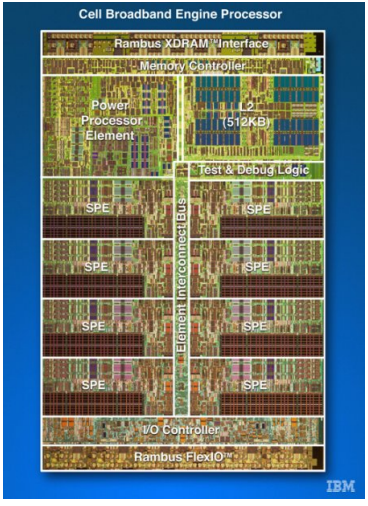
Source: indiatimes.com

Source: bq.com

Source: wearabledevices.es

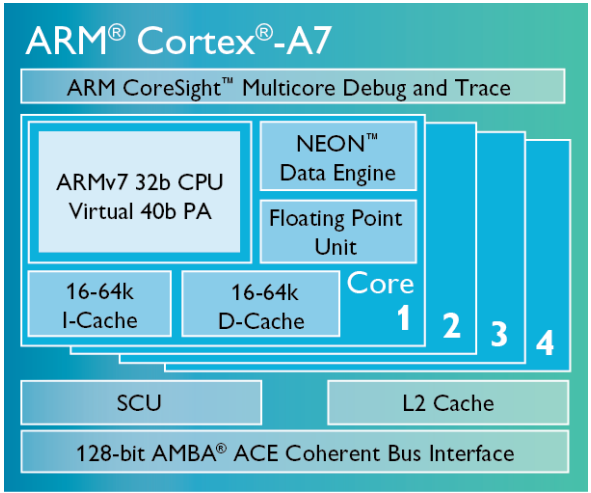
Parallel Computers are everywhere!

Play Station 3



Parallel Computers are everywhere!

Cell phones



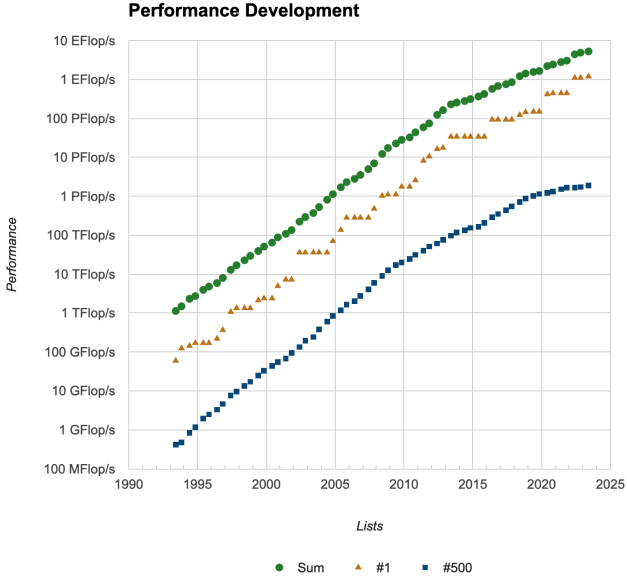
Source: arm.com

Fundamental axiom of High-Performance Computing

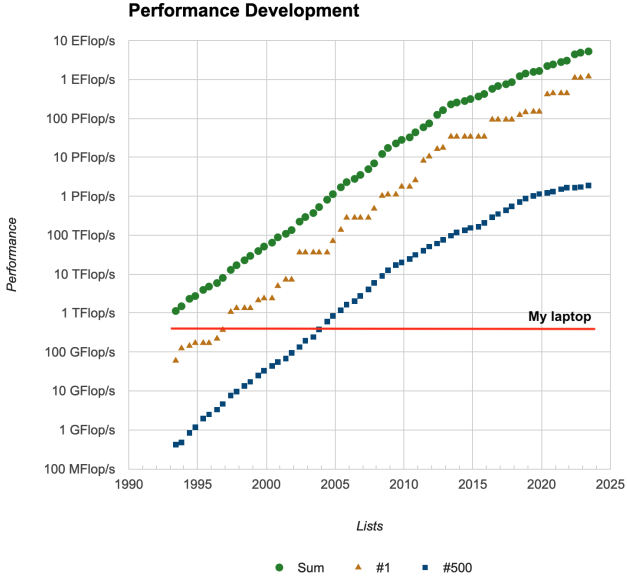
Fundamental axiom of High-Performance Computing

Every computer is a supercomputer, even your laptop!

Top 500 supercomputers in the world



My laptop $>$ #1 supercomputer of 1997 !!!



Analogy: Supercars

We all own Formula-1 racing cars

Analogy: Supercars

We all own Formula-1 racing cars

do we drive them as such?

Analogy: highly parallel computers

. . . often used sequentially

Sequential pseudo-code

Sequential pseudo-code

Dijkstra shortest path (from Wikipedia)

```
1  function Dijkstra(GRAPH, SOURCE):
2
3      create vertex set Q
4
5      for each vertex v in GRAPH:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[SOURCE] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```

Sequential pseudo-code

Dijkstra shortest path (from Wikipedia)

```
1  function Dijkstra(GRAPH, SOURCE):
2
3      create vertex set Q
4
5      for each vertex v in GRAPH:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[SOURCE] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```

Top-to-bottom execution: **One line of the program is completed before the next one is executed.**

Sequential code

Dijkstra shortest path (from geeksforgeeks.org)

```
#include <limits.h>
#include <stdio.h>
#include <stdbool.h>
int  minDistance(int dist[], bool sptSet[]);
void printSolution(int dist[]);

void dijkstra(int graph[V][V], int src) {
    int  dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}
```

Sequential code

Dijkstra shortest path (from geeksforgeeks.org)

```
#include <limits.h>
#include <stdio.h>
#include <stdbool.h>
int  minDistance(int dist[], bool sptSet[]);
void printSolution(int dist[]);

void dijkstra(int graph[V][V], int src) {
    int  dist[V];
    bool sptSet[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    printSolution(dist);
}
```

The code dictates how the computational steps unfold. Everything else is left to the compiler.

Parallel code

$v_i^{\text{next}} := f(v_{i-1}^{\text{now}}) + f(v_i^{\text{now}}) - v_{i+1}^{\text{now}}$

Parallel code

$v_i^{\text{next}} := f(v_{i-1}^{\text{now}}) + f(v_i^{\text{now}}) - v_{i+1}^{\text{now}}$

```
  MPI_Isend(v_now, n, MPI_INT, proc_left, proc_left, MPI_COMM_WORLD, &req_send_left);  // v_i -> p_{i-1}
  MPI_Irecv(v_left, n, MPI_INT, proc_left, proc_left, MPI_COMM_WORLD, &req_recv_left); // f(v_{i-1}) <- p_{i-1}
  MPI_Irecv(v_right, n, MPI_INT, proc_right, me, MPI_COMM_WORLD, &req_recv_right);     // v_{i+1} <- p_{i+1}

   f(v_now, v_next, n);    // v_next = f(v_now)

  // copy fv and send to next
  memcpy(fv, v_next, n * sizeof(int));
  MPI_Isend(fv, n, MPI_INT, proc_right, me, MPI_COMM_WORLD, &req_send_right);

  // Receive in any order
  MPI_Request reqs2 = {req_recv_left, req_recv_right};
  int idx, p;

  for (p=0; p<2; p++) {
     MPI_Waitany(2, reqs, &idx, MPI_STATUS_IGNORE);
     if (idx == 0) // Received from proc on the left
        {
           printf("%d Received from left (p%d): %d\n", me, proc_left, v_left0);
           for (i = 0; i < n; i++)
              v_nexti += v_lefti;
        } else // Received from proc on the right
        {
           printf("%d Received from right (p%d): %d\n", me, proc_right, v_right0);
           for (i = 0; i < n; i++)
              v_nexti -= v_righti;
        }
  }
 // Wait for the sends to complete
 MPI_Wait(&req_send_left, MPI_STATUS_IGNORE);
 MPI_Wait(&req_send_right, MPI_STATUS_IGNORE);
```

RED: computation BLACK: synchronization, data movement

No assumptions can be made about which processor starts or completes first.

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?
  Not really. Maybe in 5-10 years? (maybe, but do not believe the hype)

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?
  Not really. Maybe in 5-10 years? (maybe, but do not believe the hype)
- Manually?

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?
  Not really. Maybe in 5-10 years? (maybe, but do not believe the hype)

- Manually? Yes, very much so.
  With (a slew of) tools, libraries, programming languages for parallel computing.
  E.g.: Charm, Chapel, Cilk, Coarray, CUDA, Hadoop, MapReduce, OpenCL, OpenSHMEM, OpenACC, Pthreads, TBB, . . . (and many more)

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?
  Not really. Maybe in 5-10 years? (maybe, but do not believe the hype)

- Manually? Yes, very much so.
  With (a slew of) tools, libraries, programming languages for parallel computing.
  E.g.: Charm, Chapel, Cilk, Coarray, CUDA, Hadoop, MapReduce, OpenCL, OpenSHMEM, OpenACC, Pthreads, TBB, . . . (and many more)

  - Unfortunately, too many, too different, too differently supported, . . .
  - Difficult to discern research projects from reliable long-term efforts
  - Difficult to predict support and involvement from all the different stakeholders

Holy grail: Automatic parallelization

How to decompose? How to coordinate?

- Automatic languages, parallelizing compilers?
  Not really. Maybe in 5-10 years? (maybe, but do not believe the hype)

- Manually? Yes, very much so.
  With (a slew of) tools, libraries, programming languages for parallel computing.
  E.g.: Charm, Chapel, Cilk, Coarray, CUDA, Hadoop, MapReduce, OpenCL, OpenSHMEM, OpenACC, Pthreads, TBB, . . . (and many more)

  - Unfortunately, too many, too different, too differently supported, . . .
  - Difficult to discern research projects from reliable long-term efforts
  - Difficult to predict support and involvement from all the different stakeholders

  - Best bet: Invest in those core tools/libraries that existed already 10 years ago, and are still being actively developed: MPI, pthreads, OpenMP, CUDA

Summarizing

- Parallel programming is critical in science and engineering

Summarizing

- Parallel programming is critical in science and engineering
- Relevant not only to supercomputers, but to any workstation/laptop

Summarizing

- Parallel programming is critical in science and engineering
- Relevant not only to supercomputers, but to any workstation/laptop
- Different platforms exhibit different architectural features, and different types and degrees of parallelism.

Summarizing

- Parallel programming is critical in science and engineering
- Relevant not only to supercomputers, but to any workstation/laptop
- Different platforms exhibit different architectural features, and different types and degrees of parallelism.
- Different forms of parallelism are handled via different programming paradigms:

  Vectorization – Data Parallelism – Multi-threading – Message Passing

Summarizing

- Parallel programming is critical in science and engineering
- Relevant not only to supercomputers, but to any workstation/laptop
- Different platforms exhibit different architectural features, and different types and degrees of parallelism.
- Different forms of parallelism are handled via different programming paradigms:

  Vectorization – Data Parallelism – Multi-threading – Message Passing
- More often than not, the best algorithmic solution depends on the specific paradigm.

Summarizing

- Parallel programming is critical in science and engineering
- Relevant not only to supercomputers, but to any workstation/laptop
- Different platforms exhibit different architectural features, and different types and degrees of parallelism.
- Different forms of parallelism are handled via different programming paradigms:

  Vectorization – Data Parallelism – Multi-threading – Message Passing
- More often than not, the best algorithmic solution depends on the specific paradigm.
- In a nutshell: **The burden is and will be on the programmer.** (Hence workshops like this one)

Vectorization

- A processor operates on data that resides in the registers.

Vectorization

- A processor operates on data that resides in the registers.
- Nowadays, one single register contains 2–16 scalars; the exact amount depends on the register width (128, 256, 512bits) and on the data type (single, double, ...).

Vectorization

- A processor operates on data that resides in the registers.
- Nowadays, one single register contains 2–16 scalars; the exact amount depends on the register width (128, 256, 512bits) and on the data type (single, double, ...).
- Hence, the processor operates on vectors!

Vectorization

- A processor operates on data that resides in the registers.
- Nowadays, one single register contains 2–16 scalars; the exact amount depends on the register width (128, 256, 512bits) and on the data type (single, double, ...).
- Hence, the processor operates on vectors!
- Let α_i, β_i and γ_i be scalars (for all i's).
  Even if you want to only perform $\gamma_1 := \alpha_1 + \beta_1$, the processor is equipped to simultaneously execute

  $$\{\gamma_1 := \alpha_1 + \beta_1, \quad \gamma_2 := \alpha_2 + \beta_2, \quad \ldots, \quad \gamma_w := \alpha_w + \beta_w\}$$

  w depend on the register width and the datatype of the scalars.

Vectorization

- A processor operates on data that resides in the registers.
- Nowadays, one single register contains 2–16 scalars; the exact amount depends on the register width (128, 256, 512bits) and on the data type (single, double, ...).
- Hence, the processor operates on vectors!
- Let α_i, β_i and γ_i be scalars (for all i's).
  Even if you want to only perform $\gamma_1 := \alpha_1 + \beta_1$, the processor is equipped to simultaneously execute

  $$\{\gamma_1 := \alpha_1 + \beta_1, \quad \gamma_2 := \alpha_2 + \beta_2, \quad \ldots, \quad \gamma_w := \alpha_w + \beta_w\}$$

  w depend on the register width and the datatype of the scalars.
- Maximum possible speedup: w.

SIMD paradigm

- SIMD = Single Instruction Multiple Data

SIMD paradigm

- SIMD = Single Instruction Multiple Data
- By hand: Streaming SIMD Extensions. Almost like Assembly.
  “MMX”, “SSE”, “SSE2”, “SSE3”, “SSE4”, “AVX”, “AVX2”, “AVX-512”, “NEON”, . . .

SIMD paradigm

- SIMD = Single Instruction Multiple Data
- By hand: Streaming SIMD Extensions. Almost like Assembly.
  “MMX”, “SSE”, “SSE2”, “SSE3”, “SSE4”, “AVX”, “AVX2”, “AVX-512”, “NEON”, . . .
- Autovectorization.

SIMD paradigm

- SIMD = Single Instruction Multiple Data
- By hand: Streaming SIMD Extensions. Almost like Assembly.
  “MMX”, “SSE”, “SSE2”, “SSE3”, “SSE4”, “AVX”, “AVX2”, “AVX-512”, “NEON”, . . .
- Autovectorization.
- Via directives: OpenMP.

Data-parallelism

- Closely related to vectorization: coarser granularity.

Data-parallelism

- Closely related to vectorization: coarser granularity.
- w might be even in the 100s and 1000s.
  Large speedups expected wrt sequential execution.

Data-parallelism

- Closely related to vectorization: coarser granularity.
- w might be even in the 100s and 1000s.
  Large speedups expected wrt sequential execution.
- Typical in accelerators and “many-core” architectures.

Data-parallelism

- Closely related to vectorization: coarser granularity.
- w might be even in the 100s and 1000s.
  Large speedups expected wrt sequential execution.
- Typical in accelerators and “many-core” architectures.
- Identify program flow that is identical for different data.

Data-parallelism

- Closely related to vectorization: coarser granularity.
- w might be even in the 100s and 1000s.
  Large speedups expected wrt sequential execution.
- Typical in accelerators and “many-core” architectures.
- Identify program flow that is identical for different data.
- OpenCL, Cuda (GPUs).

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.
- Feature: Main memory is shared among threads.
  Some of the cache memories might also be shared.

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.
- Feature: Main memory is shared among threads.
  Some of the cache memories might also be shared.
- No need for communication, BUT, need for synchronization.

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.
- Feature: Main memory is shared among threads.
  Some of the cache memories might also be shared.
- No need for communication, BUT, need for synchronization.
- Example: students working on a large calculation at the blackboard.

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.
- Feature: Main memory is shared among threads.
  Some of the cache memories might also be shared.
- No need for communication, BUT, need for synchronization.
- Example: students working on a large calculation at the blackboard.
- pthreads, OpenMP

Shared-memory parallelism – Multi-threading

- Multiple cores, multiple threads per core.
- Feature: Main memory is shared among threads.
  Some of the cache memories might also be shared.
- No need for communication, BUT, need for synchronization.
- Example: students working on a large calculation at the blackboard.
- pthreads, OpenMP
- Max speedup: Tipically in the 10s.

Multi-threading

- Iteration-based parallelism.

Multi-threading

- Iteration-based parallelism.
- Task-based parallelism.

Multi-threading

- Iteration-based parallelism.
- Task-based parallelism.
- Dependencies.

Multi-threading

- Iteration-based parallelism.
- Task-based parallelism.
- Dependencies.
- Scheduling; pinning; false-sharing.

Multi-threading

- Iteration-based parallelism.
- Task-based parallelism.
- Dependencies.
- Scheduling; pinning; false-sharing.
- Synchronization; race-conditions, mutual exclusion.

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.

- Data distributed among participants.

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.
- Data distributed among participants.
- Feature: Independent computing.
  Data has to be exchanged = communication.
  It is possible to overlap communication with computation.

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.
- Data distributed among participants.
- Feature: Independent computing.
  Data has to be exchanged = communication.
  It is possible to overlap communication with computation.
- Example: students working independently on (parts of) the same homework, and exchanging information on the phone.

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.
- Data distributed among participants.
- Feature: Independent computing.
  Data has to be exchanged = communication.
  It is possible to overlap communication with computation.
- Example: students working independently on (parts of) the same homework, and exchanging information on the phone.
- MPI: Message Passing Interface

Distributed-memory parallelism – Message passing

- Many nodes, each with its own local memory.
  From few to 10^5+ nodes.
- Data distributed among participants.
- Feature: Independent computing.
  Data has to be exchanged = communication.
  It is possible to overlap communication with computation.
- Example: students working independently on (parts of) the same homework, and exchanging information on the phone.
- MPI: Message Passing Interface
- Max speedup: Unbounded.

Message passing

- Data distribution

Message passing

- Data distribution
- Communication primitives.

Message passing

- Data distribution
- Communication primitives.
- Collective communication.

Message passing

- Data distribution
- Communication primitives.
- Collective communication.
- Point-to-point comuunication.

Message passing

- Data distribution
- Communication primitives.
- Collective communication.
- Point-to-point comuunication.
- One-sided communication.

Message passing

- Data distribution
- Communication primitives.
- Collective communication.
- Point-to-point comuunication.
- One-sided communication.
- Synchronous & asynchronous communication.

General pattern for parallel algorithms

1. Decomposition of the problem into sub-problems
2. Parallel solution of the sub-problems
3. Composition of the sub-solutions