Parallel Algorithms and Programming

Introduction

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

Organization of the course

Schedule

- 15 hours of lectures
- 6 hours of tutorials
- 12 hours of labs

Grading

- 50% graded labs (2 graded labs)
- 50% final exam

Useful links

- The slides
- The Moodle page

Teaching staff

- Thomas Ropars (thomas.ropars@univ-grenoble-alpes.fr)
- Sebastien Valat (sebastien.valat@atos.net)

Content of the course

- Parallel programming challenges
- Parallel architectures
- Shared-memory parallel programming
- Message-passing systems
- Linear algebra in message passing systems
- Map-Reduce (?)
- Fault tolerance (?)
- Performance evaluation (?)

Introduction

Outline of the lecture

- Why we need parallel systems?
- Applications of parallel computing
- Challenges of parallel computing
- Performance of parallel programs and metrics
- A case study of performance analysis
    - Sequential code: matrix multiplication

References

The content of this lecture is inspired by:

- The lecture notes of F. Desprez
- Introduction to Parallel Computing by B. Barney
- The lecture notes of K. Fatahalian
  - CS149: Parallel Computing @Standford
  - 15418: Parallel Computer Architecture and Programming @CMU
- Parallel Programming – For Multicore and Cluster System. T. Rauber, G. Rünger
- The lecture nodes of S. Lantz

Definition of parallel computing

```
Parallel computing uses multiple processing elements simultaneously to
solve a problem *quickly*.
```

Goal: Performance

- Solve problems faster
- Solve larger problems
- Get a better answer to a problem

Parallel vs Concurrent programming:

- Concurrent programming refers to multiple flows of executions executing concurrently.
- Concurrent programming does not necessarily imply parallelism

Definition of parallel computing

```
Parallel computing uses multiple processing elements simultaneously to
solve a problem *quickly*.
```

Goal: Performance

- Solve problems faster
- Solve larger problems
- Get a better answer to a problem

Parallel vs Concurrent programming:

- Concurrent programming refers to multiple flows of executions executing concurrently.
- Concurrent programming does not necessarily imply parallelism
    - Multiple threads executing on a single processor

Note that there are many possible definitions of *concurrent* programming. Not a very interesting debate here.

Why we need parallel computing?

Until 2000's

- To solve problems that a single processor cannot solve

Today

- To keep increasing the performance of systems
- **End of Moore's law**

Moore's law

An observation made by Gordon Moore in 1965 that the number of transistors in processors chips is doubling every 2 years (because of transistor size reduction)

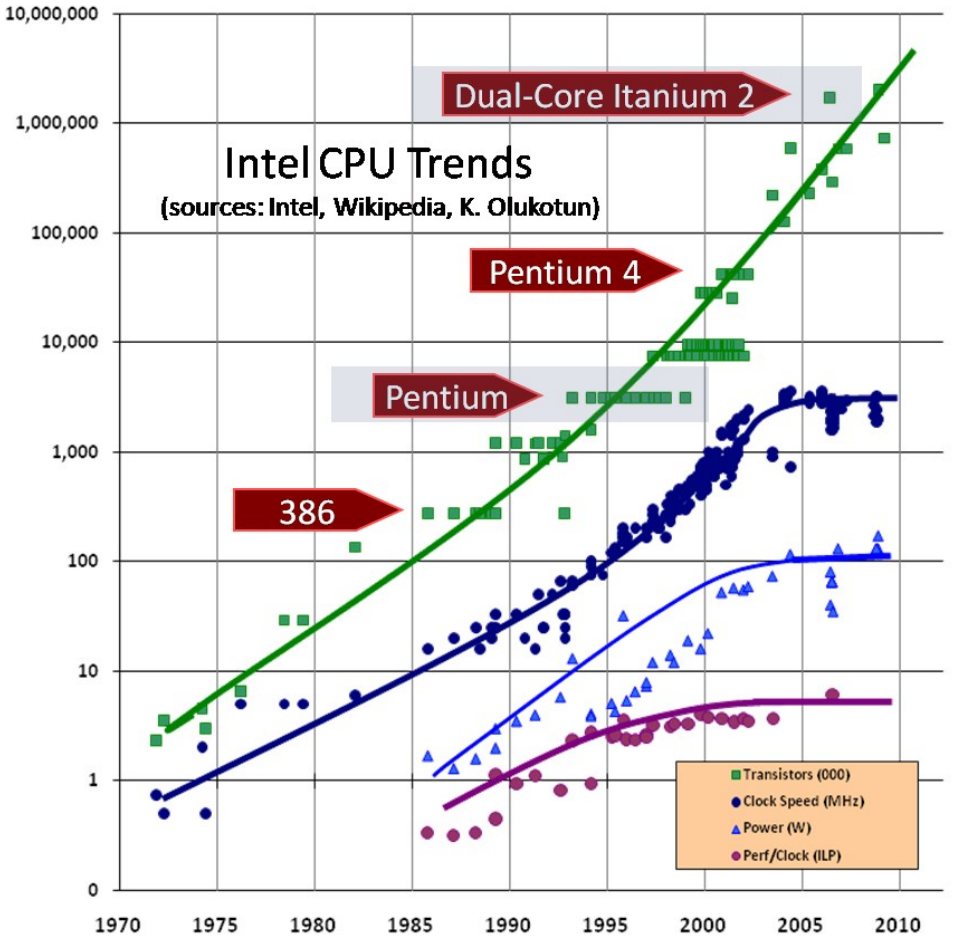
- It was later updated to *doubling every 18 months*

It was implying that:

- the clock frequency could be increased while keeping the power consumption constant (Dennard scaling).

```
To get more performance, you just had to wait for the next
generation of processors
```

The end of the free lunch (H. Sutter)



Lessons learned from previous graph

Lessons learned from previous graph

Moore's law was having two side effects:

- Allowing increasing the clock frequency
- More transistors were allowing more complex logic to be implemented
  - Optimizing the execution = doing more work per processor cycle
  - **Instruction Level Parallelism** (ILP)

In the recent years

- No frequency improvement
- No further improvement in ILP

Lessons learned from previous graph

Moore's law was having two side effects:

- Allowing increasing the clock frequency
- More transistors were allowing more complex logic to be implemented
  - Optimizing the execution = doing more work per processor cycle
  - **Instruction Level Parallelism** (ILP)

In the recent years

- No frequency improvement
- No further improvement in ILP

Current state

The number of transistors per chip keeps increasing

- Increase in the number of processor cores
- Trend: large number of simpler cores

```
To run faster, a program has to be *parallel*
```

About ILP

ILP is a measure of the number of instructions per cycle.

Several techniques to increase the number of instructions per cycle:

- Pipelining
- Branch prediction
- Out-of-order execution
- Superscalar architecture
- Vector operations

```
The expected performance improvement is limited by the parallelism
that can be found in the sequence of instructions of the program to
execute
```

About ILP

A simple C code:

```
a = x*x + y*y + z*z
```

About ILP

A simple C code:

```
a = x*x + y*y + z*z
```

5 operations to be executed (ignoring data movements)

ILP

About ILP

A simple C code:

```
a = x*x + y*y + z*z
```

5 operations to be executed (ignoring data movements)

ILP

time	operations
1	`o1=x*x; o2=y*y; o3=z*z`
2	`o4=o1+o2`
3	`o5=o4+o3`

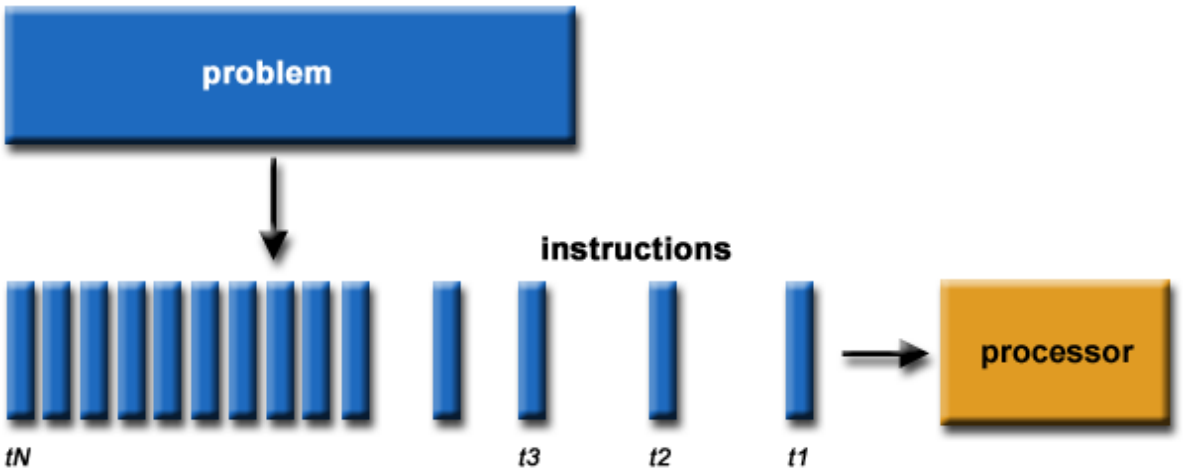
At most 3 operations can be executed in parallel

> Most available ILP is exploited by issuing 4 instructions per cycle

Exemple from K. Fatahalian

What is parallel computing?

Serial computing

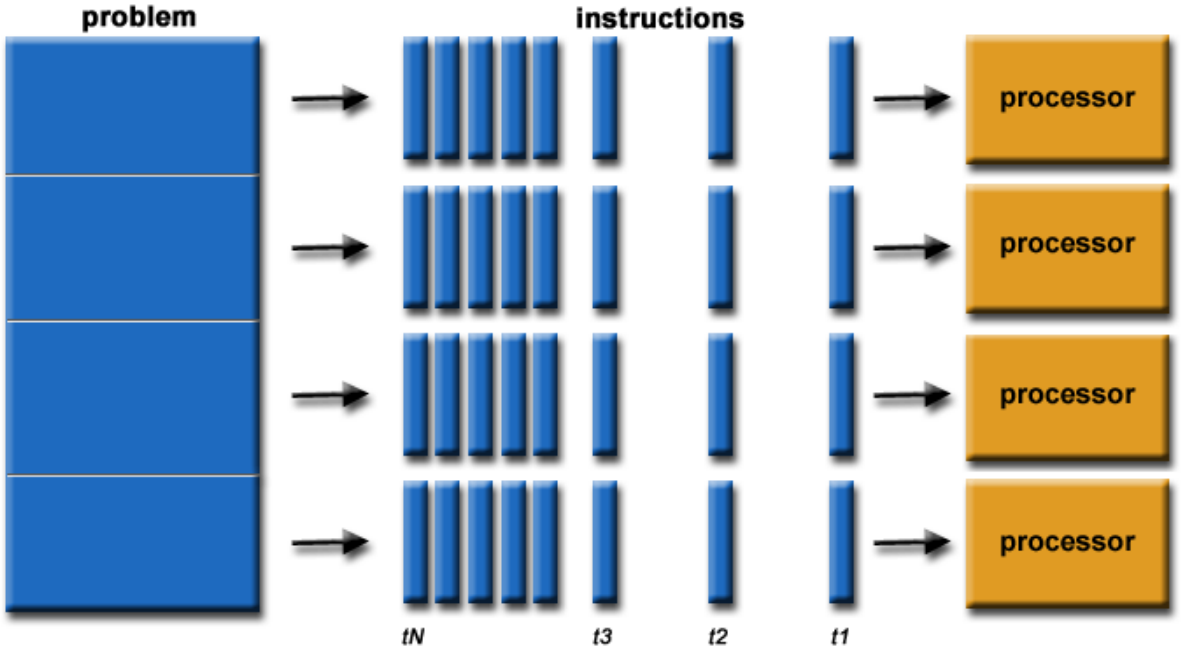


- One instruction is executed at a time (ignoring ILP)

Illustration by B. Barney

What is parallel computing?

Parallel computing



- A problem is broken into multiple parts
- Instructions from multiple parts are executed in parallel
- An overall control/coordination mechanism should be employed

Illustration by B. Barney

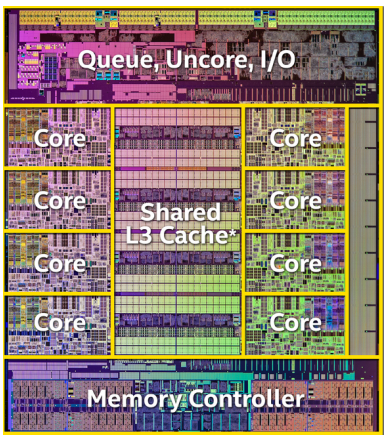
Parallel systems

Desktop computer (and mobile devices)

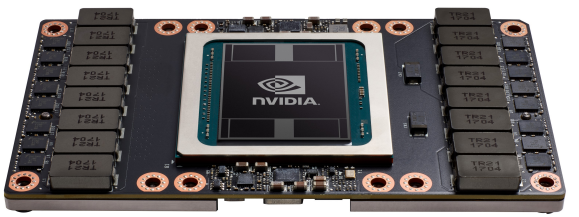
- Multicore processors
- GPUs

Distributed architectures

- Cluster of commodity nodes
- Cloud computing
- Supercomputers (High Performance Computing)



Intel Haswell



NVIDIA Tesla v100

About distributed architectures

```
Large set of computing resources that communicate through a network
```

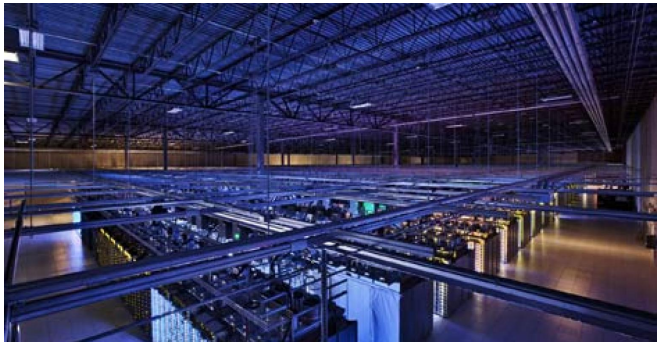
- Allow building very large scale systems at a reasonable price (Scale out)
  - Is in general less expensive than scaling up (updating the hardware with more powerful -- and more expensive -- components)

Level of parallelism

- Multicore processors: 10's of processing cores
- GPUs: 1000's of processing cores
- Supercomputers: Millions of processing cores (CPU and GPU)
- Cloud computing infrastructures: Millions of servers

Distributed infrastructures

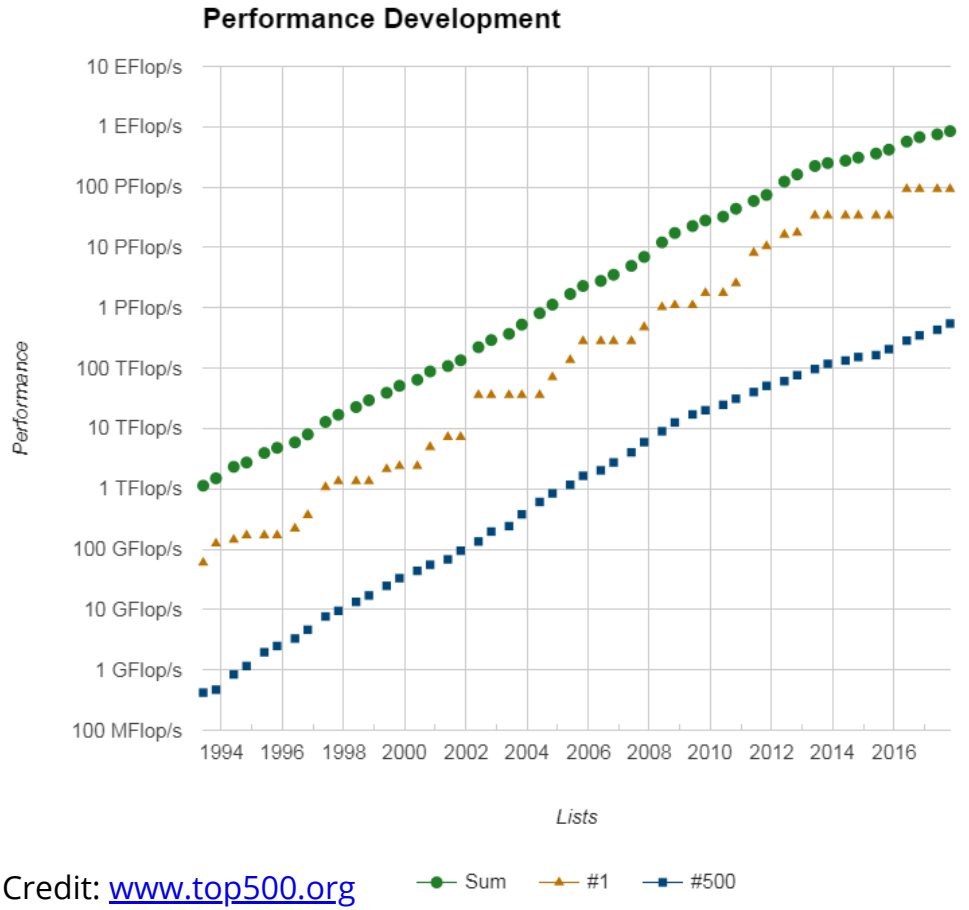
Summit Supercomputer (Oak Ridge National Lab)



Google data center

Statistics from TOP 500

Ranking of the 500 hundred most powerful supercomputers



Credit: www.top500.org

Applications of parallel computing

Use of parallel computing

With the evolution of processors, going parallel is the only way to significantly improve the performance of any application

Type of applications requiring large amount of computation

- Numerical simulations
- Data analysis
- Artificial intelligence
- etc.

Numerical simulations

- Simulation can be used to study complex systems/phenomenons
- Without simulation:
  - Study on paper
  - Perform real experiments

Advantages of simulation

- Reduces costs and risks
  - Plane crash
  - Nuclear experiments
  - Drugs
- Allows simulating phenomena that are hard to study in reality
  - Tsunami
  - Climate evolution

Numerical simulations examples

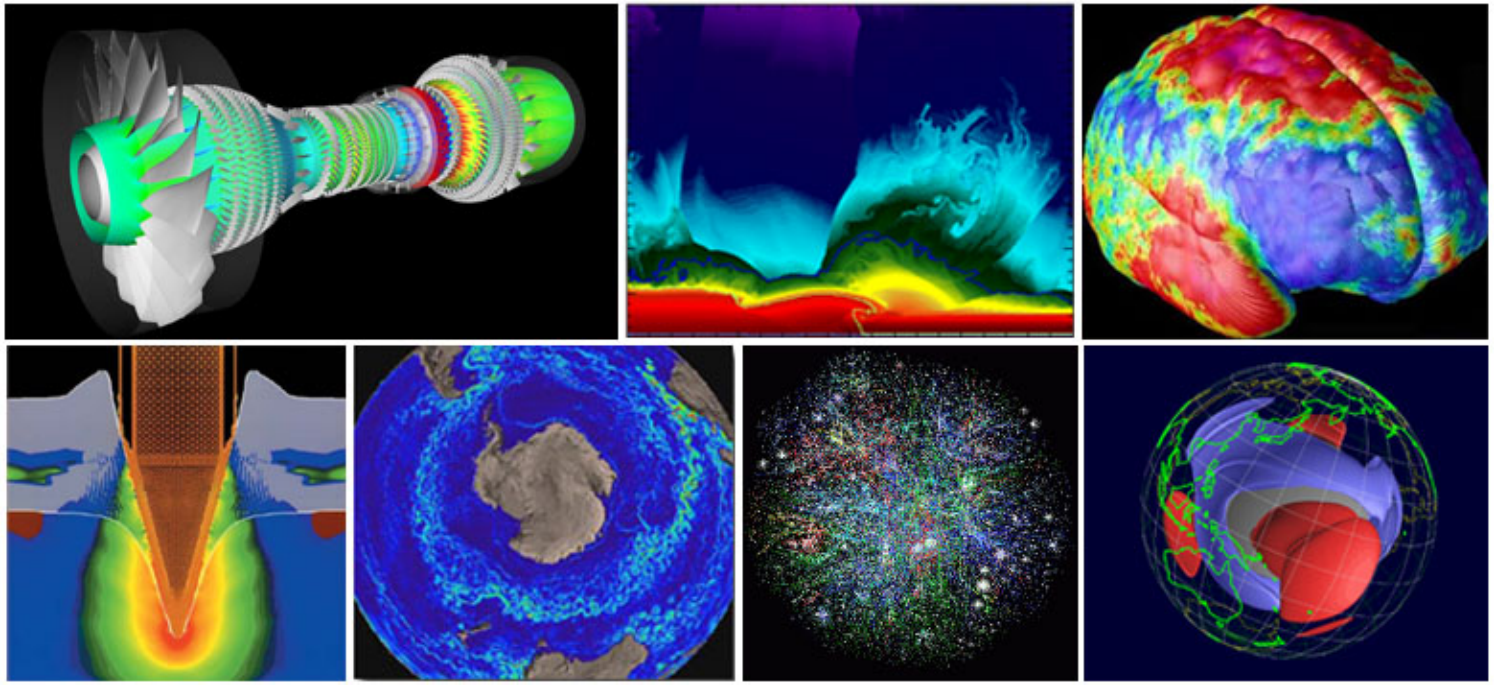


Illustration by B. Barney

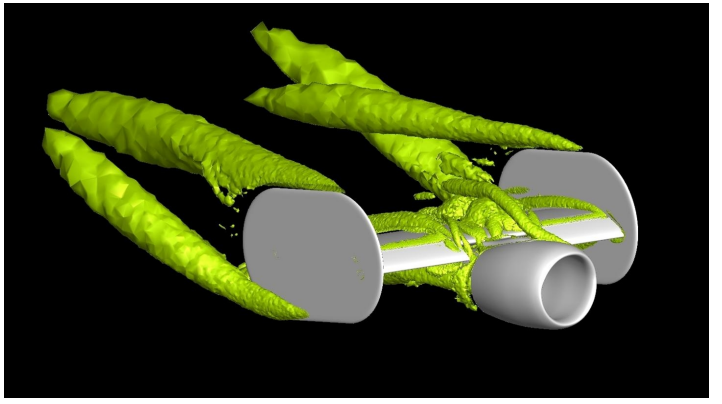
Domains where numerical simulations is used

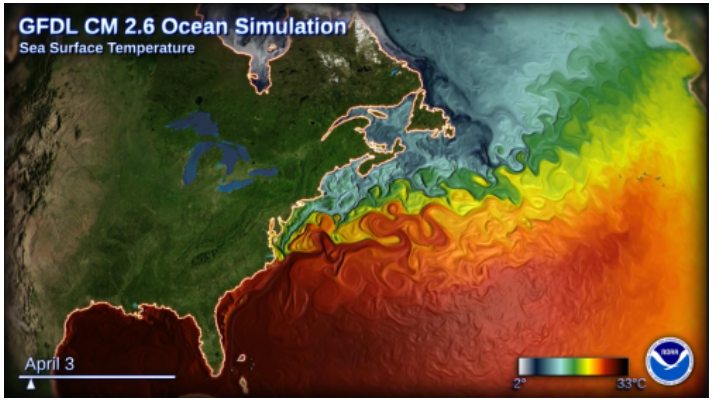
Science

- Climate prediction
- Molecular science
- Seismology
- Bio-science

Engineering

- Engine design
- Airplane design
- Structural modeling



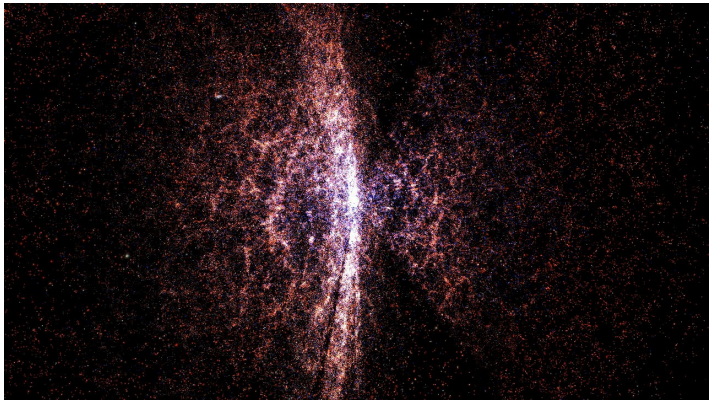


Data analysis (Big Data)

Parallel systems can also be used to process huge amounts of data.

Domains of application

- DNA analysis
- Data generated by large equipments
  - Telescopes
  - Large Hadron Collider (CERN)
- Data from the Web
  - Analysis of data from social networks
  - Search engines



Credits: NASA/University of Chicago and Adler Planetarium and Astronomy Museum

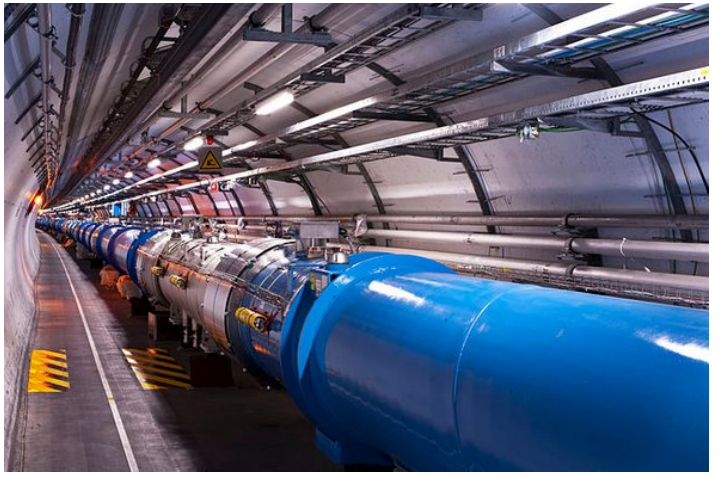
Some numbers

Big Data

- Every 2 days, we create as much information as we did since 2013
- 30M messages posted on Facebook every minute
- 570 new web sites every minute

Large Hadron Collider

- The most powerful instrument ever built to investigate elementary particles
- 40 PB of raw data per second during an experiment
- 10 PB of useful data per year



Credit: https://www.slideshare.net/BernardMarr/big-data-25-facts

Credit: Maximilien Brice (CERN)/Wikimedia Commons

Entertainment industry

Computer-generated imagery

- Advanced graphics
  - Animation movies
- Virtual reality

2001, Pixar, Monster Inc.: 250 servers with 14 processors (3500 processors)

Artificial intelligence

Machine learning

> Build a mathematical model out of training data

- Classification
- Clustering
- Regression

Applications

- Recommendation systems
- Anomaly detection
- Speech recognition
- etc.

The case of deep learning

- Execution on clusters of GPUs (from 10's to 1000's of nodes)
- Training a neural network = Operations on large matrices

Challenges of Parallel Computing

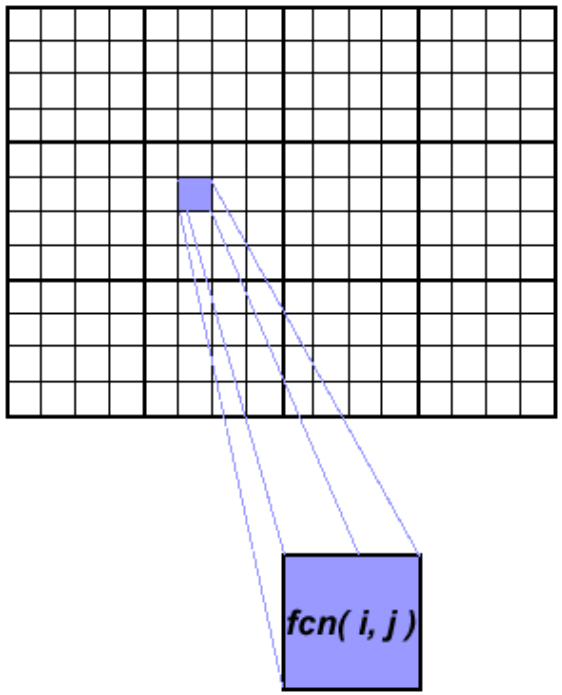
A first example of parallel computation

The problem : Array processing

- 2-dimensional array of elements
- A function has to be applied to each element

The sequential code

```
for(j=0; j< n; j++){
  for(i=0; i<n; i++){
    a[i][j] = fcn(a[i][j])
  }
}
```



Credit: Blaise Barney, Lawrence Livermore National Laboratory

A first example of parallel computation

```
The computation of each element is independent. This is an
 *embarrassingly parallel problem*
```

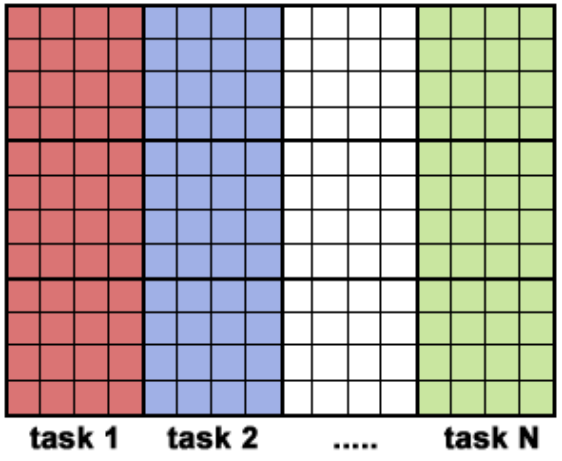
Solution

- Create N tasks
- Each task runs the computation for one sub-part.
- Best distribution scheme depends on the programming language
    - Depending on the language the data can be arranged differently in memory
    - Goal: Improving memory access locality

A first example of parallel computation

Parallel code for one task

```
for(j=my_start; j< my_end; j++){
  for(i=0; i<n; i++){
    a[i][j] = fcn(a[i][j])
  }
}
```

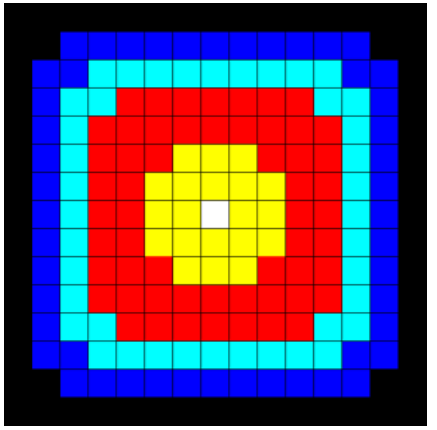


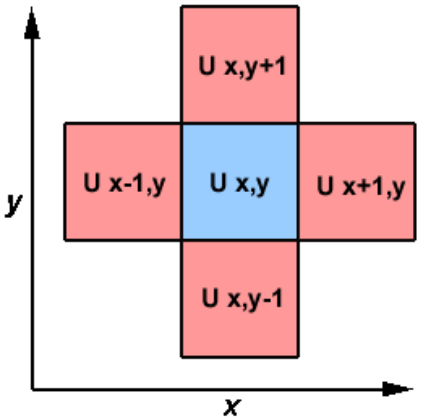
About the division of work

- The proposed division is good for Fortran programs but not for C programs
- To have good locality in C, each task should be given a set of lines

Second example

- 2-dimensional array of elements
  - Each element is the temperature at this position in the 2D space
- A time-stepping algorithm is used to compute the evolution of the temperature
- At each time step:
  - The temperature of the cell depends on the one of its neighbors






```
U[x][y] = U[x][y] + C_x * (U[x-1]
[y] + U[x+1][y] - 2 * U[x][y]) +
C_y * (U[x][y-1] + U[x][y+1] - 2 *
U[x][y])
```

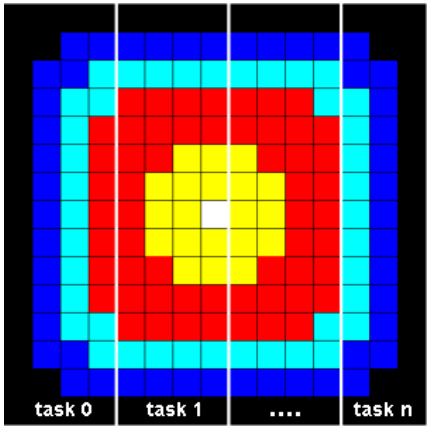
Credit: Blaise Barney, Lawrence Livermore National Laboratory

Second example

Problem

- Data dependencies between the computation

- Same solution as before
  - Create N tasks
  - Divide the work into sub-tasks
- But:
  - **Need to communicate the values at the border at the end of each iteration**



Last example

Problem

- N+1 persons
  - N students
  - 1 professor
- W exam sheets to evaluate
  - Giving a grade to each exam sheet
  - We assume grading one exam sheet is a short task (Multiple-choice questions)
  - We assume that W is unknown (but large)
  - The professor has all the exam sheets initially
- **Final goal:**
  - Compute a distribution (histogram) of the grades

Last example (First solution)

Actions of the professor

```
- Count the number of exam sheets
- Give W/N sheets to each student
- Wait
- Get back the graded sheets
- Compute the histogram
```

Actions of the students

```
- Wait to receive a batch of exam sheets
- Grade the batch
- Give the graded batch back to the professor
```

Last example (First solution)

Problems

- Counting the number of exam sheets may take time
- Computing the histogram may take time
- Probably not all students grade at the same speed

Last example (solutions)

Counting the number of exam sheets may take time

Last example (solutions)

Counting the number of exam sheets may take time

- Divide the work into pre-defined fixed-size batches
- Should the size of the batches be large or small?

Last example (solutions)

Counting the number of exam sheets may take time

- Divide the work into pre-defined fixed-size batches
- Should the size of the batches be large or small?
  - Large batches:
    - Low number of synchronization/communication with the professor
    - Risk of imbalance for the last batches (some students may remain idle at the end)
  - Small batches:
    - Imbalance problem becomes negligible
    - High number of synchronization/communication with the professor

Last example (solutions)

Computing the histogram may take time

Last example (solutions)

Computing the histogram may take time

- Distribute the work to be done among the students

Last example (solutions)

Computing the histogram may take time

- Distribute the work to be done among the students
  - Organize the students so that they compute the histogram in parallel
    - We can organize the students in a binary tree
    - This is a **reduce** collective operation

Probably not all students grade at the same speed

Last example (solutions)

Computing the histogram may take time

- Distribute the work to be done among the students
  - Organize the students so that they compute the histogram in parallel
    - We can organize the students in a binary tree
    - This is a **reduce** collective operation

Probably not all students grade at the same speed

- Need for load balancing to avoid that slow students impact the whole process

Last example (solutions)

Computing the histogram may take time

- Distribute the work to be done among the students

  - Organize the students so that they compute the histogram in parallel
    - We can organize the students in a binary tree
    - This is a **reduce** collective operation

Probably not all students grade at the same speed

- Need for load balancing to avoid that slow students impact the whole process

  - Using small batches can solve the problem
    - Lower the risk to have to wait for a slow student at the end
    - The fast students will process more batches

Last example (new algorithm)

Actions of the professor

```
while exam sheets to be corrected:
    give a batch to next student
wait until the final distribution has been computed
```

Actions of the students

```
while exam sheets to be corrected:
    get a batch from the professor
    grade the batch
    update the local histogram
Participate in the global reduction for the histogram
```

Challenges for parallel programming

Making a program parallel

- Find parallelism in the problem
- Divide the work into multiple tasks
- Study the dependencies between the tasks
- Study the need for communication and synchronization
- Study the need for load balancing

Challenges for parallel programming

Making a program parallel

- Find parallelism in the problem
- Divide the work into multiple tasks
- Study the dependencies between the tasks
- Study the need for communication and synchronization
- Study the need for load balancing

Is this the whole story?

Challenges for parallel programming

Making a program parallel

- Find parallelism in the problem
- Divide the work into multiple tasks
- Study the dependencies between the tasks
- Study the need for communication and synchronization
- Study the need for load balancing

Is this the whole story?

```
We also need to take into account the characteristics of the
 *underlying hardware*
```

Challenges for parallel programming

- The best way of parallelizing a computation depends on the target platform
  - Levels of parallelism
  - Storage hierarchy
  - Topology
  - Communication model

Different levels of parallelism

- Example: A cluster of commodity machines
  - Multiple nodes
  - Each node has several processors (multi-socket)
  - Each processors has multiple cores
    - Each core implements simultaneous multi-threading
    - Each core can run vector operations

Challenges for parallel programming

Storage hierarchy

- Each level has different performance and capacity
- Data transfers have a huge impact on the performance of programs

	latency	bandwidth
L1 cache	ns	KB
L3 cache	10's ns	MB
DRAM	100 ns	10 GB
SSD	10 us	100 GB
Disk	10 ms	TB
PFS	s	PB

Search for "Latency numbers every programmer should know"

Challenges for parallel programming

Topology

- Inside a node
  - Cores that share some levels of cache can communicate faster
  - Cores may have faster access to some parts of the main memory
    - Non Uniform Memory Access (NUMA)

- At the network level
  - Faster to communicate with neighbors that with distant nodes
  - Different network topologies
    - Tree
    - Mesh
    - Torus
    - etc.

Performance of parallel programs

Performance of parallel programs

To analyze the performance of parallel programs, metrics need to be used.

Two important points to evaluate

- The absolute performance (execution time, FLOPS)
- The scalability = evolution of the performance when increasing the number of computing resources
  - Speedup
  - *Strong* scaling
  - *Weak* scaling

About FLOPS

Floating point operations per seconds

Interesting because scientific computations require floating point operations.

- Even in scientific applications, many instructions are not floating point operations
  - Load/Store
  - Control
  - Integer arithmetics
  - Etc.
- Allows evaluating the amount of useful work

2 usages

- Can be used to evaluate the capacity of hardware
  - Defines the peak performance of a computing system
- Can be used to evaluate the efficiency of an algorithm on a given hardware

CPI (Cycles per instruction)

- Another metric that can give an idea of how well a program is behaving on a given hardware

```
CPI = total execution time / total number of instructions
```

- Measures how often a processor stalls
- For instance, can indicate a bad use of the caches

Question:

- Is it possible to achieve $CPI < 1$ on one processor core?

Speedup

Speedup

For a sequential execution time T_s , and a parallel execution time T_p:

$$Speedup = \frac{T_s}{T_p}$$

> When executing on N computing resources, we would like that the speedup to be N

Speedup

Speedup

For a sequential execution time T_s , and a parallel execution time T_p:

$$Speedup = \frac{T_s}{T_p}$$

> When executing on N computing resources, we would like that the speedup to be N

Question: Can the speedup be more than N?

Speedup

Speedup

For a sequential execution time T_s , and a parallel execution time T_p:

$$Speedup = \frac{T_s}{T_p}$$

> When executing on N computing resources, we would like that the speedup to be N

Question: Can the speedup be more than N?

- **Super-linear speedup**
- May happen in different cases:
  - Less instruction executed in the parallel version of the code (search)
  - Better usage of the cache/memory/storage hierarchy

Amdahl's law

```
Speedup of parallel code is limited by the sequential part of the
code
```

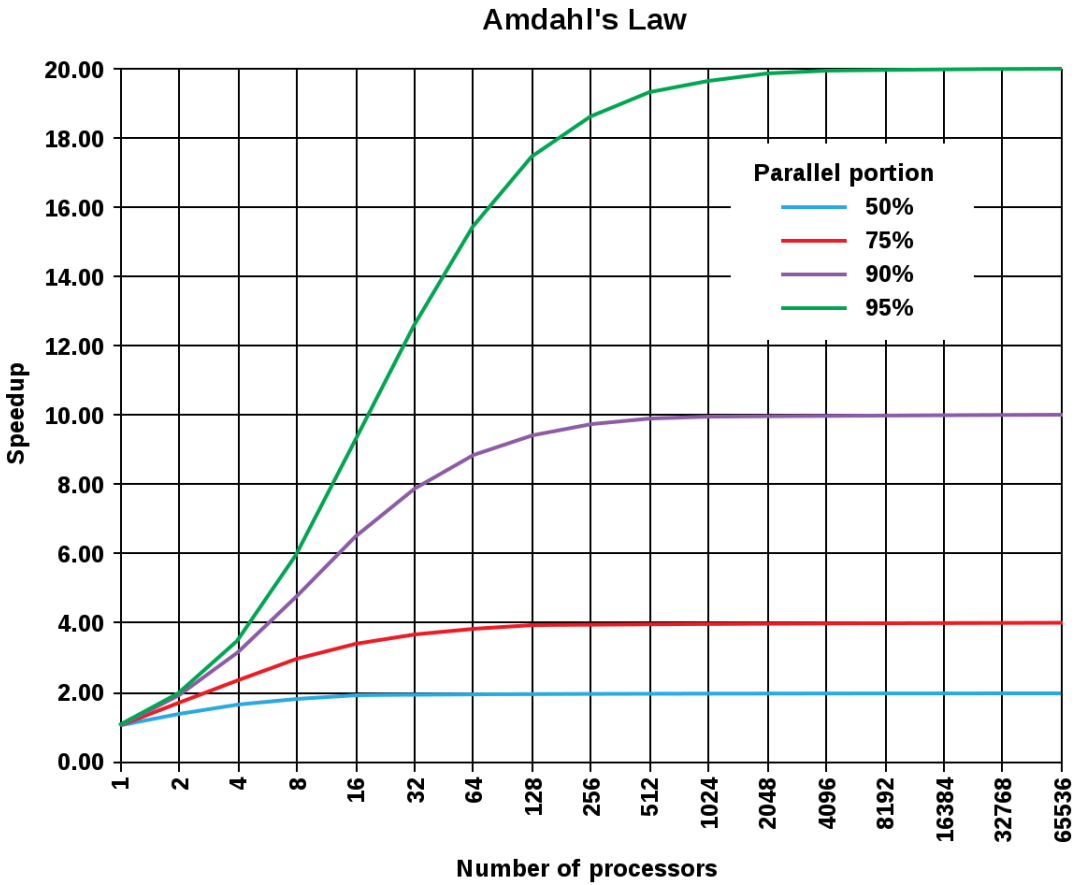
In a program where a fraction P of the code is parallel, the maximum speedup is:

$$Speedup_{max} = \frac{1}{1-P} = \frac{1}{S}$$

For a program running on N computing resources, with S being the serial fraction:

$$Speedup(N) = \frac{1}{\frac{P}{N} + S}$$

Amdahl's law



Credit: Daniels220 at English Wikipedia

Scalability

```
*Scalability* evaluates whether the performance improvement is
proportional to the number of processors used.

*Strong scaling*: Compute a problem N times **faster** using N
computing resources

*Weak scaling*: Compute a problem N times **bigger** in the same
amount of time using N computing resources
```

Limit of strong scaling

Scalability

```
*Scalability* evaluates whether the performance improvement is
proportional to the number of processors used.

*Strong scaling*: Compute a problem N times **faster** using N
computing resources

*Weak scaling*: Compute a problem N times **bigger** in the same
amount of time using N computing resources
```

Limit of strong scaling

- Amdahl's law

Limit of weak scaling

Scalability

```
*Scalability* evaluates whether the performance improvement is
proportional to the number of processors used.

*Strong scaling*: Compute a problem N times **faster** using N
computing resources

*Weak scaling*: Compute a problem N times **bigger** in the same
amount of time using N computing resources
```

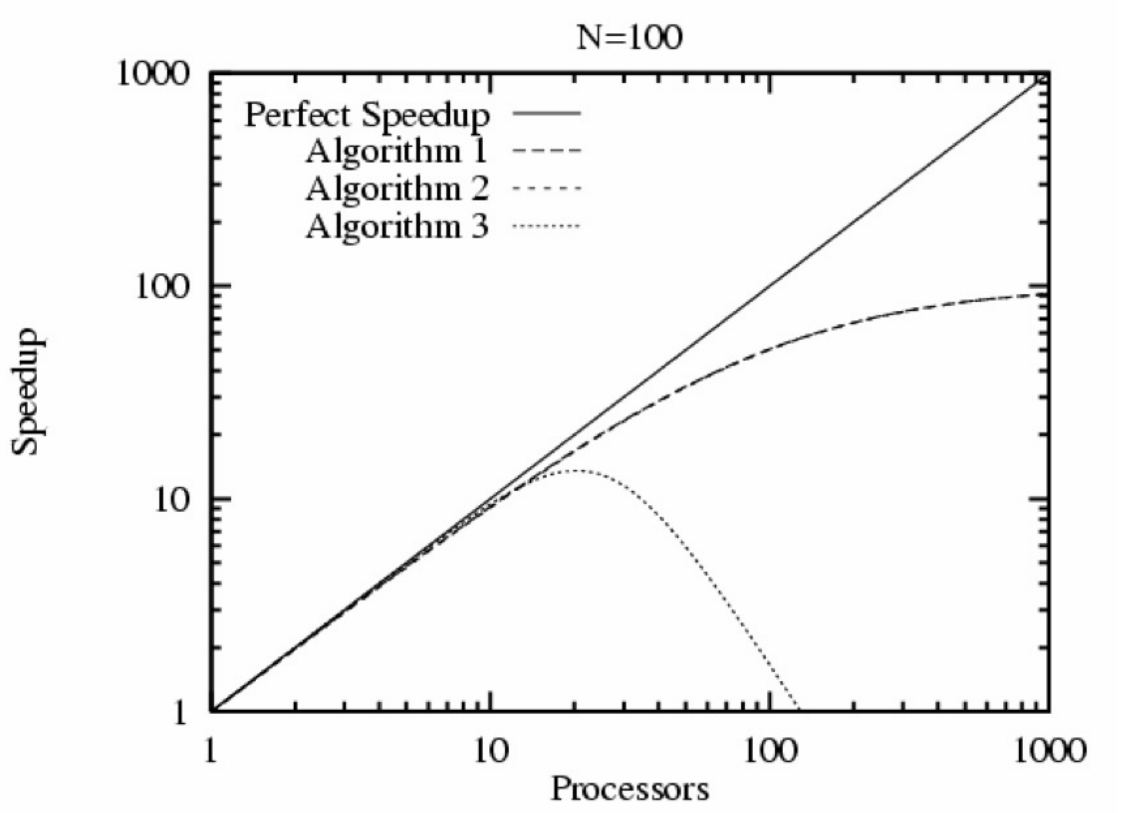
Limit of strong scaling

- Amdahl's law

Limit of weak scaling

- Works well if the amount of serial work and the amount of communication remains constant when the problem size increases

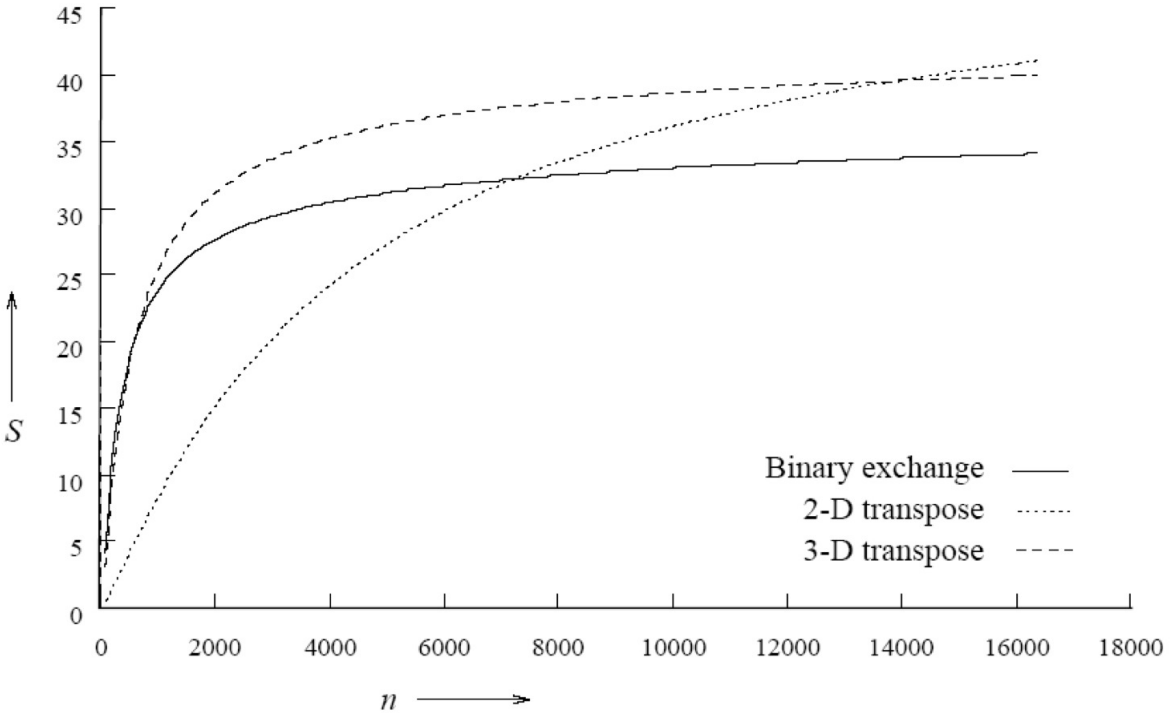
Illustrations of scalability



At small scale, all the evaluated algorithms perform equally well.

Illustrations of scalability

- Speedup as a function of the problem size (64 procs)



Choosing the best algorithm depends on the case.

Some comments about scalability and speedup

- Speedup should be computed based on the most efficient sequential algorithm
  - A parallel algorithm might not perform well sequencially

- At the end the most important is the absolute performance
  - A very slow algorithm that scales well is not interesting

A case study of performance analysis

Matrix product

Motivation

Matrix multiplication

- An important kernel in several numerical applications
  - A bottleneck of many applications
- Very well studied algorithm

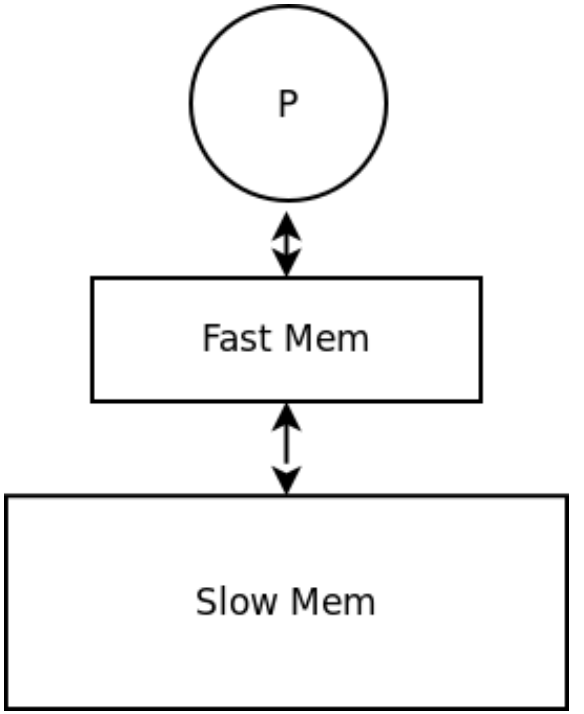
Study of the sequential performance on a single processor

- Processors are usually not used efficiently
- Most performance loss is usually due to data movements
- **We would like to understand this point**

A simplified model (for hardware)

- Two levels in the memory hierarchy
  - A fast small memory
  - A slow large memory

- The processor operates on data stored in the fast memory



A simple performance model

Parameters of the algorithm:

- m: Number of memory words moved from slow to fast memory
- f: Number of floating point operations

Parameters of the hardware:

- t_m: Time to move one word from slow to fast memory
- t_f: Time per floating point operation ($t_f << t_m$)

Assumption:

- All data initial in slow memory
- The cost to access data in fast memory is 0

Basic computation

We define T_{min} as the minimum execution time assuming all data are in fast memory:

$$T_{min} = f \times t_f$$

We can compute the effective execution time T:

$$T = f \times t_f + m \times t_m$$

Basic computation

We define T_{min} as the minimum execution time assuming all data are in fast memory:

$$T_{min} = f \times t_f$$

We can compute the effective execution time T:

$$T = f \times t_f + m \times t_m$$

Computational intensity

The computational intensity $q = \frac{f}{m}$ gives an estimation of how well the resources of a processor are used.

$$T = f \times t_f * (1 + \frac{t_m}{t_f} \times \frac{1}{q})$$

Basic computation

We define T_{min} as the minimum execution time assuming all data are in fast memory:

$$T_{min} = f \times t_f$$

We can compute the effective execution time T:

$$T = f \times t_f + m \times t_m$$

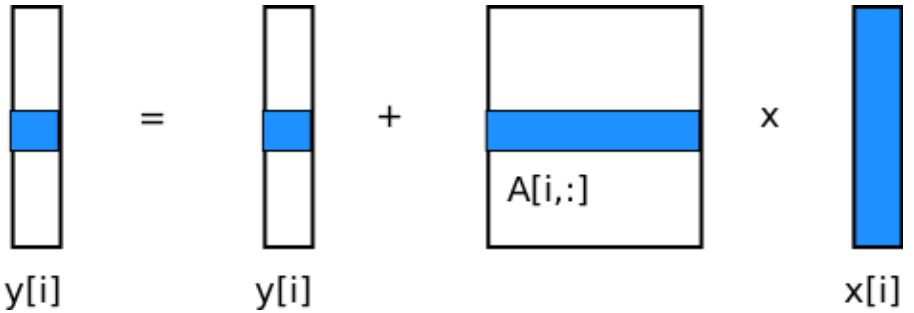
Computational intensity

The computational intensity $q = \frac{f}{m}$ gives an estimation of how well the resources of a processor are used.

$$T = f \times t_f * (1 + \frac{t_m}{t_f} \times \frac{1}{q})$$

A larger q leads to smaller execution time

Matrix-vector product




```
// y = y + A * x

for(i=0; i< n; i++){
    for(j=0; j<n; j++){
        y[i] = y[i] + A[i,j] * x[j]
    }
}
```

Matrix-vector product analysis

```
Transfer x[1:n] to fast memory
Transfer y[1:n] to fast memory
for(i=0; i< n; i++){
    Transfer row i of A to fast memory
    for(j=0; j<n; j++){
        y[i] = y[i] + A[i,j] * x[j]
    }
}
Transfer y[1:n] to slow memory
```

- Words moved from slow to fast memory: $m = n^2 + 3 \times n$
- Floating point operations: $f = 2 \times n^2$

- Computational intensity: $q = \frac{f}{m} \approx 2$

Matrix-vector product analysis

```
Transfer x[1:n] to fast memory
Transfer y[1:n] to fast memory
for(i=0; i< n; i++){
    Transfer row i of A to fast memory
    for(j=0; j<n; j++){
        y[i] = y[i] + A[i,j] * x[j]
    }
}
Transfer y[1:n] to slow memory
```

- Words moved from slow to fast memory: $m = n^2 + 3 \times n$
- Floating point operations: $f = 2 \times n^2$

- Computational intensity: $q = \frac{f}{m} \approx 2$

The matrix-vector product is limited by the speed of the slow memory

Simplifying assumptions

Simplifying assumptions

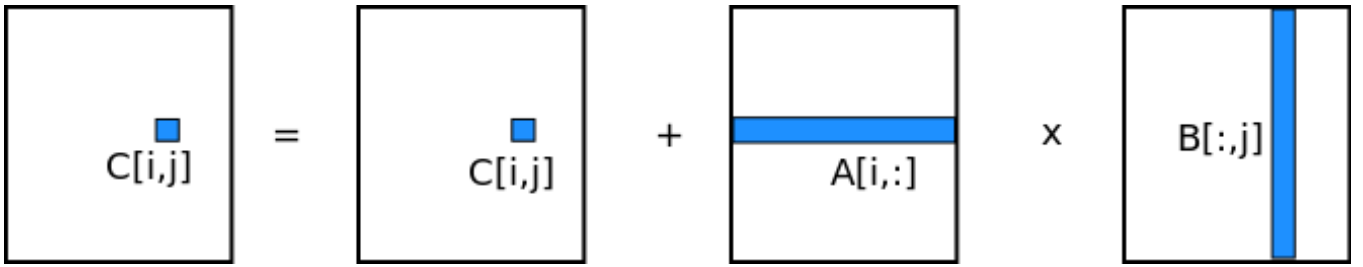
We assumed that 3 vectors can feet in the fast memory

- If we consider the processor registers, this is unrealistic
- If we consider the caches, it can be ok
  - But note that even the latency of accesses to L1 cache is not 0

We have ignored some level of parallelism

- Memory accesses and floating points operations might be run in parallel
  - The prefetchers might bring data to fast memory before we need them.
- The processor might be able to execute floating point operations in parallel (superscalar)

Matrix multiplication




```
// C = C + A * A

for(i=0; i< n; i++){
    for(j=0; j<n; j++){
        for(k=0; k<n; k++){
            C[i,j] = C[i,j] + A[i,k] * B[k,j];
        }
    }
}
```

Matrix multiplication (naive version)

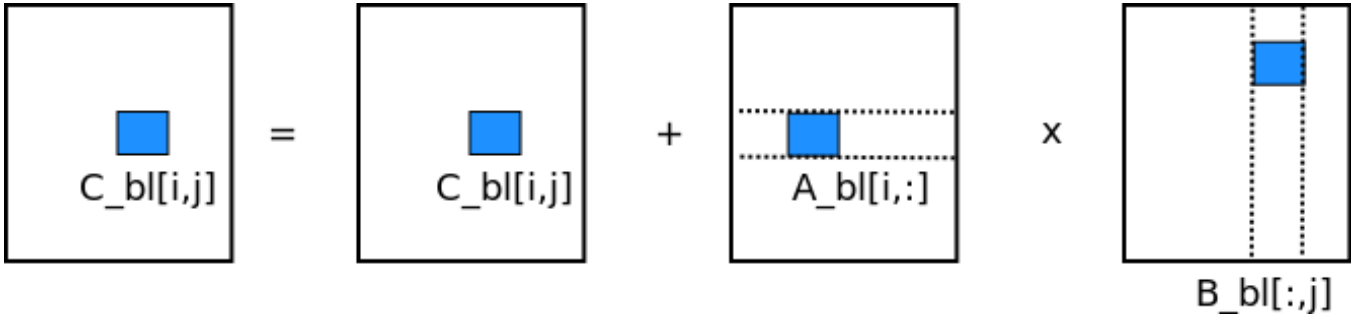
```
for(i=0; i< n; i++){
    Transfer row i of A to fast memory
    for(j=0; j<n; j++){
        Transfer C[i,j] to fast memory
        Transfer column j of B to fast memory
        for(k=0; k<n; k++){
            C[i,j] = C[i,j] + A[i,k] * B[k,j];
        }
        Write C[i,j] to slow memory
    }
}
```

- Words moved from slow to fast memory: $m = 3 \times n^2 + n^3$
  - Read column of B n^2 times : n^3
  - Read row of A n times : n^2
  - Read and write matrix C: $2 \times n^2$
- Floating point operations: $f = 2 \times n^3$
- Computational intensity: $q = \frac{f}{m} \approx 2$ (when n is large)
  - Same computational intensity as the matrix-vector product

Block matrix multiplication

Idea: Divide the matrix in blocks to increase the compute intensity

- A, B, C are matrices of size $n \times n$
- Divide them in blocks of size $b \times b$ with $b = \frac{n}{N}$



Block matrix multiplication

```
for(I=0; I< N; I++){
    for(J=0; j<N; J++){
        Transfer block C_bl[I,J] to fast memory
        for(K=0; K<N; K++){
            Transfer block A_bl[I,K] to fast memory
            Transfer block B_bl[K,J] to fast memory
            // matrix multiplication applied to the blocks
            C_bl[I,J] = C_bl[I,J] + A_bl[I,K] * B_bl[K,J];
        }
        Write C_bl[I,J] to slow memory
    }
}
```

Block matrix multiplication

- Words moved from slow to fast memory:
  - Read block of B "N^3" times: $N^3 \times b^2 = N^3 \times \left(\frac{n}{N}\right)^2 = N \times n^2$
  - Read block of A "N^3" times: $N \times n^2$
  - Read and write block of C "N^2" times: $2 \times N^2 \times b^2 = 2 \times n^2$

$$m = (2 + 2 \times N) \times n^2$$

- Floating point operations: $f = 2 \times n^3$

- Computational intensity: $q = \frac{f}{m} \approx \frac{n}{N} = b$ (when n is large)

Block matrix multiplication

- Words moved from slow to fast memory:
  - Read block of B "N^3" times: $N^3 \times b^2 = N^3 \times \left(\frac{n}{N}\right)^2 = N \times n^2$
  - Read block of A "N^3" times: $N \times n^2$
  - Read and write block of C "N^2" times: $2 \times N^2 \times b^2 = 2 \times n^2$

$$m = (2 + 2 \times N) \times n^2$$

- Floating point operations: $f = 2 \times n^3$

- Computational intensity: $q = \frac{f}{m} \approx \frac{n}{N} = b$ (when n is large)

  - Computational intensity is proportional to the block size
  - Can increase the block size to improve performance (as long as 3 blocks fit in fast memory)

Lessons learned

- A careful use of the cache can improve the performance of algorithms
    - More generally, reducing data movements can have a significant impact on algorithms performance
- Different algorithms can lead to performance that are significantly different
    - This will also be true for parallel algorithms

Conclusion

Single-thread performance increases very slowly

- To improve performance, the programs have to be parallel

Many problems require much more computing power than what a single processor core can provide

- Parallel systems can feature huge numbers of processing resources

Writing parallel programs is a challenging task , it requires:

- Finding parallelism in the applications
- Designing appropriate synchronization between tasks while limiting the data movement
- Dealing with work unbalance
- Designing solutions that can make best use of the underlying hardware

Parallel architectures can be very complex with multiple level of parallelism