Introduction to Parallel Computing

Jorge Barbosa
University of Porto
www.fe.up.pt/~jbarbosa

Introduction

Until recently:

CPU Gflop/s increased by increasing frequency

"the more ticks you have per second, the more work will get done"

Why not push the clock faster?

Speed/power tradeoff
It's no longer worth the cost in terms of power consumed and heat dissipated.

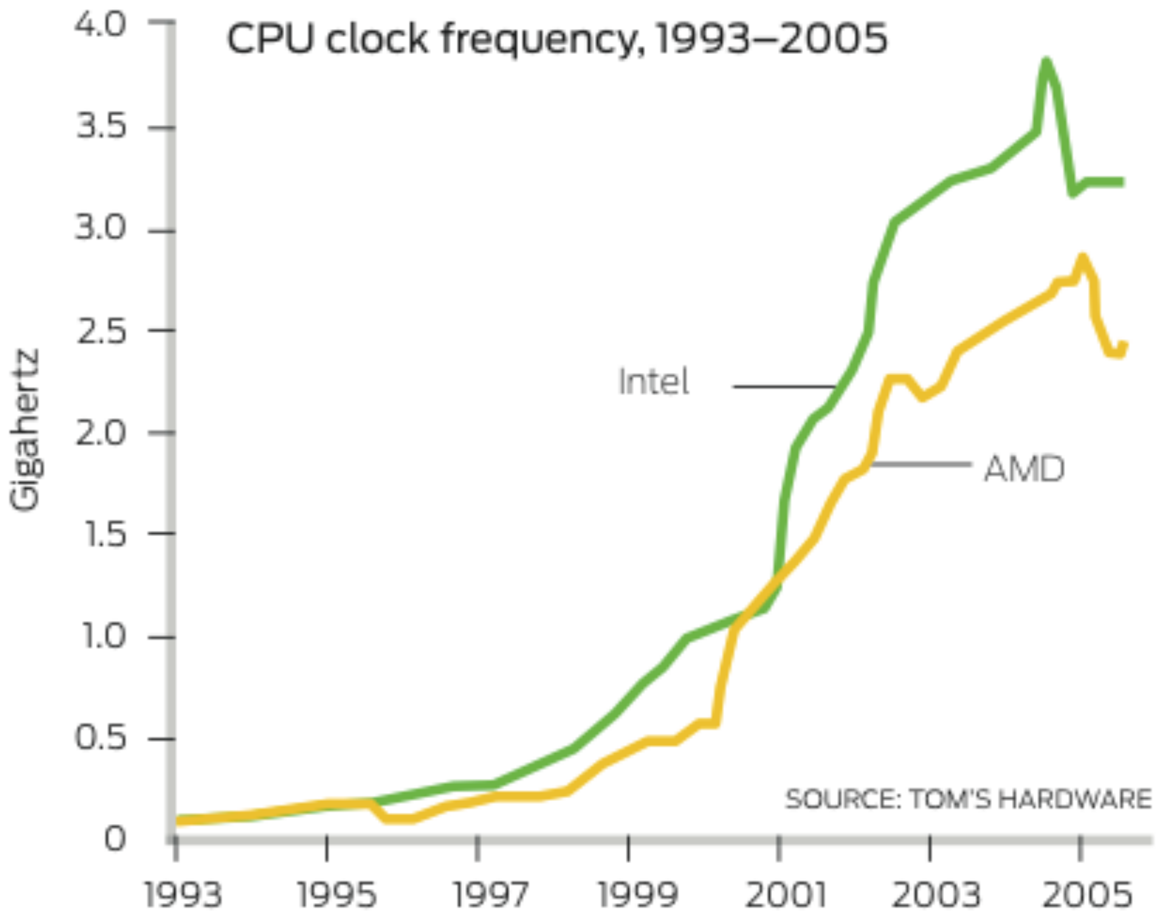
Underclocking a single core by **20%** saves **50% of the power** while sacrificing just **13%** of the performance.

Dividing the work between **two cores** running at an **80%** clock rate, we get **43% better** performance for the **same power**.

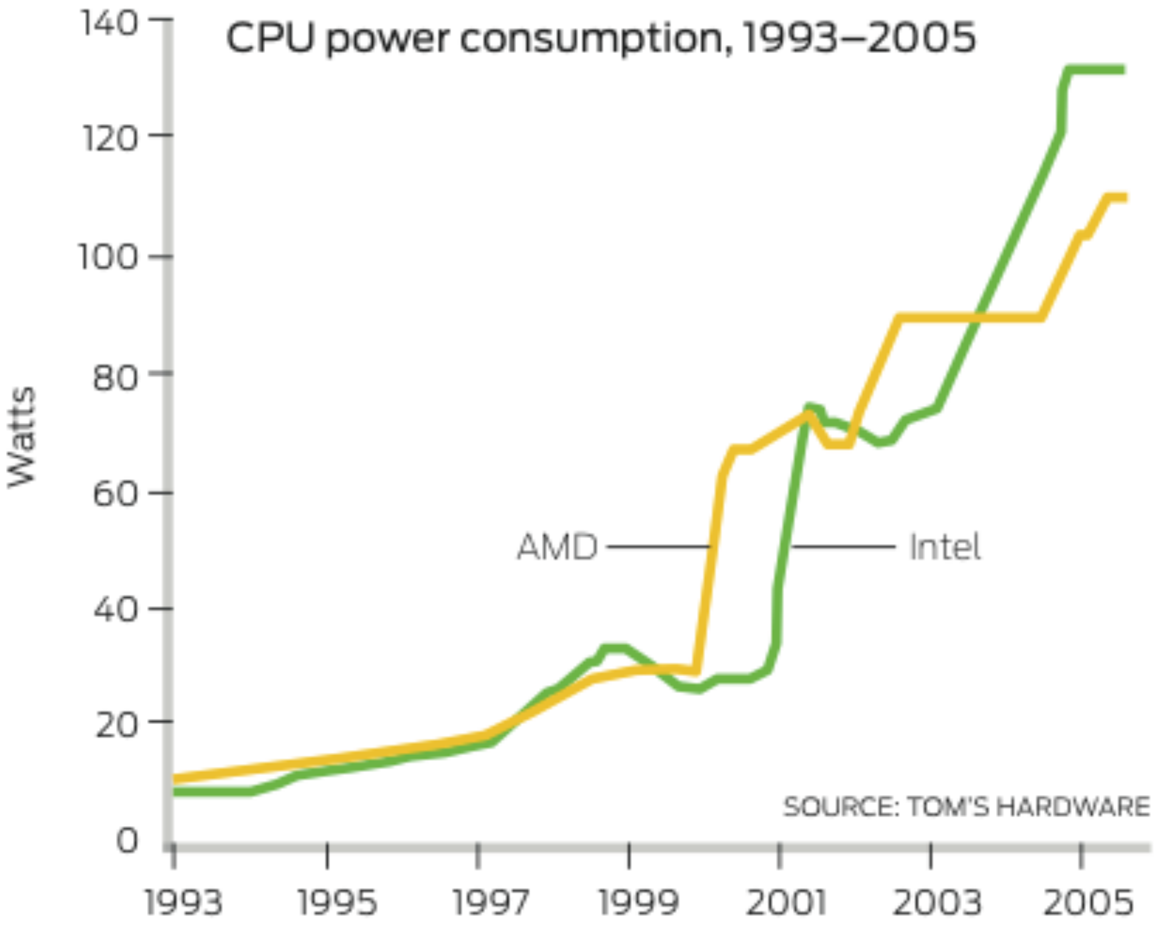
Source: "Why CPU Frequency Stalled" By Philip E. Ross, IEEE Spectrum April 2008

2004 was the turn over year!

CPU clock frequency

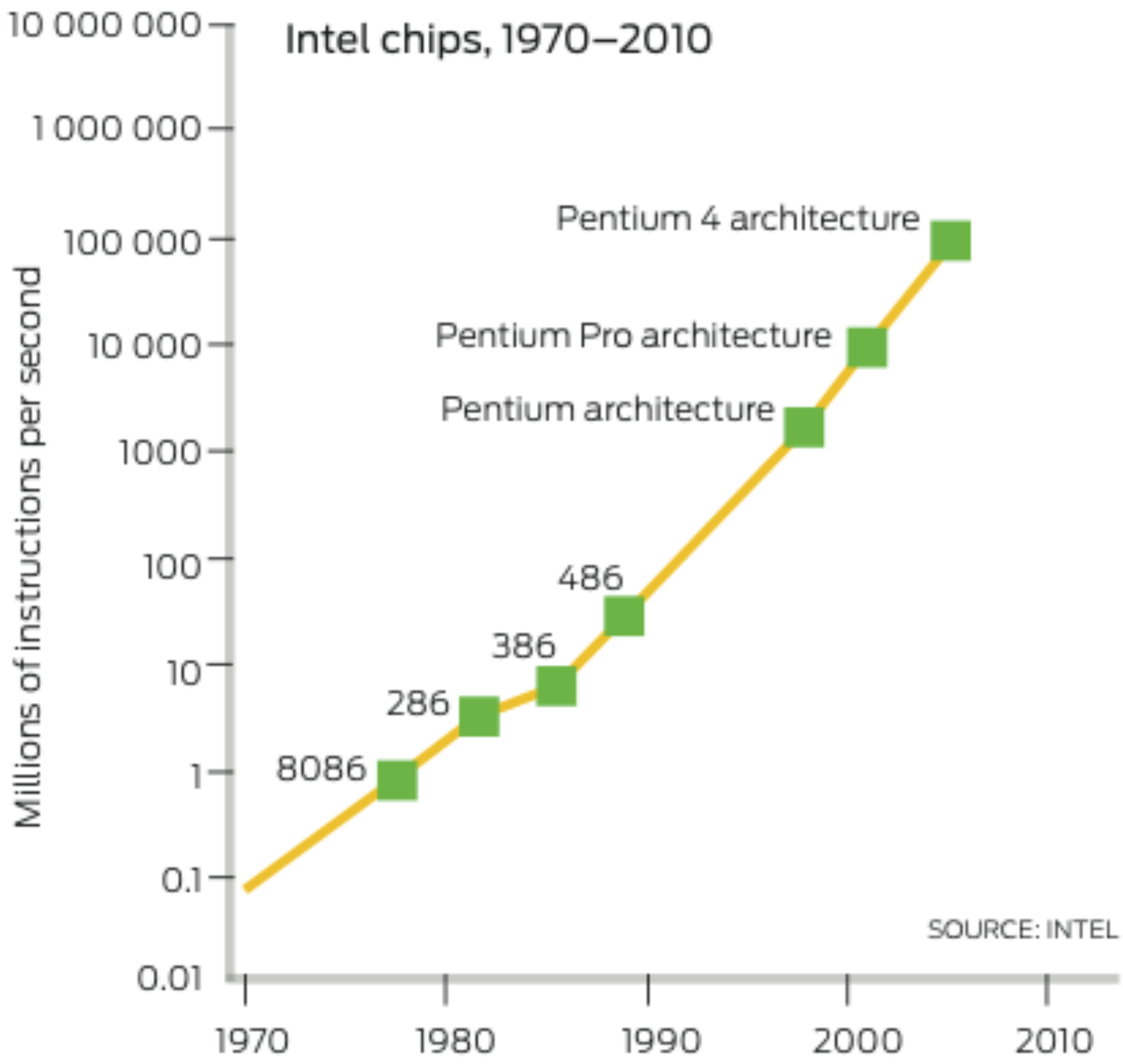


CPU power



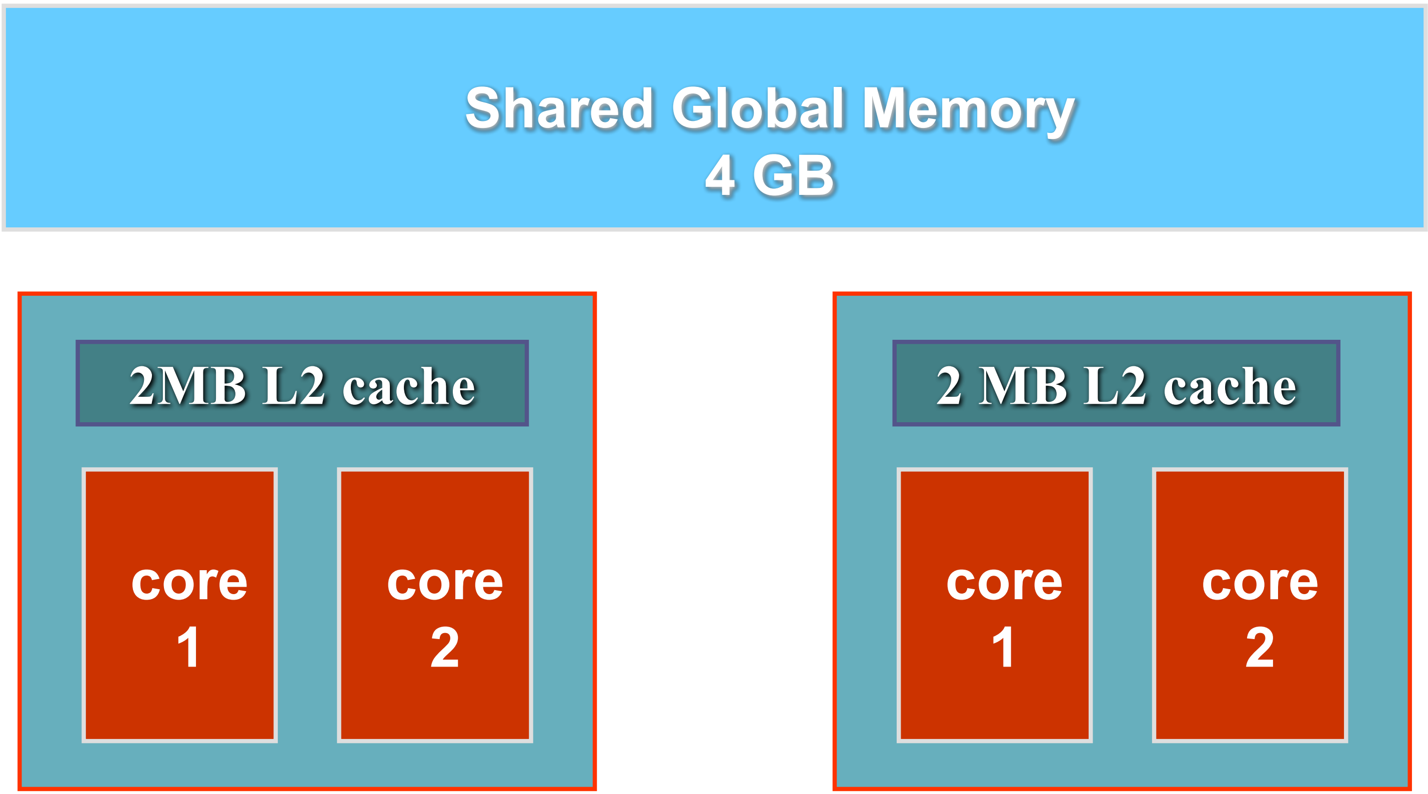
CPU power consumption, 1993–2005
Watts
140
120
100
80
60
40
20
0
AMD
Intel
SOURCE: TOM'S HARDWARE
1993
1995
1997
1999
2001
2003
2005

CPU MIPS



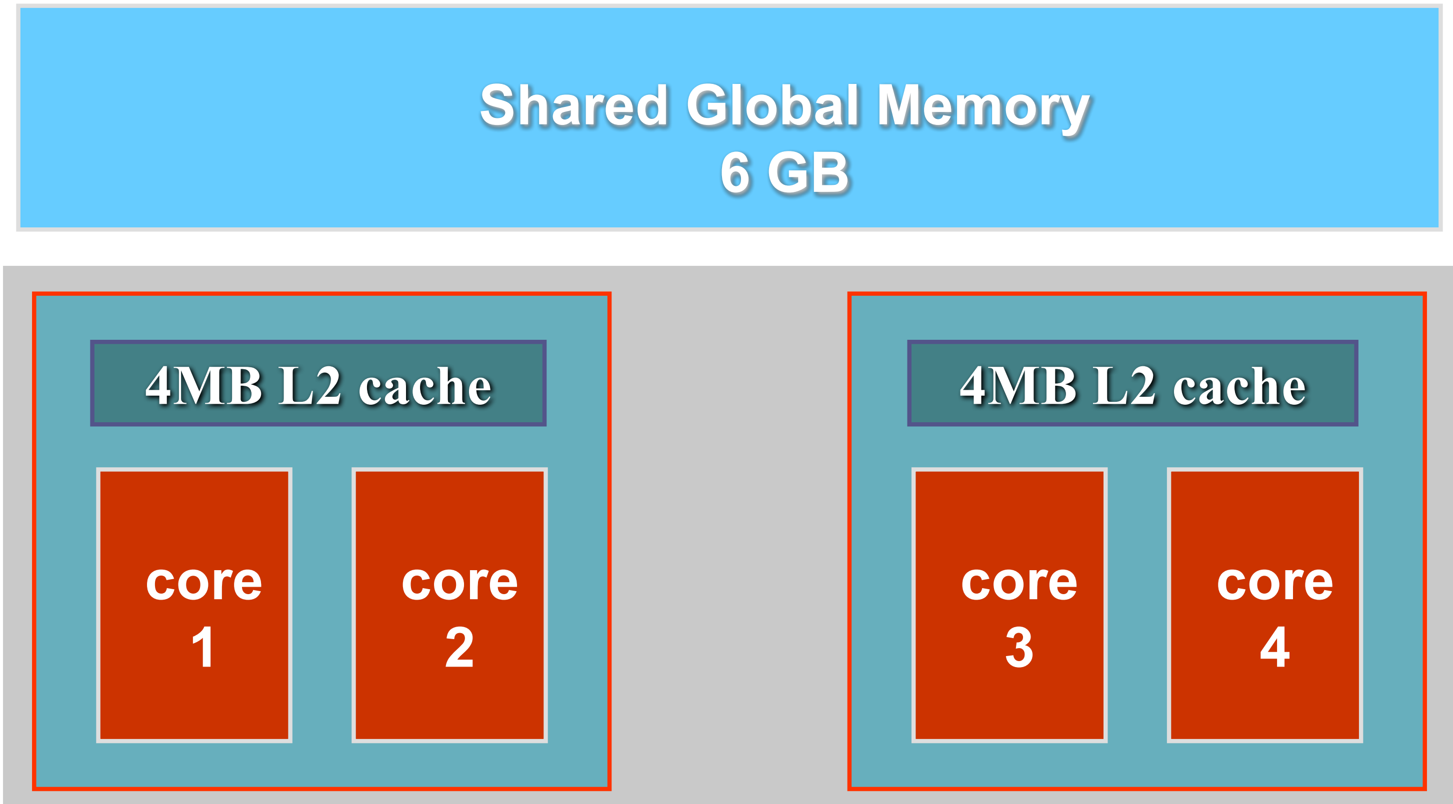
Intel chips, 1970–2010
Millions of instructions per second
10 000 000
1 000 000
100 000
10 000
1000
100
10
1
0.1
0.01
Pentium 4 architecture
Pentium Pro architecture
Pentium architecture
486
386
286
8086
SOURCE: INTEL
1970
1980
1990
2000
2010

Example of a IBM cluster node PPC 970 (2006)

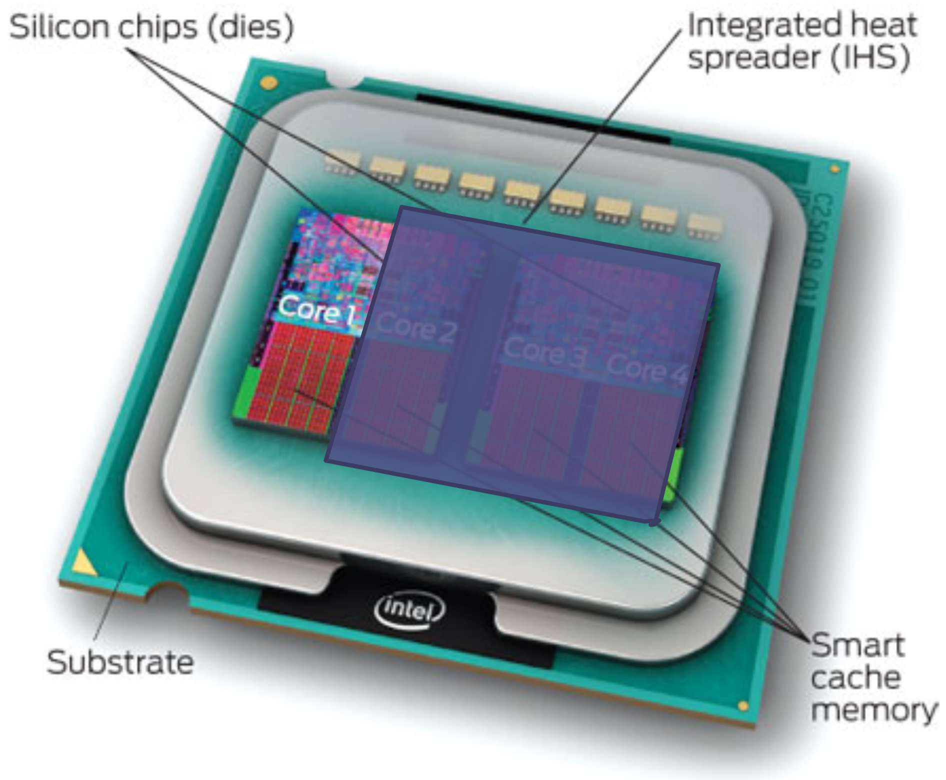


Intel Core 2 Quad Q6600 Processor (2008)

Available on desktop Computers!



Intel Core 2 Quad Q6600 Processor (2008)



- A sequential program only uses 25% of the capacity

Intel Core i7 Q3, 2013

Brand Name & Processor Number[1]	Base Clock Speed (GHz)	Turbo Frequency[2] (GHz)	Cores/ Threads	Cache	Memory Support	TDP	Socket (LGA)	Pricing (1k USD)
NEW Intel® Core™ i7 4960X Unlocked	3.6	Up to 4.0	6/12	15 MB	4 channels DDR3 1866	130W	2011	$990
NEW Intel® Core™ i7 4930K Unlocked	3.4	Up to 3.9	6/12	12 MB	4 channels DDR3 1866	130W	2011	$555
NEW Intel® Core™ i7 4820K Unlocked	3.7	Up to 3.9	4/8	10 MB	4 channels DDR3 1866	130W	2011	$310
Intel® Core™ i7-4770K Unlocked	3.5	Up to 3.9	4/8	8 MB	2 channels DDR3 1600	95W	1150	$317

Intel Xeon Phi (2013)

Intel® Xeon Phi™ coprocessor 5110P:
Ideal for high density environments

- Highly parallel applications using over 100 threads
- Memory bandwidth-bound applications
- Applications with extensive vector use

Buy the Intel® Xeon Phi™ coprocessor 5110P today >

xeon-phi-serverblade-feature-320x160.jpgKey specifications:

- 60 cores/1.053 GHz/240 threads
- 8 GB memory and 320 GB/s bandwidth
- Standard PCIe* x16 form factor, passively cooled
- Linux* operating system, IP addressable
- 512-bit single instruction, multiple data instructions
- Supported by the latest Intel® software development products
- Built using Intel's 22nm process technology—Intel's most energy efficient process yet—featuring the world's first 3-D tri-gate transistors.

60 Intel cores in a desktop

Manycore GPUs (attached processors)

- **GeForceGTX 280**
  - **240 scalar cores**
    - **Organized in blocks of 8 scalar cores**
      - **16K 32-bit registers (64KB)**
      - **usual ops: float, int, branch, …**
    - **Shared double precision unit**
    - **…**
- **TESLA**
  - **Up to 2880 scalar cores**

- **Manycore programming**
  - **CUDA -- NVIDIA only**
  - **OpenCL -- integration of CPU and GPU**
  - **OpenACC**

Mobile Computing

Quad-Core 1.4GHz

How to program multicore processors?

- Will compilers do the job?
  - Unfortunately they won't
  - Even for sequential programming we need to write code carefully if we want to get performance and scalable programs (data size and locality).

- Main challenge
  - To write **scalable** programs that:
    - Keep the efficiency level as Data increases
    - Keep the efficiency level as more cores are available

Parallel Computing technologies

Multicore programming:

OpenMP (Open Multi-Processing), **OpenCL**

Intel TBB (Parallel Studio)

Multi-computer programming (cluster):

MPI – message passing user interface

Multicore clusters / processors:

OpenMP + MPI

Manycore processors:

CUDA, OpenCL, OpenACC

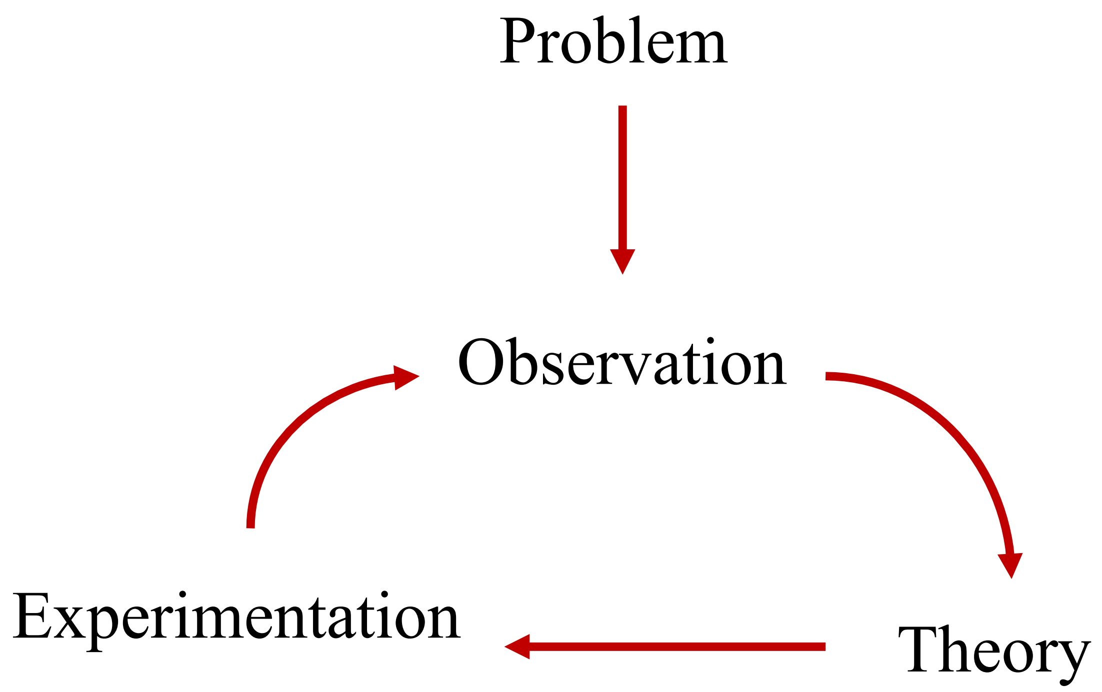
Main goal of Parallel Computing

- Scalable (resource-aware) computing

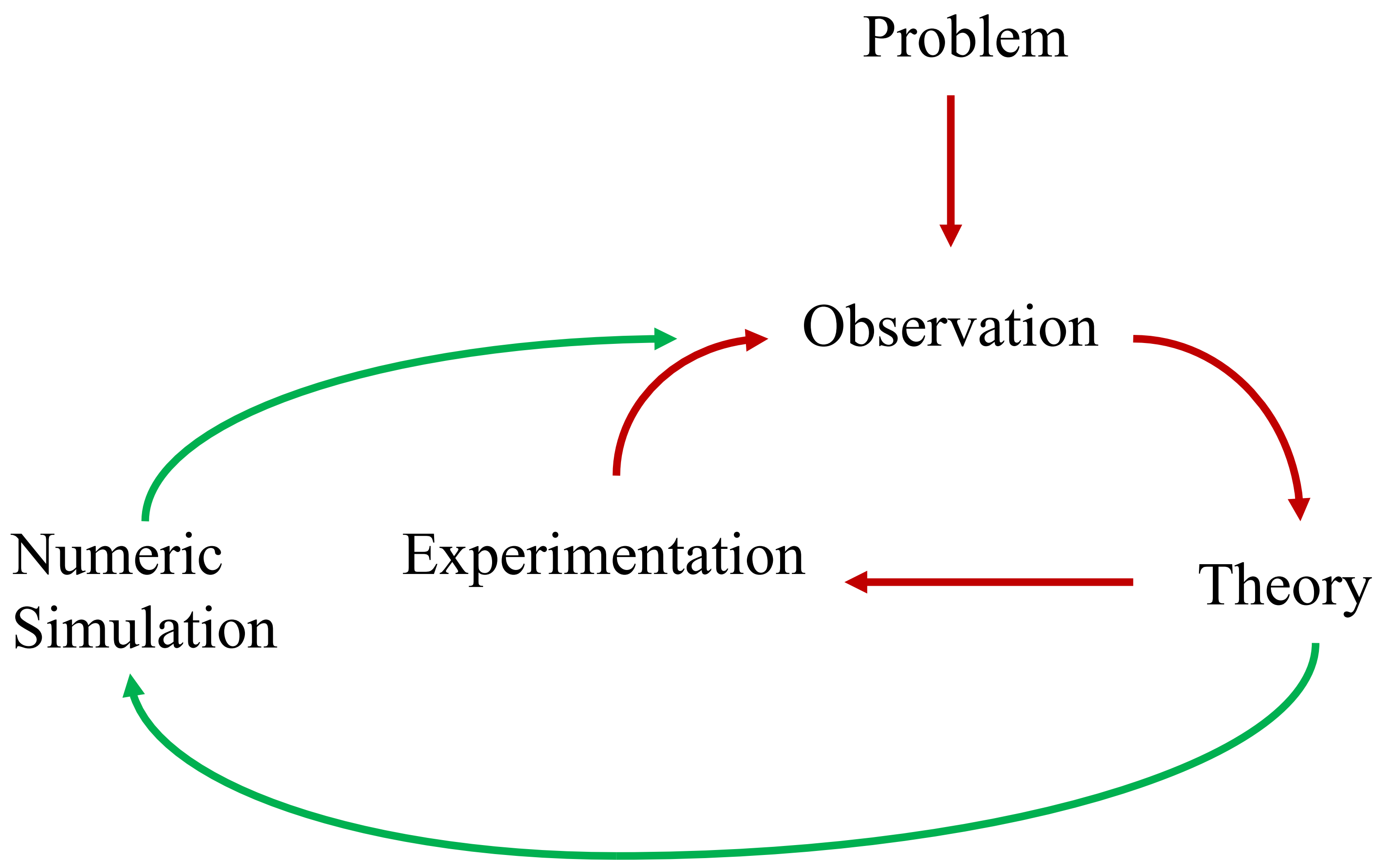
- Resources in computing:
  - sets of (processor + memory + interconnection)
  - understand the trend past-present-future
  - be prepared for heterogeneity: general-purpose & attached devices

- Performance evaluation
  - **Performance** and **Efficiency** measures
  - **Scalability** analysis

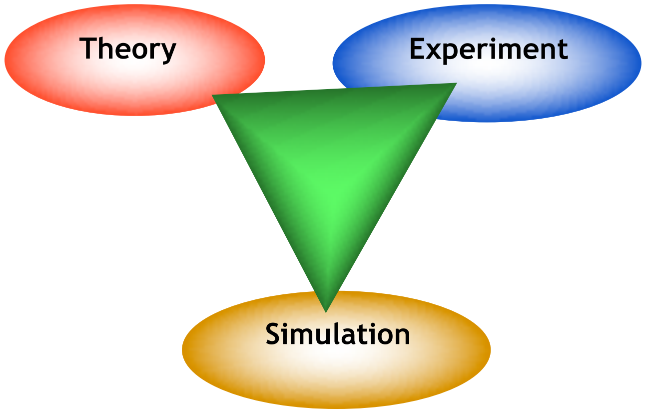
Scientific method: Classic approach



Modern Scientific method



Scientific Computing

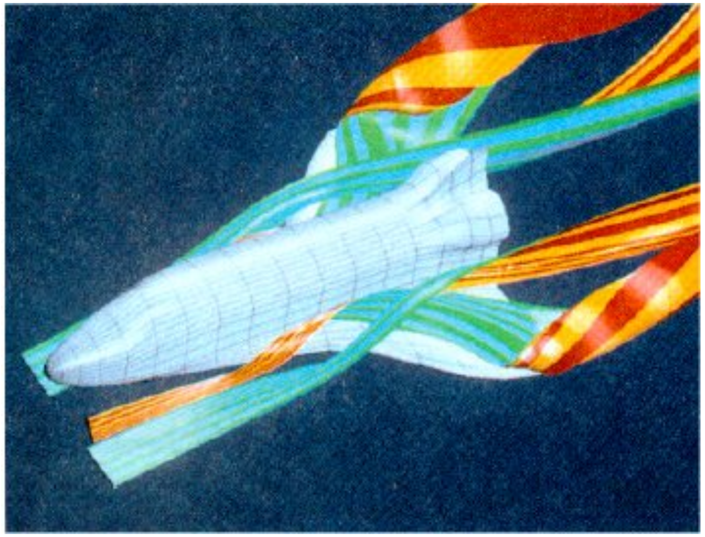


Simulation: The Third Pillar of Science

Limitations:

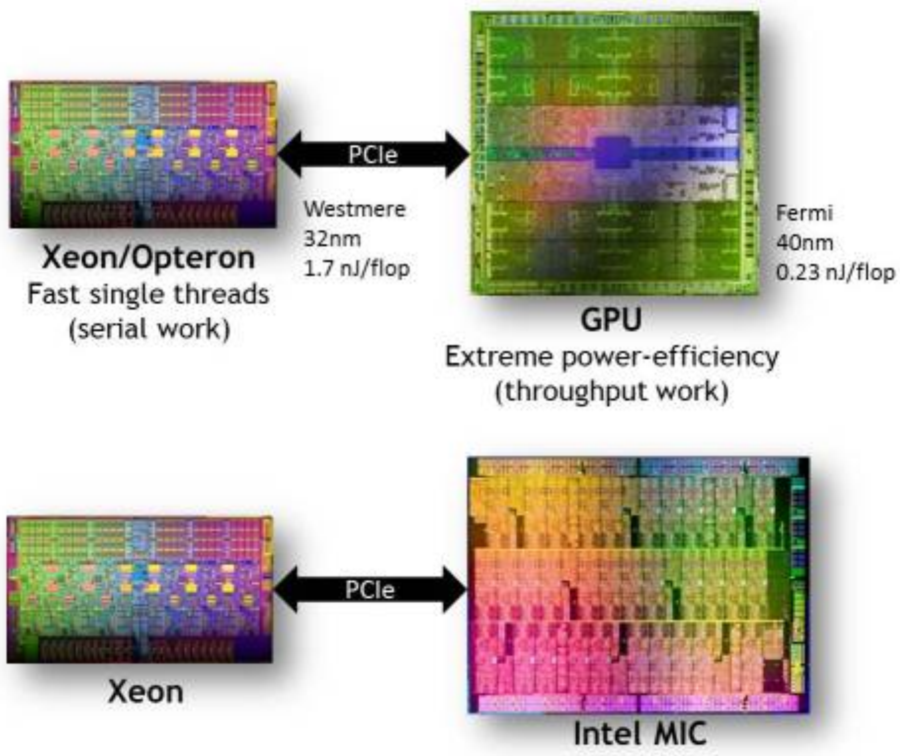
- **–To difficult—build large wind tunnels**
- **–To expensive—car crash tests**
- **–To slow—wait for climate or galactic evolution**
- **–To dangerous—weapons, drug design, climate experimentation**

Audi A8 car-crash model contains numerous materials and structural components modeled by 290,000 finite elements (shown here as squares on a grid). The model predicts the extent of deformation in the car after a crash.



Heterogeneous Computing

- Evolution of computing systems:
  highly parallel & heterogeneous !
  - new computing units: gpGPU/MIC/…



HPC systems in Top500: #1,2,6,10 with Intel Xeon MIC & NVidia GPU

…

Tianhe-2: 3,120,000 cores 16,000 nodes

…

NVidia K20x: 2,880 arith cores

Top 500

Tianhe-2 (MilkyWay-2) : National University of Defense Technology
No.1 from Jun 2013 until Nov 2014

Titan: Oak Ridge National Laboratory
No.1 in Nov 2012

Sequoia: Lawrence Livermore National Laboratory
No.1 in Jun 2012

K Computer: RIKEN Advanced Institute for Computational Science
No.1 from Jun 2011 until Nov 2011

Tianhe-1A: National Supercomputing Center in Tianjin
No.1 in Nov 2010

Jaguar: Oak ridge National Laboratory
No.1 from Nov 2009 until Jun 2010

Parallel Computing

- Why shall we use parallel computing?
  - Possibility of solving bigger problems and with more realistic representation (higher accuracy/detail)
    - Example: weather forecast for more days and with more accuracy
  - To reduce development costs
  - To have higher freedom to “explore” alternatives.
  - To explore modern multi-core processors and GPUs.

Performance

- Performance metrics
  - MIPS
    - ***m**illion **i**nstructions **p**er **s**econd*
    - For integer operations
      - Also called "***M**eaningless **I**ndicator of **P**erformance*"
  - FLOPS
    - ***fl**oating-point **op**erations per **s**econd*
    - For scientific applications

- Peak performance (R_{peak} *Top500*)
  - Related to the CPU *speed*

- Maximum performance (R_{max} *Top500*)
  - Maximum performance for a given algorithm (Linpack for *Top500* list)

- N_{max} - Problem size to achieve R_{max}

Performance

- Sustained performance
  - *Computer performance* depends on several factors: I/O speed, data access pattern, memory hierarchy.
  - The relevant performance is the one that results from the real execution of an algorithm
  - The sustained performance depends also on the algorithm design
    - An implementation compatible with the computer architecture can achieve the same performance (sustained) for a wider range of input data
  - Example: matrix multiplication algorithm

Parallelism and Amdahl law

- In an application there is always a part that cannot be parallelized.
- Amdahl Law
  - Let ***s*** be the piece of work that is sequential ***(1-s)*** will be the piece of work that can be parallelized.
  - ***P*** – number of processors

- Even if the parallel part is perfectly scalable, the performance (Speedup) is limited by the sequential part.

Amdahl Law

The gain obtained with the parallel program is defined as *Speedup*:

$$Speedup = \frac{T_1}{T_P}$$

The Amdahl Law imposes a limit for the *Speedup* that can be obtained with ***P*** processors.

$$T_P = \frac{(1-s)}{P} + s$$

$$Speedup = \frac{1}{\frac{1-s}{P} + s}$$

Example: if the total execution time of an algorithm is 93s and the sequential time susceptible of parallelization is 90s, then:

(1-s) = 90/93=0.968 → 96.8% of the code can be parallelized

s = 1-0.968 = 0.032 → 3.2% of the code is inherently sequential

Amdahl Law

Code susceptible of parallelization:

Is the part of the code that executes with Speedup=P if it runs on P processors.

Code inherently sequential:

Is the part of the code that cannot be parallelized, such as data input/output, variable initialization, etc.

If $P \rightarrow \infty$ the Speedup $\rightarrow 1/s$.

For the last example the maximum speedup will be:

$Speedup_{Max} = 1/0.032 = 31.25$

In conclusion: whatever the most number of processors used the processing time will not be less then 1/31.25

Example 1

- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

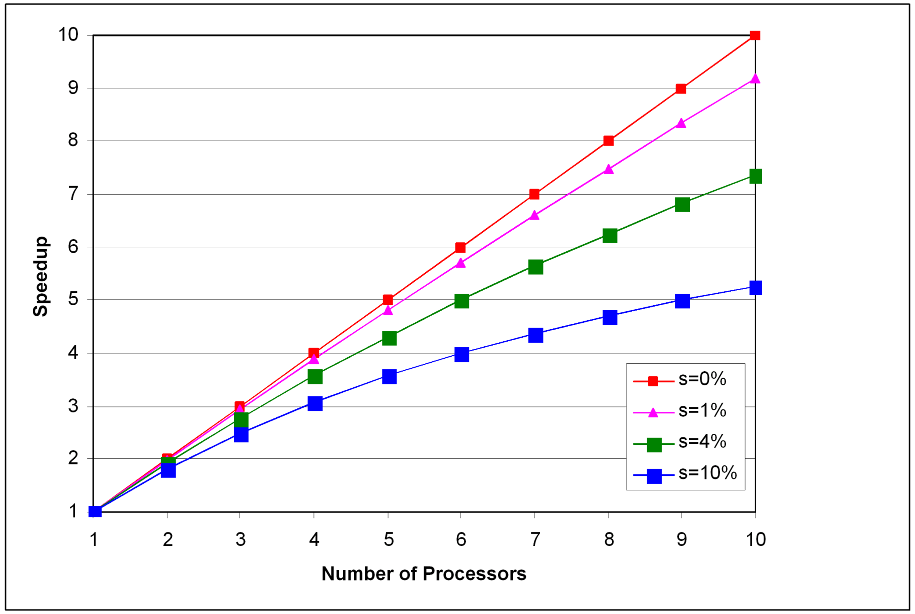
$$Speedup \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

Example 2

- 20% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

$$\lim_{p\to\infty} \frac{1}{0.2+(1-0.2)/p} = \frac{1}{0.2} = 5$$

Amdahl Law

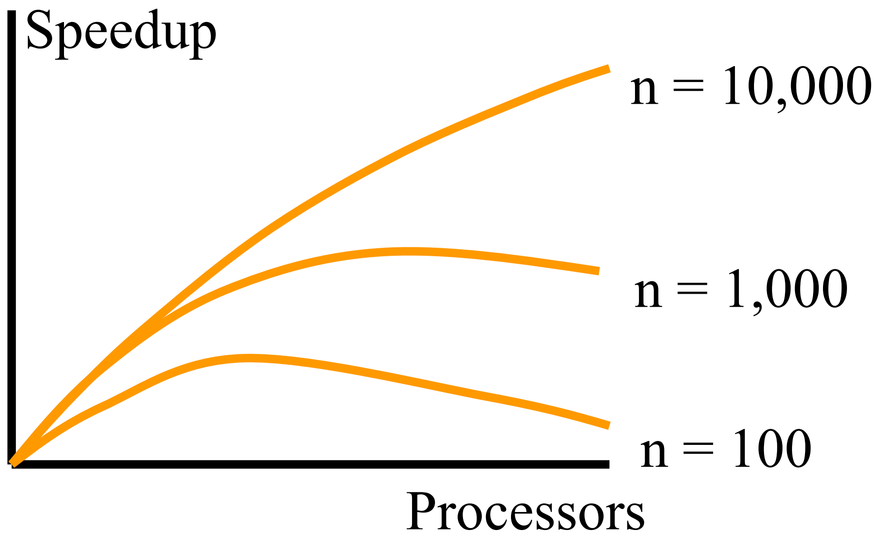


Theorectical Speedup according to Amdahl Law

Several important considerations are taken from Amdahl Law:

1. It allows to have a realistic expectation, for a given algorithm, about what we can obtain with the parallel approach.
2. It shows that to achieve higher Speedups it is necessary to reduce or eliminate the algorithm sequential blocks.
3. It also gives a comparison metric to measure parallelizability of several algorithm for the same problem.

Amdahl Law



Observed Speedup

In fact the observed speedup when *P* increases is exemplified in the figure.

This behavior is due to the fact that the inherently sequential part ***s*** increases as ***P*** increases.

The increase of the number of processors leads to an increase of communication times, conflicts to access resources (memory, network), CPU cycles spent to support parallelism and process synchronization.

The *Speedup* function increases until a given number of processors *P,* and decreases after that. The number of processor that ensures the minimum processing time will be less then the obtained by Amdahl law.

Ways of extracting parallelism

- Functional Parallelism
- Data Parallelism
- Streaming

Functional Parallelism

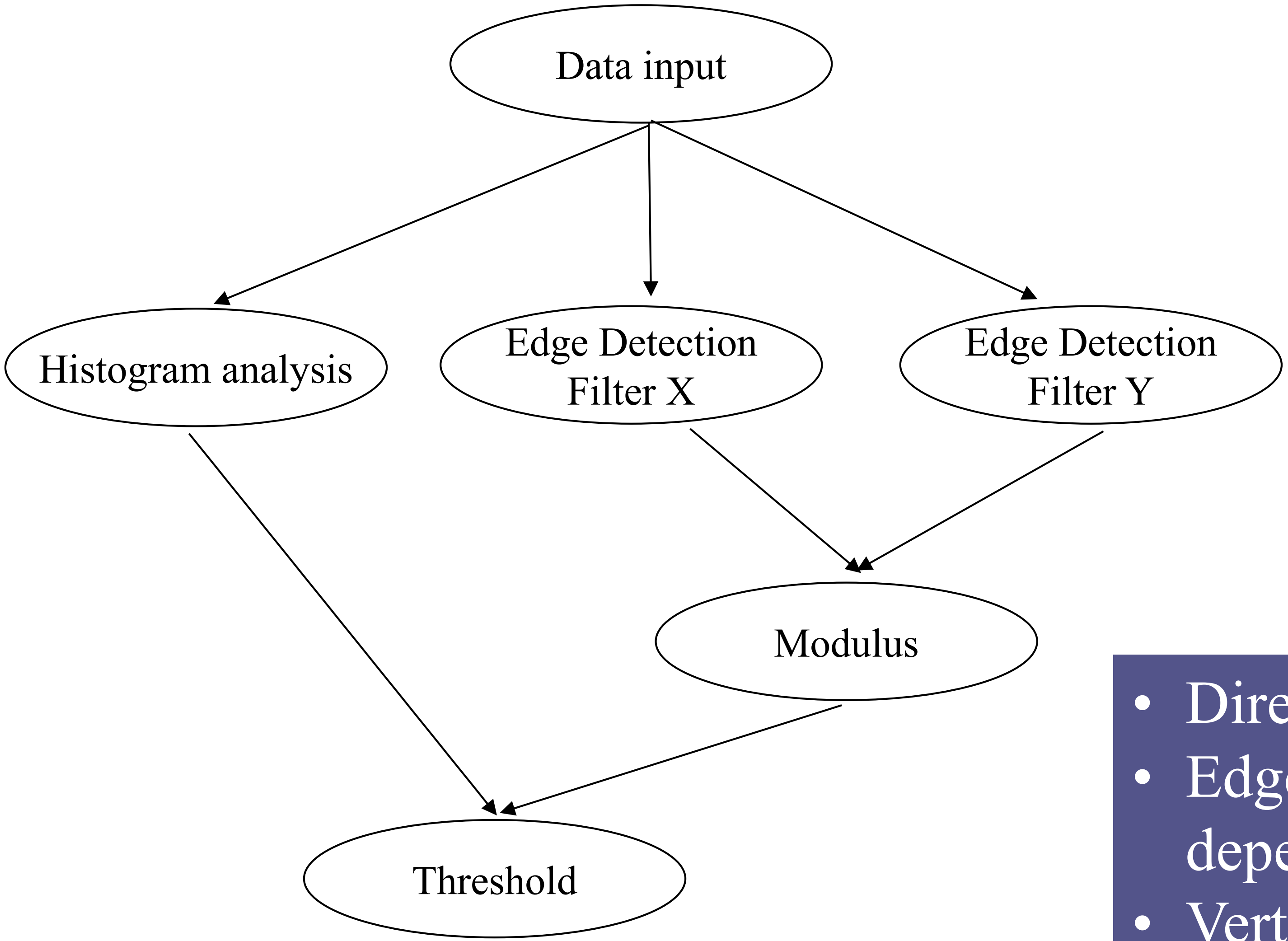
- Independent tasks execute different operations on different data sets

Example:

1. $a = 2$
2. $b = 3$
3. $m = (a + b) / 2$
4. $s = (a^2 + b^2) / 2$
5. $v = s - m^2$

- Instruction 1 and 2 are independent
- Instructions 3 and 4 are dependent from 1 and 2 but are independent from each other.

Functional Parallelism: data dependency graph



- Direct acyclic graph
- Edges: Functional dependencies
- Vertices: tasks

Example

- Sum the elements of a vector x

Data Parallelism

- Independent tasks execute the same operation over different data.

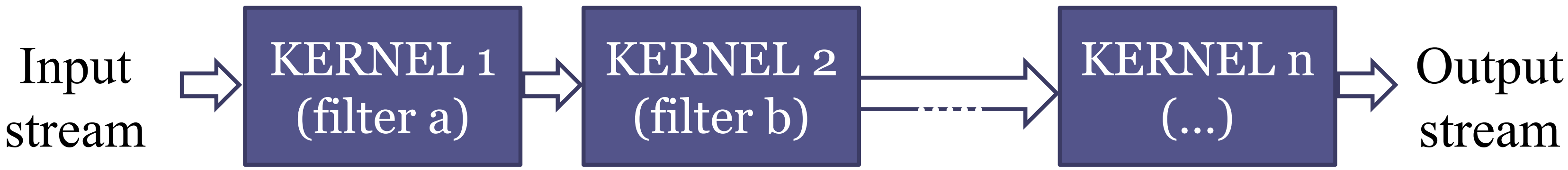
Example:

```
For (i = 0; i< 99; i++)
   a[i] = b[i] + c[i]
```

The vectors elements can be added in a independent way. The sum operation can be applied simultaneously over the different vector elements ***b*** and ***c***.

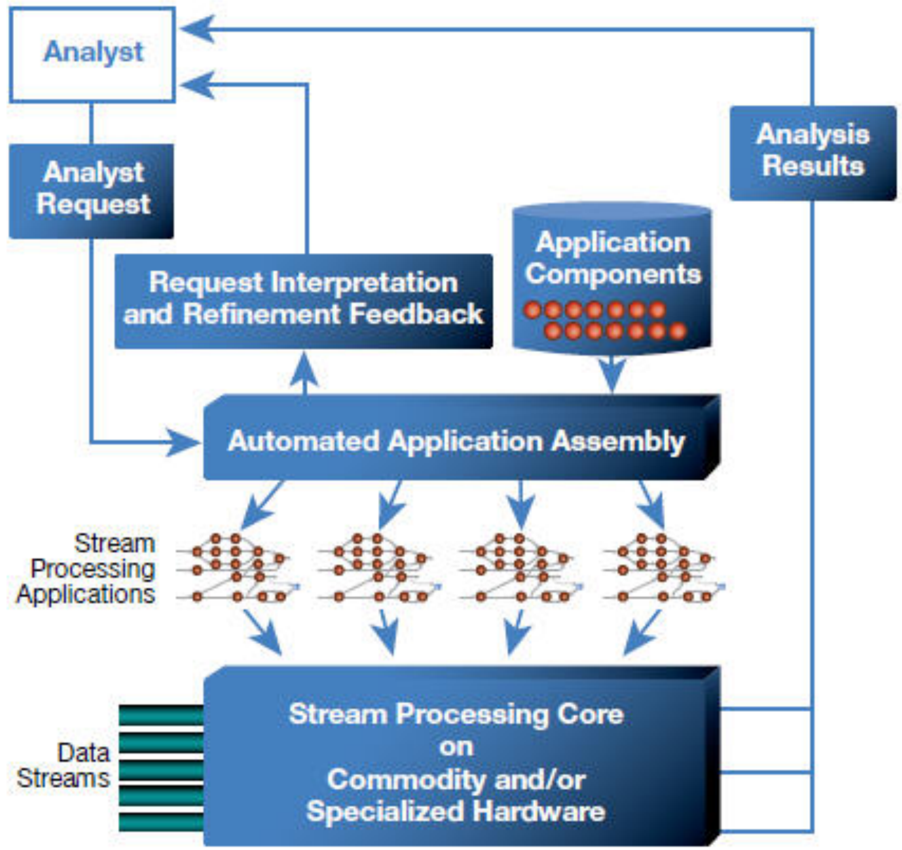
Streaming (1)

- To process streams of data
  - Divide the process in steps
  - The number of steps limits the Speedup.



Streaming (2)

- To process multiple streams of data
  - Examples: real time data analysis; real time decision making support.

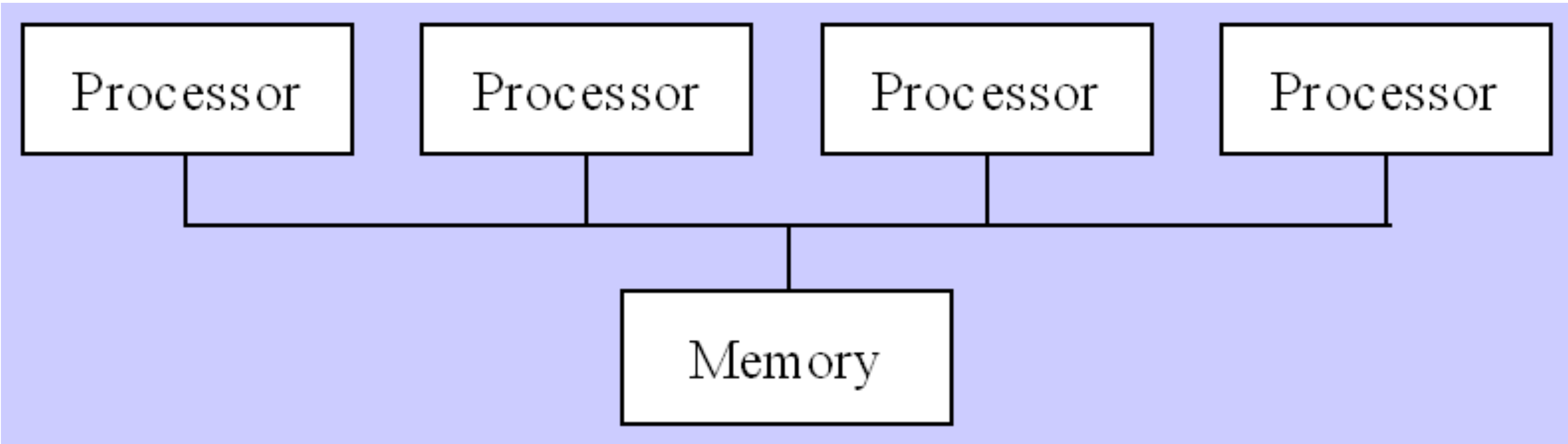


The diagram shows the business user (top left corner), and how the user's analysis request is converted into a stream processing application, deployed into the compute environment as a distributed stream processing job. It also shows how the analysis results are returned, rendered as a dynamic mashup and presented to the business user. (Credit: IBM)

Parallel Programming models

- Shared Memory Model
- Distributed Memory Model

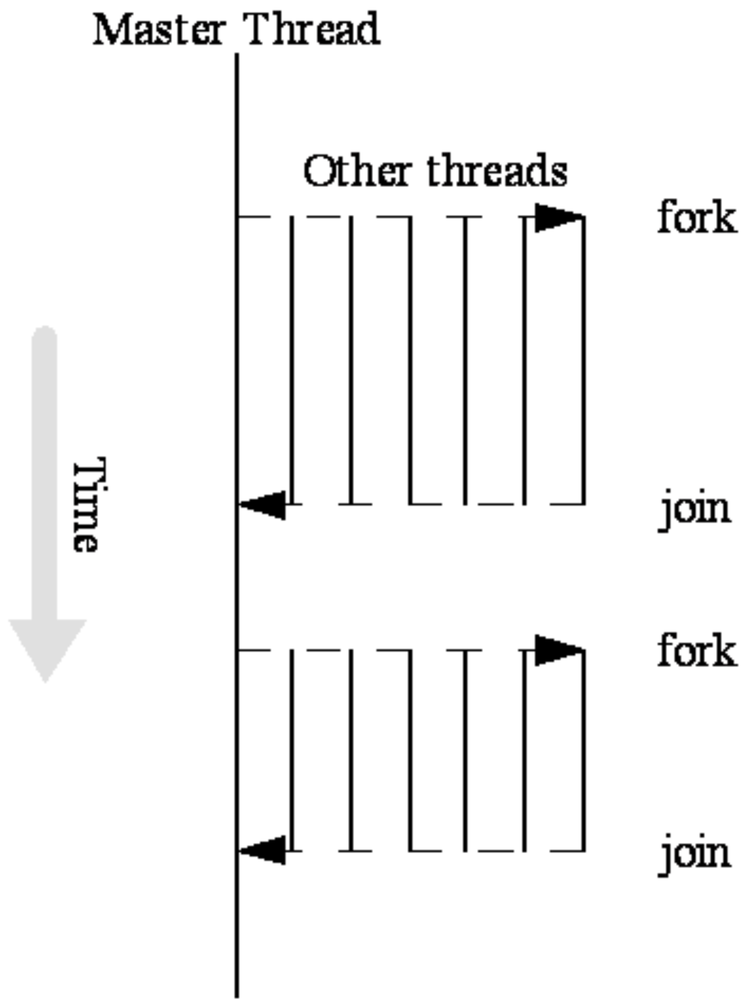
Shared memory model



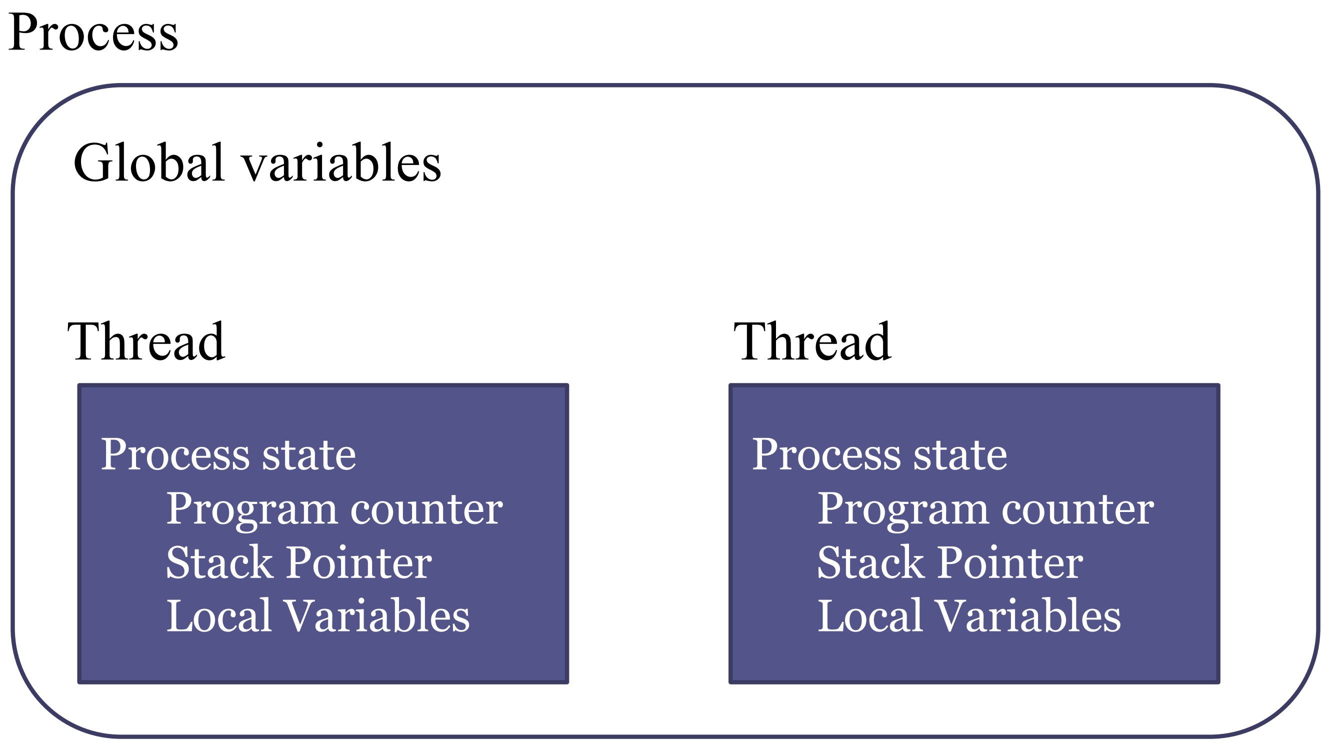
- Each processor (or core) executes a thread
- Threads interact by shared variables

Shared memory model

- Fork/Join parallelism
  - Number of fork/joins influences performance



Shared memory model



- Threads
  - Each thread has its own process state, but share global variables defined by the master thread

Shared memory model

- Parallel for Loops
  - C programs often express data-parallel operations as `for loops`

```
for (i = first; i < size; i += prime)
      marked[i] = 1;
```

  - A multithreaded program can split the `for loop` to execute concurrently

Shared memory model

- With OpenMP
  - Format:

```
#pragma omp parallel for num_threads(k)
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

  - Implicitly **k** threads are created
    - **Each thread computes N/k elements**

Shared memory model

- With POSIX threads

```
int main(){
        ...
    for (i = 0; i < k; i++)
       thread_create(mythread, i);

    for (i = 0; i < k; i++)
       thread_join();
}
void mythread(int id){
    int it_per_thread = N/k;
    int first = id * it_per_thread;

    for (i=start; i<start+it_per_thread;i++)
      a[i] = b[i] + c[i];
}
```

Example

- Consider the program to compute π using the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Performance
n = 10^8

Example 1st solution

- If we simply parallelize the loop...

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition

- … we set up a race condition in which one process may "race ahead" of another and not see its change to shared variable **`area`**

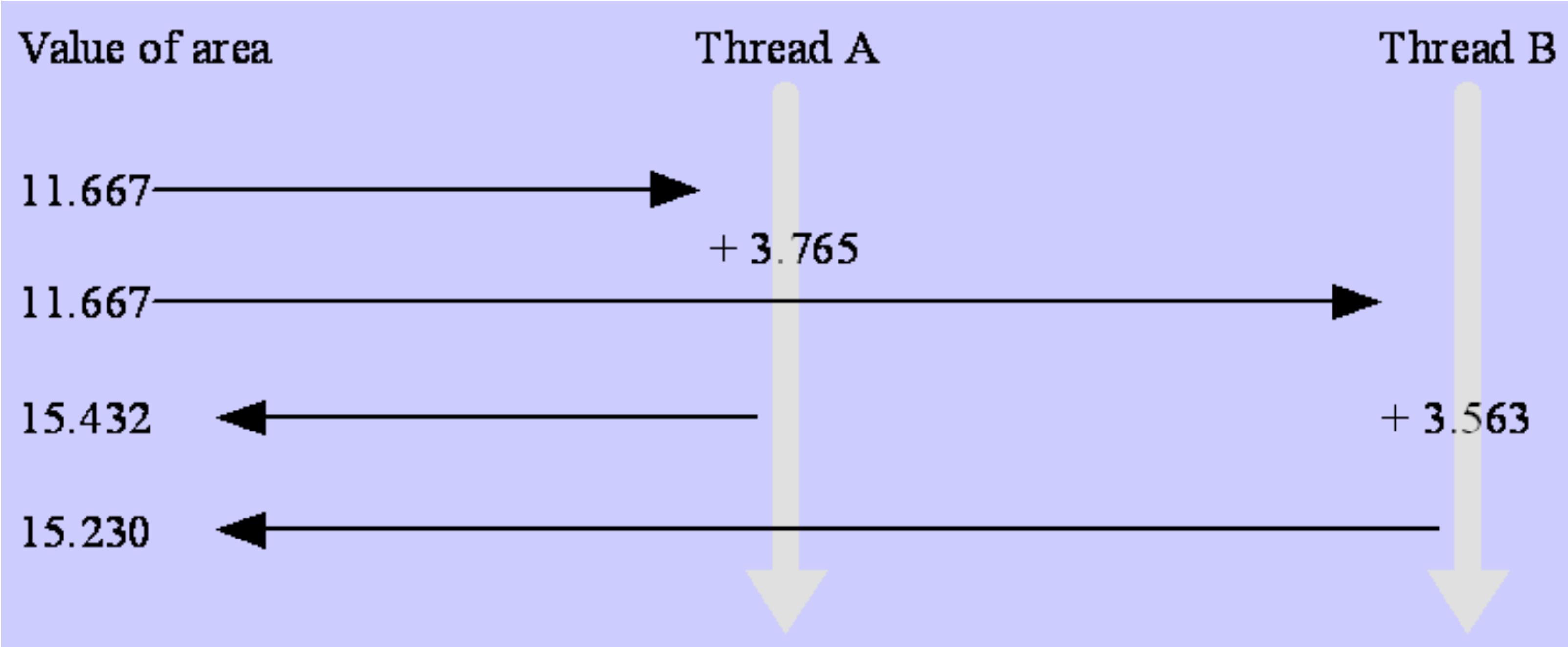
`area` 15.230 **Answer should be 18.995**

Thread A 15.432

Thread B 15.230

```
area += 4.0/(1.0 + x*x)
```

Race Condition Time Line



- A date race occurs when two or more threads can modify the same memory location at the same time

Critical section

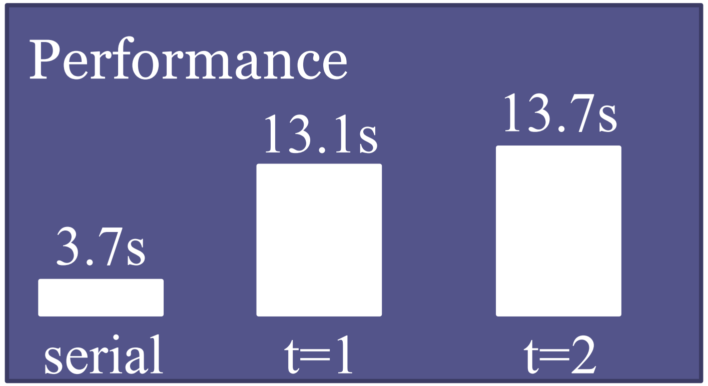
- Critical section: a portion of code that only a thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code

Example 2nd solution

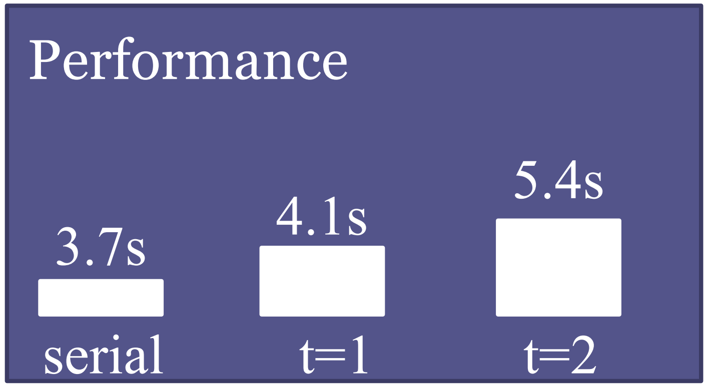
```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```



Why not to put AREA as private?

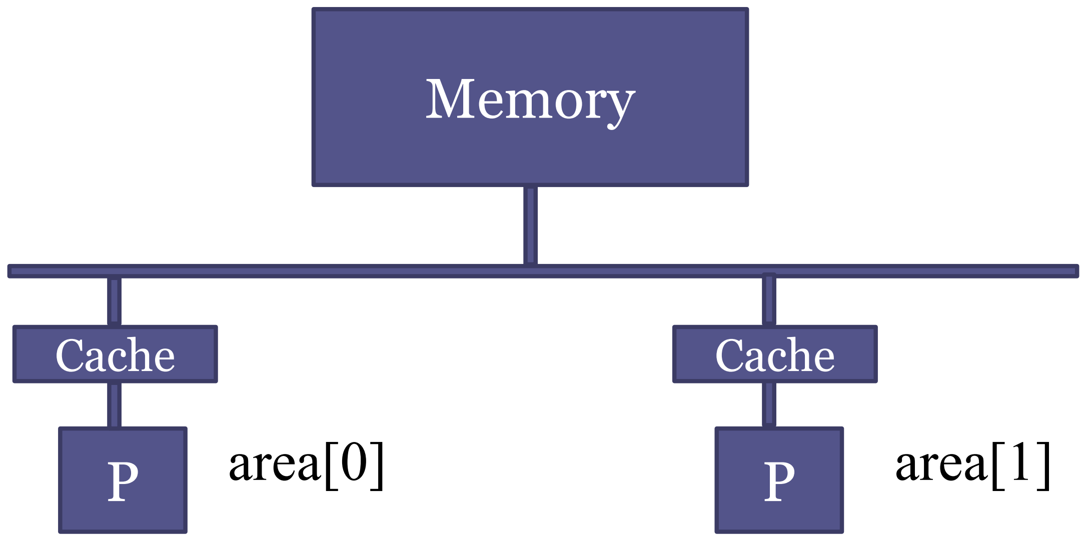
Example 3rd solution

```
double area[2], pi, x;
int i, n;
...
for (i=0; i<2; i++) area[i]=0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area[omp_get_thread_num()]+= 4.0/(1.0 + x*x);
}
pi = 0;
for (i=0; i<2; i++)
      pi += area[i];
pi /= n;
```



False sharing

- False Sharing: occurs when 2 or more threads access different data on the same cache line (read/write).
- Example: Access close positions of a global vector

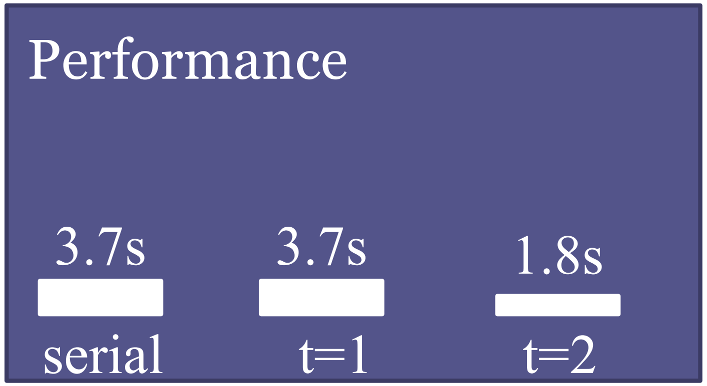


- The effort required to maintain consistency degrades performance

Example 4th solution

- Reduction Clause

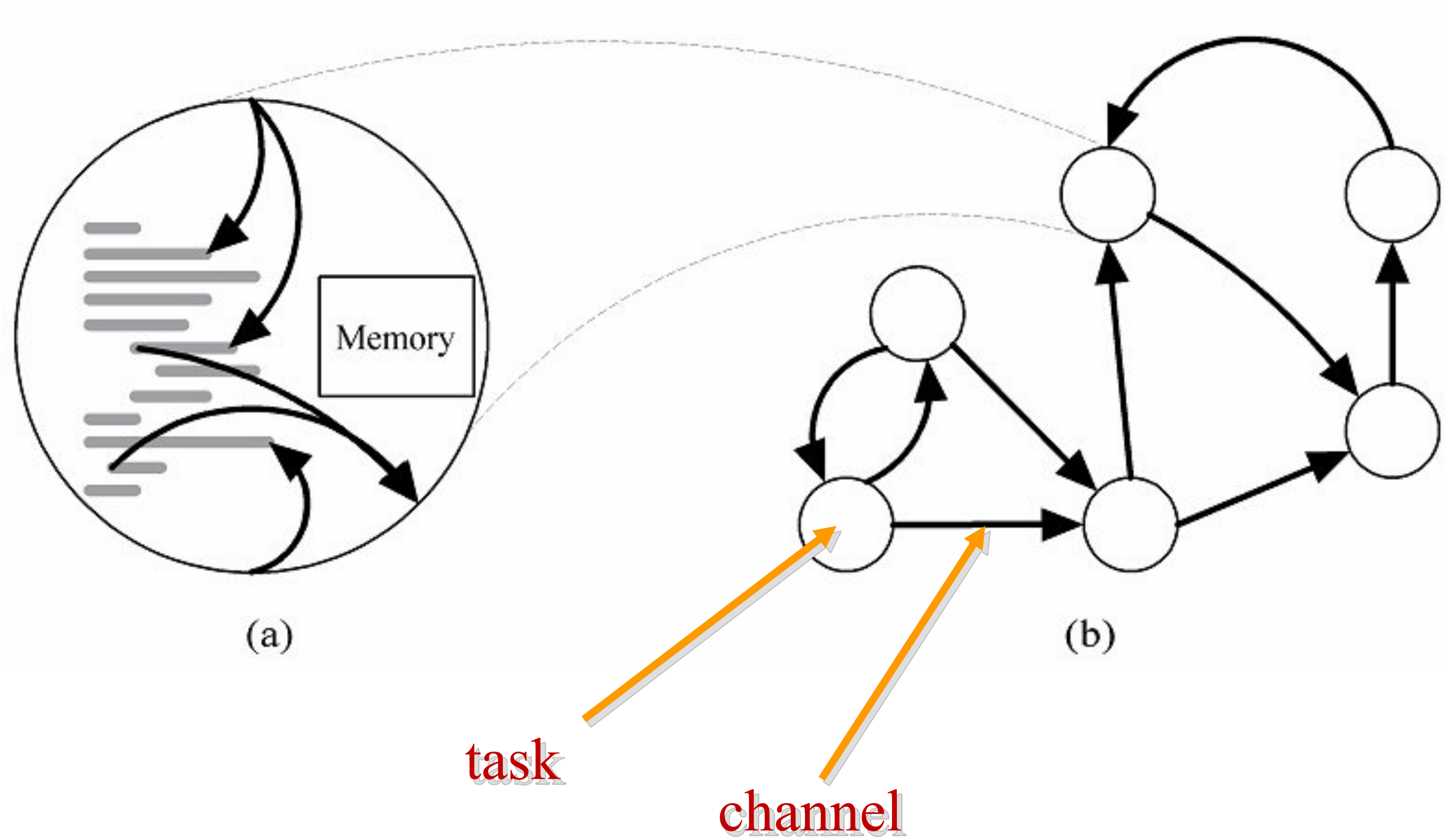
```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
        private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x)
}
pi = area / n;
```



Distributed Memory Model

Task/channel model ⇔ Developed for a Distributed Memory Computer

Abstraction to develop parallel algorithms.



(a) (b)

Distributed Memory Model

Parallel Program = a set of tasks executing concurrently.

- Task
  - Sequential Program (von Neumann model)
  - Local memory
  - A set of I/O ports
- Tasks interact by sending messages through the communication channels.

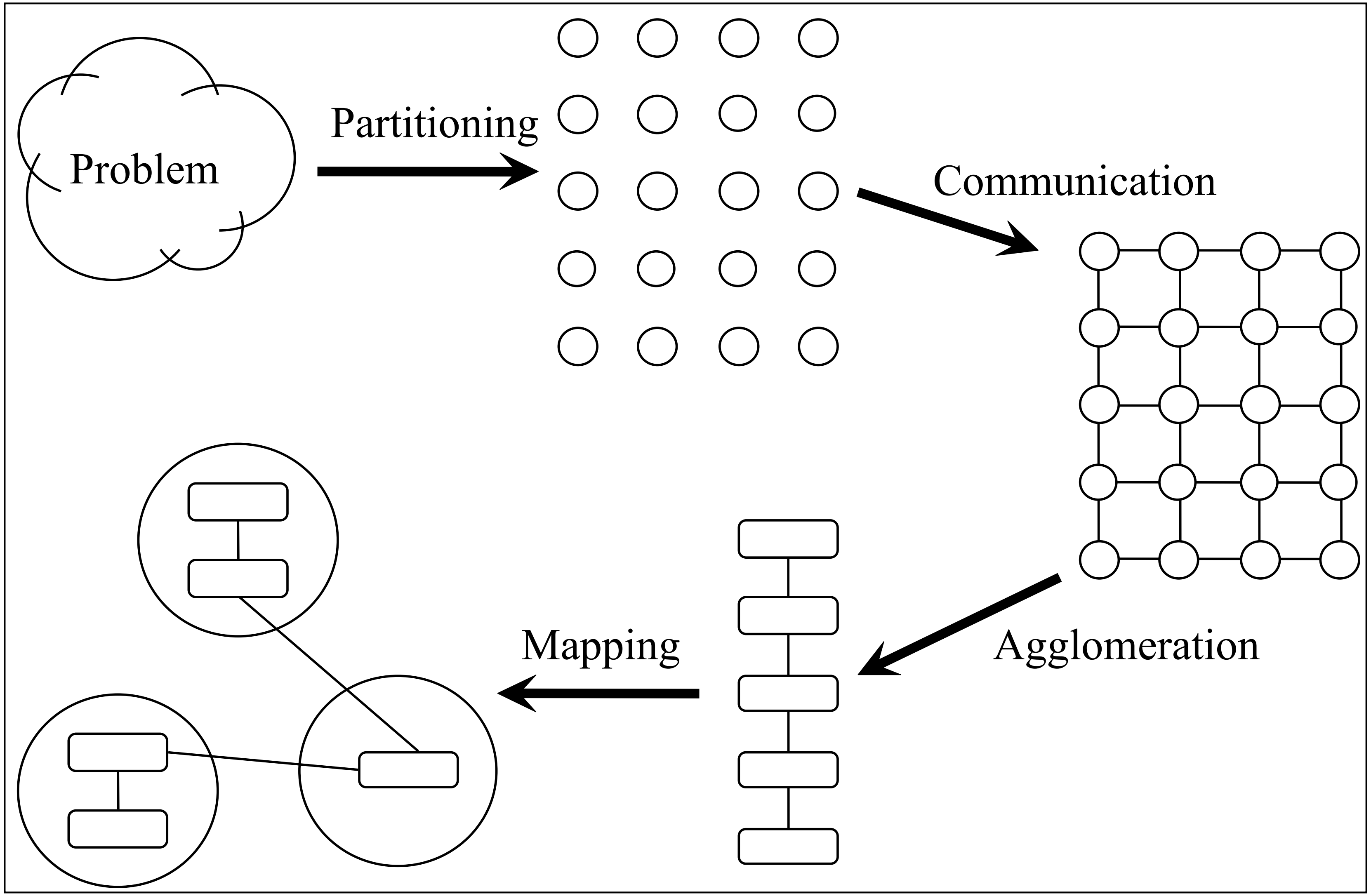
Distributed Memory Model

Methodology to develop parallel programs:

- Problem partitioning
- Communication Patterns
- Agglomeration
- Mapping

This methodology addresses first the problem characteristics, such as data dependencies, and postpones the analysis related with the parallel machine.

Parallel Programming



Lab work

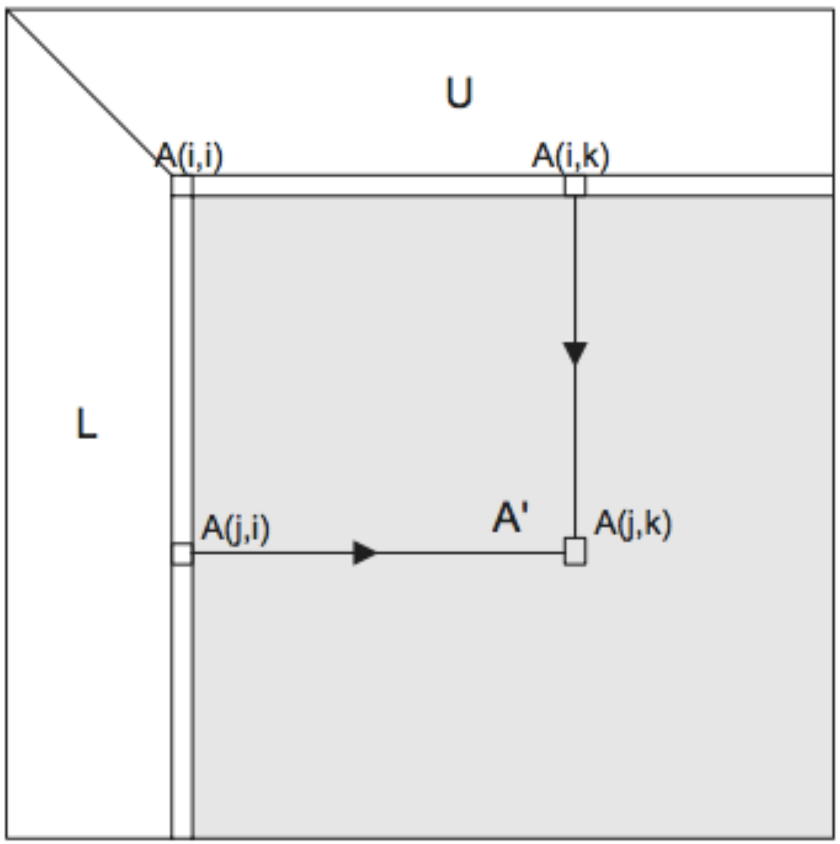
- Download the pi_openmp.zip file
- Compare sequential and parallel execution
- Register the maximum precision obtained
- Propose and implement a solution able to improve precision.

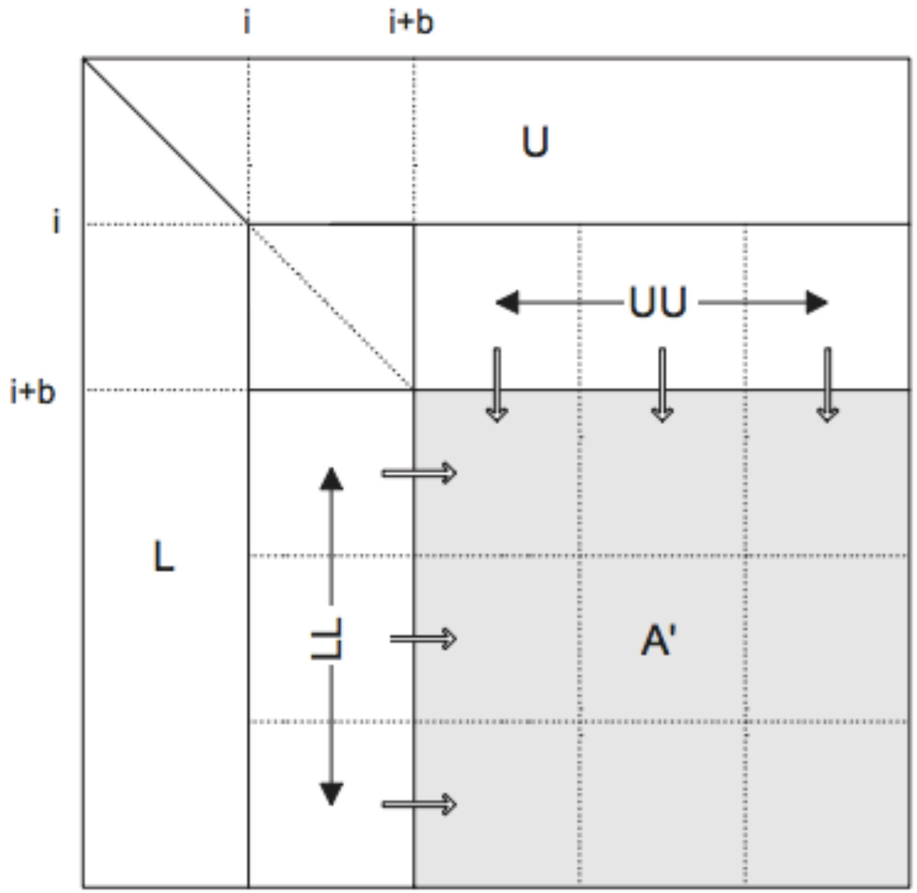
Classification of the operations

- Sequential operations
  - Operations that require some effort to be parallelized. The computation of the current element uses a previously computed element.

- Parallel operations
  - Operations that are embarrassingly parallel

LU Decomposition – sequential operation





$$A' = A(i+1:n-1, i+1:n-1) = A(i+1:n-1, i+1:n-1)$$
$$-A(i+1:n-1, i) \times A(i, i+1:n-1)$$

Matrix multiplication – parallel operation

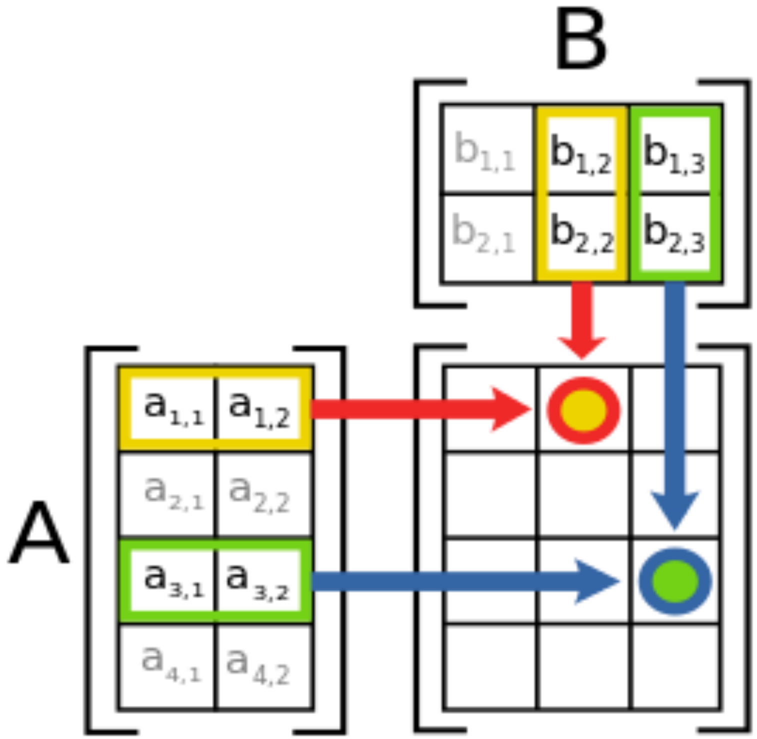
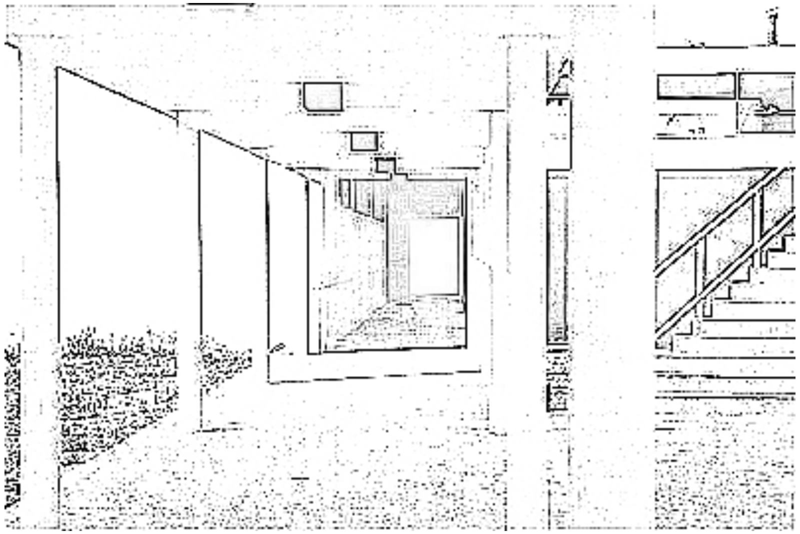


Image from wikipedia

Parallel version: block oriented

Edge detection: convolution operator



Parallel or sequential operation?