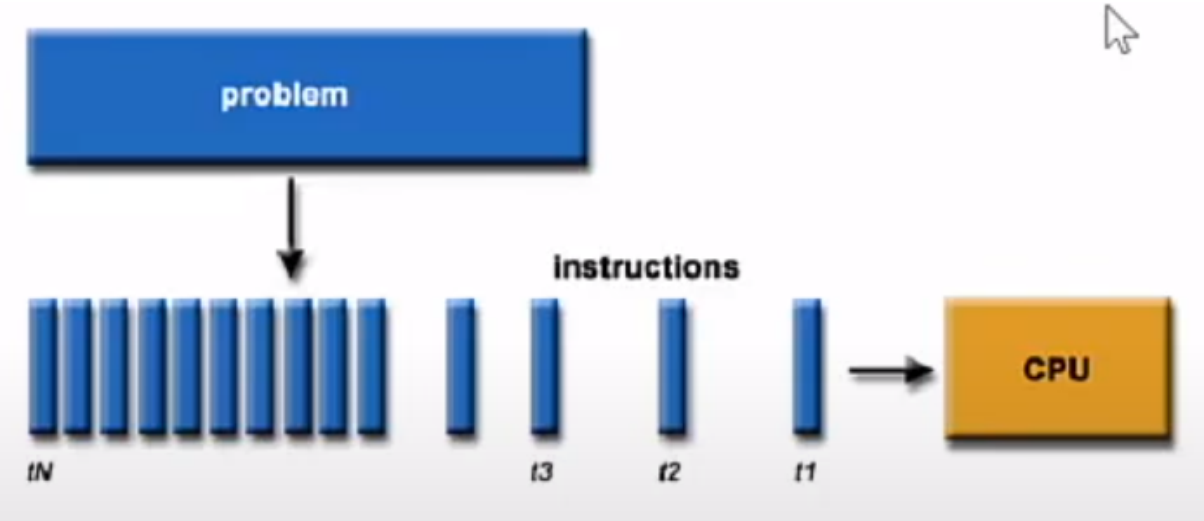
Introduction

In this unit we are going to understand why parallel computing has become so popular, how the execution time of an application can be reduced and the main differences between throughput and parallel computing.

Serial execution

We know that programs are made of instructions. When they are executed in the right order, they are useful to solve the program for which they were designed.

Usually, programs are written in a serial execution model.
This means that the execution of the program implies the execution of a sequence of instructions in the CPU (one after another, one at a time).



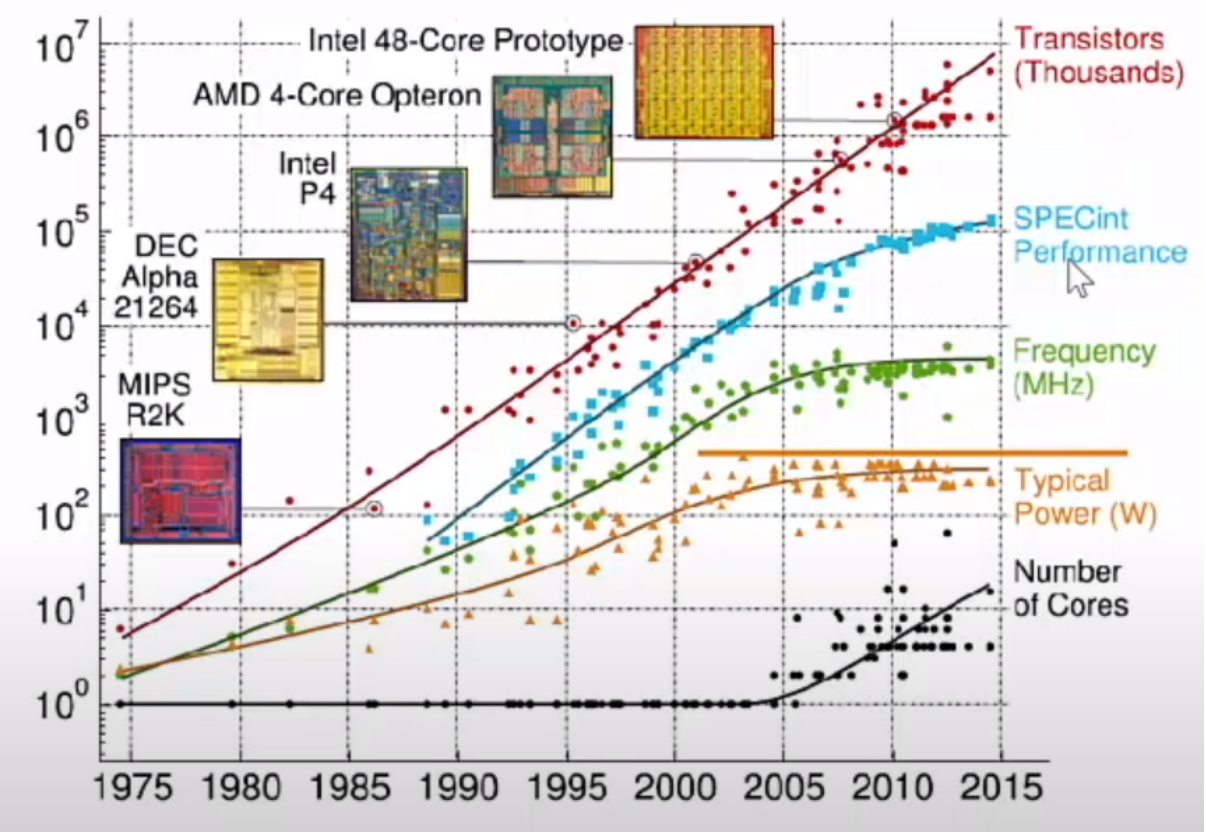
Computers with a single CPU basically interpret the program instructions in this way.

If instructions are executed one after another, the expected execution time is:

- If the program has N instructions to be executed
- The CPU is able to execute F instructions per second
- The execution time is N/F

Different processors in the market may operate at different frequencies. We execute the program faster by augmenting the frequency of the processor (F).

Technological limitations lead microprocessor industry to multicores



The number of transistors in a chip has increased (it doubles every 2 years → More's law).
SPECint → Performance
Frequency of the processor
Typical Power (W) consumption. It reaches a point in which it is not sustainable.
Number of cores

Increased operation frequencies of electronic circuits typically imply higher power consumption and heat dissipation.Around 2004, computer designers shifted into a new paradigm. In order to take profit of the number of transistors that could fit in a chip, without increasing the CPU complexity and operating frequency, they started to increase the number of processing units (cores) in the same chip.
Meaning that there are several processors available in a single chip. This was the starting point of the so called Multi-core era.

The question is: How can we exploit those features of the current processors? How can we use this architecture to make programs faster if the processor complexity and operation frequency do not increase?
So, from 2004 the importance of exploiting parallel architectures has become a must.

Back to reducing the execution time of a program, another possibility is to exploit the parallelism of an application. Problems can be divided in parts and instructions can be runned at different CPUs at the same time, in parallel.

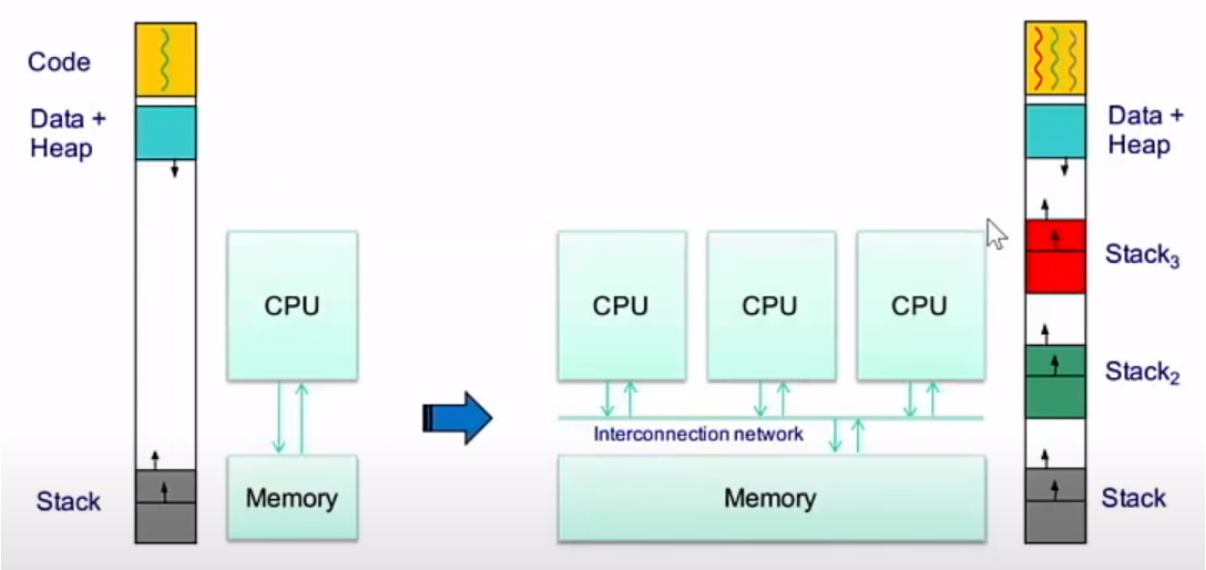
We will refer to these different parts as “tasks”.

Multiprocessor systems are traditionally classified in 2 groups:

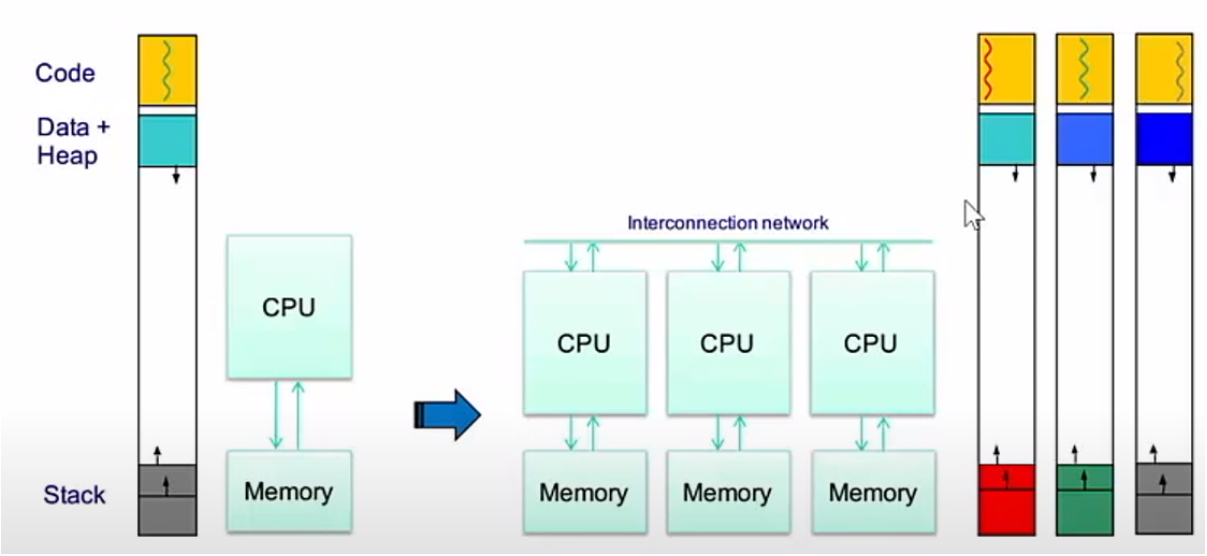
- **Shared memory architectures:** The memory is shared by all CPUs in the system. Each CPU is connected to the shared memory through an interconnection network usually called “bus”.
  Thus, all CPUs can access any memory location.

  As in single CPU architectures, there is a single address space.
  All execution threads share the access to code:
  - Heap for dynamically allocated data structures
  - Data for statically allocated data structures.

  However, in this shared data space, each CPU has access to its own stack for its private data.



- **Distributed memory architectures:** Each CPU is directly connected to its local memory, using regular loaded storage instructions to access it.
  However, other CPUs do not share the access to it. Instead, they communicate through an interconnection network, which allows the data to be shared.



As we can see on the right, the address space is different for each of the processes running in the CPUs. Each process has its own code, heap, data and stack.

Going back to the execution time of a parallel application, ideally it can be divided among the "P" CPUs in a shared memory multiprocessor system. So, the running time will be:

$$T = (N \div P) \div F$$

Compared to a serial execution time, this is much faster.
Of course, those parts in which we split the application (we named as "tasks") may need to be coordinated to obtain the same output result as with the serial execution.
So, we need to manage and coordinate the execution of tasks, ensuring correct access to shared resources.

So far, we have talked about how to exploit a parallel computer using parallel computing.
We can also exploit parallel computers by running different unrelated applications at the same time.

So, if we have “K” applications and “P” processors:

- Those applications can use P/K processors each, to speed up their processing using parallel computing or just executing serially.
- Thus, in a parallel computer with enough resources to execute all K applications in parallel (starting all of them at the same time and using throughput computing), the applications will finish their executions in the same number of units of time as the slowest application.

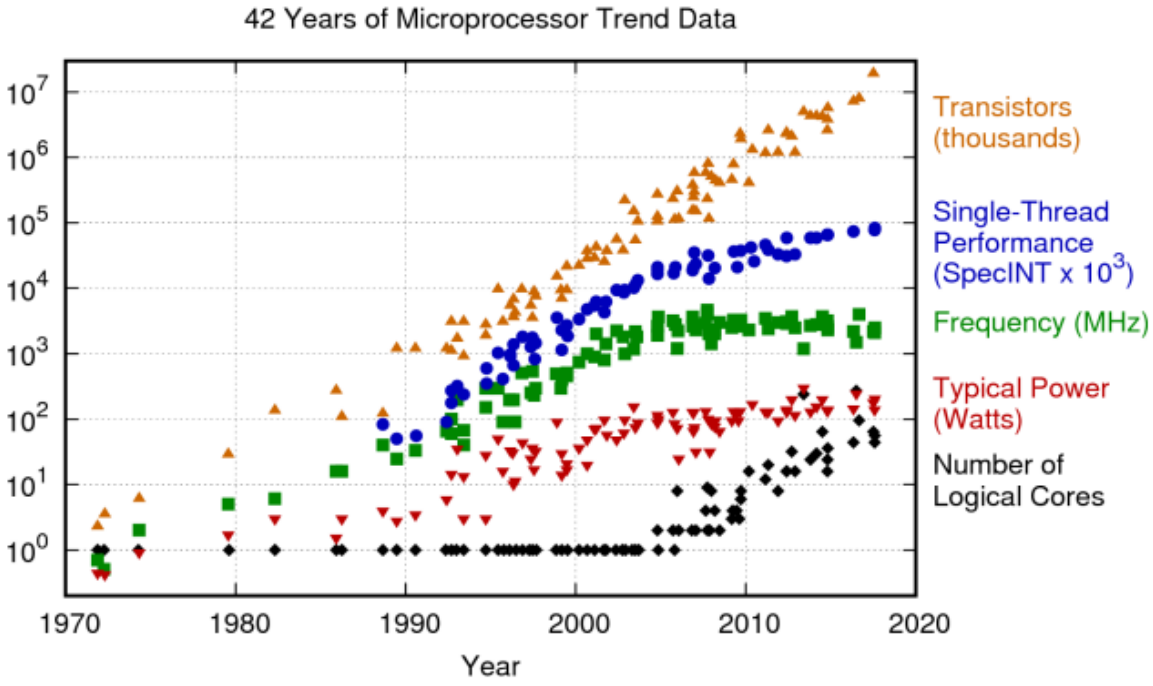
  For example, if we have to run 4 applications and we only have 1 processor, the total execution time will be addition of the execution time of all 4 applications.
  If we have 4 processors, each application will be executed in a different processor. Thus, the execution of all 4 applications will be finished by the time the slowest of all 4 applications finishes.

Unit 1. Introduction and Motivation

Uniprocessor and multicore performance evolution

As we can see, the trend has continued to grow:

- The number of transistors: Gordon Moore created a law that stated “in view of the trends, the number of transistors we would have in a chip will double every 2 years”. For many years, these transistors were put together to create very powerful microprocessors → Translated in higher performance
  The way transistors were organized also made the frequency of operations increase (but up to some point).
  The thing is that the power consumption was growing too much until it was too much (electricity bill was too high).

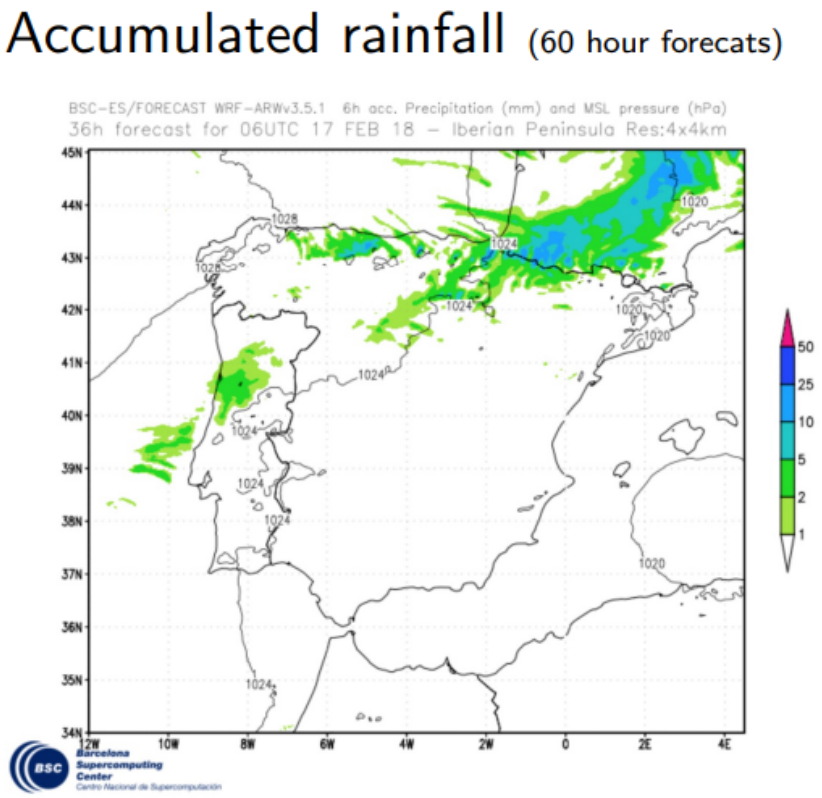


Why parallelism in the 2000-present?

- Power consumption is putting a hard technological limit, as mentioned before.
- Diminishing returns when trying to use transistors to exploit more instruction-level parallelism (ILP). This means that at some point, having more transistors doesn’t help.
  Therefore, there was a shift of paradigm for creating microprocessors → Using more cores (as we can see at the bottom of the graph).
- To scale performance, put many processing cores (CPU) on the microprocessor chip instead of increasing clock frequency and architecture complexity
  - This vision creates a desperate need for all computer scientists and practitioners to be aware of parallelism.
  - Using many cores reduces power consumption, diminishes returns in ILP… The thing is that in the future we will have lots and lots of cores.

Example of using more cores to solve a problem:
Here we have an image of the accumulated rainfall in Spain.
We have a mathematical model, a lot of observations (data), sensors…

If we want to have a weather forecast for the next 60 hours, we can not wait 8 days to obtain the results. So, we need to obtain the results fast.



Machine	Parallel	Sequential
MN3	32 min (128 cores)	2.5 days approx.
MN4	23 min (128 cores)	

In this table we have information from 2 different computers and if they use parallel or sequential execution.

- Sequential execution is not useful because it is too slow. Maybe it's correct, but it is too late.
- Parallel execution for MareNostrum3 is very fast and for MareNostrum4 is even faster.

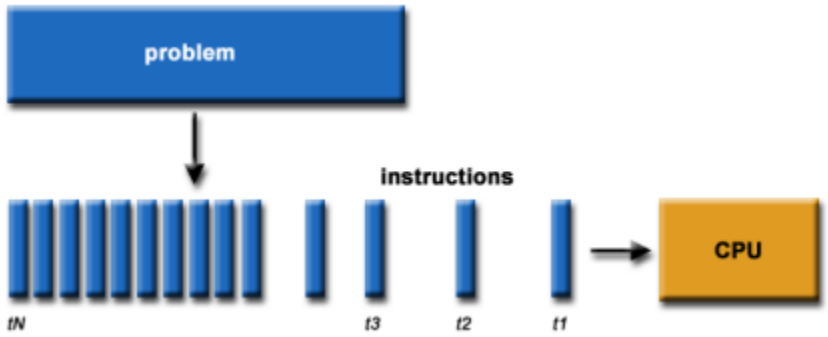
Having more cores (processors) is not indicative of having more power.

Concurrency and Parallelism

Serial execution

Traditionally, programs have been written for serial execution

- To be run on a computer with a single processor (CPU)
- Program is composed of a sequence of instructions and they are executed one after another, only one at any moment in time.



Concurrent execution

Exploiting concurrency consists in breaking a problem into discrete parts, to be called **tasks** (group of instructions that need to be executed), to ensure their correct simultaneous execution.

Let's imagine that we take these 4 rectangles (instructions) and name them as task 1, then I take the next 3 and name them as task 2, then I take the next 10 and call them task 3.
Note that the tasks do not need to have the same amount of instructions.

Concurrent execution → The tasks will be executed sequentially

But multiple tasks multiplex (interleave) their execution on the CPU so that the CPU time is distributed among the tasks. The OS is in charge of saying which is the currently running task, for example after its time quantum expires, starting or resuming the execution of another task.
Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources.

As a summary, concurrent execution means that we make an effort of grouping the different sources that conform our program into different pieces (tasks) and then they are sent for execution. If we only have one worker, it will execute the tasks one at a time (sequential execution).

The tasks do not need to be executed in the same order. We can start solving the first task and then returning to the first task (as long as the final result is okay).
Because if we are adding 2 numbers, I first need to know which are the 2 numbers. So, sometimes there are instructions that depend on other information and therefore the order of the tasks should not be arbitrary.

Question: The instructions within a task need to be executed in the same order?

Example: Client/server application

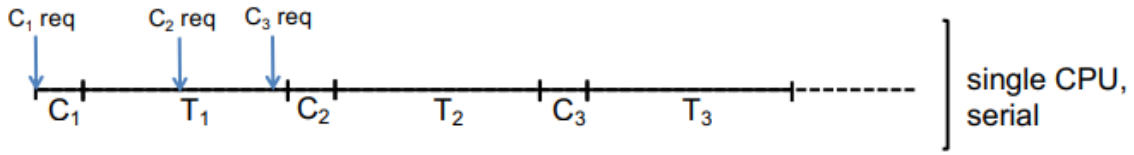
Let's imagine that we have a single CPU.
Client connection implies the execution of the client task (C). As a response, the server task (T) is executed.

Serial execution of client and server tasks:
When a client request arrives, we will be executing the code to see what is needed. Then we will be serving it with T1, which represent the work that the server needs to do serve something.

Let's imagine that we have several more requests (C1, C2, C3).
To execute C2, we need to execute another piece of code. So, even if we receive multiple client requests, we can not execute their corresponding piece of code unless we have finished the previous requests.

So, as we can see, we first receive C1, then we read which is the request and we execute T1, then we read C2 and execute T2…

Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw/deposit some money in bank account)



Only when we finish T1, we will execute the code for the second client C2.
So, we are executing a single thing at a time.

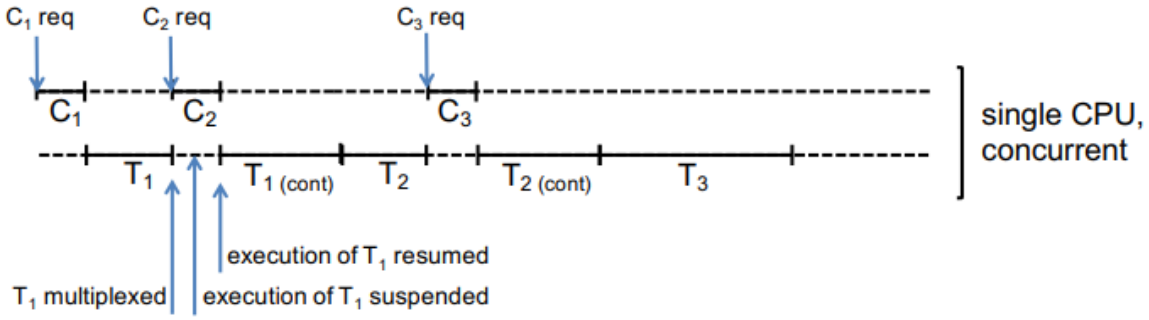
Concurrent execution of client and server tasks, but server tasks serialized:

So, we need to break the work into pieces and then decide when we can do the different tasks.

We still have a single CPU but concurrently executing client and server tasks.

Let's say that we want to execute the tasks of the client fast. We can just stop the server tasks and run the client tasks.
So, we freeze the server tasks (to have the CPU available) in order to execute the client tasks.

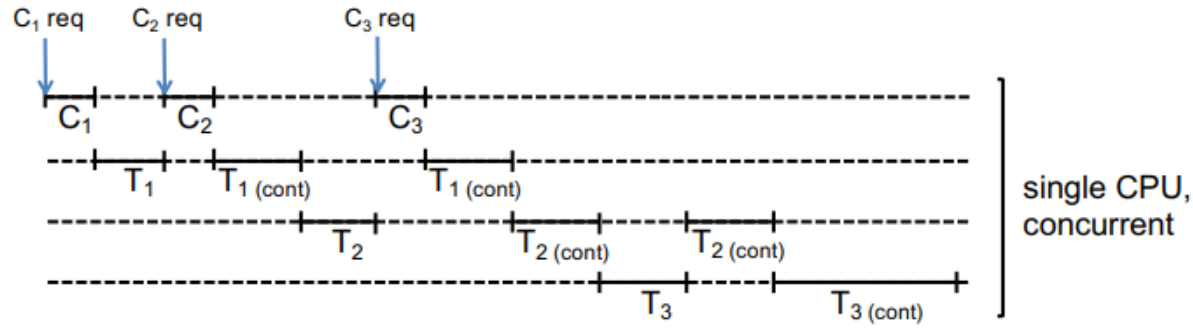
The OS is responsible for this and it is called multiplexed in time.



The dash line means that no CPU is used and the continuous line means that the CPU is used. For this reason, we do not have a continuous line in both Server and Client.

We can have the impression that there is parallelism, because the frequency of operations is very fast.

Concurrent execution of client and multiple server tasks



Here we also have concurrent execution between the different tasks. So, we have concurrent execution between client and server and also within the server.

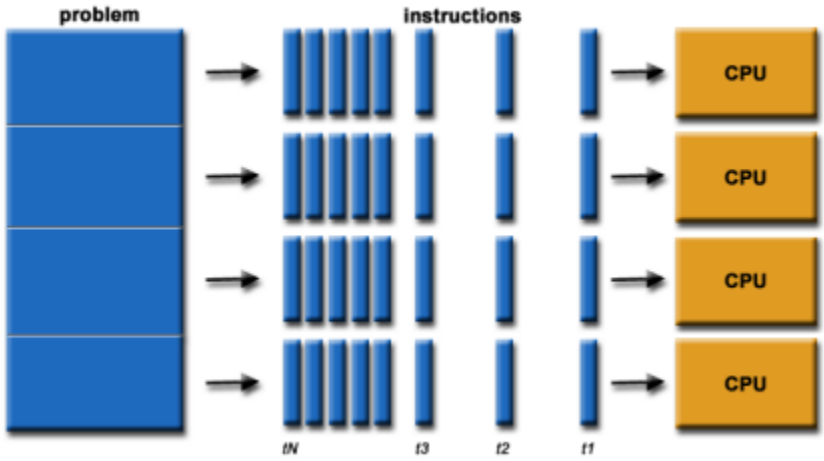
In the other example there was only concurrency between the client and the server.

If we add multiple CPUs, then we will have parallelization and therefore we will have multiple continuous lines in the different levels.

Parallel execution

In the simplest sense, parallelism is when we use multiple processors (CPU) to execute in parallel the tasks identified for concurrent execution

- Ideally, each CPU could receive 1/p of the program, reducing its execution time by p

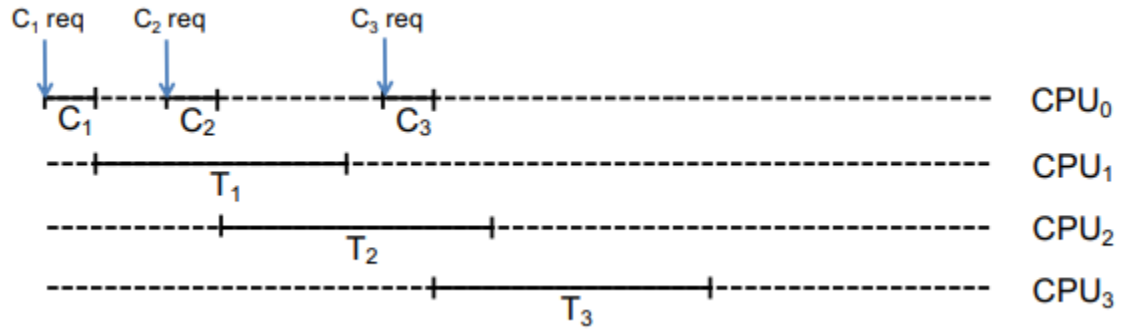


This does not happen in the real world because some instructions need information from previous executions. Thus, there must be an order and not all instructions can be executed at the same time. Moreover, there is an “overhead”, meaning that there are other combinations of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task (creating the task, telling each processor what to do, scheduling the tasks to the processors, providing the data for each task, recovering results…).

If we want to add many numbers, the CPUs need to give their private counter value to a general counter. So, there is an extra operation.

Moreover, not all the tasks will have the same amount of instructions (normally, there is no perfect load balance) → **Granularity**

Parallel execution of client and server tasks on several processors



Throughput vs parallel computing

Multiple processors can also be used to increase the number of programs executed per time unit (throughput).

- **Throughput computing:** Multiple, unrelated, instruction streams (programs) executing at the same time on multiple processors. Like listening to music, executing other code and doing dynamic simulations, for example.
- n programs on p processors; if ($n \geq p$) each program receives p/n processors, one processor otherwise

When we have a system (a cluster, for example), one of the goals of the system can be to support many users. In this respect, we talk about throughput.

Notice that this is not the same as parallelism, whose objective is to reduce the execution (response) time of a single program.

HTC vs HPC

High Throughput Computing (HTC)

- Independent tasks that execute concurrently on multiple processors
- Shared access to resources by several applications
- Goal: Optimize the number of tasks per time unit.

High Performance Computing (HPC)

- Multiple, related, interacting instruction streams (single program) that execute simultaneously
- Exclusive/shared access to a large number of computational resources by a single parallel application
- Goal: Reduce the execution (response) time of a single application. Not many independent things at a time (like HTC)

Processors vs processes/threads

The processor is a hardware and the workers that are going to execute the instructions can be organized as threads or processes.

- Processes/threads are logical computing agents, offered by the OS (Operating System), that execute tasks.
- Processors are the hardware units that physically execute those logical computing agents
- In most cases, there is a one-to-one correspondence between processes/threads and processors, but not necessarily (it is a OS decision)

What do I need to execute a program?

- Know which is the problem…
- Write the code…which is in the disk of our computer
- To execute this program we need the processor. Thus, we need to somehow bring the program into the processor for execution. The processor has all the transistors (which implement logical gates and therefore they are able to execute instructions to add, multiply, store in memory, load…).

  So, we do not only need the program but also a processor to execute it and the memory.

  We also need to organize things well. When we say that we want to execute a program, the graphical shell will receive our order to execute that program. Then, it will create a process (unit of allocation of resources). This process will be given some resources (memory, for example).

  Now I can execute my program because I can tell the processor: Point to this instruction in memory, read it and execute it.

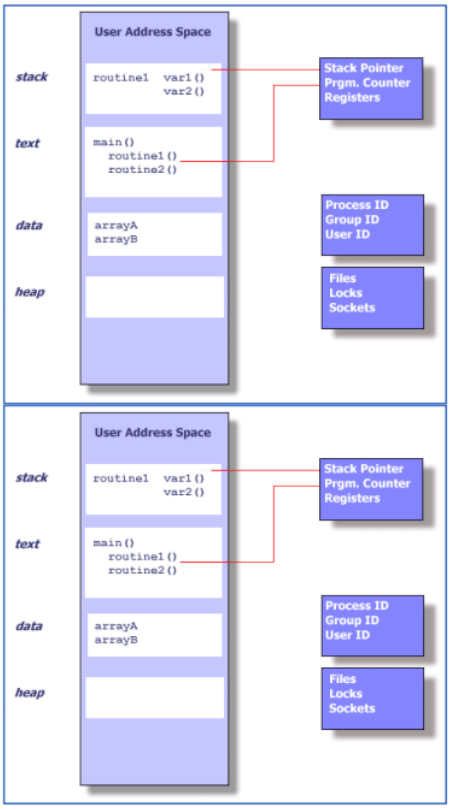
Processes and threads

If instead of having a sequential execution, we have a parallel execution, we have 2 ways to go:

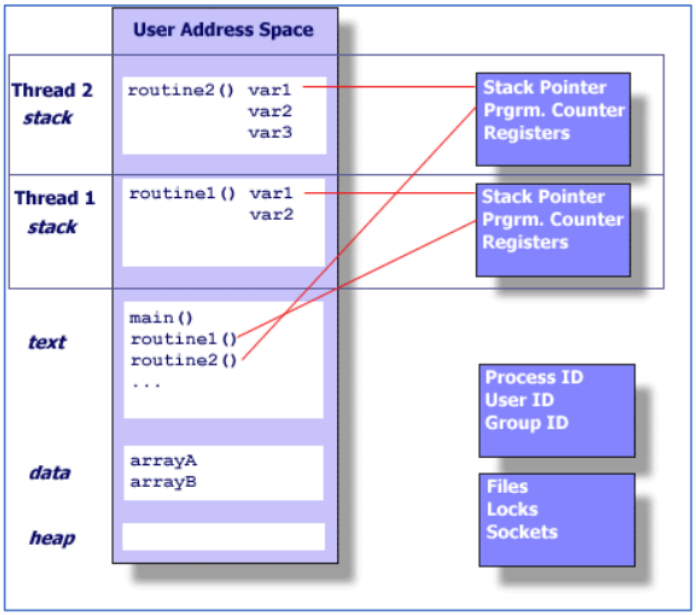
- Replicating the process (creating another process) so that different threads can individually work and at some point put the things together → MPI (Message Passing Interface)
- Still with a single process, we can internally replicate something but still share the code and data. Several workers (threads) share the resources towards solving a common program → OpenMP

Processes do NOT share memory.
Threads DO! Still the threads cna fight with each other and therefore we need to organize them.

Parallel program (processes)

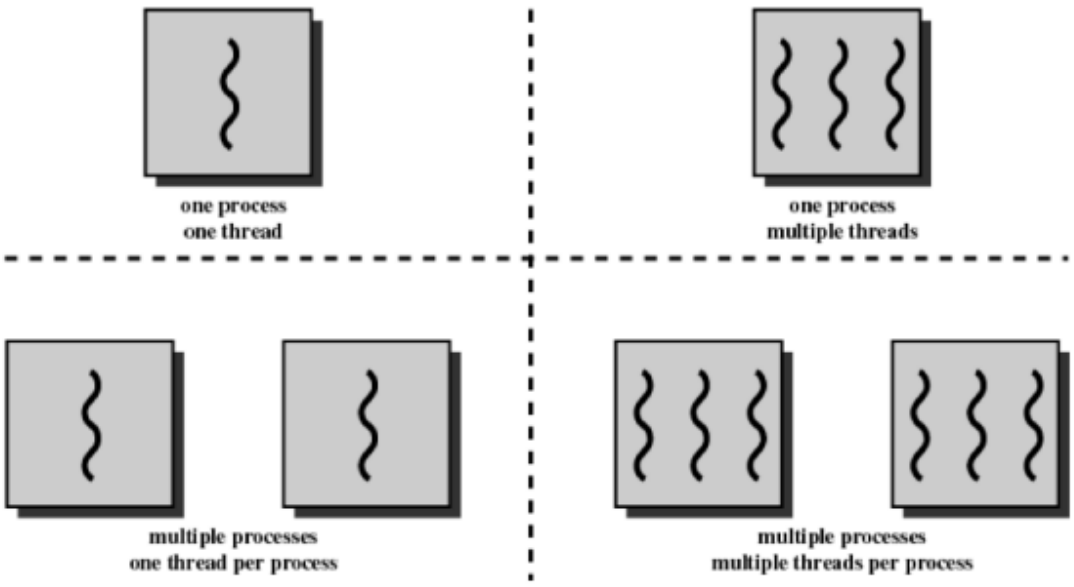


Parallel program (threads)



Processes do not share memory.
Threads do!

Processes and threads and co-exist in the same parallel program.



Parallel Architectures

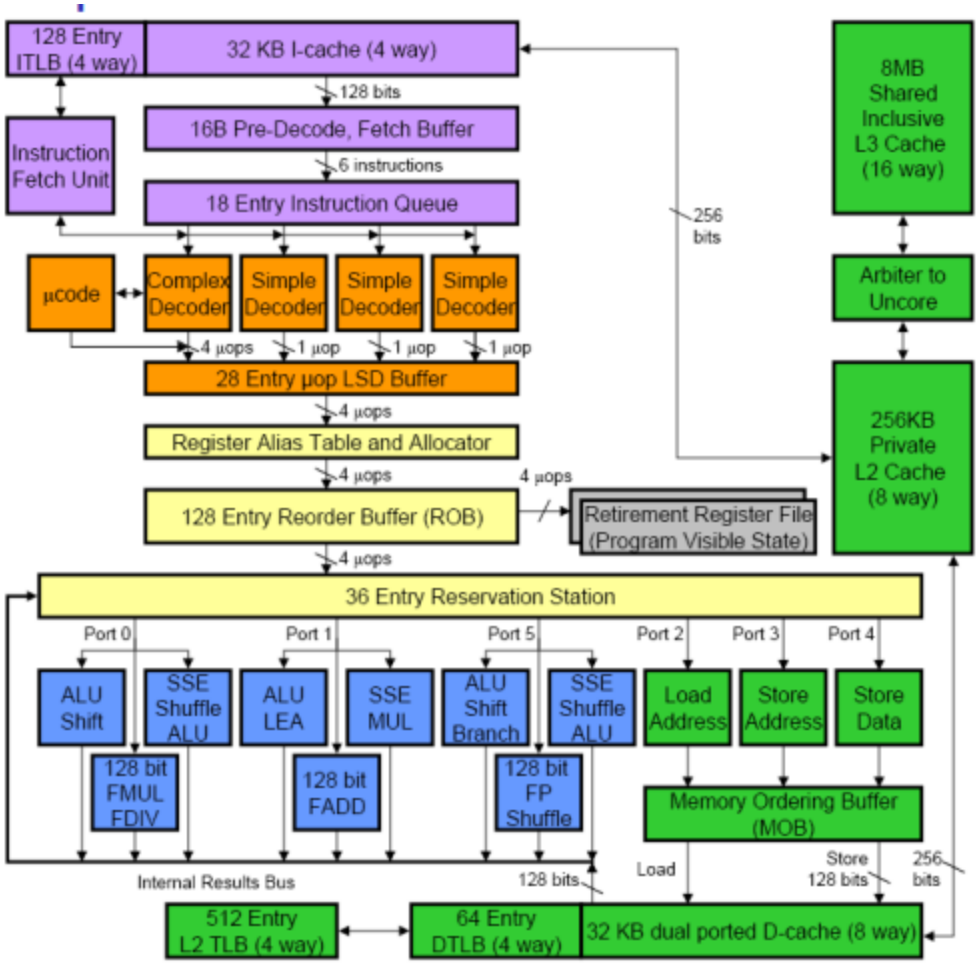
Uniprocessor parallelism (Not Important)

For years, the number of transistors increased. Thus, the processors need to get organized in some way. So, it is the same as our brain:

- Some neurons are doing one thing anthers are doing another thing
- Some transistors are doing one thing and other another thing:
    - Instruction fetch
    - Decoding
    - Executing
    - Computing arithmetic logic operations
    - Writing the results
    - ETC

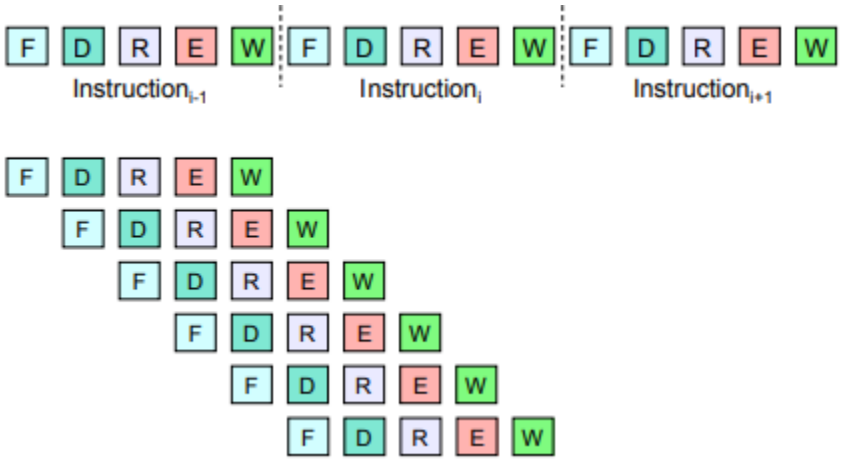
So, several instructions can be executed in parallel within one processor. This is called instruction level parallelism, which was heavily exploited.

Example of uniprocessor architecture: Intel Nehalem i7



Pipelining

- Execution of single instruction divided in multiple stages
- Overlap the execution of different stages of consecutive instructions
    - While I am decoding the first instruction, I am also fetching the second instruction…
- Ideal: IPC=1 (1 instruction executed per cycle)



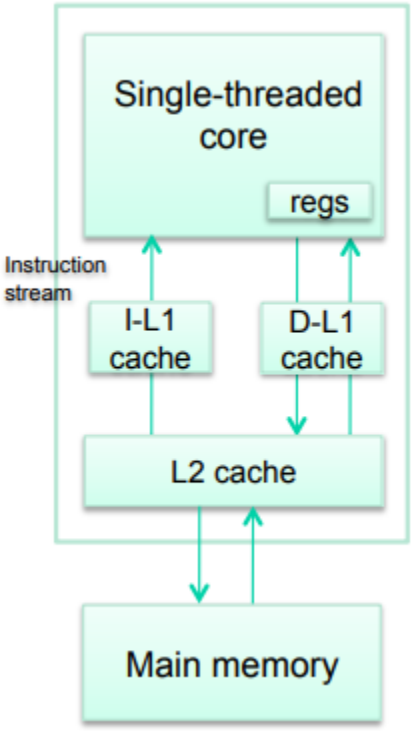
- IPC<1 due to hazards (structural, data, control), preventing the execution of an instruction in its designated clock cycle

Memory hierarchy

Other sources of parallelism is, while I am doing things in the uniprocessor, I can be accessing memory. Addressing the yearly increasing gap between CPU cycle and memory access times

There is a hierarchy in memory:

- Main memory
- Intermediate memory → Cache memory. It is used to store information temporarily. It is very fast
- Registers



It makes sense to use this other memories because of the locality principle:

- When I access some memory location, something close by will be accessed soon. If I access a vector, probability I will visit an element that is close by very soon.

Temporal locality: If an item is referenced it will tend to be referenced again soon (e.g., loops, reuse).
Spatial locality: If an item is referenced items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)

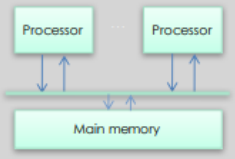
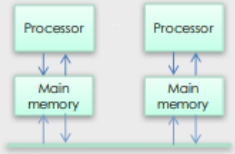
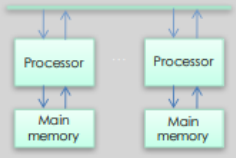
Multiprocessor Architectures (Important)

Classification of multiprocessor architectures

First classification according to the memory:

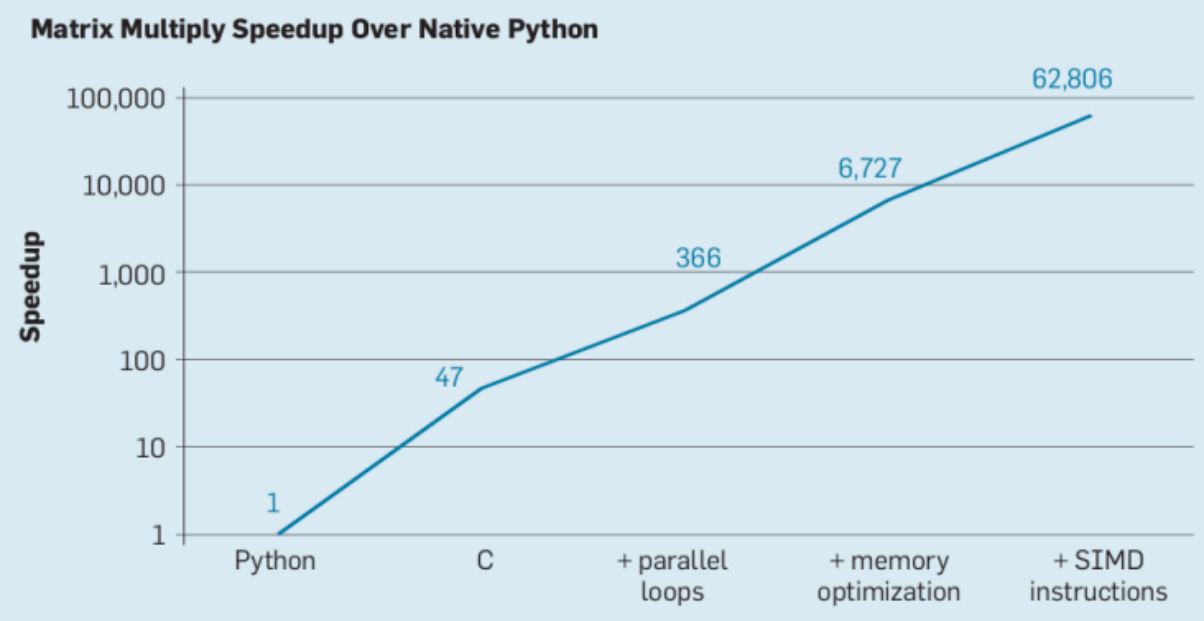
- Shared memory: I have a single process that has the code and the data and several threads of execution are going to share them.

- Distributed memory: When I think of different processes, then I can execute the processes in the same machine (and have the same memory there) or they can go in distributed memory.
  They all have their own memory. But they can still work together as long as they have a network (they can communicate).

Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors	• SMP (Symmetric Multi-Processor) architecture • UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors	• DSM (Distributed-Shared Memory architecture • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	• Message-passing multiprocessor • Cluster Architecture • Multicomputer

Implications on Code Optimization

Matrix Multiply Speedup Over Native Python



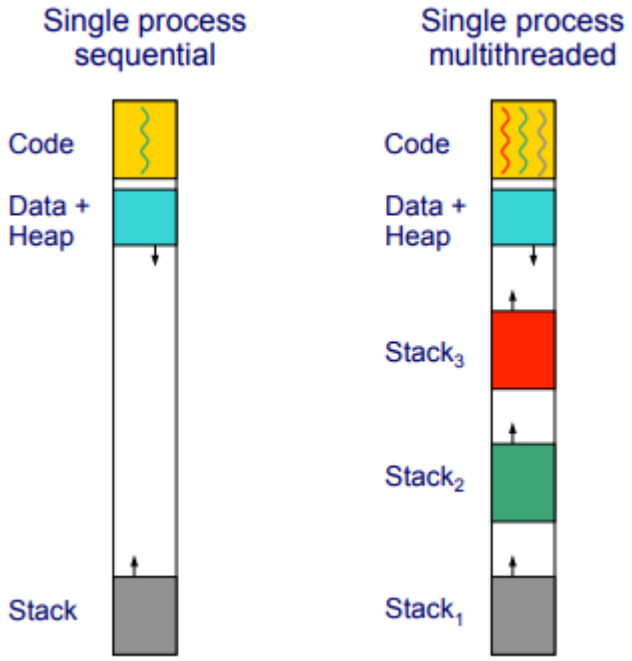
1. Simply rewriting the code in C from Python increases performance 47-fold. 0 and 1 are what the transistors understand. If we use a more “native” language of the computer, it will go faster.
2. Using parallel loops running on many cores yields a factor of ≈ 7.
3. Optimizing the memory layout to exploit caches yields a factor of 20,
4. A final factor of 9 comes from using the hardware extensions for doing single instruction multiple data (SIMD) parallelism operations that are able to perform 16 32-bit operations per instruction.

All told, the final, highly optimized version runs more than 62,000x faster on a multicore Intel processor compared to the original Python version.

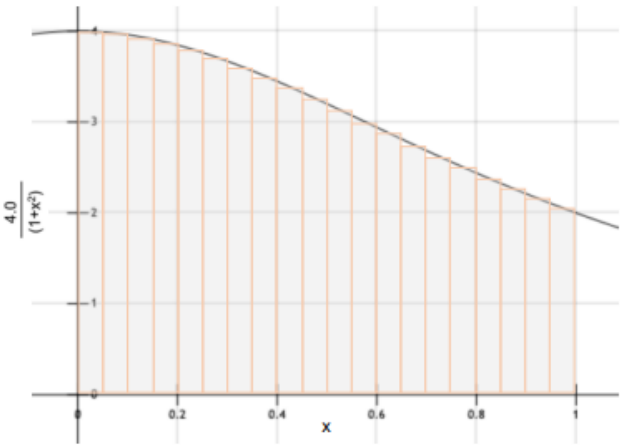
Address Spaces and Parallel Programming Models

Shared memory: address space

We just have a single address space and internally we are going to have several threads of execution.



Shared memory programming: Example



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Sequential code:

```
void main ()
{
   int i;
   double x, pi, step, sum = 0.0;
   long num_steps = 100000;

   step = 1.0/(double) num_steps;

   for (i=1;i<= num_steps; i++) {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Work distribution: each processor is responsible for computing some rectangles (in other words, to execute some iterations of loop i). We need to guarantee that all processors have computed their contribution to sum before combining it into the final shared value (reduction to a single final scalar obtained by adding partial results).

If we use small rectangles, it will give a good but slow result. If we use bigger rectangles, it will be fast but bad.

OpenMP code (not the shortest one, just for illustration purposes):

```
#include "omp.h"
#define NUM_THREADS 4
void main ()
{
   int i, id;
   double x, pi, step, sum[NUM_THREADS];
   long num_steps = 100000;

   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);

   #pragma omp parallel private(id)
   {
      id = omp_get_thread_num();
      sum[id] = 0.0;
      #pragma omp for private(x,i)     // Implicit barrier synchronization at the end of for
      for (i=id; i<= num_steps; i+=NUM_THREADS) {
           x = (i-0.5)*step;
           sum[id] += 4.0/(1.0+x*x);
      }
      #pragma omp single               // Only one thread computes final result
      for (i=0, pi=0.0; i<NUM_THREADS; i++)
          pi += step * sum[i];
   }
}
```

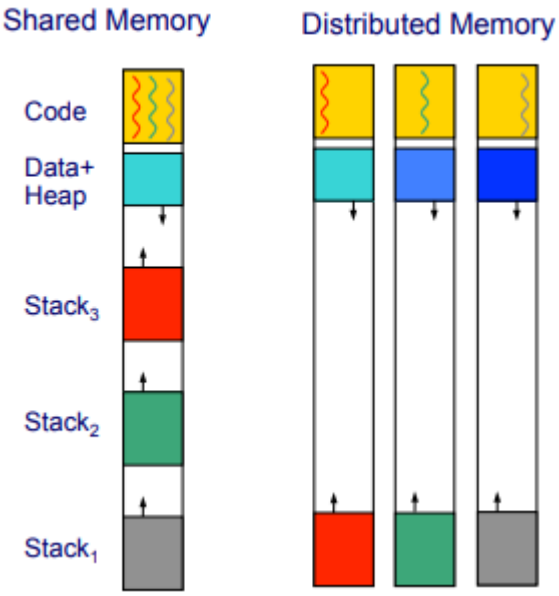
Optimal OpenMP code:

```
void main ()
{
   int i;
   double x, pi, step, sum=0.0;
   long num_steps = 100000;

   step = 1.0/(double) num_steps;

   #pragma omp parallel for private(x) reduction(+:sum)
   for (i=1;i<= num_steps; i++) {
        x = (i-0.5)*step;
        sum += 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

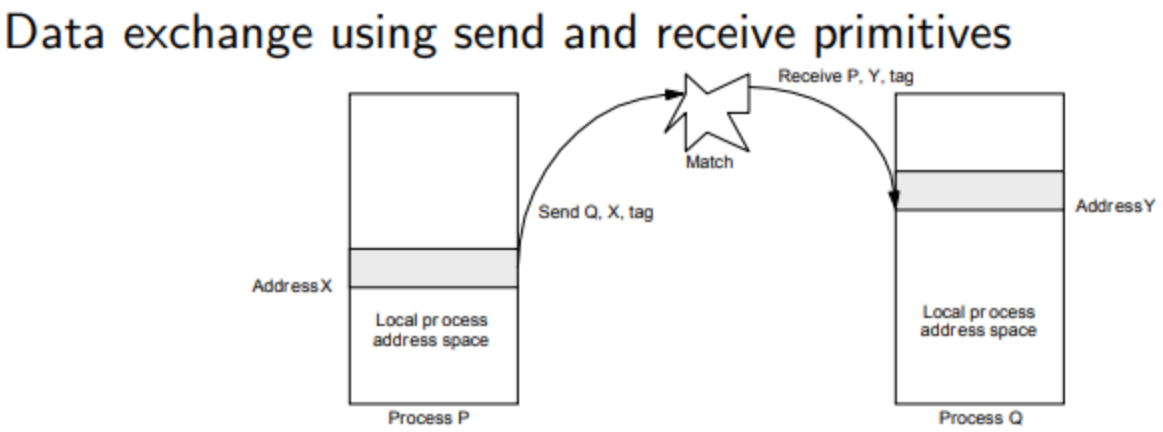
Distributed memory: address space



Programmer needs

- Distribute work
- Distribute data
- Use communication mechanisms to share data explicitly
- Use synchronization mechanisms to avoid data races

Distributed memory: Communication model



- Send specifies buffer to be sent and receiving process
- Receive specifies sending process and application storage to receive into
- Optional tag on send and matching rule on receive
- Optional implicit synchronization (e.g. blocking receive)

Distributed memory programming: Example

Sequential code:

```
void main ()
{
   int i;
   double x, pi, step, sum = 0.0;
   long num_steps = 100000;

   step = 1.0/(double) num_steps;

   for (i=1;i<= num_steps; i++) {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Data/work distribution: each processor is responsible for computing some rectangles and has a private copy of sum. We need to reduce partial results stored in (local) copies of sum into a final value.

MPI code:

```
#include <mpi.h>
void main ()
{
   int i;
   double x, pi, step, sum=0.0;
   long num_steps = 100000;
   int numprocs, myid;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   step = 1.0/(double) num_steps;

   for (i = myid; i < num_steps; i += numprocs) {
        x = (i + 0.5)*step;
        sum += (4.0 / (1.0 + x*x));
   }
   mypi = step * sum;

   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   MPI_Finalize();
}
```

Hybrid MPI/OpenMP code:

```
#include <mpi.h>
#include "omp.h"
void main ()
{
   int i;
   double x, pi, step, sum=0.0;
   long num_steps = 100000;
   int numprocs, myid;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   step = 1.0/(double) num_steps;

   #pragma omp parallel for private(x) reduction(+:sum)
   for (i = myid; i < num_steps; i += numprocs) {
        x = (i + 0.5)*step;
        sum += (4.0 / (1.0 + x*x));
   }
   mypi = step * sum;

   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   MPI_Finalize();
}
```

Video Before Class 2. Motivation

As seen, we have a database of the cars available in different retail stores.
For each database record, we can see the different fields:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

We want to perform the following query: **How many green cars are available to sell?**

The simplest way to proceed would be to traverse all the records in the DB and count the number of times there is a green car.

This traversal can be expressed in the form of a sequential program:

```
count = 0;
for ( i = 0 ; i < n ; i++ )
    if (X[i].Color == "Green") count++;
```

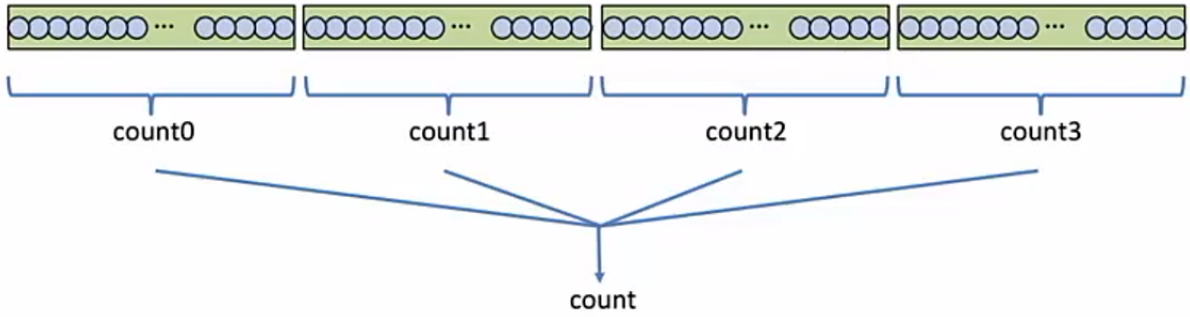
We can clearly see that the computational time of this loop is proportional to “n” (number of records in the DB). Since we need to traverse the whole DB.

Another option would be to divide the traversal of all the elements in the DB in P groups that we will name **tasks**. Each task will traverse N/P consecutive records and count the number of occurrences in a per-task “private” copy of variable count.

Is any dependency among the computations done in different tasks? In other words, can you compute the number of occurrences in the first task independently of computing the number of occurrences in the last task? YES. Since each task is using a private copy of the variable count.

Thus, if they are independent, P workers could do the computations in parallel.

However, we still need to “globally” count the number of records found that match the condition by combining the individual “private” counts into the original count variable.



In this case, we can anticipate that the computation time would be divided by the number of tasks P if P workers are used to do the computation

$$T_P = T_1 \div P$$

With an additional “overhead” to perform this global reduction of the per-task private count variables into the global count variable.

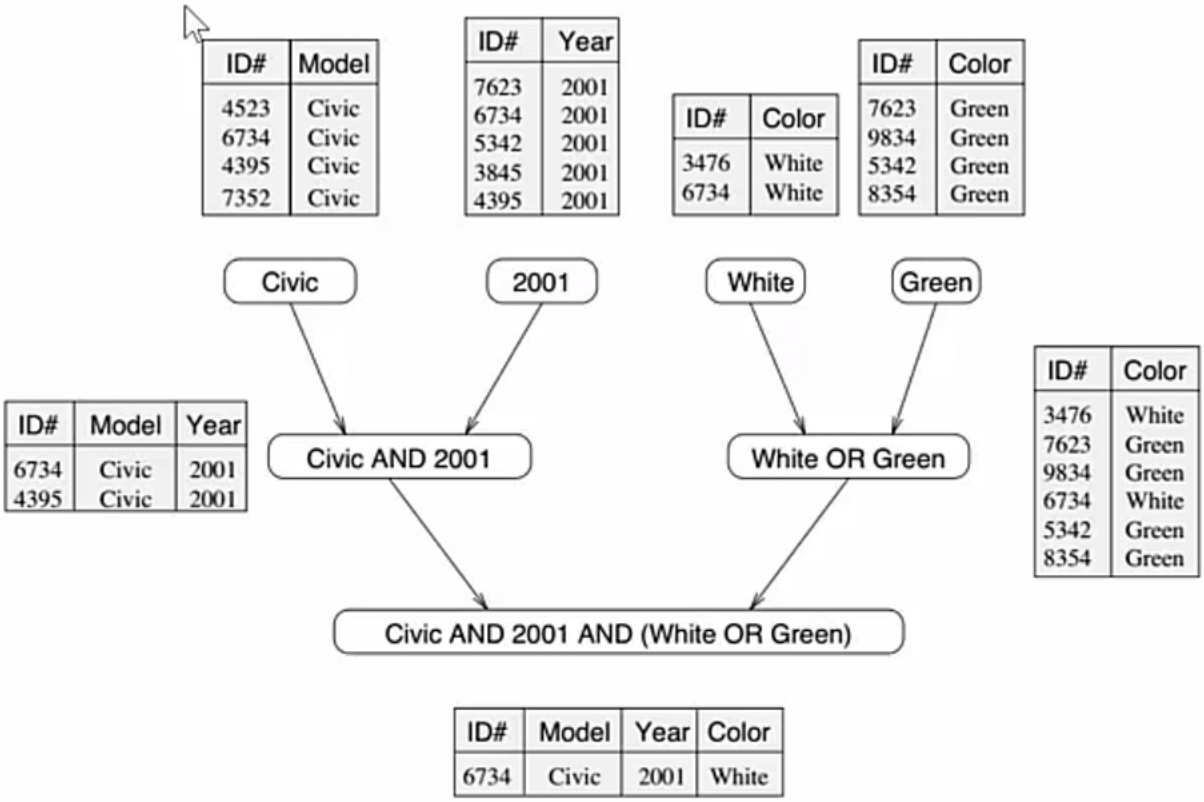
$$T_p = T_1 \div P + T_{ovh}(P)$$

This “overhead” should be proportional to the number of workers P.

Let's consider a more complicated query:

- MODEL = 'CIVIC' **and** YEAR = 2001 **and** (COLOR = GREEN **or** WHITE)

A possible query plan could be:



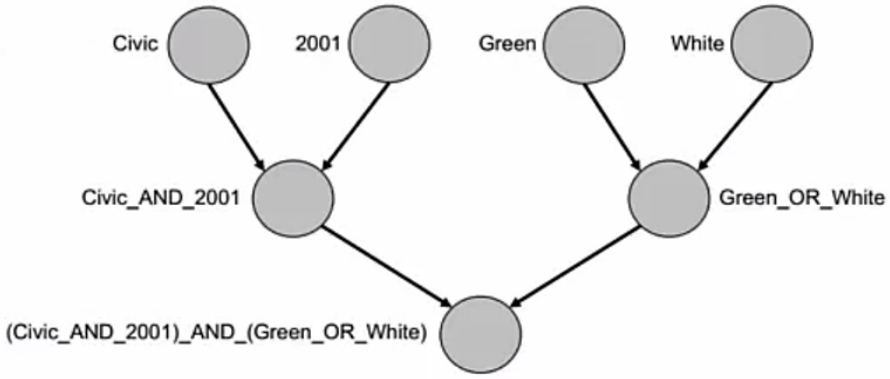
Select all the entries in the DB that fulfill a particular condition individually and create a subset for each of them.
Then we select the entries that fulfill 2 conditions. For each of these 2 queries, a new subset of entries is generated.
Finally we select the common entries in the DB that satisfy all conditions.

Each of these operations in the query plan could be a task, each computing an intermediate table of entries that satisfy particular conditions.

Are they independent?

- Some of them are, for example tasks "CIVIC", "2001", "GREEN" AND "WHITE". So, all inicial subsets are independent.
- Others are not independent. For example task CIVIC_and_2001 can not start its execution until both tasks "CIVIC" and "2001" are complete.

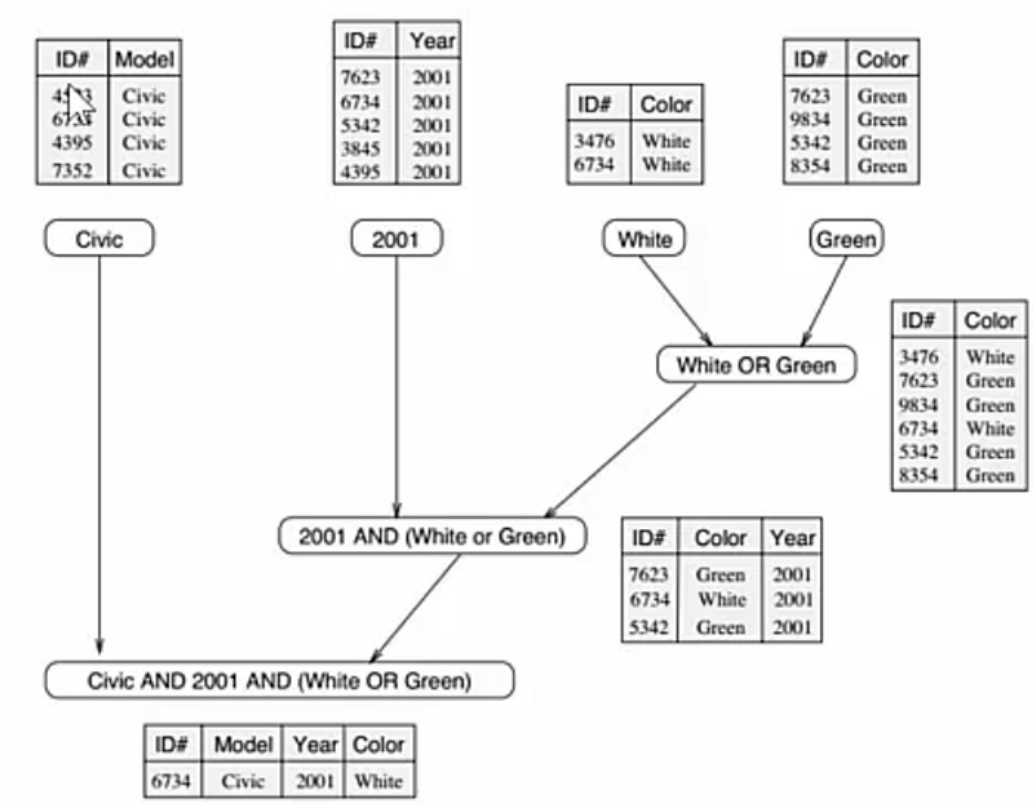
Dependencies impose task execution ordering constraints that need to be fulfilled in order to guarantee correct results



As we can see, tasks “CIVIC”, “2001”, “GREEN” and “WHITE” can run in parallel. Once they have finished, tasks CIVID_AND_2001 and GREEN_OR_WHITE can also run in parallel.

Task dependence graph: Graphical representation of the task decomposition.

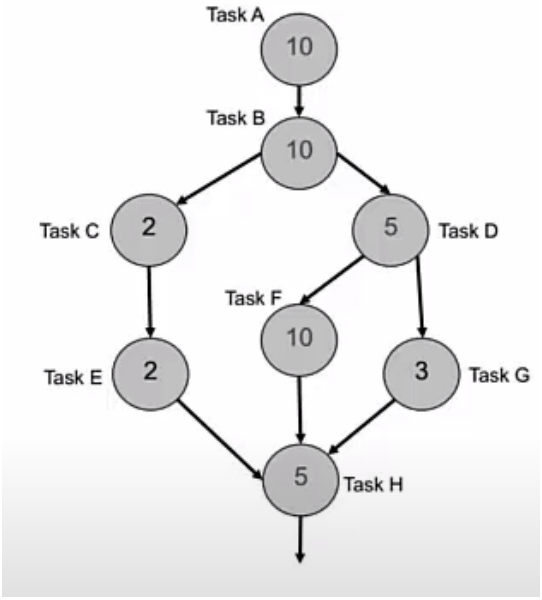
Before going to some formalism on this task dependence graph representation, let’s comment that other query plans would be possible for the same query.



This dependence graph has different potential opportunities to execute tasks in parallel.

Let's go now into some formalism that will allow us to compute the potential parallelism available in a task decomposition.
As we have seen, the task dependence graph is a:

- Directed acyclic graph
- Node = task, its weight represents the amount of work to be done by the corresponding task. Time units, for example.
- Edge = dependence between nodes. The node that appears as a destination of an edge, requires something produced by the node which the edge leaves.
  Thus, the successor node can only execute after the predecessor node has completed its execution.



For example, Task H can not be executed until tasks E, F and G are completed.

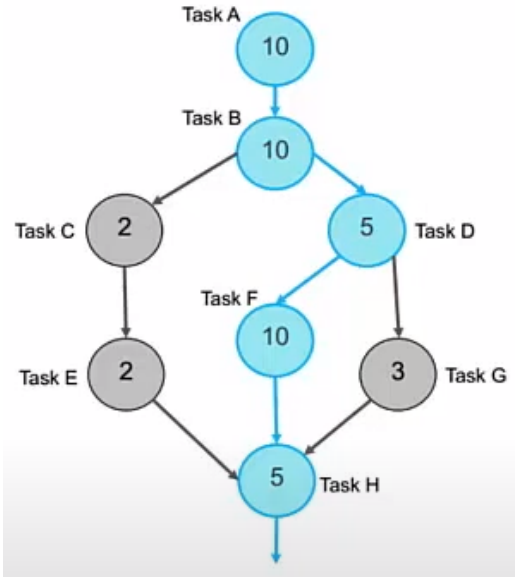
The execution of tasks C and E can proceed in parallel with tasks D, F, and G.

Let's assume a very simple abstraction for a parallel machine composed of P identical processors (each processor executes a node at a time).

In this graph, we can compute T1 as the sum of the weight for all the nodes.
As you may have noticed, T1 corresponds to the sequential execution (or we only have 1 processor).

$$T_1 = \sum_{i=1}^{nodes} (work\_node_i)$$

Critical path: Path in the task graph with the highest accumulated work.



T_{inf} is the weight of that critical path.

$$T_\infty = \sum_{i \in criticalpath} (work\_node_i)$$

It represents the minimum time required to execute the task decomposition expressed in a task dependence graph.

It assumes that there are sufficient resources in the parallel architecture to execute the rest of the nodes in the task dependence graph, in parallel with all those in the critical path.

The quotient $T1/T_{inf}$ represents how much faster we could execute the task decomposition if sufficient processors were available. This quotient is called “parallelism”.

This parallelism can only be obtained if a minimum number of processors are available. This is denoted by “P_{min}”:

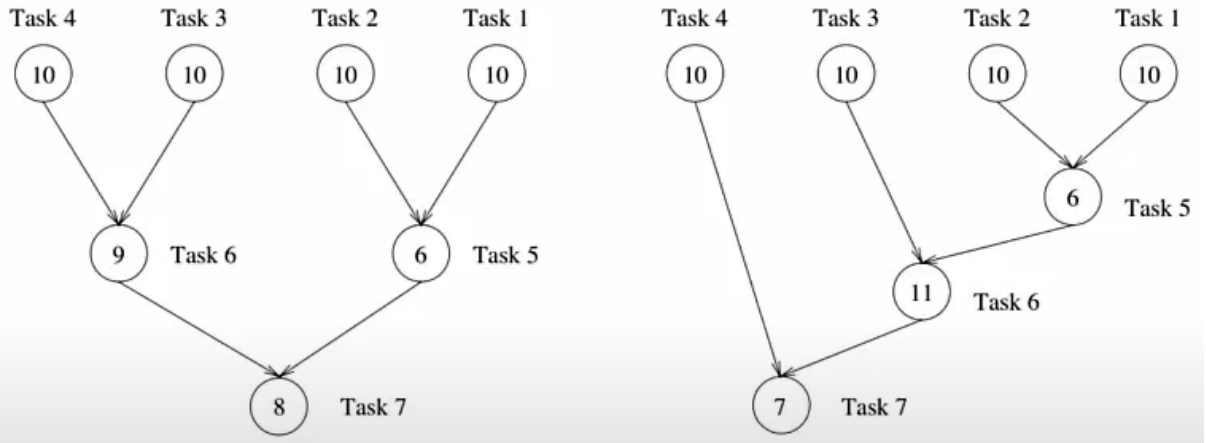
- If the number of processors used is smaller than P_{min}, then the execution time will be larger than T_{inf}. Thus, we will not achieve the parallelism available in the task decomposition.
- Notice that the parallelism is a metric independent of the number of processors. It is a metric inherent to the task decomposition.

In the previous example, the metrics mentioned are:

- **T1** = Tasks A, B, C, D, E, F, G, H = 47
- Possible paths:
    - Tasks A, B, C, E, H = 29
    - Tasks A, B, D, F, H = 40
    - Tasks A, B, D, G, H = 33
- **T_{inf}** = Tasks A, B, D, F, H = 40
- **Parallelism** = 47/40 = 1.175
- **P_{min}** = 2. One processor to execute the critical path and another one to execute the rest of the nodes.

Note that while one processor is executing tasks D and F (15 time units), the other processor can execute tasks C, E and G (7 time units).
Using 3 processors would not reduce the execution time.

Going back to the 2 query plans that we mentioned before, we have the following task dependence graphs:



We will consider that each node is labeled with the work of that node, which is equal to the number of inputs to be processed by the corresponding task. Thus, in the 4 firsts tasks, they all have the same weight (there are the same number of entries to be traversed).

For the first task dependence graph, the critical path is determined by any of the two task sequences {3, 6, 7} or {4, 6, 7}.

- $T1 = 63$
- $T_{inf} = 27$
- Parallelism = 63/27 = 2.33

For the second task dependence graph, the critical path is determined by any of the two task sequences {1, 5, 6, 7} or {2, 5, 6, 7}.

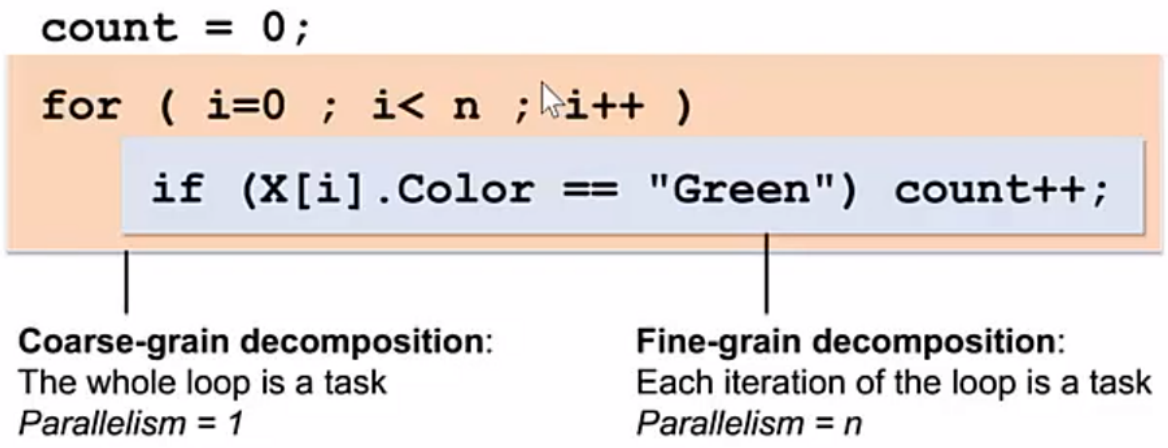
- $T1 = 64$
- $T_{inf} = 34$
- Parallelism = 64/34 =1.88

Do both query plans require the same minimum number of processors Pmin to achieve this potential Parallelism?

Granularity

The last concept we will present in this lesson is the granularity of a task decomposition, which refers to the computational size of the nodes (tasks) in the task graph.

Example: Counting matches in our car dealer DB.



At the **coarse-grain granularity level**, a single task can be identified consisting of all the iterations of the loop. This would result as a single node in the task graph, with execution time equal to N·K.

- N is the number of iterations
- K is the time to execute each iteration

In this case, the parallelism would be 1, which means no parallelism would be exploited.

At the **fine granularity level**, each iteration of the loop is a task. This would result as N nodes in the task graph, each with execution time equal to K.
In this case, the parallelism would be N, which means parallelism could be exploited.

In between both extreme cases, a task could be in charge of checking a number of consecutive elements M of the DB:

- With potential parallelism = N/M

It would appear that the parallelism is higher when going to fine-grain task decompositions

	Coarse grain	Fine grain	Medium grain
Number of tasks	1	n	n / m
Iterations per task	n	1	m
Parallelism	1	n	n / m

However, there is a tradeoff between potential parallelism and overheads related with its exploitations.

Unit 2. Understanding parallelism

Finding concurrency/parallelism

Can the computation be divided in parts?

- Based on the processing to do:
  - Task decomposition (e.g. functions, loop iterations)
- Based on the data to be processed:
  - Data decomposition (e.g. elements of a vector, rows of a matrix) (implies task decomposition)

There may be (data or control) dependencies between tasks
The decomposition determines the potential parallelism that could be obtained

Task graph abstraction

Computing the execution time

Computing the parallelism

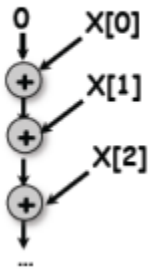
Example 1: Vector sum

Compute the sum of elements X[0] … X[n-1] of a vector X

Sequential algorithm

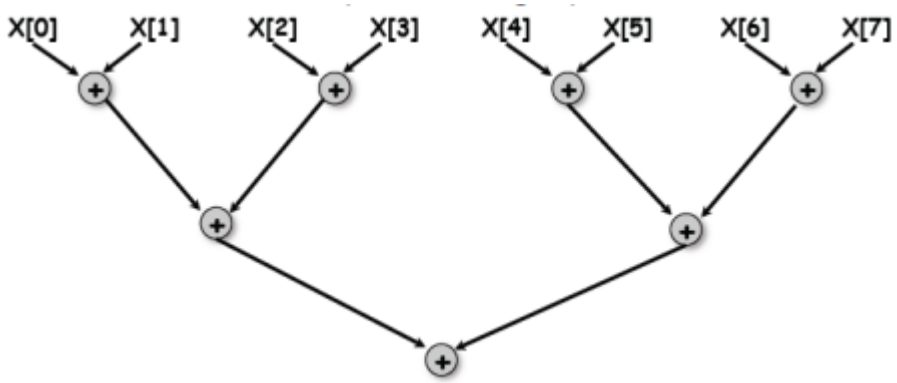
```
sum = 0; for ( i=0 ; i< n ; i++ ) sum += X[i];
```

- Computation graph



$$T_1 \propto n$$
$$T_\infty \propto n$$
$$Parallelism = 1$$

How can we design an algorithm (computation graph) with more parallels?



- $T_1 \propto n$; $T_\infty \propto \log_2(n)$; $Parallelism \propto (n \div \log_2(n))$
- How to restructure the sequential algorithm to have this computation graph? (iterative vs. recursive solutions)

Recursive algorithm

```
int recursive_sum(int *X, int n)
{
   int ndiv2 = n/2;
   int sum=0;

   if (n==1) return X[0];

   sum = recursive_sum(X, ndiv2);
   sum += recursive_sum(X+ndiv2, n-ndiv2);
   return sum;
}

void main()
{
   int sum, X[N];
   ...
   sum = recursive_sum(X,N);
   ...
}
```

Example 2. Database query processing

Video Before Class 3. Motivation

We are going to use the computation of the Mandelbrot set to introduce the key concepts.
The Mandelbrot set is the set of complex numbers p that fulfill a particular condition.
Let's define the following recursive function:

$$z_{n+1} = z_n^2 + p$$

p is the point of interest and we start with z0 = 0.

For each point "p" that we want to evaluate, the sequence starts with element p since z0 = 0.
The rest of the sequence is calculated using the recurrence just defined.

The next value of Zn^2 will be p^2. Thus, Zn+1 = p^2 + p

$$p,\ p^2 + p,\ (p^2 + p)^2 + p,\ ((p^2 + p)^2 + p)^2 + p,\ \ldots$$

The Mandelbrot set is then the set of complex numbers p in a delimited 2-dimensional space for which the sequence created by the recurrence just defined fulfills that the norm of the complex numbers $z_\infty < 2$

Given a complex point "p" with a real component stored as p.real and an imaginary component stored as p.imag, in this slide we show a code which computes a sequence of complex numbers and checks if the previous condition is met.
Each complex number in the sequence is stored in variable "z" with a real component stored as z.real and an imaginary component stored as z.imag.

The important thing for us is to know that the code has two external loops, not shown in the video, which are used to calculate each pixel in a 2D image.

```
n = 0; z.real = z.imag = 0;
do  {
    temp = z.real*z.real - z.imag*z.imag + p.real;
    z.imag = 2*z.real*z.imag + p.imag;
    z.real = temp;
    norm_sq = z.real*z.real + z.imag*z.imag;
} while (norm_sq < (2*2) && ++n < max);
```

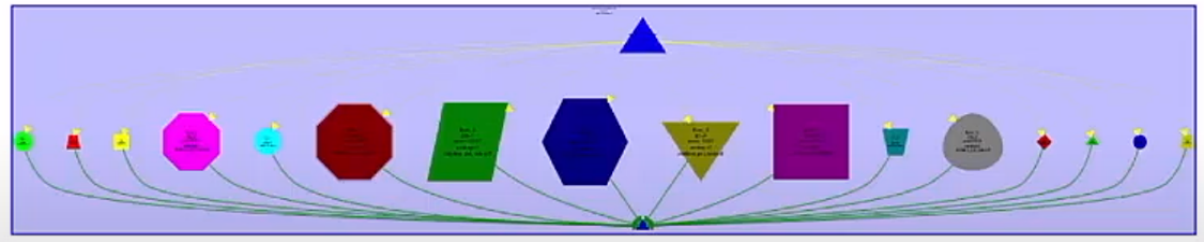
This variable is initialized to 0 before the sequence is computed.
Then within the loop, the next value for complex numbers "z" in a sequence is computed from the previous value.
In order to make the computation of the set practical, the number of steps in the sequence is limited to a maximum value defined by "max".

The do-while loop continues while the maximum number of steps is not reached and the norm of the new value is smaller than 2 (or, equivalently, its squared value is smaller than 4). This last trick is used to avoid computing the square root, which is a costly operation.

The points that will fulfill the condition are painted in black in the graphical representation. The other points are painted with a color according to the number of steps “n” up to which norm of Zn is smaller than 2.

Can the computation of the set be done in parallel? Yes, because each point of the set can be computed independently of the others.
Assume a task corresponds with the computation of the previous recurrence for a set of consecutive rows of the 2-dimensional space.

If we ask Tareador to analyze the task decomposition, it gives us the following task dependence graph:



Each of the nodes corresponds to the computation of a set of rows.

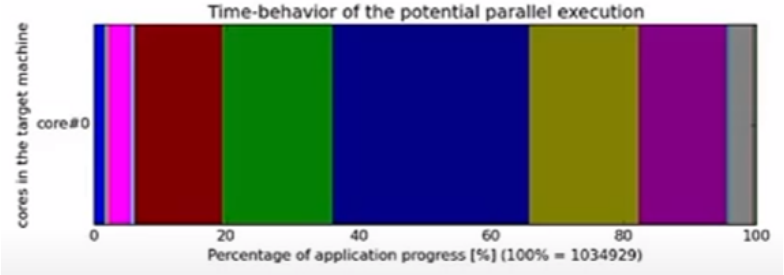
From this graph, we can extract 2 conclusions:

- The tasks are fully parallel → This is called “Embarrassingly parallel decomposition”
- The decomposition is heavily unbalanced in terms of computational load, since some tasks are small and others are big.

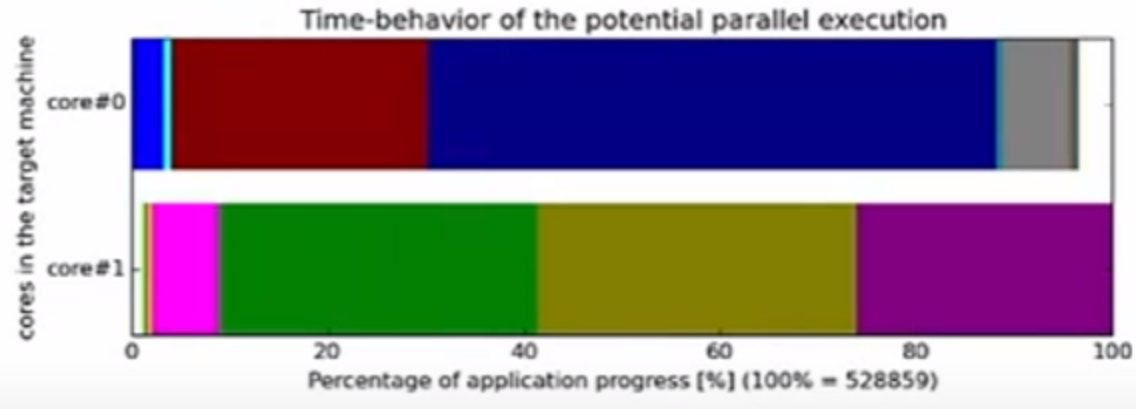
  Is this an issue that should worry us? As we will practice in the second hands-on practice, Tareador can simulate the execution of the task decomposition with different numbers of processors and report the execution time.

  If we execute it with a single processor, the execution time T1 will be 1.034.929 time units, which we will take as reference time for the sequential execution.

  In the following timeline, we can see how the different tasks are sequentially executed (one after the other)



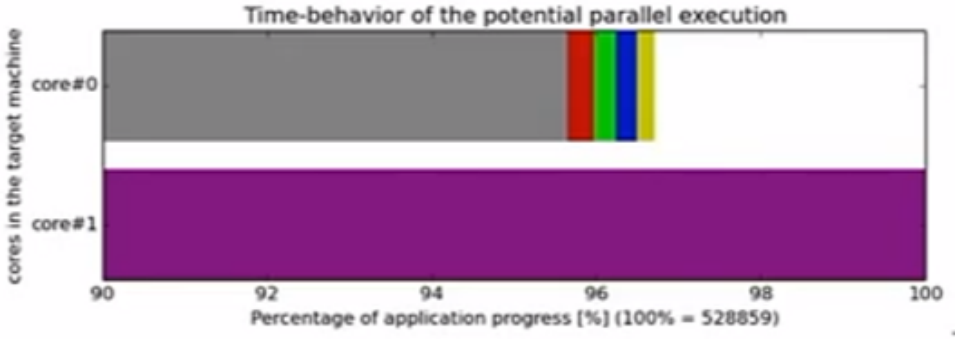
What happens if we execute the task decomposition with 2 processors?
This timeline reveals that tasks are distributed between the 2 processors available.
Each processor is assigned with a subset of the tasks that are sequentially executed



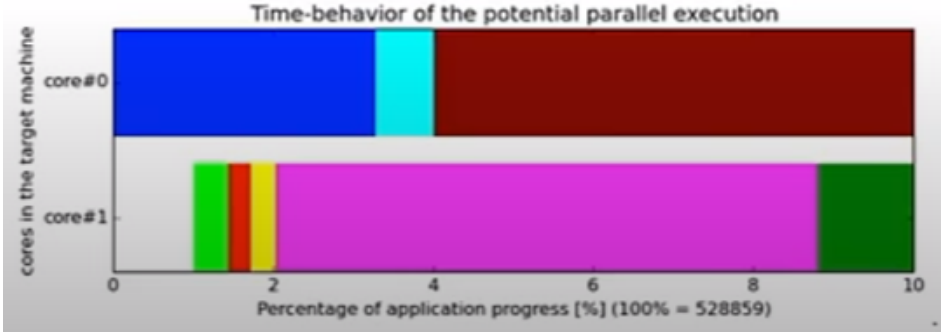
Resulting in an execution time of T2=528.859 time units, 1.95 times faster than the sequential execution.

Why do we not obtain an execution 2 times faster than the sequential execution?
Here we can see a zoomed view of the initial and final 10% of the execution:

- On the one hand, due to the load unbalance detected before, processor number 0 is finishing its job assignment a little bit before processor 1 (leaving one of the processors unused).



- The blue bar at the beginning of the execution of processor 0 represents the time that is needed to create all the tasks that have to be executed.
  This is an overhead of the parallel execution that is also delaying the start of processor 1 and the real execution of tasks in processor 0.



The unbalance and overhead get worse as we increase the number of processors.

- T4 = 3.18 times faster
- T8 = 3.25 times faster
- T16 = 3.25 times faster

So, parallel computing is not an ideal thing.
Although the task decomposition for the Mandelbrot set problem is embarrassingly parallel, the execution in a parallel architecture is not that ideal.

Recap

- In parallel computing, an embarrassingly parallel workload, or embarrassingly parallel problem, is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency between those parallel tasks.
- Embarrassingly parallel problems tend to require little or no communication of results between tasks, and are thus different from distributed computing problems that require communication between tasks, especially communication of intermediate results
- Load balancing is achieved when there is an even (equitable) distribution of the work to be done in the tasks originated in a parallel decomposition among the processors in a parallel system.
- Load imbalance is one of the sources of performance loss in HPC systems and applications.

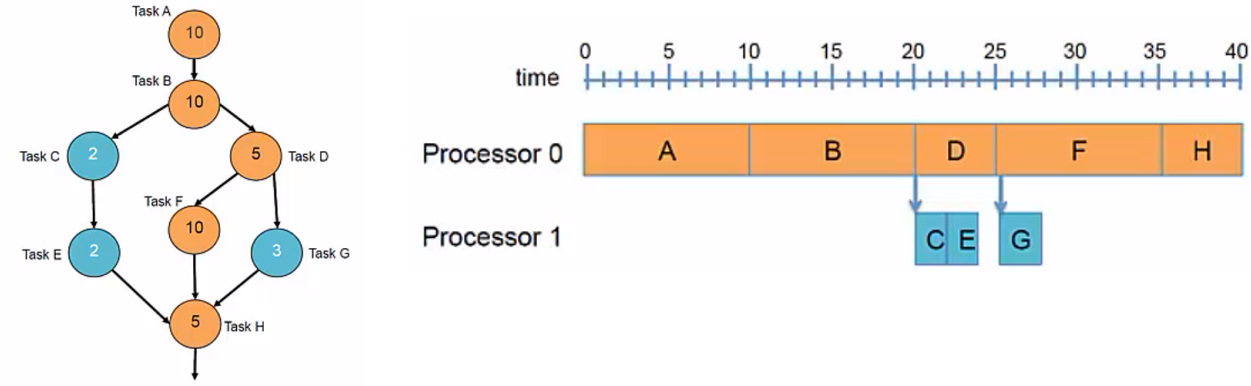
Execution time bounds on P processors

Tp = execution time on P processors
This execution of time depends on the assignment of tasks to processors, which is normally called “Task scheduling”.

Task scheduling can be either:

- Static → Either the compiler or the programer decides who is going to execute each task.
- Dynamic → As soon as a processor is free, it gets the next task that is free of dependencies.

In this example, one processor is assigned with the critical path and the other one with the rest of the nodes.



As we can see, there are some dependencies. Task C does not start until task B is finished. This assignment of tasks to processors, results on a T2 = T_{inf}. Since the rest of the tasks can be executed while the critical path is being executed.

Lower bounds

- Tp >= T1/P. Because overhead… As we have seen before
- Tp >= T_{inf} .Since we can not execute faster than the critical path

Speedup Sp: Relative reduction of the sequential execution time when using P processors.

$$S_p = T_1 \div T_p$$

The speedup with P processors tells us about how fast our parallel program goes when executed with a certain number of processors P.

In our example, with 2 processors:

- T1 = 47
- T2 = 40, which is the length of the critical path

S2 = 47/40 = 1.175 times faster than the sequential execution.

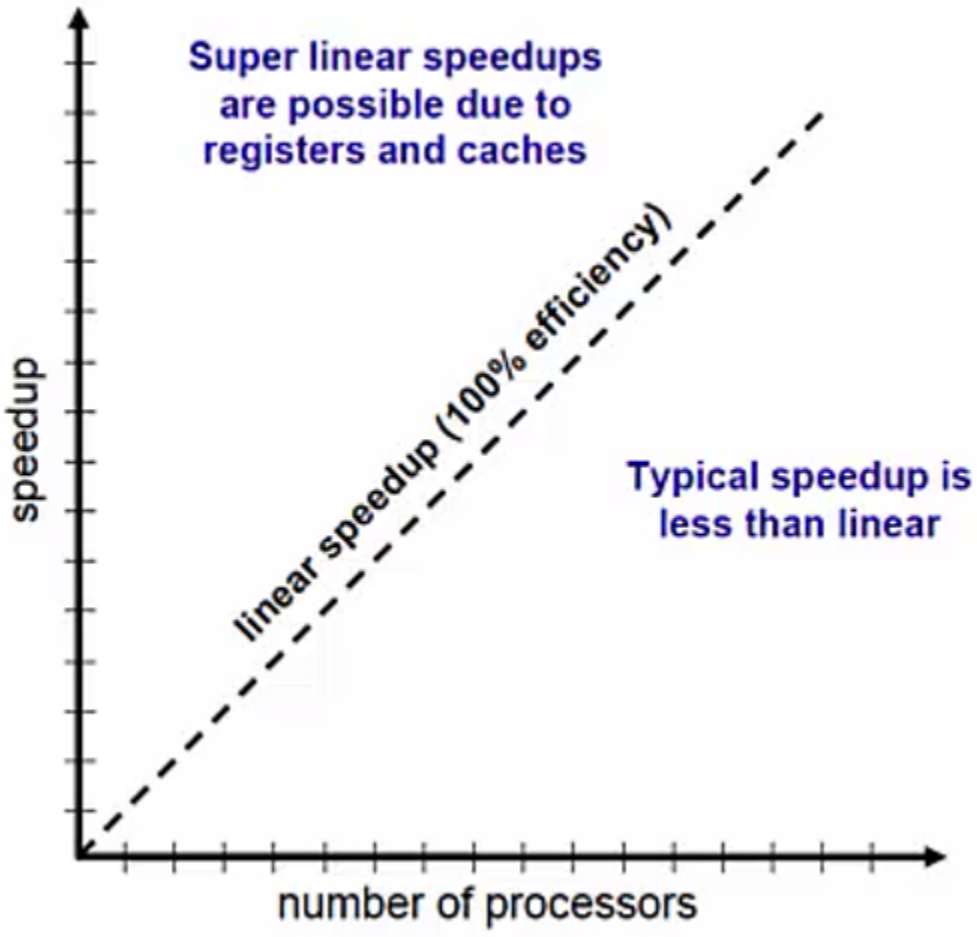
Scalability and efficiency

Scalability: How the speed-up evolves when the number of processors is increased
Efficiency: Tells us how well used are the processors available for parallel execution.

$$E_p = S_p / P$$

This plot shows 3 possible behaviors for the efficiency:

- Linear speedup (100% efficiency), which is the ideal case and it is difficult to achieve → When the speedup = P
- The usual case is to obtain sublinear speedup → Efficiency < 1
- Super linear (efficiency > 1) speedup is possible due to registers and caches. This means that using P processors, the program can run faster than P times.



Strong vs weak scalability

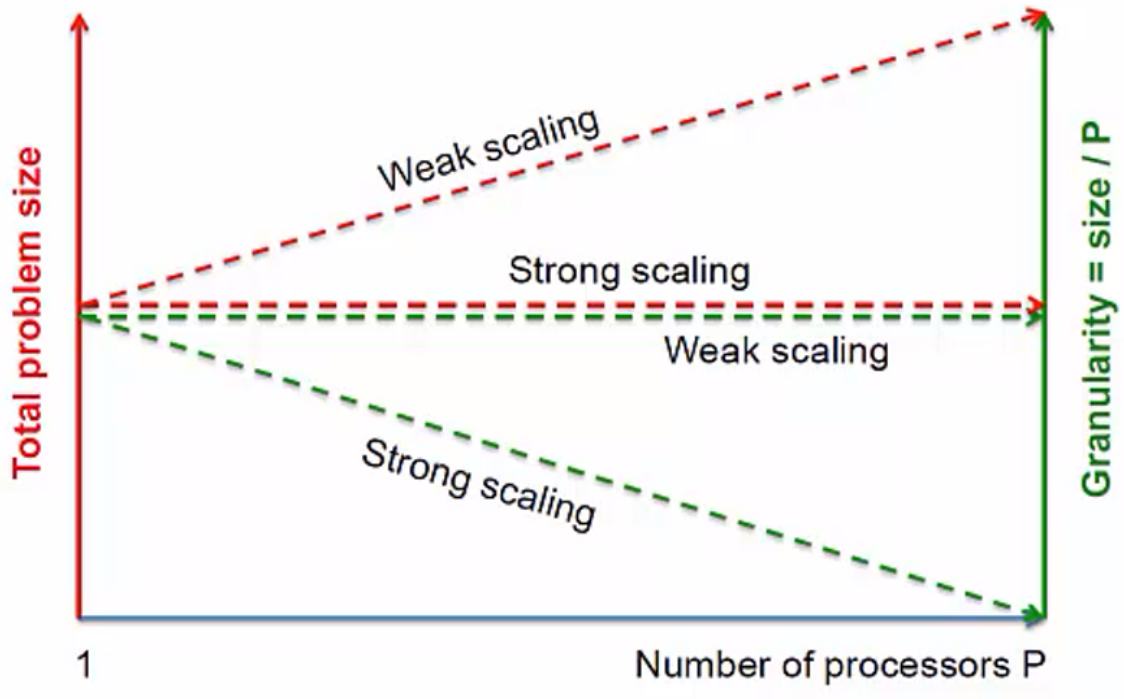
Two usual scenarios to evaluate the scalability of one application:

- Increase the number of processors P with constant problem size (**strong scaling** → reduce the execution time). The purpose of strong scaling is to use parallelism to reduce the execution time of the application.

  On the plot, this is represented with a horizontal red line, which indicates no variation in the total problem size. This implies that the amount of work to be executed per each processor (the granularity of the task decomposition) is finer and finer when the number of processors is increased.
  This is represented by the decreasing green line

- Increase the number of processors P with the problem size proportional to P (**weak scaling** → solve larger problem).

  With weak scaling, the purpose is to use parallelism to execute the application with a larger problem size in the same execution time.
  This is represented with an increasing red line (which indicates the total problem size). This implies that the granularity of the task decomposition is kept constant, which means that all processors execute the same amount of work (as represented by the horizontal green line in the middle).



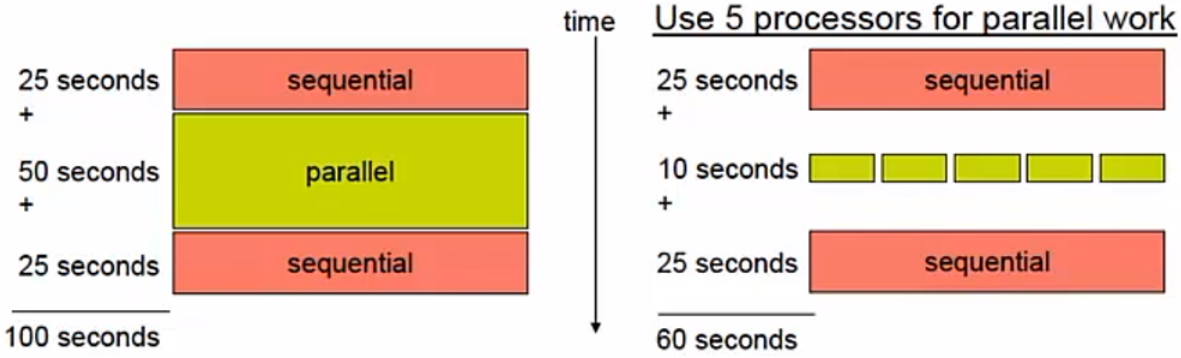
Amdahl’s law

Now we are going to discuss the impact of 2 aspects that may reduce the efficiency of the task decomposition process:

- The first one deals with the serial part in our parallel application. Meaning that parts in the code can not be parallelized.
- The second one deals with possible overheads introduced by the parallel execution.

The discussion in the following lines is based on Amdahl's law which says that the performance improvement of the task decomposition is limited by the fraction of time the program does not run in fully parallel mode, which includes sequential parts and other parts with less parallelism than processors available.

Example:



A sequential program running for 100 seconds with half of the code that can be ideally parallelized.
In the right part of the diagram, we show what would happen if the code is executed using 5 processors.

- The time execution of the sequential portion remains the same
- The time execution of the parallel portion is divided by the number of processors.

Parallel part is 5 times faster

$$Speedup_{parallel\_part} = 50/10 = 5$$

Parallel version is just 1.67 times faster

$$S_p = 100/60 = 1.67 \quad E_p = 1.67/5 = 0.33$$

Let's try to generalize the previous example in order to see the effect of that serial part in our parallel application.
T1 can be decomposed in 2 components:

- T_{seq} which corresponds to the time spent in the parts of the application that can not be parallelized
- T_{par} which corresponds to the time spent in the parts of the application that can be parallelized

$$T_1 = T_{seq} + T_{par}$$

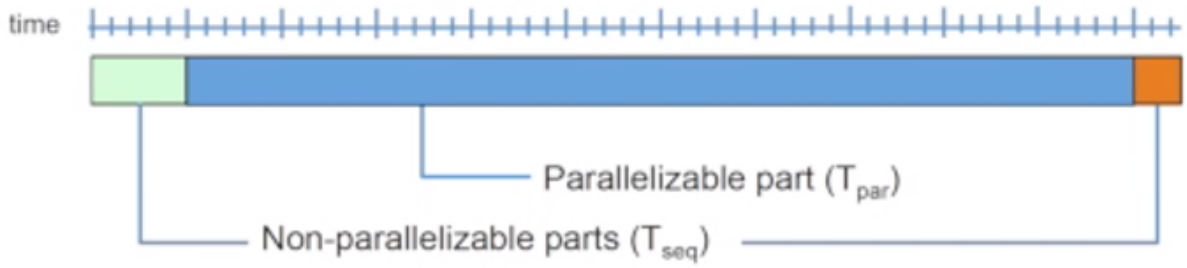
Phi is the fraction of time the program is able to run in parallel.

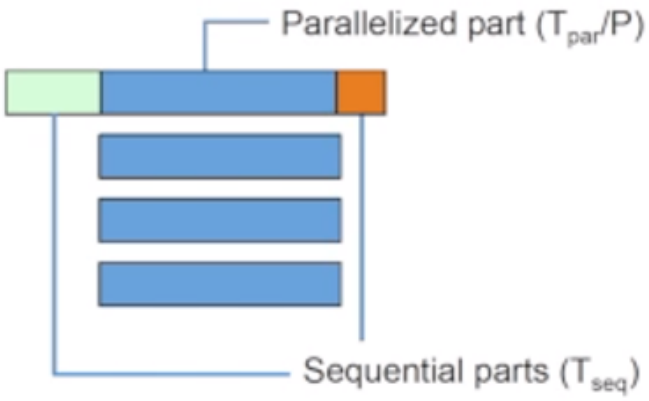
$$\varphi = T_{par}/T_1$$

We can rewrite the expression:

$$T_{seq} = (1 - \varphi) \times T_1$$
$$T_{par} = \varphi \times T_1$$
$$T_1 = (1 - \varphi) \times T_1 + \varphi \times T_1$$

When executed in parallel, only the parallel fraction of the code benefits from getting more than one processor. Ideally dividing its execution time by P.





The non parallelizable part remains the same.

$$T_P = T_{seq} + T_{par}/P$$
$$T_P = (1-\varphi) \times T_1 + (\varphi \times T_1/P)$$

From the 2 expressions for T1 and Tp, we can rewrite the expression for the speedup

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi) \times T_1 + (\varphi \times T_1/P)}$$

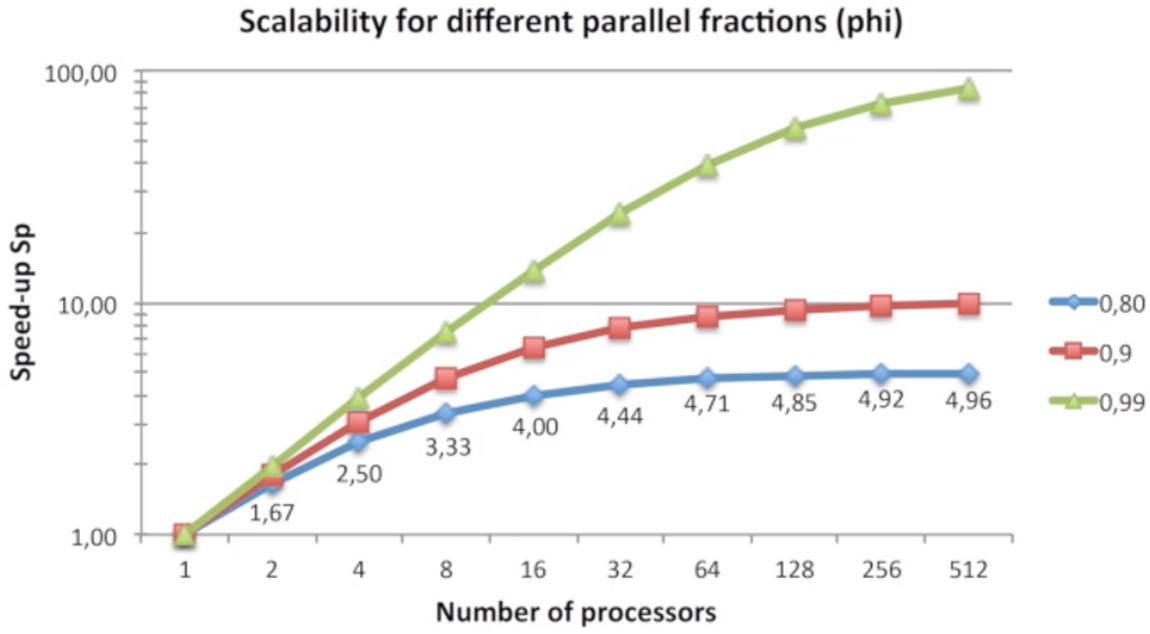
$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$

Two particular cases:

- If Phi = 0, the code has no parallel parts and therefore the speedup = 1
- If Phi = 1, we did a wonderful job parallelizing the application, then the speedup is linear to the number of processors P → Sp = P

Let's see how the expression evolves for other values of Phi.
In this plot, we can see the speedup for different values of Phi and for different numbers of processors:

- Phi = 0.8 (parallelize 80%)
- Phi = 0.9
- Phi = 0.99



For phi = 0.8 and 512 processors, we obtain an Sp of 4.96 → Terribly bad efficiency in terms of processor utilization.

Observe that the curve reaches a plateau of Sp=5 when P tends to infinity.
This plateau can be obtained from the expression in the previous slide when P tends to infinity, obtaining this new simplified expression.

$$S_{p\to\infty} = \frac{1}{(1-\varphi)}$$

For Phi = 0.8, $S_{p\to inf} = 5$

If we manage to parallelize 90% of the application, the maximum speedup that is expected is equal to 10.

In light of the above, the sequential part left on our parallel application will limit the benefits of parallel computing.
If only a certain part can be parallelized, the Amdahl's law gives us the upper limit for the speedup that we can achieve.

Note: We can only use Amdhal's law if all regions can be perfectly parallelized. Divide all tasks perfectly.

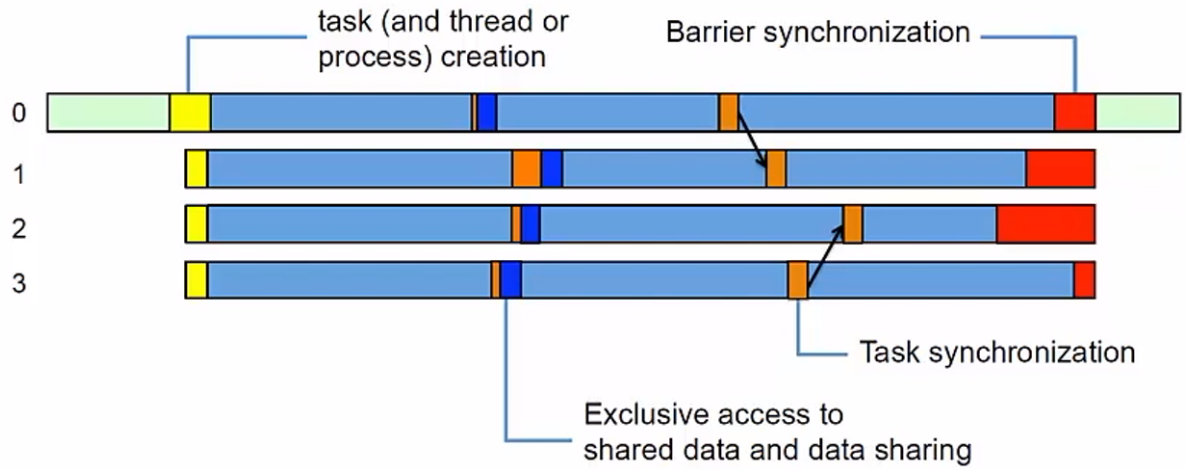
- **Not a task 5 sec, another one 3 sec…**

Sources of Overhead

Parallel computing is not for free, we should account overheads.
So, the serial part of our application is not the only factor that critically influences the scalability of our application.

Overheads represent any cost in time that needs to be added to the parallel computation time in order to effectively run the program in parallel, guaranteeing task ordering and access to shared data.

In this timeline we show some of the more relevant ones:



On the left → The overhead of creating the tasks and, if necessary, the entities offered by the OS and hardware to execute them.

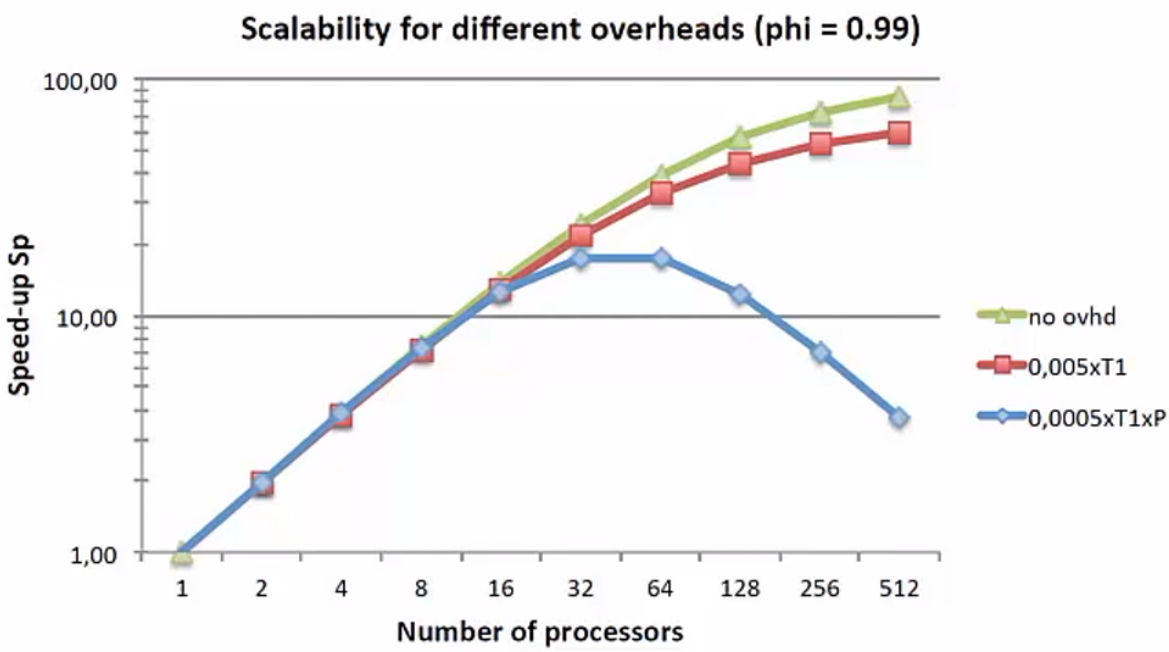
In the middle → The overhead to guarantee that shared objects and verbals are accessed with mutual exclusion, avoiding data raises and the extra latency to memory to access shared data.
Also the overhead to synchronize tasks in order to ensure that the dependencies between tasks are satisfied.

On the right → The synchronization cost that needs to be paid to wait for all tasks to terminate their execution in parallel and continue with the serial part of the application, which can not proceed until all tasks are finished.

How can these overheads influence the scalability that we have formulated with Amdahl's law?
Remember that the time with P processors includes a sequential fraction, which does not benefit from having more than 1 processor and a parallel fraction that does benefit.
We can also add a new component, the overhead, which can be a constant term independent of the number of processors or not (more realistic).

$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead(p)$$

Let's graphically represent the new expression for the speedup, when the parallel fraction is 99%.



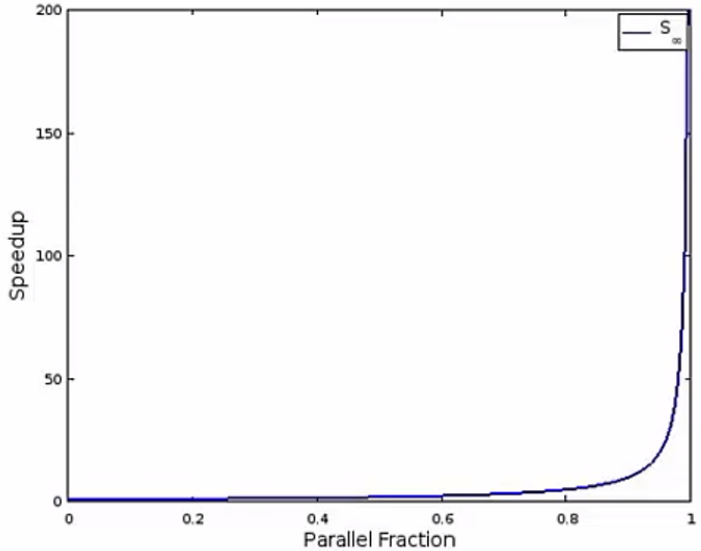
With no overhead, we obtain the green line that approaches the value of 100 when P tends to infinite.

With a constant overhead equivalent to 0.5% of the original sequential time (T1), we obtain the red curve, which is more flattened. This case approaches a speedup of 66, which results from dividing 1 by 0.01+0.005 (when P tends to infinite)

Finally, with a linear overhead of 0.0005 times P, we obtain the blue curve. Which shows that it is possible to have worst speedup when we increase the number of processors. In this case, the maximum speedup is achieved for 32 processors.
After that, the overhead introduced reduces any benefit due to the use of additional processors.

As a conclusion, the figure plots the speedup as a function of the parallel fraction. It is clear that the speedup obtained by the parallelization can only be high for very large parallel fractions.

This can be very disappointing and discourage someone from applying parallel computing.



But, why are there very large parallel systems being built with hundreds of thousands of cores?
The answer falls in the fact that Amdahl's law only applies for cases where the problem size is fixed. It assumes that the computing requirements will stay the same, given increased processing power.

In other words, an analysis of the same data will hopefully take less time given more computing power → Strong scaling

However, in practice, more computing power allows the data to be more carefully and fully analyzed, producing more accurate solutions in the same amount of time.
As more computing resources become available, they tend to get used in larger problems. When the problem size is increased, the parallel portion typically expands faster than the serial portion.

For example, let's consider an increase in problem size in a multiplication of 2 matrices. The initialization of the matrices increases linearly with the size of the matrix (N). However, the actual computation is proportional to N^2, which clearly dominates the total execution time for large numbers of N.

Therefore, the speedup often grows as the problem size is increased and more processors are available.

MPI resources

As an introduction to MPI, we need to be aware that we need to make some use of some primitives in the MPI standard.

We are going to see this with a simple example:

```
#include <mpi.h>
#include <stdio.h>

int rank;
int nproc;

int main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Nothing to do */
    printf("Process: %d out of %d:
            Hello World!\n", rank, nproc);

    MPI_Finalize();
}
```

This code is trying to print “Hello World” and we surround it with several calls to primitives to the MPI standard (all of them start with MPI_).

- We need to start with one to initialize the environment and the last one will finalize it.

In order to compile this code, we use an specific compiler:

```
mpicc -o VSP.x VSP.c
```

mpicc (compiler) followed by the output file and the input file with the code.

Once we have compiled it, we have a binary file that we can execute. But we will try to execute it upfront and we will just see one time “Hello World”.

```
VSP.x
Process: 0 out of 1: Hello World!
```

If we want to use MPI for distributing the money systems in order to do parallel processing, what we need to do is to specify somehow that we want to run this several times.
The way this is done in MPI is through some sort of wrapper that is going to call our program specifying how many processes we want to execute.

The way run the program if we want to do it in parallel is to use:

- mpirun
- Flag that specifies the number of processes “-np”
- Input file

In the first case, we obtain the result when using 2 processors.

```
mpirun -np 2 VSP.x
Process: 0 out of 2: Hello World!
Process: 1 out of 2: Hello World!

mpirun -np 3 VSP.x
Process: 1 out of 3: Hello World!
Process: 2 out of 3: Hello World!
Process: 0 out of 3: Hello World!
```

As we can see, we do not get any particular order of execution.

How do we get this information about the number of processes and the unique identifier for each of the processes?
In the code we can see that within MPI_Init and MPI_Finalize we have 2 other calls:

- **MPI_Comm_size** → Is going to return in the second parameter the total number of processes that have been created by the time we are launching the execution of this program.
- **MPI_Comm_rank** → Is going to return a unique identifier within the set of processes that we are using. So, if we have 2 processes, the identifiers will be 0 and 1.

MPI_Init

We need to provide the parameters argc_ptr and argv_ptr[]

MPI_Comm_size

```
int MPI_Comm_size( MPI_Comm comm, /* in */
                   int* size );   /* out */
```

The first parameter is a parameter type of MPI_comm, which stands for “communicator”. This will be a communication channel that MPI establishes for all the processes so that they can communicate.
At some point we can be interested in using some communication media or another, so this is a variable that can be defined at some point.

MPI_COMM_WORLD is created by default and all processes can share it. With this communicator, all the processes that we have created will share this communication channel.

We are interested in retrieving the total number of processes, so we will provide the address of an integer variable in which we are going to retrieve that value.

MPI_Comm_rank

```
int MPI_Comm_rank ( MPI_Comm comm,    /* in */
                    int* rank );  /* out */
```

We also use the communicator and we will retrieve the identifier of the process.

MPI_Abort

```
int MPI_Abort( MPI_Comm comm,  /* in */
               int errorcode );   /* in */
```

In case we have trouble, we can use MPI_Abort to force all processes of an MPI job to terminate.

Here we have another very simple example:

```
#include        <stdio.h>
#include        <mpi.h>

int main(int argc, char *argv[])
{
  int rank;

  /* Initialize MPI */
  MPI_Init(&argc, &argv);

  /* Find out my rank in the global communicator MPI_COMM_WORLD*/
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* Insert code to do conditional work if my rank is 0 */
  if(rank == 0){
    printf("Hello, World!!! (from masternode)\n");
  } else{
    /* Insert code to print the output message "Hello, World!"*/
    printf("Hello, World! (from worker node)\n");
  }
  /* Exit and finalize MPI */
  MPI_Finalize();

}/* End Main */
```

We also want to print “Hello World”. But in some cases we are interested in distinguishing what different processes have to do.
Depending on the rank, we address the processor to one part of the code or to another.

- Process with identifier 0 to print “Hello, World!!!” from the masternode.
- Other processes to print “Hello, World!” from the worker nodes.

The output of the execution will be:
With 3 processes, only one is printing “Hello, World!!!” and the other 2 are printing “Hello, World!”

As we can see again, there is no specified order.

Compilation

```
mpicc -o VSPS.x VSPS.c
```

Execution

```
mpirun -np 3 VSPS.x
Hello, World!!! (from masternode)
Hello, World! (from worker node)
Hello, World! (from worker node)

mpirun -np 3 VSPS.x
Hello, World! (from worker node)
Hello, World! (from worker node)
Hello, World!!! (from masternode)
```

This brings us to the need for synchronization.
Using **MPI_BARRIER** we can specify that we need to wait for all the processes to reach that point before they can all continue.

```
int main( int argc, char* argv[] ) {
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    // Ensure that all processes arrive
    // here before timing
    MPI_Barrier(MPI_COMM_WORLD);
    /* Something to do in local */

    MPI_Finalize();
}
```

Point-to-point communication

Let's try to do some useful work by sending information from one process to another. We will start with point to point communication

There are many flavors for communications in MPI.
Hints: When we communicate with someone:

- We can do a phone call
- We can drop a message, we do not wait for it to arrive, we just send it.
- We can receive a message

Blocking communications → We have all processes trying to establish a communication. They are waiting for that communication to happen.

Non-blocking communications → Processes request the communication while they continue to do other work.

We have the following image:

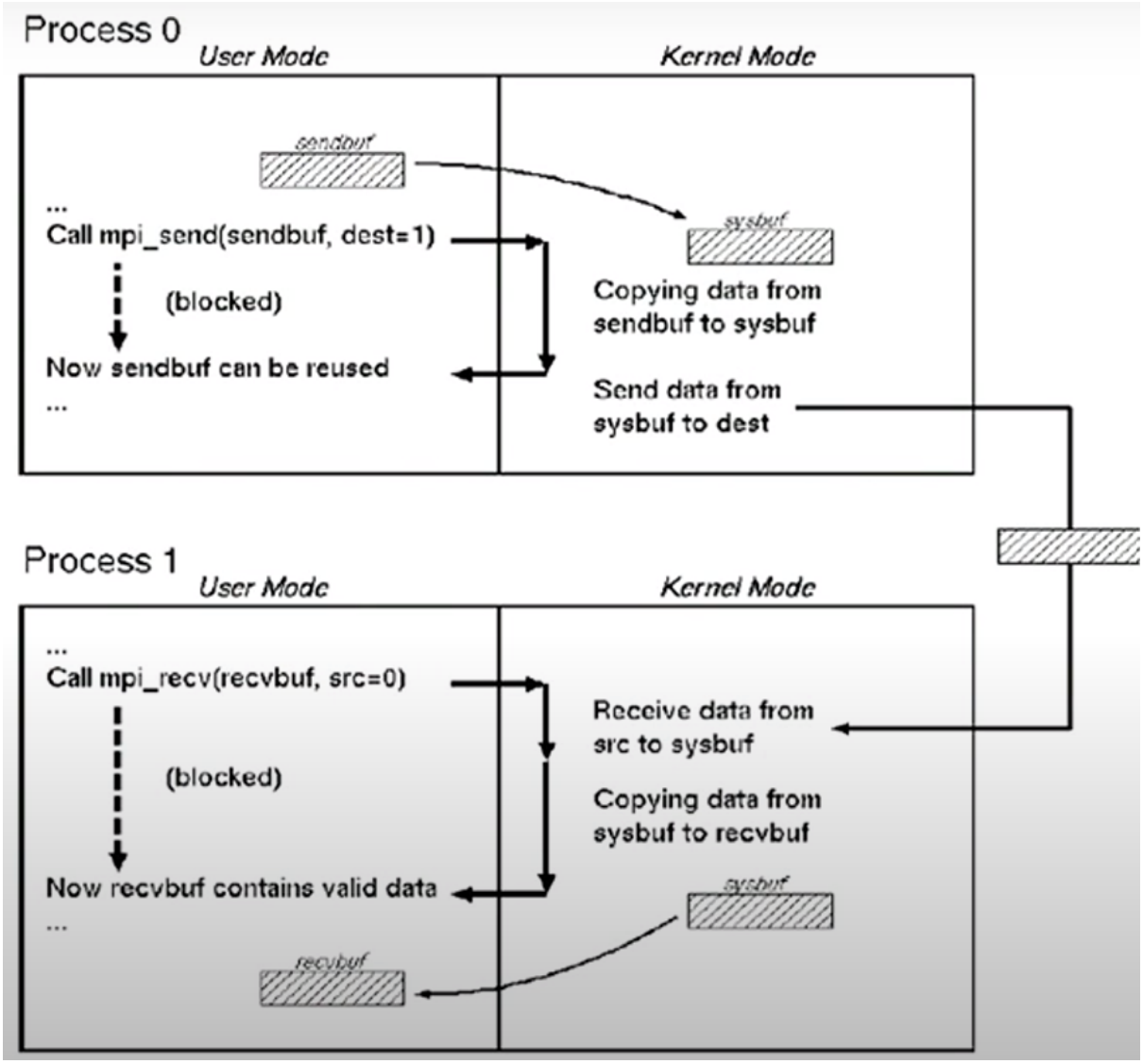
- The process above wants to send a message to the process below

Process 0 has some information in memory (in a buffer). By using primitives, it is going to try to send this message to another process.

To send information we need to specify which is the address in memory where the information is and who is the destination.
This will go into MPI that will try to send this information for us.

- If the other process is in a distant system, we can connect through a fast speed network. Eventually, the other process will receive the message and copy this information somewhere in memory in the destination.

  For this, the process at the destination point will need to specify that it is interested in receiving information and this information needs to be left in some place in memory.



The communication will be blocking if we are sending the call into MPI_send (primitive used for blocking communications) or receiving the call into MPI_recv.
Meaning that they will get stuck until the communication takes place (they are blocked until the communication takes place).

Here we have the correct MPI syntax

```
#include "mpi.h"

int rank, nproc;

int main( int argc, char* argv[] ) {
    int isbuf, irbuf;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if(rank == 0) {
        isbuf = 9;
        MPI_Send( &isbuf, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
    } else if(rank == 1) {
        MPI_Recv( &irbuf, 1, MPI_INT, 0, 1, MPI_COMM_WORLD,
                              &status);
        printf( "%d\n", irbuf );
    }
    MPI_Finalize();
```

In **MPI_Send** we need to specify:

- The address in memory
- The number of elements
- The data type of those elements
- The process to which we are going to send (the process from which we want to receive in **MPI_Recv**)
- A label (needs to be the same in the call of the Send and Receive) to distinguish between different messages (maybe one has higher priority).
- The communicator.

With this communication example, the sending in process 0 will not end until the reception is done in process 1.
Reception will not return until it receives the message that has been sent from process 0.
This implies a synchronization between these 2 processes.

After this happens, the destination can use the information that was previously only known by the sender.

With this scheme, the 2 processes were just engaging the communication, they could not do anything else. So, in a distributed memory system, communications are very expensive (they can take forever compared to the speed of the CPU).
Thus, it could be interesting to use another flavor of communication that allows us to avoid blocking.
The idea would be to request a communication. We still make a call from MPI, we specify what we want to send, the destination… but we do not wait for this communication to happen, we just request it.

The primitive for doing that will be adding an “i” (stands for immediate) in front of send and receive. This means that we do not want the call to block until the communication has finished.
The library will write what needs to be done and as soon as the information is kept safe, it will return control to the process and it will return an identifier specifying that this is a particular input-output communication request.

Obviously, the process sending the information can not be sure if this communication took place, so it has to keep the message in the input buffer (so that we do not lose that information). Meaning that we can not overwrite this message until we are sure that it has been received.

When we try to receive, we can specify that we are interested in the reception of some information but we are not willing to wait for it (getting blocked).

As soon as the library writes the information internally about what needs to be done, it will return control to the process and the process can continue doing other things.
If this process specifies a need for a reception, at some point it will be interested in using that information. This will require some future synchronization.
For this, there are some other calls that allow us to specify that given a communication request, we want to synchronize with that request and we want to make sure that it has finished.

For this, we have MPI_Wait and MPI_Test:

- **MPI_Wait:** We wait for that communication to finish. If it has already finished, it will return quickly specifying the status of the communication.
  With this status we can reclude information about the number of elements that were received, who was sending the information, which was the tag…

- **MPI_Test:** If we are interested in checking whether the information arrived or not, but we do not want to block, then we use MPI_Test.

With this we can build programs that specify the communication but do not block while the communication takes place. This allows the overlap of computations and communications

Here we have modified our previous example that was trying to communicate 2 processes, sending from 0 and receiving from 1.

- Instead of using MPI_Send we use MPI_Isend
- We add a last parameter (in red), which will return the identifier for this sending request.

```
int main( int argc, char* argv[] )
{
    int rank, nproc;
    int isbuf, irbuf, count;
    MPI_Request request;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    if(rank == 0) {
        isbuf = 9;
        MPI_Isend( &isbuf, 1, MPI_INT, 1, 1,
                   MPI_COMM_WORLD, &request );
```

For the reception:

- Instead of using MPI_Recv we use MPI_Irecv
- We add a last parameter (in red), which will return the identifier for this request.

After the synchronization with MPI_Wait, we would be able to use that information. But, in between this reception, we could be doing other work (**this is the advantage of using the Non-Blocking communications**).

```
    } else if(rank == 1) {
        MPI_Irecv( &irbuf, 1, MPI_INT, MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &request);
    /* OTHER WORK TO DO */
        MPI_Wait(&request, &status);
        MPI_Get_count(status, MPI_INT, &count);
        printf( "irbuf = %d source = %d tag = %d
                 count = %d\n", irbuf, status.MPI_SOURCE,
                 status.MPI_TAG, count);
    }
    MPI_Finalize();
}
```

Hands On

We are going to use some code that illustrates a potential “deadlock” if we do not take care of our communications.

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
#define MSGLEN 2048
  int ITAG_A = 100,ITAG_B = 200;
  int irank, i, idest, isrc, istag, iretag;
  float rmsg1[MSGLEN];
  float rmsg2[MSGLEN];
  MPI_Status recv_status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);

  for (i = 1; i <= MSGLEN; i++)
    {
      rmsg1[i] = 100;
      rmsg2[i] = -100;
    }
```

```c
  if ( irank == 0 )
    {
      idest  = 1;
      isrc   = 1;
      istag  = ITAG_A;
      iretag = ITAG_B;
    }
  else if ( irank == 1 )
    {
      idest  = 0;
      isrc   = 0;
      istag  = ITAG_B;
      iretag = ITAG_A;
    }
```

Here we have the initialization of some variables that we will use in the MPI_Send and MPI_Recv.

```
  printf("Task %d has sent the message\n", irank);
  MPI_Send(&rmsg1, MSGLEN, MPI_FLOAT, idest, istag, MPI_COMM_WORLD);
  MPI_Recv(&rmsg2, MSGLEN, MPI_FLOAT, isrc, iretag, MPI_COMM_WORLD, &recv_status
);
  printf("Task %d has received the message\n", irank);
  MPI_Finalize();
}
```

Here we just print 2 messages:

- The message has been sent
- The message has been received

We execute the program and we obtain the first message but then the program gets stuck and we do not obtain the second message.

The problem here is that both processes are making a call to MPI_Send (both of them are trying to call each other). Since MPI_Send is blocking, the calls will only return when the other processes make the reception. But the reception will never happen because the other process is also trying to send.

Two things can be done here to solve this:

- Go to the If and ELSE IF and say: If I am process 0 I force send and if I am process 1 I force receive.
  So, by using a different order for sending and receiving in process 0 and process 1: Only one of them should send first while the other should receive first.

- Use Non-blocking communication: MPI_Isend (add the identifier parameter at the end). Not MPI_Irecv, since only the send is blocking.
  - We will need to add an MPI_Wait after the last print.

Now the code is perfect.
Both processes try to send and move quickly to reception.
At some point the communication arrives and then we print the last message.
Note that the reception is blocking. Thus, it will only write the last message when the information has been received.

Collective communication

Another possibility in MPI is to express communications that take place within a set of processes → Collective Communications

If we switch on the TV or radio, we will receive information that is being broadcasted to anybody that is being tuned in that channel.

With collective communications, we have a set of primitives that involve several processes (not just 2, as in point to point communications we have seen before).

- **MPI_Bcast** → In Broadcasting, we will be sending information from 1 process to all the other processes that are involved in that communication.

  Parameters:
  - Address in memory of the buffer
  - Number of elements
  - Data type
  - **Root** (is of the process that sends the information)
  - Communicator

  Here we do not distinguish the code that is doing one process and the code that the other processes are doing. So, even if we are broadcasting (one process sends information to all the other), all processes will make the same call.
  This is a particularity of collective communication.

```
#include <mpi.h>
void main (int argc, char *argv[])
{
   int i, my_id, numprocs, num_steps;
   double x, pi, step, sum = 0.0 ;
   MPI_Init(&argc, &argv) ;
   MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
   if (my_id==0) scanf("%d",&num_steps);
   MPI_Bcast(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD)
   step = 1.0/(double) num_steps ;
   my_steps = num_steps/numprocs ;
   for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++){
   x = (i+0.5)*step;
   sum += 4.0/(1.0+x*x);
   }
   sum *= step ;
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE,
      MPI_SUM, 0, MPI_COMM_WORLD) ;
   MPI_Finalize() ;
```

This code computes the number Pi.
We enclose this loop within some MPI primitives.

So, all processes are going to contribute to the computation of Pi.

We need to know how many processes are going to be involved, which is the ID for each of them and, depending on that ID and the total number of processes they are going to split the workload.
The workload comes from the execution of a loop, so each processor will compute a fraction (some iterations) of that loop.

In this example here, we are going to start the loop with a position that will depend on the identifier of the process and then we will be traversing several consecutive iterations from there until we reach the end of the part given to each process.

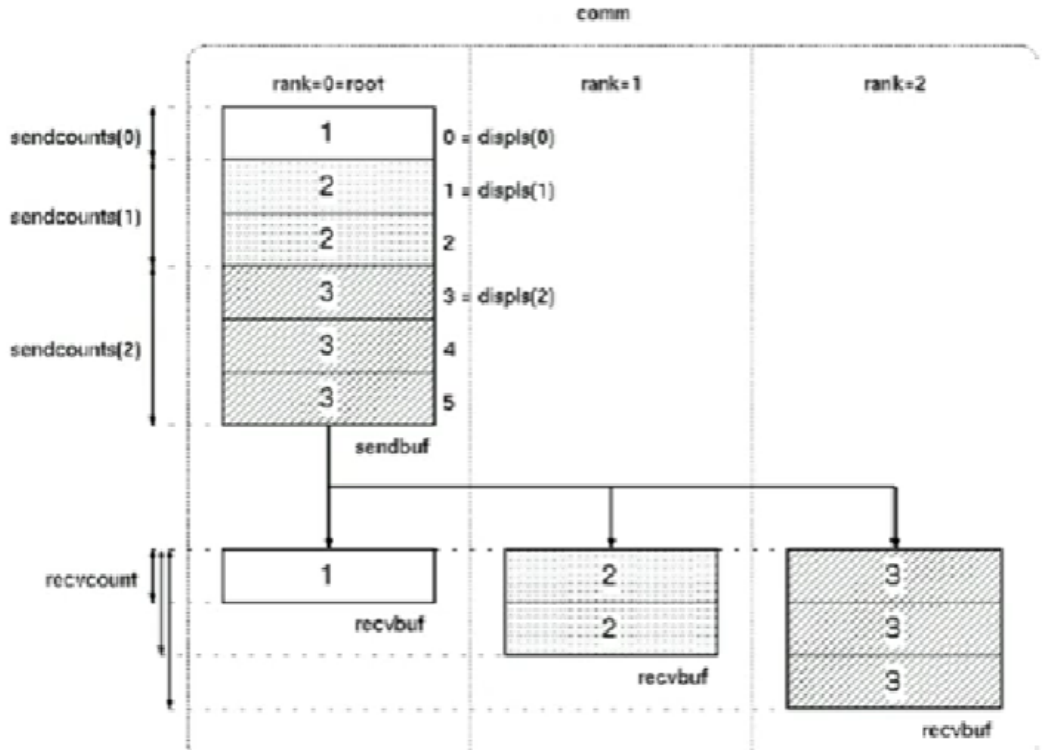
Process 0 is sending some information that we force it to receive from the input (the *scanf* that is before the MPI_Bcast). When this information gets transferred, all the processes start computing and they will come up with some partial result.
All these operations take part independently, so at the end we need to put in common the partial results.

Contributing with several values and eventually using some operation (an addition in this case) to produce a single value, this is a common pattern that is known as a reduction.
MPI has support for reductions → MPI_Reduce. We need to specify the address in memory that will hold the result, the address in memory that holds the partial result, the number of elements that we need to transfer, datatype (operation +), the process that is going to receive the final result and the communicator.

- **MPI_Reduce** → N to 1 (explained before)

- **MPI_Scatter** → In some cases, we want each process to work with some part of the data. Idea of playing cards, each player has some cards. 1 to N

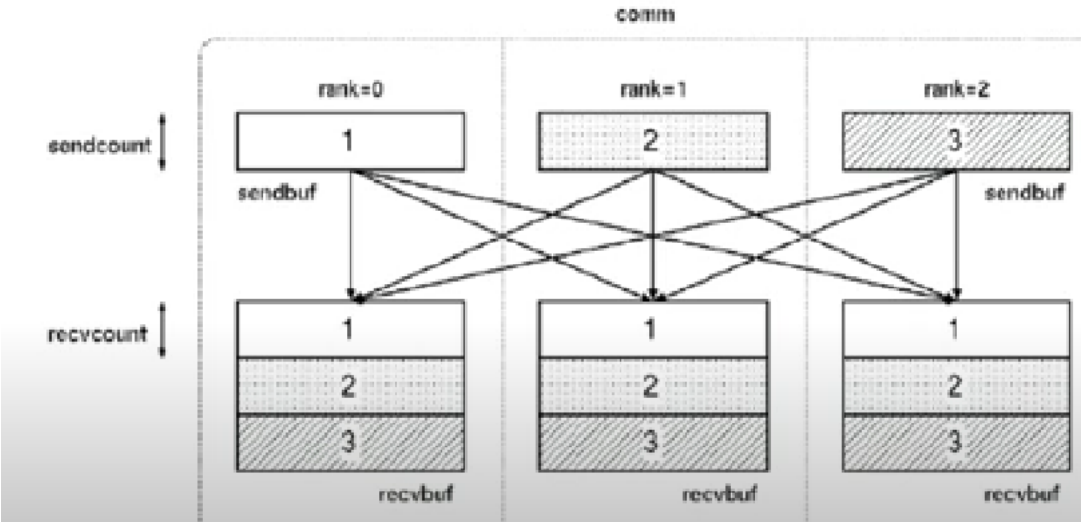
- **MPI_Scatterv** → Sometimes the players want different number of cards. This is useful for heterogeneous systems. The V is for Variable number of elements



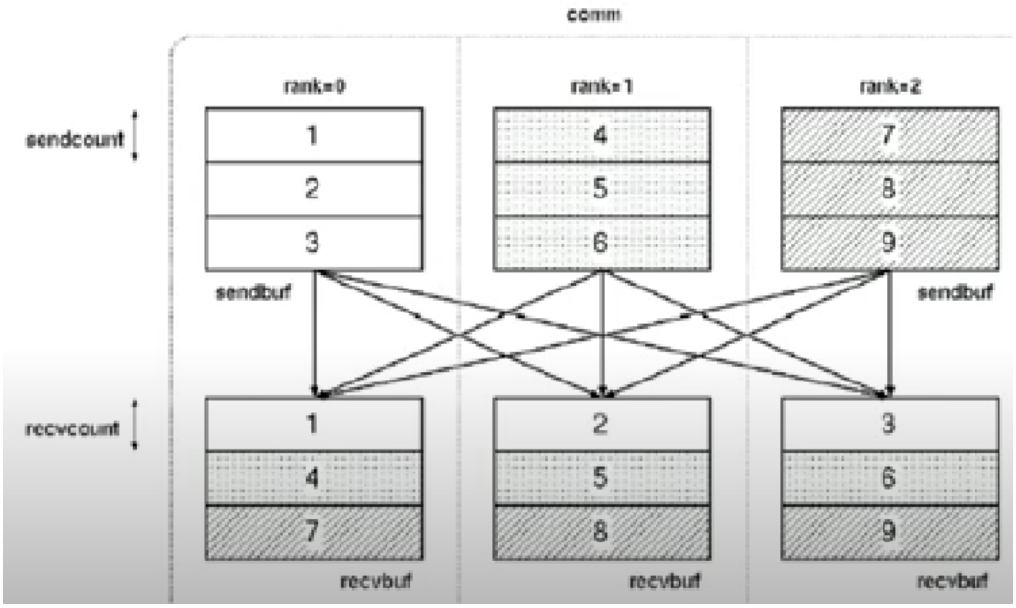
- **MPI_Gather** → To recover all the cards in a single place at the end of the game. So, it is the opposite of the scatter. N to 1

- **MPI_Gatherv** → Gather information from variable number of elements into a single process.

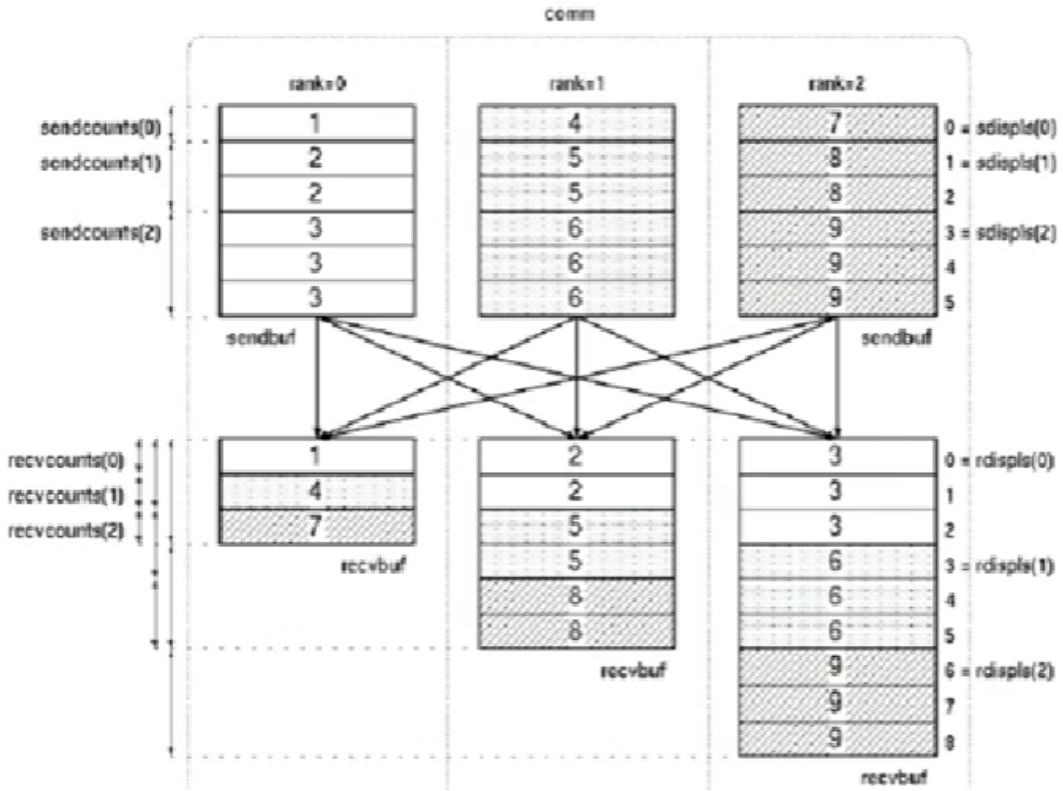
- **MPI_Allgather** → This is a combination of gather and scatter. Here the information is replicated in all nodes. So, at the end all nodes gather all the information.
  It's like a MPI_gather followed by a MPI_broadcast.



- **MPI_Alltoall** → It is a type of scatter of all nodes. Useful for doing the transposition of a matrix.



- **MPI_Alltoallv** → We can also specify the number of values each process receives.

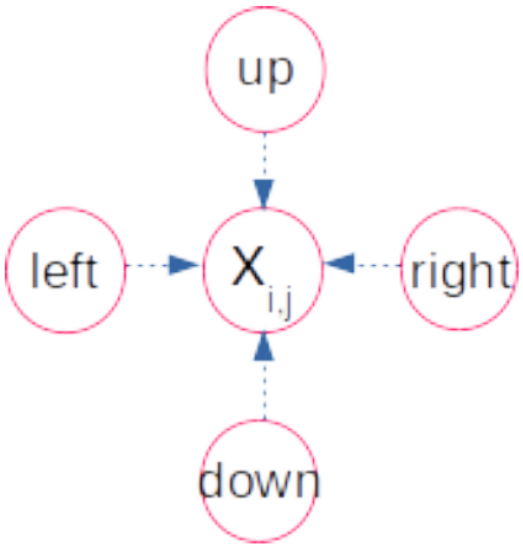


Gauss-Seidel: Characteristics

In the Gauss-Seidel method, the code only uses a matrix (input & output) that is going to be modified during the execution of the loops.
Contrary to Jacobi, in which we had 2 matrices:

- Matrix for input
- Matrix for output

Gauss-Seidel also uses a 5 point stencil as Jacobi:



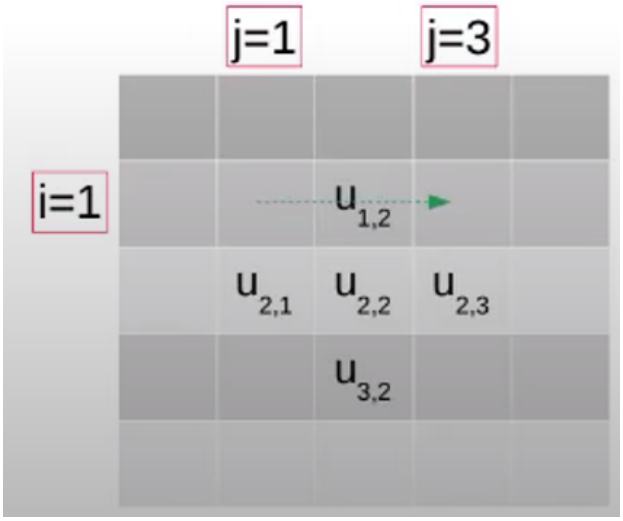
The difference is that the matrix output matrix will be the same as the input matrix (as we said before). Thus, we will need to keep an eye on the original sequential order and see how the elements in the matrix are being modified.

As we can see, there are 2 loops that start at 1.

```
void compute( int n, double *u) {
  int i, j;
  double tmp;

  for ( i = 1; i < n-1; i++ ) {
     for ( j = 1; j < n-1; j++ ) {
        tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
              u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
              4 * u[n*i + j];                    // element u[i][j]
        u[n*i + j] = tmp/4;
     }
   }
}
```

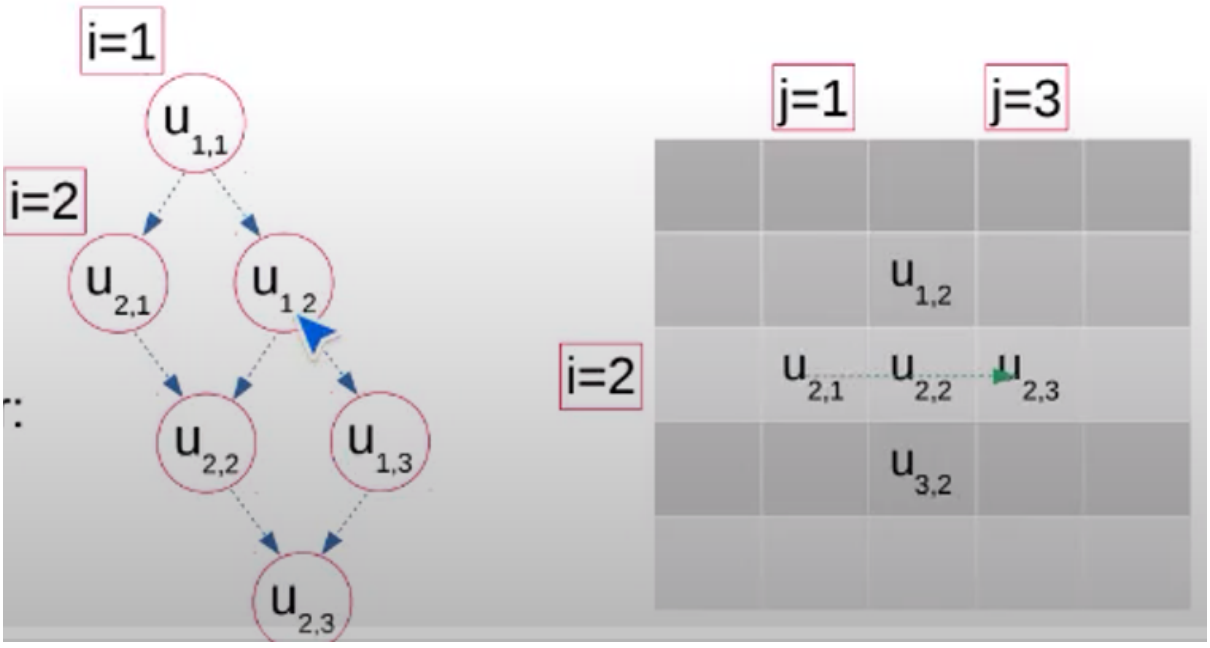
Thus, we will start adding values to the matrix in position [1, 1], then [1,2]... following the green arrow.



Note that to compute element [1,2], he needs the element to his left, which is element [1,1]. So, for each element we will need the element that was computed in the previous iteration of the inner loop.

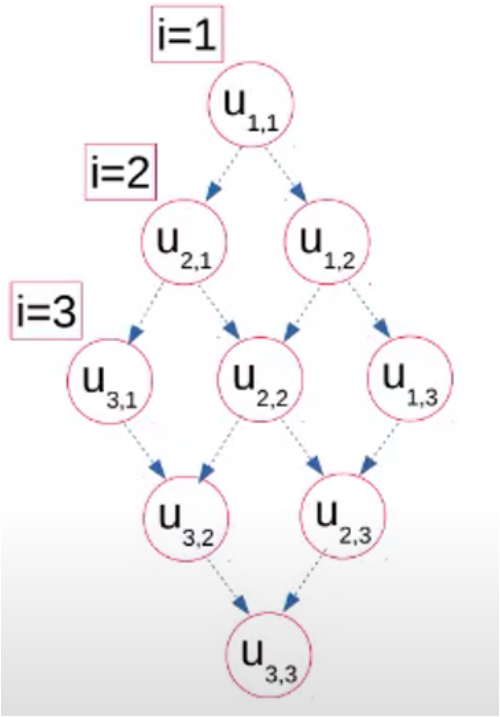
- There are loop carried dependencies

When we make the next iteration of the outer loop we will move to the third row. In this case, element [2,1] will need the element above.
Thus, we need something that was computed in the previous iteration of the outer loop when j = value we have now (1 in this case).



If we want to compute element [2,2], we will need the element above [1,2] and the element on its left [2,1].

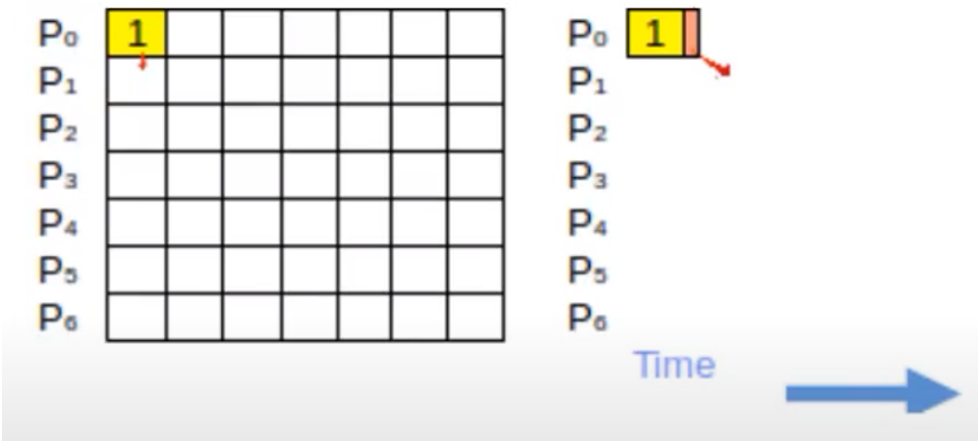
If we continue all the way to the 3x3 matrix (not considering the Halo), we obtain the following Task Dependence Graph:



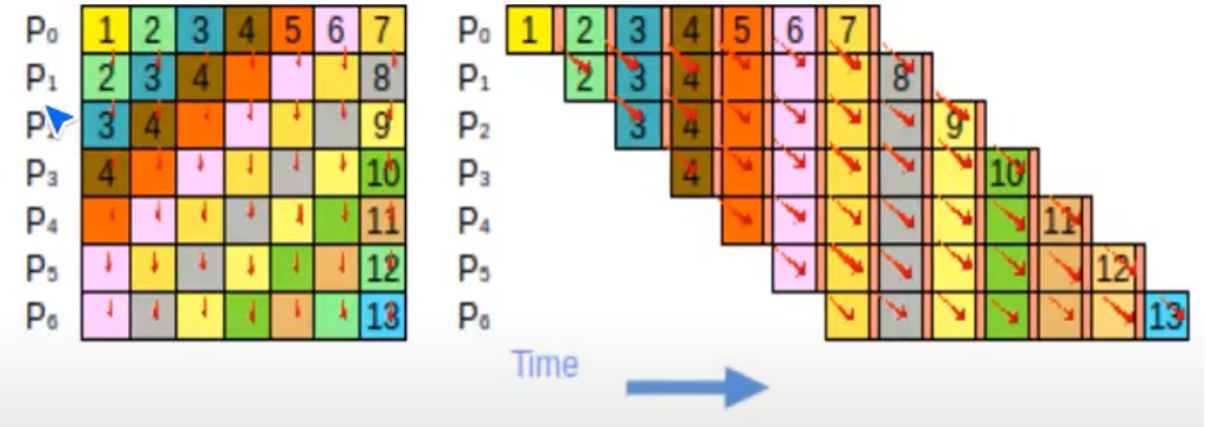
Each task is computing an individual element.

The opportunity that this TDG gives us for parallel execution is:

- At the beginning we can only compute element [1,1]. Thus, we would only need 1 processor.
  Once processor 0 has done the first task, it needs to inform the tasks that compute elements [2,1] and [1,2] that they can start because we have computed the element [1,1]. Thus, we must add a synchronization/communication step.



- Then we can execute in parallel elements [2,1] and [1,2]. Then we will have another synchronization/communication step.
- This pattern will continue until the end of the matrix.



As we can see, at each step we are increasing the number of processors that we are using.

- This is called Wavefront Parallelism

In Jacobi all elements could be computed in parallel, but in this case we can't because we need to respect the dependencies.

In a sequential execution, it would take 49 computation steps.
Using parallelism we only need 13 computation steps. But we also add some overhead.

Gauss-Seidel. Data sharing and movement

As we did with Jacovi, we are going to give a consecutive set of rows or segments to each processor.
For the segments that are in the middle, they will need data that belongs to other processors.

- Upper and lower boundaries

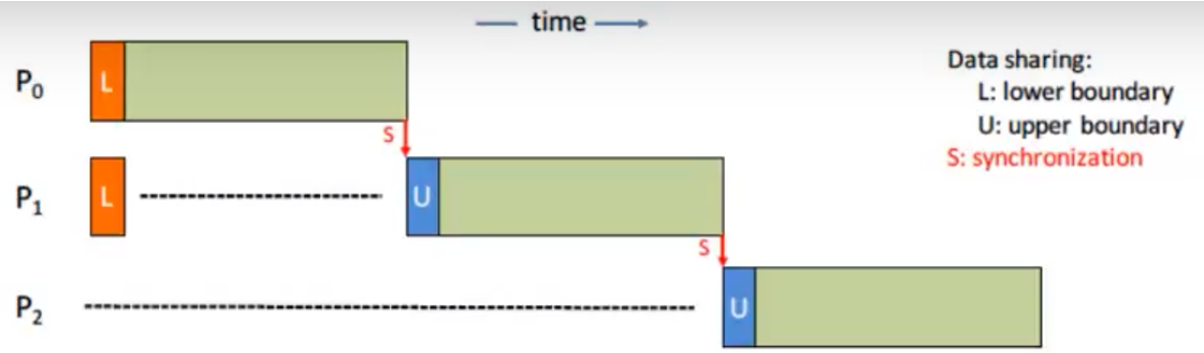
For this, we will need to exchange messages if we have a distributed memory machine.

We are going to assume that each processor will start by computing a task that consists in working with its own segment (as we did for Jacovi). Therefore, the inner loop will be the same (from 1 till the end) but the outer loop, that corresponds to the rows, will be modified so that we can specify the segment that each processor will work with (number of rows / number of Processors).

So, each processor will do a task that consists in computing n/P consecutive rows.

The last row is only available at the end. This last row of processor “i” will be the upper boundary for processor “i+1”.

- This implies a sequential execution.



As we can see, we can do some communications at the beginning, which correspond to the lower boundaries.

- The data in the lower boundary is not modified.

Even though we know that this is a sequential execution, we want to model the parallel time as an exercise.
As we have seen, we have processes that do more things (communications L and U) and therefore we will need to account for the worst case.

- The longest task will be any task with no rows from the border (need information from U and from L). Meaning that they need information from the process before and after.

So, which is the time for this overhead using the model for communications that we developed in the course with some start time (Ts) + some time that has to do with the length of the message.

- First we need to model the time for the lower boundary. It is a whole row “n”.

  Ts + n·Tw

  We count this only once because they can all happen in parallel (in the diagram we can see that they are VERTICAL).
- For the Upper boundaries, which will also communicate a whole row (**Ts + n·Tw**), we need to multiply by the number of processors -1. We do this because in this case they can not happen in parallel, we must do them sequentially.

$$T_{overhead} = \{ \text{lower boundary at the beginning} + \text{upper boundary during execution} \}$$

$$T_{overhead} = (t_s + n \times t_w) + (t_s + n \times t_w) \times (P-1)$$

For the computations:

- Each task is computing n/P consecutive rows.
- Each row has “n” columns

Thus, the total number of elements will be: **n · n/P**

The cost of computing each element is **Tbody**.

Thus, (n^2 · Tbody) / P will be the cost for computing one of such tasks in any of the processors.

Since we can not overlap any time (it is sequential), we must multiply it by **P**.

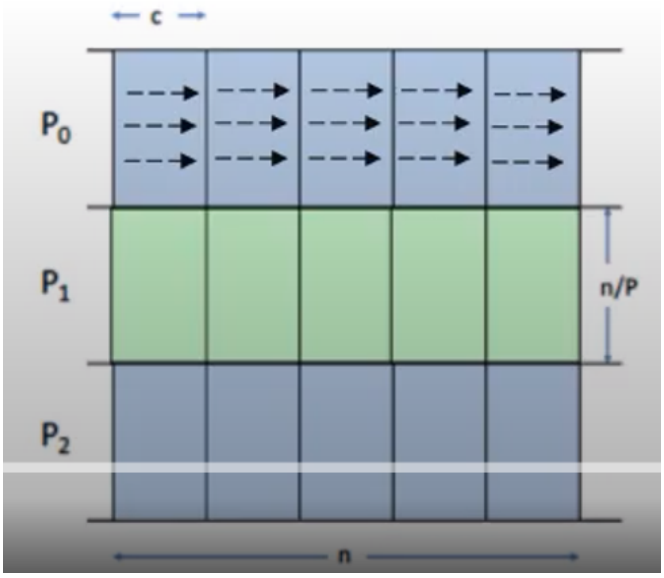
$$T_{calc} = P \times (n \times n/P) \times t_{body} = n^2 \times t_{body}$$

This computation time is exactly the same as if we use a single processor.
In fact, it is worse than sequential execution because we have the communication overheads.

Gauss-Seidel. Finer Grain parallelization blocking by c columns

We want to execute the code in parallel, but we have seen that using the strategy mentioned before, we will not obtain any benefit, since we are sequentializing the execution.

Thus, we need another strategy. We are going to use the idea of blocking, which consists in splitting the loop that traverses the columns.
Once we arrive at the end of this blocking factor, instead of continuing to the right we will start operating in the next row.

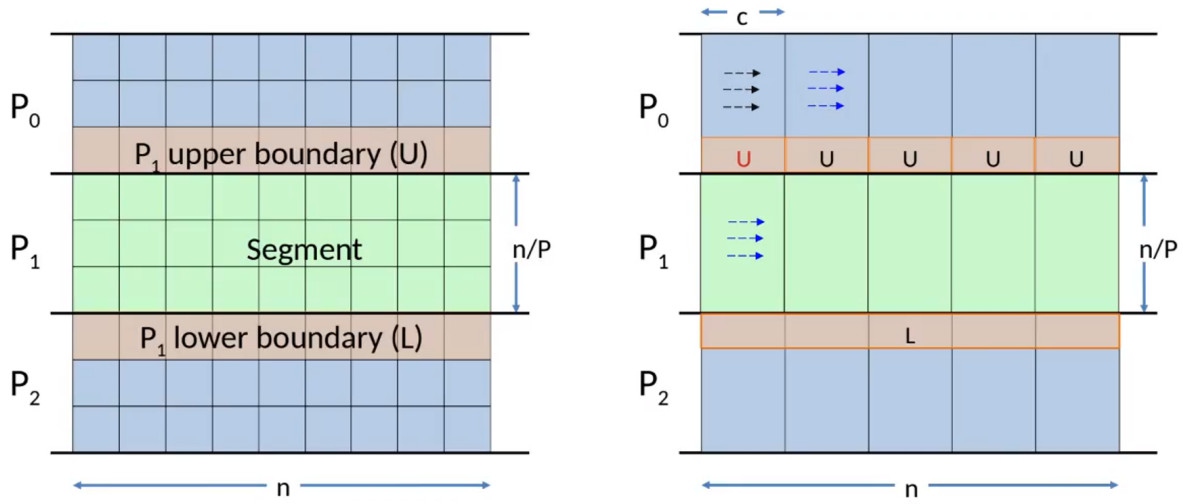


So, we have a new task definition:

- Each task is going to compute N/P rows but instead of computing the whole row, it will compute only C columns.

By doing this, we will have several tasks operating in parallel.

With this new strategy, by the end of the execution of the first task, we will have a part of the upper boundary ready (unlike the previous strategy in which each task was computing the whole row).

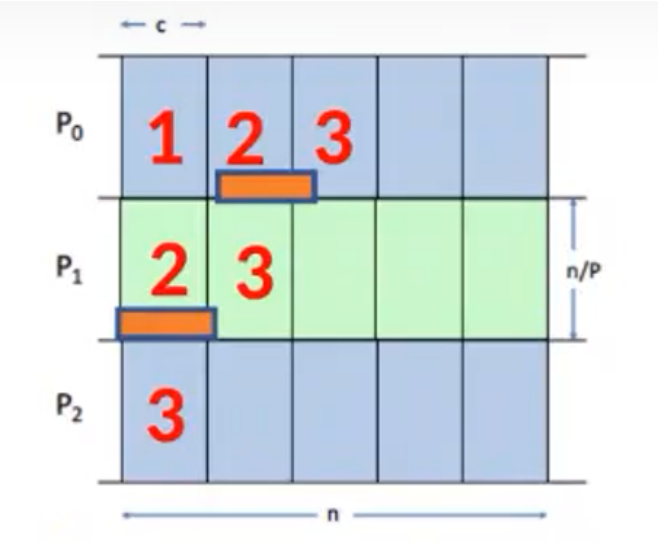


Thus, P1 can get the Upper boundary as soon as P0 finishes the first task.
Thus, P1 will be able to start this first task in parallel with the second task of P0.

As mentioned, the Lower boundary will be sent at the very beginning with a single message of length N.
The Upper boundary will be sent once the process finishes its task (length C).

Pseudo-Algorithm

1. Exchange information of the lower boundary (all processes -1 in parallel).
2. P0 starts with the first task
3. Wait for termination of task computing the same block in previous processor, if any (dependence).
4. Access to C elements for upper boundary (if needed)
5. Apply stencil algorithm to the block → P1 starts working with task 1 and P0 starts with task 2.
6. Repeat tasks 3, 4 and 5 until the end.



Wavefront parallelism!

Gauss-Seidel. Parallel execution timeline with blocking

We want to get an expression for the parallel execution time of the Gauss-Seidel method once the blocking factor in the calls has been applied.

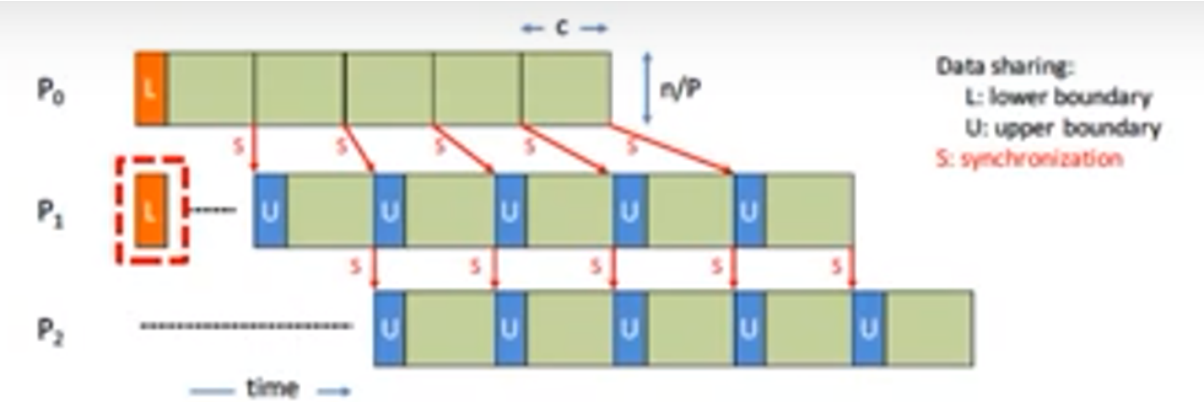
We will separate the execution time in 2 parts:

- Overheads due to synchronizations
- Computation time

The overhead of sharing the data (before and during parallel computation):

- The lower boundary can still be done at the very beginning in parallel.

  Ts + n·Tw → We have a single message of size N
- The upper boundary needs to be delayed until it has been computed.
  As we can see it has been splitted in different parts as we are using Blocking.



This means that when we send the upper boundaries we have several messages of size C instead of a single message of size N.

Some of these communications can be overlapped (parallel).

The cost of each message is → **Ts + c·Tw**

Then we need to account for how many of these communications we need to do:

- Nº of Upper boundaries before the longest task starts → **P-1**
- +
- Nº of Upper boundaries while the longest task performs the computations.
  We have N/C blocks.
  The number of communications that are left is 4.

  (N/C)-1

If we rearrange the formula, we obtain:

$$T_{overhead} = (t_s + n \times t_w) + (t_s + c \times t_w) \times (P-2 + n/c)$$

For the computations:

- We have multiple tasks for each processor → There are N/C tasks for each processor.
  The cost of each task will have to do with the number of elements being computed.
  - There are C columns and N/P rows
  - The computation cost for computing 1 element is Tbody.

- Then we need to account for the number of blocks.

$$T_{calc} = t_{body} \times (n/P \times c) \times (P - 1 + n/c)$$

Introduction to OpenMP

A **CPU**, or Central Processing Unit, is the primary component of a computer system responsible for executing instructions and performing calculations. It is often referred to as the "brain" of the computer because it carries out the majority of the processing tasks required for the system to function.

The CPU interprets and executes instructions from the computer's memory, performing basic arithmetic, logical, control, and input/output (I/O) operations.
Modern CPUs consist of multiple processing cores, which allow for parallel execution of instructions and improved performance.

A **core** is an individual processing unit within a CPU that is capable of executing instructions independently. This allows parallelism, meaning that multiple tasks or threads can be executed concurrently.

When we run a program in a computer, that program is going to be a sequential program (sequence of machine instructions) and those instructions are going to be referred to as threads. So, a **thread** or task refers to a unit of execution within a program. It represents a sequence of instructions that can be scheduled and executed independently by a CPU core or a processing unit.

Each core will have its own memory, the Cache Memory L1, L2, L3 (L3 is shared between all cores). And the whole CPU will have a common Main Memory.
This is a shared memory system because the memory (L3 and main memory) is shared between all cores.

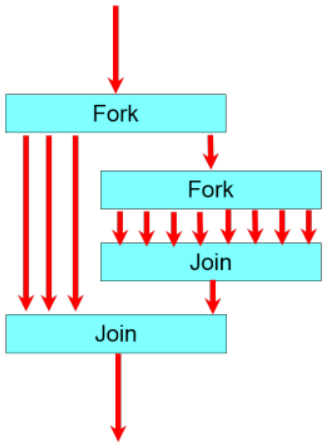
If I have a lot of money, I can use two different machines (MareNostrum) and connect them through the network. But this is a distributed memory system because we can't see the memory of the other machine, only mine.

When running the applications, you normally specify the number of threads you want to use, but in each case the argument is different:

- For example, -t 10 means that you want to use 10 threads in BWA
- In other programs you need to use -threads 10

OpenMP uses a fork-join model

- The master thread spawns a team of threads that joins at the end of the parallel region
- Threads in the same team can collaborate to do work



The idea here is that you have a sequential application (1 thread running) and at some point I am able to open several threads of execution and run the program in parallel.

All threads are going to do the same type of instructions with a different subset of data.

The ideal situation is to run one thread per core (having extra threads does not pay off sometimes). If I have 4 cores, dividing my application into 4 threads is enough in most of the cases. If I have more threads, they will have to wait until the cores are free (the threads are going to run 4 at a time).

There is hyperthreading technology, which means that a core has enough resources to hold 2 threads at a time. So, physically we have one core but I have extra stuff to run 2 threads simultaneously.

OpenMP defines a relaxed memory model

- Threads can see different values for the same variable
- Variables can be shared or private to each thread

We need to know 3 things:

- How do we open and close multiple threads
- Guarantee that the access of the data is safe. Multiple threads can not modify the data at the same time.
- Synchronize threads and memory and wait for termination of tasks.

Constructs or pragmas are going to be added into our programs.

- #pragma omp (then we add the construct that we want)

Creation of threads

We need to specify the parallel construct. We will open a parallel block that will be runned in parallel:

- #pragma omp parallel [clauses]

Main clauses are:

- Num_threads(integer)
- if (expression)
- shared (list-variables)
- private ()
- firstprivate ()
- reduction ()

How do we specify the number of threads? There are 3 mechanisms:

- Use an environment variable: **OMP_NUM_THREADS** → It is used to specify the default number of threads before the program starts running. In the command line
  - *OMP_NUM_THREADS=4 ./file*

- We can use a function: **omp_set_num_threads** → Modify the number of threads during the execution. Inside the program
  - *omp_set_num_threads(4);*

- We can use the "**num_threads()**" pragma directive in each block. In each pragma
  - *#pragma omp parallel num_threads(N)*

Example 1.

```
#include <omp.h>

 main(int argc, char *argv[]) {

 printf ("Starting...",\n);

 /* Fork a team of threads  (assuming that OMP_NUM_THREADS
  variable is set to 4) */

 #pragma omp parallel
  {
    printf("Hello World from thread = %d\n",
           omp_get_thread_num());

   }

 printf ("Finishing...", \n);

}
```

How do I specify in Linux that the variable OMP_NUM_THREADS is equal to 4?

export OMP_NUM_THREADS=4

The output is going to be:

Starting … **# This is done sequentially (only one thread)**

Hello World from thread = 0 **# This is done in parallel (4 threads)**
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3

Finishing … **# This is done sequentially (only one thread)**

This is the order that we will see (opening and joining forks act as a synchronization point). But inside the parallel block, each thread will give the output as soon as they end (so they can be not organized).

If we want to give a concrete order, we need to specific clauses:

- #pragma omp ordered

If we are doing a loop, we use the ordered clause along with the ordered construct:

- # pragma omp for ordered
  for (int i = 0; i < 10; i++) {
  #pragma omp ordered {...

Example 2.

```
#include <omp.h>

 main(int argc, char *argv[]) {

 printf ("Starting...",\n);

 /* Fork a team of threads  (assuming that OMP_NUM_THREADS
  variable is set to 4) */

 #pragma omp parallel
  {
  int i;
  for (i=0; i<2; i++)
    printf("Hello World from thread = %d iteration = %d \n",
           omp_get_thread_num(), i);

   }

 printf ("Finishing...", \n);
}
```

Starting …
Hello World from thread 0 iteration 0
Hello World from thread 1 iteration 0
Hello World from thread 2 iteration 0
Hello World from thread 3 iteration 0
Hello World from thread 0 iteration 1
Hello World from thread 1 iteration 1
Hello World from thread 2 iteration 1
Hello World from thread 3 iteration 1
Finishing …

Important: We are going to see iteration 0 before iteration 1!

Example 3.

```
#include <omp.h>

main(int argc, char *argv[]) {
int nthreads;

/* Fork a team of threads with each thread having a private
tid variable */
#pragma omp parallel
  {
  /* Obtain and print thread id */
  int tid;
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  } /* All threads join master thread and terminate */
```

Each thread has a private tid variable because this variable is inside the parallel block.

Hello World from thread = 0
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
Number of threads = 4

Number of threads can not appear before “Hello World from thread = 0”.

Sequential Computation of pi

```
static long num_steps = 100000;
double step;
main() {
    int i;
    double x, sum = 0.0, pi = 0.0;

    step = 1.0/(double) num_steps;
    for (int i = 0; i < num_steps; ++i) {
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x*x);
    }
    pi = step *sum;
}
```

Parallel Computation of pi

```
static long num_steps = 100000;
double step;
main() {
    int i;
    double x, sum = 0.0, pi = 0.0;

    step = 1.0/(double) num_steps;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (int i = 0; i < num_steps; ++i) {
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x*x);
    }
    pi = step *sum;
}
```

X has to be a private variable.
Sum has to be global, since it holds the final result.

Data Sharing Attribute Clauses

Because OpenMP is based upon the shared memory programming model, most variables are shared by default.

- Global variables include:
    - File scope variables, static. If the variable is declared at the top of the program, it is going to be shared.

- Private variables include:
    - Loop index variables
    - Stack variables in subroutines called from parallel regions. If a variable is defined inside a parallel block, it is going to be private.

The global variables can be declared into private variables by adding the clause “private”. There are many other clauses:

- SHARED
- PRIVATE

- **FIRSTPRIVATE**: We will have a private variable but its initial value will be equal to the original value of the shared variable. It has the value that it had before entering the parallel block.
  If it is not defined, it will have an unknown value.

- **LASTPRIVATE**: The variable inside the construct combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable. So, all threads will have a private variable with the same initial value. But the value copied back into the original variable object is obtained from the last iteration or section of the enclosing construct.

- **REDUCTION**: Used to collapse the values of private variables into a global variable. We use “reduction(operator : list)”
    - Operators → +, -, *, min, max
    - The compiler creates a private copy of each variable in list that is properly initialized to the identity value

Synchronization

Some OpenMP synchronization mechanisms:

- **#pragma omp barrier** → Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
    - Some constructs have an implicit barrier at the end, for example the parallel construct.

```
#pragma omp parallel
{
       foo ()

#pragma omp barrier     % Forces all ocurrences to happen
       bar ()           % before all bar ocurrences

}    % Implicit barrier at the end of the parallel region
```

- **#pragma omp critical (name)** → Provides a region of mutual exclusion where only one thread can be working at any given time.

```
int x=1, y=0;
#pragma omp parallel num_threads (4)
{
#pragma omp critical (x)
       x++;                     % Different names: one thread
#pragma omp critical (y)        % can update x while another
       y++;                     % updates y
}
```

- **#pragma omp atomic** [update || read || write]→ Ensures that a specific storage location is accessed atomically, avoiding the possibility of multiple, simultaneous reading and writing threads
    - By default it is set to update

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i] *= 2;
}
```

Example 4.

```
#include <stdio.h>
#include <omp.h>

int main(){
   int x=2;

   #pragma omp parallel num_threads(2) shared(x)
   {
   if (omp_get_thread_num() == 0) {
        x = 5;
   } else {
        printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   }
   #pragma omp barrier
   if (omp_get_thread_num() == 0) {
        printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   } else {
        printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
   }
   }
}
```

1: Thread 1: x = 2 or 5 (normally it will be a 2)
2: Thread 0: x = 5
3: Thread 1: x = 5

There is only one box for x!

Useful Routines

- int omp get num threads()
- int omp get thread num()
- void omp set num threads()
- int omp get max threads()
- double omp get wtime()

Loop parallelism

#pragma omp for [clauses]
    for (… ; … ; ...)

Where some possible clauses are:

- private
- firstprivate
- reduction
- schedule (type)
- nowait
- collapse (n)
- ordered (n)

The loop affected by the pragma is going to be automatically divided by the compiler.

- Loop iterations must be independent.
- The default data-sharing attribute is shared
- It can be merged with the parallel construct. This is used if we didn't open a parallel fork beforehand.
    - #pragma omp parallel for

Example 5.

```
int main () {
int i, id, MAX = 4;

  omp_set_num_threads (4) ;
  #pragma omp parallel for private (i, id)
  for (i = 0; i< MAX ; i++)
  {
    id = omp_get_thread_num();
    printf ("(%d) Hello iter. = %d \n", id, i);
  }
}
```

0 Hello iter = 0
1 Hello iter = 1 # The number of the thread can change, but the iterations are ordered.
2 Hello iter = 2
3 Hello iter = 3

The private(i) is not necessary, since it is private by default.

Example 6.

```
int main () {
int i, id, MAX = 4;
  omp_set_num_threads (4) ;
  #pragma omp parallel
  {
  #pragma omp for private (i, id)
    for (i = 0; i< MAX ; i++)
    {
      id = omp_get_thread_num();
      printf ("(%d) Hello iter. = %d \n", id, i);
    }
  printf ("End of first loop\n");
  #pragma omp for private (i, id)
    for (i = 0; i< 2*MAX ; i++)
    {
      id = omp_get_thread_num();
      printf ("(%d) Goodbye iter. = %d \n", id, i);
    }
  }
  printf ("End of second loop\n");
```

0 Hello iter = 0
1 Hello iter = 1 # The number of the thread can change, but the iterations are ordered.
2 Hello iter = 2
3 Hello iter = 3

End of first loop
End of first loop
End of first loop
End of first loop

0 Goodbye iter = 0
0 Goodbye iter = 1
1 Goodbye iter = 2
1 Goodbye iter = 3
2 Goodbye iter = 4
2 Goodbye iter = 5
3 Goodbye iter = 6
3 Goodbye iter = 7

End of second loop

Example 7.

```
int main ()  {
int i, id, MAX = 4;

  omp_set_num_threads (4) ;
  #pragma omp parallel private (i, id)
  {
    id = omp_get_thread_num();
    for (i = 0; i< MAX ; i++)
       printf ("(%d) Hello iter. = %d \n", id, i);
  }
  #pragma omp parallel for private (i, id)
  for (i = 0; i< MAX ; i++)
  {
    id = omp_get_thread_num();
    printf ("(%d) Goodbye iter. = %d \n", id, i);
  }
}
```

0 Hello iter = 0
0 Hello iter = 1
0 Hello iter = 2
0 Hello iter = 3
1 Hello iter = 0
1 Hello iter = 1
1 Hello iter = 2
1 Hello iter = 3
2 Hello iter = 0
2 Hello iter = 1
2 Hello iter = 2
2 Hello iter = 3
3 Hello iter = 0
3 Hello iter = 1
3 Hello iter = 2
3 Hello iter = 3
0 Goodbye iter = 0
1 Goodbye iter = 1
2 Goodbye iter = 2
3 Goodbye iter = 3

Example 8.

```
#include <omp.h>
#include <stdio.h>

int main ()  {

  int tmp =10;

  #pragma omp parallel num_threads(3)
   #pragma omp for firstprivate (tmp) lastprivate (tmp)
       for (int j= 0; j<9; ++j)
             tmp = tmp+j;
   printf ("Final value of tmp =%d\n", tmp);
}
```

Schedules

We can distribute the iterations in different schedules:

- **Static, N:** The iteration space is broken in chunks of approximately size N/num_threads. We can also define the size N of the chunks.
    - Low overhead
    - Can have load imbalance problems

- **dynamic, N:** Threads dynamically grab chunks of N iterations until all iterations have been executed. So, we are not distributing all the iterations at the beginning, but only a size N. The other iterations will be distributed to threads that are available.
  By default, N = 1
    - High overhead
    - Solve imbalance problems

- **guided, N:** Variant of dynamic. The size of the chunks decreases as the threads grad iterations, but it is at least of size N. If no chunk is specified, N=1

Nowait

When we have a for loop, there is an implicit barrier at the end. So, threads will not move on until the loop has been finished unless we specify it using the clause Nowait.

This allows to overlap of the execution of non-dependent loops or tasks.

#pragma omp for nowait

Collapse

It is useful when we have nested loops, but the nest must traverse a rectangular iteration space.

- For example N iterations by M iterations. Instead of having 2 layers of iterations, we take the N·M iterations and distribute them.

If we do not collapse, we are going to distribute the outer loop. If we have 2 threads, one thread will do N/2 iterations of the outer loop and its corresponding iterations of the inner loop. The other thread will do the other N/2 iterations and its corresponding iterations of the inner loop.

Ordered

The order of the operations inside the loop are non-deterministic.

If we want to establish an order, we have 2 mechanisms:

- **Ordered**
- **Doacross**: We say that we need to do something before doing another thing, we create a dependance. We make sure that a previous iteration has been completed before doing another thing.

Single

Only one thread if the team executes the structured block
There is an implied barrier at the end.

#pragma omp single [clauses]

where clauses can be:

- private
- firstprivate
- nowait

Memory consistency

OpenMP lacks consistency memory model, which means that threads are modifying variables in the local memory, even threads that have a shared variable, they have a local copy of the variable. From time to time, this shared variable is going to be updated to the global memory.

At some particular point, the value of a thread affecting a global variable has not been made visible. This affects the order of operations and the behavior of the program.

If we want to make sure that the global copies of the variable are in the global memory, we need to use the flush construct, which will update the variables into the global space.

#pragma omp flush (list)

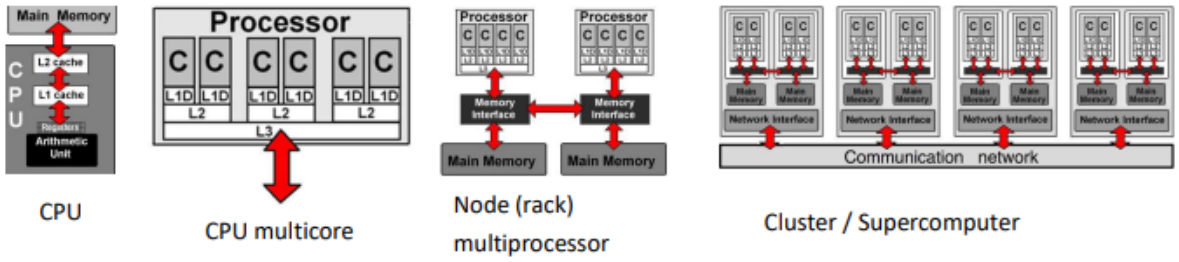
We are ensuring that we do not have a local copy of a shared variable that has not been updated in the global space.

SLURM

We have seen that a processor has several cores, I can run a thread in each core and we have several layers of cache memory.

I can put several processors on top of a board and we will have a server machine.

Any processor in this kind of machine can access any memory in the system, even though some bunch of memory is closer to one processor than another.



This is called NUMA Architecture. The time it takes to access one particular data depends on which memory band it is using.

If I have many racks, I can connect them together through the network and I will have a supercomputer.

If I want to increase the efficiency, I can add a GPU inside each one of these racks.

How do I use this supercomputer?

We do not have direct access to all these nodes of the system (it's not like working with a laptop).

We need to connect to an entry point machine, establishing an ssh connection (secure connection) to the machine. Once you have logged in, you can have access to the resources of the system.

We must take into consideration that there are hundreds of thousands of users.
To manage this, we add an extra layer on top of the OS that will manage many users that will try to run many applications on many systems. This is called **Batch Queue System**.

- The Batch Queue System that we will use is **SLURM**

SLURM performs 3 basic operations:

- Allocates machines or resources to a user
- Starts and controls the execution
- Once it finishes the execution, it manages a queue of pending jobs.

All the machines that we have are splitted into partitions. So, machines are classified in partitions, so that you submit a job to a particular partition (set of machines).

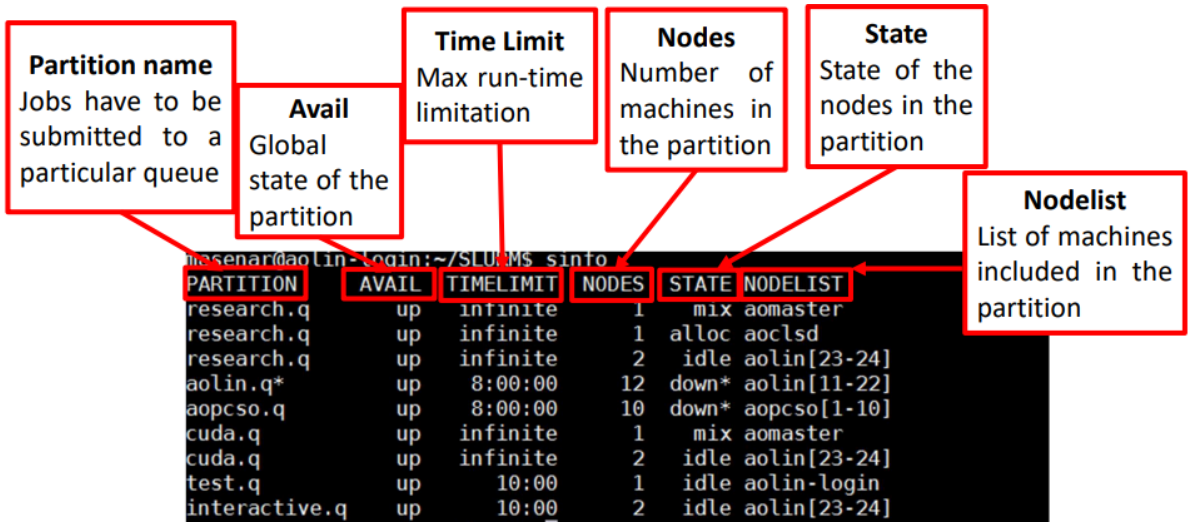
- Interactive partition
- Execution partition

User Commands:

- Information of all partitions on a cluster: sinfo
- Job submission: sbatch, srun
- Job deletion: scancel
- Job status: squeue –u
- Queue list: squeue
- Resource allocation: salloc
- Job control: scontrol
- User accounting: acctmgr

Job specification

- script directives: #SBATCH (SLURM)

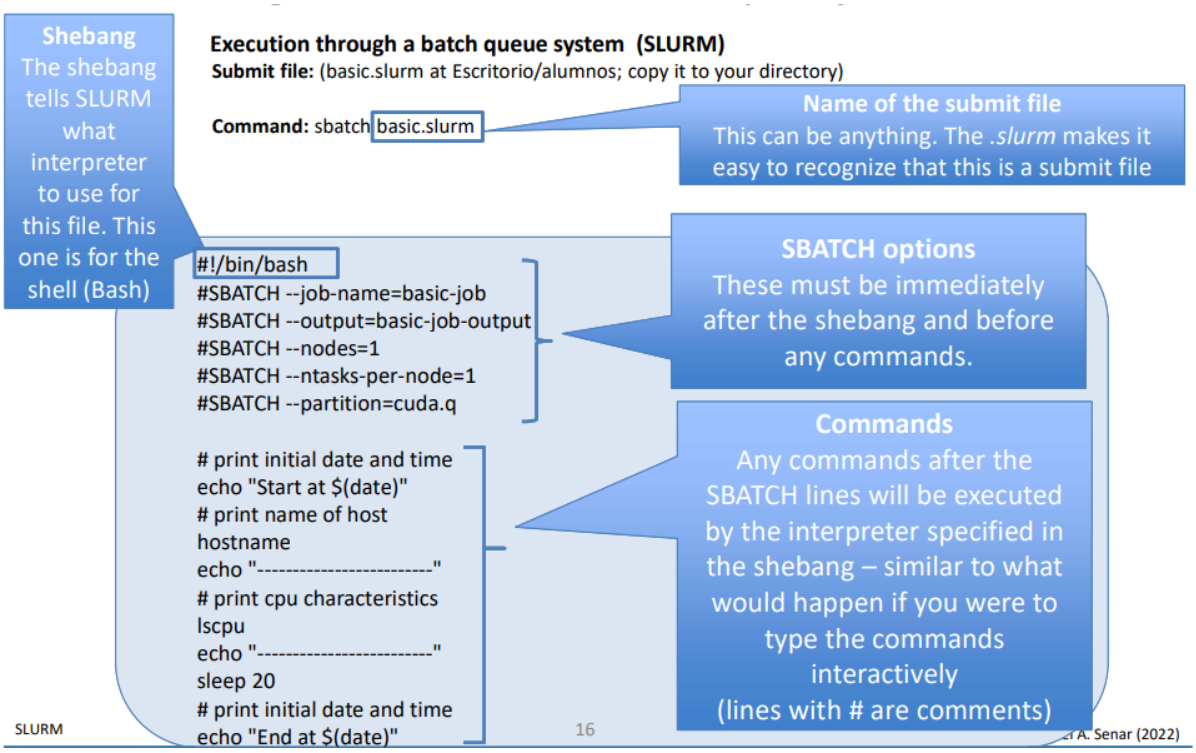


There is no interactivity between the program and the user. You just submit the job to the system and the system decides when the job will be runned.
Thus, if your program requires some input, you need to modify the slurm file so that all the input can be read at any time.

The output will be given in a file, not in the terminal.

What do we submit to SLURM? sbatch name.slurm

The slurm file is composed of a header and shell script operations



The 5 files after the Shebang are only understood by the SLURM.

In this case we are using one node **(#SBATCH –nodes = 1**) and within that node we have multiple CPUs and within these CPUs we have multiple cores.

How many cores do we need to run this file? Just one. For this reason **#SBATCH –ntasks-per-node = 1**.

So, we are specifying the number of threads. So, we are requesting one node but only one of its cores (element that only has one thread).

If we have an MPI application, we are going to use more than 1 node.
If we have an OpenMP application, we are going to use more than 1 task per node (multiple threads).

Command	What it does
--nodes	Number of nodes requested
--time	Maximum walltime for the job – in DD-HH:MM:SS format
--mem	Real memory (RAM) required per node – can use KB, MB and GB units – default is MB Request less memory than total available on the node
--ntasks	Number of tasks to run
--tasks-per-node	Number of tasks that will be launched per node
--cpus-per-task	Number of processors required by each task
--mem-per-cpu	Minimum of memory required per allocated CPU – default is 1GB
--output	Filename where STDOUT will be directed – default is slurm-<jobid>.out
--error	Filename where STDERR will be directed – default is slurm-<jobid>.out
--job-name	How the job will show up in the queue

Environmental variables

They can be used in the command section of a submit file (passed to scripts or programs via arguments). Cannot be used within an #SBATCH directive → Use Replacement Symbols instead

Environment Variable	Description
SLURM_JOB_ID	batch job id assigned by SLURM upon submission
SLURM_JOB_NAME	user-assigned job name
SLURM_NNODES	number of nodes
SLURM_NODELIST	list of nodes
SLURM_NTASKS	total number of tasks
SLURM_QUEUE	queue (partition)
SLURM_SUBMIT_DIR	directory of submission
SLURM_TASKS_PER_NODE	number of tasks per node

Imagine that we want to write the hostname into a file that has the same name as the job (basic-job). How can we do this?

hostname > $SLURM_JOB_ID_hostname.txt

Replacement Symbols

Symbol	Value
%A	Job array's master job allocation number
%a	Job array ID (index) number
%j	Job allocation number (job id)
%t	Task identifier (rank) relative to current job
%N	Short host name – (name of the first node in the job)
%u	User name
%x	Job name

A number can be placed between % and the following character to zero-pad the result

Example:

- #SBATCH –output = job%j.out → Create job777.out for job_id=777
- #SBATCH –output = job%9j.out → Create job00777.out for job_id=777

Array Job Submissions

We can use the same submission script to run multiple jobs that are similar. You have several jobs doing the same stuff and SLURM is going to create several instances of these jobs.

You need to specify: **#SBATCH –array=<array number>**

If we use #SBATCH –array=1, 5, 10 i will have 1 main job and 3 childs with name "common_ID.1, .5, .10"

These array jobs are useful when you need to perform the same operations in different files.

Example 9. What is this job doing?
In this case we have a single job with a for loop!

```
#!/bin/bash
#SBATCH --job-name=for-job
#SBATCH --partition=execution

# print initial date and time
echo "Start at $(date)"

for value in {0..4};
do
  echo $value >> for_$value.out
done

# sleep 10 seconds
sleep 10
# print final date and time
echo "End at $(date)“
```

It prints the Start Date

Creates 5 different files with the following information:

- for_0.out with value 0
- for_1.out with value 1
- for_2.out with value 2
- for_3.out with value 3
- for_4.out with value 4

Then it sleeps for 10 seconds and prints the Final Date.

Note that the general output of the job is going to be stored in: slurm-1777887.out

Since every time we submit a job and we do not specify the output file, 2 files are going to be created:

- slurm-job_ID.out → It will contain the Start Date and End Date
- slurm-job_ID.err

If we submit this job again, we will add in the same files the same information again (no overwrite). Since we are using >>, which means that we append at the end of the file.

Example 10. What is this job doing?
In this case we are submitting an array job. So, 5 instances of the same job are going to be executed. In other words, I will have in the system 5 independent jobs with the same ID with a personalized sub-ID (slurm-178727.0, slurm-178727.1… for example).

```
#!/bin/bash
#SBATCH --job-name=array-job
#SBATCH --partition=execution
#SBATCH --array=0-4

# print initial date and time
echo "Start at $(date)"

echo $SLURM_ARRAY_TASK_ID >>
array_$SLURM_ARRAY_TASK_ID.out

# sleep 10 seconds
sleep 10

# print final date and time
echo "End at $(date)“
```

Since they are independent jobs, they will run at the same time.

Prints the Start Date in each of the 5 slurm-ID_sub-ID.out.

Creates 5 different files with the following information:

- array_0.out with value 0
- array_1.out with value 1
- array_2.out with value 2
- array_3.out with value 3
- array_4.out with value 4

Then it sleeps for 10 seconds and prints the Final Date in each of the 5 slurm-ID_sub-ID.out

Job Dependencies

Allows you to queue multiple jobs that depend on the completion of one or more previous jobs. Maybe because the job before generates files that are going to be used by the next job.

When submitting the job, use the –d argument followed by specification of what jobs and when to execute.

So, the syntax is the following:
sbatch -d <when to execute> : <job_id>

Instead of -d, we can also say –dependency

Common cases:

- after → After the specified jobs have begun execution
- afterany → After the specified jobs have been completed
- afterok → After successful completion
- afternotok → After non-successful completion

If we want so specify multiple jobs, we use: sbatch -d <when to execute> : <job_id> : <job_id>

Job dependencies in a script

It is used when you want to execute multiple jobs that are dependent. You are not going to wait for each task to be completed… So you create a BASH file with the dependencies.

```
#!/bin/bash

# print initial date and time
echo "Start dependentjob at $(date)"
dia=$(date)
echo $dia

# first job - no dependencies
jid1=$(sbatch --partition=aolin.q job1.slurm | cut -f 4 -d' ')
echo $jid1

# multiple jobs can depend on a single job
jid2=$(sbatch  --partition=aolin.q --dependency=afterok:$jid1 job2.slurm | cut -f 4 -d' ')
jid3=$(sbatch  --partition=aolin.q --dependency=afterok:$jid1 job3.slurm | cut -f 4 -d' ')

# a single job can depend on multiple jobs
jid4=$(sbatch  --partition=aolin.q --dependency=afterany:$jid2:$jid3 job4.slurm | cut -f 4 -d' ')
jid5=$(sbatch --partition=aolin.q --dependency=afterany:$jid4 job5.slurm | cut -f 4 -d' ')

# show dependencies in squeue output:
squeue -u $USER -o "%.8A %.4C %.10m %.20E"

# print final date and time
echo "End dependent job at $(date)"
```

When submitting a slurm file (sbatch bla bla bla), we obtain the following output:
Submitted batch job <ID>

If we want to obtain the ID of this job, we can use: **cut -f 4 -d ‘ ’**

We will not submit this script, since it is not a SLURM script. It’s a BASH script and therefore we will run it!

srun

The srun command allows the immediate execution of a particular job. We have a set of nodes that can be used to run immediately a particular job.

If we use a sbatch submission, we are submitting the job to the system and the system decides when to run it (when the resources are available).

The normal scenario is that we have a particular partition with a small number of nodes available to run interactive jobs using the srun command.
Before using the srun command, we normally use the salloc command to allocate a particular machine.

Using srun to run several jobs

Srun can also be used to run several jobs

Let’s take a look at this SLURM job:

```
#!/bin/bash
#SBATCH -N 2          (We ask for 2 nodes  :      --nodes)
#SBATCH -n 2          (We will run 2 tasks  :      --ntasks)
#SBATCH -c 1          (Each task uses one core : --cpus-per-task)
#SBATCH -p execution

test.cmd
```

test.cmd

```
#!/bin/bash
date
echo "I'm sleeping on..."
hostname
sleep 40
echo "Done sleeping at"
date
```

I am requesting 2 nodes, I will run 2 tasks and each task is going to run one task per cpu.

So, I want to run 2 instances of this program.

The output of this:
If we run this example, only 1 instance is going to run. The system will provide you 2 nodes but only one instance of the program is going to be executed.

If we want to use both machines, we need to use srun. So, inside the SLURM file, i need to specify → srun test.cmd

It will run as many times as we specify in the number of tasks.

Resource specification for parallel jobs

We have several options to specify how many machines I can claim, how many tasks I want to start…

--nodes=number (number of nodes to use; equiv. -N)
--ntasks=number (number of processes to start; equiv. -n)
--ntasks-per-node=number (number of tasks assigned to a node)
--cpus-per-task=number (number of resources (i.e. cores) used by each task; the number of resources (cores) assigned to the job will be the total_tasks number * cpus_per_task number; equiv. -c)
--ntasks-per-socket (number of tasks to run per CPU socket)
--gres=gpu:number (number of GPU assigned to a node)

With all these elements, you can submit large applications. Imagine that you have a MPI application where every process has already been parallelized using OpenMP (each process has multiple threads). So, I have 2 parallelization layers:

- MPI processes
- OpenMP threads within each process

If I have a machine with several nodes, with several CPUs in each node, with several cores in each CPU. The question is, if I want to run this MPI application on this system and I want to take advantage of the maximum parallelism available in my system, how should I submit the job?

Imagine that I have 4 MPI processes and 4 nodes with 2 CPUs in each node with 4 cores in each CPU. How would you submit a job?

Dependencies

There are different types of dependencies, but the only one that really matters is the loop carried dependency, which will prevent parallelization (the other ones can be handled).

A data dependence exists if the computation of one data point requires other data previously computed. If the required data are computed in another loop iteration, the dependence is called “loop carried”.

- Loop carried dependence are often a problem because they assume an execution order that does not exist in a parallel loop.

Example 1

In the first case, we can see that to compute X[i] we need the information from the previous iteration.

```
for (i=1; i<N; i++)
   X[i] = (X[i]+X[i-1])/2;
```

So, to compute element 3, we need to have previously computed element 2. So, there is a dependency. The element that causes the dependence needs to be written before and then read it (read after write dependence).

Example 2

Element Y is generating the dependence.

Because all the iterations after the iteration i==1 need to have variable Y initialized to 1.

```
for (i=1; i<N; i++){
   if (i==1) then
     y=i;
   X[i])=y;
}
```

If there is no order of execution, another iteration can be executed before and Y will have an unknown value (since it has not been initialized) and we will obtain a wrong result.

Anti dependence

This is a “backwards” data dependence in that one iteration requires data that would be modified “later” in the serial case, implying an order that is not there in the parallel case.

Here we have a write after read dependency and therefore I can not parallelize it immediately.

```
for (i=0; i<N-1; i++)
   X[i] = (Y[i]+X[i+1])/2;
```

This can be handled easily using a new array (Z) equal to the initial values of X[i] that will never be modified. Then:

X[i] = (Y[i] + Z[i+1]) / 2;

Output dependence

This is a data dependence that implies a serial loop order, usually by relying on a specific loop iteration being executed last, and using a variable from inside the loop outside of it.

Output dependencies occur when something computed at the last iteration of the loop is used after.

In this case, “a” is causing the dependency.

Thus, we can not parallelize directly.

This can be handled using the lastprivate(a) clause.

```
for (i=0; i<N; i++){
    a = (Y[i]+X[i])/2;
    Z[i] = a;
}
Result = sqrt (a+b);
```

Removing dependencies

Type	Removal Techniques
Loop Carried Flow Dependence	Reduction() clause (OpenMP) Loop skewing Induction variable elimination
Loop Carried Anti Dependence	Auxiliary array
Loop Carried Output Dependence	Lastprivate () clause (OpeMP)
Non Loop Carried	Not necessary

Flow dependence: reduction()

Flow dependence can be removed by reduction if the operation that causes it does not depend on order.

Each thread will have a private sum which will be reduced into a global sum variable!

```
for (i=0; i<N; i++)
   sum += X[i]/2;
```

```
#pragma omp for reduction (+:sum)
for (i=0; i<N; i++)
   sum += X[i]/2;
```

Flow dependence: loop skewing

If the computation of one array element in one iteration depends on an element of another array from a “previous” iteration, shifting computations to another iteration (“loop skewing”) solves the problem.

```
for (i=1; i<N; i++){
   X[i]= (X[i]+ Y[i-1])/2;
   Y[i]= Y[i] + Z[i];
}
```

Can’t be parallelized because iteration i needs Y element from iteration i-1

```
X[1]= (X[1]+Y[0])/2;
#pragma omp for
for (i=1; i<N; i++){
   Y[i]= Y[i] + Z[i];
   X[i+1]= (X[i+1]+ Y[i])/2;
}
Y[N-1]=Y[N-1]+Z[N-1];
```

Order reversed. Regrouping one line makes dependency non-loop carrying

Only in special cases

Flow dependence: elimination of induction variables

In some cases, variables that establish a data dependence can be eliminated by reference to the loop index.

```
factor = 1;
for (i=0; i<N; i++){
   X[i]= factor * Y[i];
   factor = factor / 2;
}
```

Factor establishes an unnecessary dependence…


```
#pragma omp for
for (i=0; i<N; i++)
  X[i]= Y[i] * square (0.5, i-1)
factor = square (0.5, N-1);
```

… and might as well be kicked out of the loop (using an appropriate function). If it is used later, we may compute it outside.

Only in special cases

Anti dependencies: auxiliary array

Anti dependencies can be resolved by copying the needed data into a new array that contains the needed elements as they were before the parallel loop was executed.

```
for (i=0; i<N-1; i++)
   X[i]= (Y[i] + X[i+1])/2;
```

```
copy (X, X_old);
#pragma omp for
for (i=0; i<N-1; i++)
  X[i]= (Y[i] + X_old[i+1])/2;
```

Since nothing has happened to X[i+1] when it is needed in serial...

...we can save the unaltered X in ”auxiliary” X_old before the loop and eliminate the dependence.

Output dependencies: lastprivate()

Output dependencies occur when the value of a variable that is used inside and outside of the loop depends on a specific iteration being executed last. The lastprivate () clause takes care of this.

```
for (i=0; i<N-1; i++){
   a = (Y[i] + X[i])/i;
   Z[i] = a;
}
Result = sqrt (a+b);
```

```
#pragma omp for lastprivate (a)
for (i=0; i<N-1; i++){
   a = (Y[i] + X[i])/i;
   Z[i] = a;
}
Result = sqrt (a+b);
```

The implicit assumption is that iteration N is last to alter a ...

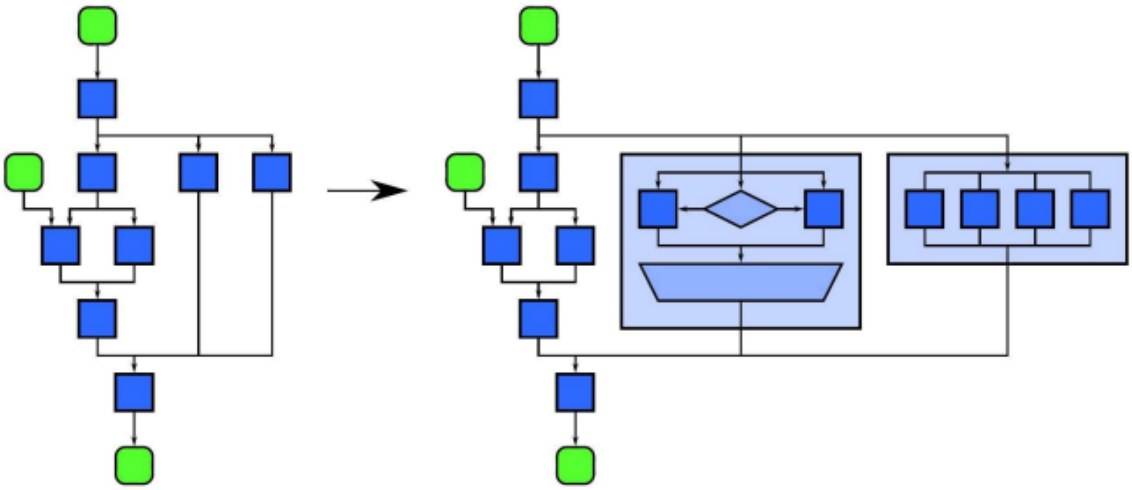
...which is exactly the effect of lastprivate(a)

Parallel patterns

1. Nesting patterns
2. Serial / Parallel Control Patterns
    a. Fork-join
    b. Map
    c. Stencil
    d. Reduction
    e. Scan
    f. Recurrence
3. Serial / Parallel Data Management Pattern
    a. Pack
    b. Pipeline
    c. Geometric decomposition
    d. Gather
    e. Scatter

Nesting patterns

I have a workflow and there are some dependencies. But maybe I can parallelize some parts of the workflow and obtain a nesting parallelism. Meaning that something can be done in parallel and the things that can be done in parallel can also be parallelized.



Parallel Control Patterns: Fork-join

Allows control flow to fork into multiple parallel flows, then rejoin later. So, I have 2 synchronization points.

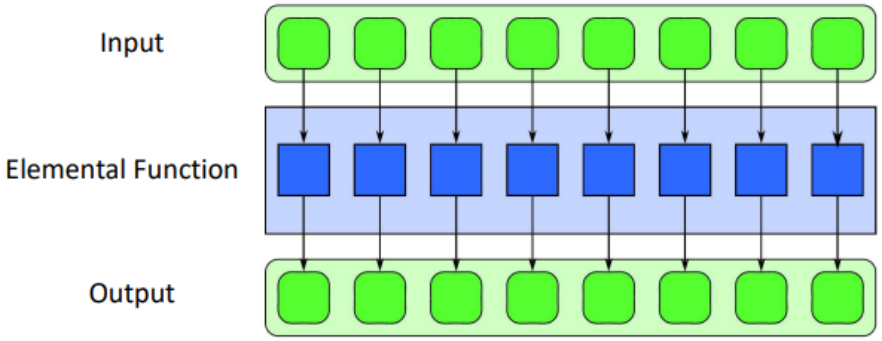
Note that a join is different than a barrier:

- Join → Only one thread continues
- Barrier → All threads continue

Parallel Control Patterns: Map

Performs a computation over every element of a collection that is independent.

Each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection.



Not Jacobi because it has neighborhood relationships. Only if we use different arrays we can use it.

One correct example is adding two arrays.

Parallel Control Patterns: Stencil

I want to compute an element and to do it, I depend on some of the neighbors. The number of neighbors can vary.

Boundary conditions must be handled carefully.

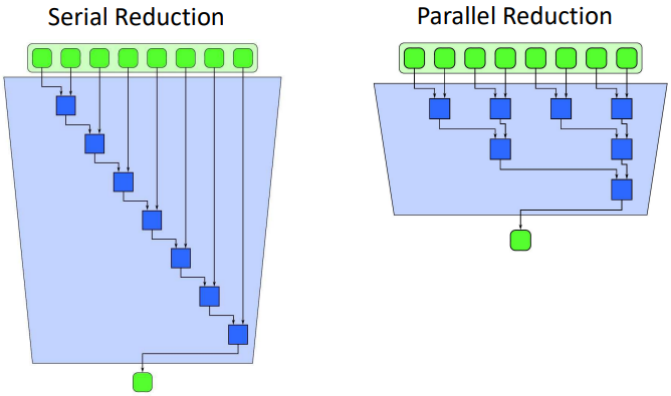
In this case, Jacobi fits this pattern.

Many physical problems can be approximated by this stencil approach:

- I have a material (steel, water…) and its elements can be splitted.
- I can know the evolution of this material. If we want to compute how the heat is transferred in a material, we can compute the heat at some point and compute the heat the next instant using the information from the neighbors.
- The temperature at each point depends on the temperature at the surrounding N elements.

Parallel Control Patterns: Reduction

Combines every element in a collection using an associative “combiner function”.



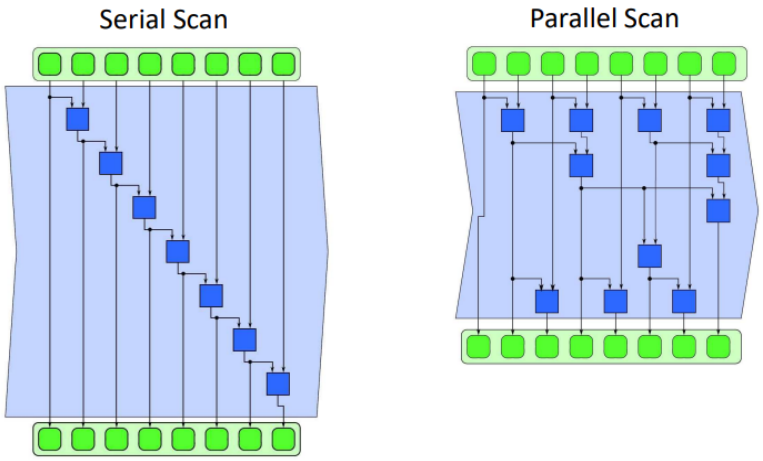
Parallel Control Patterns: Scan

Computes all partial reductions of a collection.
For every output in a collection, a reduction of the input up to that point is computed

It was used in the practice of the sum vector.

- Apply some parallelism at the first level
- Apply corrections later on

To compute a value I need to perform an operation on the previous one.



Parallel Control Patterns: Recurrence

More complex version of map, where the loop iterations can depend on one another.

Map 2 sequences and evaluate the distances. Wavefront parallelism

- Once I have computed one front, I can compute in parallel the next front.

Example: Smith Waterman

GPU Computing and OpenACC

The CPU is the Central Processing Unit and it is used for primary calculations.
The GPU is the Graphics Processing Unit and it is an auxiliary processor designed for parallelizable problems. It has many cores but a small memory.

- Make the same operation many times, for each pixel, for example.

The idea is that some of the computations are going to be delivered from the CPU to the GPU.

Steps:

1. Setup inputs on the host (CPU-accessible memory)
2. Allocate memory for outputs on the host
3. Allocate memory for inputs on the GPU
4. Allocate memory for outputs on the GPU
5. Copy inputs from host to GPU
6. Start GPU kernel → Computations at the GPU are known as kernel executions (each program that is executed in the GPU is known as kernel)
7. Copy output from GPU to host

Latency: Delay that takes place between the start of the operation and the end.

- GPU large latency → A simple operation takes a lot of time
- CPU short latency

The GPU is useful not because of its latency but because of its huge parallelism because it has a lot of cores.

- A lot of throughput

OpenACC

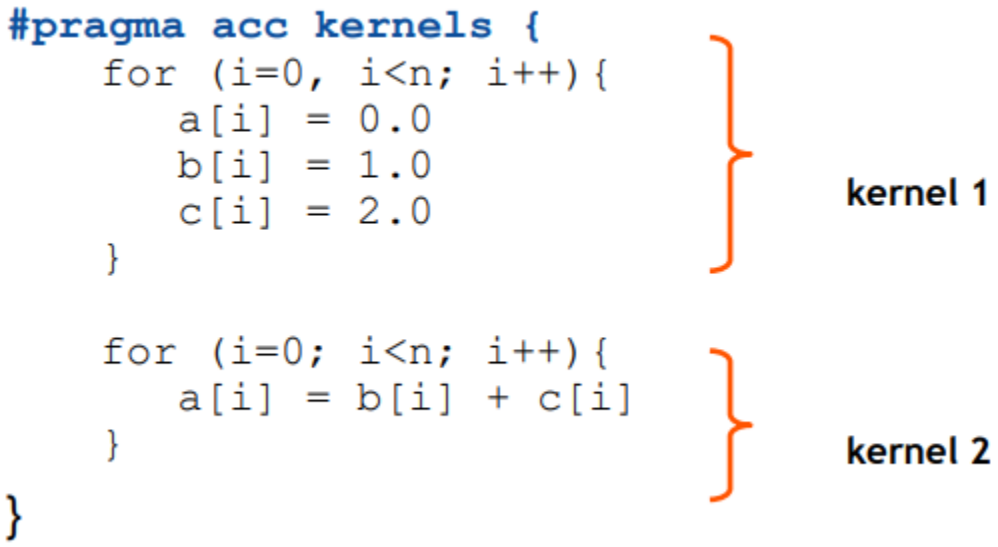
To program an application that will run in a GPU, we must use OpenACC.

When using the directive:

#pragma acc kernels

We are telling the compiler that we want to parallelize this loop and the compiler will do it automatically.

We request that each loop executes as a separate kernel on the GPU. Compiler will be responsible for parallelization.



This looks easy, since the compiler does everything, even the movement of data. **Why don't we just throw kernels in front of every loop? Better yet, why doesn't the compiler do this for me?**

There are two general issues that prevent the compiler from being able to just automatically parallelize every loop:

- Data Dependencies in Loops
- Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results and reasonable performance

Data Management

#pragma acc data [clause]

List of the possible clauses:

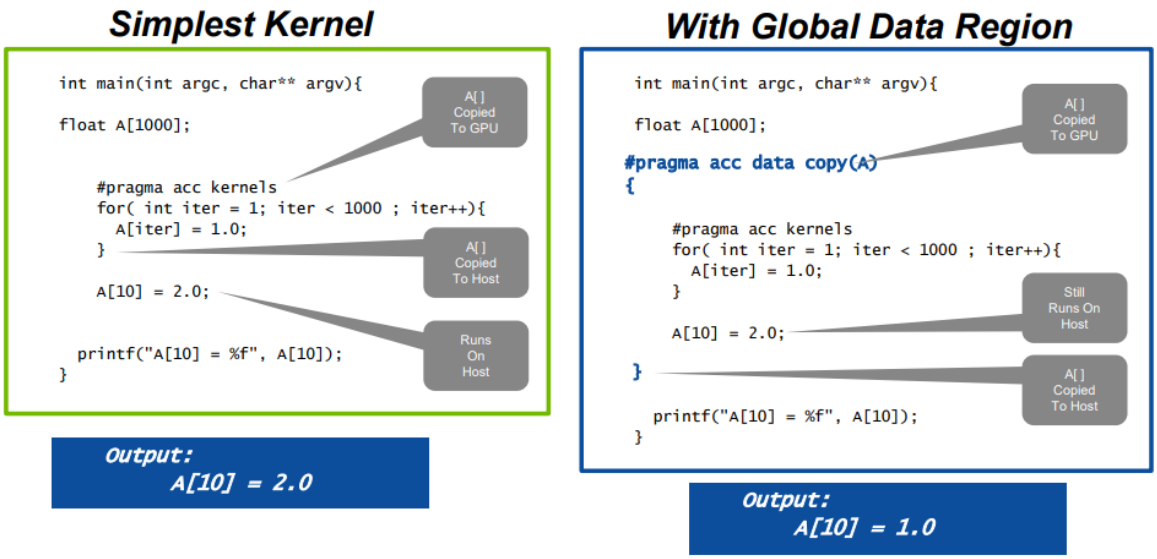
- **Copy(list):** Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

- **Copyin(list):** Allocates memory on GPU and copies data from host to GPU when entering region.

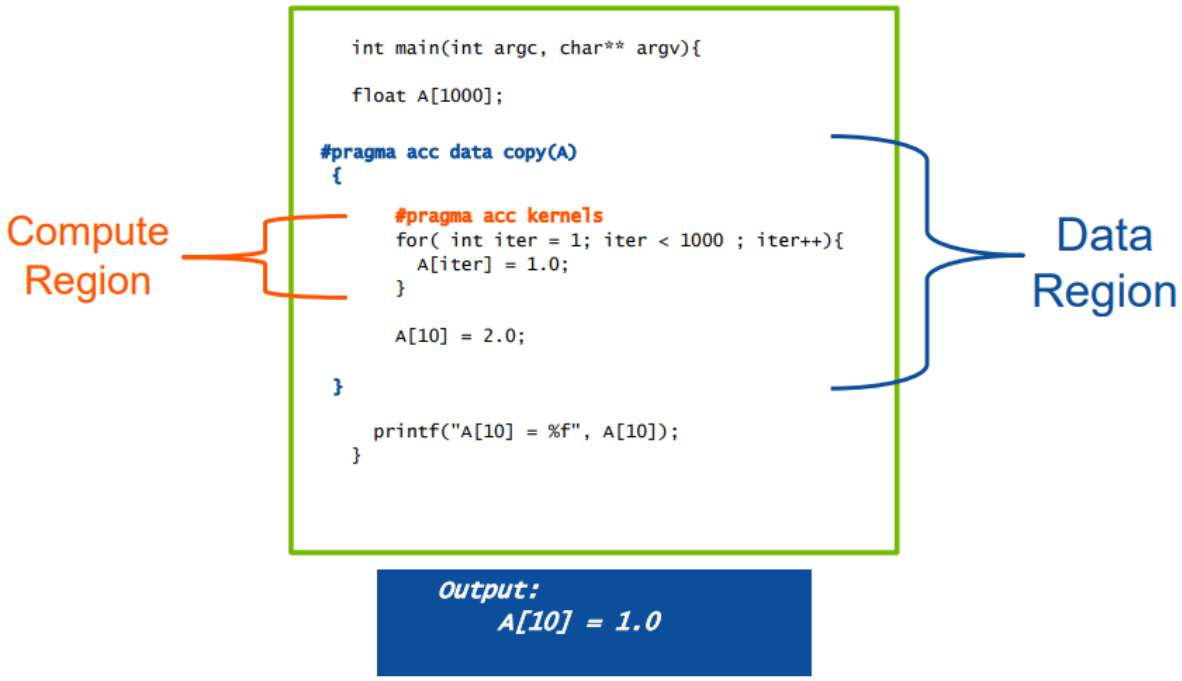
- **Copyout(list):** Allocates memory on GPU and copies data to the host when exiting region.

- **Create(list):** Allocates memory on GPU but does not copy.

Why in the first case the output is 2 and in the second case the output is 1?



Because in the second case the value of A[10] is overwritten.



The parallel directive

So far we've concentrated on the KERNELS directive. The KERNELS directive tells the compiler I may want the loops in a section of code parallelized, but leaves the final decision to the compiler. But what if I know I want the loop to be parallelized? Can I take more control?

#pragma acc parallel loop

You are specifically saying that you want to parallelize that part of the code. In the kernel case, everything is decided by the compiler.

We are responsible if there are data dependencies…