Parallel and distributed systems: paradigms and models

William Simoni

February 2021

Contents

1 Scenario **6**
- 1.1 The beginning of the multi-core era . . . 6
- 1.2 GPU . . . 7
- 1.3 FPGA . . . 8
  - 1.3.1 Evolution of FPGA . . . 12
- 1.4 Conclusion . . . 12

2 Parallel & distributed computation concepts **14**
- 2.1 From sequential to parallel speedup . . . 14
  - 2.1.1 Translate a Book . . . 15
  - 2.1.2 Counting wireless devices in a classroom . . . 17
  - 2.1.3 Assembling conference bags . . . 18
  - 2.1.4 The bank . . . 21
- 2.2 Parallel patterns . . . 24

3 Measures **27**
- 3.1 Latency and Service time . . . 27
- 3.2 Derived Measures . . . 29
- 3.3 Completion time . . . 31
  - 3.3.1 Maximum Parallelism Degree . . . 32
- 3.4 Amdahl Law . . . 35
- 3.5 Gustafson Law . . . 37

4 Laboratory 1 **38**
- 4.1 Measure Time . . . 38

4.2 Threads . . . 43
4.2.1 Synchronization . . . 45

5 Stream and Data Parallel Examples **51**
5.1 Pipeline optimization . . . 51
5.2 GrPPI and Stream data pattern . . . 53
5.3 MAP pattern library examples . . . 55
5.4 Histogram Example . . . 56

6 Stencil Pattern **61**
6.1 Parallel Stencil . . . 64

7 Overhead **66**
7.1 Architectural overhead . . . 66
7.2 False sharing . . . 68
7.2.1 Solutions . . . 70
7.3 Memory allocation . . . 70
7.3.1 Solutions . . . 71
7.3.2 Jemalloc . . . 71

8 Load Balancing **73**
8.1 Static . . . 73
8.2 Dynamic . . . 74
8.2.1 Variable Chunks . . . 74
8.2.2 Job Stealing . . . 75
8.3 OpenMP . . . 76

9 Laboratory 2 **77**
9.1 Async . . . 77
9.2 Packaged Task . . . 78
9.3 OpenMP . . . 78
9.3.1 Sections . . . 79
9.3.2 Locks . . . 80
9.3.3 Barriers . . . 80
9.3.4 Utilities [TO DO] . . . 81
9.3.5 Variables [TO DO] . . . 81
9.3.6 Tasks [TO DO] . . . 81

10 Hiding communication cost **82**
10.1 Shared Memory Multicore . . . 83
10.2 NoW/CoW (Network/Clusters of workstations) . . . 83
10.2.1 Triple Buffering . . . 84
10.3 Accelerators . . . 85

11 Vectorization **87**
11.1 -o3 flag . . . 88
11.2 Vectorization and Parallelization . . . 90
11.3 Loop unrolling . . . 91

12 Work span model **94**
12.1 The model . . . 94

13 Algorithmic skeleton vs Parallel design patterns [TO DO] **97**

14 Refactoring **98**
14.1 Refactoring Rules . . . 98
14.1.1 Refactoring Tree . . . 99
14.2 Normal Form of stream parallel pattern trees . . . 100
14.3 Optimization rules . . . 103
14.4 RPLsh . . . 104
14.5 Automatic management . . . 105
14.5.1 MAPE loop . . . 105
14.5.2 Behavioural Skeleton . . . 108

15 State access patterns **110**
15.1 Odd Even Sort Example . . . 111
15.2 Farm Example . . . 114
15.3 Stream parallel patterns (only) . . . 115
15.3.1 Read only state . . . 115
15.3.2 Owner writes . . . 115
15.3.3 Accumulator . . . 116
15.3.4 ”resource” state . . . 116
15.3.5 Take-away message . . . 118

16 GPU **119**
16.1 SIMD programming model . . . 121
16.1.1 GPU-friend computations . . . 122

16.1.2 Programming model examples 123
16.2 Exploiting Parallelism in kernel workflows 130
16.2.1 Pipeline 130
16.3 Memory 131

Chapter 1

Scenario

1.1 The beginning of the multi-core era

The evolution of processor follows the Moore law: the "power" of the machine, with the same programming model, doubles every two years. A program written for a processor, works also in all the evolution of that processor. This works because:

- The set of instruction of a given architecture was used or extended by the next architectures.
- They used always the same simple abstraction (figure 1.1). There was no way to run more than one instruction in parallel. Pipelining (executing instruction as in a production pipeline) was one of the improvement.

At the beginning of 2000, two problems appeared:

- it was more and more difficult to provide improvement with a single processor. Adding more features was costing much more than the advantage they were getting. For instance, in order to add a feature F_a, we were spending 15 % of the area (of the processor) but getting only 7% of speedup.
- A processor works with a given clock speed. The higher is the clock speed, the higher is the heat the computer must dissipate. Most of the improvement at the time was related to the increase of the clock speed. This lead to the necessity to dissipate more and more heat. Engineers

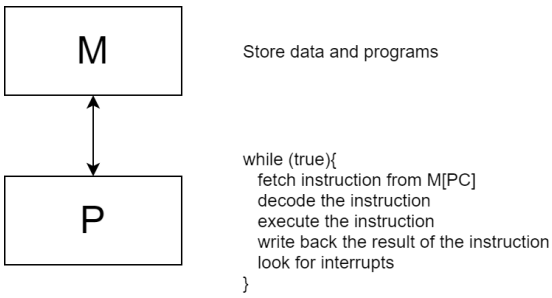


Figure 1.1: Abstraction used by processors. M is the memory and P is the processor.

realized that the heat is proportional to the amount of computing power that we provide. And, if we divide an area in 4 parts (4 processor), we would achieve something that is 4 times more powerful (than having only one processor), without getting 4 times dissipation need. The dark silicon is a chip with a lot of features but that cannot be switched on all together, because if that happen the chip will melt.

These two problems brought to the **multi-core era**.

Nowadays, we have many cores per processor. For instance, an Ampere chip can have 80/128 cores. Therefore, if we want to exploit this processor to increase our application speed, we need to split our application into 128 different flows of control.

1.2 GPU

GPU (graphic processing unit) were hardware co-processors that could be asked to perform some of the operation that were performed by the CPU, at a much higher speed. The GPU were called to operate on the shared graphic memory (shared between CPU and GPU). GPU uses a number of different cores commanded altogether by a processor (so, every processor has not its own flow of control). As shown in figure 1.3, The general processor broadcasted to all the processor the same instruction, and the processors executed that instruction on a different data according to the SIMD model (Single instruction Multiple Data).

GPU evolved becoming the GP-GPU (General Purpose GPU). These GPUs could run arbitrary programs in the same style of the previous GPUs.

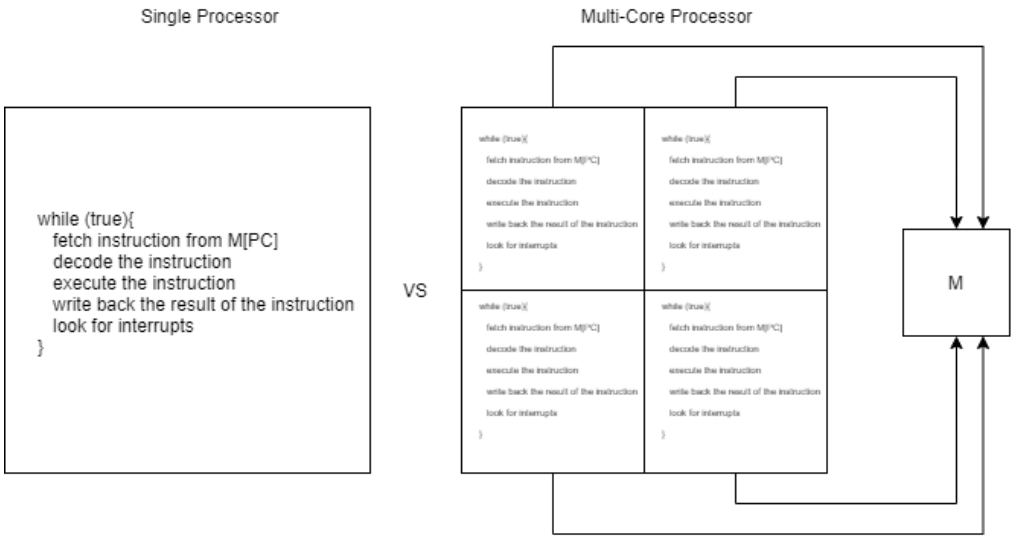


Figure 1.2: A single processor performs only one instruction at a time. In a multi-core processor, there are several independet flows of control that can cooperate using a shared memory.

With GP-GPU we can run **arbitrary data parallel programs**.

Nowadays, GPU are able to accommodate some K of "stupid" cores commanded by some units of general processors. This means that a GPU can easily host 4000 cores that are commanded in groups by 16/32 different general processors that command the stupid cores with the same instruction. The GPU became an accelerator with respect to CPU.

The current architecture, with the GPU, is shown in figure 1.4.

Some computations are much faster if executed by a GPU instead of a CPU. The problem is recognizing these computations.

1.3 FPGA

Besides the GPU, also the FPGA (Field Programmable Gate Array) is another kind of accelerator. Usually the FPGA is a co-processor and shares the same design shown in figure 1.4 where despite having a GPU we have a FPGA.

The kind of acceleration is different with respect to GPU. Indeed we can imagine an FPGA as a matrix made by some communication channels that

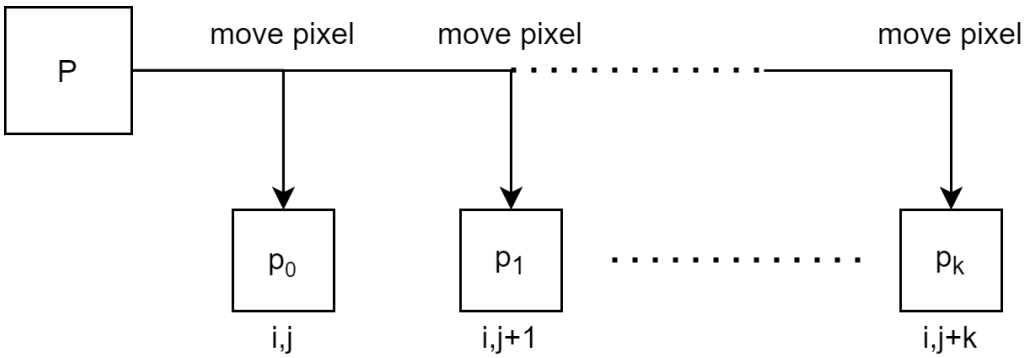


Figure 1.3: The main processor of the GPU broadcasts the instruction to all the processors in the GPU. Every processor execs the instruction on a defined partition of data. For instance, in this case p_0 execs the instruction over the pixel [i,j], p_1 over [i,j+1] and so on.

go vertically or horizontally, and each one of the crossing contains a **cell** (see figure 1.5a). The **cell** is a particular hardware device that can be configured in three different ways:

- A small Boolean function. It is a configurable ROM (read only memory): you can imagine an array of 0 and 1 of size k, where k is the size, in bit, of the input of the Boolean function. The output for an input t, is the value of the bit stored in position t inside the ROM.
- A bit of memory.
- A routing device, that is a set of connections that can be configured to move data for instance north to east, north to south and so on.

Every FPGA contains hundred to thousands of millions of cells per chip. Therefore, we can use parts of the chip to implement different functions. All these computations happen at the hardware level, so we can use the configuration of Boolean functions, bits of memory and routing devices to install on silicon some kind of device which would be more or less the same device we would have printed on silicon if we were using a more traditional silicon compiler technology.

Because of this configurability, the network will work at a very low speed. Typically, the clock speed will go at most in the hundreds of megahertz.

To use the FPGA, the main tools are:

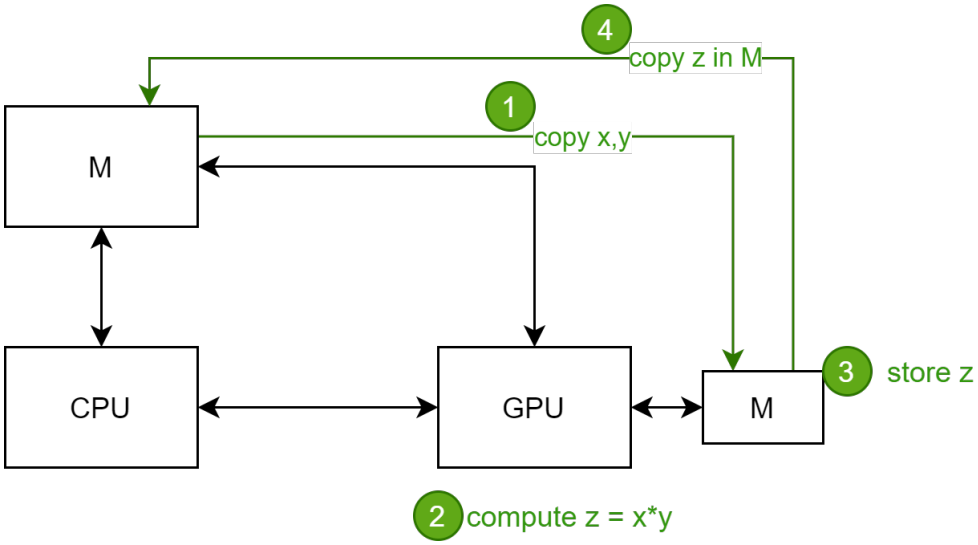


Figure 1.4: z = x*y, where x, y and z are vectors, is one of the computations that are much faster in a GPU. In this case, it is faster to copy x and y in the GPU memory and then compute the multiplication using the GPU than compute x and y using the CPU (without copy x and y).

- The RTL (Register Transfer Languages). Sample of these is the Verilog language. These languages are very complex languages.
- High Level Synthesis languages. These languages are very similar to OpenCL that are the very same languages used to program GPUs. These languages are much simpler thant RTL.

In the case of FPGA, the most convenient thing to do is to use several functions, for instance f_1, f_2, f_3 and f_4, that can be all mapped to a collection of the cells that are typical of the FPGA, and that are chained, as in a pipeline, to perform some computation that eventually, for some x, give us $f_4(f_3(f_2(f_1(x))))$.

The key point is that if we succeed in keeping the functions reasonably simple, we can get a **good clock cycle** τ. Therefore, each function can be computed in one clock cycle. The point is that if we have to compute x_1, x_2, x_3 and x_4, than the computation of $f_1(x_3)$ may happen in the meanwhile we compute $f_2(f_1(x_2))$ and $f_3(f_2(f_1(x_2)))$. So, **we can overlap the computation of different stages relative to different input data**. Example 1.3.1 should clarify this.

Example 1.3.1

Let us suppose we want to compute:

$$\forall i \quad x_i = a_i * b_i$$

The sequence of simple action we should do is the following:

$$LOAD \quad a_i \rightarrow LOAD \quad b_i \rightarrow EXEC \quad * \rightarrow STORE$$

Let us see what happen inside the FPGA in the first five clock cycles:

- time t_0: a_0 is loaded.
- time t_1: b_0 and a_1 are loaded.
- time t_2: b_1 and a_2 are loaded. It is executed $x_0 = a_0 * b_0$.
- time t_4: b_2 and a_3 are loaded. It is executed $x_1 = a_1 * b_1$. x_0 is stored in memory.
- time t_5: b_3 and a_4 are loaded. It is executed $x_2 = a_2 * b_2$. x_1 is stored in memory.

So, at every clock cycle, an x_i is stored in memory. Note that this is totally different from how a GPU would handle this computation. Indeed, it would simultaneously execute the same load operation over all the element of a, than it would execute the same load operation over all the element of b, and so on.

The difficulty is that we have to recognize whether a kind of computation is suitable from being directed to the accelerator, and then we have to put in place a lot of operation that are completely unnecessary with respect to the logic of the program, but that are necessary to the enrollment of the operation in the accelerator.

So, in general, whenever we have a problem:

- We define an algorithm.
- We decompose the algorithm in parts that can be accelerated and parts that cannot be accelerated.

FPGA are very suitable for neural networks computations. Many vendors incorporates FPGA directly in the processor.

1.3.1 Evolution of FPGA

In modern FPGA, besides the cells, we also have DSPs and memories that are regularly sparsely distributed inside the matrix (see figure 1.5b):

- Memories are small memories (1 kb). It is more efficient to use these memories than using 1000 cells to implement a memory of 1 kb. Can be used to split the computation in such a way that the clock cycle is kept small.
- DSPs are simple hardwares that compute MADD (Multiply and Add) operations. In particular, they had a register, and they can perform $register + (x * y)$. So, they execute operations that would require more hardware.

In addition to this, modern FPGA contains traditional "cores". In this case, the FPGA becomes a SOC (System On Chip). So, we can split the computations using assembly code for those computations that can be efficiently performed on a classical core, and the FPGA resources for those computations that need to be accelerated.

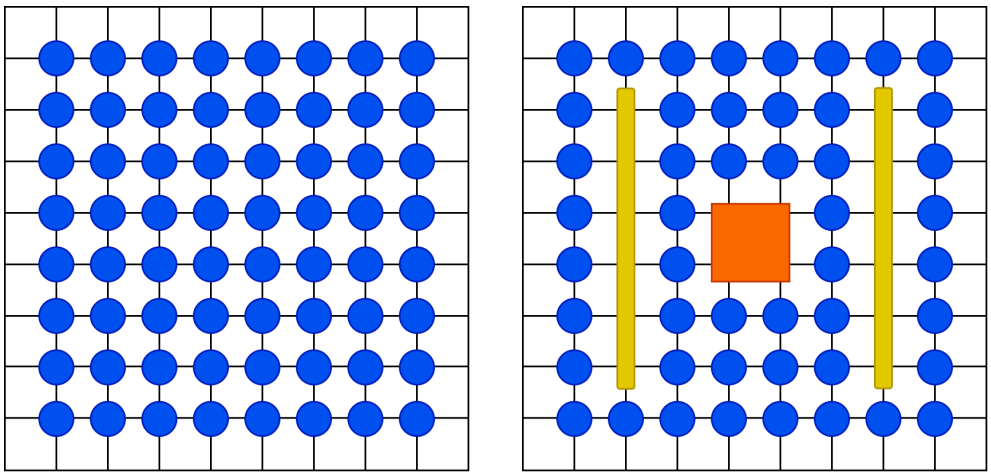
1.4 Conclusion

On one hand we have multi-core systems with a parallelism degree that is 10 to 100 cores. On the other hand, we have GPU with more or less 1000 cores, and FPGA with larger numbers of small devices (what we called cores). All these things require **parallel programs**.

We could create an interconnection network that links several machines to compute one single result. For instance, Top500, and Green500 sites show the most powerful computers in the world.

During the course, we will see that the technologies, methodologies and programming techniques that we need to program supercomputers, are not so different from the techniques that we need to use when we program smaller things like our computers.

There are several degree of effort to make a program parallel:



(a) Basic FPGA (b) FPGA with memories and cores

Figure 1.5: Every blue dot is a cell, the orange square is the traditional CPU and yellow rectangles are DSPs and memories.

- If we take plain C++ stuff, we could candidly write `#pragma omp parallel for` over a parallelizable FOR, then compile the code passing the flag `-fopen mp`, and we finished. Unfortunately, there are not many patterns in which we can simply do this.
- This kind of things on a GPU/FPGA is rather more complicated.
- If we want to use a super computer, it is even worse because we have also to orchestrate the communication between different nodes.

Chapter 2

Parallel & distributed computation concepts

2.1 From sequential to parallel speedup

A program is **SEQUENTIAL** if it does one activity after the other. A program is **PARALLEL** if it does several activities simultaneously. in real life, we can achieve parallelism using some helpers to do more than one thing at a time.

In this section we will discuss what is the speedup we would achieve transforming a sequential program into a parallel one. Where the speedup is the ratio between the time needed by the sequential program and the time needed by the parallel one.

Let us consider a simple example with three activities that respectively require time T_a,T_b and T_c. If we perform these activities sequentially, the program will terminate in $T_a + T_b + T_c = T_{seq}$. On the other hand, if we start executing these activities in the same moment, and we accomplish them in parallel, the time needed will be $max\{T_a, T_b, T_c\} = T_{par}$. Therefore, the speedup is:

$$speedup(3) = \frac{T_{seq}}{T_{par}} = \frac{T_a + T_b + T_c}{max\{T_a, T_b, T_c\}}$$

Assuming that $T_a = T_b = T_c$, the speedup is **3**.

The argument of the speedup function indicate the number of workers used in the parallel program. In this case we used 3 workers. In general, we will indicate this argument with n.

This is of course a very simplistic analysis, in the next examples, we will add new details and see that things are not as beautiful as appear.

2.1.1 Translate a Book

We want to translate a book of m pages. We translate any page in t_p seconds. We have n workers to perform the translation. Repeating the calculations made in the previous chapter, we have that $T_{seq} = m \times t_p$ while $T_{par} = \frac{m}{n} \times t_p$. Therefore:

$$speedup(n) = \frac{m * t_p}{\frac{m}{n} t_p} \approx n$$

One thing we have not yet considered is the **overhead** due to additional things we need to do in the parallel case such as counting the workers, splitting the pages, handing the pages, receiving the translations and merging the translations. Considering the overhead, we have that:

$$T_{par} = \underbrace{t_{split} \times n}_{\text{OVERHEAD}} + \frac{m}{n} t_p + \underbrace{t_{merge} \times n}_{\text{OVERHEAD}}$$

If we again try to look at the speedup, this time we will have:

$$speedup(n) = \frac{m * t_p}{n(t_{split} + t_{merge})\frac{m}{n} t_p}$$

If $m >> n$ and $t_{split} + t_{merge} < n$ (the result without overhead), then the speedup is $\approx n$ yet.

Assuming that the sequential program has the same processing power of the parallel workers, the best speedup we can get is n. Indeed, looking at figure 2.1, the speedup will be always under the blue line. Let us try to prove by contradiction that it is impossible to have a better result than n:

Let us say absurdly that $speedup(n) > n$. Thus:

$$\frac{T_{seq}}{T_{par}} > n \quad \Leftrightarrow \quad \frac{T_{seq}}{n} > T_{par}$$

And this is an absurd. For instance, consider to have one sequential program that terminates in ten hours and the parallel version that concludes in half an hour. Therefore, we have a 20× speedup. But what happens if we execute the workers of the parallel version sequentially? We would terminate in five hours even though the sequential program needs ten hours.

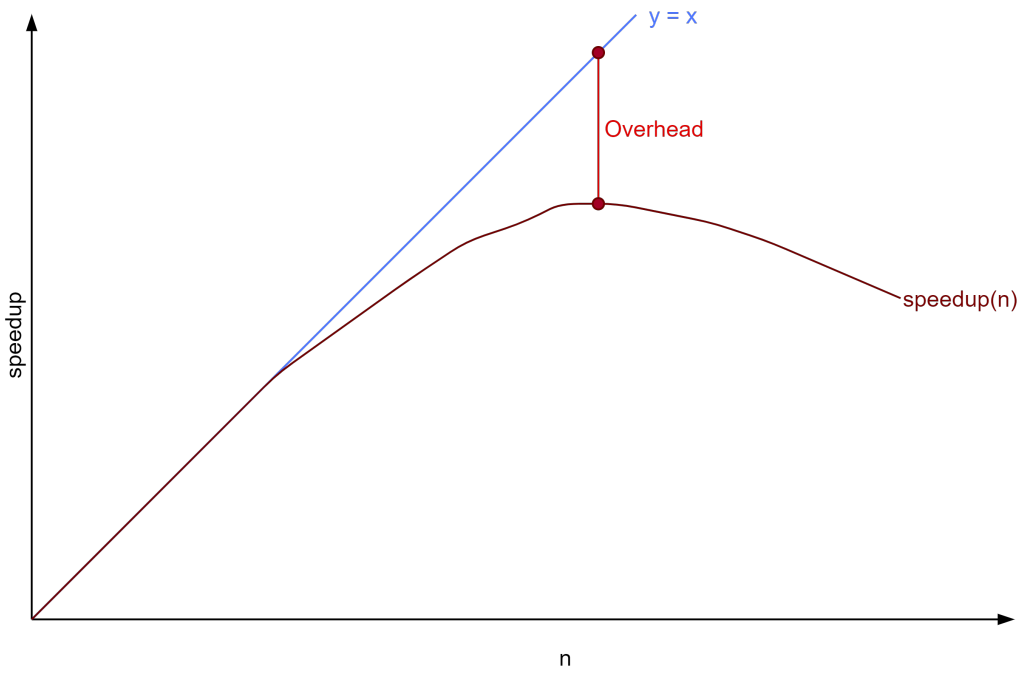


Figure 2.1: Typical curve of a speedup.

The red line in figure 2.1 indicates the overhead between the ideal case and the real case with the overhead. As the number of worker n increase the overhead increase too. At a given point, the work spent to perform the task will be lower than the overhead needed to perform the task in parallel. At that point, adding new workers will decrease the speedup.

An other consideration we need to take into account is the fact that there could pages that require more time than others to be translated. Indeed, we may have pages with small amount of pictures for which a worker needs half an hour, and other pages with quite a lot of pictures where the time spent to translate the pages could drop to 1/4 of an hour. This means that, if we merely split the book into n parts, and we assign each part to each worker, there probably will be workers that will finish much earlier than others. So, we would exploit our resources inefficiently. An alternative approach to avoid this problem would be assign k pages, with $k < \frac{m}{n}$, to every worker. When a worker has terminated, it adds the translated pages to the result and takes other k pages. The drawback of this solution is the increase of the overhead.

Ideally, we could divide the book in phrases instead of pages, but several

times this is not possible. There are limits that are intrinsic in the problem.

This kind of computation is called **embarrassingly parallel computation**. We have this kind of parallel computation whenever we have workers that complete their task completely independently, without any collaboration with other workers. In this case, each worker takes a certain amount of pages, translates autonomously, and finally delivers the result.

Another name we could associate to this kind of computation is **data parallel computation** in which we split in parts the data. Each part is worked to produce a set of partial result. From the partial results we can get the global result. So, the parallelism comes from the single input data.

The next example will be another type of computation that is data parallel but it is not embarrassingly.

2.1.2 Counting wireless devices in a classroom

Let us suppose we want to count the number of wireless devices that are present in a classroom. We could (see also figure 2.2):

1. Do it sequentially, asking one student at a time how many wireless devices he/she has.

2. Sum up by row in parallel. And then sum the partial sums.

3. Organize the students as a binary tree and do the summation bottom-up following the tree structure.

The second method could be a good trade-off between the other methods. Indeed 1. is sequential and is not likely the best way of solving the problem, but on the other hand 3. requires dividing the students in a complicated manner. We can think of the classroom as a matrix, with as many rows as the number of rows in the class, and with as many columns as the number of positions for each row. Therefore, 2. and 3. are both **data parallel** computations because they take a single set of data, which is the matrix. They divide the matrix into parts. Then the students of each chunk sum up the number of devices, and finally, the leaders of each part collaborate to merge the final result. Besides, 2. and 3. are not **embarrassingly parallel** because the student must collaborate to give a partial result to the leader, and the leaders must collaborate to deliver the professor the final sum.

The correctness of the results is ensured by the associativity and commutativity of the sum binary function. As a consequence of these properties,

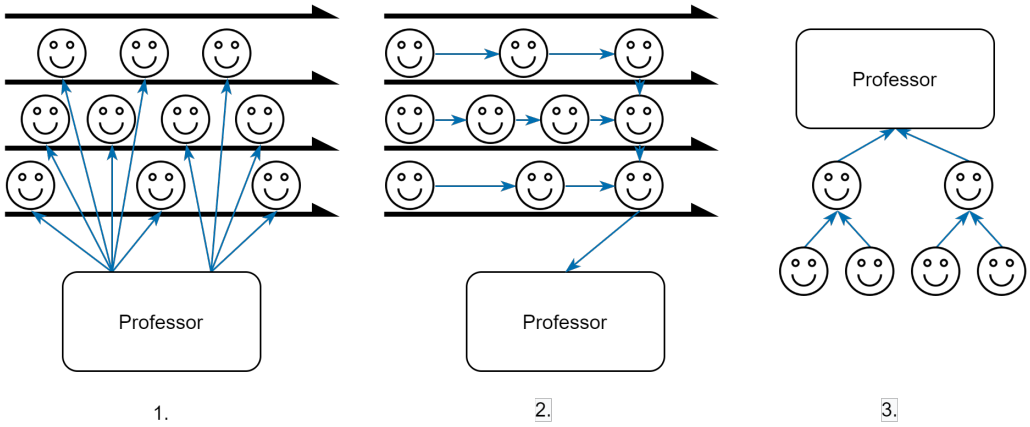


Figure 2.2: How we could solve the problem of count the number of wireless devices. The order in which the images appear follows the order in which the methods are presented.

any order in the execution of the + operations leads to the correct result. For instance, in figure 2.2.2, we could count from the right to the left instead of from the left to the right.

2.1.3 Assembling conference bags

Suppose that we are managing an assembly line that produces food bags as in figure 2.3. A robot adds an empty bag on a conveyor belt, then other robots sequentially add hamburgers, pizzas, and eventually chicken cutlets.

We can see this process as a sequence of three workers that execute a determined function: the first worker fulfills the function **f** =" add the hamburger into the bag", the second worker executes the function **g** =" add the pizza into the bag", and finally, the third worker executes the function **h** =" add the chicken cutlets into the bag". The final bag is the result of the composition of all these functions: $h(g(f(x)))$. Assuming that the time to fulfills the functions are respectively t_f, t_g and t_h, the time required to generate one bag is $t_f + t_g + t_h + some\ overhead$. For now, let us assume that $t_f = t_g = t_h$.

If the assembly line has three workers instead of one, it can handle more than one bag at a time. Indeed, when the assembly line is at regime, a robot can add the hamburger into the bag, while another robot can add the pizza into the bag it received from the previous robot.

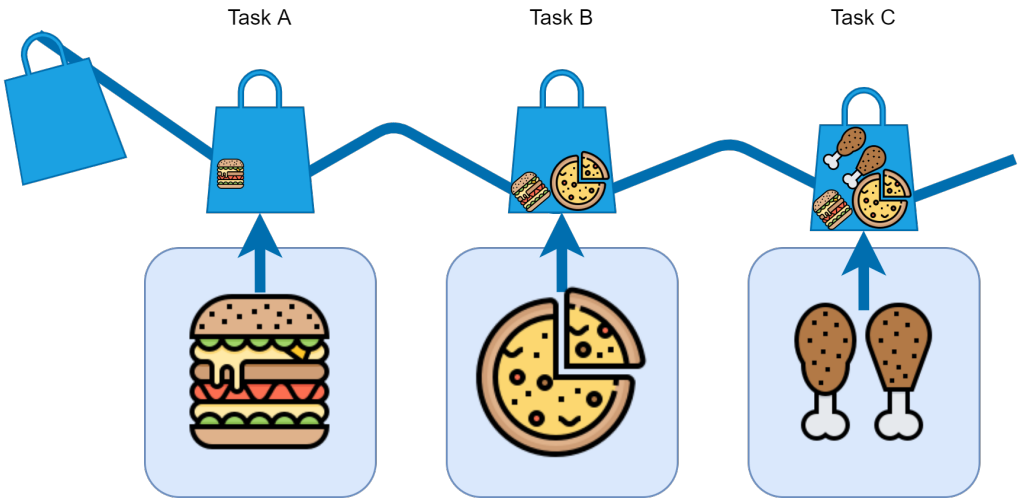


Figure 2.3: An assembly line.

In general, if the number of bags to produce is m, and the number of workers is n, then the assembly line will require $(n-1)t_f + m(t_f) \approx m(t_f)$ if $m >> n$. Indeed:

- $(n-1)t_f$ is the time needed to reach full regime.
- $m(t_f)$ is the remaining time to generate all the bags.

To be as most clear as possible, consider the case with $m = 4$ and $n = 3$. We have to produce x_1, x_2, x_3 and x_4. Figure 2.4 shows the execution of the assembly line. As you can see, we need $2t_f$ to reach full regime ($(3-1)t_f$), and others $4t_f$ to terminate. In the sequential case, we would have needed $m(3t_f)$. Therefore, the speedup is $\frac{m(3t_f)}{m(t_f)} = 3$.

What if now we rid of the assumption that $t_f = t_g = t_h$. There could be workers that work slower than other workers. At regime, the slowest worker will rule the system and determine the throughput of the entire system. All the other ones will have to comply with the slowest one. Suppose, for example, that the slowest worker is the second one. The first worker will have to wait for the second worker to send him data. The same holds for the third worker. Both the workers will have some idle time (time in which they do not anything).

In this case, if the number of bags to produce is m, and the number of

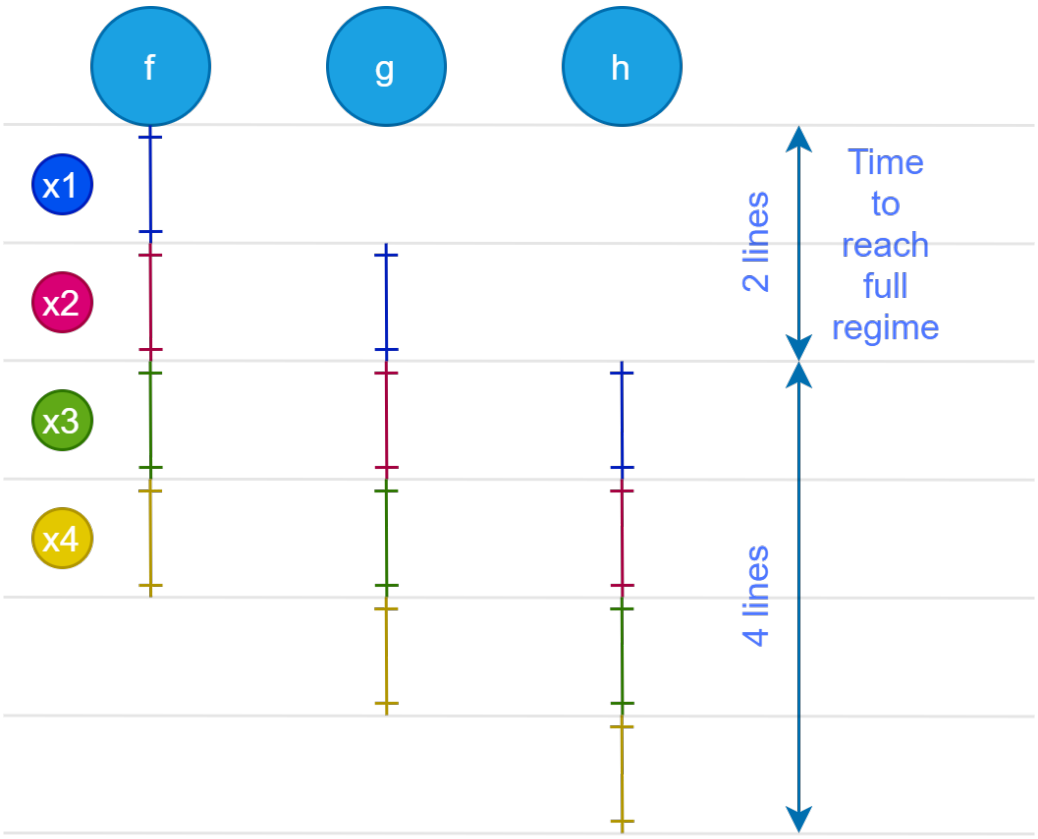


Figure 2.4: Example with 4 bags.

workers is n, then the assembly line will require:

$$T_{par} = \underbrace{(t_f + t_g + t_h)}_{\text{first steps}} + \underbrace{(m-1)\max\{t_f, t_g, t_h\}}_{\text{rest of the computation}} + some\ overhead$$

We call this kind of computation **stream parallel computation**: we have a stream of tasks that comes at the system. However, there is no point in time where all the tasks to be computed are available altogether. So, if task x_i is available at time t_i, the next task will be available at time $t_{i+1} > t_i$. The time that passes between t_i and t_{i+1} is called interarrival time.

Note that this process is not an embarrassingly process since the different workers must collaborate.

2.1.4 The bank

We are managing a bank with e employees that, on average, can serve a customer in t_e minutes. A client enters the bank every t_a minutes and exits from the bank once an employee serves him/her. So:

$$\begin{aligned} interarrival\ rate &= \frac{1}{t_a} \\ service\ rate &= \frac{1}{t_e} \times e \end{aligned} \tag{2.1}$$

The clients can proceed entering until the bank is not full (the buffer of the program becomes full).

Figure 2.4a shows the simplest case in which we don't consider the interarrival rate, but the customers arrive when the employees are ready to serve. In this case, the bank can serve two clients every t_e minutes, therefore, using formula 2.1, the service rate is $2 \times 1/t_e$.

If we also consider the interarrival rate, two things may happen:

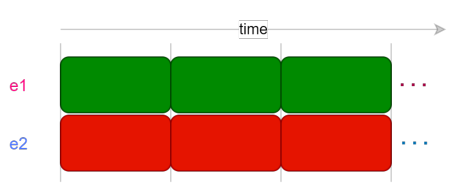
- The interarrival rate is sufficiently small for each one of the employees so that they can serve clients continuously as in figure 2.4b. The time spent to serve m clients is:

  $$\frac{m \times t_e}{e} + (e)t_a$$

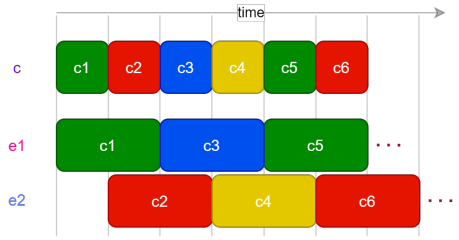
  For instance, in figure 2.4b two employees must serve $m = 6$ clients. To reach the steady state (full regime) the bank needs that $e = 2$ clients enter inside, and since clients arrive every $t_a = 1$ line, the time needed is $(e)t_a = 2 \times 1 = 2$ lines. Then, we need other $\frac{6 \times 2}{2} = 6$ lines to serve the clients

- If t_a is too small concerning the number of employees and their service rate, the bank queue will start growing up to the point when no more customers can enter the bank. Figure 2.5 exhibits exactly this case: the employees require 5 lines to serve one client, but the clients arrive every 1 line. Therefore the queue starts growing, and at a certain point, it will be full.

  We could add more employees. Ideally, we should match a situation where $t_a \approx t_e/e$. In our example, we reach the ideal situation with 5 employees as shown in figure 2.6.



(a) The bank employees execution table if customers arrive in couple every t_e



(b) The bank employees execution table if customers arrive every t_a.

Figure 2.4: Bank employee execution time.

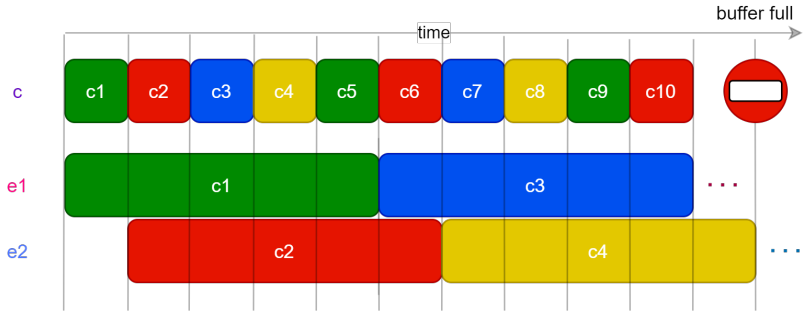


Figure 2.5: Case in which the bank employees are too slow to manage the customers entering the bank. At a certain point, the bank will become full (the program buffer will become full), and it will not accept new customers.

Let us slightly change the scenario:

- some employees spend more time serving a client than other employees.
- The clients are already in the bank and are waiting for their turn in a queue. There is one queue for each employee, and the customers are equally assigned to every queue.

Let us assume that we have 2 employees and that the first is faster than the second, which is $t_{e_1} < t_{e_2}$, then the first employee will complete its queue faster than the second employee. With m clients, we will assign $m/2$ clients to the first employee, and the others to the second one. The first employee will conclude in $t_{e_1} \times m/2$ minutes while the other will need $t_{e_2} \times m/2$ minutes. Figure 2.7 shows this situation with 8 customers.

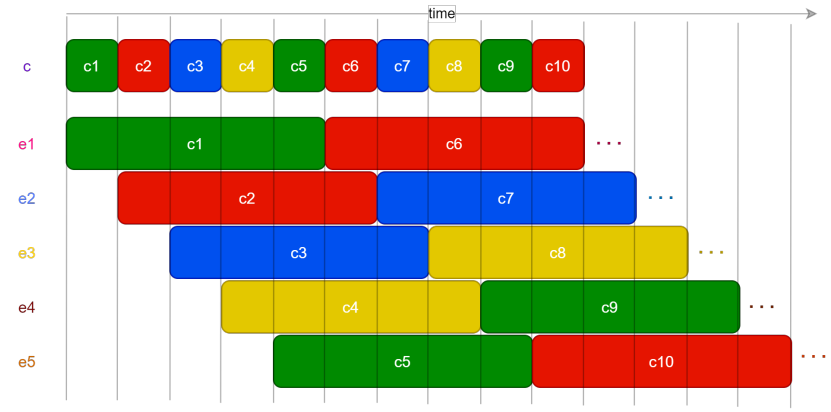


Figure 2.6: With four employees, the bank can handle the number of clients that are entering.

The smart thing to do could be to allow to move clients from a queue to another. In other words, the faster employee should "steal" the customers from the second employee.

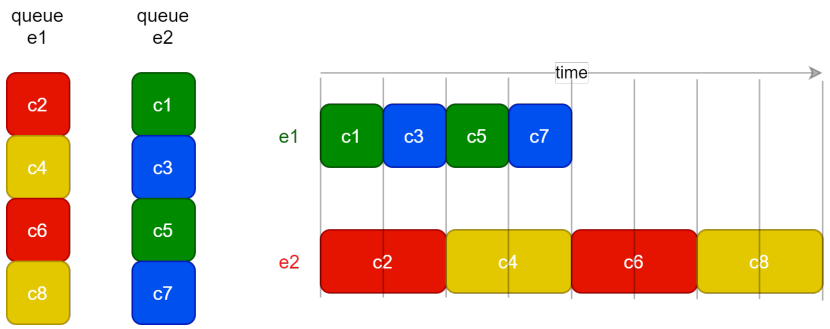


Figure 2.7: The clients are divided equally into two queues. The speediest employee concludes much earlier than the slowest one.

The bank implements the so-called **Farm pattern** that is a **stream parallel computation**. In the first scenario described, we need a scheduler that assigns the clients to the workers as in figure 2.8. The time needed to schedule clients, indicated with $t_{schedule}$, represents an overhead that we would not have in the sequential case.

In the second scenario, we have several queues, one for each employee. So, we don't need a scheduler, but we may reach a situation where we have to move one client from a queue to another queue, bringing some overhead.

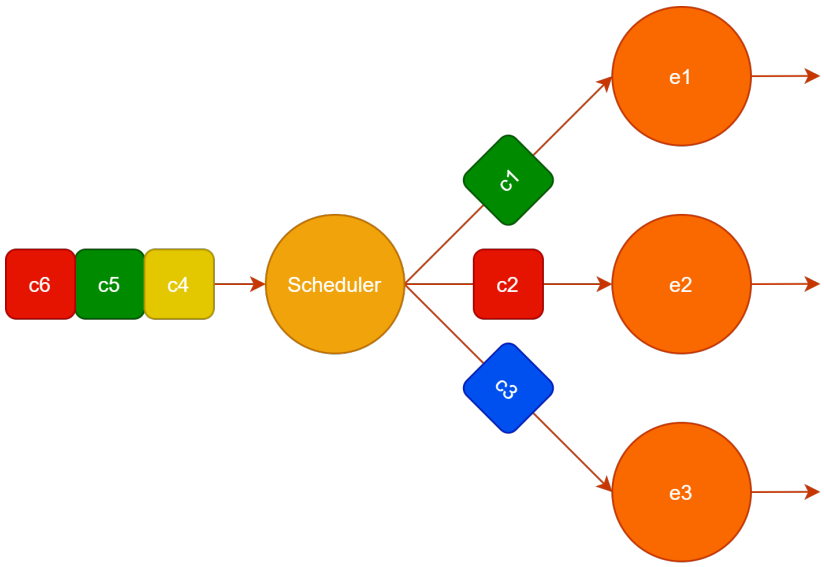


Figure 2.8: The scheduler assigns the customers to the employees. In this case, it assigns c1 to e1, c2 to e2 and c3 to e3.

2.2 Parallel patterns

In the previous section, we have seen several examples, each of which represents a different parallel computation pattern:

1. the "**Translate Book**" example is an example of **MAP** pattern. It is indeed a map of a function f: it takes a single data, it splits the data into partitions, and it produces data with the same structure as the original one, but where all the elements have been mapped. See figure 2.9.

2. The "**Counting wireless devices**" example is an example of **REDUCTION**: we take a collection of data, we split it into partitions, and then we summarize the partial results into one single result. This operation is usually indicated by the $\oplus$ symbol (called oplus): if we partitions the data into $x_1, x_2, \ldots, x_N$ parts, then the result of the REDUCTION will be $x_1 \oplus x_2 \oplus \ \ldots \ \oplus x_N$.

3. The "**Assembling conference bags**" example is a **PIPELINE** pattern: we have a stream of requests that appear at different time.

A fist worker compute $f_1(x_i)$ and delivers the result to the second worker, which will compute $f_2(f_1(x_i))$. The final worker computes $f_N(f_{N-1}(\dots f_1(x_i)\dots))$ and delivers the result to an output stream. See figure 2.10.

4. The "**bank**" example is a **FARM**: we have a stream of requests. Unlike the pipeline case where all the workers compute a different function, in the FARM pattern, all the workers compute the same function and deliver the results to an output stream. See figure 2.11.

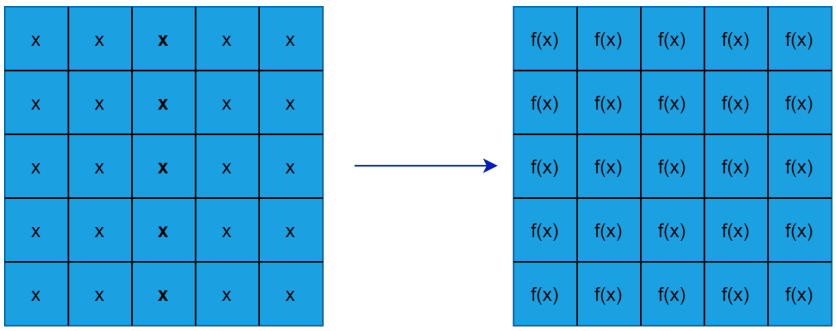


Figure 2.9: MAP pattern example. Note that we do not change the structure of the original data.

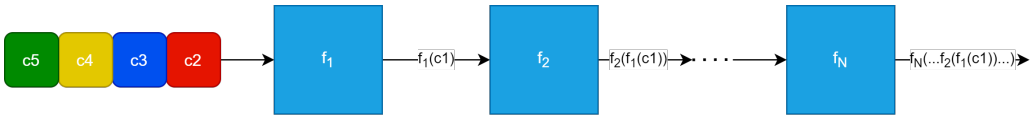


Figure 2.10: PIPELINE pattern example. We have one worker for each function.

The first two patterns are **data parallel** computations. The other two are **stream parallel** computations. In the **data parallel** computation, we try to reduce the time we spend to calculate on a single piece of data. In the **stream parallel** computation, we want to process more input data at a single time. The computation of the single item takes the same time we would have in the sequential case, but the system throughput increases.

Of course, we can combine different patterns to obtain better performances. Figure 2.12 shows an example where we combine the PIPELINE and the FARM patterns.

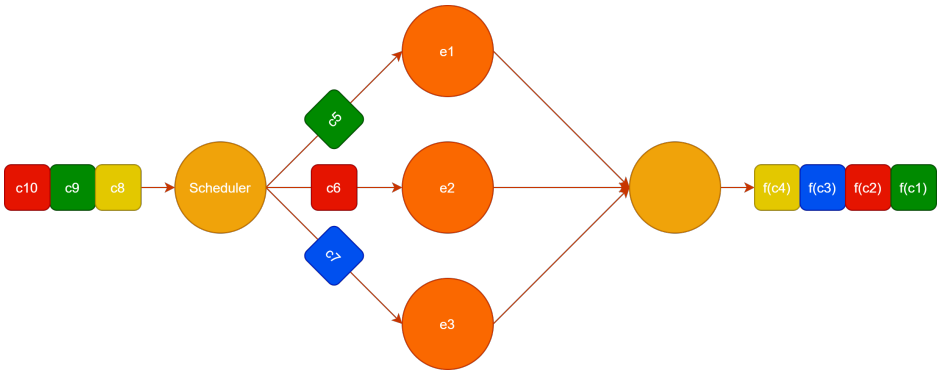


Figure 2.11: FARM pattern example.

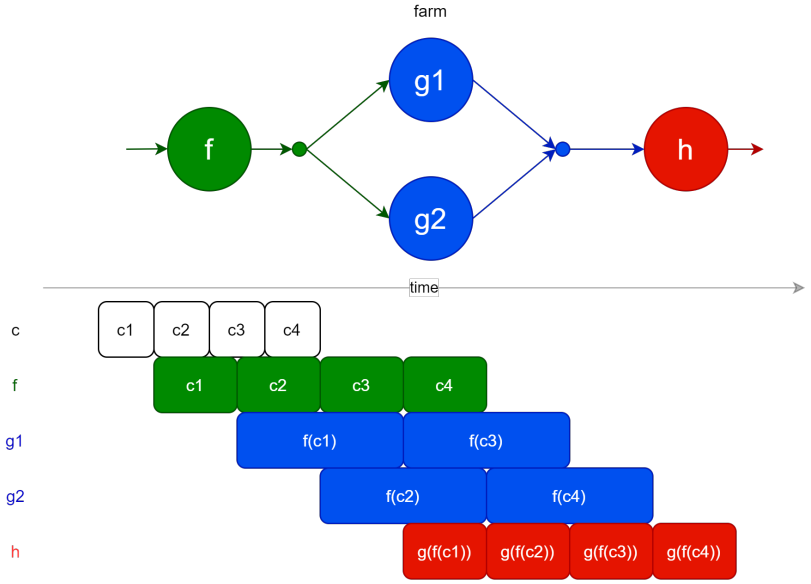


Figure 2.12: Inside a pipeline, we noticed that function g was longer to compute than f and g. Therefore, we create a farm of two workers to compute g.

Chapter 3

Measures

In this chapter we will present and describe the measures used to evaluate parallel computations.

3.1 Latency and Service time

Latency (L) the time spent between the moment a given activity receives input data and the moment the same activity delivers the output corresponding to the input.

Let us now consider what latency is in the four kinds of computation we have to deal with by writing some pseudo-code that calculates the latency. In the code, we will use the `gettimeofday()` function that returns the current time.
In the sequential and data-parallel cases, the latency is merely the time between the start and the end of the computation. In the data-parallel case, besides the `computation`, we also have to consider the overhead due to split and possibly merge the results.

Listing 3.1: Latency sequential case

```
t0 = gettimeofday()
computation
t1 = gettimeofday()
L = t1 - t0
```

Listing 3.2: Latency data parallel case

```
t0 = gettimeofday()
split
compute in parallel
merge
t1 = gettimeofday()
L = t1 - t0
```

In the pipeline case, the latency is the time needed for the data to pass through all the stages.

Listing 3.3: Latency pipeline case

```
t0 = gettimeofday()
compute stage 1 on item i
compute stage 2 ...
...
compute stage k ...
t1 = gettimeofday()
L = t1-t0
```

Finally, in the farm case, the latency is the time a worker needs to elaborate on a single piece of data.

Listing 3.4: Latency farm case

```
t0 = gettimeofday()
compute farm worker on item i
t1 = gettimeofday()
L = t1-t0
```

Service Time (T_s) the time intercurring between the delivery of two consecutive output items (or alternatively, the time in between accepting two consecutive input tasks to compute).

The service time has a sense only in **stream parallel** computations. Indeed, both in the sequential and in the data-parallel cases, latency and service time are equal. However, we may think of the data-parallel computation as a pipeline with three stages: 1) a worker that split the data, 2) a farm that

performs the main computations, 3) possibly a worker that merges the results. Figure 2.12 shows this situation taking f as the split worker, g1 and g2 as the farm workers, and finally h as the merge worker. In this case, and in general, in the stream parallel case, the service time is defined as follow:

$$T_s = \max\{T_{s_1}, T_{s_2}, \ldots, T_{s_N}\} \tag{3.1}$$

Where t_i is the time needed to terminate the i-th stage. So, it is the maximum service time among the service times of all the stages.

3.2 Derived Measures

The derived measures are measures that are function of the base measures and of the parallelism degree.

Throughput (B) is the amount of tasks computed per unit of time. In other words, it is the inverse of the service time:

$$B = \frac{1}{T_s} \tag{3.2}$$

Speedup $(s(n))$ the ratio between the *best known* sequential execution time (on the target architecture at hand) and the parallel execution time. Speedup is a function of n the parallelism degree of the parallel execution:

$$s(n) = \frac{best\ sequential\ time}{parallel\ time\ with\ n} = \frac{T_{seq}}{T_{par}(n)} \tag{3.3}$$

Scalability $(scalab(n))$ the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n:

$$scalab(n) = \frac{T_{par}(1)}{T_{par}(n)} \tag{3.4}$$

So, the speedup tells us how much we improve with respect to the sequential state-of-the-art. On the other hand, scalability tells us how much we improve the performance by increasing the parallelism degree. We don't mind about how much is the time achieved using just one thread, we mind if we can

double the performance whether we use x threads and then we use $2 \times x$ threads.

We are usually not interested in scalability since it does not consider the sequential program. Indeed, we could have ideal scalability, but our program may always require more time than the sequential one.

Speedup and scalability can both be defined in terms of latency and service time. In data-parallel computations, we will be more interested in speedup in latency. In the stream parallel computations, we will be more interested in speedup in service time.

Efficiency(n) (ϵ) measure of the capacity to work with the available resources. It is the ratio between the ideal execution time and actual execution time:

$$\epsilon(n) = \frac{T_{id}(n)}{T_{par}(n)} \tag{3.5}$$

Ideal execution time ($T_{id}(n)$)

$$T_{id}(n) = \frac{T_{seq}}{n} \tag{3.6}$$

Using 3.6 in 3.5, we can alternatively express efficiency as:

$$\epsilon(n) = \frac{T_{seq}}{nT_{par}(n)} = \frac{sp(n)}{n} \tag{3.7}$$

The maximum efficiency value is **1**. However, it is impossible to reach 1 due to the overhead: the less is the overhead, the more we are close to 1. We can indeed rewrite 3.7 also considering the overhead:

$$\epsilon(n) = \frac{T_{id}}{\frac{T_{seq}}{n} + overhead} = \frac{T_{id}}{T_{id} + overhead} \tag{3.8}$$

Usually, we are satisfied when the efficiency is above 90%.

Note that the sequential implementation has $\epsilon(1) = 1$:

$$\epsilon(1) = \frac{T_{seq}}{1T_{seq}} = 1$$

3.3 Completion time

Completion time (T_c) time from the beginning of the first input to the end of the last output.

As you can notice from the definition, the completion time refers to a stream parallel computation. In the **pipeline** case, the completion time is:

$$T_c = \underbrace{\sum L_i}_{\text{first steps}} + \underbrace{(m-1)\max\{L_i\}}_{\text{rest of the computation}} \tag{3.9}$$

if we consider the pipeline pictured in figure 2.5, the completion time is 8 "lines": 2 "lines" to reach the steady state (full regime) and other 6 "lines" to terminate.
Using the formula to compute the service time in the parallel stream case (3.1), formula 3.9 is equal to:

$$T_c = \sum L_i + (m-1)T_s \tag{3.10}$$

We can eventually assume that $\sum L_i$ is negligible when m is very high, obtaining:

$$T_c \approx mT_s \tag{3.11}$$

We will now see that the same approximation holds for the **farm** case. As figure 3.1 displays, in the farm case, we have an emitter that sends data to the farm workers with a service time of t_e, a set of n_w workers that complete a task in time t_w, and lastly, a collector that gather the results and delivers them to an output stream with a service time of t_c. So, the completion time is:

$$T_c = \underbrace{n_w \cdot t_e}_{\text{to reach steady state}} + \underbrace{\frac{m}{n_w} \cdot t_w}_{\text{to compute all the tasks}} + \underbrace{t_c}_{\text{to send the last result}} \tag{3.12}$$

For example, in figure 3.1, $n_w = 2$, $m = 6$, $t_e = 1$ "line", $t_w = 2$ "lines" and $t_w = 1$ "line". Therefore, the completion time is:

$$2 \cdot 1 + \frac{6}{2} \cdot 2 + 1 = 2 + 6 + 1 = 9 \text{ "}lines\text{"}$$

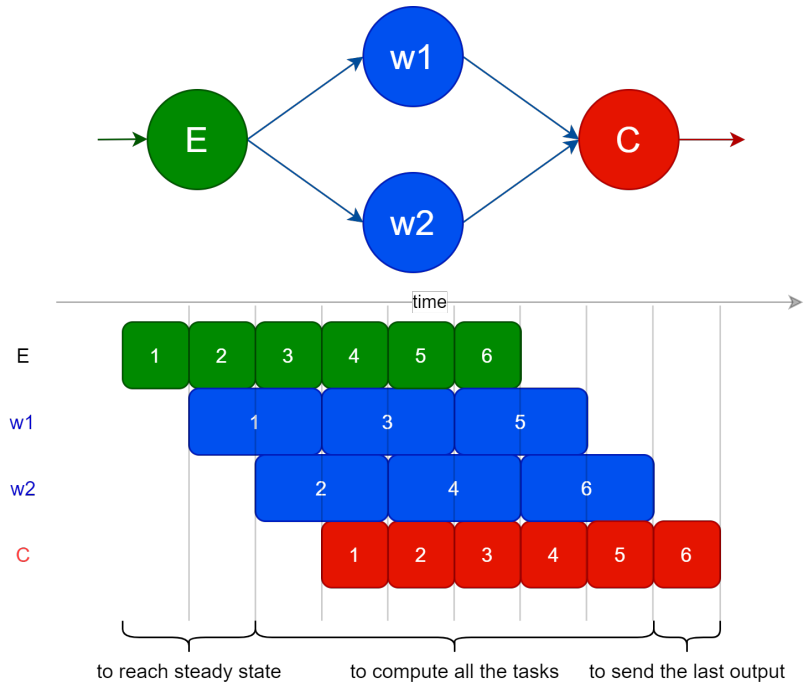


Figure 3.1: The picture shows the execution table of a farm with two workers. We spend 2 "lines" to reach the state where all the farm workers do something, others 6 "lines" to complete all the tasks, and finally 1 "line" to deliver the last result to the output stream.

Using again the service time definition (3.1), we have that:

$$T_s = \max\{t_e, \frac{t_w}{n_w}, t_c\} \tag{3.13}$$

Assuming that t_e and t_c are negligible concerning $\frac{t_w}{n_w}$:

$$T_s = \frac{t_w}{n_w}$$

And therefore:

$$T_c \approx mT_s$$

3.3.1 Maximum Parallelism Degree

Notice that the more is n_w, the more is the time needed by the emitter to schedule the tasks. So, increasing the n_w, the completion time will decrease

up to a point where $\max\{t_e, \frac{t_w}{n_w}, t_c\} = t_e$. From that point on, if we increase the number of workers, we will not get benefits since the service time will not decrease any more. So, we can increase the number of workers BUT not infinitely. Figure 3.2 should clarify this fact.

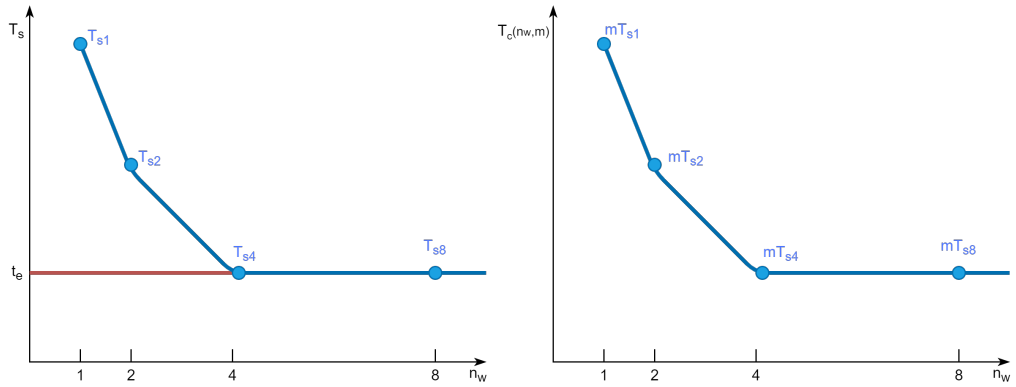


Figure 3.2: When t_e becomes larger than $\frac{t_w}{n_w}$, there is no sense if we increase the number of workers. In this case, once we use $n_w = 4$ workers, it does not fetch any benefit increase any more the number of workers because of $t_e \approx \frac{t_w}{n_w}$

.

On the other hand, in the pipeline case, we cannot increase the parallelism degree as much as we want. Indeed, if the pipeline has k stages, the maximum parallelism degree will be exactly k. Of course, as already explained, we can have further speedups by inserting farms when we have stages that require more time than other stages.

Examples

Suppose we have a pipeline with 3 stages: s_1, s_2 and s_3. As we said, the maximum parallelism degree we can achieve is 3, and therefore, the maximum speedup will be 3.

But, we can have for both s_1, s_2 and s_3 a farm of respectively n_1, n_2 and n_3 workers. In this way, we increase the parallelism degree to $n_1 + n_2 + n_3$ and the maximum speedup to $n_1 + n_2 + n_3$. Assuming that every farm has its own emitter and collector, the service time of the i-th farm will be:

$$T_s = \max\{t_{e_i}, \frac{t_i}{n_i}, t_{c_i}\}$$

So, in order to have $T_s = \frac{t_i}{n_i}$, we have to choose:

$$n_i \approx \left\lceil \frac{t_i}{\max\{t_{e_i}, t_{c_i}\}} \right\rceil$$

The problem of this implementation is that we have many nodes performing scheduling and gathering the results: one for each farm. We could summarize the communication in the middle with one single node. If we are very clever, we could also rid of the entities in the middle and use, for instance, a passive data structure.

The completion time (3.11) of the final solution is:

$$T_c = m \cdot \max\{\frac{t_1}{n_1}, \frac{t_2}{n_2}, \frac{t_3}{n_3}\}$$

An alternative solution is executing in parallel several sequential pipelines as shown in figure 3.3. The completion time will be:

$$T_c = m \cdot \frac{t_1 + t_2 + t_3}{n}$$

The two solutions will have the same completion time if:

$$\frac{t_1 + t_2 + t_3}{n} = \max\{\frac{t_1}{n_1}, \frac{t_2}{n_2}, \frac{t_3}{n_3}\}$$

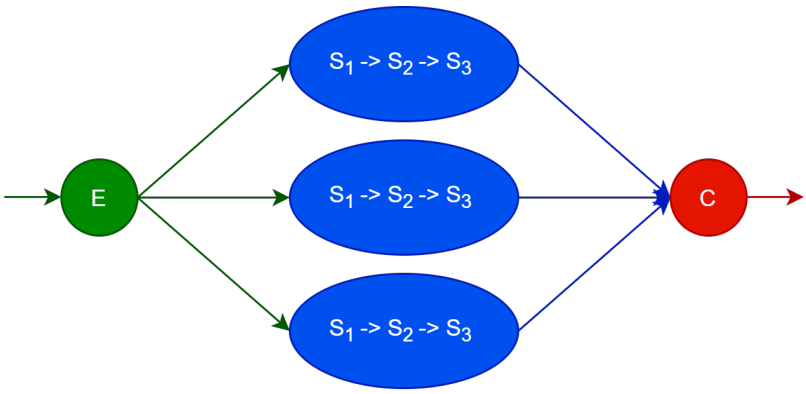


Figure 3.3: Pipelines executed in parallel.

3.4 Amdahl Law

The Amdahl law defines an upper bound to the speedup we can achieve in parallel computation.

The idea of the law is that we can often divide the computation into two portions (see also figure 3.4):

the **serial fraction** that is the part that is necessarily sequential.

the **non-serial fraction** that is the part we can parallelize.

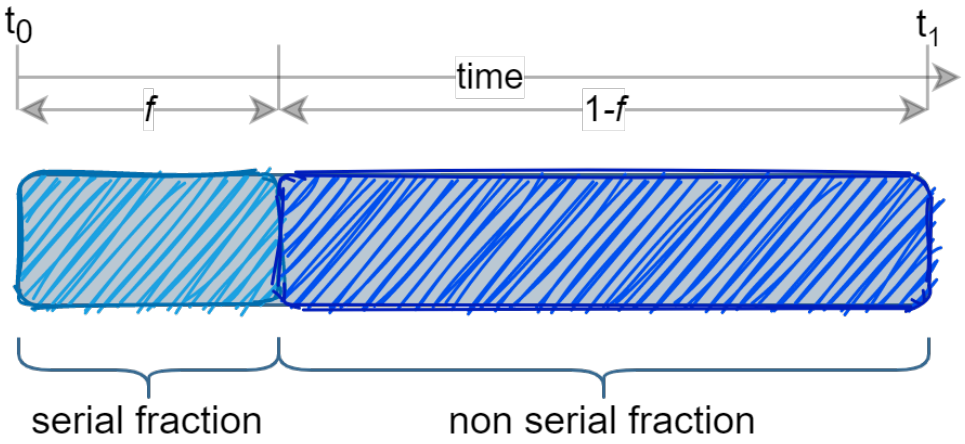


Figure 3.4: The initial part of the computation is necessarily sequential. For instance, we have always to initialize the threads or the data structures. Once we have finished these preliminary computations, we can do things in parallel

Since the non-serial fraction is parallelizable, we can split the computation into more and more parts up to the point that the time required by the non-serial fraction is insignificant concerning the serial fraction, as shown in figure 3.5. This lead us conclude that:

$$\lim_{n\to\infty} T_c = time\ serial\ fraction \tag{3.14}$$

Recalling the definition of speedup, we have that:

$$\lim_{n\to\infty} sp(n) = \lim_{n\to\infty} \frac{T_seq}{T_par(n)}$$

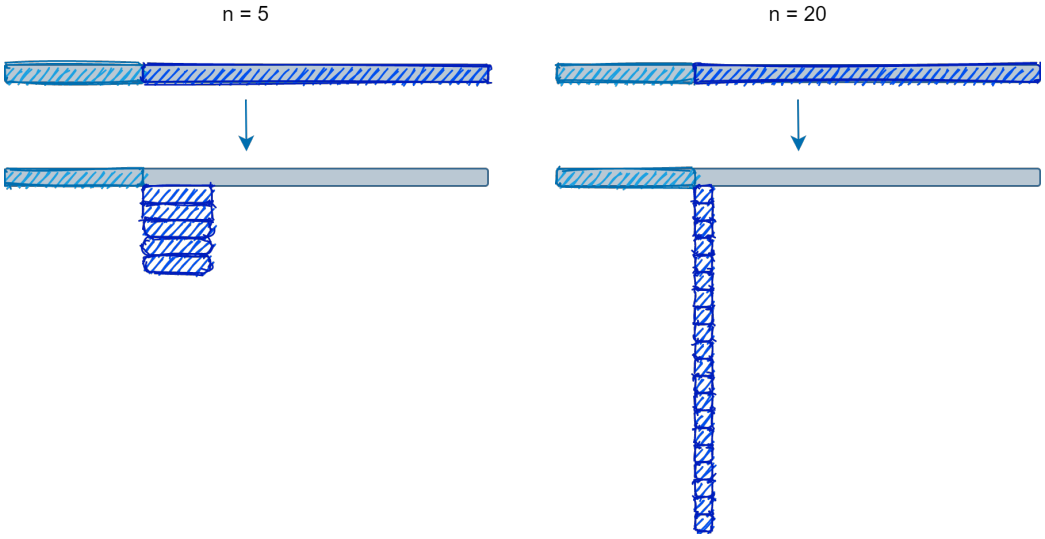


Figure 3.5: As we increase the number of workers (n), the serial fraction always remains the same while the non-serial fraction becomes smaller and smaller.

That, using the concept of the division of the program in serial and non-serial fraction becomes:

$$\lim_{n\to\infty} \frac{T_seq}{fT_{seq} + \underbrace{\frac{(1-f)T_{seq}}{n}}_{=0}} = \lim_{n\to\infty} \frac{T_{seq}}{fT_{seq}} \tag{3.15}$$

Where f is the percentage of the serial fraction in the computation, while $1-f$ is the percentage of the non-serial fraction (see figure 3.4). Taking 3.15, we can finally state the **Amdahl law**:

$$\lim_{n\to\infty} sp(n) = \frac{1}{f} \tag{3.16}$$

The Amdahl law is important and, at the same time negative. Indeed, suppose that our computation lasts 10 hours and that the sequential fraction is 1 minute, then the maximum speedup we can obtain according to this law is $sp(n) = 600$. Luckily there are ways to surpass the law. For instance, Gustafson introduces another law that tries to consider the problem from a different point of view.

3.5 Gustafson Law

Let us take a data-parallel computation as mapping the values of a vector. The steps needed to make the computation parallel would be:

1. Splits the array into n parts to assign to the thread. In this case, we have just to compute the ranges in which each thread has to work.
2. Set up thread assigning to each thread one range.
3. Leave the thread working.
4. Wait them finish.

Steps 1., 2. and 4. are part of the so-called serial fraction mentioned in the previous section, while step 3. is part of the non-serial fraction. Steps 1., 2. and 4. are independent of the amount of data, so by increasing the data, we do not increase the cost of these steps. On the other hand, step 3. depends on the data: the larger is the data, the larger is the number of things the threads have to do. Therefore, increasing and increasing the amount of data, the cost of steps 1., 2. and 4. will be negligible concerning the cost of the third step. So, as we increase the data, we increase the speedup. Note that, of course, the cost of steps 1., 2. and 4. will increase as we increase the parallelism degree.

There are problems where we can't increase the amount of data we want, but in other problems, that is possible. For instance, in the weather forecast, we can split an area into smaller and smaller grids. Each grid contains sensors measuring temperature, pressure, humidity, winds, and so on. The smaller is the grid, the better will be the forecast. So, using data-parallel computation, in this case, is like taking two birds in one stone: we increase the quality of the forecast, and at the same time, we raise the speedup over the limits imposed by the Amdahl law.

Chapter 4

Laboratory 1

4.1 Measure Time

Time is a relevant measure we have to consider when assessing the performance of a program. You will notice that your program will spend almost always a different amount of time to terminate. That happens because your program is not the only one that is running: there are plenty of background activities or programs you have installed that may increase the running time. These differences may be larger when dealing with small programs (with small execution time).
Knowing this, we can follow two simple rules of thumb that should allow us to overpass this problem:

- Run the executable several times and then take the average.
- Run the executable several times, discard the outliers and then take the average of the rest. An outlier is a time that is too large or too small.
- Run the executable several times, and always take the best one time. The previous two methods are preferable.

Of course, every execution must use the same parameters.
Usually, we do these tests using the so-called **Dummy inputs** such as random vectors. In the case of a random vector, we frequently have a function that initializes the random sequence given a **seed**, and another function that

returns, one at a time, the elements of the random series. In this case, one of the parameters should be the **seed**. Another parameter that will usually be in our programs is the parallel degree, which we commonly put as the final parameter.

The most simple way to measure the execution time of our program is by the command-line instructions. In Unix, we can use the following instruction to measure the execution time of the executable a.out:

```
time ./a.out params
```

However, this solution has two serious problems:

- It gives times in milliseconds, but we will usually need better resolution timers.
- It takes into account the whole time spent, including loading the code, start the code, and the end up code.

A better solution is to use timers within the code. In C++ we can use the standard library **chrono**:

```
#include <chrono>
#include <iostream>
#include <vector>

using namespace std;


int main(){
    // record start time
    auto start = chrono::system_clock::now();

    // initialize vector
    vector<int> vect(10000);
    //DO things with the vector...

    // record end time
    auto end = chrono::system_clock::now();

    //calculate difference
```

```
    chrono::duration<double> diff = end-start;

    //output difference counting milliseconds in diff
    cout << "Time to do something "
         << chrono::duration_cast<chrono::milliseconds>(diff).count()
         << " ms\n";
    return 0;
}
```

Instead of writing every time these lines of code, we often write some MACROs that then we use inside the code:

```
#define START(timename) auto timename = chrono::system_clock::now();
#define STOP(timename,elapsed) auto elapsed = \
chrono::duration_cast<chrono::microseconds>\
(chrono::system_clock::now() - timename).count();
```

Using MACROs, the previous code becomes:

```
#include <chrono>
#include <iostream>
#include <vector>

using namespace std;

#define START(timename) auto timename = chrono::system_clock::now();
#define STOP(timename,elapsed) auto elapsed = \
chrono::duration_cast<chrono::microseconds>\
(chrono::system_clock::now() - timename).count();

int main(){
    // record start time
    START(start)

    // initialize vector
    vector<int> vect(10000);
    //DO things with the vector...

    // record end time
```

```
    auto end = chrono::system_clock::now();

    //calculate difference
    STOP(end, tot)

    //output tot
    cout << "Time to do something "
         << tot
         << " ms\n";
    return 0;
}
```

That is much more cleaner.

The last approach we present follows the RAII (Resource acquisition is initialization) idiom: we can imagine using the creation end destruction of an object to respectively get the start and end times. So, we can create a class called **Utimer** defined as follow:

```
1  class utimer
2  {
3      std::chrono::system_clock::time_point start;
4      std::chrono::system_clock::time_point stop;
5      std::string message;
6      using usecs = std::chrono::microseconds;
7      using msecs = std::chrono::milliseconds;
8
9  private:
10     long *us_elapsed;
11
12 public:
13     utimer(const std::string m) : message(m), us_elapsed((long *)NULL){
14         start = std::chrono::system_clock::now();
15     }
16     utimer(const std::string m, long *us) : message(m), us_elapsed(us){
17         start = std::chrono::system_clock::now();
18     }
19     ~utimer(){
20         stop =
21             std::chrono::system_clock::now();
```

```
22          std::chrono::duration<double> elapsed =
23              stop - start;
24          auto musec =
25              std::chrono::duration_cast<std::chrono::microseconds>
26                  (elapsed).count();
27          std::cout << message << " computed in " << musec << " usec "
28                    << std::endl;
29          if (us_elapsed != NULL)
30              (*us_elapsed) = musec;
31      }
32  };
```

Note that the class constructors memorize the moment in which the object is created in the `start` variable (line 14 and line 17). On the other hand, the class deconstructor (line 19) , which is called when the object is destroyed, stores in the variable `end` the current moment and finally prints the elapsed time between the variables `end` and `start`. It possibly stores the result in the variable `u_elapsed`.
Using Utimer, our example can be written as follow:

```
#include <chrono>
#include <iostream>
#include <vector>

using namespace std;

#include "utimer.cpp"

int main(){
    // record start time
    {
        utimer t("msg");

        // initialize vector
        vector<int> vect(10000);
        //DO things with the vector...

    }
```

```
    return 0;
}
```

We insert the code we want to test into a code block in which we create an Utimer object. Once the system executes all the instructions in the code block, the object prints the elapsed time, and it is destroyed.
We can then execute the program several times using, for example, a bunch of bash instructions:

```
for((i=0; i<10; i++)); do ./example params; done
| grep msg
| awk '{print $4$}'
```

4.2 Threads

A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system and multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.
In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time. Threads are an independent flow of control that shares the memory space with other threads. We can create a thread by including the `thread` package, and then by calling the thread constructor:

```
auto thread = new thread(f, p1, p2, ..., pN)
```

where `f` is the function the thread will execute and `p1, p2, ..., pN` are the arguments to pass to the new function.

As written in the definition, threads share a memory space that consists of global variables or data structures. The access to those elements must be **synchronized**. Indeed, let us assume that we want to sum all the entries of an array. We first write a sequential program and then a parallel program with two threads that share a `sum` variable and that update as they proceed in scanning the partition of array associated with them. In the end, we print the result of the sequential program and the parallel one. The code is as follow:

```cpp
#include <iostream>
#include <vector>
#include <functional>
#include <thread>

using namespace std;

int sum = 0;

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

    int n = atoi(argv[1]);
    int seed = atoi(argv[2]);

    const int k = 128;

    vector<int> x(n);

    srand(seed);
    for (int i = 0; i < n; i++)
        x[i] = rand() % k;

    int seqsum = 0;
    for (int i = 0; i < n; i++)
        seqsum += x[i];

    cout << "Seq sum " << seqsum << endl;

    auto f = [&](int start, int stop) {
        for (int i = start; i < stop; i++)
            sum += x[i];
    };

    auto tid1 = new thread(f, 0, n / 2);
    auto tid2 = new thread(f, n / 2, n);
    tid1->join();
    tid2->join();
```

```
39      cout << "Par sum " << sum << endl;
40
41      return (0);
```

The sequential calculation happens in lines `23-25`, while the parallel one in lines `29-37`. In `37-38` we wait for the termination of the two threads we have created in lines `35-36`. We should always call the `join` function once we generate a thread. The output of the program with large vector dimensions will always outline that the results of the sequential computation are larger than the parallel computation:

```
./example 100000 123
Seq sum 6362256
Pas sum 3376209
./example 100000 123
Seq sum 6362256
Pas sum 4022227
```

Note that we always use 123 as seed and that parallel computation results change at every execution.

4.2.1 Synchronization

Atomic variables

When we have to synchronize the access to a single variable, we can merely define that as atomic. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined. We can use the atomic object only for a few bunches of types and operations such as the `+=` or the `++`. See the documentation for further information.

So, taking the previous code, we include the `atomic` package, then we change line `8` by writing `atomic<int> sum;` and finally, before the thread initialization, we initialize the sum value with `sum = 0`.

```
1  #include <iostream>
2  #include <vector>
3  #include <functional>
4  #include <thread>
5  #include <atomic>
6
```

```
7  using namespace std;
8
9  atomic<int> sum;
10
11 int main(int argc, char *argv[])
12 {
13
14     int n = atoi(argv[1]);
15     int seed = atoi(argv[2]);
16
17     const int k = 128;
18
19     vector<int> x(n);
20
21     srand(seed);
22     for (int i = 0; i < n; i++)
23         x[i] = rand() % k;
24
25     int seqsum = 0;
26     for (int i = 0; i < n; i++)
27         seqsum += x[i];
28
29     cout << "Seq sum " << seqsum << endl;
30
31     auto f = [&](int start, int stop) {
32         for (int i = start; i < stop; i++)
33             sum += x[i];
34     };
35
36     sum = 0;
37     auto tid1 = new thread(f, 0, n / 2);
38     auto tid2 = new thread(f, n / 2, n);
39     tid1->join();
40     tid2->join();
41     cout << "Par sum " << sum << endl;
42
43     return (0);
```

Lock (Mutex)

When we cannot use atomic variables, we can use locks (also called mutexes). So if we have a critical section, a segment in the code where threads can access shared variables, we can define a mutex and add a `mutex.lock()` before the critical section and a `mutex.unlock()` after. During the program execution, only one thread can own the key to unlocking `mutex.lock()` at a time. So, only one thread at a time can access the critical section. All the others must wait, **actively**, for that thread to unlock the key.
Let us look at the code that uses mutexes:

```
#include <iostream>
#include <vector>
#include <functional>
#include <thread>

using namespace std;

int sum = 0;
mutex mu;

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

    int n = atoi(argv[1]);
    int seed = atoi(argv[2]);

    const int k = 128;

    vector<int> x(n);

    srand(seed);
    for (int i = 0; i < n; i++)
        x[i] = rand() % k;

    int seqsum = 0;
    for (int i = 0; i < n; i++)
        seqsum += x[i];

```

```
29      cout << "Seq sum " << seqsum << endl;
30
31      auto f = [&](int start, int stop) {
32               for(int i=start; i<stop; i++) {
33                  mu.lock();
34                  sum += x[i];
35                  mu.unlock();
36               }
37      };
38
39      auto tid1 = new thread(f, 0, n / 2);
40      auto tid2 = new thread(f, n / 2, n);
41      tid1->join();
42      tid2->join();
43      cout << "Par sum " << sum << endl;
44
45      return (0);
```

The mutexes are in the `thread` package, so we do not need to import other packages. We defined the mutex at line `9`, and inserted the `mu.lock()` at line `33` and `mu.unlock()` at line `35`, so that the update of the variable `sum` is protected by the multiple access of threads.

In this example, the best solution was to use an atomic variable since it is notably quicker. Indeed, the code with the atomic objects terminates on average in 3800 microseconds while the mutex version finishes on average in 11000 microseconds.

Instead of the standard mutex, which is very prone to errors, we could use something that makes our life simpler that is the **lock_guard**. Lock_guards use an IRII style mechanism. As for the Utimer, we merely have to insert the critical section into curly brackets and then define at the beginning of the block code a lock_guard. We still have to define a mutex object and pass it to the lock_guard object when we create it. The mutex is unlocked as soon as the object goes out of scope and is destroyed. Time may be larger than the standard lock. However, it is much easier to avoid stupid errors.
Lock_guards are in the `mutex` package. So, we have to import that package, then define the mutex, and finally create the lock_guard object into the critical section:

```
1  #include <iostream>
```

```
2  #include <vector>
3  #include <functional>
4  #include <thread>
5  #include <mutex>
6
7  using namespace std;
8
9  int sum = 0;
10 mutex mu;
11
12 int main(int argc, char *argv[])
13 {
14
15     int n = atoi(argv[1]);
16     int seed = atoi(argv[2]);
17
18     const int k = 128;
19
20     vector<int> x(n);
21
22     srand(seed);
23     for (int i = 0; i < n; i++)
24         x[i] = rand() % k;
25
26     int seqsum = 0;
27     for (int i = 0; i < n; i++)
28         seqsum += x[i];
29
30     cout << "Seq sum " << seqsum << endl;
31
32     auto f = [&](int start, int stop) {
33              for(int i=start; i<stop; i++) {
34                 lock_guard<mutex> lg(mu);
35                 sum += x[i];
36              }
37     };
38
39     auto tid1 = new thread(f, 0, n / 2);
```

```
40      auto tid2 = new thread(f, n / 2, n);
41      tid1->join();
42      tid2->join();
43      cout << "Par sum " << sum << endl;
44
45      return (0);
```

Condition variables

The time required by every mutex is high because they demand an active wait of the thread. There is another mechanism of the **condition variables** that uses locks but provides a different pattern that allows passive wait. Every condition variable has three methods:

- **Wait**: the thread that calls the wait atomically releases the mutex and is descheduled. The thread reacquires the lock when it is scheduled again. A thread can invoke the wait only if it owns the lock.
- **Signal**: the thread awakens one of the threads waiting in the waiting list of the condition variable.
- **Broadcast**: the thread awakens all the threads waiting in the waiting list of the condition variable.

In C++, the corresponding methods are:

- `cv.wait(lock, predicate)`, where the first argument is a lock that the thread must own, and the second argument is the predicate that returns false if the waiting must continue. It is like to write:

```
while (!pred()) {
    wait(lock);
}
```

- `cv.notify_one()` unblocks one of the waiting threads.
- `cv.notify_all()` unblocks all the waiting threads.

To use condition variables, you need to include the `condition_variables` package.

Chapter 5

Stream and Data Parallel Examples

5.1 Pipeline optimization

In chapter 3, we discussed we could increase the performances of a pipeline parallelizing each stage. This section will treat the argument in a more deeply manner. Let us make two hypotheses on the computation the pipeline must execute:

Stages are independent of each other.

Stages compute functions, and so have not internal state.

The optimization of the pipeline consists of two steps:

1. Look for possible bottlenecks. A bottleneck is a stage that is slow with respect to the other stages.

2. Try to remove the bottleneck by for instance inserting a farm.

Let us consider a pipeline with k stages with a bottleneck in the jth stage. The jth stage has thereby a much higher latency than the other stages:

$$L_j = \max\{L_1, L_2, \ldots, L_k\} \tag{5.1}$$

Our goal is to make stage j working such that the real service time T_s is equal to the service time T_{s_j}, we would get with a pipeline without stage j:

$$T_{s_j} = \max\{L_i | i \in [1, k] \wedge i \neq j\} \tag{5.2}$$

Because we are in a stateless computation, we can turn stage j into a farm. To reach our goal, the farm must have as many workers as needed to make T_s equal or nearby to T_{s_j}. Using formula (3.13) that defines the service time of a farm and assuming that the emission and collection time is negligible, the service time of the stage j is equal to:

$$T_{s_{farm}} = \max\{t_e, \frac{t_{w_j}}{n_{w_j}}, t_c\} = \frac{t_{w_j}}{n_{w_j}} = \frac{L_j}{n_{w_j}}$$

So, n_{w_j} should be such that:

$$\begin{aligned} \frac{L_j}{n_{w_j}} = T_{s_j} \quad &\leftrightarrow \quad L_j = T_{s_j} n_{w_j} \\ &\leftrightarrow \quad n_{w_j} = \frac{L_j}{T_{s_j}} \end{aligned} \tag{5.3}$$

Consider a pipeline with three stages with latency 3, 10, and 5 seconds. The second stage represents the bottleneck of the computation, so we parallelize it employing a farm. Following the same steps we have done in the general case, we first measure the service time without the second stage:

$$T_{s_2} = \max\{3, 5\} = 5$$

Using 5.3, and assuming that t_e and t_c are negligible, the number of workers for which the pipeline has an actual service time equal to T_s is:

$$n_{w_2} = \frac{10}{5} = 2$$

The service time is indeed now equal to 5:

$$T_s = \max\{3, \max\{t_e, t_c, \frac{10}{2}\}, 5\} = 5$$

We can apply the same reasoning another time, adding a worker to the second stage to reduce the time to 3.3 seconds and employing a farm of two workers for the third stage, reaching 2.5 seconds. So, the service time becomes:

$$T_s = \max\{3, \max\{t_e, t_c, 3.3\}, \max\{t_e, t_c, 2.5\}\} = 3.3 < 5$$

The extreme limit of this reasoning is that at a certain point increasing the number of workers, the time needed by the emitter or the consumer becomes higher than $\frac{t_w}{n_w}$.

$$\max\{t_e, t_c, \frac{t_w}{n_w}\} = t_e$$

Where we supposed that the maximum is t_e. Of course, it could have also been t_c. If we use a farm for every stage of the pipeline, and we reach this limit, even the service time will be t_e:

$$\begin{aligned} T_s &= \max\{farm(s_1, n_{w_1}), farm(s_2, n_{w_2}), farm(s_3, n_{w_3})\} \\ &= \max\{t_e, t_e, t_e\} \\ &= t_e \end{aligned} \tag{5.4}$$

Note that we have not been concerned about other aspects that could decrease the performance of the program.

5.2 GrPPI and Stream data pattern

GrPPI is an open-source parallel pattern programming interface developed at University Carlo III of Madrid. From the GitHub repository (GitHub repository), you can download the GrPPI code. Then, you can use GrPPI as a standard library, or as a library that you link by indicating the path with the -I flag in the compilation command. To define a pipeline using GrPPI, we must have in mind two concepts:

- Stages are standard C++ functions.
- Streams are modeled through optional types (`optional<>`) because GrPPI uses the {} as the EOF of the pipeline, which indicates the end of the tasks.

For instance, let us suppose we want to build a pipeline with four stages, of which the intermediate ones execs simple manipulations to the number they receive. The first stage awaits 10 milliseconds and then increases the number of one:

```
int inc(int i) {
  activewait(10ms);
  return(i+1);
}
```

The second stage awaits 50 milliseconds and then doubles up the number:

```
int twice(int i) {
  activewait(50ms);
  return(i+i);
}
```

Inside the main, we initially define the source function that will generate the number from 0 up to `m-1`. Once it has been generating `m` numbers, it sends the {} as EOF. Note that the value returned by the source is `optional<int>` , and that is why {} is a legal value.

```
auto source =
    [&m] () -> std::optional<int> {
                                    static int n = 0;
                                    if(n == m)
                                      return {};
                                    else
                                      return{n++};
        };
```

On the other side, the drain will print the number it receives:

```
  auto drain =
    [] (int n) {
      cout << "Received " << n << endl;
    };
```

At this point, we can finally write the GrPPI code that builds the pipeline:

```
    pipeline(parallel_execution_native{nw},
             source,
             inc,
             farm(5,twice),
             drain);
```

Where the first argument indicates the back-end that will be used during the computations. `parallel_execution_native{nw}` executes the pipeline in parallel using `nw` workers. We could also have written `sequential_execution{}` to have a sequential execution. Note also that we speed up the third stage using a farm of five workers.

Using GrPPI and Utimers, we can see if the performance models defined in the previous chapters for pipelines and farms have a basis of reality. We

can notice that using the configuration written above the total execution time with $m = 100$ and `nw` $= 8$, will be 1184 msec that is approximately what 3.11 would return: $T_c \approx mT_s = 100 \times 10 = 1000$ msec. Recall that the service time in a pipeline is equal to the highest service time among its stages (3.1). In our example, the first and last stages have negligible service time, the second stage has $T_s = 10$ while the third has $T_s = \max\{t_c, t_e, \frac{L_3}{n_w} = \frac{50}{5} = 10\}$, that assuming that t_e and t_c are negligible, is 10. So, the service time of the pipeline is 10. Indeed, we have chosen five workers for the third stage so that both the intermediate stages have the same service time.

5.3 MAP pattern library examples

The MAP pattern is the one explained in the book translation example where we apply the same function to each element in the collection. Since MAP was the first kind of parallelism ever thought, it is now very straightforward to use it.

For instance, using OpenMP, we plainly have to write a `pragma` extension just before the for which would exec the map sequentially:

```
#pragma omp parallel for num_threads(nw)
for (int i=0; i<n; i++){
    vect[i] = 5 + vect[i];
}
```

Note that OpenMP is compiler-based, so we need to slightly change the compilation command by adding `-fopenmp`:

```
g++ -03 -std=c++17 -pthread example.cpp -o example -fopenmp
```

In that way, the pragma extension will send directives to the compiler, and the for will be executed in parallel. Without the flag, the compiler will compile the file without errors, but the pragma will be ignored and the for executed sequentially.

Also, GrPPI has its MAP implementation. As for the pipeline, we initially have to indicate the backend. Then, we have to designate the first and last element of the collection from which starts and finishes the mapping, the first element of the collection storing the mapping, and eventually, the function to be applied. The equivalent of the previous example using GrPPI is:

```
map(parallel_execution_native(nw),
    vect.begin(), vect.end(), vect.begin(),
    [](int x){ return (5 + x);});
```

Note although that using the pragma extension permits us to do more things besides a simple MAP. For instance:

```
#pragma omp parallel for num_threads(nw)
for (int i=1; i<n; i++){
    vect[i] = vect[i] * vect[i-1]
}
```

5.4 Histogram Example

In this example, we will try to build a histogram in parallel. A histogram is a collection of bins used to represent a data set. Each element of the data set is assigned to a bin that stores the number of data elements assigned to it. Suppose, for instance, that x_i can be between 0 and 3, a histogram upon a simple array as [1 0 2 1 3 3 2 3] would be [1 2 2 3].

We will try to build a histogram of 256 bins that represents a random black and white image. The ith bin will contain the number of pixels with value i.

So, we first generate the image by writing:

```
const int bn = 256;

vector<int> image(n);
srand(seed);
generate(image.begin(), image.end(),
        []() { return(rand()%bn); });
```

The following code build the histogram sequentially:

```
vector<int> histo(bn,0);
for_each(image.begin(), image.end(),
         [&] (unsigned char i) { histo[i]++; });
```

The time spent to sequentially build an histogram of 1000000 pixels is approximately 3000 μs. Our goal will be to reduce this time parallelizing the computation with 4 threads.

Let us write the function that each thread will execute:

```
auto thread_body = [&](int threadno) {
    // compute range
    auto delta = n / nw;
    auto start = (threadno * delta);
    auto stop = (threadno == (nw - 1) ? n : (threadno + 1) * delta);
    // compute histogram
    for (int i = start; i < stop; i++)
    {
        histo_thr[image[i]]++;
    }
    return;
};
```

Every thread initially determines the set of elements it must deal with and then increases the histogram bins according to which pixels it scans. Note that the histogram is a global data structure that is shared among all the threads. The time needed by this solution is around 10000 μs which is much more than the sequential version. In addition to this problem, since we do not control the accesses to the histogram, the result will be very often wrong. Indeed what will happen is that several threads will try to update the same bin at the same time, determining the loss of at least one update. So, the bins in the histogram obtained from this first solution will contain smaller or equal numbers than the bins contained in the sequentially made histogram.

The simplest solution is to define a mutex (`mutex globallock;`) and then use it before and after accessing the histogram:

```
for (int i = start; i < stop; i++)
{
    globallock.lock();
    histo_thr[image[i]]++;
    globallock.unlock();
}
```

Unfortunately, this solution slows down the program dramatically to hundred of μs. That is because only one thread can update the histogram at a time, even though it should be possible to simultaneously update two different bins. Indeed, the succeeding solution creates a vector of mutexes, one for each bin, so that if one thread is working on a bin, another thread can work on another bin:

```
vector<mutex> globallock(bn);

auto thread_body = [&](int threadno) {
    // compute range
   ...
    // compute histogram
    for (int i = start; i < stop; i++)
    {
        auto v = image[i];
        globallock[v].lock();
        histo_thr[v]++;
        globallock[v].unlock();
    }
    return;
};
```

This solution goes slightly better, generating the histogram in around 30000 μs. However, we are far above the sequential program. The next improvement is to define a histogram as a vector of atomic instead use mutexes. Indeed, the operation we apply to the histogram's bins is ++ which atomic objects support.

```
vector<atomic<int>> histo_thr(bn);
for (int i = 0; i < bn; i++)
    histo_thr[i] = 0;

auto thread_body = [&](int threadno) {
    // compute range
    auto delta = n / nw;
    auto start = (threadno * delta);
    auto stop = (threadno == (nw - 1) ? n : (threadno + 1) * delta);
    // compute histogram
    for (int i = start; i < stop; i++)
    {
        histo_thr[image[i]]++;
    }
    return;
};
```

Unfortunately, even this solution does not help us to do better than the sequential program since it still needs 15000 μs to terminate.

The key idea to go below the sequential program is to **reduce the concurrent accesses and increase the local operations**. In order to do that, we create a histogram for each thread so that they all work locally with their data structure. Then, we merge the partial results in the final histogram. We can implement this concluding step:

- Sequentially at the end, once every thread has terminated.
- Letting the threads themselves update the histogram. In this case, we hope that they terminate at slightly different times and do not have to compete to update the histogram.

```
vector<atomic<int>> histo_thr(bn);
for (int i = 0; i < bn; i++)
    histo_thr[i] = 0;

vector<vector<int>> locals(nw);

auto thread_body = [&](int threadno) {
    // compute range
    auto delta = n / nw;
    auto start = (threadno * delta);
    auto stop = (threadno == (nw - 1) ? n : (threadno + 1) * delta);
    // alloc locals
    locals[threadno].resize(bn, 0);
    // compute histogram
    for (int i = start; i < stop; i++)
    {
        auto v = image[i];
        locals[threadno][v]++;
    }
    // now update the global histogram
    // using atomics
    for (int i = 0; i < bn; i++)
        histo_thr[i] += locals[threadno][i];
    return;
};
```

We finally reach a solution that requires less time than the sequential program terminating in more or less 1 μs.

So, the take-away messages of the example are:

- **We have to reduce the concurrent accesses as much as we can**.
- **We have to think a clever strategy to parallelize**.

Indeed we start from a simple idea, and then step by step we build the final solution that gives us a speedup of 2/3.

Note also that in this example, we have used both MAP and REDUCTION patterns. Indeed, we can compute H[i] by first scanning the image vectors, substituting the elements that are equal to i with 1 and the others with 0 (MAP). And then by summing up the elements of the vector (REDUCTION).

$$\text{histo[i]} = \text{reduce(+,map(f))} \tag{5.5}$$

As 5.5 is an operation we perform on a single item, we can map all these operations over all the items of the histogram:

$$\text{map(reduce(+,map(f)))}$$

Chapter 6

Stencil Pattern

The stencil pattern is very similar to the map pattern, but instead of having a function that takes in input one item, we have a function that takes in input a central item and its neighborhood defined by a neighborhood function. Many

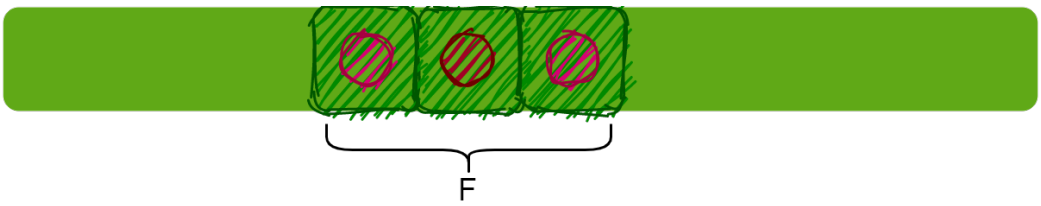


Figure 6.1: The function takes in input the central red point and the neighborhood countersigned by pink points

math problems use this pattern. For instance, suppose we want to compute the average temperature of the parts of a metal vector. We know that the left-most side is 0° C, the right-most side is 100° C and every vector element is 20° C. As shown in figure 6.2, we compute the average temperature of the first element by taking the average between its value and the values of its neighborhood.

We can extend the pattern to more complex data structures as a matrix. For example, we can see the life game invented by John Horton Conway as a matrix. We initially sign each cell as alive being or not by using a random probability distribution. Then, at each step of the simulation, each living being: 1) starves if too many living beings surround it 2) dies of loneliness if

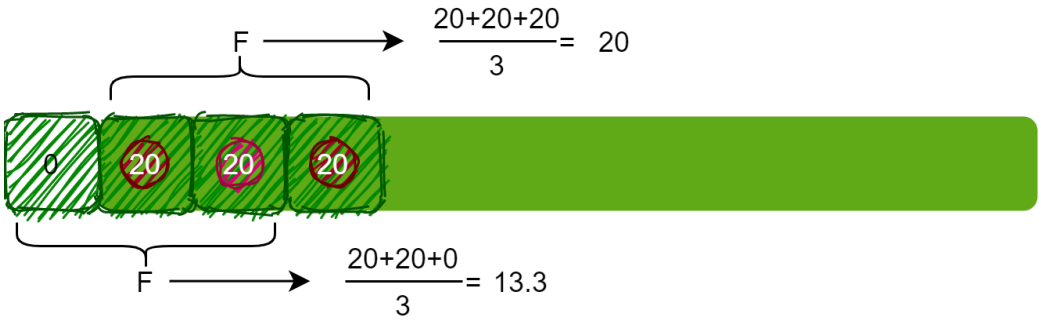


Figure 6.2: Calculating the average temperature of the first element of the vector

it is excessively lonely, 3) reproduces itself otherwise.

Even if the STENCIL pattern seems to be close to MAP or REDUCE patterns, there are additional problems. Consider a for where we compute the average of the values of three adjacent elements in an array:

```
for (i=0; i<N; i++)
    x[i] = average(x[i-1], x[i], x[i+1])
```

That is wrong. Indeed, considering the example in figure 6.2 and using this code, the result for the second example would be 17.7 while the correct answer should be 20. That is because the code updates x[i], and then when it computes x[i+1], instead of taking the original x[i], it takes the updated one. So, we need to take into account to keep consistency across iterations keeping old values at least until they are still needed to compute new values. We can do this in two ways:

Alternative Buffers: we keep two data structures, x, and y, with the same shape. At iteration k, we read x, and we put the new values in y. At iteration k+1, we read y, and we put the new values in x. The implementation of this strategy is very straightforward. For example, in C or C++, we can write a code that always refers to reading x and writing y, but at the end of each iteration, we swap the pointers of x and y:

```
vector<float>* x = new vector(n);
vector<float>* y = new vector(n);
```

```
//exec stencil

//swap
auto temp = x;
x = y;
y = temp;
```

Temporal Buffer: we store the newly computed values into a buffer until the old values they would substitute are needed for the next computations. This solution depends on the neighborhood shape since that will be approximately the same shape as the buffer. Taking the metal vector example, we can use a buffer B of size 1 to store the new value. So, at step i, the buffer will contain the new x[i-1]=f(x[i-1-1],x[i-1],x[i]) while the position i-1 of the vector will still carry the old x[i-1] which is needed for the computation of f(x[i-1],x[i],x[i+1]). Once, we compute the new x[i] = f(x[i-1],x[i],x[i+1]), we copy the content of the buffer in i-1 and then the new x[i] is copied in the buffer.

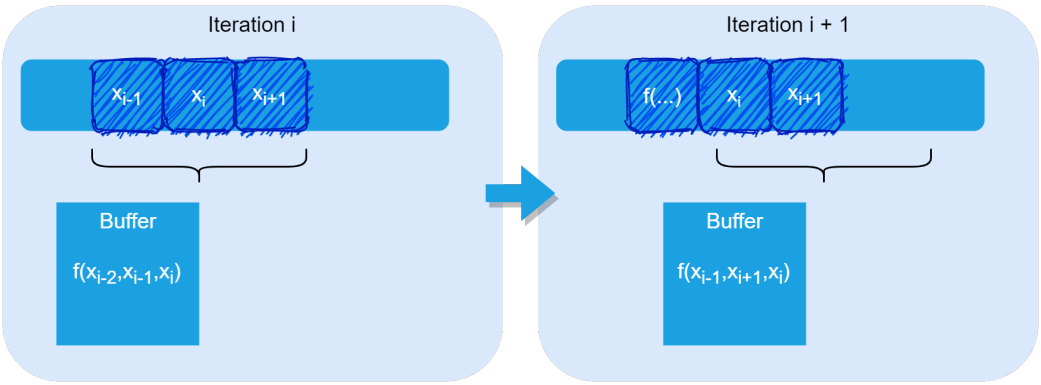


Figure 6.3: Temporal buffer example

The first solution needs two times the size of the whole data structure but has a very simple control flow. On the other hand, the second solution requires one time the size of the data structure plus one time the neighborhood size. But it has a slightly more complicated control flow.

A problem we still have not considered is the borders. What happens when we have to compute x_i with $i = 0$ or $i = N - 1$? Several cases depend

on the problem we are trying to model with computations. Here we present three methods:

- We consider some extra dummy values not existing in the original vector we can use as x_0 and x_N.
- We take $x_{0-1} = x_N$ when i = 0, and $x_{N-1+1} = x_0$ when $i = N - 1$. In other words, we consider our vector as a toroid (see figure 6.4).
- We consider the last and the first values fixed, and therefore we compute new values for i that goes from 1 to N-2.

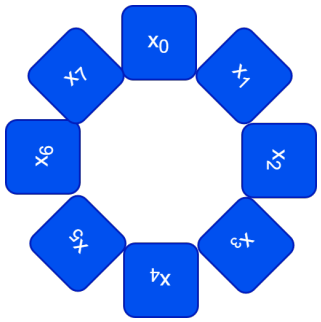


Figure 6.4: Toroid shape

6.1 Parallel Stencil

As for the other patterns, we split the data structure into as many pieces as the number of threads. Considering to work with a vector, suppose to have a thread dealing with the positions from 0 to k, what happens when it has to compute the value in position 0 or k? If we are using a toroid vector, the thread will have to read positions k+1 and 0 which are in charge of another thread. Therefore, we need some kind of synchronization. For instance, at the end of each iteration, all the threads could send to the threads that handle nearby zones the new border values as figure 6.5 displays. In the case of processes that run on different machines (clusters), there is also a communication cost because there is not shared memory space.

So, in general, threads work on their partition until, at a given point in time, they collectively stop within a "barrier" to exchange new values with

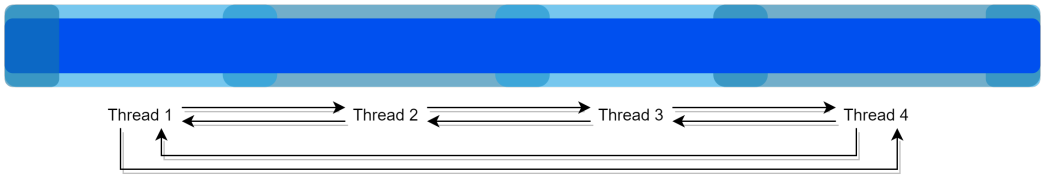


Figure 6.5: The thread must synchronize at the end of each iteration. In this case, the vector is a toroid, so also thread 1 and thread 4 must exchange values.

the other threads (collective synchronization). Barriers require quite an effort even if we implement them using a simple atomic variable:

1. We initialize the atomic variable to the number of threads.
2. Each thread decreases the variable once it has terminated the synchronization phase.
3. Each thread awaits the variable to reach 0.

In architectures explicitly designed for executing parallel computations, there is a specialized intercommunication network that, through broadcast messages, implement barriers across a large number of threads/cores.

A simpler solution that avoids barrier usage is merely forking k threads at the beginning of each iteration, joining them at the end, and finally update the boundaries:

```
for (i = 0; i < num_iterations; i++){
    fork threads
    join threads
    update boundaries
}
```

However, we will pay fork and join at each iteration, so we should prefer a model where we create the threads at the beginning and synchronize them in the barrier.

Chapter 7

Overhead

7.1 Architectural overhead

Suppose we have a pipeline with three stages with latency 3, 10, and 5 seconds, each implemented using a farm. The first stage uses a farm with 3 workers, the second a farm with 10 workers, and the last one employees 5 workers. Hence, the total number of workers employed by the pipeline is 18. We need thereby 18 cores of process resources. But if we run the pipeline in a PC with an Intel i7, we will have only 4 cores/8 threads. Thus, we have to find a number of workers such that the time service of the stages are balanced, so they are almost equal, and the number of workers is less or equal to 8.
For instance, a solution could be the first stage with 1 worker, the second with 4 workers, and the last with 2 workers. In this way, the total number of workers is 7, and the service time of each stage is almost equal:

$$3 \approx 2.5 \approx 2.5$$

Therefore, the service time is 3.

However, having more threads has some drawbacks. For instance, if the computation does not last so much, the time needed to fork and join the threads could be considerable. In a i7 processor, the average time to fork a thread is 10 μs. Thus, we need $10\mu \times 100 = 1\ ms$ to generate 100 threads. Note that although this time also depends on the machine on which the program is running. For instance, in a remote computer with a Xeon Phi, fork a thread requires 100 μs. So, generate 100 threads requires $100\mu \times 100 =$

$10ms$ that could be the time to run the entire task.
Of course, this overhead is irrelevant if the number of tasks m is significant. Indeed the computation would be much larger than the overhead to set up the threads.

Another aspect we should consider is that the operating system moves our threads from one position to another. We can see it writing a program in C++ that uses `sched_getcpu()` to keep track of where threads are running. As you can notice looking at one possible output of the program, the OS transfers thread from one position to another:

```
Worker 0 executing on cpu 3 (was -1)
Worker 1 executing on cpu 0 (was -1)
Worker 2 executing on cpu 6 (was -1)
Worker 3 executing on cpu 1 (was -1)
Worker 2 executing on cpu 7 (was 6)
Worker 3 executing on cpu 2 (was 1)
Worker 3 executing on cpu 1 (was 2)
```

For instance, the worker executing on 1 is moved to 2 and then is moved again in 1. The fact of moving a thread to another core (in picture 7.1, 0 and 4 are in the same core, 1 and 2 are in different cores) is a matter of moving data among the caches. Indeed, the thread working on new position 2 will behave worse than the one that was working on position 1 because it does not have data in the cache.

We can use some specific calls to "**PIN**" threads to the core. In this way, the operating system will not move threads. Of course, in this case, we have to pay attention when positioning the threads inside the CPU. For example, putting all threads of a pipeline in different cores is not a wise decision because every communication would need the main memory. A better solution would be putting the threads within the same core in order to make them communicate only by the cache.

The **take-away** message is that there are many things we have to take into account, such as architectural details, that are entirely out of the business logic of our program but can cause significant overhead in our computation. So, we have to pay attention to all the hidden overheads. In particular to those related to setting up the parallel activities and managing the parallel activities. Those overheads depend on the feature of the machine on which the program is running. And thereby, we need to build an implementation that exploits the features of that machine.

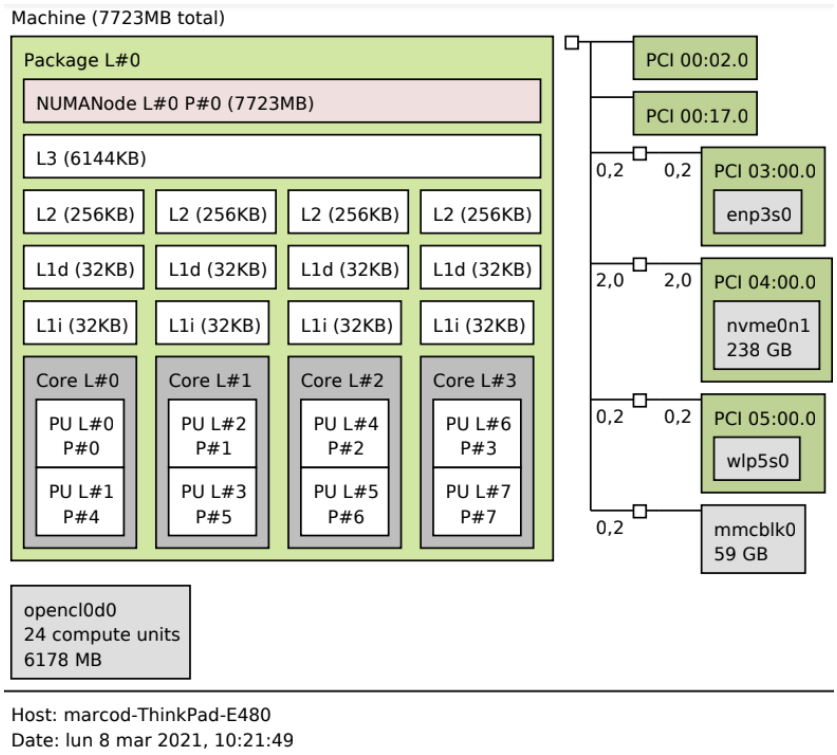


Figure 7.1: Intel i7 processor architecture.

Our idea will be to take a general concept as a farm and design implementation for our target architecture. On top of this implementation, we will provide a library that we will use inside our application. So, we move from an abstraction level about threads, sharing memory, mutexes, and so on to a level of patterns. Therefore, we can implement our application using the patterns rather than going down to the lower level.

7.2 False sharing

Recall that every modern PC is based on an architecture whit a processor, which can have several cores, accessing a memory subsystem composed possibly by several cache levels. Those caches host the memory space that the thread executing in the core needs to work. The cache is organized into lines. Every line includes a tag containing some control information and a

part where actual data are stored (usually 8-16 words).
In a multi-core environment, each core has its first-level cache, and the caches are kept coherent to enforce the model to have a single general memory accessed by all threads. The protocol that maintains the coherency is called cache cohering protocol and is implemented at the hardware level. Indeed it is one of the most expensive parts.
The cache coherency is ensured per line. Therefore if a thread modifies x, the whole line containing x is replaced in the other caches hosting x. The lines are more than a single word to enforce the principle of spatial locality.

The cache coherency could be a severe problem in many cases adding up to 10/20% of overhead. For instance, let us suppose we are performing the stencil pattern over a vector using two threads as in figure 7.2. When thread 1 writes x_i, cache coherency updates the same cache line hosting x_i for thread 0, even if it is useless. So, updates go on at each iteration, moving data to and from the caches causing some useless extra overhead.

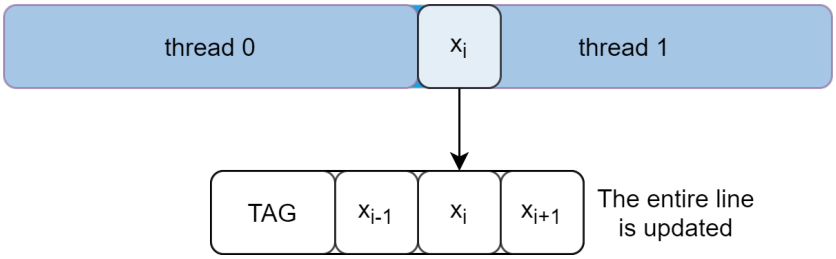


Figure 7.2: When thread 1 updates x_i, all the caches hosting that line are updated due to the cache coherency protocol. So, even the cache for thread 0.

Let us consider a worse case taking the histogram example in section 5.4. A possible solution to calculate the histogram in parallel would have been to have as many threads as bins. Each thread scans the whole vector and updates just one bin. So, all the operations do not need any synchronization since all the threads work only on their bins. Here comes the problem: the histogram is an array, and therefore several adjacent elements will be in the same line. Therefore, guessing that h[0], h[1], h[2], and h[3] are in the same line, when thread 0 updates h[0], the cache coherency protocol will updates also the lines of threads 1, 2 and 3 even if thread 1, 2 and 3 will never access h[0]. The protocol will keep updating the caches for something completely useless.

The protocol where we have one thread writing one value in a data structure is called **Single Owner Computed Rule**. As you can notice, false sharing is particularly nasty with this protocol.

7.2.1 Solutions

To avoid that the cache coherence protocol will affect our vector (or data structure), we can enlarge the vector elements by adding some padding values so that each line can only contain one element. Hence, recalling the previous example, the cache line will hold h[0] and then only some padding values. So, when thread 0 updates h[0], there will not be other caches hosting the same cache line, so the coherence protocol will not work.

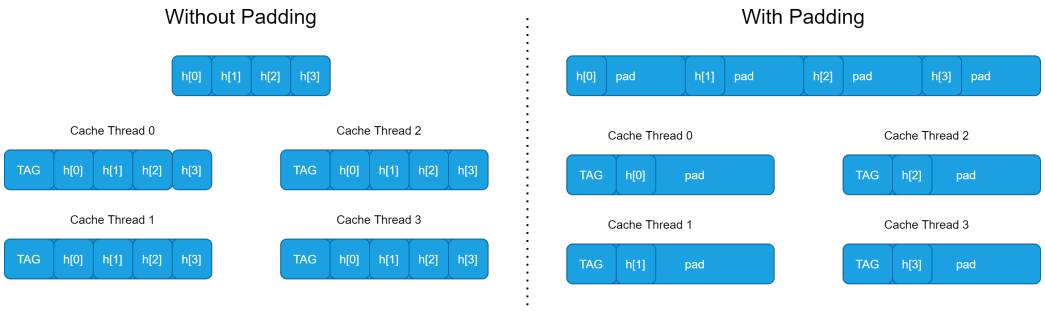


Figure 7.3: On the left, we have the vector without padding. On the right, we add some padding so that each cache line contains one vector element.

7.3 Memory allocation

When we work with programming languages, it often happens we have to allocate memory on the heap. For instance, when we create an object. In multithread applications, the access to the heap is synchronized because it must be kept consistent. Thus, the threads will compete to access the heap, and that will bring some overhead that could become very high if the threads need to access or write the heap in their main loop.

7.3.1 Solutions

We will rely on different tools. But in general, knowing we have k threads going allocate memory from the heap, what we can do is:

1. Allocate k "big chunks" from the general heap to threads.
2. The new operations from thread i are solved using "local" heap chunk.
3. When the "local" chunk is finished, we compete to allocate another chunk from heap.

Of course, we will not directly implement it, but we will rely on some convenient libraries that solve these problems as horde or gperf. The usage of these libraries can bring important benefits to programs that have large amounts of calls to the heap from different threads.

7.3.2 Jemalloc

Jemmalloc is a general-purpose malloc implementation that emphasizes fragmentation avoidance and **scalable concurrency support**. You can download it and then follow the instruction in INSTALL.md to configure, build and install jemalloc. Then we can set LD_PRELOAD to the path that leads to the file `./jemalloc/lib/libjemalloc.so.2`. That file will be loaded **before** any other library (including libc.so that contains the malloc code):

```
LD_PRELOAD=./jemalloc/lib/libjemalloc.so.2 ./yourprogram youprogparam ...
```

Taking a program that extensively allocates memory in heap, the improvement in performance using jemalloc is massive:

```
#without jemalloc
./prog 100000 97 123
runtime computed in 1608077 usec

#with jemalloc
LD_PRELOAD=./jemalloc/lib/libjemalloc.so.2 ./prog 100000 97 123
runtime computed in 617416 usec
```

Note that we did not change any line of code. We have just changed the malloc implementation used by the program.

We can make the LD_PRELOAD permanent by typing:

```
export LD_PRELOAD=./jemalloc/lib/libjemalloc.so.2
```

And undo the LD_PRELOAD by writing:

```
export LD_PRELOAD=
```

When you have to exec the make to install jemalloc, we suggest using more than one thread to terminate earlier. To do that with 4 threads, you just need to write:

```
make -j 4
```

Chapter 8

Load Balancing

We would like to give each thread the same amount of work. Unfortunately, do this could be very nasty because the amount of work depends on the data we supply to the threads. Even in an example simple as the translation book, there could be pages whose translation requires more time. The load balancing, which is the process of distributing a set of tasks over a set of resources, is crucial if we want that there will not be threads awaiting the slower ones without doing anything. Two main approaches exists:

Static load balancing policies device a scheduling policy of concurrent activities to processing elements **before** the computation actually start.

Dynamic load balancing policies devise the scheduling policy while computation is running.

8.1 Static

The classical static scheduling policies are:

Block distribution The m concurrent activities are partitioned into n chunks of size $\Delta = \frac{m}{n}$. That is we assign activity a_i to thread P_j such that $j = \text{floor}(n/m)$.

Cyclic distribution The m concurrent activities are partitioned into n chunks of size 1 and distributed to the threads circularly. That is we assign activity a_i to thread P_j such that $j = i \bmod n$. In general, we could also partition the activities into chunks of size `chunk_size` > 1.

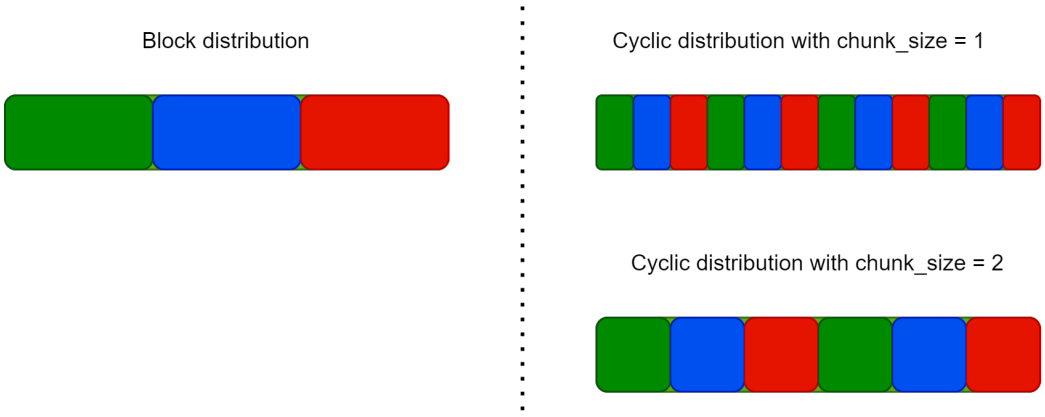


Figure 8.1: On the left the block distribution. On the right the classic cyclic distribution and below a cyclic distribution with chunk size higher than one. Each color represent a thread

Let us analyze the pros and cons of the two methods. Consider a situation where the first tasks require less time than the later ones. Of course, the block schedule will fall into trouble since the red thread will pick all the long tasks. On the other hand, the cyclic scheduling handles the situation distributing the long tasks to all the threads. So, in general, the cyclic distribution performs better when "hot spots" are present (that is, a portion of concurrent activities with consecutive indexes that all require execution time higher (or lower) with respect to the average execution time).

The drawback of the cyclic distribution is that threads must read non-consecutive elements, which means that we cannot use the spatial locality, and there will be many cashes misses. So, it is much less efficient in memory.

8.2 Dynamic

8.2.1 Variable Chunks

To try to put a patch on the problems of the previous policies, we could do something as follow:

1. Given an initial set of concurrent activities to be computed, an initial, large portion of the activities is assigned to processing elements statically with either block or cyclic policies.

2. The rest of the concurrent activities to be computed is divided in smaller and smaller chunks that are assigned to those processing elements that finish their previous assignments.

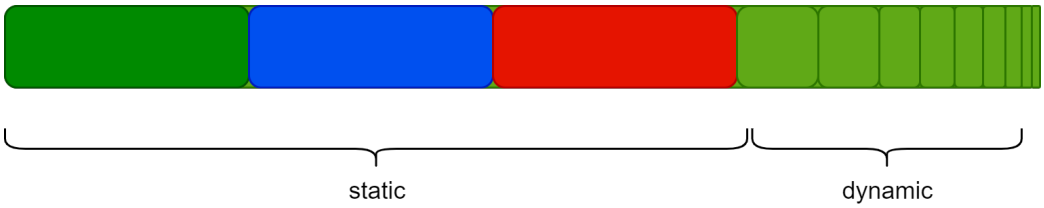


Figure 8.2: The first three portions are assigned statically. The rest of the chunks, whose size becomes smaller and smaller, are assigned dynamically

8.2.2 Job Stealing

Another quite popular approach consists of initially assign all the concurrent tasks. In general, we use static policies. Then let threads that finish earlier "stealing" activities to those threads that still have activities to compute. For instance, consider two thread computing 200 concurrent tasks each. At the time t, the first thread terminates while the second still has to calculate 100 tasks. Thus the first thread could pick 50 of the remaining tasks so that both threads will finish earlier and avoid idle time. In order to set up job stealing, we need to define:

- A way to figure out which are the processing elements still having something to compute. (who should I steal?)
- How and how much steal.

We can answer the first point in two alternatives:

- Use a centralized entity to which the quick thread could ask what are the thread still running. It is pretty simple to implement. However, it introduces a centralization point, which is a single point of failure and a bottleneck. Moreover, we should implement a policy that assures that all the threads do not steal from the same thread.
- Steal from a thread randomly and locally selected. The con is, of course, that the locally devised candidate could have nothing to steal.

The second point is about how the thread should steal tasks:

How much The closer we are to the end of the computation, the smaller amount the thread should try to steal.

How A thread will not be aware that someone is stealing its tasks. Therefore, the runtime of the threads should be properly programmed to accept external "steal" requests. For instance, we could assign to each thread a **double end synchronized queue**, so that the main thread pops the element from the head while the thief threads pop the element from the tail. In this way, the main thread will always find the lock for the head open unless the queue has one item. On the other hand, the thief threads will compete to get the lock for the tail. So there will never happen that everybody tries to steal everything making the queue empty in a low time.

8.3 OpenMP

We have seen that using OpenMP, we can parallelize a for just writing:

```
#pragma omp parallel for
for (...)
{ ... }
```

OpenMP lets us also the possibility to set the scheduling policy to be used. For instance, the following code will say to OpenMP to schedule the threads using a static scheduling type:

```
#pragma omp parallel for schedule(static)
for (...)
{ ... }
```

For further information, we suggest reading the documentation*.

Chapter 9

Laboratory 2

9.1 Async

Async provides a way to start a concurrent activity that will be performed by another thread completely asynchronously to the computation of the thread that uses an async call. The async function returns a future object that will eventually hold the result of that function call. To retrieve the result, we have to use the method get of the future object. The get will block the calling thread until the result of the asynchronous execution is available. The async function takes three arguments:

- The launch policy that can be `async` or `deferred`. `async` says that the asynchronous thread should execute immediately. `deferred` says that the call to the function is deferred until the calling thread does not access the shared state of the returned future (with wait or get).
- The function to be executed asynchronously.
- The arguments of the calling function.

The following code calculates the ith element of the Fibonacci sequence. In particular, we commit half of the computations to another thread using the async mechanism:

```
int fib_async(int n){
    if (n < 2)
        return n;
```

```
    auto x = async(launch::async,
                   fib_async,
                   n-1);
    int y = fib_async(n-2);
    return x.get() + y;
}
```

Notice that if we compare the time needed by this implementation against a plain sequential implementation, the difference is enormous, and the winner is unexpected. The async implementation requires 1722810 μs while the sequential one terminates in only 105 μs.

9.2 Packaged Task

A package task is similar to the async function, but instead of returning a future object, it wraps a Callable target (function, lambda expression, bind expression, or another function object) so that it can be invoked asynchronously later. To get the future object, we have to use the `get_future` method. Then the story is the same as the async function.

As for the async function, we again propose the Fibonacci example:

```
int fib_pkg(int n){
    if (n < 2)
        return n;
    packaged_task<int(int)> task1(fib_pkg);
    future<int> x = task1.get_future();
    task1(n-1);

    int y = fib_async(n-2)
    return x.get() + y;
}
```

The time needed by this code is 1550780 μs, still higher than the sequential time but slightly better than the async code.

9.3 OpenMP

In this section, we will consider other magic things we can do using OpenMP. To recall, we can exec a for in parallel with a given number of threads `nw` by

merely writing:

```
#pragma omp parallel for num_threads(nw)
for (...)
{ ... }
```

If we wrote only `#pragma omp parallel for`, OpenMP would have used the maximum number of thread for your computer.

Note that we can also parallelize blocks different from the for:

```
#pragma omp parallel num_threads(nw)
{ ... }
```

9.3.1 Sections

Sections allows to define a region of structured blocks that will be distributed among the threads. The syntax of this construct is:

```
int main()
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections
        {
            #pragma omp section
            function_1();

            #pragma omp section
            function_2();
        }
    }

    return 0;
}
```

In the example, OpenMP will fork two threads, and then a thread will exec function1 while the other thread will exec function2.

9.3.2 Locks

OMP also offers mutexes object to deal with critical sections. The code below shows the syntax to initialize and use them:

```
omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
   omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

Besides, we can also use **omp critical** to identify section of codes that must be executed by a single thread at a time:

```
#pragma omp critical
{
    // one thread at a time stuff
}
```

9.3.3 Barriers

The omp barrier directive identifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point. Statement execution past the omp barrier point then continues in parallel[1].

```
  #pragma omp parallel private(i) shared(n) {
    #pragma omp for
```

[1] IBM source

```
    for (i=0; i<n; i++)
        //do stuff 1

    //wait all threads terminate
    #pragma omp barrier

    #pragma omp for
    for (i=0; i<n; i++)
        //do stuff 2
}
```

9.3.4 Utilities [TO DO]

9.3.5 Variables [TO DO]

9.3.6 Tasks [TO DO]

Chapter 10

Hiding communication cost

Suppose we have a pipeline stage f sequentially implemented. Thus, its pseudocode will be as follow:

```
while("not eos") {
    receive xi
    compute f(xi)
    send f(xi)
}
```

So, from a time perspective, f will behave as follow:

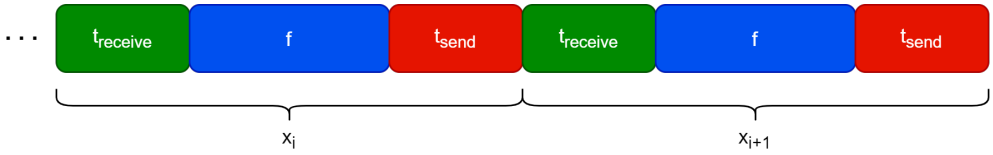


Figure 10.1: Sequential implemented stage. The program must receive, compute and send sequentially.

In this chapter we will discuss of how hide $t_{receive}$ and t_{send}. The general idea will be to pipeline the stage by creating a stage for receiving, a stage for computing, and a stage for sending. In this way, the computation and communications cost will overlap, and so at the pipeline steady-state, the service time will be equal to $max\{t_{send}, t_{receive}, t_f\}$. However, there are cases whose cost of $t_{receive}$ and t_{send} is so low that pipeline the stage is useless.

Others, due to a remarkably high communication time, need to use this approach.

10.1 Shared Memory Multicore

In this architecture, receiving and sending operations are mere reads and writes operations to shared memory locations. In particular, we can implement those operations using pointers: the stage before f, instead of sending x_i, sends the memory address that points to x_i. The same does f when it sends the result to the next stage. Writing/reading pointers require nanoseconds. Besides, the stages must use a synchronized data structure to communicate. Therefore we have to add to the communication cost approximately 10 msec. To summarize, the total communication cost will be:

$$10ms + ns$$

Since communication usually endures for much more than 10 ms, the communication cost will be negligible. Therefore, it is not worth having three threads.

10.2 NoW/CoW (Network/Clusters of workstations)

In this case, we will have several computers, each with a processor and a memory, communicating by an interconnection network running, for instance, the TCP/IP protocol. So, if stage g has to communicate x_i to stage f, it must pass x_i through the network, which will require a large amount of time.

In detail, the communication, in most of the modern architectures, would follow these steps:

1. Prepare the send buffer filling it up with x_i.
2. Call a library to send x_i. The library transfers the control to the OS and tells him which is the buffer address to be transmitted.
3. The OS uses DMA to access x_i and put it into the network interface. The network interface controller (NIC) will then handle the communication autonomously without using the CPU cores. Thus, we have to perceive the NIC as another general-purpose core.

It is remarkably worth taking this approach, but it will require more memory. Indeed, suppose to have two threads as in figure 10.2, where Thread1 reads x_i and writes it into Buffer1. Then Thread2 reads Buffer1, computes $f(x_i)$, and writes it into Buffer1. Thread1 can not use Buffer1 until Thread2 use its content to calculate $f(x_i)$, therefore to read another value in parallel to Thread1, it needs another Buffer into which store x_{i+1}. In general, as we have written, we will deal with three threads, one for reading, one for computing, and one for sending, and therefore we will need at least three buffers.

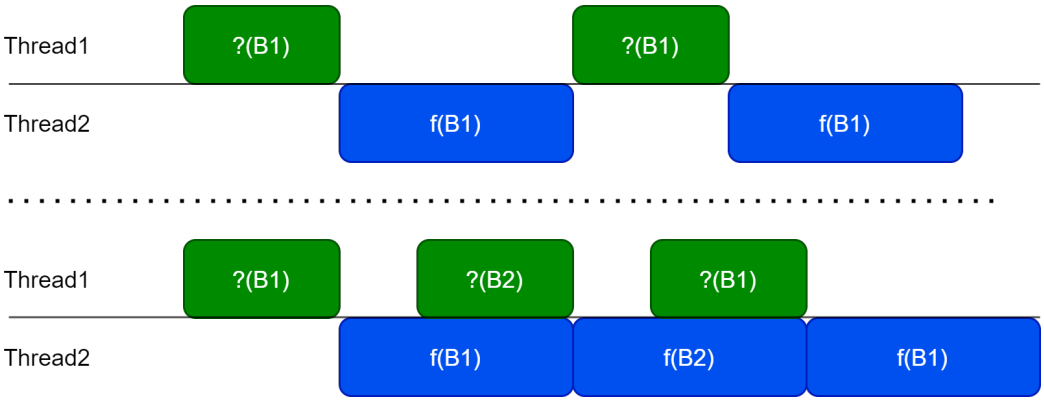


Figure 10.2: The top image shows the case in which we have only one buffer. Thread1 must wait for the termination of Thread2. The bottom image pictures the case with two buffers, where Thread1 can start reading $x_i + 1$ while Thread2 is still computing $f(x_i)$

.

10.2.1 Triple Buffering

We can set up three threads working on three distinct buffers: B_0, B_1, and B_2. Initially, Thread0 will manage to receive data in buffer B_0 while Thread1 and Thread2 will be idle. When Thread0 will have received the whole input task, it can move to receive the next input task into the second buffer B_1, while the Thread1 can start processing task in B_0. After Thread0 will have received the second task in B_1, and Thread1 will have computed the results relative to the first input task in buffer B_0, Thread0 will start receiving the third input task in B_2, Thread1 will start processing the input task already

in B_1 and Thread2 will start sending the processed result in B_0 to the next pipeline stage.

From this moment on the three threads will work at steady state. At each time slot, with duration:

$$\max\{T_{receive}, T_{process}, T_{send}\}$$

each thread processing/sending/receiving on buffer B_i, will move to repeat its operation on buffer $B_{(i+1)\%3}$. In case the times needed to send and receive data are smaller than the time spent to compute/process the incoming task, the communication costs will be completely hidden.

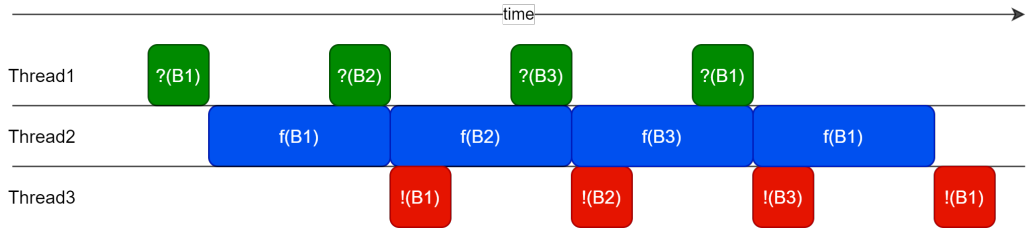


Figure 10.3: Three buffer example

.

All this works under three conditions:

- $T_{send} > T_{receive}, T_{process}$
- The hardware supports parallelism in between communications and computation.
- The hardware has enough memory

10.3 Accelerators

In the first chapter, we discussed that GPUs are accelerators suitable to compute data-parallel operations. GPUs are connected with the CPU via a PCIe bus. When we move computations from the main CPU to the GPU, we need to move data into the GPU memory through the IO bus. Then, the GPU can compute something, then put the result into its memory, and eventually sends the result into the main computer memory.
Again, we have to pay the cost of the communication. However, GPUs

usually have 1/2 "DMA" devices supporting the data transmission between CPU and GPU. "DMA" is in quotes because it is almost DNA; it also controls the bus and operates over it.

If we have two DMAs, we can use the first to receive and the second to send. Otherwise, with a single unit, the DMA will initially send the first task to the GPU. Then the GPU will start computations, and in the meanwhile, the DMA can receive new tasks or send formally computed tasks, but of course, can not send a new task to the GPU.

If the time to send plus receive is lower than the time needed to compute one task, the communication time will be completely hidden (Figure 10.4 case 1). However, if the time to send and receive is larger than the time needed to calculate one task, we will end up with GPU cores in idle, as shown in Figure 10.4 case 2.

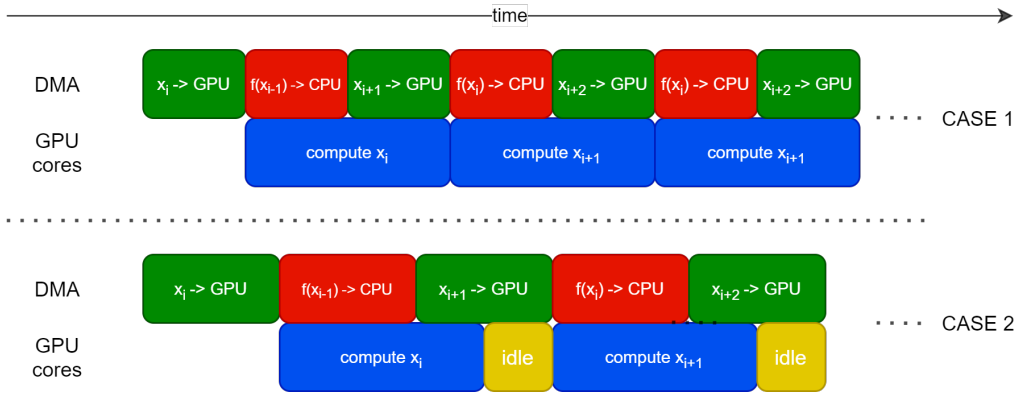


Figure 10.4: Single DMA in a GPU. The first case depicts the ideal case, while the second case shows the situation where we will waste GPU resources

Chapter 11

Vectorization

Code vectorization is a technique aimed at executing in a very fast and efficient way operations on vectors, in all those cases where the operations applied to different vector elements happen to be independent.

The technique is based on the presence of the SIMD extensions in the ASSEMBLER instructions. The ASSEMBLER language is the lowest level we can use to program the CPU. ASSEMBLER languages work assuming a collection of instructions that refer to a set of registers R. For instance, ADD R0 R1 R2 could mean compute R1 + R2 and put the result into R0 (in other languages could be different), where R0, R1, and R2 are registers.

With the SIMD extension, the hardware provides a *vector register* whose components (the vector registers) may be considered as a whole or as a collection of smaller registers. As an example, in a vector register set of 16 registers of 128 bis each, each register may be referred as a 128 bit register, as 2 64-bit registers, as 4 32-bit registers, etc. The SIMD instruction set then applies the same operation simultaneously to two, four, or more pieces of the array. So, vectorization (simplified) is the process of rewriting a loop so that instead of processing a single element of array N times, it processes 4 array elements simultaneously N/4 times. For instance, if we divide our 128-bit register into two 64-bit registers, we can simultaneously exec the same instruction over both the registers.

Since registers can be used at different sizes, the hardware has to provide a number of ALUs (Arithmetic Logic Units) such that they operate computing a function of the correspondent parts of two vector registers and writing the result in the correspondent part of another vector register. As an

example, hardware may provide 4 32 bit ALUs, such that we can compute the sum/subtraction/multiplication/... of the 4 parts of two 128 bit vector registers in a single shot and write the four results in another 128-bit vector register.

It is important to remark that the operations applied to the vector must be independent. Indeed, look at the example in Figure 11.1, where we sum vectors of 8 bit first considering the entire vector and then dividing the vector in 4-bit registers.

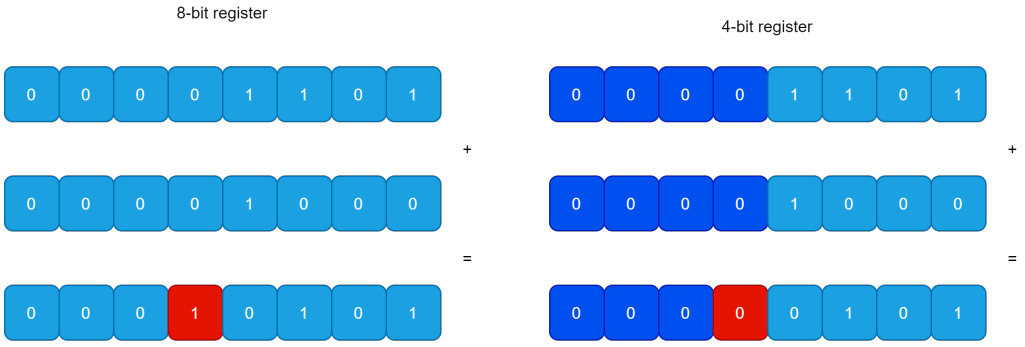


Figure 11.1: Problems with vectorization. The two sums have two different results

11.1 -o3 flag

The g++ compiler permits automatically vectorize the code using the -o3 flag. 3 stands from level 3 of optimization that includes automatic vectorization.

For instance, we will vectorize the following code that sequentially compute the pi greek:

```
1  int main(int argc, char * argv[]) {
2
3      if(argc != 3) {
4          std::cout << "Usage is: " << argv[0] << " num_steps nw " << std::er
5          return(-1);
6      }
7      long num_steps = atoi(argv[1]);
```

```
8
9       double step;
10      int i;
11      double x, pi, sum = 0.0;
12
13      step = 1.0/(double) num_steps;
14      for (i=0;i< num_steps; i++) {
15          x = (i+0.5)*step;
16          sum = sum + 4.0/(1.0+x*x);
17      }
18
19    pi = step * sum;
20
21    std::cout << "Pi = " << pi << " (Computed in "
22              << elapsed << " secs)"  << std::endl;
23    return(0);
24  }
```

The time required by this program without vectorization and with 1.000.000 steps is approximately 0.006 milliseconds. Let us vectorize the code compiling it as below:

```
g++ pi-seq.cpp -o3 -o pi-seq
```

The average time with the same parameter as before is now 0.002 milliseconds. But we can do better by adding the flag `-fopt-info-vec-missed` so that the compiler will advise us if it is not able to vectorize a loop:

```
g++ pi-seq.cpp -o3 -o pi-seq -fopt-info-vec-missed 2> >(grep pi-seq.cpp)

pi-seq.cpp:14:13: missed: couldn't vectorize loop
pi-seq.cpp:1:5: missed: not vectorized: unsupported data-type double
...
```

The compiler informs us that it can not vectorize the loop starting at row 14 because it uses double type. The solution to also vectorize that loop is straightforward since we just have to add the `-funsafe-math-optimization` flag:

```
g++ pi-seq.cpp -o3 -o pi-seq -funsafe-math-optimizations
    -fopt-info-vec-missed 2> >(grep pi-seq.cpp)
```

The compiled code now requires 0.001 milliseconds on average.

In general, the compiler can automatically vectorize loops with these properties:

- Finite loops (for not while). Indeed, the compiler needs to know the number of iterations.
- Number of iterations are know.
- No dependencies between iterations.
- Vector accesses are aligned to optimize access in memory.
- No library calls in the iteration body.

The for in pi-seq respects all the rules. There are cases in which we can overpass these rules as shown at https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

11.2 Vectorization and Parallelization

Suppose to have a sequential code that requires t_{seq} to terminate without optimization. When we insert the sequential code into a #pragma omp for, we will approximately reduce the time to $t_{seq}/n_{threads}$ + overhead.

When we use the -o3 optimization flag, t_{seq} goes down to $t'_{seq} << t_{seq}$. If we put again the pragma into the code, the programm will terminate in $t'_{seq}/n_{threads}$ + overhead where the overhead is the same as before. Therefore, parallelize with vectorization will be less efficient since the percentage of overhead will increase.

In OpenMP, we can use:

```
#pragma omp simd
    for ...
```

That will just look for vectorization instead of working with threads in the thread pool as `#pragma omp parallel for` does.

Notice that putting transformations like pragma over a vectorized loop may introduce a kind of factor that impairs the vectorization of the loop. So, we have always to compare vectorized loops with vectorized loops after parallelization. Or non-vectorized loops with non-vectorized loops after parallelization.

C++ has a pragma which is directed to C++ compiler whose name is `ivdep` that tells the compiler that the loop following has independent iterations

11.3 Loop unrolling

Loop unrolling is a technique that comes from the fact that modern processors are pipelined, which means that the processor executes each ASSEMBLER instruction into several stages. A simple view consists of five stages:

1. Fetches the instruction.
2. Decodes the instruction.
3. Execs the instruction.
4. Accesses data memory. We can skip this step if it is not needed.
5. Writes back results.

The ASSEMBLER instruction `branch *` causes the processor to begin executing the instructions from `*`. Therefore, given the sequence of instructions `i1, i2, branch .L2, i3, i4`, when the processor execs the `branch` in stage 3, it will jump to the instruction labeled by L2, discarding `i3` and `i4` that are in stages 1 and 2. So, the processor will not exec anything for at least two clock cycles.

Loop unrolling is a compiler optimization applied to certain kinds of loops to reduce the frequency of branches and loop maintenance instructions. It is easily applied to sequential array processing loops where the number of iterations is known prior to execution of the loop. The general idea of loop unrolling is to replicate the code inside a loop body a number of times. The number of copies is called the loop unrolling factor. The number of iterations is divided by the loop unrolling factor [1].

As an example, consider the following code:

```
void ffsum(int * x, int * y, int * z, int n) {
    for(int i=0; i<7; i++)
        z[i] = x[i]+y[i];
```

[1] https://www.d.umn.edu/ gshute/arch/loop-unrolling.xhtml

```
    return;
}
```

The corresponding unrolling we get by compiling it with the -03 flag is:

```
ldr ip, [r1]            #load M[r1] in ip           (load x[0])
ldr r3, [r0]            #load M[r0] in r3           (load y[0])
add r3, r3, ip          #r3 = r3 + ip
str r3, [r2]            #store r3 into M[r2]        (store z[0])
ldr ip, [r1, #4]        #load M[r1 + 4] in ip       (load x[1])
ldr r3, [r0, #4]        #load M[r0 + 4] in r3       (load y[1])
add r3, r3, ip          #r3 = r3 + ip
str r3, [r2, #4]        #store r3 into M[r2 + 4]    (store z[1])
ldr ip, [r1, #8]
ldr r3, [r0, #8]
add r3, r3, ip
str r3, [r2, #8]
ldr ip, [r1, #12]
ldr r3, [r0, #12]
add r3, r3, ip
str r3, [r2, #12]
ldr ip, [r1, #16]
ldr r3, [r0, #16]
add r3, r3, ip
str r3, [r2, #16]
ldr ip, [r1, #20]
ldr r3, [r0, #20]
add r3, r3, ip
str r3, [r2, #20]
ldr ip, [r1, #24]
ldr r3, [r0, #24]
add r3, r3, ip
str r3, [r2, #24]
bx lr
```

While the ASSEMBLER code we would get without optimization is:

```
.L2:
    ldr r3, [ip, #4]!
```

```
ldr lr, [r1, #4]!
cmp ip, r0
add r3, r3, lr
str r3, [r2, #4]!
bne .L2                    #branch instruction
ldr pc, [sp], #4
```

Chapter 12

Work span model

In this chapter, we will consider another performance model to expects the speedup our program will have. We have already presented the Amdahl law that relates the maximum speedup to the serial fraction, which is the percentage of non-parallelizable code. The Amdahl law states that:

$$\lim_{n\to\infty} sp(n) = \frac{1}{f}$$

Where $sp(n)$ is the speedup with parallelism degree n, and f is the serial fraction. We also have seen that we can overpass the Amdahl law, for instance, using the Gustafson law. In this chapter, we will see another model called **work span model**.

12.1 The model

The work span model is based on the concept of **dependency**. We say that CodeB depends on CodeA if CodeA produces some data x and CodeB consumes that data. That implies that CodeA must terminate before the start of CodeB. We call this dependency data dependency. Of course, there are other dependencies, such as the control dependencies.

To use the model we have to build a DAG (direct acyclic graph) representing our application such that:

- It has start node without input arcs.

- It has a end node without output arcs.
- The arcs represent dependencies (data dependencies).
- Each node is associated with some measure of time needed to compute the function that takes the input and produce the output.

In the following, we will consider the DAG pictured in Figure 12.1, where:

1. A produces data needed by B and C.
2. A produces data needed by B and C.
3. C produces data needed by D and E.
4. B provides data needed by F.
5. F produces code needed by G.
6. G returns the final output.

The span model uses two measures we can calculate looking at the DAG:

Work: the total "work" necessary to fully compute our program. It is the time needed if the program runs sequentially. In our example, the work is $t_a + t_b + t_c + t_d + t_e + t_g$.

Span: the longest path between the start and the end node. "Longest" means in terms of time spent traversing the path. Supposing that all nodes require 1 second, that is $t_a = t_b = \cdots = t_g = 1\ s$, the longest path in Figure 12.1 is the blue one (p2). We call p2 a critical path. With the span, we represent the set of activities to be executed sequentially. That is, we will never be able to run in parallel A, C, D, F, and G since G depends on the output of F, F on the output of D, and so on.

We denote the work with T_1 and the span with T_∞. The maximum speedup we can reach is:

$$\max speedup = \frac{T_1}{T_\infty} \tag{12.1}$$

In our example, we have that $T_1 = 7$, while $T_\infty = 5$, therefore:

$$\max speedup = \frac{7}{5} = 1.4$$

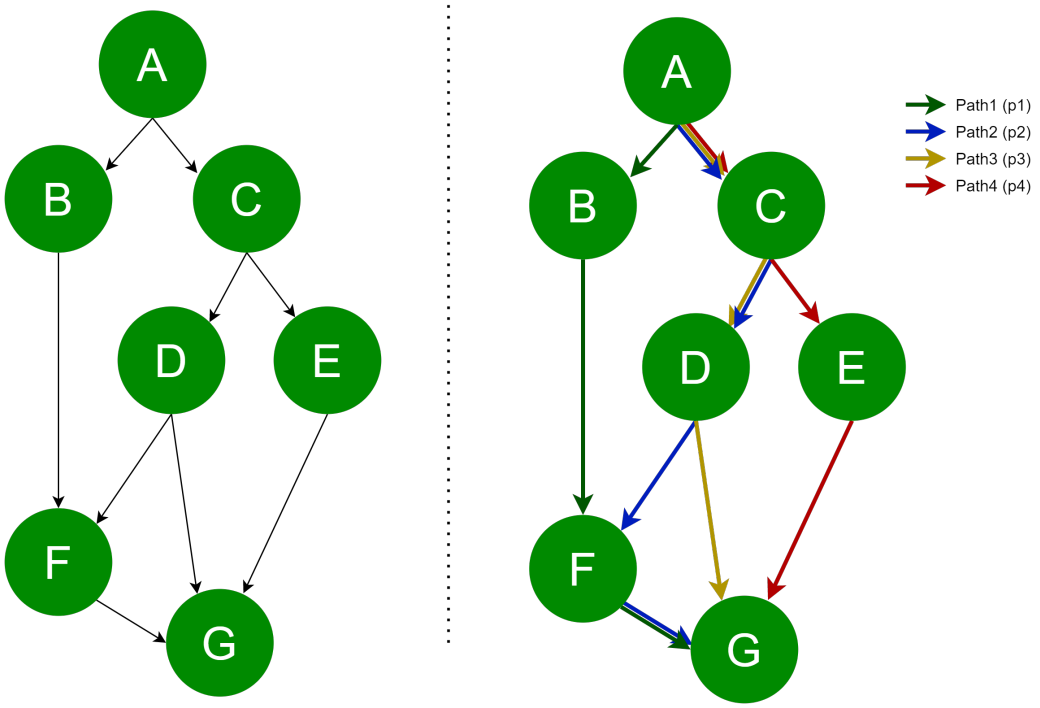


Figure 12.1: On the left, a dependency DAG used for our model span examples. On the right, all the paths from the start node to the end node.

Chapter 13

Algorithmic skeleton vs Parallel design patterns [TO DO]

Chapter 14

Refactoring

In computer programming and software design, code refactoring is the process of restructuring existing computer code - change non-functional properties - without changing its external behavior - do not change functional properties. For those who have never heard what is the difference between functional or non-functional properties:

- **functional** properties refer to "what" the program compute. For instance, the fact that a program returns $f(g(x))$ is a functional property.
- **non-functional** properties refer to "how" the program compute. We can list as non-functional the performances or the kind of implementation.

14.1 Refactoring Rules

The refactoring process usually goes through the usage of so-called refactoring rules. A refactoring rule is an equivalence between two codes with the same functional properties but with different non-functional properties. For instance, the following equation is a refactoring rule:

$$\text{pipe(S1, S2)} = \text{comp(S1, S2)} \tag{14.1}$$

The rule states that executing the skeletons S1 and S2 in a pipeline returns the same result as performing S1 and then S2 sequentially. However, it is clear that the pipeline has a service time equals to $\max\{t_{S1}, t_{S2}\}$ while the

other solution has a service time of $t_{S1} + t_{S2}$. Moreover, the pipeline solution uses two threads, while the second solution works with just one. The thread number and the service time are other examples of non-functional properties. Other rules we will use in the follow are:

$$\text{farm(S1)} = \text{S1} \tag{14.2}$$

$$\text{map(comp(S1, S2))} = \text{comp(map(S1),map(S2))} \tag{14.3}$$

Concerning 14.2, the farm computes in parallel the items it receives from a stream, while the sequential solution computes the items one after the other, but all in all, the functional properties will be the same. However, the ordering of output results - usually a non-functional property - may change. Indeed, $f(x2)$ may require less than $f(x1)$ and, therefore, the output stream will contain $f(x2)$ earlier than $f(x1)$. In the sequential case, $f(x1)$ will always be before $f(x2)$.

The third rule is about data-parallel computations. The result of both calculations is the same because applying S1 and then S2 sequentially, or sequentially computing the map of S1, and then the map S2, is equivalent.

14.1.1 Refactoring Tree

Only use these three rules can generate a lot of configurations producing the same result. Let us consider the stream parallel case where we initially use a simple composition of two functions: comp(S1, S2). Then we can apply different rules over this expression, and in the resulting sub-expressions, we can again employ other rules. Thus, we will generate a refactoring tree where all the nodes are pattern trees computing the same configuration with very different non-functional features. Figure 14.1 shows an example of a refactoring tree of the expression comp(S1, S2).

Once we have built the refactoring tree, we can map a cost function over all the nodes and finally return the node with the least cost.

Note that we have not still considered the parallelism degree. Thus, we can take into account other types of refactoring rules as the following:

$$\text{farm(S1, nw)} = \text{farm(S1, nw + k)} \; \textit{with} \; k \in \mathbb{Z} \tag{14.4}$$

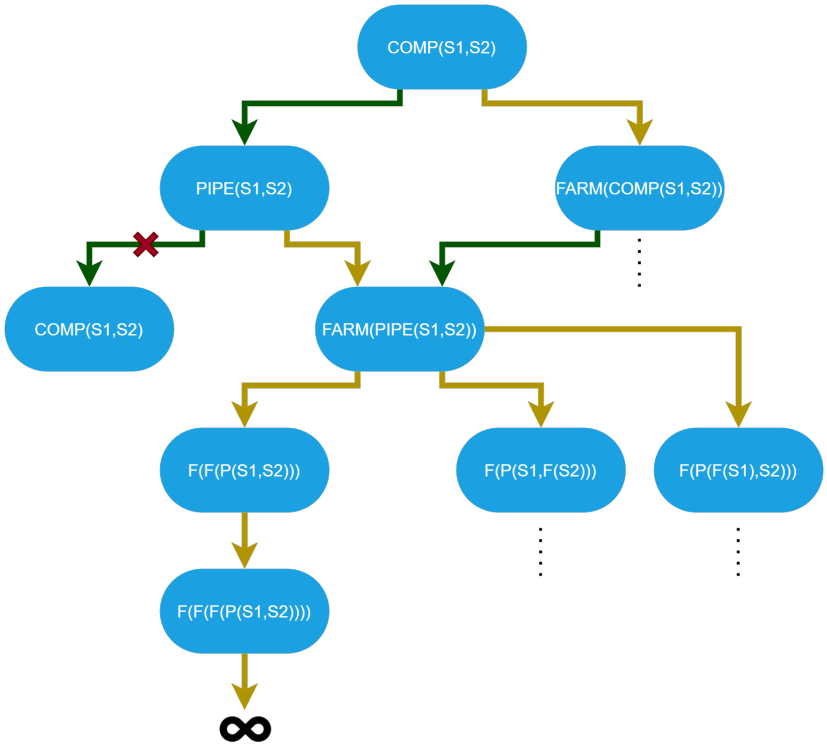


Figure 14.1: Example of a refactoring tree for the expression comp(S1, S2). Note that we do not consider all the possible nodes. We mark the usage of rule 14.1 with the green arrow, while yellow arrows indicate the application of the rule 14.2. Note also that the tree may have infinite height. The red arrow indicates that the configuration the directed line leads to is already in the tree.

14.2 Normal Form of stream parallel pattern trees

In this section, we will take into account only stream parallel pattern trees. We will suppose to have a tree whose nodes are pipes (P), farms (F), sequential code (S), and compositions (C) as the one in figure 14.2. To get the so-called normal form of stream parallel pattern trees, we have to follow these steps:

1. Take "fringe" left to right. The frindge contains the sequential leafs in

the order we get reading them left to right. Thus, the fringe of Figure 14.2 is {S1, S2, S3, S4}.

2. Make a `comp` out of it, which means return comp(S1, S2, S3, S4).

3. Put a farm on top: farm(comp(S1, S2, S3, S4)).

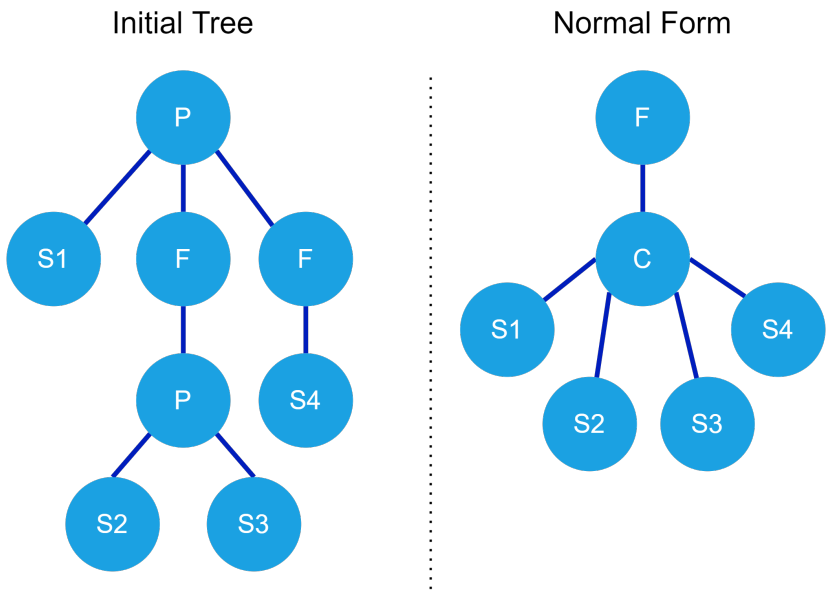


Figure 14.2: On the left, our reference pattern tree for examples. On the right, the corresponding normal form

Theorem normal form

The normal form (NF) is **as efficient as the original one or better.**

We will not give any proof of this theorem. However, we can reason and try to get the reasoning behind it.

Using the tree in Figure 14.2, we can see if the normal form is actually functionally equivalent to the initial tree. To do that, we can start from the initial tree and then try to apply the refactoring rules previously presented. Figure 14.3 shows this process.

Let us now compare the service time of the initial tree compared with the normal form tree. We will assume to know the time t_i needed to execute Si. Thus t_1 is the time to compute S1. We will also assume that we know

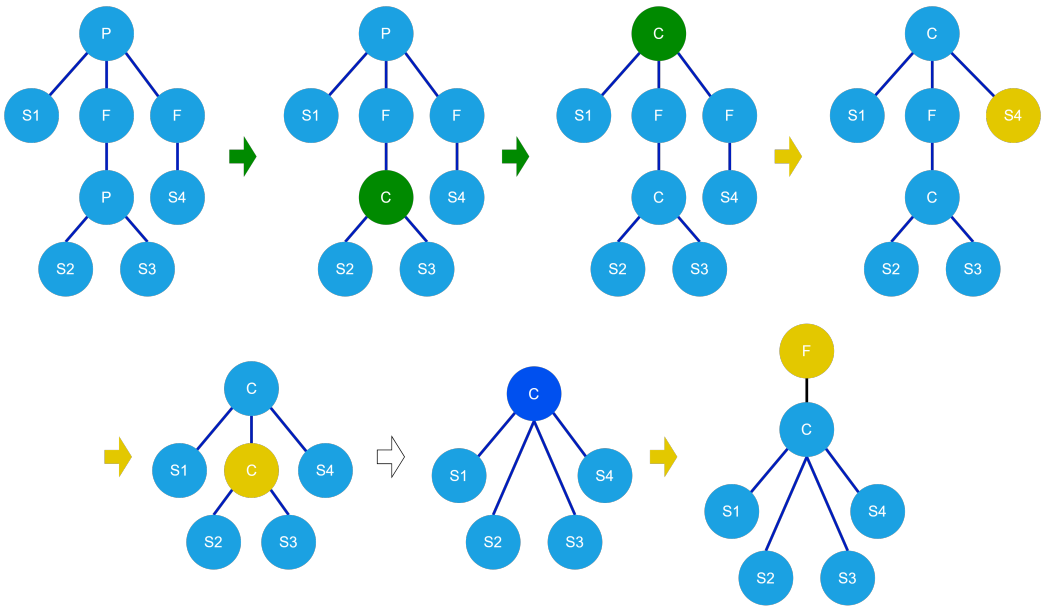


Figure 14.3: The Figure shows that the normal form and the initial tree are functionally equivalent. We mark the usage of rule 14.1 with the green arrow, while yellow arrows indicate the application of the rule 14.2. Note that at each step except the last one, the service time of the tree increases.

the emission (t_e) and the collection time (t_c) for the farms. Besides, we also know the parallelism degree of the farms. In particular, the first farm has 4 threads while the second one has 8 threads. Using formula 3.1 regarding the pipeline service time, and Formula 3.13 concerning the farm service time, we have that the service time of the initial tree is:

$$T_s = \max\{t_1, \max\{t_e, t_c, \frac{\max\{t_2, t_3\}}{4}\}, \max\{t_e, t_c, \frac{t_4}{8}\}\}$$

That, assuming the collection and the emission time are not the bottlenecks of the farms, it is equal to:

$$T_s = \max\{t_1, \frac{\max\{t_2, t_3\}}{4}, \frac{t_4}{8}\}$$

Concerning the normal form, we instead have:

$$T_s = \max\{t_e, t_c, \frac{\sum t_i}{n_w}\}$$

That, assuming again the collection and the emission time are not the bottlenecks of the farms, it is equal to:

$$T_s = \frac{\sum t_i}{n_w}$$

So, if, for instance, the service time of the initial tree is $\frac{t_4}{8}$, then we just need to pick a sufficient number of workers in the normal form to reach a time smaller than $\frac{t_4}{8}$:

$$n_w \cdot \frac{t_4}{8} = \sum t_i \quad \leftrightarrow \quad n_w = \frac{8 \sum t_i}{t_4}$$

Using that number of workers, and assuming that t_e and t_c are negligible, the service time will be, at least, equal to the initial tree time. Besides, the number of resources used in the normal form will be smaller since we do not have the many intermediate nodes we have in the initial tree between the farms.

14.3 Optimization rules

Considering also rules such as 14.4 allow us to apply some optimization rules. For instance, if, for each farm, we have to set the number of workers, we could use an optimization rule to figure out the best solution given the computer resources.

Suppose, for example, that our computer has 8 parallel resources and that we have to set the number of workers for the farms in Figure 14.4. We also know that S1 lasts 10 seconds, S2 40 seconds, and S3 60 seconds.

As already mentioned in previous chapters, we want that all pipeline stages have approximately the same service time - our optimization rule. We thus may give four threads to the first farm, six to the second, and so each stage will have a service time of 10 seconds. However, this solution requires:

$$T_{par} = \underbrace{1}_{\text{S1}} + \underbrace{4}_{\text{1° farm}} + \underbrace{6}_{\text{2° farm}} + \underbrace{4}_{\text{emitters and collectors}} = 15$$

That is much higher than 8, which is the number of parallel resources available. Hence, we need to reduce the number of workers. In this case, the best solution is to give one worker to the first farm and two to the second farm. In

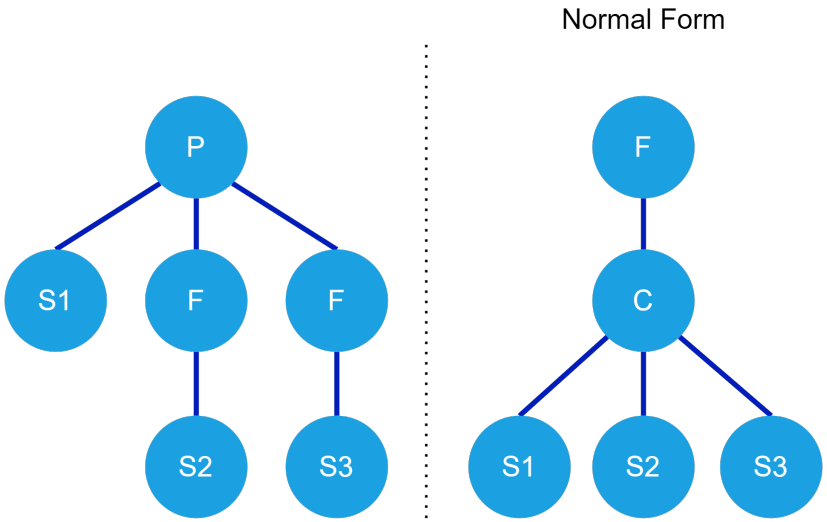


Figure 14.4: On the left, our reference pattern tree for examples. On the right, the corresponding normal form

this way, the service time will be 40 seconds, which is surely better than the 60 seconds with one worker for each farm. The number of resources required by this solution is 8:

$$T_{par} = \underbrace{1}_{\text{S1}} + \underbrace{1}_{\text{1° farm}} + \underbrace{2}_{\text{2° farm}} + \underbrace{4}_{\text{emitters and collectors}} = 8$$

Finally, note that the normal form refactoring offers a better service time, with the same number of resources. Indeed, we can have a farm with six workers (plus two for emitter and collector) and get a service time equal to (3.13):

$$T_s = \frac{10 + 40 + 60}{6} = \frac{110}{6} \approx 18.34s$$

14.4 RPLsh

We can automatize the process using RPLshell: a shell where we can express pattern compositions and apply refactoring rules. We leave here the GitHub link: https://github.com/t-costa/rplsh.

14.5 Automatic management

Given a set of **composable** parallel patterns, a performance model, and a set of refactoring rules, we can do what is called **autonomic management** (of NF features). An automatic management cares of non-functional features to deal with different situations that can appear during the execution of the program.

Consider, for instance, having a pattern tree that represents our application. The set of refactoring rules allow us to switch from one pattern tree to another. The performance model instead enables us to know the service time of each configuration. Everything is fine if our present pattern tree has a service time equals 10 msec, and the current interarrival time between tasks is 10 msec. But, if, for some reason, at some time, the interarrival time becomes 1 msec, what happens is that our program will not be able to handle the load of tasks. Here comes in the automatic manager, which will switch to a pattern tree more suitable for the situation. As soon as the interarrival time returns to 10 msec, the automatic manager will again shift to the original pattern tree because otherwise, we would waste resources.

14.5.1 MAPE loop

A standard way to implement an automatic manager is the Monitor Analyse Plan Execute (MAPE) loop. The MAPE loop, as the name suggests, consists of four stages iteratively executed in a loop in the following order:

1. **Monitor**: receives data from the **sensors** installed on the application. Thus, we need to add "sensors" to the application.
2. **Analyse**: sets up premises for the planning phase.
3. **Plan**: plans the actions to be executed.
4. **Exec**: execs the action using the **actuators** installed on the application. Hence, we also need to add "actuators" to the application.

Let us for instance consider a farm with 10 workers. Then:

The **sensors** are the methods that can be called to know T_a, T_s, T_w, T_e and T_c. Where T_a is the interarrival time, T_s is the farm service time, T_w is the time a worker need to accomplish a task, T_e is the emitter time, and eventually T_c is the collector time.

The **actuators** are the methods that can be called to modify the **non-functional** program behaviours (e.g. increase/decrease the number of workers).

The corresponding MAPE loop may do the following things:

1. **Monitor**: gathers measures: $\{T_a, T_s, T_w, T_e, T_c\}$
2. **Analyse**: look at $T_a \approx T_s$. If $T_a \gg T_s$, we are wasting resources, so we should decrease the workers' number. On the other hand, if $T_a \ll T_s$, our program is cold sweating, and therefore we should increase the number of workers.
3. **Plan**: If $T_a \ll T_s$ then we have to call increase(nw) - an actuator - that will increase the workers' number of k. If $T_a \gg T_s$ then we have to call decrease(nw) that will decrease the workers' number of k. k is computed so that $T_a \approx T_s$ again.
4. **Exec**: Use the actuators to increase/decrease nw by k.

So, we implement the MAPE loop, adding sensors, actuators, and a manager (MGR) that executed the MAPE loop. Sensors are the easy part of the story since we just need to deliver an API that allows the MGR to read measures.

Actuators

Implementing actuators can be a pain. In particular, implementing an actuator that changes the structure of the pattern tree can be arduous. For example, consider the actuator farmelim, which substitutes a farm with a sequential execution. That would require creating a new thread for the sequential code and then redirect the tasks from the farm to the new sequential node.

A standard strategy consists of keeping both the sequential and the farm parts ready for execution. Then a director node, according to the MGR, redirects the incoming tasks towards an implementation. Eventually, a collector node maintains a queue that receives the results from both implementations. Of course, that means we will keep occupied several threads to implement both the implementations. So, when we use the sequential implementation, we will not use the threads for the farm. And vice-versa, when we use the

farm, we will not use the thread for the sequential part. However, inactive threads do not consume computing resources but occupy only memory. Moreover, sooner or later, the code will be swapped out from the main memory, and therefore in the long term, it is not a big problem.

Finally, note that adding a director node means increasing the overhead.

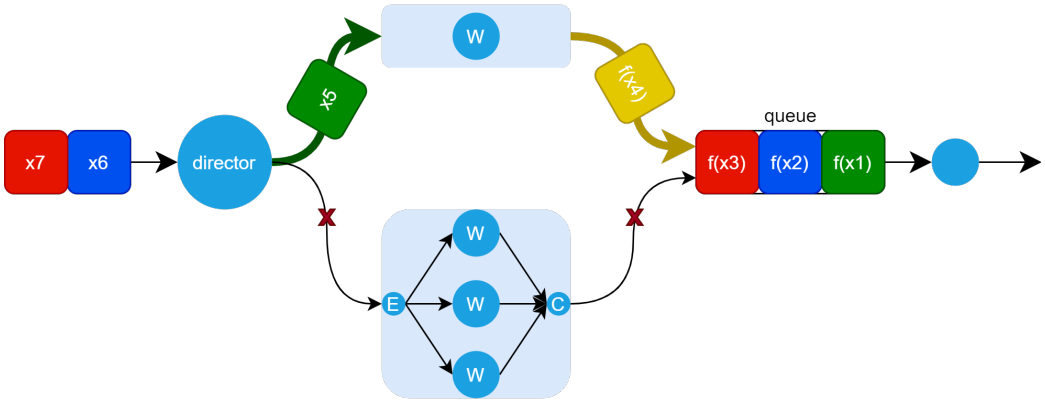


Figure 14.5: The standard strategy to implement an actuator that changes the pattern tree. In this case, the director is sending the new tasks to the sequential code

MGR

We usually implement the MGR using a set of decision rules - a set of condition action rules. Every rule consists of two parts:

Predicate over the measurements coming from the sensors.

Action: a sequence of actuators invokes.

So, our MGR will take the measurements from the sensors and look for the rules whose predicate is true. Among those rules, with some priority, it will choose the actions listed in the action part of the rules and exec them.

Taking again our example, a simple set of decision rule may be:

$T_a \ll T_s \rightarrow \text{activate}(nw++)$

$T_a \gg T_s \rightarrow \text{activate}(nw--)$

$T_a \gg T_s \;\&\&\; nw == 2 \rightarrow \text{activate}(\text{farmelim})$

Of course, we can extend the reasoning to any other kind of pattern, such as the pipelines.

14.5.2 Behavioural Skeleton

Let us now consider the case with a composition of patterns, as in Figure 14.4. What may happen is that the user asks the MAPE loop to have a specific service time, and the MGR will try to accomplish the "contract".

So, assume we request a service time of 5 seconds for the pattern tree on the left in Figure 14.4. We will assume that S1 has a latency of four seconds, S2 ten seconds, and S3 twenty seconds. The steps the MGR would do are the following:

1. The first element that receives the request is the root of the tree, which, in this case, is a pipeline. The pipeline "thinks": "I am a pipeline, how do I reach a service time of 5 seconds? Right, all my stages must have a service time that is less or equal to 5 seconds!". Thus, the pipeline requires a service time that is less or equal to 5 seconds to all its direct descendants.

2. S1 has a latency of 4 seconds, so it is ok.

3. The first farm has to deal with a task requiring ten seconds. Assuming T_e and T_c are less than 5 seconds, the farm needs two workers to offer a service time of 5 seconds. Indeed, using 3.13, the service time is then $\frac{10}{2} = 5$.

4. Using the same reasoning, the third farm needs four workers to reach a service time of 5 seconds.

So, the procedure percolates flows the initial service time requirements down to the other nodes so that the contract we send to lower nodes depends on the contract we receive and which pattern we are in.

What if, for instance, S3 requires 25 seconds, and the third farm can not use more than four workers, so the contract fails:

1. The farm manager reports failure to the pipeline, saying that the best it can do is $\frac{25}{4} = 6.25$.

2. The pipeline can do various operations. For example, it can propagate the new service time of 6.25 to all its direct descendants. That is because it is useless that other stages work at a service time of 5 or less if there is a bottleneck that can not achieve more than 6. In this case, nothing changes.

So, in general, whenever a node fails to accomplish the contract, it informs who provided it the contract.

What we have done is called Behavioral skeleton. It was developed by a project founded by the EU called CoreGRID.

Chapter 15

State access patterns

Until now, we have discussed patterns where each parallel resource was a pure function. In this chapter, we will consider the case where there is a shared state among the different parallel resources, and therefore their output depends also on this shared state. For instance, viewing a pipeline, the output of the ith stage will depend on both x_i and the current state s:

$$f_i(x_{i-1}, s)$$

What is a state? A state is a data with a given structure accessed from different processing elements in the computation. The data can be either **simple** as a Boolean or **complex** as an object. Same for access; a processing element may either simply set a Boolean flag to 0 (simple) or increase an integer (complex).

Now, the question is, how do we maintain the state consistent?

In the simplest case, where both the data and the accesses are simple, we do not have to do anything. For instance, suppose that the data is a Boolean variable x and that the access consists of setting x to False. If we consider a shared memory multi-core environment, with four cores, each with its level 1 cache, a shared level 2 cache, and global memory, the operation of setting x to False, is merely a single memory operation: `STORE #0 -> M[addr(x)]`. The STORE operation is under the management of a memory arbiter, a hardware entity, that arbitrates the access to the memory and allows only one operation at a time for a specific cell in memory. Therefore, the memory arbiter will do the job of maintaining the state consistent.

If the data is simple, but the access is complex, we will probably need to use mutexes or atomic variables. Indeed, consider having an integer variable that the processing elements must increase during the computation. In this case, the instructions the processor must perform are:

1. `LOAD M[addr(x)] -> Ri`
2. `ADD Ri + 1 -> Ri`
3. `STORE Ri -> M[addr(x)]`

And as you could imagine, depending on how the scheduler lets the threads work, it could bring to inconsistent states.

Finally, if both the data and the access are complex, we will need some policies to regulate the accesses. In particular, these accesses will need some mechanisms to implement the policies.

Knowing how the parallel resources are logically arranged can let us implement optimizations we could not perform in an unstructured case. For instance, consider a farm, where each worker has its queue to receive the data from the emitter, and there is one single queue between the workers and the collector. Therefore, in the queues between the emitter and the workers, we do not have to reason about concurrency problems because only one thread writes and only one thread reads. On the other hand, in the queue between the workers and the collector, we have to manage the write accesses.

15.1 Odd Even Sort Example

The odd even sort is a silly sorting algorithm that at each iteration, given a vector x, works in two phases:

1. **Even phase**: takes pairs whose first element has even index (e.g. 0, 2, 4 and so on), and for each of these pairs $< x[i], x[i+1] >$:

   ```
   if (x[i] > x[i+1]) then swap(x[i],x[i+1])
   ```

2. **Odd phase**: takes pairs whose first element has odd index (e.g. 1, 3, 5 and so on), and for each of these pairs $< x[i], x[i+1] >$:

   ```
   if (x[i] > x[i+1]) then swap(x[i],x[i+1])
   ```

The algorithm keeps going until at least one swap was performed during the iteration.

Example 15.1.1 Odd Even sort

Consider having the sequence: 2 5 1 3 4 7 8 9. The algorithm will perform the following steps:

1. **Even phase**: consider the pairs $< 2,5 >$, $< 1,3 >$, $< 4,7 >$, $< 8,9 >$. Nothing swapped.
   Odd phase: consider the pairs $< 5,1 >$, $< 3,4 >$, $< 7,8 >$. $< 5,1 >$ swapped in $< 1,5 >$.
   The resulting array is 2 **1 5** 3 4 7 8 9

2. **Even phase**: consider the pairs $< 2,1 >$, $< 5,3 >$, $< 4,7 >$, $< 8,9 >$. $< 2,1 >$ swapped in $< 1,2 >$, and $< 5,3 >$ swapped in $< 3,5 >$.
   The resulting array is **1 2 3 5** 4 7 8 9
   Odd phase: consider the pairs $< 2,3 >$, $< 5,4 >$, $< 7,8 >$. $< 5,4 >$ swapped in $< 4,5 >$.
   The resulting array is 1 2 3 **4 5** 7 8 9.

3. **Even phase**: consider the pairs $< 1,2 >$, $< 3,4 >$, $< 5,7 >$, $< 8,9 >$. Nothing swapped.
   Odd phase: consider the pairs $< 2,3 >$, $< 4,5 >$, $< 7,8 >$. Nothing swapped.

We can see the odd even sort as a data-parallel computation. In particular, we can perform a MAP over even pairs and then a MAP over odd pairs. So, it is a composition of MAPs. We can imagine using a flag such that:

- is set to True at the beginning of each iteration.
- is set to False if we swap some elements during the iteration.

The algorithm terminates when this flag is still True at the end of the iteration. This flag is our state. In code:

Listing 15.1: odd even sort

```
do {
    flag = true
```

```
        map F over "even pair"
        F: if (first > second) {
            swap(first, second)
            flag = false
        }

        followed by

        map F over "odd pair"
        F: if (first > second) {
            swap(first, second)
            flag = false
        }

    }while(not(flag))
```

How do we keep the flag consistent? Here we present three ways, from the worst to the best:

- Define a flag as an atomic variable.
- Do nothing since we just set the flag to false, and therefore the memory arbiter will do the job for us.
- Use local flags, that is, one flag for each worker. Then we perform a reduction of the local flags. In particular, we put them in AND and use the result to update the general one.

The first solution is worse than the second because it adds useless overhead (with mutexes). The third solution is better than the second because it avoids using the cache coherency protocol (section 7.2).

Indeed, suppose to have two threads with their level 1 cache. Each cache will have a copy of the partition of the array the thread must process and the flag. If we use a global flag, as in solutions 1 and 2, when the thread in the first core updates it to False, then the cache coherency protocol will also update the flag in the other cache. The same happens when the thread in the second core does the same. That means there is significant traffic of the cache coherency protocol.

15.2 Farm Example

Let us now consider a farm where the workers also have a shared global state. Each worker computes the output for the collector y_i and the new state value s':

$$y_i = f(x_i, s)$$
$$s' = g(x_i, s)$$

As you can notice, both the computations depend on x_i and the current state s. Let us suppose that T_f is the time needed to compute y_i, while T_{st} is the time needed to compute s'. Assuming that the farm has nw workers, its service time will be:

$$T_s = \frac{T_f}{nw} + T_{st}$$

T_{st} is not divided by nw because we must access the state in mutual exclusion. Therefore, using 3.11, the farm completion time is:

$$T_c \approx m \cdot T_s = m \cdot (\frac{T_f}{nw} + T_{st})$$

Finally, indicating with m the number of tasks, we can compute the farm speedup (3.3):

$$sp(nw) = \frac{T_{seq}}{T_{par}(nw)} = \frac{m \cdot (T_f + T_{st})}{m \cdot (\frac{T_f}{nw} + T_{st})}$$

So, the maximum speedup is:

$$\begin{aligned} \lim_{nw \to \infty} sp(nw) &= \lim_{nw \to \infty} \frac{T_f + T_{st}}{\frac{T_f}{nw} + T_{st}} \\ &= \lim_{nw \to \infty} \frac{T_f + T_{st}}{T_{st}} \\ &= \lim_{nw \to \infty} \frac{T_f}{T_{st}} + \frac{T_{st}}{T_{st}} = 1 + \frac{T_f}{T_{st}} \end{aligned} \tag{15.1}$$

So, the ratio between T_f and T_{st} dictates the maximum speedup we can achieve. The maximum is the difference between the two, the better is the achievable speedup. So, we obtain a result very similar to the one we get with the Amdhal Law.

15.3 Stream parallel patterns (only)

In this section, we will present some access patterns that are significant for stream parallel patterns:

- **Read only state**: threads can only read the state.
- **Owner writes**: the state is split into portions, each of which can be modified only by a specific thread. Consider, for instance, an array where we associate position i to thread Th_i. Using this pattern, only thread Th_i can write in position i; all the other threads can only read the value in that position.
- **Accumulator**: we obtain the current state as a combination of the previous states plus some values. The plus operation has to be associative and commutative. So, the order in which threads perform the plus operations does not matter.
- **"resource" state**: denote states we do not know anything, So, it is the worst situation. For instance, the previous Farm example is this kind of pattern.

15.3.1 Read only state

In many cases, we can build this access pattern so that there is no extra overhead due to concurrent read operations. The possible strategies are:

- Let the memory subsystem manages local copies (in cache) of the state.
- Create explicitly copies of S near the processing elements.

15.3.2 Owner writes

Usually, in these patterns, the threads do not read the positions associated with other threads. However, it is the main application that regularly reads the whole state to know its current status. If it is that case, we can avoid any synchronization mechanism.

15.3.3 Accumulator

We can distinguish two cases:

- We do not mind intermediate values. In this case, we can have a farm where each worker updates their local state. Then, the collector aggregates all the local results. For more explanations, look at Figure 15.1.

- We mind intermediate values. Each worker should update a global state S. We will therefore need synchronization mechanisms.

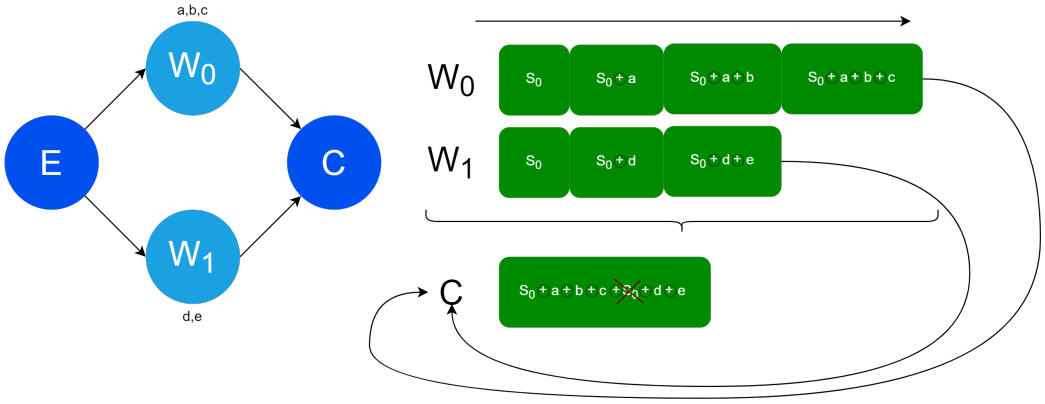


Figure 15.1: Consider a farm with two workers. The first worker "adds" the values a, b and c, while the second one "adds" d and e. The operations are performed on a local copy of S_0. Collector C aggregates the local computations to get the final result. Note that it deletes an S_0 from the final computation because otherwise, there would have been two S_0.

15.3.4 "resource" state

Let us indicate with:

- T_s: time spent to operate on state s

- T_f: time spent to produce the output item given the input one.

If updating the state is made independently of the computation of the value we deliver to the output stream, we have the same situation discussed in the Farm example. On the other hand, if computing the output value is not independent of computing the new state value, each worker must read the current state, calculate the output value, and finally update the state. So, the computation is sequential, and we can not do many things without further assumptions.

Monotonic updates

We could assume that the state values increase or decrease monotonically, that is, $s_i \leq s_{i+1}$ or $s_i \geq s_{i+1}$. In that case we could assume to have a farm where:

- The emitter broadcasts the state value to all workers.
- All workers send the updates of the state to the collector.
- The collector checks whether the new state value is less (or greater) than the current one. If the state changes, it feed-backs the new value to the emitter.

Example 15.3.1 Monotonic updates

Suppose that the initial state value is 100, that we accept only decreasing values, and that our farm has two workers (W_A and W_B). Then, a possible sequence of things that could happen is:

1. At the beginning W_A and W_B know that $s = 100$.
2. W_A computes $s' = 80$ knowing $s = 100$.
3. $s' < s$, therefore the collector feed-backs the new value 80 to the emitter.
4. W_B computes $s' = 90$ knowing $s = 100$.
5. $s' > s$, therefore the collector does not do anything.
6. The emitter broadcasts the new state value.

From the example above, we can see that at step 4. W_B has computed something based on an old state value. That could cause some extra overhead, but if everything is right, updates should be rare, and most of the workers will look at the correct state value.

We can use this method, for example, when we are looking for the global minima of a function. We split the function domain and assign each portion to a worker. Then, each worker W_i, given a x_i, computes $s' = f(x_i)$ and the state is updated if $s' < s$.

For more information, we suggest to read the paper "State Access Patterns in Stream Parallel Computations".

15.3.5 Take-away message

The take-away messages are:

- We always have to look at patterns in state accesses because they are a source of optimization. We should pair these patterns with computation patterns (patterns used to organize parallel activities in computation).
- State access patterns may be reused

Chapter 16

GPU

GPU stands for Graphics Processing Unit. They indeed were initially designed to support only operations on the screen. They supported data-parallel "collective" operations, mostly maps and stencils. There were properly compiled graphic libraries that the programmer could use transparently.

Then, GP-GPU (General Purpose GPU) came into play, bringing more functionalities to operate on data collections independently of if these data collections were for display something on-screen or not. The drawback was that GP-GPU required (and requires) special programming languages (e.g. CUDA for Nvidia).

Pros and cons of using CPUs or GPUs are presented in Figures 16.1 and 16.2.

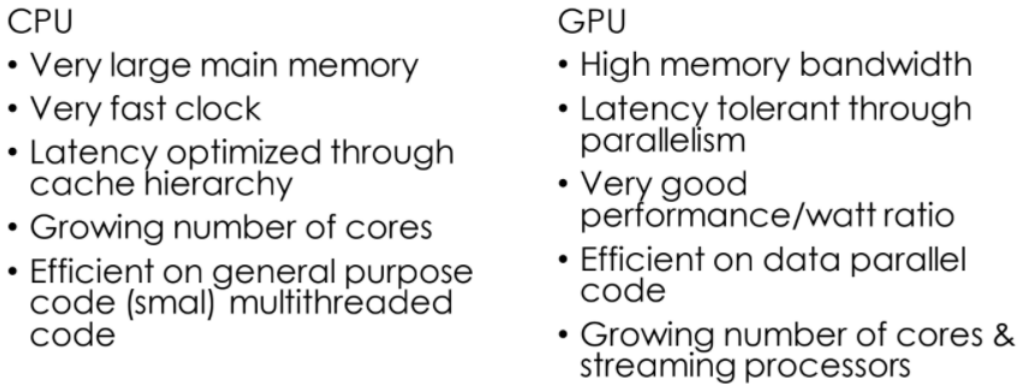


Figure 16.1: Pros of using CPUs or GPUs

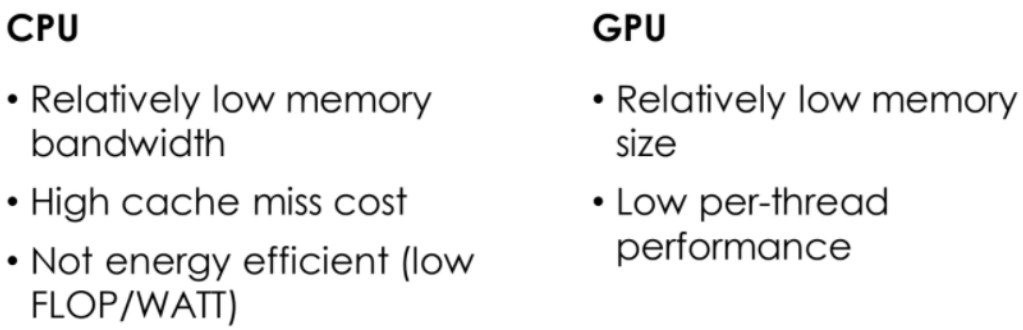


Figure 16.2: Cons of using CPUs or GPUs

Each GPU has a global memory accessible from everywhere but whose access cost is high. Then each group of cores has a local memory of limited size that is ultra-fast. The cores alone do not decide anything. Indeed, cores are divided into groups, each associated with a streaming multiprocessor (SM). The SM fetches, decodes, and broadcasts the instructions to its groups. The cores merely receive the instructions and execute them. Figure 16.3 shows a schematic of an Nvidia GPU architecture, where we can see the division in groups of the cores, each associated with an SM.

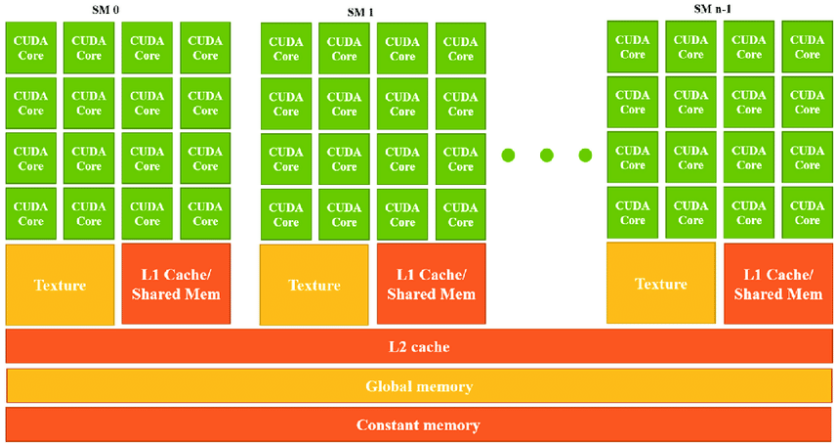


Figure 16.3: Schematic of NVIDIA GPU architecture, where SM refers to streaming multiprocessor.

From Figure 16.3 we can also notice that the number of cores is extremely

high (at the moment, we can reach approximately 3000 cores)

16.1 SIMD programming model

GPUs use a SIMD programming model, which is the same used for vectorization. Due to the SIMD execution model, we could have problems when the code is **divergent**. For instance, the following code is not divergent:

```
for (i=0; i<N; i++){
    x[i] = x[i] + 1
}
```

On the other hand, an example of divergent code is shown below:

```
for (i=0; i<N; i++){
    if (x[i] mod 2 == 0)
        x[i] += 1
    else
        x[i] -= 1
}
```

In that case, the streaming multiprocessor should send some instructions to some cores and other instructions to other cores. However, it can only send the same instruction to all the cores. One solution could be to divide the even and odd entries of x and then send even elements to an SM and odd ones to another SM. An alternative is to use some of the solutions used in vectorization. That is, rewrite the code as follow:

```
for (i=0; i<N; i++){
    x[i] += (x[i] mod 2 == 0)*1 + (x[i] mod 2 != 0)*-1
}
```

In this way, the SM can send to all its cores the same instruction.

When programming a GPU (or an accelerator in general), we usually follow these steps:

1. Declare a kernel, that is the code to be executed on the GPU.

2. Declare data that should be transferred in GPU memory.

3. Move in data to GPU from main memory. We can do this by moving word by word under the control of the processor or using a DMA transfer.

4. Execute the kernel.

5. Move out data from GPU to main memory.

16.1.1 GPU-friend computations

GPUs are suitable for the following computations:

- **Data parallel computations**; maps and stencils mainly. The Reduce/Prefix operations need some care.
- **FLOP intensive computations**, that are those executing more floating point operations on the same data.

Let us take an example to clarify what we mean with FLOP intensive computations. Consider the operation of adding two vectors of size N. To compute that on the GPU, we need to:

1. Transfer the two vectors from the main memory to the GPU memory ($2O(N)$).
2. Compute the sum ($O(N)$).
3. Transfer the result from the GPU memory to the main memory ($O(N)$).

The time needed to transfer the data is higher than the time required to compute the addition. So, do this operation using the CPU would probably have demanded less time because there would not have been any transfer time. In this case, we say that the operation is not FLOP intensive. On the other hand, let us consider the multiplication of two squared matrices of shape $N \times N$. As before, let us see the steps to compute it in the GPU:

1. Transfer the two matrices from the main memory to the GPU memory ($2O(N^2)$).
2. Compute the multiplication ($O(N^3)$).
3. Transfer the result from the GPU memory to the main memory ($O(N^2)$).

In this case, the multiplication cost is one order of magnitude higher than the cost of transferring data. Therefore, the time spent for transfer data is worth to be spent. In this case, we say that the computation is FLOP intensive.

16.1.2 Programming model examples

There are different programming models to program a GPU:

1. CUDA/OpenCL where all operations are explicit. They represent the first generation of programming models for GPU.

2. OpenMP, AAC. Code annotations to identify/manage/execute kernels.

3. Sycl. Most of the operations hidden in proper library calls/language extensions

CUDA

Let us try to follow the steps presented above to perform a vector addition in a GPU. So, we first have to declare a kernel:

```
__global__ void vectorAdd(
        const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

The kernel is a `__global__` function where we write what the GPU should execute. In the beginning, we have to compute the index of the array that has to be processed: blockDim is the block dimension associated with the thread, blockIdx is the id associated with the block, and threadIdx is the id associated with the thread. Notice that we have to check we do not go out of the array bounds.

Then we have to allocate the vectors in the main memory:

```
int numElements = 50000;
size_t size = numElements * sizeof(float);

// Allocate the host input vector A
float *h_A = (float *)malloc(size);

// Allocate the host input vector B
float *h_B = (float *)malloc(size);

// Allocate the host output vector C
float *h_C = (float *)malloc(size);
```

And allocate the memory for each vector on GPU:

```
cudaMalloc((void **)&d_A, size);
cudaMalloc((void **)&d_B, size);
cudaMalloc((void **)&d_C, size);
```

Once we have allocated everything, we can copy the host input vectors A and B in main memory to the device input vectors in GPU memory:

```
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Now we can finally run the kernel:

```
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Eventually, we can copy the device result vector in GPU memory to the host result vector in main memory:

```
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

And free GPU and main memory:

```
//free GPU memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

```
// Free host (main) memory
free(h_A);
free(h_B);
free(h_C);
```

One of the improvements brought to CUDA after the first version is the **unified memory**. The idea is to move the memory management burden to compiler tools to improve the applicative programmer experience. In other words, we do not have to think anymore about transferring data from main memory to GPU memory, or vice versa. For example, write the following code, allocate accessible memory, that is accessible both from CPU and GPU:

```
 // Allocate Unified Memory -- accessible from CPU or GPU
 cudaMallocManaged(&x, N*sizeof(float));
 cudaMallocManaged(&y, N*sizeof(float));
```

Then, we can initialize the vectors in the CPU:

```
// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
}
```

We can launch the kernel:

```
// Launch kernel on 1M elements on the GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

And eventually wait for GPU to finish:

```
// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();
```

OpenCL

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. Conformant implementations are available from

Altera, AMD, Apple (OpenCL along with OpenGL is deprecated for Apple hardware, in favor of Metal 2), ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS [1].

OpenCL is quite similar to Cuda. In the following we present the code for vector addition using OpenCL:

```
int SIZE = 1024;

    // Allocate memories for input arrays and output array.
    float *A = (float*)malloc(sizeof(float)*SIZE);
    float *B = (float*)malloc(sizeof(float)*SIZE);

    // Output
    float *C = (float*)malloc(sizeof(float)*SIZE);


    // Initialize values for array members.
    int i = 0;
    for (i=0; i<SIZE; ++i) {
            A[i] = i+1;
            B[i] = (i+1)*2;
    }

    // Load kernel from file vecAddKernel.cl

    FILE *kernelFile;
    char *kernelSource;
    size_t kernelSize;

    kernelFile = fopen("vecAddKernel.cl", "r");

    if (!kernelFile) {

            fprintf(stderr, "No file named vecAddKernel.cl was found\n");

            exit(-1);
```

[1] https://en.wikipedia.org/wiki/OpenCL

```
}
kernelSource = (char*)malloc(MAX_SOURCE_SIZE);
kernelSize = fread(kernelSource, 1, MAX_SOURCE_SIZE, kernelFile);
fclose(kernelFile);

// Getting platform and device information
cl_platform_id platformId = NULL;
cl_device_id deviceID = NULL;
cl_uint retNumDevices;
cl_uint retNumPlatforms;
cl_int ret = clGetPlatformIDs(1, &platformId, &retNumPlatforms);
ret = clGetDeviceIDs(platformId,
        CL_DEVICE_TYPE_DEFAULT, 1, &deviceID, &retNumDevices);

// Creating context.
cl_context context = clCreateContext(NULL, 1, &deviceID,
                                     NULL, NULL,  &ret);


// Creating command queue
cl_command_queue commandQueue = clCreateCommandQueue(context, deviceID,
                                                     0, &ret);

// Memory buffers for each array
cl_mem aMemObj = clCreateBuffer(context,
    CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
cl_mem bMemObj = clCreateBuffer(context,
    CL_MEM_READ_ONLY, SIZE * sizeof(float), NULL, &ret);
cl_mem cMemObj = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, SIZE * sizeof(float), NULL, &ret);


// Copy lists to memory buffers
ret = clEnqueueWriteBuffer(commandQueue,
    aMemObj, CL_TRUE, 0, SIZE * sizeof(float), A, 0, NULL, NULL);;
ret = clEnqueueWriteBuffer(commandQueue,
    bMemObj, CL_TRUE, 0, SIZE * sizeof(float), B, 0, NULL, NULL);
```

```
// Create program from kernel source
cl_program program = clCreateProgramWithSource(context,
    1, (const char **)&kernelSource,
    (const size_t *)&kernelSize, &ret);

// Build program
ret = clBuildProgram(program, 1, &deviceID, NULL, NULL, NULL);

// Create kernel
cl_kernel kernel = clCreateKernel(program, "addVectors", &ret);


// Set arguments for kernel
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&aMemObj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&bMemObj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&cMemObj);


// Execute the kernel
size_t globalItemSize = SIZE;
// globalItemSize has to be a multiple of localItemSize.
// 1024/64 = 16
size_t localItemSize = 64;
ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
    &globalItemSize, &localItemSize, 0, NULL, NULL);

// Read from device back to host.
ret = clEnqueueReadBuffer(commandQueue, cMemObj, CL_TRUE, 0,
    SIZE * sizeof(float), C, 0, NULL, NULL);

// Write result
/*
for (i=0; i<SIZE; ++i) {
        printf("%f + %f = %f\n", A[i], B[i], C[i]);
}
*/

// Test if correct answer
```

```
for (i=0; i<SIZE; ++i) {
        if (C[i] != (A[i] + B[i])) {
                printf("Something didn't work
                    correctly! Failed test. \n");
                break;
        }
}
if (i == SIZE) {
        printf("Everything seems to work fine! \n");
}

// Clean up, release memory.
ret = clFlush(commandQueue);
ret = clFinish(commandQueue);
ret = clReleaseCommandQueue(commandQueue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(aMemObj);
ret = clReleaseMemObject(bMemObj);
ret = clReleaseMemObject(cMemObj);
ret = clReleaseContext(context);
free(A);
free(B);
free(C);
```

SYCL

Well, both CUDA and OpenCL code appears and are a bit complex. As already mentioned, SYCL is an evolution of OpenCL that tries to simplify the programmers' life. In the following, we write the `vectorAdd` code for SYCL:

```
main(){
    vector h_a(SIZE), h_B(SIZE), h_C(SIZE); //host vectors
    ...
    initialize host vector's data
    ...
    {
```

```
            Buffer d_a(h_a); //automatically copies data to device (GPU)
            Buffer d_b(h_b); //also, allocates device memory
            Buffer d_c(h_c); //with the same size of host memory

            queue q;
            command_group(q, [&]() {
            auto a = d_a.get_Access<access::read>();
            auto b = d_b.get_Access<access::read>();
            auto c = d_c.get_Access<access::write>();
            }

            parallel_for(SIZE, vector_addition([=](id <> item)){
                int i = item.get_global(0);
                c[i] = a[i] + b[i]
            }
        }
    }
```

16.2 Exploiting Parallelism in kernel workflows

In this section, we will try to exploit parallelism in between data transfers (DMA concern) and kernel execution (GPU concern).

16.2.1 Pipeline

We can see the MAP computation as a composition of three operations: 1) transfer data to GPU, 2) compute, 3) transfer results from GPU. In formulas:

Map(f) = Map(Comp(transfer data to GPU, compute, transfer results from GPU))

As we have already seen, we can turn a composition into a pipeline:

Map(f) = Map(Pipe(transfer data to GPU, compute, transfer results from GPU))

Note that we will need a system with a 2 way DMA. If the pipeline is balanced - all steps require the same order of magnitude in time - then we will reach a speedup of 3.

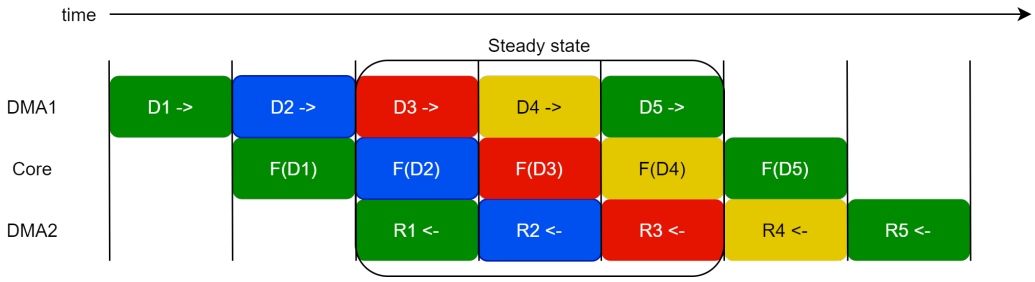


Figure 16.4: Pipe a map. The first DMA transfers data from the host to the device. The cores compute the map. Finally, the second DMA transfers data from the device to the host.

With Nvidia GPUs, we can do this using CUDA STREAMS. The operations are directed to the GPU through streams, which are lists of commands the cores in the GPU will have to execute. For instance, the sequence [operation1, operation2, operation3] represents a stream. The operations in the same stream are serialized, which means that the GPU will execute them sequentially. Therefore, the GPU will perform operation1, then operation2, and eventually operation3. However, operations in different streams can overlap.

We can apply the same optimization to part of the computations (MAP to MAP). So, given a MAP(MAP(f)), we can turn it into MAP(PIPELINE(TRANSFER, COMPUTE KERNEL, TRANSFER)).

16.3 Memory

Another aspect we have to consider when programming GPU programs is the memory hierarchy. The hierarchy consists approximately of three parts:

1. The global memory.
2. The shared memory among threads of the same block.
3. The memory for the single thread.

The closer we are to the thread, the faster the memory is, and the higher the bandwidth of the memory is. Data is initially transferred into the global memory. Then, when we schedule threads for executions, data is moved from

the global memory to the shared or thread memory. Moving data from and to the different levels of the hierarchy leads to some overhead.

As for the main memory of the host, also for the GPU, we should exploit locality as possible we can. Moreover, we should do aligned accesses (coalesced accesses). "Aligned accesses" means that accesses do not overlap each other. So, we may take data needed for a group of threads more or less at once.

Concluding, the GPU code developmenet workflow is the following:

1. Identify potential kernels
2. Manage thread/group allocation
3. Optimize data transfers
4. Optimize memory usage
5. Monitor/profile execution resources