Parallel and Distributed Systems

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction 2
0.2 General Paradigms of Parallel Programming 2
0.3 Measures 4
0.3.1 Base Measurements 4
0.3.2 Derived Measurements 5
0.4 Technicalities 6
0.4.1 Threads 6

0.1 Introduction

Prof.: Marco Danelutto

Program Techniques for both parallel (single system, many core) and distributed (clusters of systems) systems. Principles of parallel programming, structured parallel programming, parallel programming lab with standard and advanced (general purpose) **parallel programming frameworks**.

Technical Introduction Each machine has more cores, perhaps multithreaded cores, but also GPUs (maybe with AVX support, which support operations floating point operations, **flops**, in a single instruction).
Between 1950 and 2000 the VLSI technology arised, integrated circuits which nowadays are in the order of 7nm (moving towards 2nm): printed circuits!
In origin, everything happened in a single clock cycle: fetch, decode, execute, write results in registers, with perhaps some memory accesses. The we had more complex control where in a single clock cycle we do just one of the phases (fetch *or* decode *or*...), like a **pipeline**. More components are used the higher the frequency but the more power we need to dissipate, and we're coming to a point were the power we need to dissipate is too much and risks to melt the circuit, so we're reaching a **physical limit** in chip miniaturization. But temperature and computing power do not go in tandem: computing power is proportional to the chip dimensions, while temperature is proportional to the area. So it's better to put more processors (**cores**) and let them work together rather than make a bigger single processor. An approach is to have few powerful cores and more less powerful cores (for example, in the Xeon Phi processors). Now, the processors follow this architecture, with the performance of a single core decreasing a bit with every generation but it's leveled by adding more cores.

Up to the 2000, during the single core era, code written years before will run faster on newer machines. Now, code could run slower due to not exploiting more cores and the decreasing in performance of the single core.
With accelerators the situation is even more different: for example GPUs, accelerator for graphics libraries, with their own memory and specialized in certain kinds of operations. This can require the transfer of data between the accelerator's memory and the main memory, so the architecture of the accelerator is impactful on the overall performance.

0.2 General Paradigms of Parallel Programming

Parallelism Execution of different parts of a program on different computing devices at the same time. We can imagine different flows of control (sequences of instruction) that all together are a program and are executed on different computing devices. Note that more flows on a singe computing device is **concurrency**, not parallelism.

Concurrency Similar concept: things that *may* happen in parallel respecting the ordering between elements.

Computing Devices

Threads, implying shared memory

Processes, implying separated memories

GPU Cores

Hardware Layouts on a FPGA (Field Programmable Gate Array)

Sequential Task A "program" with its own input data that can be executed by a single computing entity

Overhead Actions required to organize the computation but that are note included in the program. For example: time spent in organizing the result. Basically, time spent orchestrating the parallel computation and not present in the sequential computation.

Speedup Fundamental things that we're looking for, it's the ratio between the sequential time and the parallel time.

$$\text{SpeedUp} = \frac{\text{Sequential time}}{\text{Parallel time}}$$

Assuming the best sequential time.
We have a slightly different measure, too

$$\text{Scalability} = \frac{\text{Parallel time with 1 computing device}}{\text{Parallel time}}$$

Stream of tasks In some cases it's not important considering just one computation but may be useful considering more computations and we want to optimize a set of tasks.

Example: Book Translation With $m = 600$ pages, for example. Let's assume I can translate a page in $t_p = 0.5$h. The sequential task is: take the book and spend time until I can deliver the translated book. The time is circa $m \cdot t_p = 300$h.
In parallel, ideally every page can be translated independently so I can split the book in two pieces of $\frac{m}{2}$ pages each (overhead), giving each half to a person. Both can translate at the same time, so ideally the time required is $\frac{m}{2} \cdot t_p$ for each, producing the translated halves. At this point I get the halves and produce the translated version (overhead). Ideally the time require is more or less $\frac{m}{2} \cdot t_p$, with "more or less" given by the time spent in splitting the book and reuniting the two halves. So the exact time is $T = T_{split} + \frac{m}{2} \cdot t_p + T_{merge}$.
What if the two person have different t_ps? For example $t_1 > t_2$. When a translator finishes, it spends some time synchronizing its work with me. With nw "workers" (translators, in this instance) $T = nw \cdot T_{split} + nw \cdot T_{merge} + \frac{m}{nw} T_{work}$ with $nw \cdot T_{split}$ time spent delivering work to each worker and $nw \cdot T_{merge}$ time in merging each result.

Init is the time where every worker has work to do, and finish is the time where the last worker finished working. So the exact formula is with a single T_{merge}.
So $\frac{m}{nw} T_{work}$ is the time that needs to happen, found in the sequential computation too, whereas the other two factors are **overhead**.

$$\text{SpeedUp} = \frac{\text{Best sequential time}}{\text{Parallel time}}$$

but the parallel time depends on the nw so

$$\text{SpeedUp}(nw) = \frac{\text{Best sequential time}}{\text{Parallel time}(nw)} \simeq \frac{\cancel{m} \cdot \cancel{t_p}}{\frac{\cancel{m}}{nw} \cdot \cancel{t_p}} = nw$$

This not taking into account the overhead. It's a realistic assumption because usually the time splitting the work is very small. But we have to take into account that, in case it's not negligible.

$$\text{SpeedUp}(nw) = \frac{m \cdot t_p}{\frac{m}{nw} \cdot t_p + \underline{nw \cdot T_{split} + T_{merge}}}$$

Example: Conference Bag $T_{bag} = t_{bag} + t_{pen} + t_{paper} + t_{proc}$ and with m bags we have $T = m \cdot T_{bag}$
We could build a pipeline, a building chain, with 4 people and each person does one task:

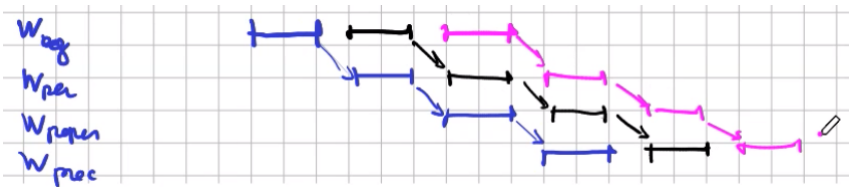
One takes the bag and gives to the next

One puts the pen into the bag and passes it

One puts the paper into the bag and passes it

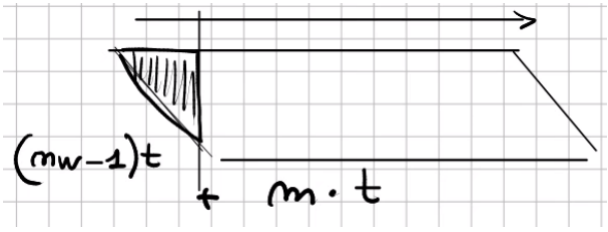
One puts the proceedings into the bag

So $w_b, w_{pen}, w_{paper}, w_{proc}$ workers. When the first worker has passed the bag, it could begin taking the next bag. Same for the others.



So in sequential we have $m \cdot (t_{bag} + t_{pen} + t_{paper} + t_{proc})$, and in parallel per 1 bag we have $t_{bag} + t_{comm} + t_{pen} + t_{comm} + t_{paper} + t_{comm} + t_{proc} + t_{comm}$ with t_{comm} spent passing the bag from one to the other, so total of $m \cdot T_{seq} + m \cdot t_{comm}$. But that's not correct, because we work in parallel: ideally we have a parallelogram of $m \cdot (t_{proc} + t_{comm}$ base, and we require $t_{bag} + t_{pen} + t_{paper} + 3 \cdot t_{comm}$ time to get up to speed and "fill the pipeline". But this required time is negligible, and in the end the overall time is given by the base of the parallelogram.

Pipeline With m tasks and nw stages, with the completion of the stage i required in stage $i+1$. So the output is $f_{nw}(f_{nw-1}(\dots f_1(x_i)\dots))$. With t time required for each stage.



We spend $(nw-1)t$ to get the last stage working and $m \cdot t$ time spent by the last stage to complete all the tasks.

$$T_{par}(nw) = (nw-1) \cdot t + m \cdot t$$

$$\text{SpeedUp}(nw) = \frac{(nw \cdot t) \cdot m}{(nw-1) \cdot t + m \cdot t}$$

So the higher the m is, the lower is the impact of the time required to get up to speed. So $m >> nw \Rightarrow T_{par}(nw) \simeq m \cdot t$

Throughput Tasks completed per unit of time.

0.3 Measures

On one side we can have more speed with more resources (computing devices). On the other side we can use more complex applications, with more resources. For example more precise computations, so extra resources not for improving the time but to improve the quality of the computations.
Finally, we could aim at computing results with less energy thanks to parallelism. This is a recent perspective on parallelism.

We've seen the $\text{SpeedUp}(n) = \frac{T_{seq}}{T_{par}(n)}$, where the plot has to lie below the bisection of the cartesian graph.

0.3.1 Base Measurements

Latency L Measure of the wall-clock time between the start and end of the single task.

Service Time T_s It's related to the possibility of executing more tasks. It's the measure of the time between the delivery of two consecutive results, for example between $f(x_i)$ and $f(x_{i+1})$
Even if x_i and x_{i+1} arrive at the same time, f would still be computing $f(x_i)$ so it'll start computing $f(x_{i+1})$ when it has finished.

Completion Time T_c The latency related to a number of tasks. $T_c = L \cdot m$ for $x_m, \dots, x_1$ inputs to a sequential system.
With a parallel system, instead, we have $T_c \simeq m \cdot T_s$.

Example A 3 stage pipeline, with each node being sequential and with latency L_i for node i.
At t_0 the first stage N_1 gets the first tasks and computes it in L_1, then N_2 computes in L_2 and N_3 computes in L_3 so a total of $t_0 + L_1 + L_2 + L_3$.
When the pipeline is filled, T_s is dominated by the longest L_i, so $T_s = \max\{L_1, L_2, L_3\}$ and $T_c = \sum L_i + (m-1)T_s$
If m is large with respect to n = number of stages, the "base of the parallelogram" would be very long, so $m >> n \Leftrightarrow T_c = m \cdot T_s$

0.3.2 Derived Measurements

SpeedUp

$$\text{SpeedUp}(n) = \frac{T_{seq}}{T_{par}(n)}$$

Could be latencies, service times... depending on what we want to measure the speedup of.

Scalability

$$\text{Scalability}(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

Efficiency

$$\text{Efficiency}(n) = \frac{\text{Ideal parallel time}(n)}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} = \frac{T_{seq}}{n \cdot T_{par}(n)} = \frac{\text{SpeedUp}(n)}{n}$$

Measures the tradeoff between what you gain with the speedup and the cost of the speedup.

Throughput

$$\text{Throughput} = \frac{1}{T_s}$$

Amdahl Law Taken the total time of a computation, T_{seq}, it can be divided into something that can and something that cannot be computed in parallel (for example, dividing the book is a sequential activity). So we can say that T_{seq} = serial fraction + parallel fraction and the **serial fraction cannot be parallelized**. $f \in [0,1] \mid f \cdot T_{seq}$ is the serial fraction.

$$T_{seq} = f \cdot T_{seq} + (1-f) \cdot T_{seq}$$

The parallel fraction can be splitted between the workers, but we would have to compute the serial fraction too. By splitting more and more and more, we have that

$$\lim_{n \to \infty} T_{par}(n) = f \cdot T_{seq}$$

$$\text{SpeedUp}(n) = \frac{T_{seq}}{f \cdot T_{seq}} = \frac{1}{f}$$

So we have a very low upper bound on the achievable speedup. This is referred to as **strong scaling**: strong meaning using more resources to get the computation faster.

Gustaffson Law

$$\text{SpeedUp}(n) = N - S \cdot (N-1)$$

With S being the serial fraction. This comes from the fact that we're considering a different perspective: Gustaffson assumes that the computation increases with the parallelism, something that's called **weak scaling**, getting the speedup from using more computational devices, using more data.

Cores In modern computers, we have a main memory (slow), a disk (even slower) and the memory is connected to at least 3 levels of cache. At the bottom we have some cores (4, 8...), each one has its own level 1 cache (usually split in data and instruction cache).
With an activity with a working set that fills the cache, in case of strong scaling splitting the computation across cores we process less data per core because the size of the problem is the same.
With weak scaling, we assume that the data increases so by using more cores we process the same data on all cores but the data grows so we could have extra overhead because of the working set size.
We will have patterns of parallel computation that differentiate in how we process the data.

Application as Graphs The applications can be seen as graphs of sequential nodes with dependencies. The maximum speedup is the work over the span, because in every case I need to go from the first to the goal node. I take the longest one because at least the longest path must be computed, and all the rest can be done in parallel and I assume to have enough resources to compute the rest in the time of the span.
We can use this model

0.4 Technicalities

Examples A simple program that "translates" an ASCII file by transforming lower letters into capital letters. We split the text into $n_{workers}$ parts, we wait for all the threads to finish and then verify the performances.
The translator is:

```
#include <string>

char translate_char(char c) {
  if (islower(c))
    return(toupper(c));
  else
    return(tolower(c));
}
```

0.4.1 Threads

We used to write instructions sequentially. At a given point now we `fork` another flow of computation: we get two flows that are executed together **in the same address space**, so the new thread inherits all the memory of the original thread.

Concurrence `todo`