Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula*†‡

Chapter 4 – Scalability in Parallel Processing

Yanik Ngoko[1] and Denis Trystram[2]

[1]Qarnot computing, France, yanik.ngoko@qarnot-computing.com
[2]Univ. Grenoble-Alpes, France trystram@imag.fr

*How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

†Free preprint version of this book: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2

‡Free preprint version of volume one: https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book

Abstract

The objective of this chapter is to discuss the notion of *scalability*. We start by explaining the notion with an emphasis on modern (and future) large scale parallel platforms. We also review the classical metrics used for estimating the scalability of a parallel platform, namely, speed-up, efficiency and asymptotic analysis. We continue with the presentation of two fundamental laws of scalability: Amdahl's and Gustafson's laws. Our presentation considers the original arguments of the authors and reexamines their applicability in today's machines and computational problems. Then, the chapter discusses more advanced topics that cover the evolution of computing fields (in term of problems), modern resource sharing techniques and the more specific issue of reducing energy consumption. The chapter ends with a presentation of a statistical approach to the design of scalable algorithms. The approach describes how scalable algorithms can be designed by using a "cooperation" of several parallel algorithms solving the same problem. The construction of such cooperations is particularly interesting while solving hard combinatorial problems. We provide an illustration of this last point on the classical satisfiability problem SAT.
Keywords: scalability, speed-up, isoefficiency, Amdahl's law, Gustafson's Law, performance analysis.
Relevant core courses: This material applies to ParAlgo courses.
Relevant PDC topics: Scalability in algorithms and architectures, speedup, Costs of computation, Data parallelism, Performance modeling
Learning Outcome: Students at the end of this lesson will be able to:

- Perform a classical speed-up analysis,
- Perform an efficiency and isoefficiency analysis,
- Understand the complementarity between Amdahl's and Gustafson's laws,
- Analyze the benefits of parallel processing,
- Determine the best approaches for designing a parallel program,
- Identify limitations in the parallelism of a program,
- Perform a trade-off analysis between time and energy consumption, and

- Envision alternative approaches for the design of scalable algorithms.

Context for use: This chapter is intended to be used in intermediate-advanced courses on the design and analysis of parallel algorithms. The material covers data parallelism, performance metrics, performance modeling, speedup, efficiency, Amdahl's law, Gustafson's law, and isoefficiency. It also presents an analysis of Amdahl's and Gustafson's laws when considering resource sharing techniques, energy-efficiency and problem types. The analysis could be too advanced for a CS2 student because it requires a background in modern parallel systems and computer architectures.

4.1 Introduction

Parallel machines are always highly powerful and complex (See the history of computing in [20]). This is obvious when we consider the evolution in the number of cores of top supercomputers in recent years [*]. This progression is driven by the conviction that with more powerful machines, we could reduce the running times in the resolution of challenging, compute-intensive problems such as real-time simulations (climate, brain, health, universe, etc.). On this latter point, let us emphasize that the experts are convinced that such simulations could be undertaken only with future exascale platforms [†].

At first glance, it might seem obvious that given a parallel algorithm and a machine, the running time of the algorithm while using x CPUs will be greater than the one we could expect with more than x CPUs. However, this is not necessarily true; indeed, the computation of a parallel algorithm is split between a computational part required for creating parallelism (a set of workers corresponding to threads or processes), computations required for running the concurrent workers, and those necessary for communication and synchronization. Given a more powerful machine (in term of cores or CPUs), the main option for reducing the running time of a parallel algorithm would consist in increasing the number of independent computations. However, this will probably induce more communication, synchronization, and a more important overhead for the creation of parallelism. Hence, it cannot be totally guaranteed that the gain induced by the *increase in parallelism* will be balanced by these additional operations.

This observation shows that we need a conceptual support to justify why supercomputers with more computational units could serve to tackle more efficiently compute intensive problems. Historically, the notion of scalability was introduced for this purpose. Roughly speaking, it describes the capacity of an algorithm to efficiently solve larger problems when it is executed on a machine with *more parallelism*.

The purpose of this chapter is three-fold. First, we intend to provide an understanding of scalability, deeper than the intuitive one. We define the concept, discuss its interest and introduce key metrics used for its quantification. The concept of scalability is also associated with two fundamental laws: Amdahl's and Gustafson's laws. Our second objective is to put these laws in perspective with the computability of problems, modern resource sharing techniques and the con-

[*] Details are available at `http://top500.org`

[†] `http://www.exascale-projects.eu/`

cept of energy-efficiency. Finally, we introduce a new statistical approach for improving the scalability of parallel algorithms.

4.2 Background on the scalability

We conclude that an algorithm is scalable from an analysis of its *behavior* when it is used in the processing of larger problems with more parallelism in the machine. For this purpose, we need metrics to characterize the behavior of a parallel algorithm. In this section, we will first introduce some classical metrics. Then, we will show how they can be used to analyze scalability.

4.2.1 Speedup and efficiency

[Speedup] Let us consider a parallel machine made of p computing units and a computational problem P. Let us assume an instance of P for which the sequential algorithm has a runtime equal to T_1. Finally, let us assume a parallel algorithm $\mathscr{A}$ whose runtime in the resolution of the instance on p computing units is T_p. Then, we define the speedup achieved by $\mathscr{A}$ when solving the problem instance as

$$S_p = \frac{T_1}{T_p}$$

The notion of computing units will depend both on the parallelism of the underlying machine and on the implementation of $\mathscr{A}$. Thus, these units might consist of cores, processors and even containers. For the sake of simplicity, in the rest of this chapter, unless otherwise stated, we will consider that computing units correspond to processors. This choice is debatable as a parallel algorithm might support different types of parallelism (cores, processors, *etc.*) However, the resulting conclusion might still hold in choosing a lower level of parallelism. Another question in this definition is how to define T_1. There are at least two choices: the execution time of the parallel algorithm on one processor, or the execution time of the best sequential algorithm for solving P. It is this latter metric that we will consider.

Given an instance of P, let us consider that the number of sequential operations to perform in its resolution is W. We will also refer to W as *the work*. In general, we could expect to have $1 \leq S_p \leq p$. The argument derives from the common sense since with p processors, we could divide the number of sequential

operations to perform into no more than p pieces of work, which leads to an *acceleration* in the resolution time of at most p. However, for several reasons, it might be possible to have $S_p > p$. One reason is that we might have more cache faults in the sequential algorithm when it processes an instance $\mathscr{I}$ whose total work is W than in the case where it processes sub-instances of $\mathscr{I}$ of work $\frac{W}{p}$.

[Efficiency] The efficiency of a parallel algorithm on a problem instance is as follows:

$$E = \frac{S_p}{p}$$

In general $E \in [0,1]$. But, as S_p could exceed p, E could be greater than 1.

In order to compute the speedup or the efficiency, we need to consider a specific instance of problem P. However, for the sake of clarity, we will consider that if two instances have the same size, they also have the same work and execution time. For instance, let us consider that P consists of multiplying two (dense) square matrices. Let us also assume two problem instances $A \times B$ and $C \times D$ where $A,B,C,D \in \mathbb{R}^{n \times n}$. The size of the first instance is the number of elements of A added to the number of elements of B, which is $2n^2$. The size of the second instance is also $2n^2$. Thus, $A \times B$ and $C \times D$ hold the same amount of work. This conclusion is confirmed in practice since the processing of both instances will require the same number of floating-point operations.

4.2.2 Asymptotic analysis of speedup and efficiency

The asymptotic analysis is a central concept in the study of parallel algorithms. Given a metric that depends on a set of parameters, its objective is to state how the metric behaves when the parameter values become infinite. The first model of asymptotic analysis that we consider focuses on speedup. Its objective is to capture the speedup behavior when the number of processors and problem size increase. This model can be used for a theoretical or experimental analysis of the parallel algorithm. The theoretical analysis is discussed in the next section by means of Amdahl's and Gustafson's laws.

In order to capture the speedup behavior through experiments, a classical tool consists in generating a 2D chart, which states the speedup reached for the various instances depending on the number of processors. An example is depicted in Figure 4.1. The curves have been obtained using the following the function:

$$T(n,p) = \left(\frac{n}{p} + p\right).4 \times 10^{-8}$$

This function is representative of the running time we could observe on the problem of finding the maximum of a vector of real numbers. Indeed, with p processors, the problem can be solved as follows. First the vector is partitioned into p pieces. Local maxima are then computed in parallel for each sub-vector. Finally, the maximum of the local maxima is returned. If we proceed this way, then the number of comparisons is $\frac{n}{p}$ for each sub-vector and p for finding the maxima among local maxima. If a comparison takes 4×10^{-8} seconds, then we have the above function.

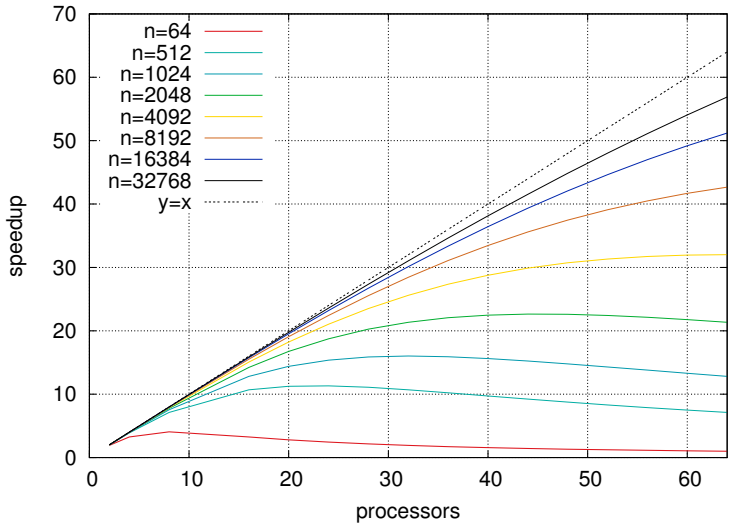


Figure 4.1: Practical example of speedup distribution in the search of the maximum element of a vector of size n.

In Figure 4.1, one can notice that the greater the size of the problem, the higher the speedup we can reach with multiple processors. This is because, when n increases, the speedup curve becomes close to the identity line $(y = x)$. We obtain a speedup close to p and hence, an efficiency close to 1. In such a situation, we conclude that the parallel algorithm is *scalable*. More generally, *we say that an algorithm is scalable if its efficiency can be kept constant when increasing the size and the number of processors*. In this example, we make a projection of the speedup in establishing that for large values of n and p, it remains close to the identity function. An interesting question is then to know whether or not it is only such a function that can be achieved in an asymptotic analysis. This is discussed in the next section.

4.2.3 Types of speedups

[Linear speedup] We say that a parallel algorithm has a linear speedup if the speedup S_p converges towards p when both the problem size and the number of processors increase.

The problem size is not included in the definition. This means that, whatever the problem instance, the parallel algorithm efficiently shares the amount of work among the processors. Linear speedups will typically be observed in parallel algorithms composed of workers that do not need to communicate. This is the case for instance of Monte-Carlo simulations.

[Super-linear speedup] We say that a parallel algorithm has a super-linear speedup if $S_p > p$ when both the problem size and the number of processors increase.

Super-linear speedups will typically be observed in parallel search algorithms based on backtracking. Indeed, assuming that the sequential depth first search space is represented as a tree, we could avoid a deep exploration of the paths that do not lead to optimal results in splitting the tree in the case where the solution is at the beginning of another path. The order of cache accesses may also play a role in super-linear speedups phenomena.

[sub-linear speedup] We say that a parallel algorithm has a sub-linear speedup if $S_p < p$ when both the problem size and the number of processors increase.

Due to some limits in the parallelization that we will discuss further, sub-linear speedups will frequently be observed.

4.2.4 Strong and weak scaling

Let us consider again the example of Section 4.2.2 (finding the largest element in a vector). We concluded that there is a convergence towards the identity function by computing the speedup for various problem sizes and processors numbers. Our conclusion was based on the distribution of the chosen points (n, p). It is important to notice that if n was only selected between 64 and 512, we would not have observed the convergence to the identity line. As we cannot evaluate all possible points, an important challenge in asymptotic analysis is to make an appropriate selection. For this purpose, two types of speedup analysis are considered in practice: weak scaling and strong scaling analysis.

In weak scaling analysis, we evaluate the speedup, efficiency or the running time of a parallel algorithm in points (n, p) where we ensure that the problem size per processor remains constant. A common practice in weak scaling analysis

consists in doubling both the size of the problem and the number of processors. If the running time or the efficiency remains constant, then the algorithm is scalable. In strong scaling analysis, we are interested in determining how far we can remain efficient given a fixed problem size. Therefore, for a fixed problem size, we increase the number or processors until we observe a change in the efficiency.

4.2.5 Isoefficiency

Given the running time function of Section 4.2.2, we computed in Figure 4.2, different values of the efficiency assuming that the problem size per processor (denoted by n_p) is 64 and 1024. As one can notice, if a linear speedup is clearly visible for $n_p = 1024$, it is not the case for $n_p = 64$. An important question in scalability analysis is then to know how to increase the problem size per number of processors. The *isoefficiency* [10] concept was introduced for this purpose.

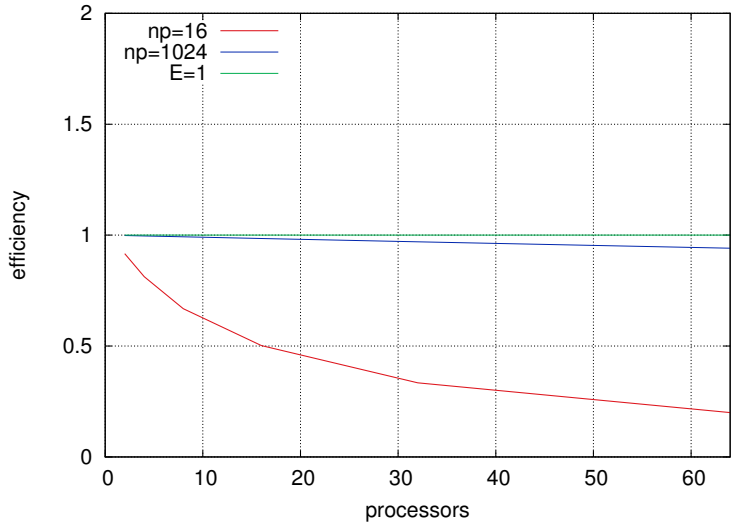


Figure 4.2: Efficiency depending on the work per processor

More generally, for a given efficiency, the isoefficiency function of a parallel algorithm shows how to increase the problem size with respect to the number of processors in order to keep a given value of the efficiency. Given a fixed efficiency value, all parallel algorithms will not have the same isoefficiency function. In general we will distinguish between algorithms for which the size must be increased as an exponential function of the number of processors (poorly scalable

algorithms) and those for which the size must be increased linearly (highly scalable algorithm).

There is no generic method for the computation of the isoefficiency function. However, in the case where an analytic formulation of the sequential and parallel algorithm execution time is available, such a function could be computed. For instance, let us assume a basic sequential algorithm whose running time is expressed as $T_1 = n \cdot t_c$ where n is the problem size and t_c a computing time. This corresponds to a simplified case where the sequential algorithm consists in performing n times a given operation. Let us also assume that the parallelization of this algorithm has a running time $T_p = \frac{T_1+T_o}{p}$. The idea in this latter formula is that given p processors, we can divide the time of the sequential algorithm by a factor of p. However, we must consider the overhead induced by communication, synchronization, and other artifacts such as the creation of parallelism. Assuming the previous expression, the efficiency becomes:

$$E = \frac{1}{1 + \frac{T_o}{n \cdot t_c}}$$

Therefore, we can relate the problem size to the number of processors and the efficiency with the following formula:

$$n = \frac{E \cdot T_o}{t_c(1-E)}$$

Thus, if we want to keep efficiency constant, we must use the equation $n = K \cdot T_o$ where $K = \frac{E}{t_c(1-E)}$. In this formula, p does not explicitly appear; however, it is implicitly considered in the overhead running time T_o that depends on the number of processors. Finally, let us observe that in practice, it might be more complex to derive the value of n to use since T_o might be a non-linear function that depends on both p and n.

4.2.6 Limits of the formalization

One of the main justification of the popularity of the concept of scalability is that it provides a theoretical background to: (1) justify the design of large parallel machines and (2) evaluate and compare parallel algorithms. In the early stage of parallel computing, scalability was mainly used to show that it is possible to build algorithms that can efficiently exploit a huge number of concurrent processors. This argument is still valid today. Indeed, we are witnessing the end of Moore's

law as it is discussed for instance in [23]; parallelism and scalable algorithms are then becoming the only option for solving computing problems faster. However, this does not mean that we have a blank check to design powerful supercomputers. It is important to notice that we are also in another era of computation where the quality of algorithms is no longer strictly based on the running time. Other dimensions such as energy consumption have gained in importance, which has led to the definition of new metrics like *energy-efficiency*. Roughly speaking, the energy efficiency of a parallel program captures the ratio between the amount of energy it uses and the time it takes. Somehow, the term *power efficiency* is more precise because this ratio corresponds to the amount of Watts used by the algorithm. We do believe that the concept of scalability must be directly extended to deal with this new notion. For instance, a scalable energy-efficient algorithm could be an algorithm that can maintain its energy-efficiency when both the problem size and the number of processors increase. In other words, the first limit of the formalization described previously is not to take into account the other qualitative dimensions of the behavior of an algorithm.

The second limit we observe is that the proposed formulation does not handle the specificity of several computing problems and algorithms. Indeed, we implicitly assume here that the size of a problem instance determines (or at least is related to) its hardness. In addition, we also assume that all instances with the same size are similarly hard. These assumptions are not true for many computing problems, in particular those which are NP-hard. For instance, on the satisfiability problem SAT, the execution time of a backtracking algorithm will depend on the distribution of exact solutions in the search space. The difference between the running times of two instances of the same size could be huge. A direct consequence of this inability to relate work and problem size is that the concept of asymptotic as described previously could no longer be applied.

Finally, we considered in our presentation that computing units correspond to processors. We also implicitly assumed that these processors have the same performance. Today however, the architecture of parallel machines has greatly evolved. Computing units could correspond to containers, virtual machines, cores or any combination of hybrid components (heterogeneity). In addition, with the complexity of machines nowadays, many other elements are related to the machine configuration could play a role in the performance of an algorithm (and its implementation). The question is then to determine whether or not the scalability results observed on a specific machine are valid on another one. In the past, similar interrogations led to the introduction of theoretical models of parallel machines like the well-known PRAM model. We encourage the reader who wish to

learn more about it to read the seminal paper [7] or the dedicated chapter in the book of Cosnard and Trystram [4].

4.3 Scalability laws

In the previous section, we showed how to use speedup and efficiency in an experimental evaluation of the scalability of a parallel algorithm. In this section, we will present how to theoretically estimate these metrics.

4.3.1 Amdahl's law

Somehow, it might be counterintuitive to consider Amdahl's law as a scalability law. In its original paper [1], Amdahl introduced the law to explain that most actual problems do not have enough parallelism that could use the full potential power of supercomputers. Amdahl's argumentation was originally based on a statistical analysis. He showed that there is little benefit in parallelizing some computing problems and particularly, those for which we only have irregular algorithms [‡]. Amdahl's analysis was right and even today, there are several computing problems on which the best parallel algorithms only achieve poor speedups. The idea to make a statistical analysis of the parallelism in term of computational problems was also ingenious. We will come back to this point and briefly introduce the P-completeness theory whose aim is to capture the problems of the P class that are hard to parallelize [11].

Although Amdahl showed that the usefulness of supercomputers might be overestimated, he proposed a simple but powerful model for the analysis of parallel algorithms. This model shows how to characterize the speedup and efficiency of a parallel algorithm as mathematical equations.

Mathematical formulation of Amdahl's law

Let us assume a sequential algorithm that solves a problem instance in W operations. The first assumption in Amdahl's law is to partition W into two frac-

[‡]Irregular algorithms are characterized by non-uniform memory pattern accesses. For such algorithms, we will *frequently* be in the situation where the data we want to access are not in the caches. Some such well-known irregular algorithms include: Cholesky factorization, finite differences algorithms, agglomerative clustering, Prim's algorithm, Kruskal's algorithm, belief propagation.

tions: namely, a sequential fraction f_{seq} and a parallelizable part f_{par} such that $f_{seq} + f_{par} = 1$. The sequential part is composed of operations that must be done one after the other and the parallelizable part corresponds to operations that can be performed simultaneously. Let us denote by t_c the execution time of a basic operation (all instructions are assumed to be identical). Then, the sequential running time of the algorithm is

$$T_1 = (f_{seq} + f_{par})W \cdot t_c$$

Given p processors, the second assumption in Amdahl's law is that we will have to distinguish between two types of computations: computations of the sequential part that will be executed on a single processor and the ones from the parallel fraction that will be shared (ideally) among all processors. This leads to the following expression:

$$T_p = f_{seq} \cdot W \cdot t_c + \frac{f_{par} \cdot W \cdot t_c}{p}$$

Consequently, the speedup is

$$S_p = \frac{1}{f_{seq} + \frac{1 - f_{seq}}{p}}$$

It is important to observe here that the assumptions underlying Amdahl's law are debatable. In particular the speedup in this model is at most linear whereas super-linear speedups can be observed in practice. This situation happens because Amdahl's law assumes a parallel algorithm issued from the parallelization of the instructions of a sequential one. But in practice, the parallel algorithm could be issued from a completely new design of the problem.

Limits to scalability

Amdahl's work pioneered several researches on the limit to scalability. In 1973, Stephen Cook introduced the P-completeness theory. This branch of the complexity theory aims at identifying problems for which there is no parallel algorithm that takes a poly-logarithmic time in the problem size, while using a polynomial number of processors. One of the objective of the P-completeness theory is to identify problems that are inherently sequential. This means that there is no efficient parallel algorithm for their resolution. In their book, Greenlaw, Hoover and Ruzzo give a compendium of P-complete problems [11] which includes several

classical problems including scheduling, minimum set cover, and linear programming.

Another important limit to scalability is the memory wall. The memory wall is due to an imbalance between the memory bandwidth, latency and the processor speed [26]. On several machines, the running time to perform a Load/Store operation in DRAM exceeds the time of a multiplication. There are several techniques that were introduced in computer machines to avoid such a wall. A possible solution is to recover data loading with computations: the processor can start another instruction if the data of a prior one are not available. With this approach, given a same parallel program, the execution order of its instructions could change from one machine to another (out-of-order execution [16]). However, even with such a solution, we can still remain constrained by the DRAM access time.

The third limit is the energy consumption. Indeed, the power consumption of a supercomputer grows with processor utilization. This consumed energy is transformed into heat that must be dissipated. Several studies showed that the cooling can account for up to 40% of the energy consumed in a datacenter[6]. To reduce this cost, the Power Usage Efficiency metric (PUE) was introduced to estimate the efficiency of datacenters. Roughly speaking, the PUE is the ratio between the total energy consumed by a datacenter and the one devoted to computations. The closer PUE to 1, the better the datacenter. In such a context, it is important to keep the parallel efficiency of an algorithm under a threshold where it does not consume too much energy in the perspective of PUE minimization.

4.3.2 Gustafson's law

The concept of scalability as it is known today owes much to the work of Gustafson [12]. Indeed, the original Amdahl's paper showed that given a fixed problem size, we will always reach a limit in its parallelization. This view is what we refer today as the strong scaling perspective. Without contradicting Amdahl's observation, Gustafson showed that this does not mean that huge parallel machines are useless. Indeed, the greater the numbers of resources, the faster the solution of large problems. Thus, he introduced the weak scaling analysis and the idea of evaluating the efficiency of the algorithm in both increasing problem sizes and number of processors. The work of Gustafson also revisited the analysis proposed by Amdahl to show how large speedups can be obtained in parallel algorithms. The general analysis he proposed is reviewed below.

Mathematical formulation of Gustafson's law

Just like with Amdahl's law, Gustafson's law is based on the concept of serial and parallelizable fraction of work (the global work is denoted by W as before). However, instead of considering these proportions in the sequential algorithm, Gustafson's analysis assumes that we know them in the parallel algorithm. Let us assume that for a parallel algorithm that runs with p processors, the serial and parallel fractions are f'_{seq} and f'_{par} respectively. The algorithm running time is

$$T_p = (f'_{seq} + f'_{par}) \cdot W \cdot t_c$$

For the equivalent sequential algorithm, the running time is

$$T_1 = (f'_{seq} + f'_{par} \cdot p) \cdot W \cdot t_c$$

This leads to a *scaled* speedup equal to

$$S_p = p + (1-p) f'_{seq}$$

In order to determine the difference between the scaled speedup and the speedup as formulated by Amdahl, let us assume that $p = 1024$ and half of the work is parallel ($f_{seq} = f'_{seq} = 0.5$). Then, while Amdahl's speedup is equal to 1.998, the scaled speedup is equal to 512.5. The difference is huge but it is easy to explain. Indeed, Amdahl's and Gustafson's analyzes are implicitly based on two different approaches in the design of parallel algorithms. In Amdahl's case, the parallel algorithm will execute (in parallel) instructions of a sequential algorithm. This envisions automatic parallelization and instruction level parallelism. In Gustafson's case, the parallelism is created depending on the number of processors. This envisions data parallelism. Further, the scaled speedup is biased by the fact that the sequential algorithm was considered as a *degenerated version* of the parallel algorithm.

4.4 Discussion about generic laws

In this section, we will extend the discussion of the Amdahl's and Gustafson's laws. The objective is to put in perspective these laws with respect to the problem types, modern resource sharing techniques and energy-efficiency.

4.4.1 Problem types in Amdahl's and Gustafson's law

As already mentioned, the type of addressed problems is a central notion in both Amdahl's and Gustafson's analysis. Indeed, the original Amdahl's paper targeted a set of problems that are hard to efficiently solve with a parallel algorithm because of irregular boundaries or non-homogeneous data distributions. In the same spirit, Gustafson introduced the notion of scaled speedup, emphasizing problems on which he obtained near-linear speedups. We do believe that the notion of problem type has received too little attention in parallel programming studies.

One of the most important theory developed for classifying the problem types in parallelization is the P-complete theory [11]. Let us recall that a problem is P-complete if: (1) it can be solved by a parallel algorithm in polynomial time, but (2) it cannot be solved in poly-logarithmic time with a polynomial number of processors, although $P = NC$. Here, *NC* is the class of problems that can be solved in poly-logarithmic time using a polynomial number of processors [19]. The fundamental question of the P-completeness theory reflects the pessimistic Amdahl's view on the parallelization (if $P \neq NC$) and the optimistic Gustafson's view (if $P = NC$). Indeed, if $P = NC$, then we can develop a highly parallelizable algorithm for all polynomial-time problems. Notice that a way to improve the algorithms is to consider the randomized version *RNC* (which aims at determining an efficient parallel solution with high probability). For instance, the problem of finding a maximal matching is in *RNC* and not in *NC*. Despite its great interest, the P-complete theory does not completely cover the class of all computational problems. In particular, there are hard problems that we can only practically address with heuristics. This includes problems of the *NP* and *PSPACE* class [8].

Another interesting view of problems type in parallelization was introduced in [2]. In their paper, the authors considered 13 key techniques or kernels to implement parallel algorithms. The techniques/kernels cover several computing domains like dense and sparse linear algebra, databases, machine learning, *etc.* Contrary to the P-completeness theory, NP-hard problems can be considered here since backtracking and branch-and-bound are part of these techniques. Despite the interest of this work, it does not however discuss one of the main aspect of Amdahl's and Gustafson's analyzes where the notion of problem types was considered within the perspective of investigating the limits we can expect from parallelization.

To conclude this "philosophical" section, we could say that when dealing with parallelization, the problem type under consideration is crucial. For instance, it is more likely to have an efficient parallel algorithm on a numerical problem than on

NP-hard combinatorial problems. Hence, we feel that, to fully complete the vision of problem types in both Amdahl's and Gustafson's works, a statistical evaluation of the most frequent parallel computing kernels implemented in parallel systems is necessary.

4.4.2 Amdahl's and Gustafson's law revisited for modern resource sharing

Amdahl's and Gustafson's discussions were about the *usefulness* of a massive parallel machine. In his original paper, Amdahl wrote: *"Demonstration is made of the continued validity of the single processor approach and of the weaknesses of the multiple processor approach in terms of application to real problems and their attendant irregularities"*. As an answer, Gustafson concluded with *"Our work to date shows that it is not an insurmountable task to extract very high efficiency from a massively-parallel ensemble"*.

The original papers of Amdahl and Gustafson share at least two common assumptions regarding the usefulness of parallel machines. The first feature is the interest in speedup optimization. Somehow, they considered that a parallel machine is useful if it can help to solve problems faster. As already mentioned, such a vision is debatable nowadays since computing has an energetic price. We will return to this point in section 4.4.3. The second feature shared by both works is to consider the usefulness of parallel machines in an algorithmic/application centered viewpoint that does not account on the margin, we could have at the operating system and middleware levels.

Today, most parallel machines are associated with a resource managing system, most often based on a client/server model. Here, each user (client side) can concurrently submit, deploy and run several parallel algorithms on a subset of processors of a parallel machine. To manage this concurrency, new concepts have emerged like the notion of job and job scheduler. A job refers to an instance of a parallel algorithm composed of features (the source algorithm to run, the input data files, the output data files, the number of processors, etc.) Each job is routed towards a job scheduler that will determine the compute nodes on which it will be executed. Depending on the parallel algorithm under consideration, a job could be parallel, moldable or malleable [5]. In the former, the requirement in term of processors for the job is fixed (typically like in MPI applications). In the moldable case, the job could run with different number of processors. In the malleable case, we additionally consider that the processor assignment of a job can change during

its execution.

With modern resource sharing techniques, it does not matter whether a job does not fully use the total number of available processors or not. In such cases, we could deploy another job that will be concurrently executed with it. It is also important to notice that important progress was done to ensure that a concurrent run will not negatively impact another one. Finally, resource sharing even goes further with virtualization [18]. With virtual machines and containers, we can artificially duplicate the physical resources of a machine that will be shared between several parallel algorithms. In addition, we can adaptively remove or add physical resources to any parallel algorithm [13].

To conclude this discussion, we argue that with modern resource sharing techniques, parallel machines became systems that are exploited to varying degrees depending mainly on the activity of the users and the interactions with the system. This does not mean that the question of the scalability of a parallel algorithm is no longer important, but that there is a complementary answer to the question of parallel machine utility. Modern resource sharing techniques have also introduced a new consideration regarding scalability. As alluded to earlier, the greater the number of requests for job processing, the more useful the parallel machine because it can be maintained *full*. However, this reasoning holds only if we are able to quickly take appropriate scheduling decisions for the submitted jobs. This means that job scheduling algorithms should also be scalable.

4.4.3 Amdahl's and Gustafson's law and energy-efficiency

When considering the question of energy-efficiency, we tend to focus excessively on the huge consumption of supercomputers while neglecting the progress made in the design of processors. At this point, it is important to recall that for several years, the design of processors followed Koomey's law which states that the number of computations per joules of energy dissipated has been doubling every year [15]. This means that for the past several years, efforts have been made to bear on improving computer hardware regarding the energy-efficiency. Unfortunately, the same is not true for computer software and algorithms. This is because energy-efficiency was not taken into account in Amdahl's and Gustafson's laws. These laws should be revisited since energy consumption could not be neglected any longer.

A naive belief is that, in order to minimize energy consumption, it is sufficient to minimize the running time. This is because the energy consumed by a parallel algorithm on a given machine can be estimated as the sum of the instantaneous

power consumed throughout the execution of the algorithm [17]. Unfortunately, this reasoning does not hold for modern large scale platforms. The instantaneous power consumption is not always a fixed quantity. It includes a variable part that depends on the algorithm run. This variable consumption will depend on the load, the frequency and voltage at which the machine is run. Therefore, we could have a faster algorithm that finally consumes more energy. However, the story does not end here. Indeed, let us observe that faster algorithms will in general be also the ones which are more compute-intensive. However, at the processor level, compute-intensive algorithms generally produce more heat. Consequently, we could even need a more sophisticated cooling mechanism to lower the temperature of a supercomputer on which we run a compute-intensive algorithm.

One of the most interesting metrics for energy-efficiency in the vision of Amdahl's law is the speedup per Watt or performance per watt ratio used in the top green 500 list (See the green500.org site for details) and well conceptualized by Woo and Lee [25]. An algorithm that scales on the speedup per Watt is able to maintain the same speedup and average watt consumption when both problem size and number of processors increase. Woo and Lee also proposed a theoretical estimation of the speedup per Watt on several types of multi-core architectures. In the proposed expressions, the speedup is defined as in Amdahl's law. The concept of speedup per Watt has some weakness, for instance if the Watts refer to a unit, it is not the case for the speedup. In addition, we could criticize the fact that it is hard to isolate the consumption of a parallel program from the one induced by the run of an operating system or middleware. Despite these weaknesses, it is certainly one of the most promising option to extend Amdahl's and Gustafson's speedups to the minimization of energy consumption.

Finally, let us observe that the isoefficiency could be used as a powerful tool to reduce energy consumption. Indeed, a parallel algorithm will not lead to the same energy consumption depending on the number of processors it uses. An interesting question is then to determine the right number. Thanks to isoefficiency, we could answer as follows: depending on the efficiency we want to maintain, we can compute for each problem size the number of processors we want to use. This observation also suggests new ideas. For instance, it might be interesting to formulate the isoefficiency while considering an average Watt consumption we wish to maintain. Such models could in particular use the important progresses made these last years on the theoretical modeling of the power consumption [17].

4.5 Designing scalable algorithms in modern large scale platforms

We propose in this section a general method for building scalable parallel algorithms. The proposed method is in particular motivated by the desire to automate the parallel resolution of NP-hard problems. However, it can be applied to a larger range of problems. It is based on three main pillars that are presented as follows.

4.5.1 Background

Pillar I: the need of new strategies for strong scaling

Fifty years ago, parallelism mainly focused on supercomputers dedicated to scientific computing, it is now available on any general purpose computer and applications. At the same time, supercomputers are always increasingly more powerful and alternative parallel systems like computational grids or clouds have emerged. This constant increase of parallel processing capabilities is challenging for the design of strong scaling algorithms. We illustrate this point on the following example.

Let us assume a machine with a huge number of processors (p_{max}). Let us also assume that we want to solve three instances I_1, I_2, I_3 of the problem P. For their resolution, we have two parallel algorithms A and A'. A has the best average execution time on the three instances while A' has the best execution time on I_1. In such a context, a rough asymptotic projection would consist in recommending A for the resolution of P [§].

Let us now assume that in the run of A on I_1, a number of processors (denoted by p_{ssl}) offers no gain in term of parallelism. We refer to this point as the *strong scaling limit* and formally define it as the smallest number of processors p_{ssl} such that:

$$\forall p > p_{ssl}, T_p \leq T_{p_{ssl}}$$

As one can notice, the strong scaling limit is not the same depending on the instance we are solving. This is clearly visible in Figure 4.1 where the limit is reached more quickly for $n = 64$ than for $n = 512$. Due to the sequential part

[§]The idea to compare algorithms based on their average running time on a set of representative computational instances is used in international competitions between algorithms. One of the most famous is the SAT competition where one goal is to solve the maximal number of SAT instances given a maximal time limit. SAT refers to the boolean satisfiability problem.

of any parallel algorithm, we could expect such a limit to be determined with $p_{max} \longrightarrow +\infty$. Returning to the asymptotic projection we made, the existence of a strong scaling limit suggests that for optimizing the efficiency in the resolution of I_1, there are $p_{max} - p_{ssl}$ processors that we should not use. As machines are increasingly powerful, we can expect in the future to have another machine whose processors are similar to the ones of the first one but with a greater number of processors p'_{max} ¶. In this latter machine, the previous asymptotic projection will still hold. However, in the resolution of I_1, we will now have $p'_{max} - p_{ssl} > p_{max} - p_{ssl}$ that are not useful. In conclusion, if the evolution of architectures leads to generations of machines with more processors, it could be inefficient to use these *additional processors* in the resolution of simpler computational instances.

To face this situation, let us assume that the strong scaling limit of A' on I_1 is $p'_l > p_{ssl}$. We could have been able to scale on $p'_l - p_{ssl}$ additional processors if it was A' instead of A that was run for solving I_1. The fact of having two algorithms with different strong scaling limits was observed in the resolution of several hard combinatorial problems, including the boolean satisfiability problem that we will present in Section 4.5.3.

As the parallelism of machines increases, we should invest in the design of cooperative executions of algorithms solving the same problem. In 1976, John Rice paved the way for a general theory of cooperative algorithms in theorizing the *algorithm selection problem* [21]. The Rice conceptualization latter inspired several studies on the automatic composition of algorithms and automatic tuning. Rice also introduced a methodology for the algorithm selection problem that we will not present here. The method we will present is inspired by the work of Huberman, Lukose and Hogg [14] on the formulation of a general theory for cooperative parallelism based on algorithm portfolios. In particular, given k parallel algorithms $A_1, \ldots A_k$, we propose to define a cooperative execution of the various algorithms as a concurrent run of each algorithm A_i on p_i processors that is stopped as soon as an algorithm finds a solution. Here, $p_i \in \{0, \ldots, p_{max}\}$ and

$$\sum_{i=1}^{k} p_i \leq p_{max}$$

We will refer to such cooperative executions as *resource sharing schedules*. Since 2010, resource sharing schedules have been successfully applied to the par-

¶This was observed on multicore machines where generations of machines integrate more cores.

allelization of the boolean satisfiability problem. In particular, several resource sharing-based solvers won the competition.

It should be noted that resource sharing schedules are not the only model of cooperative parallelism based on algorithm portfolios. Alternatives like time and malleable sharing schedules were proposed [9]. Nonetheless, they will not be discussed in this chapter. In this part, we will show how with resource sharing schedules, we can envision a new method for the design of scalable algorithms.

Pillar II: benchmark instead of problem size

In the previous weak scaling analysis, our conclusions on the general behavior of a parallel algorithm were based on observations made on a subset of instances characterized by their problem size. As already mentioned, however, the notion of problem size is not meaningful with all types of problems. On an NP-hard problem, we could have the following situations:

- the running time of a small instance exceeds the time of a larger instance
- the running times of two instances of the same size completely differ

This suggests that the idea of projecting the general behavior of a parallel algorithm based on a subset of instances, chosen mainly on their sizes, could be wrong. In addition, such a selection might not have any sense if we consider the problem resolution in a business perspective. Indeed, in this context, each problem will be associated with a context or domain that will constraint the types of practical instances. For instance, a delivery company that frequently solves the traveling salesman problem in France will not necessarily be interested in problem instances coming from Africa or the USA. When designing an algorithm for such a company, the question is not to be able to scale on any problem instance but on those representative of its business activity.

For these reasons, we do believe that in the evaluation of a parallel algorithm, we should constitute a reference benchmark of instances that might be representative of the context in which the problem will be solved. Fortunately, such benchmarks exist for several classical computational problems like the resolution of sparse linear systems or the satisfiability problem.

A main drawback while considering benchmarks is that we need another definition of the scalability. Indeed, the previous definition was based on maintaining a value of the efficiency when both the problem size and the number of processors increase. With this definition, the design of a parallel algorithm has a clear

objective, that is to target linear or super linear speedup in an asymptotic analysis. What should then be the objective if we restrict ourselves to a (limited) benchmark viewpoint?

To address this question, we propose to proceed as follows. Let us assume that U is the universe of problem instances on which we want to be efficient. Here, U could be infinite. Let us also assume that there exists a finite set B of *representative problem instances* (*i.e.,* our benchmark). Let $T(I,p)$ be the running time for solving instance I with p processors. We define the average efficiency in the resolution of B as:

$$\phi(B) = \frac{1}{|B|}\sum_{I\in B}\phi(I), \text{ where } \phi(I) = \left(\frac{T(I,1)}{p_{max}\cdot T(I,p_{max})}\right)$$

We then say that we correctly scale if

$$\zeta(B) = \frac{1}{|B|}\sum_{I\in B}|\phi(I)-\phi(B)| \longrightarrow 0$$

This definition of scalability shares a core idea with the prior one we considered in the previous sections of the chapter. It is to maintain an average efficiency over the benchmark. Indeed, we scale when the efficiency on any benchmark instance get close to the average efficiency. However, there is a major difference since the proposed definition is *machine-aware* in the sense where the speedup is always computed on the total number of available processors. Thus, this new definition somehow combines features of strong scaling (behavior of a single instance on large number of processors) with those of weak scaling (general behavior of several instances). However, there is a weak point in the proposed definition: given U, what we really want is to have

$$\frac{1}{|B|}\sum_{I\in U}|\phi(I)-\phi(B)| \longrightarrow 0$$

Thus, the choice of B is critical because it must be representative of the instances we have in U. An open question at this stage is to know how we apply this new definition of efficiency to cooperative executions. We will return to this point in Section 4.5.2.

Pillar III: the need of auto-tuning-based approaches

The increasing complexity of current machines makes auto-tuning unavoidable [22]. Any auto-tuning approach aims at solving a fundamental problem whose abstract

view is the following: we assume an algorithm that can be configured on a set of parameters θ. We also consider a performance criteria (running time, energy consumption, *etc.*) on which we want to optimize the run of the algorithm. Each parameter θ_i is associated with a definition domain $dom(\theta_i)$ that defines the values it can take. The goal is then to decide on the values to set for each θ_i (in the run of the algorithm) in the perspective of optimizing the performance criteria we considered. Nowadays, auto-tuning is unavoidable because on modern parallel architectures, there are several architectural parameters that can be configured to optimize the implementation of an algorithm. For a short view on such parameters, we refer the interested reader to works related to the optimization of dense linear algebra kernels [24].

We are convinced that the design of a parallel algorithm could no longer be restricted to the formulation of a computational process that states how to generate a correct output from a given input. The algorithm designed must be associated with a search optimization process that will state how to automatically tune the algorithm in a particular machine. The method we will propose formulate such a search process in the case of a parallel algorithm thought as the cooperative execution of several other ones.

4.5.2 Computation of cooperative executions

Given a computational problem P, we propose the following method to design an efficient parallel algorithm for its resolution.

- **Phase 1:** Collect a set of parallel algorithms $A_1, \ldots, A_k$ that solves P.
- **Phase 2:** Create a set B of reference instances in the resolution of P.
- **Phase 3:** Compute the running times $T_{A_j}(I, p)$ for solving any instance I by algorithm A_i when using p processors ($1 \leq p \leq p_{max}$).
- **Phase 4:** Determine the resource sharing schedule for which $\phi(B)$ is maximized and
$$\zeta(B) = \frac{1}{|B|} \sum_{I \in B} |\phi(I) - \phi(B)| \longrightarrow 0$$
.
- **Phase 5:** Encode the resource sharing schedule as a new parallel algorithm.

In this method, we assume that the running time of the resource sharing schedule are defined according to the equations:

$$T(I, p_{max}) = \min_{1 \leq j \leq k} T_{A_j}(I, p_i)$$
$$T(I, 1) = \min_{1 \leq j \leq k} T_{A_j}(I, 1)$$

As mentioned earlier, this is because, in resource sharing schedules, the execution is halted as soon as an algorithm finds a solution. Summarizing our method states how from a set of parallel algorithms and a benchmark, we can tune and build a cooperative executions of algorithms. We state how to optimize the cooperation of algorithms on the running time. But, the method could be extended to other performance criteria like the minimization of energy consumption. In this case, one challenge is to define aggregation rules that states how to deduce the energy consumed by a cooperative execution on p processors, from the one measured on $p' < p$ processors.

In the proposed method, we could have several choices in Phase 4. For instance, let us assume that we have 2 processors and 5 parallel algorithms. Then, there are 20 valid resource sharing schedules we could consider. This result is obtained as follows: we have 5 potential schedules where only one processor is used, 5 potential schedules where one algorithm strictly uses the two processors and 10 schedules in which two algorithms are run concurrently, each with one processor. A challenging question is to choose between all these schedules. For this choice, our conviction is that in the case where p_{max} is not too large, a brute force algorithm might be used. However, in the general case, we do believe that the optimal solution can be found from a search that uses the strong scaling limits as the frontiers of the search. We can also use the different heuristics proposed in [3].

To illustrate the different options in the choice of a resource sharing schedule, let us consider that we have the running time distribution of Figure 4.3 on a basis B of 3 representative instances. Such running times are assumed to be collected in Phase 3 of the proposed method. It is important to notice that this phase can be extremely time consuming. Indeed, given k algorithms and p_{max} processors, we have $k \times p_{max}$ running time values to compute. Let us assume that any estimation takes in average t_a seconds. Then, the expected duration for data collection is $k.p_{max}.t_a$ seconds. In addition, we must repeat the execution of the algorithms in order to have an accurate estimation. If we repeat s times, then the expected

#*processors*	1					2				
	A_1	A_2	A_3	A_4	A_5	A_1	A_2	A_3	A_4	A_5
I_1	40	60	60	80	41	37	56	51	68	38
I_2	100	130	70	150	125	89	100	65	125	100
I_3	10.2	10.6	10.5	10.3	8.4	10	10.5	10.4	10	9

Figure 4.3: Example of running time on 1 and 2 processors for 5 algorithms and 3 instances.

duration is $k.p_{max}.t_a.s$ seconds. On NP-hard problems, t_a could be high. This means that it is interesting to study how we could reduce the duration of the data collection processes. We will not discuss these aspects on this chapter.

Let us come back to the running times of Figure 4.3. On 2 processors, our method clearly shows that the resource sharing we consider will lead to different behavior. For instance, if we deploy only A_1 on two processors, then we obtain

$$\begin{aligned}\phi(B) &= \frac{40}{(37\times 2)} + \frac{70}{(89\times 2)} + \frac{8.4}{(10\times 2)} \\ &\simeq 0.45 \\ \zeta(B) &= \frac{1}{3}\left(\left|\frac{40}{(37\times 2)} - \phi(B)\right| + \left|\frac{70}{(89\times 2)} - \phi(B)\right| + \left|\frac{8.4}{(10\times 2)} - \phi(B)\right|\right) \\ &\simeq 0.059\end{aligned}$$

If now we consider the schedule that runs A_3 on one processor and A_5 on another processor, then we have: $\phi(B) = 0.49$ and $\zeta(B) = 0.0054$. As one can remark, this latter schedule is preferable to the prior one on both objectives.

For the method to work, we need to already have several algorithms solving the same problem. Fortunately, this is the case for most computational problems. We also need to be able to estimate the running times of an algorithm on problem instances. Unfortunately, such estimations are not easy to obtain on some algorithms like those based on random choices. In the next section, we will describe in detail a practical case study of this method.

4.5.3 Case study

The boolean satisfiability problem

The objective in the boolean satisfiability problem (known as SAT) is to determine whether or not, a propositional formula written in Conjunctive Normal Form (CNF) is true (satisfiable) or not (unsatisfiable) [8]. Let us recall briefly the context of this classical problem: a CNF formula is defined as a conjunction of clauses over a finite set of boolean variables. More precisely, let us consider n boolean variables $x_1, \ldots, x_n$, a literal has either a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals. For instance, $C_1 = x_2 \vee \neg x_4$ is a clause and $(x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_3)$ is a CNF formula.

SAT is a good candidate for illustrating the previous method for the following reasons:

- this problem is NP-complete and the hardness to solve an instance is not always correlated to its size; For instance, if x_u in a clause and $\neg x_u$ in another clause, it is easy to remark that the formula is unsatisfiable (whatever its size);
- it is rather easy to find several algorithms for solving SAT. Indeed, new solvers are proposed each year in the SAT competition (see www.satcompetition.org);
- there exist several benchmarks on the problem. The SAT competition regularly proposes a set of benchmark instances. The benchmarks are grouped in different classes reflecting practical scenarios and/or hardness to solve some instances;
- resource sharing schedules have already been applied successfully on SAT. Several winners of the SAT competition implemented a portfolio of solvers based on the resource sharing schedule model. However, to the best of our knowledge, such schedules were not tuned according to the method we proposed in the previous section.

Building resource sharing schedules for SAT

To illustrate our method, let us consider the data of the SAT competition available at [‖]. It is composed of a benchmark of 300 instances (corresponding to set B) and 4 parallel SAT solvers ($A_1, \ldots, A_4$). The running times of the solvers are known

[‖] http://www.cril.univ-artois.fr/ hoessen/penelope.html

for all the instances for both 8 and 32 cores. Using these data, we were interested in studying if the proposed method could be used to build a better solver on a larger number of cores.

Since we already have a set of solvers and SAT instances, the requirements of phases 1 and 2 of our method are met. For phase 3, we should have performed a benchmark evaluation. However, we choose to only use the running time estimation we already have. These data are available from the website of the Penelope solvers. The drawback of this choice is that first we do not have the estimation for all numbers of cores, and second, we cannot compute the efficiency because the estimation of the sequential run is not known. On the first point, we assumed that any of the available solver could only be run with 8 or 32 cores. Regarding the second point, we propose to minimize the cumulative running time $\sum_{I \in B} T(I, p_{max})$. Let us remark that *this goes in the direction of the maximization of* $\phi(B)$. Finally, we did not encode the parallel algorithm corresponding to the cooperative execution of our resource sharing schedule. Figure 4.4, shows the cumulative time of the different solvers. As we can notice, there is a gain in the running time when increasing the number of cores. However, this gain is far from what could be expected in a linear speedup.

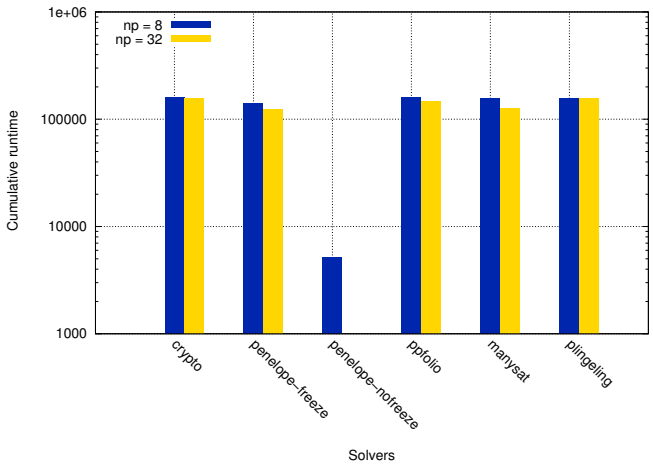
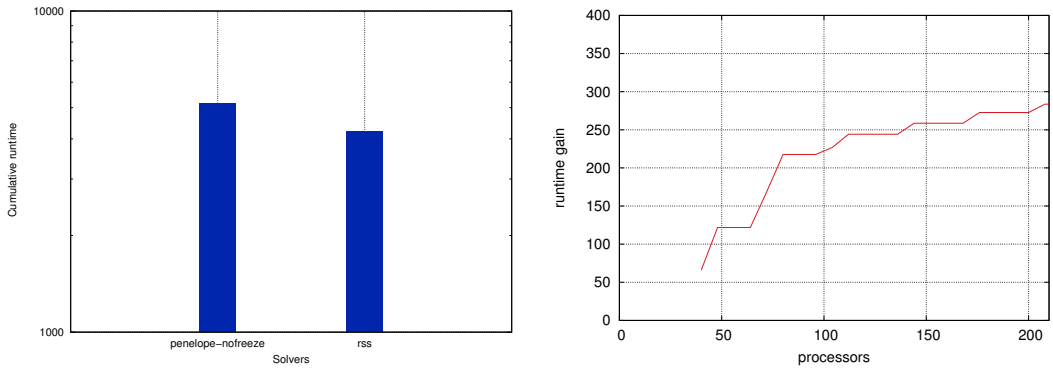


Figure 4.4: Cumulative runtime

On 8 cores, the best solver suggested by this figure is the *penelope-nofreeze* solver. As it was run only with 8 cores (the results for 32 cores are not available), an interesting question is then to know if we could obtain a better solver on 32

cores in combining the prior ones. The answer is yes, as shown in Figure 4.5(a), we were able in combining 4 different solvers to compute a better resource sharing schedule on 32 cores. The gain obtained here is 946.64 seconds. The gain here is the difference between the cumulative runtime of the best solver and the best resource sharing schedule.



(a) Resource sharing schedules on 32 cores

(b) Gain using more than 32 cores

Figure 4.5: Running times of resource sharing schedules

The second question is to know if in combining the solvers, we could obtain a better solution on more than 32 cores. The answer again is yes. In Figure 4.5(b), we depicted the absolute running time difference between the best solver we found on $p > 32$ cores and the best solver on 32 cores. The increase in running time could be observed until reaching $p = 208$ cores.

These results showed that the proposed method can be used to build an efficient parallel algorithm by composing several algorithms solving the same problem SAT. However, it is important to notice that we only provided here a theoretical validation. An effective implementation of the resource sharing schedule can in practice add a runtime overhead. However, the gain states that there is still an important margin. It is also important to notice that we did not handle the maximal time limit set in the run of the solvers and the correctness of the results. This latter point is important since some of the solvers were based on a heuristic search.

4.6 Conclusion

Amdahl's and Gustafson's law are still valid for modeling the performance of parallel algorithms. However, as discussed in this chapter, we need to extend them to other qualitative dimensions (such as energy-efficiency) and to the specificity of modern parallel platforms (virtual machines or containers, *etc.*) We also need a general formulation of scalability that could handle a larger class of problems; in particular problems for which it is not reasonable to assume that the larger the problem size, the more compute intensive the instance.

In this text, we introduced a candidate solution for the modern design of scalable algorithms. It is based on cooperative parallelism; it shows how to define a parallel run based on a computable optimization model. The model can be adjusted to optimize the run on several criteria like the efficiency, the runtime or even the energy consumption. It is also noteworthy that the proposed method focuses on strong scaling that will become a major issue, as the parallelism available in modern machines will continue increasing. Finally, our method is based on automatic tuning that is inescapable as the complexity of machines continues increasing. Improvements to our proposed method include the choice of the benchmark of instances or the reduction of the runtime required to measure the running time of the algorithms on instances.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Marin Bougeret, Pierre-François Dutot, Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Approximating the discrete resource sharing scheduling problem. *Int. J. Found. Comput. Sci.*, 22(3):639–656, 2011.

[4] Michel Cosnard and Denis Trystram. *Algorithmes et Architectures parallèles (english version by Intenat. Thomson publishing 1995)*. InterEditions, France, 1993.

[5] Pierre-Francois Dutot, Grégory Mounié, and Denis Trystram. Scheduling Parallel Tasks: Approximation Algorithms. In Joseph T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 26, pages 26–1 – 26–24. CRC Press, 2004.

[6] Richard Brown et al. Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431. Technical report, Lawrence Berkeley National Laboratory, 2008.

[7] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[9] Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Malleable resource sharing algorithms for cooperative resolution of problems. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[10] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, August 1993.

[11] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. Oxford University Press, Inc., New York, NY, USA, 1995.

[12] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[14] Bernardo. A. Huberman, Rajan. M. Lukose, and Tad. Hogg. An economic approach to hard computational problems. *Science*, 27:51–53, 1997.

[15] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Ann. Hist. Comput.*, 33(3):46–54, July 2011.

[16] Bich C. Le. An out-of-order execution technique for runtime binary translators. *SIGPLAN Not.*, 33(11):151–158, October 1998.

[17] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. *SIGMETRICS Perform. Eval. Rev.*, 31(1):160–171, June 2003.

[18] Susanta Nanda and Tzi-cker Chiueh. A survey of virtualization technologies. Technical report, SUNY at Stony Brook, 2005.

[19] Nicholas Pippenger. On simultaneous resource bounds. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS '79, pages 307–311, Washington, DC, USA, 1979. IEEE Computer Society.

[20] S. K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu. *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version I*. Online: http://www.cs.gsu.edu/ tcpp/curriculum/, 55 pages, USA, 2012.

[21] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[22] Walter Tichy. Auto-tuning parallel software: An interview with thomas fahringer: the multicore transformation (ubiquity symposium). *Ubiquity*, 2014(June):5:1–5:9, June 2014.

[23] Moshe Y. Vardi. Moore's law and the sand-heap paradox. *Commun. ACM*, 57(5):5–5, May 2014.

[24] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[25] Dong Hyuk Woo and Hsien-Hsin S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, December 2008.

[26] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

4.7 Exercises

4.7.1 Exercises for Section 4.3.1

1. In Amdahl's law, assuming that $f_{seq} = 0.4$, what is the maximal number of processors to use to achieve an efficiency of at least 0.38?

2. On a machine with 8 cores, what is the maximal speedup in the multicore parallelization of 90% of a program?

3. Let us consider the computer program

```
for (i = 0; i < 100; i++){
   a[i] = b[i] + c[i];
   d[i] = a[i] + d[i-1]/2
}
```

   where a,b,c and d are arrays of integers of size 100.

   (a) In considering only the inner loop instructions, what is the fraction of additions of the program (`b[0] + c[0]` is an addition)?

   (b) What is the maximal speedup we can expect from the parallelization of the algorithm?

4. With the MapReduce paradigm, we can count the number of words in a document by the means of a process that includes four steps: splitting, map, shuffle, reduce.

   At the beginning, given a document of n lines, the master node splits it into n sub-documents, each corresponding to a line. It then assigns these sub-documents to different workers. The map step follows where each worker runs a map function that consists of sending the pairs (”key”, 1) where ”key” is a word found in the sub-document it processes. After the completion of the map step, the master node groups the emitted pairs by keys and sends all the data of a given key to a distinct worker. The process ends with the reduce step where each worker adds up the number of keys it has and returns the cumulative value.

   In Figure 4.6, a graph illustration of this process is provided with an input document of 3 lines. The objective is to count the number of occurrences of each of the words. This number is obtained after the reduce step.

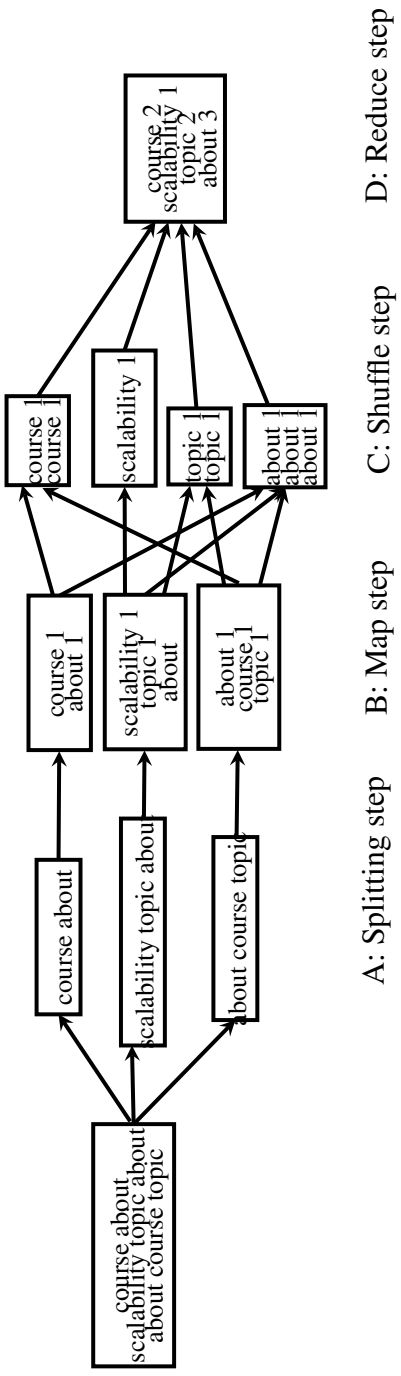


Figure 4.6: MapReduce example

In a MapReduce process, a step of the MapReduce process is only started if the prior one is completed. Let us assume that in the map step, we have p workers, each deployed on a distinct processor. Let us also assume that the map function given a line l_i of length $|l_i|$ will run in $\Theta(|l_i|)$. We consider for the sake of simplicity that all computations are done using a shared memory.

(a) In the worst case, what is the completion time of the map step assuming that each worker will get $\frac{n}{p}$ lines?

(b) How many workers should we use in this phase to achieve a speedup of c (in the step)?

(c) Assuming that in each line l_i, we have the same probability to have $1, 2$ or $|l_i|$ distinct words, how many workers should we use in the reduce step to maximize the efficiency of this step?

(d) Assuming that the time of the splitting and shuffle steps are known, given p workers in the map phase and q workers in the reduce step, propose a theoretical estimation of the execution time.

5. The well-known Fibonacci numbers are defined by the recurrence

$$F(0) = 0\ ,\ F(1) = 1\ ,\ F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

Let us consider a multi-threaded program that proceeds as follows: given a value $n > 1$, it creates two threads that respectively computes $F(n-1)$ and $F(n-2)$. It then adds the value produced by the thread and returns it.

(a) How many threads are created in the computation of $F(n)$?

(b) Propose an asymptotic analysis of the scalability of this program.

(c) What are the limits to the scalability of this program?

(d) Propose a better parallelization for the computation of the Fibonacci numbers.

6. Given a square matrix of size n and a vector of n elements, let consider an algorithm for matrix-vector multiplication whose cost on p processors is given by the following equations:

$$T_1 = n^2 t_c$$
$$T_p = t_c \left(\frac{n^2}{p} \right) + t_s \log p + t_w n$$

where, t_c, t_s and t_w are constants.

Determine the isoefficiency function of this algorithm.

4.7.2 Exercises for Section 4.3.2

1. Let us consider the product $C = A \times B$ where $A, B, C \in \mathbf{R}^{m \times m}$. The computation of C is done by using a block matrix multiplication algorithm that splits A, B, C into square blocks of size q. Thus, we have:

$$C = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix} \text{ where } nq = m \text{ and } C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

   Let us consider a multi-threaded implementation on a p cores machine. At the beginning of the execution, a master thread creates n^2 tasks and put them in a stack S. Here, each task corresponds to the computation of a block C_{ij}. p other threads are next created; their processing consists of iteratively removing a task from S that they then process. All threads are stopped as soon as S becomes empty.

   (a) Propose a parallel implementation of this algorithm.

   (b) Develop an asymptotic analysis of the scaled speedup of the proposed implementation in the case where $q = n$

   (c) Assuming 4 threads, deduce the value of q that optimizes the scaled speedup (Use experimental results of the proposed implementation for this).

2. In the prior algorithm, let us now consider that the sum $\sum_{k=1}^{n} A_{ik} \cdot B_{kj}$ is parallelized. This means that when a thread steals a task corresponding to the computation of C_{ij}, it next creates d other sub-threads such that the sub-thread l will compute

$$C_{ij} = C_{ij} + \sum_{k=(l-1)(\frac{n}{d})+1}^{l(\frac{n}{d})} A_{ik} \cdot B_{kj}$$

   Propose an asymptotic analysis for ($q = n$, $d = 2$) and ($q = n$, $d = \sqrt{n}$).

3. Let us consider a vector of real numbers $\bar{x} = (x_1, \ldots x_n)$ on which we want to compute the standard deviation and mean. Here,

$$\sigma(\bar{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}, \text{ and } \mu(\bar{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

For the parallelization of this computation we consider a two-phase algorithm. The first phase computes the mean in parallel. Assuming that we have p cores, one subdivides the list into p near-equal partitions. Each thread then adds up the number in its partition and a final thread adds up the sub-sums of the other threads. In the second phase, one proceeds in the same way as for the standard deviation. The threads compute the squared differences of the numbers in their partitions and a last thread computes the standard deviation.

   (a) What is the computational complexity of this algorithm?

   (b) What is its theoretical speedup?

   (c) Could we do better while parallelizing the sums?

4. Let us consider a web server associated with a queue of incoming requests. The server processes each request in Δ seconds.

   (a) Which minimal number of requests per second ensures that the size of the queue will be always greater than 1?

   (b) In order to reduce the processing time, one decides to create p instances of the web server. All the instances are associated with the same queue. What is the number of requests that will be processed per second?

   (c) At which date a request that enters in the queue at date t_0, with d predecessors in the queue, will be processed? What is the efficiency of the processing?

4.7.3 Exercises for Section 4.5

1. Let us assume a benchmark of instances B and the running time $T(I, p)$ for $I \in B$ and $1 \leq p \leq p_{max}$

(a) Write a brute force algorithm that computes the resource sharing for which $\phi(B)$ is maximized and

$$\frac{1}{|B|}\sum_{I\in B}|\phi(I)-\phi(B)| \longrightarrow 0$$

(b) What is the computational complexity of your algorithm?

(c) Discuss the desirability of computing such resource sharing schedules on matrix multiplication algorithms (consider the case of dense and sparse matrices).

2. Let us reconsider the case study of the satisfiability problem. Assuming that $T(I,1) = T(I,8)/8$, apply the algorithm of the previous exercise to determine the best resource sharing schedule.

3. In this case study, compute the best resource sharing schedule on 16 processors and propose a parallel implementation of this algorithm.