PARALLÉLISATION MASSIVE DES ALGORITHMES DE BRANCHEMENT

par

Andres Pastrana Cruz

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 10 septembre 2021

Le 10 septembre 2021

Le jury a accepté le mémoire de Andres Pastrana Cruz dans sa version finale

Membres du jury

Professeur Manuel Lafond
Directeur
Département d'informatique

Professeur Jean-Pierre Dussault
Membre interne
Département d'informatique

Professeur Gabriel Girald
Président-rapporteur
Département d'informatique

Sommaire

Les problèmes d'optimisation et de recherche sont souvent NP-complets et des techniques de force brute doivent généralement être mises en œuvre pour trouver des solutions exactes. Des problèmes tels que le regroupement de gènes en bio-informatique ou la recherche de routes optimales dans les réseaux de distribution peuvent être résolus en temps exponentiel à l'aide de stratégies de branchement récursif. Néanmoins, ces algorithmes deviennent peu pratiques au-delà de certaines tailles d'instances en raison du grand nombre de scénarios à explorer, pour lesquels des techniques de parallélisation sont nécessaires pour améliorer les performances.

Dans des travaux antérieurs, des techniques centralisées et décentralisées ont été mises en œuvre afin d'augmenter le parallélisme des algorithmes de branchement tout en essayant de réduire les coûts de communication, qui jouent un rôle important dans les implémentations massivement parallèles en raison des messages passant entre les processus.

Ainsi, notre travail consiste à développer une bibliothèque entièrement générique en C++, nommée *GemPBA*, pour accélérer presque tous les algorithmes de branchement avec une parallélisation massive, ainsi que le développement d'un outil novateur et simpliste d'équilibrage de charge dynamique pour réduire le nombre de messages transmis en envoyant les tâches prioritaires en premier. Notre approche utilise une stratégie hybride centralisée-décentralisée, qui fait appel à un processus central chargé d'attribuer les rôles des travailleurs par des messages de quelques bits, telles que les tâches n'ont pas besoin de passer par un processeur central.

De plus, un processeur en fonctionnement génère de nouvelles tâches si et seulement s'il y a des processeurs disponibles pour les recevoir, garantissant ainsi leur transfert, ce qui réduit considérablement les coûts de communication.

Sommaire

Nous avons réalisé nos expériences sur le problème de la couverture minimale de sommets, qui a montré des résultats remarquables, étant capable de résoudre même les graphes DIMACS les plus difficiles avec un simple algorithme MVC.

Mots-clés: Équilibrage de charge ; couverture de sommet ; algorithmes parallèles ; parallélisme évolutif ; algorithmes de branchement

Abstract

Optimization and search problems are often NP-complete, and brute-force techniques must typically be implemented to find exact solutions. Problems such as clustering genes in bioinformatics or finding optimal routes in delivery networks can be solved in exponential-time using recursive branching strategies. Nevertheless, these algorithms become impractical above certain instance sizes due to the large number of scenarios that need to be explored, for which parallelization techniques are necessary to improve the performance.

In previous works, centralized and decentralized techniques have been implemented aiming to scale up parallelism on branching algorithms whilst attempting to reduce communication overhead, which plays a significant role in massively parallel implementations due to the messages passing across processes.

Thus, our work consists of the development of a fully generic library in C++, named *GemPBA*, to speed up almost any branching algorithms with massive parallelization, along with the development of a novel and simplistic *Dynamic Load Balancing* tool to reduce the number of passed messages by sending high priority tasks first. Our approach uses a hybrid centralized-decentralized strategy, which makes use of a center process in charge of assigning worker roles by messages of a few bits of size, such that tasks do not need to pass through a center processor.

Also, a working processor will spawn new tasks if and only if there are available processors to receive them, thus, guaranteeing its transfer, and thereby the communication overhead is notably decreased.

We performed our experiments on the *Minimum Vertex Cover* problem, which showed remarkable results, being capable of solving even the toughest DIMACS graphs with a simple MVC algorithm.

Abstract

Keywords: Load balancing ; vertex cover ; parallel algorithms ; scalable parallelism ; branching algorithms

Remerciements

Je tiens à remercier ma mère Yadira d'avoir toujours cru en moi et d'être si positive.

Je tiens à remercier mon épouse, Angela, qui m'a soutenu dans toutes mes épreuves, mes absences, mes crises de dépit et d'impatience. Elle m'a apporté son soutien et son aide, a discuté d'idées et a évité plusieurs faux pas.

Je tiens à remercier mon superviseur, le Dr. Manuel Lafond, pour avoir fourni des conseils et des commentaires tout au long de ce projet.

Je tiens à remercier Calcul Canada, pour m'avoir permis d'accéder à leurs serveurs pour effectuer des tests exhaustifs de mon projet.

Je tiens à remercier les mauvaises circonstances qui ont fait de moi quelqu'un de plus exigeant et discipliné.

Abbreviations

BnB	**B**ranch a**n**d **Bound**
FPT	**F**ixed-**P**arameter **T**ractable
MVC	**M**inimum **V**ertex **C**over
DLB	**D**ynamic **L**oad **B**alancing
IPC	**I**nter-**P**rocess **C**ommunication

Contents

List of Figures

List of Tables

Introduction

In this chapter, we present the subject of research and part of the motivation of this study.

Context

A great variety of problems in Computer Science are approachable by the means of algorithms that are aimed to find a particular solution. There exist multiple approaches for the same problem, which result in a vast miscellany of algorithms. Most algorithms start from their naive version, which is the most intuitive and first attempt to solve a specific problem. However, most of them are improved over time due to scientific interest. This improvement is typically performed because these algorithms obtain solutions in exponential time, which in practice, the age of the observable universe would not be enough to attain a solution depending on the input. The class of NP-complete problems is well-known for the fact that there is no known algorithm capable of finding a solution in polynomial time [1].

Computational capacity per processor has increased significantly since a few decades ago, yet scientific needs outrun this performance improvement by far. In order to tackle hardware limitations, supercomputers allow to massively cooperate to perform parallel computations, which is to be seized as the solution to accelerate branching algorithms to handle large instances.

Exponential branching algorithms

The idea behind a recursive branching algorithm is to split a given instance into sub-instances that are explored recursively, such that one of these sub-instances leads to an optimal solution. In most scenarios, a solution is found by trial-and-error but following some logic steps, which is essentially trying all possible combinations to achieve an acceptable or exact solution. Since the number of combinations may lead to an enormous number and therefore no solutions; there exist some heuristic algorithms that attain solutions in a reasonable time-frame, yet exact algorithms keep being the main interest for the scientific community [27].

These strategies are used in several areas such as in bioinformatics for the cluster editing problem [54] or the vertex cover problem[57]. Thus, various techniques are constantly applied for these exact algorithms to narrow down the complexity as much as possible such that *Branch-and-Bound* (BnB) and *Fixed-Parameter Tractability* (FPT).

BnB algorithms consist of recursively apply branching steps to sub-instances $I_1, I_2, \ldots, I_l$ of I, until they become simple tasks. Thus, this algorithm creates a *search tree* where each recursive call corresponds to a node. The exploration of a path is halted when in the recursive path, the series of decisions detect that this branch will lead to a non-optimal solution [18]. This approach allows the program to be aware of the best solution so far and therefore ceasing the search as soon as a current solution is equal or worse than the existing one.

On the other hand, the FPT approach wants to answer the question if there exists a solution of size at most k, where k is a parameter that will be typically reduced as the exploration progresses. If a solution of size k is found, unlike the BnB, the FPT approach halts the whole search and returns a YES, otherwise, the algorithm returns NO [18]. The previous techniques can be combined to attain even better performance with adequate implementations.

Although the scientific community has accomplished great advances, even the most efficient algorithm is still exponential. For instance, the classic Minimum Vertex Cover problem in its most naive approach is of complexity $O^*(2^n)$, where the O^*-notation suppresses all polynomial factors and n is the size of the input, for which its most efficient exact algorithm so far is of complexity $O(kn + 1.2738^k)$, where k is an upper bound on the size of the target solution [16]. Such branching algorithms are nowadays challenging, which may lead the scientists to launch calculations during days, months, and even more than that depending on the nature and urgency of the problem. See [13] for more applications of FPT algorithms in bioinformatics.

Thus, consider the Minimum Vertex Cover (MVC) problem, where given an undirected graph $G = (V, E)$, we must find a subset $V' \subseteq V$ of minimum size that touches every edge, i.e. such that for all $uv \in E$, $u \in V'$ or $v \in V'$. A pseudo-code of an algorithm is presented in Algorithm 1. Under the ideal circumstances, we could say that the *binary search tree* of this algorithm is well balanced, that is, the number of leaves on the left-hand side is equal to the number of leaves on the right-hand side, as illustrated in Fig. 1. This scenario may lead us to jump to the conclusion that speeding up the solution by parallelism may be trivial.

```
function mvc($G = (V, E)$, $S$)
    if $|V| = 0$ then
        if $|S| < |best|$ then
            $best = S$
        return
    Let $uv$ be an arbitrarily chosen edge, with $uv \in E(G)$
    mvc($G - u$, $S \cup \{u\}$)
    mvc($G - v$, $S \cup \{v\}$)
    return
```

Algorithm 1: Minimum Vertex Cover pseudo-algorithm.

Nevertheless, branching algorithms are typically unbalanced and even though they may have a specific branching factor of c, all recursions might not be required to be traversed. For instance, the solution of Algorithm 1 may be accelerated by changing the search strategy, where the highest degree vertex u is arbitrarily chosen for the left-hand side and the neighbors of this vertex, $N(u)$, chosen for the right-hand side. This may cause to reach a leaf on the right branches faster than going into the left ones. Then, each branch can be seen as a task that can be delegated to another pro-

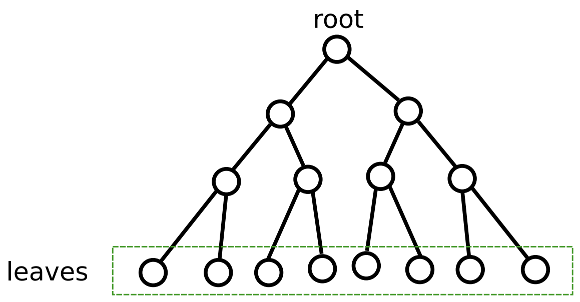

Figure 1 – Well-balanced binary tree.

cessor, which makes it a non-trivial work since the search tree nodes that are worth exploring are determined dynamically and thus are not known in advance. Therefore, this requires a dynamic task distribution strategy, and performing this optimally is still a debated topic.

Hence, our center of attention in this study is to tackle the distinct approaches to improve the performance of these algorithms by the means of parallelization techniques.

Our contributions

In this thesis, we present a novel *semi-centralized* inter-process communication strategy along with a novel *Dynamic Load Balancing* strategy, which combined show competitive performance. The semi-centralized approach minimizes the inter-process message passing such that a process is designated as the center in a lightweight fashion where its responsibilities are reduced. The center is essentially responsible for remembering the workers' state and assigning workers to assist other workers with light messages of only a few bits. This worker assignment guarantees that a message is delivered to its destination, and thereby avoiding failed requests. Messages are sent asynchronously to improve performance and restrain from blocking calculations by the processors. Thus, heavy tasks are sent directly among the workers without the need for job queues and without tasks passing through the center, which differs from centralized strategies where the whole communication is controlled by the center resulting in a bottleneck that restricts scalability. The dynamic load balancing approach allows each processor to keep track of its highest priority search tasks or unvisited nodes in the search tree, which reduces considerably the number of generated tasks and therefore the communication overhead. Furthermore, our approach uses a *process-thread* hybrid implementation, which reduces the number of spawned processes and allows each process to manage its own threads and memory domain.

We performed experiments on the *Minimum Vertex Cover* problem implementing a basic MVC algorithm of complexity $O(2^n n)$. Our results not only demonstrated to be competitive against the decentralized technique presented by Abu-Khzam et al., [2], which was the best approach known so far, but also achieved super linear speedups on certain scenarios.

This thesis is organized as follows. In Chapter 1, we present the preliminary notions to provide the reader with a sufficient understanding of this research. In Chapter 2, we present the state of the art in massive parallelization for branching algorithms and load balancing strategies. In Chapter 3, a paper submitted to the *Journal of Parallel and Distributed Computing* is presented with the most relevant information of this study. In Chapter 4, we present the most significant challenges that were not included in the paper, followed by the Conclusion.

Chapter 1

Preliminary notions

In this chapter we address some concepts that will be relevant along this thesis.

1.1 Branch & bound algorithms

Branch and Bound algorithms commonly abbreviated as **BnB**, are fundamentally used to produce exact solutions. Their goal is to find a representative value that maximizes or minimizes an optimization criterion by the means of creating a search space that explores all possible combinations and then choosing the most appropriate solution [36].

As its name suggests, it consist of branching in a search space according to some decisions and then bounding as a prospective solution is found. Putting this into perspective, a real world analogy follows.

Analogy 1.1.1. An individual is trying to find the shortest path through a labyrinth. In order to achieve this task, the most naive way is to explore all pathways. This individual will have to decide which pathway to take every time it reaches a fork and then measures its distance. Once it has found the first pathway, it will keep exploring the other pathways that he left behind at every fork.

Assuming that the individual encounters the same number of forks on every pathway, this leaves it with at most c^n paths to explore, where c is the forking factor and n is the number of forks per pathway which is directly related to its length. For instance, $c = 2$ and $n = 5$ would leave the walker with at most 32 pathways to tour, since some other pathways might be shorter but it is unknown yet.

It is impractical to tour all possible paths to find the shortest pathway. Thus, bounding comes with a promising solution, which is allowing the individual to remember the shortest path found so far. Therefore, if after exploring the first pathway, it found its distance to be 3, when exploring other pathway and reaching a distance of 3 but not reaching the end, it is safe to say that it is not worth it to keep walking in this direction, then it should return to explore the other path at the preceding fork.

Considering the aforementioned bounding analogy, this does not guarantee that our character will explore fewer pathways because if all pathways are of the same length, then it will always reach the end of each pathway. However, if the individual is lucky and finds the shortest path to be 2 on its first attempt, then it will not explore more than 2 length units on the other forks, which will decrease from 2^5 explored possible pathways to only 2^2 and still finding the shortest path.

As seen in the Analogy 1.1.1, branching happens every time a decision must be taken whether it splits in two or more ways, and bounding potentially decreases the task complexity. Also, each time a fork is reached, all solutions from now on depend on the path traveled up to that moment, which is essentially the definition of recursion.

Note that, whether all branching algorithms are recursive, not all recursive algorithms are of branching type.

Additionally, the previous analogy was aimed to find the shortest path, but also its purpose could easily be adapted to find the longest path, which makes it suitable for virtually any computable problem. Note that BnB algorithms apply more naturally to minimization problems, because in maximization problems, the best solution found so far does not allow us to cut branches prematurely.

As an additional example, consider the cluster editing problem as per the Algorithm 2, which is a common experimental benchmark [11, 24], where given a graph $G = (V, E)$, the goal is to insert or remove a minimum number of edges so that each connected component G is a clique. To achieve this, all connected components of G are cliques if and only if G is P_3-free, i.e. there is no path on 3 vertices without a shortcut. Thus, the algorithm should find some vertices u, v, w that form a P_3, and branch into the three possibilities to disassemble it, which are: splitting $u - v$, splitting $v - w$ and joining $u - w$. In this case, the branching factor is $c = 3$, which should typically give a complexity of $O(3^n n)$, where n is the number of vertices in G. However, to ensure that the algorithm converges to a solution, some rules must be applied such that if two vertices have been joined, they get tagged as *forced*. If two vertices have been split, they get tagged as *forbidden*. For a *FPT* application of the cluster editing problem see [24].

```
function clusterEditing($G = (V, E), forced, forbidden$)
    if each connected component is a clique then
        $nbEdges = forced.size() + forbidden.size()$
        if $nbEdges < best$ then
            $best = nbEdges$
    Let $u, v, w$ be a $P_3$ of $G$, with $uv, vw \in E(G)$
    if $uv$ is not in $forced$ then
        clusterEditing($G - uv$, $forced$, $forbidden \cup \{uv\}$)
    if $vw$ is not $forced$ then
        clusterEditing($G - vw$, $forced$, $forbidden \cup \{vw\}$)
    if $uw$ is not in $forbidden$ then
        clusterEditing($G + uw$, $forced \cup \{uw\}$, $forbidden$)
```

Algorithm 2: The cluster editing problem.

As mentioned above, these rules allow the algorithm to split vertices that have been previously joined or vice versa, thus, it is up to the algorithm to decide if a branch must be explored or not. Therefore, this algorithm is likely to be unbalanced

1.2 Fixed-parameter tractable algorithms

Algorithms with running time $O(f(k) \cdot n^c)$, for a constant c independent of both n and k, are called *fixed-parameter algorithms*, or FPT algorithms. Typically the goal in parameterized algorithmics is to design FPT algorithms, trying to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be put in contrast with less efficient XP algorithms (*for slice-wise polynomial)*, where the running time is of the form $f(k) \cdot n^{g(k)}$, for some functions f, g. There is a tremendous difference in the running times $f(k) \cdot n^{g(k)}$ and $f(k) \cdot n^c$. In parameterized algorithmics, k is simply a *relevant secondary measurement* that encapsulates some aspect of the input instance, be it the size of the solution sought after, or a number describing how "structured" the input instance is [18].

The classical optimization problem in computer science, the Minimum Vertex Cover (MVC) problem is a good example of it, which consists of finding a set of vertices that touches at least one endpoint of every edge in a graph, as defined in Section 3.1.

Thus, a brute force and naive algorithm to solve this problem would be choosing an edge uv arbitrarily and include one of its ends, either u or v, leaving a new subgraph G' such that $G' = G \backslash u \vee G' = G \backslash v$, as presented in Algorithm 1. This creates an exponential time algorithm of complexity $O^*(2^{|V|})$, where 2 is the branching factor because the decision is based on the number of vertices per edge and $|V|$ is the number of vertices.

One could easily see how finding a solution to this problem becomes rapidly unpractical, since, after a certain graph size, the maximum theoretical number of possible solutions would take more time to be solved than the age of the observable universe even using the most powerful computer in the world.

To exemplify, given a graph $G = (V, E)$ with $|V| = 1000$ vertices, this brute force algorithm has a complexity $O(2^{1000})$, which would yield to $2^{1000} \approx 1.07 \cdot 10^{301}$ possibilities. For the FPT version, if the MVC is of size k, the algorithm will receive this parameter and at each time a decision is made, the value of k will be reduced to $k-1$, representing a solution that has been achieved. Thus, the algorithm will return false when $k = -1$ and true when $k \geq 0 \wedge G = \emptyset$, which allows us to constrain the algorithm to $O(2^k n)$ since the solution is at most k, where $0 < k \leq n$.

Depending on the graph topology, this k could lead to a super-fast solution in the best scenarios or just no improvement at all in the worst scenario. If for the previous MVC example, if k = 10, then $2^{10}n$ is reasonable, but if $k = 500$, this would still require at most $\approx 2^{500}$ computations.

One important detail of a FPT algorithm is that these algorithms search for a YES or NO result since its purpose is only to ascertain if there exists a solution of size k. Once a YES condition is met, the whole search can be halted, and depending on the user's interest, a new search can be launched with different k. This also means that, if we are lucky enough, the solution may be found within the first few seconds, also depending on the search strategy.

In addition, an FPT agorithm of a problem can be easily modified to return the the cover by the means of a partially growing cover as stated in Algorithm 1.

1.3 Parallelization

A task is typically completed by a series of instructions, whether it is digging a hole in the ground, where each instruction would be shoveling, or solving a graph computationally where each instruction may be a primitive operation like multiplication, summation or division.

Most of the time, the simplest instructions are independent of the others, meaning that their individual outcomes do not affect the individual outcomes of other instructions, though it does affect the final combined outcome.

Allow us the following analogy.

Analogy 1.3.1. Considering the aforementioned digging-a-hole example, this task could be accelerated by increasing the number of workers, leading to a faster solution. However, some scenarios must be considered as follows:

— There might be a limited number of shovels, therefore, it might be unpractical to hire more workers than the actual number of available tools.
— There is a limited space, therefore, too many workers will not necessarily accelerate the job.

Knowing that the simplest instruction is a shovel load, it is feasible to split the whole task into as many possible shovel loads and a worker per instruction. However, let's say that a worker needs at least one square meter to work efficiently without bothering his colleague. It will take more effort to assign a single instruction for a single worker than allowing a few workers to resolve multiple instructions each.

In the previous analogy, shovels might be interpreted as physical cores on a computing device, and workers as the instance of a computer program which could be a process or a thread. For additional details and extension of this subject, we refer the reader to [53].

1.3.1 Process

A process is a program in execution which should be distinguished of only the word *program*. A *program* is a *passive entity* that contains a series of instructions on disk, commonly know as the executable file, whereas a process is this *active entity*. A process typically contains the **stack**, **data section**, **heap** and **text section**. The **stack** contains temporary data as function parameters, return address and local variables. The **Data section** contains global variable, the **heap** is memory that is dynamically allocated during the process run time, and the **text section** contains other than lines of code [47].

A program becomes a process when it is loaded to the memory and it is worth mentioning that multiple processes may be associated with the same program, however, they will be independent, unless special communication techniques are applied, which will be discussed later. A process is typically independent, has its own ID, and its memory is private to other running processes. By default, a process spawns at least one thread, which will be discussed in the next section [47].

1.3.2 Thread

A thread is a basic unit of CPU utilization; similar to a process, it has its own ID. A thread has privileged access to all the information contained within a process. Threads are instances that perform the instructions, resolve tasks and they share all information with other threads, [47].

Allow us to illustrate with an analogy.

Analogy 1.3.2. A thread could be seen as a worker of a company, and a process as this company. As in the real world, a company cannot run without at least one employee but it can perfectly operate having just a single one. Companies are usually independent of others and their information is typically private, just like a process. Analogously, a company may have multiple employees to perform more than one task at a time and all employees are able to communicate easily with each other and access the company data (assuming that there are no position hierarchies).

1.3.3 Multiprocessing

As its name suggests, multiprocessing is the technique of using multiple processes to perform operations. These operations may have a wide degree of independence, meaning that an operation of a process may participate somehow in the operation of another process, or they might be simply fully independent [39].

There are several factors that could lead a user to implement multiprocessing, in which one of them is taking advantage of the independence, ie. the tabs of an internet browser could be managed by different processes, in which a given failure scenario, will affect only the handling process. Then, as long as the inter-process communication is properly implemented, a failure of a process will not affect another running process, which would allow to restart of the failing process and continue its job where it was left [39].

Since processes manage their own resources and are independent, this makes them appropriate to massive parallelization because as mentioned in Section 1.3.1, they can be attached to the same program, meaning that communication can be established as long as there is a communication band in between.

Hence, having a supercomputer cluster, it is possible to spawn processes across all computing nodes, allowing to initiate communication regardless of the process location and therefore utilizing all the processors of the cluster [39].

1.3.4 Multithreading

Analogously to multiprocessing, this is the technique to perform operations using multiple threads. A thread may be running independent tasks but it will share resources with its parent process, meaning that it relies on its parent limitations. However, multithreading is computationally cheaper than multiprocessing [40].

Conversely, parallelism implementation is typically more efficient with multithreading because it maximizes the utilization of the processor execution units [40], and depending on the problem, it may be easier to apply because its implementation is usually supported by the programming language already.

Let us introduced the following analogy.

Analogy 1.3.3. A construction company is hired to build a bridge, the contractor provides it with a fixed number of tools that are meant to be utilized continuously to attain a deadline. Here, the hired company already has a limitation which is the work-site and their tools, just as a process would have with memory and number of processors. The company is free to hire as many workers as it considers appropriate, though no worker is able to do anything without a tool. If there are more workers than the number of tools, then they will be forced to share tools. On the other hand, if there are fewer workers than the number of tools, it would result in a delay in the construction.

Thus, as per Analogy 1.3.3, if a processor is meant to be used continuously, a proper number of threads, matching the number of cores, should be spawned to optimize CPU utilization.

1.4 Multiprocessing + Multithreading (hybrid)

Multithreading and Multiprocessing have both their corresponding advantages and disadvantages. For a super intense and cooperative task that needs to be solved in parallel, it would make more sense that multithreading would deliver more performance. Although, nowadays technology has hardware limitations like the number of processors per computing node and memory size [41].

Hence, it is mandatory to implement multiprocessing such that massive parallelization can be implemented. Though, to decrease communication overhead due to interprocess messaging, the number of processes should be minimized, which could be one process per computing node. Thus, as stated in Section 1.3.3, multiprocessing offers the benefit of establishing communication between several nodes whereas multithreading offers the benefit of sharing resources for direct communication between processors working on a problem.

The hybrid implementation allows to send the heaviest tasks to processes located on different computing nodes, then these heavy tasks can be processed by the local threads within the process domain, and thereby decreasing communication overhead and achieving better performances.

1.5 Critical section

When it comes to accessing shared data, there might exist the case where two threads attempt to modify a particular cell in memory. Consider two threads each of which is to add 1 to a shared data item, var. To add 1 to var, it will necessary to read the cell, compute the operation $var + 1$, and then write back the result to the same location. If the original value was 0, and the two threads read the value at the same time, they would compute the operation $0+1$ simultaneously and then write $var = 1$, which will result in $var = 1$, rather than $var = 2$; this is called *race condition* [53].

In order to avoid a *race condition*, there exist several techniques to stop a thread from accessing a section of code where shared data access would take place; which is called *critical section*. A *critical section* characterizes for starting typically with a *Lock* and ending with an *Unlock* which can be emulated using *semaphores* or *mutexes*, where a *mutex* is the most common. A *Lock* essentially operates as a door lock, where only one thread can acquire the lock at a time, and the other threads are compelled to wait until the owner thread of the lock has unlocked it [53].

1.6 Not embarrassingly parallel

The parallelization of a for-loop is usually the easiest implementation as long as each iteration is independent of the preceding one, otherwise, a for-loop would not be parallelizable. This is easy because the number of possible independent tasks are well-known from the beginning and it is typically known before running a program, which can be easily assigned to a fixed number of processors, a.k.a embarrassingly parallel [26].

However, certain implementations as parallelization of branching algorithms are far from being trivial, since the number of tasks evolves dynamically and their number is not known in advance, as illustrated by the cluster editing algorithm in Algorithm 2. Furthermore, branching algorithms are not guaranteed to explore all their branches, which makes it even more difficult to reach parallelization.

These algorithms can potentially create an exponential number of tasks, which should be properly handled such that parallelization is doable and efficient, otherwise, parallelization would be implemented but attaining worse performance.

1.7 Probability of speedup

In this section, we present the governing law of parallelism which states viability and limitations based on multiple factors.

1.7.1 Amdahl's law

Amdahl's law states that given a piece of a program for which we aim to parallelize if there is a proportion that does not get benefit from parallelization, that is, sequential operations are enforced, then there is a maximum theoretical performance that can be attained regardless of the number of processors utilized for the problem.

Thus, for a program that needs 20 hours to reach a solution using a single processor, if the non-parallelizable portion, x, takes one hour, there are therefore 19 hours that could benefit from parallelism, yet it will never be less than 1 hour. Amdahl's law writes as follows.

$$speedup = \frac{1}{x + \frac{1-x}{N}}$$

where N is the number of processors used to perform the parallelizable portion. Note that this is subject to a non-parallelizable section that must be run sequentially for the whole program. On the other hand, if a program is composed of individual independent tasks, x will decrease to only critical sections where a global variable would be modified by the processors. From the equation above, we see that when x approaches to zero, the speedup approaches the ideal value which is N [7].

Furthermore, a critical section does not necessarily mean that it has to be sequential, on the contrary, it only means that a thread can access a memory cell at a time to avoid race conditions as mentioned in Section 1.5 because each processor may be executing an independent task. If a task can be split into N sub-tasks, one could expect an average speedup of N, however all tasks may be conditioned to halt when the first sub-task has attained a solution, thus resulting in speedups above N if this sub-task is solved in in less time than the average as we will see in Chapter 3.

1.8 Types of branching algorithms

For the sake of this project, we created two groups to classify all branching algorithms, as follows:

— **Waiting algorithm**: At every recursion level, it returns an element that would be crucial to decide if exploring other branches or it is required for comparison or merging purposes. A branching algorithm is of waiting type, because if each branch were executed by a different processor, then the processor in charge of the scope will have to wait for their return value before being able to leave the scope. An example of this algorithm is the merge-sort [32]. It is worth mentioning that different techniques to avoid waiting could be implemented, however for the sake of this study and simplicity, this is applied to the aforementioned scenario.

— **Non-waiting algorithm**: finds a solution when reaching a leaf and this local solution is compared with a global variable to see if a more optimal solution has been found or if it has to be discarded. A branching algorithm is of type non-waiting because if each branch were executed by a different processor, then the current processor will not have to wait from their return value, and then the processor in charge of the scope is free to leave it as soon as it finishes its corresponding task. i.e., the cluster editing algorithm.

When implementing an algorithm programmatically, this one is essentially defined as a function, regardless of the programming language. All programming languages have in common that functions might return a value (int , float , ..., etc) or not (void).

Thus, a waiting algorithm is coded as a function (returns a value) and a non-waiting algorithm as a procedure (function that returns nothing). This is important because each one has its respective limitations and advantages when being solved in parallel.

Consider the pseudo-code presented in Algorithm 3 for a *waiting algorithm*, this algorithm receives an instance, which is processed to create the sub-instances I_{left} and I_{right} at each recursion level. Let's say, there are two processors, p_1 and p_2 that will be used to divide a binary branching algorithm, where p_1 is the processor running the current scope. I_{left} could be sent to p_2 and I_{right} recursed by p_1 at level 0. Since it is a waiting algorithm, if either p_1 or p_2 returns before the other, then one of them will be unutilized by the algorithm until the other has finished. This brings us to two scenarios as follows:

— p_2 **returns first**: since p_2 was invoked in this scope, then it is no longer attached to the result. Hence, it becomes idle and able to receive another task that p_1 may spawn within its search domain.

— p_1 **returns first**: it is compelled to await(idle) p_2 and this processor cannot be assigned to another task until it has already culminated its original instance.

If a task queue is being employed to distribute loads over the processors, where a task can be a branch call, it can be seen that the queue size must be constrained to the number of available processors. Assuming a queue size of 3 and considering the same task distribution as per the previous paragraph; if p_2 spawns a sub-task, it will be compelled to wait until some processor has treated it, but p_1 is also forced to wait for the return value from p_2. This is an infinite waiting, because p_1 could be waiting for the return value from p_2, and p_2 ends up waiting for any processor to handle the sub-task, but no processor is currently available to handle any other task.

```
1 function binary(I)
2     if I meets termination condition then
          // return branch_solution
3
      // Some operations to obtain $I_{left}$ and $I_{right}$
4     $r_{left} \leftarrow$ binary($I_{left}$)
5     $r_{right} \leftarrow$ binary($I_{right}$)
      // scope_solution $\leftarrow$ do something with $r_{left}$ and $r_{right}$
6     return scope_solution
```

Algorithm 3: Binary waiting algorithm.

Now consider the pseudo-code presented in Algorithm 4, if tasks are distributed as per the previous example, in this case, the processors are free to return without waiting for the result from any other processor. Once a processor becomes idle, special techniques can be used to re-assign these tasks. Also, if a task queue strategy is employed, there is no limit on this size, however, branching algorithms could make this queue grows exponentially, for which proper control should also be implemented.

```
function binary(I)
    if I meets termination condition then
        // Update global value

    // Some operations to obtain $I_{left}$ and $I_{right}$
    binary($I_{left}$)
    binary($I_{right}$)
    return
```

Algorithm 4: Binary non-waiting algorithm.

1.9 Inter-process communication strategies

In this section, we discuss the two major approaches used to establish communication among processors, especially if massive parallelization is part of the objectives.

1.9.1 Centralized topology

The most common communication model is centralized, where a single processor is in charge of tasks assignments. In this type of topology, all generated tasks typically pass by the center processor, which decides which processor would receive these tasks. Thus, the center acts as a tasks redirector [53].

Whether it is a multiprocessing or multithreading implementation, a centralized approach characterizes for having a single processor dedicated only to handle communication and tasks redirection, as illustrated in Fig. 1.1.

However, even though this strategy attempts to decrease communication overhead it still faces this issue since all tasks must pass by the center. That is, if the weight of a task is 100 MB, then this memory data should be transferred between a worker to the center, which is then transferred from the center to another worker. This saturates the communication band twice for the same task, which would be equivalent to

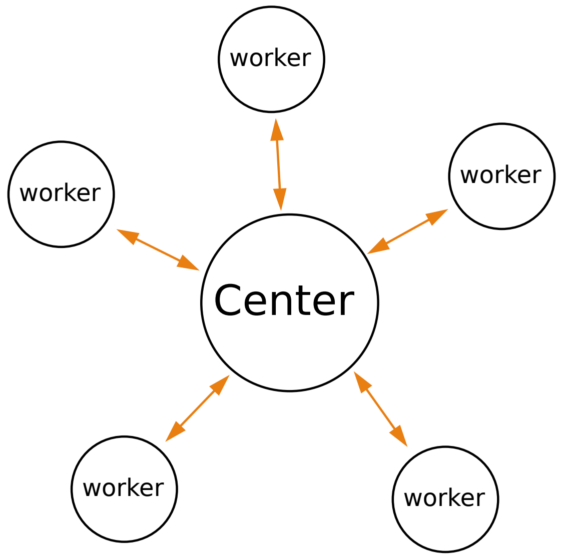

Figure 1.1 – Centralized topology. Arrows are tasks passing.

transferring 200MB.

It seems that all responsibilities lie on the center processor, but workers need to constantly keep track of their local tasks and communicate to center when they request or generate tasks. In addition, there is continuously a request/reply protocol that will result in success or failure for approved transfers or neglected transfers respectively, thereby saturating the communication band.

It is worth highlighting that most of the communication overhead is irrelevant when processors belong to the same computing node due to the possibility of sharing memory (if implemented). That is, tasks do not necessarily need to be copied and sent over the network. On the other hand, massive parallelization takes place on supermachines that require network communication in which copying tasks is unavoidable when communicating two processors on different machines.

1.9.2 Decentralized topology

Since the centralized approach faces at least two package transfers for the same task and the other aforementioned issues; moreover, a significant disadvantage of the centralized strategy is that the master process can only issue one task at a time.

The decentralized strategy aims to improve the communication surplus by allowing all processors to communicate with each other, in which an initial processor is in charge of splitting the first task and therefore the other processors will split theirs as long as there are available processors, for which some sort of hierarchy is typically implemented to avoid chaotic communication attempts, as illustrated in Fig. 1.2.

We should acknowledge that there are two mechanisms to implement a decentralized topology, which are essentially by using multiprocessing or multithreading[53].

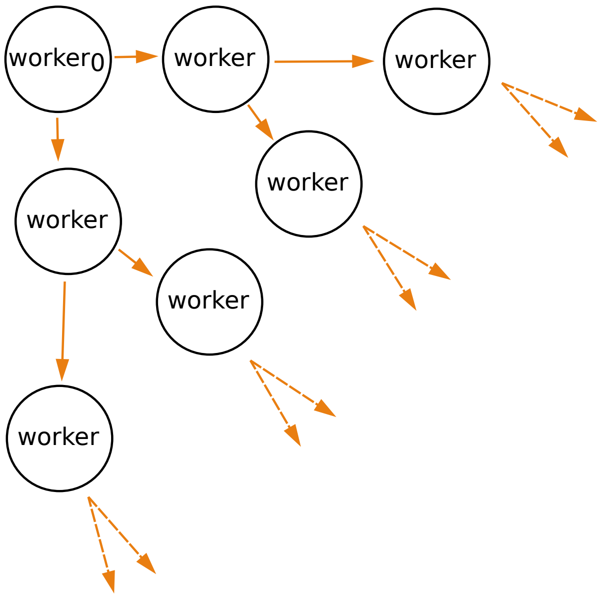

Figure 1.2 – Decentralized topology. Arrows are tasks passing and the sub-index 0 refers the process that initializes the execution.

In principle, multithreading implementation is decentralized by default, since all threads belong to the same process and therefore share memory. Multithreading synchronization has been well studied and is also well defined. This approach is generally tackled by implementing a thread pool, which allows spawning threads that match the number of processors only once. Threads generally dequeue tasks from a global job queue meaning that processors are utilized continuously.

Special techniques should be applied to avoid infinite waits for the waiting algorithms, which do not allow to enqueue more tasks than available processors because they might end up in line waiting for one another, where $p_1 \rightarrow p_2 \ldots p_{n-1} \rightarrow p_n \rightarrow unassigned\_task$ and the last processor waits for a task to be assigned.

Although non-waiting algorithms do not face this infinite wait scenario, the task queue should be adequately controlled, otherwise, it might grow indefinitely causing memory overflow, as discussed in Section 1.8.

Lastly, the greatest limitation of multithreading is that they cannot be distributed across several computing nodes because they are attached to a single process, which does not make it suitable for massive parallelization.

For multiprocessing implementations, proper synchronization should be attained using asynchronous communication the most, rather than synchronous communication, where asynchronous communication allows a processor to continue performing tasks as the communication is completed in the background as explained in Chapter 4. That is, threads have mutexes that are optimized for them where they turn into sleeping mode when awaiting a critical section. This can be emulated using processes but should be avoided since communication overhead is now part of the equation.

Inter-process communication should guarantee that processors continue working on the problem even if there is still a delay of the global information. This delay is strongly bounded to the bus or the network depending if processes are on a single machine or scattered across multiple computing nodes.

Thus the challenge is to adequately establish communication between processes, meaning that a process will attempt to request/notify another process that it needs a new task or it has a new task to share.

Even though this technique is promising, it is difficult to maintain a global value across the computing nodes because the processes might not be up-to-date and would be computing useless instances since another process might have discarded it already. This is solved by broadcasting the best solution so far and their availability to compute more tasks. Unfortunately, broadcasting comes with a significant communication overhead, which is part of the issues we address in this study.

1.10 Reduction rules

Problems like the Minimum Vertex Cover, Travelling salesman, or Cluster Editing, are typically solved using an enhanced brute force approach, which is to say branch and bound. Algorithms to these problems are typically exponential time in which for the worst scenario, all possible combinations must be explored to achieve the optimal solution.

Hence, the larger the graph the longer it takes to attain a solution and this solution time grows accordingly with the algorithm complexity. Thus, for an algorithm that has a complexity of $O^*(2^n)$, where n is the graph size; each additional vertex duplicates the number of instructions under the worst scenario. Then, applying a binary brute force algorithm to a graph of size 400 would represent at most 2^{400} possible solutions, which is an absurdly large number and therefore almost impossible to solve with nowadays technology.

There are certain logic statements that can be applied to these graphs in order to reduce their initial size and therefore the total solution time. These logic statements, also known as preprocessing rules, analyze the input graph and prune vertices depending on the problem.

Chen et al., [15], presented some of the most common rules for the MVC problem, as listed below.

— **Rule N.1** An isolated vertex (one of degree zero) cannot be in a vertex cover of optimal size. Because there is no edges incident upon such a vertex, there is no benefit in including it in any cover. Thus, in Fig. 1.3, an isolated vertex, u, can be eliminated, reducing n_0 by one. This rule is applied repeatedly until all isolated vertices are eliminated.

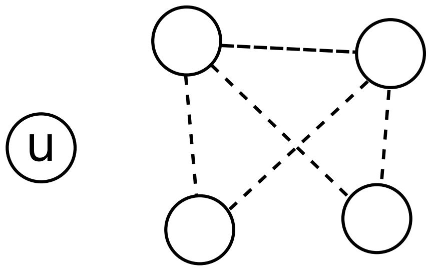

Figure 1.3 – First rule

— **Rule N.2** In the case of a pendant vertex, u (one of degree one), there is an optimal vertex cover that does not contain u but does contain its unique neighbor, v. Thus, in Fig. 1.4, v can be added to a growing partial solution. It can then be removed, along with all its incident edges that it covers. As a result, u becomes an isolated vertex and can also be removed due to the previous rule. This reduces n_0 by the number of deleted vertices and reduces k by one, where k is the FPT value. This rule is applied repeatedly until no more vertex can be removed in this way.

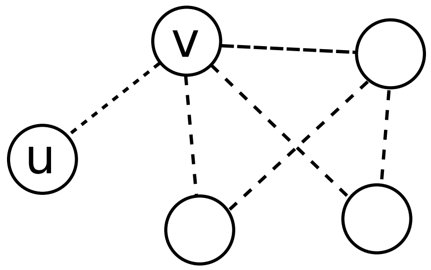

Figure 1.4 – Second rule

— **Rule N.3** Suppose that there is vertex u with exactly two neighbors v and w, such that v and w are adjacent. Then we may assume that both v and w are in an optimal solution. Indeed, if v is not in the solution, then both u and w must be in it to cover the vw and uv edges. In this case, we can replace u by v in this solution. The same argument applies if w is not in the solution. Then parameter k is reduced by 2.

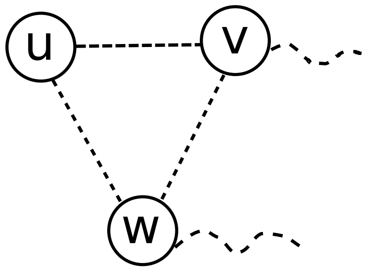

Figure 1.5 – Third rule

— **Rule N.4** Having a vertex u with exactly two neighbors v and w such that v and w are not adjacent to each other. Then u, v, and w are removed and replaced by u' that is adjacent to all neighbors of v and w, also known as the folding rule. Once a solution is achieved, the cover S is scanned for the folded vertices; thus, if $u' \in S$ then v and w are present in the cover, otherwise u is in the cover.

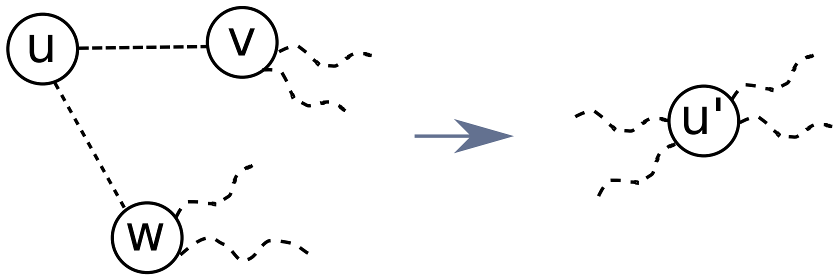

Figure 1.6 – Fourth rule

Thus, for our graph of size 400, after applying these preprocessing rules, its size may narrow down to the point it finds the optimal solution without even running an algorithm, or it may not be applicable due to its own topology. These reduction rules are just examples of the available techniques, for which several other problems admit these types of strategies.

1.11 Vertical tree search

There are typically two strategies to solve a branching algorithm considering its search tree.

The easiest way is the vertical search, also known as *depth-first search*, which is a conceptual idea were the nodes in a tree are visited by giving priority to the deepest available node when exploring. That is, for a binary search scenario and the leftmost branch is the first one called in the algorithm, once a leaf is reached within this left-most branch, then the next branch to be treated will be the immediate rightmost of the nearest parent node and so on and so forth.

This vertical search is practical for sequential algorithms because if branch & bound techniques are applied, then a great number of branches will be just skipped. If we are lucky enough to explore this tree, we might not even need to explore the rightmost branch of the root, which implies at least 50% fewer branches, therefore less execution time, as the example presented in Analogy 1.1.1.

Whether this seems to be optimal enough, it is conflicting when attempting to apply parallelization. Thus, having a fixed number of processors, each branch is a potential new task that can be delegated to another processor, also known as *randomized load balancing* [50]. However, even though the algorithm might be bounded, processors will likely receive light tasks because these tasks are being generated from down the search tree, more likely from bottom to top which might result in exponential parallel calls. Thus, assigning an exponential number of tasks to processors will not necessarily accelerate the solution in the worst-case scenario because forwarding a task might be more expensive than solving it sequentially.

Another fact to mention is that the leftmost branch will likely be the one generating more tasks, since other processors will receive light tasks that will not have that many branches to explore, thereby constraining the CPU utilization.

As per the latest paragraph, the CPU load is quite balanced from the point of view that they are constantly working on tasks, yet there might be more processors awaiting probably light tasks than processors generating meaningful jobs. Thus, these other processors will remain on standby for a little fraction of time until a new task is created and assigned to them, which is considerably significant in branching algorithms with exponential time solutions.

In consequence, CPU idle time and an excessive number of parallel calls must be properly managed with a correct load balancing strategy. If CPU idle time is decreased as much as possible, this could accelerate the solution a few folds in combination with the reduction of parallels calls. The second and better approach is discussed in Section 3.3.

Chapter 2

State of the Art

In this section, we discuss the current state of the art relevant to parallel implementations of branching algorithms.

2.1 Dual processor scheduling with dynamic reassignment

In 1979, Bokhari [12], presented his work aiming to optimally partition a modular program over a dual-processor system. His idea can be summarized as taking an input task and split it into equal parts such that each processor receives a sub-task. Having a search tree of a branching algorithm, all branches at depth $\log_c p$, where p is the number of processors and c is the branching factor, will become new sub-tasks that will be assigned to a processor P_i, with $0 < i < p$. An illustration is presented in Fig. 2.1.

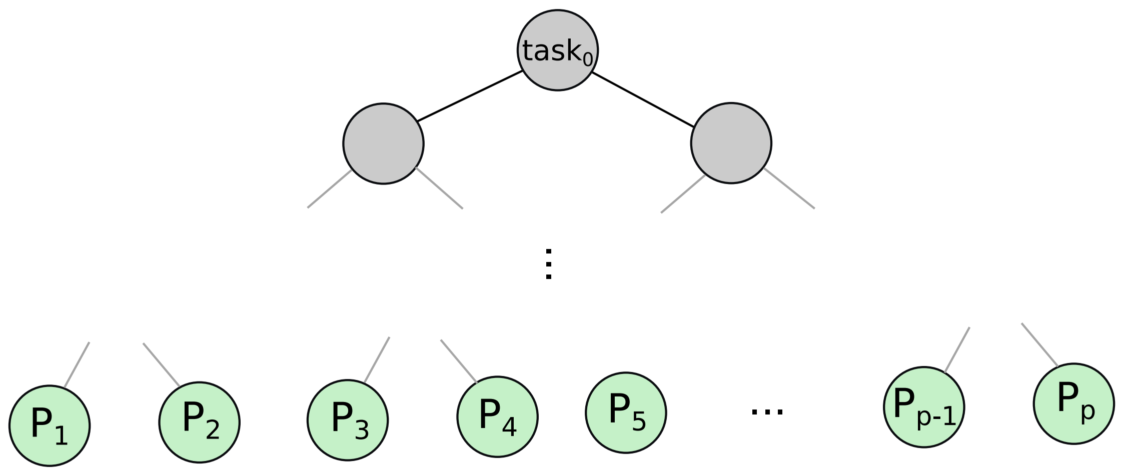

Figure 2.1 – Equitable task distribution. At depth $\log_2(p)$, each processor receives a task.

However, this comes with an important assumption, which is that once a task is assigned to a processor, it remains on that processor while the characteristics of the computations are constant. Additionally, it assumes that the search tree is well-balanced.

Hence, whether it is the most intuitive way to parallelize a branching algorithm, there is no guarantee that all branches are equally complex. That is, $p-1$ processors could exhaust their tasks and turn into idle mode really quickly because the heavy operations might be in the remaining branch, meaning that only one processor ends up performing the rest of the search. This happens because processors are not re-used after they return from their tasks, and therefore, the performance attained from this parallelization technique might be insignificant. Please see [52] for additional information on the performance of this strategy.

2.2 Multicomputers: message-passing concurrent computers

In 1988, Athas et al., [8] improved the previous attempt by dynamically allocating processors to newly generated tasks in a random fashion. They also called it *Random placement*. They achieved this by reusing the processors as soon as they finish their tasks.

The approach consists of a processor-like pool where a processor p_i starts exploring the branching algorithm. Before going sequentially into a branch, it checks if there is an available processor. If there is at least one, it will prune that branch from its search domain, to avoid performing it twice, and it sends it to the available processor, as illustrated in Fig. 2.2. This process is repeated always before branching, allowing to keep all processors busy and decreasing the idle time. This strategy has also been called the *Work-stealing approach*.

The author reported that random placement performs remarkably well and requires no global information. They also concluded that there is a software overhead regarding the message passing technique implemented, which was subject to further studies.

While this technique successfully accomplishes parallelization and decreases idle time, there is a bias towards assigning tasks lower in the search tree, meaning that processors are likely to receive light tasks that might have been faster to solve sequentially rather than being sent to another processor. Since these tasks are being generated from an exponential time algorithm, then there is a great number of them due to the vertical task creation. It is not easy to see either that each processor takes a fraction of time to dispatch its last job and then notify somehow that it is free to receive another task. This might be only a few CPU cycles, but they impact meaningfully in the overall performance due to the exponential number of calls.

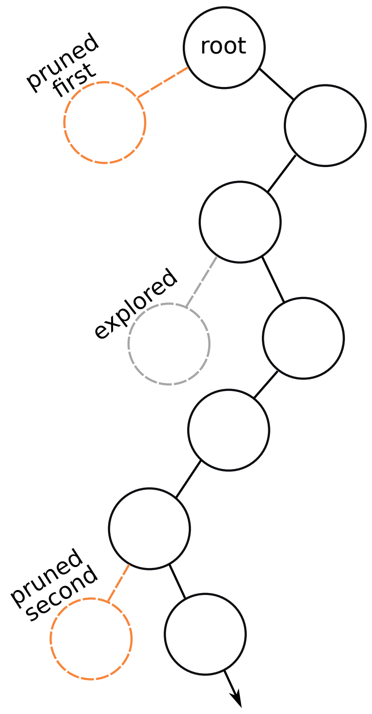

Figure 2.2 – Greedy Load Balancing within the search domain of a processor. Dotted nodes have already been resolved, orange for another processor and gray for sequential by the current processor.

2.3 Scalable Parallel Algorithms for FPT Problems

Aiming to minimize processors interruption, the authors in [4], developed a proactive parallel decomposition strategy, which acts essentially as a fully centralized approach. In their research, they created a driver that acts as a center and is in charge of assigning tasks to all the participating processes as long as they are available. The center maintains a job queue where the unvisited branches are stored temporarily, which is used to constantly feed processors and keep them busy, and thereby minimizing the idle time, as illustrated in Fig. 2.3. The job queue is constructed by addressing the tasks that are more likely to contain a potential solution and, tasks that may generate more heavy sub-tasks.

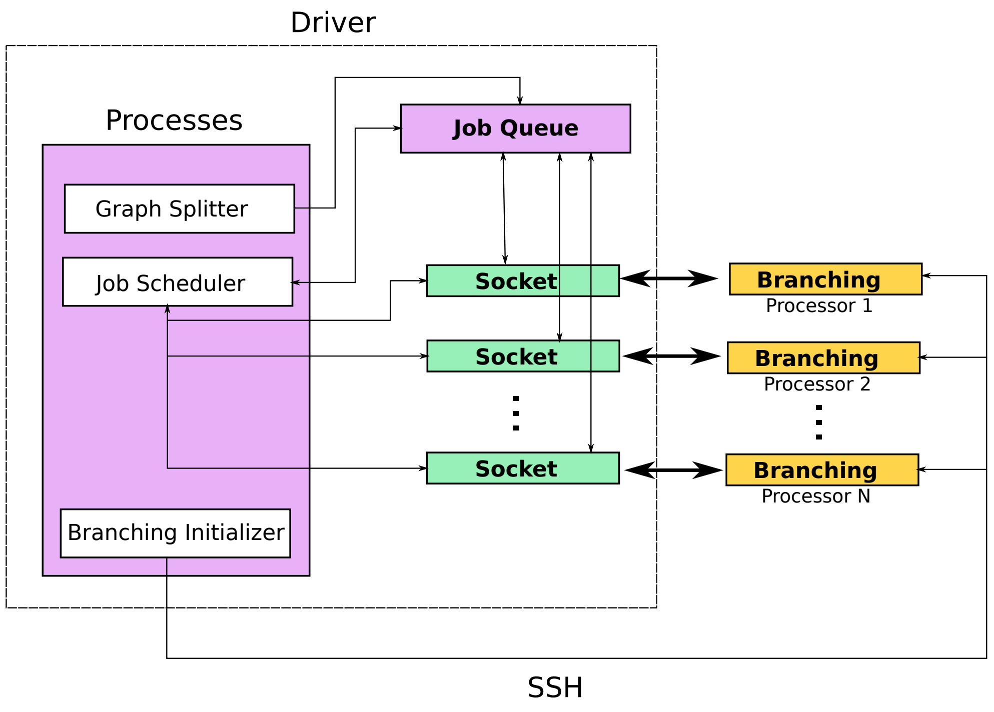

Figure 2.3 – The proactive parallel decomposition.

The execution is initially done by populating the processors with the branches at depth $\log p$, where p is the number of processors in an equitable fashion. The main challenge they faced was to properly control the job queue since when decomposing a task using a branching algorithm, this queue can grow exponentially. Moreover, constraining the queue size may lead to a job starving state, where processors will end up in standby mode.

The authors highlighted that no message-passing library was employed like MPI, because active processes had no need to communicate with one another. They reported to accelerate algorithms that in its static(equitable) parallelization test lasted 6+ days, to barely a couple of hours.

The queue size was controlled by splitting only "large subgraphs" that may lead to a significant amount of computation, which is, in turn, a downside due to the need of developing a method that enqueues the most promising tasks and the communication overhead introduced by properly controlling the queue such that it does not starve. These large subgraphs may lead to an unbalanced search tree, since these sub-instances may appear more often on one side of the search space, depending on the algorithm itself. In addition, branching algorithms tend to saturate quickly job queues, and thereby skipping relevant tasks that might have been worth it to process in parallel. A fully centralized strategy was also implemented in [31].

2.4 The buffered Work-Pool approach

Abu-Khzam et al. [6], presented a centralized topology that combines threading and message passing among processors of any supercomputer. As per Fig. 2.4, it consisted of creating a center-worker technique where the center is in charge of distributing tasks according to workers' availability. This approach allows a process to manage their own local shared work-pools using their local threads. Then, communication overhead is attempted to be decreased by allowing inter-processes communication for certain tasks but attaining the benefit of shared memory for threads within the process domain.

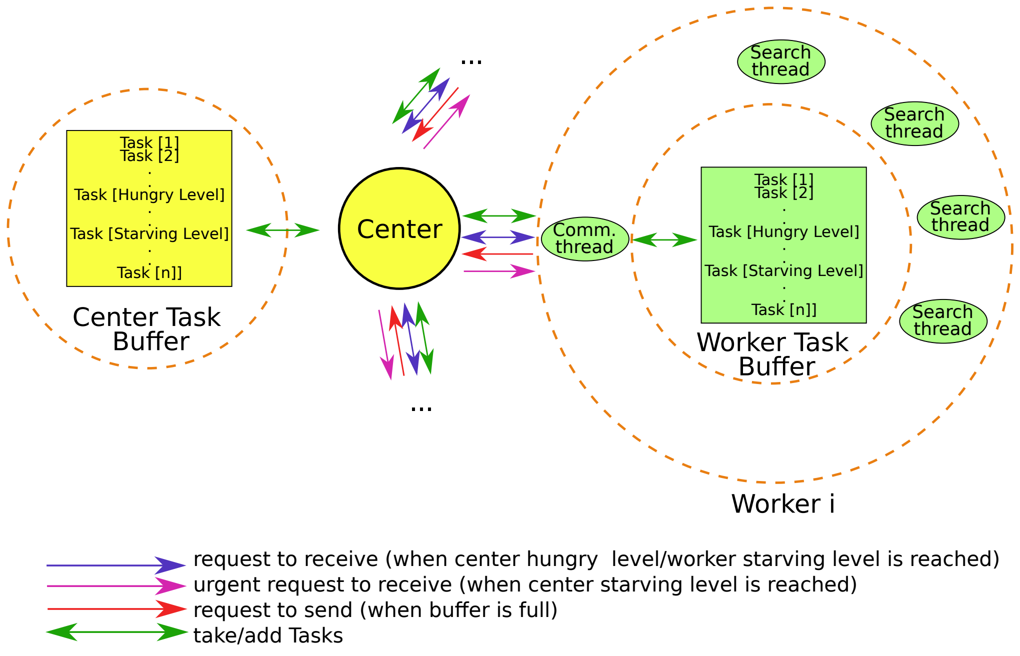

Figure 2.4 – Buffered Workpool approach.

Their approach used a system of task queues, where each process has a fixed-size queue of prioritized tasks that serves as a constant task supplier such that processors are always busy, and therefore reducing idle time.

It is worth mentioning that the authors presented a preliminary study, for which no massive parallelization was conducted and their average results demonstrated relative linear speedup. Though, this implementation is promising for more generic scenarios, where they were testing for problems like the *Minimum Vertex Cover*, *SAT* and *Maximal Cliques Enumeration*. Nevertheless, further studies and tests were pending to show their effectiveness on large scale.

2.5 Parallel Vertex Cover: A Case Study in Dynamic Load Balancing

The authors in [52], implemented a parallel approach for the *Minimum Vertex Cover*. They used one of the simplest strategies to tackle the *MVC* algorithm combined with *kernelization* and *FPT* techniques. For the *kernelization*, they used some of the rules previously mentioned in Section 1.10. As for the branching strategy, one of the highest or the highest degree vertex v is chosen and included in a partially constructed cover S, and k is decremented by 1 for the left-hand branch. For the right-hand branch, the neighbours of the vertex $N(v)$ are selected and included in S, while the parameter k is decremented accordingly, $k - N(v)$, as shown in Fig. 2.5.

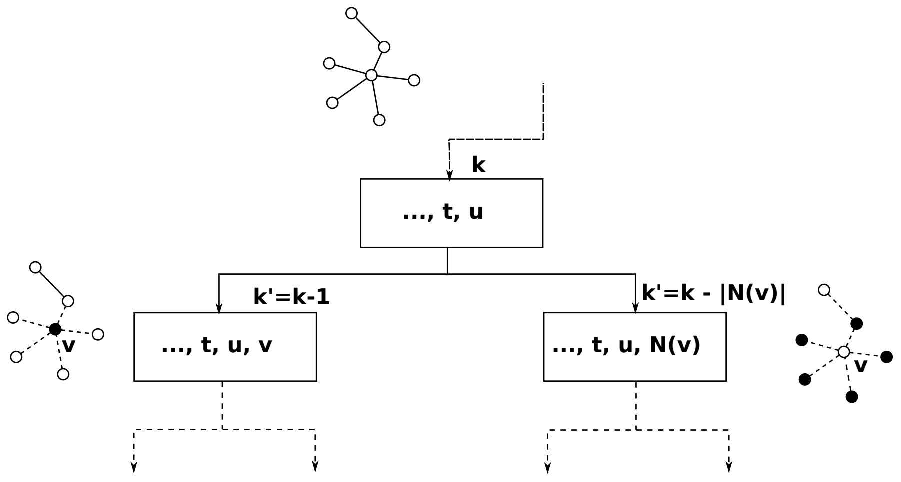

Figure 2.5 – Branching algorithm for the *MVC*.

The authors compared the *equitable* parallelization method previously presented in Section 2.1 in order to demonstrate how parallelism does not always result in better performance. Thus, to tackle this *equitable* parallelization approach, they developed a *Dynamic Load Balancing* strategy so the processors could be re-utilized each time a task is fulfilled.

Based on the fact that the leftmost branch removes a vertex one by one, they came to the conclusion that this might be the hardest instance. Thus, their strategy consisted of a *scheduler* that dedicates a processor to this branch, for which they called it the *donor*. Then, the *donor* is the only one in charge of job donating due to the computing cost of tracking tasks, and when a processor turns into waiting mode, it notifies the scheduler of its availability. The scheduler notifies the *donor*, and this one sends a task to the requesting processor. Once the *donor* finishes its instance, it notifies the *scheduler* and this one sends a termination signal to the other processors, meaning that the *donor* has no more task to give.

They demonstrated to achieve scalability of this approach up to 2,400 processors. However, this strategy was specific to the *MVC* problem, which does not make it directly suitable for other algorithms. Besides, it can be seen that it is a constrained scenario of the *work-stealing* technique presented in Section 2.2 for the tasks management, and whether the left-most branch of an algorithm may be the hardest instance, decomposition of this branch may not necessarily lead to a faster optimal solution.

2.6 On scalable parallel recursive backtracking

In order to minimize the fully centralized strategy flaws, Abu-Khzam et al. [2], developed a virtual core topology where the cores are initially assigned hierarchically such that a core has a parent to request tasks from and children to send tasks to, as shown in Fig. 2.6.

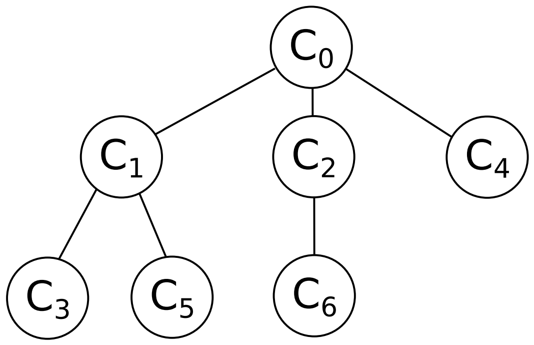

Figure 2.6 – Core topology, for c = 7.

The idea is to equitably distribute the most urgent tasks at depth $\log c$ of the search tree, where c is the number of participating cores, as shown in Fig. 2.7. This core topology is intended to match the search tree topology, yet it is not enforced to be maintained, as it evolves when a task request fails, then the core switch to another parent. During the exploration, each core c_i updates its parent, and each time it concludes its task, it attempts to request another from its parent. If the parent is idle or has no tasks, it switches to another parent. This is repeated until the core obtains a task. Authors presented their strategy to allow this core topology to be initially constructed, which is flexible as each core should find a parent that gives it a task, as specified in Algorithm 5.

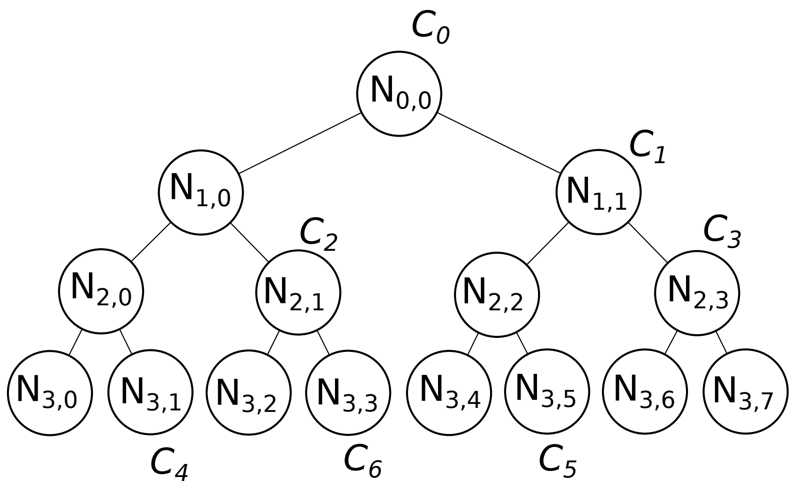

Figure 2.7 – Initial task-to-core assignment and core hierarchy, for $c = 7$.

When a core receives a task request from another, it will spawn a task applying a modified version of the strategy presented in Section 2.2. The authors applied a task indexing technique to keep track of the highest unvisited branches, in which each core will prune this branch when it needs to send a task. This indexing technique is relatively easy to implement on well-balanced search trees, however, it becomes more difficult to apply when the search tree is heterogeneous.

```
function getParent(r,c)
    parent ← 0
    for $i = 0..c - 1$ do
        if $2^i > r$ then
            break
        parent ← $r - 2^i$
    return parent
function getNextParent(r, c)
    parent ← (parent + 1) mod c
    if parent = r then
        parent ← (parent + 1) mod c
    return parent
```

Algorithm 5: GetParent and getNextParent functions. r is the core rank, such that $0 \leq r < c$.

This study demonstrated to be highly scalable, yet there is room for improvement. Since a core has to request a task, this introduces a fail request possibility, which increases according to the number of cores. Furthermore, due to the unbalanced nature of most algorithms, some cores are more likely to receive help than others.

Chapter 3

A Lightweight Semi-Centralized Strategy for the Massive Parallelization of Branching Algorithms

This article presents a novel inter-process communication protocol, which combines the advantages of a centralized topology by the means of a center process that acts as a task manager without passing tasks through. In a nutshell, a center process is responsible for dynamically assigning process-to-process delegations such that all inter-process requests result in successful communication.

We distinguish the center in our approach from the previous centralized work. In our implementation, the workers do not saturate the center with task requests since the center arranges only the strictly necessary communication, thus, a task is transferred between a pair of processes only once and it is always guaranteed to be delivered.

A simplistic but robust dynamic load balancing (*DLB*) is also presented, which is flexible to implement with heterogeneous search trees. The *DLB* guarantees that each generated task is likely the hardest task from the search tree it comes from, and thereby avoiding task queues. Priority is given to processes by default such that the heaviest jobs are sent to these processes, where each process will partition them accordingly with the number of threads they manage locally, yet a user is able to adapt priorities.

Thus, a proper combination of inter-process communication topology and dynamic load balancing, resulted in competitive performance, reaching super-linear speedups as a frequent outcome.

Contributions

The contributions within the domain of the article are listed here below.

— A novel fully generic framework for almost any branching algorithm, applicable for massive parallelization.

— A novel dynamic load balancing model for branching algorithms.

— Performance demonstration and effectiveness of the study.

This chapter is a slightly modified version of a paper submitted on the July 08th, 2021 to the *Journal of Parallel and Distributed Computing*.

A Lightweight Semi-Centralized Strategy for the Massive Parallelization of Branching Algorithms

Andres Pastrana, Manuel Lafond

Département d'informatique, Université de Sherbrooke,2500 Boulevard de l'Université, Sherbrooke, QC J1K 2R1

Abstract

Several NP-hard problems are solved exactly using exponential-time branching strategies, whether it be branch-and-bound algorithms, or bounded search trees in fixed-parameter algorithms. These are usually limited to small inputs, while massive parallelization has been shown to significantly increase the size of instances that can be solved exactly. However, previous centralized approaches require too much communication to be efficient, whereas decentralized approaches are more efficient but have difficulty keeping track of the global state of the exploration.

In this work, we propose to revisit the centralized paradigm while avoiding previous bottlenecks. In our strategy, the center has lightweight responsibilities, requires only a few bits for every communication, but is still able to keep track of the progress of every worker. In particular, the center never holds any task but is able to guarantee that a process with no work always receives the highest priority task globally.

Our strategy was implemented in a generic C++ library called GemPBA, which allows a programmer to convert a sequential branching algorithm into a parallel version by changing only a few lines of code. An experimental case study on the vertex cover problem shows that even with the simplest algorithm, we can solve the toughest instances from the DIMACS challenge graphs.

3.1 Introduction

Several scientific disciplines require solving NP-hard problems for which no polynomial-time algorithm is believed to exist. This includes, for instance, clustering proteins in biological networks [28], maximizing influence in a social network [33], or optimizing weights in neural networks [29]. Such NP-hard problems are usually handled by fast heuristics or approximation algorithms when the running times are crucial. However, the recent ease of access to high-performance architectures, combined with novel algorithmic techniques, have allowed researchers to aim for exact algorithms in reasonable times, even if above polynomial. These are usually exponential in the input size, and recent research has focused on making the algorithmic complexity tolerable for some practical purposes [23] (for instance, achieving a complexity of $O^*(1.23^n)$ for maximum independent set [22], where O^* suppresses polynomial factors). Branch-and-bound algorithms, which brute-force every possible solution but skip those that cannot do better than the current optimal, have also been studied extensively [17, 37]. Another recent research trend consists of fixed-parameter tractability (FPT), which aims to design algorithms that are exponential, but only with respect to a parameter that is expected to be small [38, 20, 19]. We also refer the reader to [55] for an excellent survey of exact algorithms for NP-hard problems.

In this paper, we focus on improving the scalability of such approaches with massive parallelization. A well-known technique for both exponential and FPT algorithms is recursive *search tree exploration*. In a nutshell, when given an instance I to solve, search tree algorithms generate a few sub-instances of I in a way that at least one of them leads to an optimal solution. The algorithm then explores each sub-instance recursively until a solution is found, or until the whole space has been searched, depending on the algorithm. This forms a recursion tree in which *nodes* correspond to function calls and *children* correspond to its recursive calls.

There is an extensive literature on the problem of parallelizing search tree algorithms. If p processes are available, the most straightforward strategy is to assign each node at depth $\log p$ its own process, and see which one finds a solution [12, 14]. This works in an idealized setting where each process is assigned a tree of about the same height, but most search trees are unbalanced and some processes will finish before others. In this case, they can and should be reassigned to other subtrees of the search tree, leading to the problem of dynamic load-balancing.

It is not obvious how to perform this optimally, since massive parallelization of search trees introduces two problems: how to distribute search trees to processes, and how to minimize communication. Unsurprisingly, there is an inherent tradeoff to choose from between communication overhead and search efficiency, as more communication allows assigning free processes to the most important or most promising task globally available. A *centralized* strategy was developed by Abu-Khzam et al. [4] with this idea in mind, where a center would maintain a queue of available tasks and nodes, and thus could always make optimal assignment choices. As argued later by Abu-Khzam et al. [2], communication overhead is not worth the gained efficiency, especially since tasks may require sending lots of information. An opposite *decentralized* strategy was therefore developed. The idea is to arrange the cores into a virtual hierarchy and let cores only accept tasks from their superior. This optimizes communication but no core has an idea of the global situation and, as we argue in this paper, this leads to suboptimal task assignment and exploration. We also refer the reader to [46, 43, 30, 48, 5, 49, 52] for further works that have focused on how to explore tasks efficiently in parallel.

Our contributions. In this paper, we propose a novel *semi-centralized* load-balancing strategy that takes advantage of both approaches and offers a balance in the communication versus exploration tradeoff. The main idea is to make use of a central process, but in an extremely lightweight fashion, in the sense that communication with the center is asynchronous, limited, and always requires only a few bits of information. The center is relieved from the heavy responsibility of maintaining a task queue, and instead is only present to maintain the status of working processes and dynamically decide which processes should exchange tasks. The heavy task communication is only performed between working processes that need to share information, and only when it is necessary to do so. In particular, an idle working process only needs to request work once, in contrast with previous solutions where such requests could fail and require multiple communication rounds. We also propose a strategy that allows each worker to maintain the hierarchy of its highest priority search tasks which is applicable to search tree algorithm with any branching factor, even if it is heterogeneous across the search tree. Moreover, our approach allows a *process-thread* hybrid implementation. That is, a subtree assigned to a process can be partitioned into further subtrees, each assigned to a different thread. Our strategy is implemented in a generic, open-source C++ library called GemPBA. The library uses Message Passing Interface (MPI), is user-centric, and a programmer is able to parallelize any existing sequential search tree function by changing a few lines of code.

We use the traditional vertex-cover problem as a case study. We demonstrate that even with the simplest branching implementation for vertex-cover, our library achieves close to linear speedup and can solve some of the toughest instances of the DIMACS challenge graphs.

3.2 Preliminary notions

In this section, we first explain the branch-and-bound and fixed-parameter search tree algorithms at a high level. To motivate the need for novel ideas in parallelizing these algorithms, we then discuss the main load-balancing strategies that have been applied to branching algorithms in the literature, along with their advantages and disadvantages.

3.2.1 Search tree algorithms

In essence, all branching algorithms have a similar structure. Given an instance I, we first verify whether I is a solution to our problem, which corresponds to a terminal case. Otherwise, we generate a set of (usually smaller) instances $I_1, \ldots, I_k$ from I in a way that at least one I_j can lead to a solution. We then explore each I_j recursively.

```
function searchTree(I)
    if I cannot lead to a solution better than best then
        return
    if I is a solution then
        if I is better than best then
            best ← I
        return
    $I_1, I_2, \ldots, I_k$ ← sub-instances of I
    for $j = 1..k$ do
        searchTree($I_j$)
    end
```

Algorithm 6: Structure of a sequential search tree algorithm.

We distinguish branch-and-bound and fixed-parameter algorithms, which we briefly describe since our methodology applies to both. In the well-known *Branch-and-Bound* (B&B) paradigm, we must optimize some value and the best solution found so far is stored globally. When reaching a terminal case, we check whether the solution is better than the best, and if so we update it. More importantly, whenever an instance is guaranteed to lead to a worse solution than the best, we stop the recursion. The *branching factor* is the maximum number of recursive calls the algorithm makes. Algorithm 6 presents the general structure of this type of algorithm. In the *fixed-parameter tractability* (FPT) paradigm, we instead have a decision problem that asks whether there exists a solution of size k. If I cannot lead to such a solution, we can return. If I is such a solution, we can return "yes" and stop all exploration (contrasting with branch-and-bound, which keep exploring). Notably, fixed-parameter algorithms are known for *kernelization*, which describe rules to reduce the I instance to a smaller size (see e.g. [21, 19]). Several parallelism ideas have been proposed for FPT algorithms [14, 2, 10].

As a concrete example of a search tree algorithm, consider the *vertex cover* problem. In the optimization version, we receive a graph $G = (V, E)$ and must find a subset $X \subseteq V$ of minimum size that touches every edge, i.e. for all $uv \in E$, $u \in X$ or

$v \in X$. A simple branching strategy goes as follows. Each recursion receives a partial solution $S \subseteq V$, with $S = \emptyset$ at the initial call. We choose $uv \in E$ not covered by S and observe that we can either 1) add u to S; 2) not add u to S. In the second case, all the neighbors $N(u)$ of u must be added to S to cover its incident edges. We thus recursively branch into two subinstances: $G - \{u\}$ with partial solution $S \cup \{u\}$, and $G - N(u)$ with partial solution $S \cup N(u)$ ($G - X$ is the graph obtained by removing the X vertices). The recursive calls then check whether these partial solutions have more vertices than the current best solution, and if so do not explore it. In terms of FPT, the algorithm is the same, except that we stop exploring if the current partial solution has more vertices than k, the parameter. Vertex cover is a standard problem that is used for several experimental benchmarking tools [4, 2, 51, 3].

3.2.2 Previous search tree parallelization strategies

Several approaches have been proposed to parallelize branching algorithms. We present the main categories that we have identified, with an emphasis on full decentralization, since it has been reported to be able to solve the most difficult vertex cover instances.

Equitable parallelization Assume that p processors are available and that the branching factor is r. In [12], the authors propose to execute the algorithm sequentially until a depth of $\log_r p$ is reached. This defines a tree with p leaves corresponding to p instances, at which point each processor is assigned a distinct instance. This distributes the search tree across processes "equally", but an obvious disadvantage of this approach is that, once a process has finished exploring the search tree of its assigned instance, it is not recycled to help exploring other subtrees. This strategy was also used in [14] to solve FPT problems (along with several other strategies, including the usage of free processors for faster kernelization).

Greedy load balancing In [46], the authors apply process recycling to the search tree exploration. When a recursive call needs to branch into a new subtree T, if some process p_i is available, then it assigns p_i to T, and otherwise explore sequentially. We must assume that each free p_i broadcasts its avaibility to the others. We call this the greedy approach since a process assigns its most recent task to the most recently freed process as soon as possible, regardless of the current state of the search tree. This idea has also been called the randomized work-stealing approach. The advantage is that processes are constantly participating in some tree exploration. However, this strategy tends to assign processes *vertically*. That is, once a process p_i is assigned to a subtree it will start digging deeper and deeper into it. When another process p_j gets freed, it will be assigned to the current location of p_i, which is likely to be deep in its recursion. This tends to bias the exploration to similar parts of the search tree. Moreover, free processes get assigned small search trees, leading them to finish quickly and broadcast information more often. In [52], a hybrid strategy is proposed, where processes are first distributed in an equitable manner, and then reassigned greedily.

Fully centralized approach In [4], Abu-Khzam et al. have developed a strategy where a central process is responsible for receiving tasks from the other processes. The center maintains a queue of tasks of bounded size. When another process requests a task, the center can choose which one to send according to some priority function. Examples of priority include the task with the largest subtree to explore, or the task with the most promising solution so far. This is an important advantage, since such a priority scheme avoids the vertical-exploration problem mentioned above. The main drawback is the large communication overhead for constant requests to center. Also, exponential algorithms tend to saturates the task queue very quickly. The ability for the center to choose tasks with priority is therefore hindered by the fact that most tasks never make it into the queue, and in the end, there is little control over task priority. Let us also mention that in [52], the authors developed a centralized scheduler-based strategy specifically for the FPT vertex cover problem.

Fully decentralized approach To address the problems of full centralization, Abu-Khzam et al. explored the other extreme by devising a fully decentralized approach [2]. In this strategy, the available cores are organised into a tree (which should be distinguished from the search tree). In this topology, the parent of core number r is $r - 2^{\lfloor \log(r) \rfloor}$. Initially, the root of the core-tree is assigned the full instance. Then, each core requests a task to its parent in the core-tree. The degree distribution of the core tree is heaviliy skewed towards nodes near the root. The root of the core-tree has around $\log c$ children, while the majority of nodes have 0 or 1 children, where c is the number of cores. This topology is chosen because it initially assigns cores as in the equitable strategy.

When a core is finished with its instance, it asks its parent for a new task asynchronously, and if none is available, the request fails and it switches to a new parent. This strategy requires no synchronization, since there is no way a process can receive multiple tasks. Another important aspect of this strategy is that when a core r has a pending request from a child core q, r chooses to give q the sub-instance that is the highest in its search tree. An indexing of tasks is proposed to maintain the highest priority, which is the highest unexplored node in the search tree. The vertical-exploration problem is therefore avoided, although each core has its own priority instead of having a center that maintains global priority. Therefore, a worker receives the most urgent task from its parent, not necessarily the most globally urgent task. Moreover, because the core-tree is imbalanced, some cores are more likely to receive help than others. For instance, a subtree assigned to a leaf of the core-tree will never receive help (unless a lucky parent reassignment occurs), whereas $\log c$ cores are available to assist the subtree assigned to core 0. This inherent bias is difficult to circumvent in a fully decentralized setting, which motivates our new strategy. Finally, let us also mention that work requests can fail, which occurs when a parent has no work for a child. This forces the child to send another request to another core and, as the experiments in [2] show, this can have a significant performance impact on difficult graphs.

3.3 A semi-centralized load-balancing strategy

As we have discussed, centralized and decentralized strategies each have their own pros and cons. Here, we propose a novel strategy that is in-between. We do make use of a central process, but its responsibilities are reduced to a minimum, as well as its communication and memory requirements. In particular, tasks do not go through the center, as it only stores the smallest amount of information required to know which processes should exchange tasks. The center is designed with three goals in mind:

1. The center must never become overloaded, and its memory usage should be independent of the number of ongoing or pending tasks;
2. Communication and synchronization should be minimized. In particular, a process should never be waiting for a reply from another process, unless it has no task to work on. Also, failed work requests should be minimized or nonexistent.
3. When a process is available, it should be possible to assign it to the task with highest global priority. Moreover, the current best solution found globally should always be available to every process.

Let us also mention that our strategy is designed to achieve the ease of use illustrated in Algorithm 7. This shows the same algorithmic structure as Algorithm 6, where *GemPBA* does some bookkeeping and performs the recursive calls as well as handling the parallelization, as will be explained in Algorithm 10. Such a delegation scheme allows almost any sequential branching algorithm to be parallelized with minor modifications.

```
1  function searchTree(I)
2      best = GemPBA.getBestSolution()
3      if I cannot lead to a solution better than best then
4          return
5      if I is a solution then
6          GemPBA.handleSolution(I)
7          return
8      $I_1, I_2, \ldots, I_k \leftarrow$ sub-instances of I
9      GemPBA.addChildInstances($I_1, \ldots, I_k$, parent = I)
10     for $j = 1..k$ do
11         GemPBA.search($I_j$)        // Library handles parallelism
12     end
```

Algorithm 7: Structure of a parallelized search tree algorithm.

Our load-balancing is also required to be compatible with a combination of multithreaded and multiprocess exploration, so that processes could use thread for their exploration without affecting the above goals.

3.3.1 Center and worker responsibilities

In addition to handling startup and termination, the center is only responsible for:

— Maintaining the list of processes available to receive new tasks, and possibly other metadata on each process;

— Determining which working process should send a task to which available node, without conflict;

— Maintaining the value of the best global solution so far.

One can see that in terms of storage, center only needs to remember a simple array of process statuses, and a numerical value for the best solution. The metadata is optional and is intended to store the priority (an integer) of the most urgent task in each process. This metadata can be used to determine, when a process p_i is available, which process p_j has the heaviest task currently. In that manner, center can let p_i know that it should send it to p_j. Other assignments are possible if no metadata is present, for instance by choosing p_j randomly.

The workers are responsible for:

— Exploring the search tree of a given instance

— Communicating with center to request work

— Sending and updating the value of the best solution found so far

— Sending their heaviest pending task to processes assigned by center

Note that we aim for work requests to never fail, in contrast with the decentralized setting. This will be achieved by ensuring that the center links idle workers in a direct fashion creating a dynamic queue-like relationship $w_i \rightarrow w_j \rightarrow \ldots \rightarrow w_n$ with $i \neq j \neq n$, where multiple queues coexist or a single one at certain point since they are dynamic. However, the head of the queue, w_i is always guaranteed to receive a task, and if w_i does not create sub-tasks and returns, then w_i is assigned to the tail, w_n, of another queue or any other running worker without a queue. In consequence, as long as there is a running worker, all the others are guaranteed to receive sub-tasks, and if no worker is running, then it is interpreted as the end of the execution and the center sends a termination signal.

Also note that maintaining the heaviest pending tasks is not trivial. Later in this section, we propose a strategy that stores the state of the pending search tree nodes to achieve this, while only requiring an amount of memory that grows linearly. Fig. 3.1 illustrates the relationship between processes.

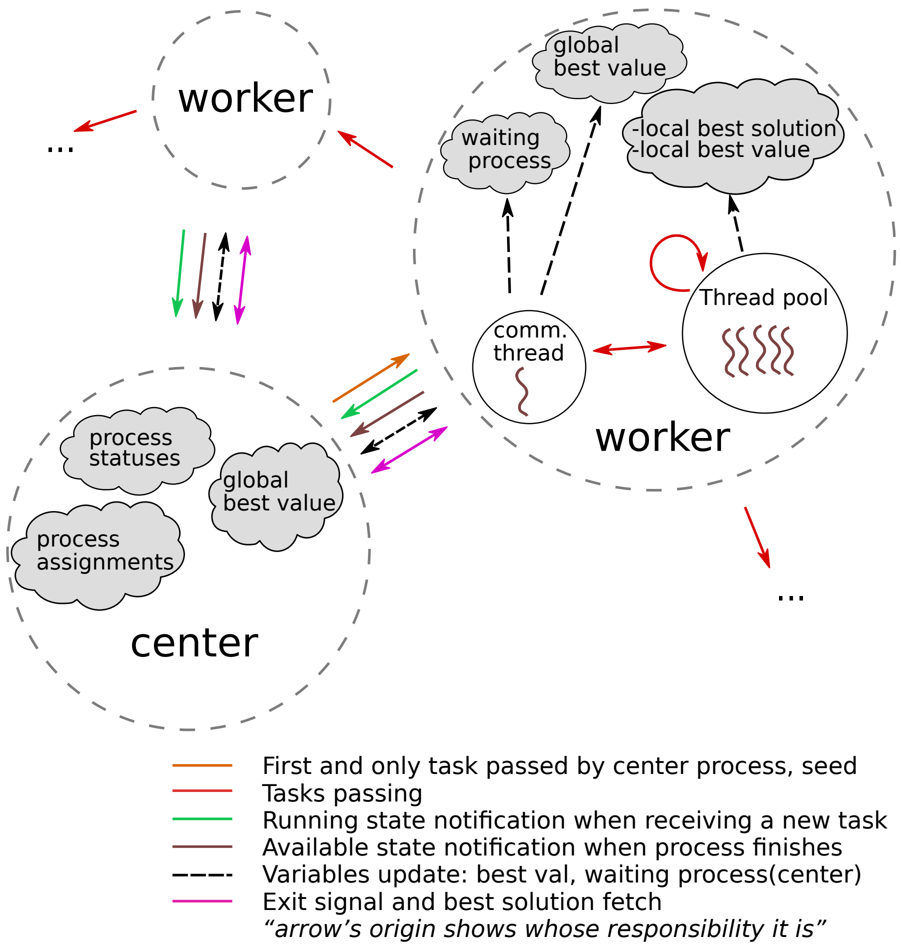
worker
...
global best value
waiting process
-local best solution
-local best value
Thread pool
comm. thread
worker
...
process statuses
global best value
process assignments
center
First and only task passed by center process, seed
Tasks passing
Running state notification when receiving a new task
Available state notification when process finishes
Variables update: best val, waiting process(center)
Exit signal and best solution fetch
"arrow's origin shows whose responsibility it is"

Figure 3.1 – Communication topology. The "seed" refers to the original instance that is sent to the first working process to initiate the exploration.

3.3.2 Center implementation

Given its lightweight set of responsibilities, the center is relatively simple to implement. Not shown here, during the initialization steps, the center loads the seed and sends it to the first worker. The high level ideas are shown in Algorithm 8. The reader should bear in mind that whenever a message is sent, there is no need to wait for a reply or a confirmation — everything occurs asynchronously. The center is in a listening loop and reacts to worker messages. To manage the optimal value, it stores a single numerical value $bestValSoFar$ to remember the global optimal and, when a worker thinks it has found a better value, it receives a *bestval_update* request. Since several such updates can be received in a short time span, center needs to verify this claim, but this is an easy check. If the best value indeed changes, center broadcasts this to all processes in a non-blocking fashion, so that workers can update their local best value when they have time.

```
function centerLoop()
    Recv($tag, source, data$)
    if $tag == bestval\_update$ and $data < bestValSoFar$ then
        $bestValSoFar = data$
        Async_Broadcast(tag = $bestval\_update$, data = $data$)
    else if $tag == available$ then
        $w = getNextWorkingNode(metadata)$
        if $w$ then
            Async_Send(dest = $w$, tag = $send\_work$, data = $source$)
            $status[source] = assigned$
        else
            $status[source] = available$
    else if $tag == started\_running$ then
        $status[source] = running$
        if $\exists w$ such that $status[w] == available$ then
            Async_Send(dest = $source$, tag = $send\_work$, data = $w$)
            $status[w] = assigned$
    else if $tag == metadata$ then
        $updateMetaData(source, data)$
```

Algorithm 8: Pseudocode of the center loop (assuming a minimization problem)

The center can also receive messages when a worker changes state. An array of states is maintained, with one entry per process. When a worker r has finished exploring its subtree, center receives an *available* message. At this point, center chooses a worker w with the $getNextWorkingNode$ function, and at this point w should send its heaviest task to r. This choice can either be made randomly, or according to some priority function based on the metadata. Importantly, the task to send does not go through center — rather, center sends a non-blocking message to w to let it know it must send a task to r, and then remembers that r is waiting by putting its state to *assigned*. This ensures that one and only one worker can now send a task to r, thereby avoiding conflict. Moreover, this assignment persists until w does send work to r, ensuring that the work request from r does not fail. It is possible that w has no work to send immediately, but in this case, center will ask some other process to send work to w, and the chain of task sending will resume. Of course, center can ensure that no cyclic dependencies are introduced, which is handled by a *GemPBA* data structure developed to follow the aforementioned rules.

When an idle worker r receives a task, it lets center know by sending a *started_running* message. If there happens to be an available worker that we were not able to assign, we can immediately ask r asynchronously to delegate to it (this happens rarely, since workers are usually assigned when they become available). Finally, workers can decide to send metadata to center when needed. Although there is no conceptual constraint in our framework, this should consist of small amounts of data, for instance an integer representing the priority of the heaviest task (Note that metadata is not related to the task to give, rather than information about the state of the exploration).

3.3.3 Worker implementation

Workers can either send new information to center, or react to center messages. This is done by periodically calling the update functions *updateWorkerIPC*, *updateWaiting-Processes* and *updateWaitingThreads* displayed in Algorithm 9. The periodical calls to these functions could be implemented in two ways. They could be called at the start of every call to the *searchTree* procedure in Algorithm 7. Another option is to dedicate a thread to updates and communications, which would call the functions in a loop. Although straightforward, the two options are worth mentioning because the former is not compatible with every inter-process communication library. Indeed, in a multithreaded environment, the first solution allows *any* thread to send remote messages, and *openmpi* has reportedly difficulty dealing with this [25]. Fixing a looping thread for these tasks is therefore easier to implement — the priority of this thread can be lowered or a small sleep can be added to make it use less CPU.

```
function updateWorkerIPC()
    if $hasMessage()$ then
        $Recv(source, tag, data)$
        if $tag == bestval\_update$ then
            if $data < local\_bestval$ then
                $global\_bestval = local\_bestval = data$
        else if $tag == send\_work$ then
            $addToWaitingProcesses(process = data)$
            $updateWaitingProcesses()$
        else if $tag == work$ then
            //this can only be received when no task is running
            Async_Send(dest = center, tag = $started\_running$)
            $sendTaskToNextThread(instance = data)$
    if $local\_bestval < global\_bestval$ then
        Async_Send(dest = $center$, tag = $bestval\_update$, data =
         $local\_bestval$)
    if $hasMetadata()$ then
        Async_Send($dest = center$, tag = $metadata$, data = $metadata$)

function updateWaitingProcesses()
    while $hasPendingTasks()$ and $hasWaitingProcess()$ do
        $sendHighestPriorityTask(dest = next\_process)$
    end
function updateWaitingThreads()
    while $hasPendingTasks()$ and $thread\_pool.hasIdle()$ do
        $sendHighestPriorityTask(dest = next\_thread)$
    end
```

Algorithm 9: Pseudocode of worker update functions. These functions should be locked by a mutex (which should be non-blocking, i.e. if a thread fails to acquire the mutex, the updates are skipped for the current pass). The tag *work* is sent within the *sendHighestPriorityTask* in Algorithm 11.

In any case, let us emphasize that workers should never be in a blocking listening mode. Instead, the update functions check whether center has left them a message in their receiving buffer and resume if not (which most libraries allow, for instance *MPI_Iprobe* using openmpi). We assume that workers store a variable *local_bestval* for the optimal value found by its threads, and *global_bestval* for the optimal value seen by center. When center sends a better value than the local, both are updated (protected by a mutex, since threads can change the local). Center can also ask to send work to another process r, in which case we add r to our waiting list. The update functions will eventually send work to those on the waiting list. Note that in our implementation, the center ensures that each waiting list has at most one element (except at startup).

After checking for center messages, the worker can decide to send its local best value if it thinks it is better than center's (and the latter will verify this). We also check whether a task can be sent to an idle thread if there is one, and metadata update can be sent to center if needed.

When a worker has finished exploring its instance, only one thread is active. At this point, the worker first lets center know of its availability by sending an *available* message (not shown). After that, it calls the $updateWorkerIPC()$ function in a loop. This allows the worker process to continue receiving all updates on the best value. This continues until a *work* message is received from another process. The worker lets center know that it started running again, and the received instance can be explored. To implement this, the received task could either be taken by the main thread, which will eventually assign tasks to the other threads, or the task could be sent to a thread in a pool, and the current thread would continue looping. Not shown here is a termination message that can be sent by center, which will end the current process if no exploration is ongoing (i.e. every process is available, and a sufficent time has passed since none of them has sent a *started_running* message).

3.3.4 Maintaining the most urgent task in workers

We have mentioned several times that the highest priority task should be sent to either free threads or processes, but have not specified how exactly. As in [2], our point of view is that in a search tree algorithm, the most urgent tasks correspond to highest nodes in the search tree (i.e. those of minimum depth). This spreads the exploration across more different parts of the search tree, which allows finding better solutions more quickly, thereby cutting useless branches more quickly as well. The recursion tree should be maintained in some way so that at any point, we can access highest nodes when a new task is required to be sent. Abu-Khzam et al. proposed to assign each node an index based on its location in the tree. A counter can keep track of the highest priority node and, when it is sent, the counter can be incremented. This is not too hard to achieve for binary search trees, but this gets arguably more complex for algorithms with higher branching factors, especially when they are heterogeneous across the tree.

Here, we propose an alternate method that is conceptually simple for any branching factor, even if heterogeneous across the search tree. The idea is simply to store the recursion tree explicitly, in a traditional tree data structure, while ensuring that the size of the tree does not grow exponentially. Maintaining a global tree in a multithreaded environment is somewhat complex to do, owing to its dynamic nature. Instead, we propose that each thread t_i maintains its own task tree T_i. The root of T_i is the task that was initially assigned to t_i, with the descending nodes resulting from the recursion. If needed, the highest priority task can be recovered by inspecting each tree stored by the threads. As usual, this management should entirely be performed by the library and should be independent of the branching algorithm.

Recall that in Algorithm 7, the *searchTree* procedure first passes the child instances $I_1, \ldots, I_k$ of parent I to *GemPBA.addChildInstances*, and then runs *GemPBA-.search* on each subinstance individually. The pseudo-code of these routines is illustrated in Algorithm 10.

```
function GemPBA::addChildInstances($I_1, I_2, \ldots, I_k, parentI$)
    for $j = 1..k$ do
        Add $I_j$ as a child of $parentI$ in the $T_i$ task tree
    end
function GemPBA::search($I$)
    if task $I$ is still in the $T_i$ tree then
        Mark $I$ as "Exploring" in $T_i$
        searchTree($I$)                         // Explore sequentially
        Remove task $I$ from $T_i$
```

Algorithm 10: Construction of the task tree T_i for thread t_i.

When an instance I generates child tasks $I_1, \ldots, I_k$ in thread t_i, they must be added in the task tree T_i. At this point, the update functions could decide to send one of these tasks at any moment to another thread or process. This is why the search procedure first checks whether an instance is still present before letting the current thread explore it sequentially. If there is only one process and one thread, this mimics the sequential version of the search tree exploration. When a child task and all its descendants are done, it can be removed from the tree.

Size of task trees. Let us note that the number of nodes in each T_i will always remain proportional to the branching factor times the depth of the search in the current thread. Assuming a constant branching factor and that the maximum exploration depth is bounded by the size of the initial instance, the size of each task tree is linear. This is because the topology of the task tree is always a *caterpillar tree*. That is, each internal node of this tree has at most one child that is another internal node — the rest are leaf-children. To see this, it suffices to observe that only tasks of the search tree explored sequentially can have child tasks. Moreover, when a sequential call is finished, its corresponding task node is removed from the task tree. It follows that the internal nodes correspond to the path of exploration undertaken by the current thread.

Obtaining highest priority tasks To find and send the highest priority task, we need a dynamic load balancing (DLB) strategy that sends and removes the first leaf-child of the root of the task tree (the root itself is being explored by the current thread). After sending several tasks, it is possible to exhaust all the leaf-children of the root, in which case it has only one child, which is a task on the path currently explored by the current thread. In this case, the root is of no interest and it can be pruned. Its single child becomes the new root. When a leaf-child task to be sent is found, it is removed from the tree. At this point, it will become the root of the thread or process it is being sent to. Note that all tree operations can be done in time $O(1)$ with appropriate data structures.

```
function GemPBA::sendHighestPriorityTask(dest)
    $r$ = root of the $T_i$ task tree
    $done = False$
    while not done do
        if $r$ has no children then
            return “No task"
        else if $r$ has one child $q$ then
            delete $r$
            Reroot the tree to $q$, and let $r = q$
        else
            Let $\ell$ be a leaf-child of $r$ not marked as “Exploring”
            Remove $\ell$ from the current tree
            $send(\ell, dest)$
            $done = True$
    end
```

Algorithm 11: Finding a high priority task and updating the task tree

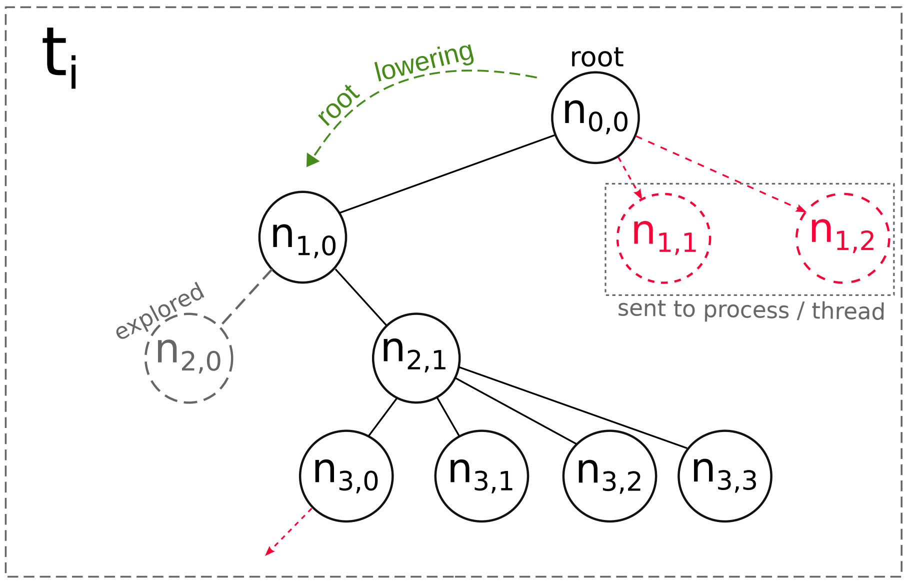

Figure 3.2 – Quasi-horizontal visualization, heterogeneous search tree. A node labeled $n_{d,j}$ indicates that its depth in the search tree is d, and that it is the j-th child of its parent. Nodes that are dotted have been removed from the tree, either because they were fully explored (gray), or sent to another process/thread (red).

As shown in Fig. 3.2, our dynamic load balancing strategy is very simplistic and is not limited to a specific search tree topology. This figure shows the search domain of a single thread t_i, where the shown root represents the original task received by the pool thread.

3.3.5 Startup phase

Even though our strategy is fully dynamic and allows any pair of processes to share tasks, it is beneficial to start the exploration as in the equitable strategy mentioned above. Since center can decide which processes exchange tasks, this can be achieved by populating the process-to-process assignments and waiting lists appropriately at startup. Our aim is that each search tree node at $depth = \log p$ to be handled by a distinct process. It is worth mentioning that in [2], the authors also adopted a strategy to achieve equitable startup, albeit in a different manner for their decentralized strategy.

```
function buildWaitingList($p_i$, $base\_d$, $b$, $p$)
    //$p_i$ = process index, $b$ = branching factor, $p$ = # processes
    //$base\_d$ is the depth of the highest search node assigned to $p_i$
    //Initial call is made with $p_i = 1$ and $base\_d = 0$
    for $d = base\_d..max\_depth$ do
        for $j = 1..b - 1$ do
            $q \leftarrow (j \times b^d) + p_i$
            if $q \leq p$ then
                Add $q$ to the waiting list of $p_i$
                buildWaitingList$(q, d + 1, b, p)$
        end
    end
```

Algorithm 12: Waiting list assignment algorithm.

Algorithm 12 populates waiting lists to achieve an assignment of search tree nodes as shown in Fig. 3.3, where the initial waiting list of a process p_i is populated based branching factor b. We assume that process p_i will always send its first $b - 1$ search tree tasks to its waiting list, proceed sequentially on the $b-$th task, and repeat the process as it goes deeper. The waiting list is built accordingly, using a parameter d that starts as the depth of the highest search task it will be assigned, and then increases until some maximum depth to mimic the sequential behavior of process p_i.

In this model, no process is aware of who it is assigned to, and all spawned tasks are sent to the processes in the waiting list in the order of assignment. That is, in Fig. 3.3, when sending tasks to other processes, p_1 will send them to $p2$, $p3$, $p4$ and lastly p_7 in that order. The search tree nodes, n_i in Fig. 3.3 are horizontally indexed for convenience, which is not directly to the process topology indexing. Note that waiting list items are purged after tasks are sent, and thus this initial assignment only holds at startup.

3.4 Implementation and experimental results

We have implemented our semi-centralized strategy in C++20, using boost [34] for task serialization and *openmpi* to accomplish inter-process communication. The code

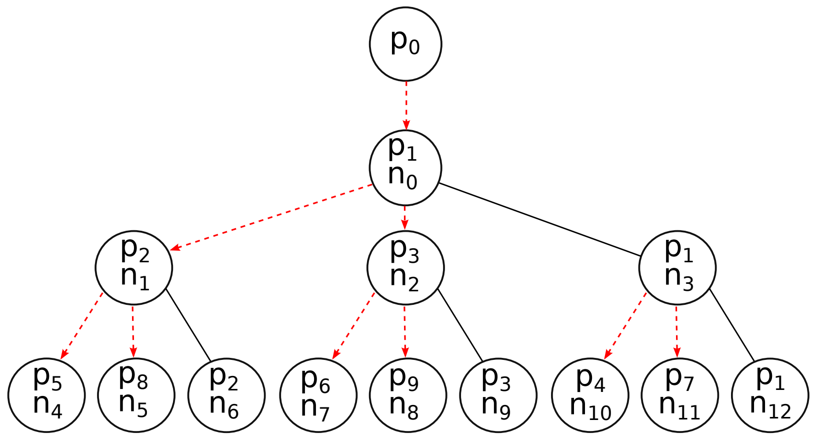

Figure 3.3 – Process topology, where red arrows refers to intended inter-process tasks passing.

is open-source and available at https://git.io/Jnx7k.

Using vertex cover as a case study, we first describe how to convert sequential code to parallel concretely, using our implementation. We then describe the performance results that we obtained on the DIMACS challenge graphs.

3.4.1 Converting sequential to parallel

First note that since processes run their own copy of a program, a multiprocess environment must be set up to ensure that only center process reads input data and writes the final solution. More details on the initial setup, which typically requires modifying *main.cpp*, can be found in 3.6.

Consider the sequential version of the vertex cover algorithm (we have commented the part not relevant to parallelism), shown in Fig. 3.4.

```
void mvc(Graph g, S s){
    if(g.size() == 0){
        if(s.size() < bestS.size()){ bestS = s;}
        else{ return; }
    }
    /* apply reduction rules, find u of maximum degree,
    g_l = g - u, g_r = g - N(u),
    s_l = s + u, s_r = s + N(u)*/
    mvc(g_l, s_l);
    mvc(g_r, s_r);
}
```

Figure 3.4 – Sequential MVC algorithm

In order to adapt this code to our framework, the function must include two additional parameters, which is the thread ID (*tid*) as the first one, and the Tree data structure (*parent*) as the last one, as seen in Fig. 3.5. Branch calls are delegated to *GemPBA* and our algorithm code is ready to go. In our implementation, adding sub-instances can be done by creating tree nodes with the given parent and letting them hold the sub-instances. The *try_push* function is analogous to an update, as it asks *gemPBA* to send tasks, possibly T_l, to waiting processes and threads. If T_l is not sent, it will be explored sequentially. The $forward$ function tells *gemPBA* to explore the given instance sequentially, without trying to send to an available worker. Lastly, the serializer and deserializer instances seen in Fig. 3.5 are necessary to pass instances via openmpi. They are lambda functions that follow a blueprint that GemPBA understands in which the user can implement their favourite serialization method.

```
using Tree = GemPBA::Tree<void, Graph, S>;
void mvc(int tid, Graph g, S s, void *parent){
    if(g.size() == 0){
        if(s.size() < gemPBA.bestValue()){
            gemPBA.holdSolution(s, serializer);
            gemPBA.updateBestValue(s.size());
        } else { return; }
    }
    /* apply reduction rules, find u of maximum degree,
        g_l = g - u, g_r = g - N(u),
        s_l = s + u, s_r = s + N(u)*/
    Tree T_l(dlb, tid, parent);  // for left branch
    T_l.holdInstance(g_l, s_l);
    Tree T_r(dlb, tid, parent);  // for right branch
    T_r.holdInstance(g_r, s_r);

    gemPBA.try_push<void>(mvc, tid, T_l, serializer);
    gemPBA.forward<void>(mvc, tid, T_r);
}
```

Figure 3.5 – Parallel MVC algorithm

Note that *try_push* sends to either a process or thread, which is handled by *GemPBA* giving priority to processes over threads by default. If only multithreading parallelism is of the user's interest, then a function called *try_pushMT* can be called to only send tasks to threads. This way, everything related to inter-process communication can be avoided and there is no need for MPI initialization or serialization.

3.4.2 Experiments

We now describe the performance of our approach on DIMACS challenge graphs.

Experimental setup

We report the results of our *mvc* implementation. Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium [1]. Niagara is a homogeneous cluster of initially 61,920 cores but expanded (in 2020) to 80,640 cores, owned by the University of Toronto and operated by SciNet. Designed for large parallel workloads, Niagara has a low-latency high-bandwidth Mellanox EDR InfiniBand interconnect in a *Dragonfly+* topology [45] with 4 wings and HDR100 InfiniBand for the fifth wing; each wing containing at most 468 nodes (i.e. 18720 cores).

We tested our framework with the Minimum-Vertex-Cover (MVC) problem, for which we used one of the most basic branching strategies as described in Section 3.2. We also implemented four reduction rules listed below, for further details we refer the reader to [15].

— **Rule 1:** Remove isolated vertices (i.e. of degree 0) from the graph.
— **Rule 2:** For each vertex u of degree 1, add its unique neighbor v to the solution and remove both u and v.
— **Rule 3:** For each vertex u with exactly two neighbors v and w such that vw is an edge, add v and w to the solution and remove u, v, w from the original graph.
— **Rule 4:** As long as there is a vertex u with exactly two neighbors v and w such that v and w are not adjacent to each other, replace u, v, w by u' and make u' adjacent to all neighbors of v and w. This is known as the folding rule. Note that u' cannot be replaced if it results from a previous folding. Once a solution is achieved, the cover S is scanned for the folded vertices; thus, if $u' \in S$ then v and w are present in the cover, otherwise u is in the cover.

Rules are applied iteratively until the graph does not change any more. Once the algorithm is running, *Rule [1-3]* are iteratively applied every time a branching (vertices removal) is executed. We used an adjacency matrix bitset implementation to

1. SciNet is funded by: the Canada Foundation for Innovation; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto [42, 35]

represent our graphs, which allowed for faster union and intersection computations for the reduction rules.

We adapted the aforementioned sequential MVC algorithm following the model in Algorithm 7, and we tested it on the following graphs.

— p_hat1000-2.clq: 1,000 vertices, 244,799 edges with a MVC of size 946.
— p_hat700-1.clq: 700 vertices, 60,999 edges with a MVC of size 635.
— frb30-15-1.mis: 450 vertices, 17,827 edges with a MVC of size 420.

The first two graphs were obtained from the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), these graphs are challenges utilized to benchmark algorithms. The third instance is one of the most challenging graphs for which the exact size of a solution was only known by theoretical predictions until it was first solved exactly by Abu-Khzam et al. [2] in 2015. For more details about the frb30-15-1.mis graph, please see [56].

Due to the system limitations, all our experiments were constrained to a maximum running time of 24 hours. Additionally, Niagara allows a maximum of 20 nodes allocation per user by default; we were granted permission to test up to 256 nodes (10,240 cores). Hyperthreading implementation was not part of the scope of this study and therefore only physical cores were taken into account for the actual calculations.

We compared the performance of each graph against the data reported by Abu-Khzam et al. [2], using the decentralized strategy, by matching the number of cores in common for each tests. All calculations were made on computing nodes with a blocking factor of 1:1, except for a special case on the *frb30_15_1* graph, which will be discussed later. Where a blocking factor is K: 1, if the maximum number of computers to share a bandwidth is K.

In order to optimize memory access, we spawned a process per cpu socket, and a thread per core within the socket, ie. on Niagara, each computing node has two cpu sockets, where each cpu has 20 physical cores. Thus, on Tables 3.1 to 3.3 the number of cores, $|c|$, involved in calculations is given by $nodes \times sockets\_per\_node \times cores\_per\_socket - 20$. It is worth mentioning that the column $|c|$ does not account for the center process, which uses only 1 core. This was a server constraint at the time of running experiments since an equal number of cores had to be assigned per process. Furthermore, the number of nodes reported on the tables were chosen according to the number of available nodes during the time of experiments. In consequence, shown speedup is relative to the preceding point where the expected ideal speedup (Exp.) is presented next to the actual speedup.

Results

In Figs. 3.6 and 3.7, we compare the ideal speedup (i.e. p process implies speedup $\times p$) against the speedup attained by our framework and the speedup reported by Abu-Khzam et al. [2] for the *p_hat1000-2 graph* and *p_hat700-1 graph* respectively. It can be seen how *GemPBA* is substantially achieving linear growth whilst accomplishing a performance nearly the ideal contrasted to the number of utilized cores. Note that Abu-Khzam reportedly implemented a different branching MVC algorithm. They use Chen et al.'s algorithm [16], which runs in time $O(kn + 1.28^k)$, where k is the vertex cover size, whereas our algorithm is much simpler but runs in time $O(2^k n)$. Also, they performed their experiments on different machines — hence the running times themselves are incomparable, but the speedup comparison should help situating the performance of our approach. In addition, we cannot exclude the possibility that our simpler algorithm benefits more from parallelism than theirs, since we evaluate more branching possibilities. Nevertheless, we emphasise that our simpler implementation was still able to solve difficult instances.

We can infer from Figs. 3.6 and 3.7 that the performance of the parallelization tool depends on the graph, meaning that some instances are more likely to benefit from massive parallelism. Notice that *p_hat1000-2* is an easier graph to solve than the *p_hat700-1* graph, even though they differ significantly in size, which results in search tree nodes exhausting faster and providing fewer eligible tasks to parallelize. Despite the uniqueness of these instances, *GemPBA* manages to make the most of parallelization.

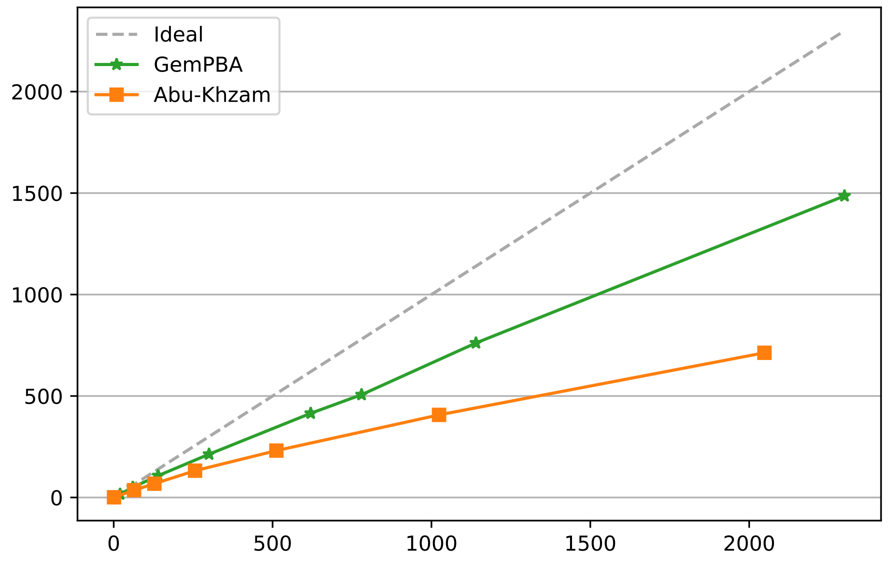

Figure 3.6 – Performance comparison on the *p_hat1000-2 graph*, number of cores (x-axis) vs speedup (y-axis).

We noticed that *GemPBA* is capable of achieving competitive results on challenging problems. For instance, in Fig. 3.8, the performance comparison between our approach vs the performance reported by Abu-Khzam et al. [2] is presented for the *frb30_15_1* graph. We would like to highlight that we performed two scenarios that are relevant when considering massive parallelization, which are strictly linked to the blocking factor that exists among computing nodes.

All our experiments showed linear speedup when considering a blocking factor (BF) of 1:1 on Niagara, however when this BF is greater than 1, this hardware limitation becomes a dominant variable by decreasing the growth rate of the performance, yet providing meaningful boost. On Fig. 3.8, the trending green line shows how *GemPBA* achieves super linear performance on the *frb30_15_1 graph*. This is possible since a great number of cores are simultaneously searching a solution, and as soon as a solution is found, it is reported and all branch explorations leading to

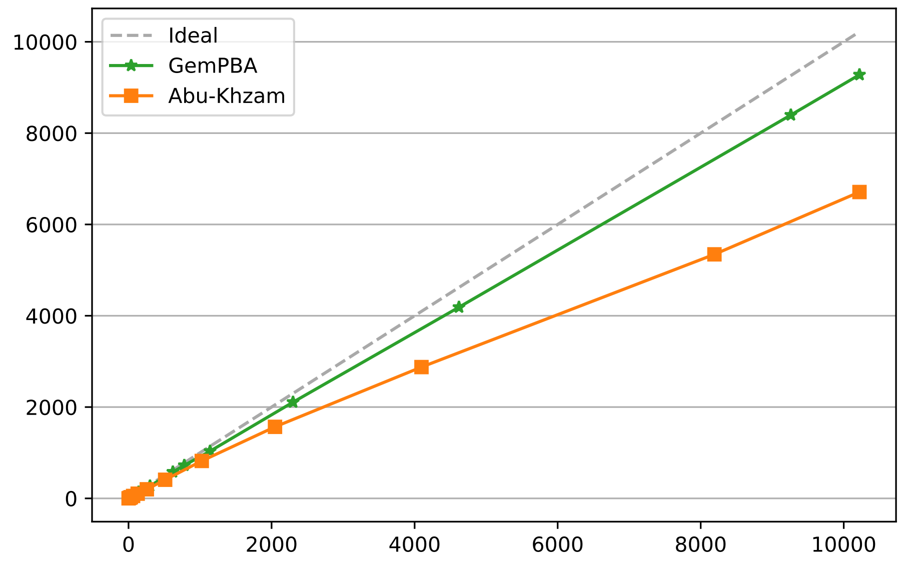

Figure 3.7 – Performance comparison on the *p_hat700-1 graph*, number of cores (x-axis) vs speedup (y-axis). *Last bullet point from Abu-Khzam is interpolated.*

a worse outcome are halted. Such branches might have been explored wastefully with fewer cores since the good solution might have taken more time to be found. The trending purple line diverges after 5100 cores, which is the last common point with the green line using BF 1:1. From this point onward, BF 2:1 started to play a significant role where super lineal speedup is not achieved any more, although our implementation still achieves remarkable performance.

Since it is impractical to launch a job solving these graphs in sequential mode, the results of our experiments and those ones reported by Abu-Khzam et al. [2] were used to project the running time of these graphs using a single core, by the means of a power regression. These projected values are reported on the Tables 3.1 to 3.3 on the first entry, marked with a star symbol ($*$).

Idle time is presented and it refers the total wall time that the cores were waiting for task assignments, which is considerably low for all the graphs compared to the total execution time, meaning that our *Dynamic Load Balancer* has positively impacted the performance.

On Table 3.3, the last entry, marked with the dagger symbol (†), shows a significant difference to the tendency of the preceding values because it is the scenario that considers a blocking factor of 2:1. The large amount of idle time of this point might be caused by side effects of communication overhead, that is, since messages take longer to be sent, our framework will address new tasks to local threads by default instead of other processes, causing that the other processes will receive more tasks from down the tree rather than the heaviest ones that have been probably taken by local threads, and since the algorithm is exponential, this also increases the total number of spawned tasks and therefore the execution time. Further to the Figs. 3.6 to 3.8, we see how our framework approaches and surpasses on certain cases the ideal speedup.

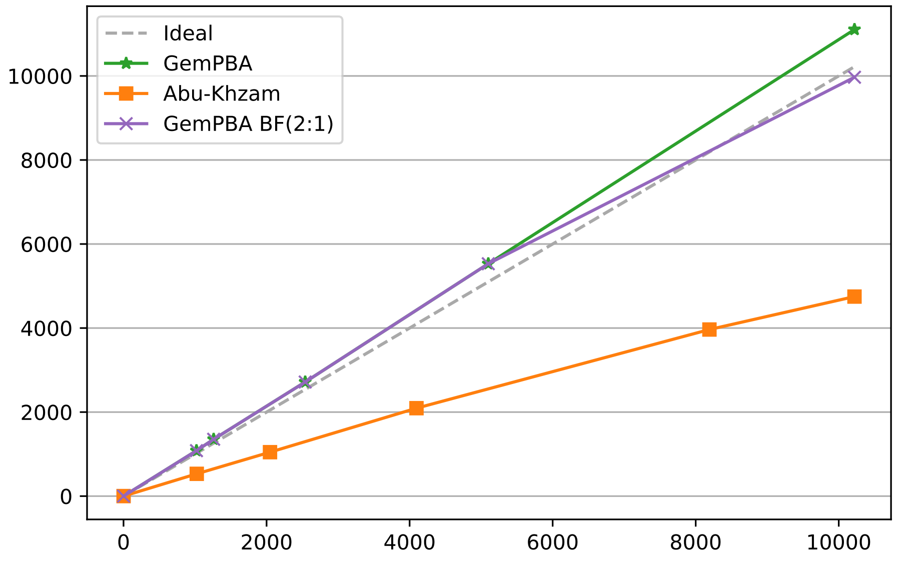

Figure 3.8 – Performance comparison on the *frb30_15_1 graph*, number of cores (x-axis) vs speedup (y-axis). *Last bullet point from Abu-Khzam is interpolated.*

nodes	p	$\|c\|$	Time	Req.	Idle time	Speedup	Exp.
		1	36.44 hr*				
1	2	20	2.277 hr	1	0.01 s		20.00
2	4	60	43.54 min	29,617	5.48 s	3.138	3.000
4	8	140	20.33 min	46,490	3.71 s	2.141	2.333
8	16	300	10.26 min	74,490	2.66 s	1.982	2.143
16	32	620	5.272 min	458,120	6.24 s	1.946	2.067
20	40	780	4.317 min	224,409	3.07 s	1.221	1.258
29	58	1140	2.872 min	328,012	3.09 s	1.503	1.462
58	116	2300	1.472 min	1,669,036	6.21 s	1.952	2.018

Table 3.1 – Performance attained for the *p_hat1000-2.clq* graph.

nodes	p	$\|c\|$	Time	Req.	Idle time	Speedup	Exp.
		1	26.54 day$*$				
2	4	60	11.25 hr	8,160	2.13 s		60.00
4	8	140	4.638 hr	24,407	2.31 s	2.425	2.333
8	16	300	2.355 hr	138,401	5.59 s	1.969	2.143
16	32	620	1.106 hr	120,850	2.30 s	2.130	2.067
20	40	780	53.11 min	400,797	5.51 s	1.249	1.258
29	58	1140	37.14 min	180,007	1.96 s	1.430	1.462
58	116	2300	18.23 min	1,007,782	4.48 s	2.038	2.018
116	232	4620	9.173 min	2,068,338	4.93 s	1.987	2.009
232	464	9260	4.574 min	1,635,128	3.76 s	2.005	2.004
256	512	10220	4.139 min	1,937,849	4.42 s	1.105	1.104

Table 3.2 – Performance attained for the *p_hat700-1.clq* graph.

nodes	p	$\|c\|$	Time	Req.	Idle time	Speedup	Exp.
		1	1.879 yr$*$				
26	52	1020	15.21 hr	135,687	1.78 s		1020
32	64	1260	12.16 hr	152,562	16.63 s	1.250	1.235
64	128	2540	6.061 hr	647,088	3.22 s	2.007	2.016
128	256	5100	2.976 hr	1,332,643	3.72 s	2.036	2.008
256	512	10220	1.482 hr	2,679,795	4.59 s	2.008	2.004
†256	512	10220	1.651 hr	22,954,998	7.63 min	2.008	2.004

Table 3.3 – Performance attained for the *frb30-15-1.mis* graph.

Preliminary data on the importance of choosing high priority nodes

As a preliminary analysis, in Table 3.4, we see how the *DLB*, i.e. always choosing the highest pending search tree task, impacts performance when multiprocessing or only multithreading is applied. We compared our task-choosing strategy against the greedy one. One can see that the discrepancy in the running times is significant. Furthermore, the idle time is a consequence of the number of requests, meaning that the threads/processes will be more often waiting for a task. The first two rows account for multiprocessing and multithreading (hybrid) implementation, whereas the last two rows account for only multithreading implementation.

case	n	p	$\|c\|$	Time	IPR	TR/IPR	T	IT
DLB	4	8	112	26.23 min	81.5 K	198	16.1 M	5.243 s
Greedy	4	8	112	38.54 min	37 K	9.91	366.7 M	15.66 min
DLB			32	1.63 hr			35.5 K	0.007 s
Greedy			32	1.92 hr			161.4 M	32.214 s

Table 3.4 – Significance of dynamic load balancing over the greedy approach, on the *p_hat1000-2.clq* graph. IPR is inter-process requests, TR is thread requests, $T = IPR \times TR$ is total requests and IT is idle time.

We note that a complete set of experiments on task-choosing strategies is beyond the scope of this paper. This will be the subject of future research.

3.5 Conclusion and future work

Our semi-centralized strategy strategy has demonstrated to be capable of maintaining CPU utilization close to 100% whilst decreasing the number of spawned tasks. To summarize the highlights of our strategy:

- Despite the presence of a center process, bottleneck are avoided by minimizing the requirements for communication.
- All task requests are guaranteed to be successful, therefore communication overhead is significantly decreased.
- The genericity of our framework allows easy migration of a serial algorithm to its parallel version in minutes with simple code modifications.
- Experimental results show how the number of generated tasks can be kept relatively low with our *DLB* strategy, which is critical to optimize parallelism.
- Super linear performance can be achieved in certain cases.
- *GemPBA* demonstrated to be scalable up to 10,240 physical cores, even including a blocking factor greater than 1. We hope to massively test our implementation with larger computing nodes and distinct blocking-factor scenarios to establish if it has some limitations.

There is absolutely a vast field to explore where we believe we could find significant improvements. We have contemplated an adaptive communication topology that can take advantage of the cluster's topology, i.e., launch a second level center process that will be in charge of the communication and task assignment of a wing on Niagara, thus there would be five second level centers plus and the absolute center, where five of the highest priority tasks would be sent to these wings and therefore a worker within a wing would need to communicate only with its wing neighbors, and thereby minimizing the side effects of the blocking factor.

3.6 Code modifications

To setup the inter-process environment, the main function is modified as in Fig. 3.9, in which three modules are initialized: GemPBA, DLB and the MpiScheduler which are in charge of branch handling, load balancing and the inter-process communication respectively; thus all processes are properly set up. Once this is done, the process ranked as $r = 0$, runs the center by passing the first task as a serialized buffer, and all other processes run the worker code.

```
int main(){
    /* instantiate: gemPBA, dlb, mpiScheduler
        graph, cover, and read input data
        buffer = serializer(graph, cover) */
    int rank = mpiScheduler.rank_me();
    gemPBA.setBestValue(graph.size());
    gemPBA.setBestValStrategyLookup("minimize");
    if (rank == 0){
        // runs MPI Scheduler for the center process
        mpiScheduler.runCenter(buffer);
    } else {
        // runs MPI Scheduler for the workers
        gemPBA.initThreadPool(numThreads);
        auto bufDecoder =
        gemPBA.bufDecoder<void, Graph, S>(mvc, deserializer);
        mpiScheduler.runWorker(gemPBA, bufDecoder, serializer);
    }
    mpiScheduler.barrier();
    if (rank == 0){
        S cover;
        stringstream ss;
        string buffer = mpiScheduler.fetchSolution();
        ss << buffer;
        deserializer(ss,cover);
        // output is stored already in "cover"
    }
}
```

Figure 3.9 – Multiprocess environment setup

3.7 Flowcharts

In this section we present a more detailed flowchart of our implementation for this study, where Fig. 10 shows the series of decisions made by the center process. Figs. 11 and 12 are the expansion of the description boxes when the center process receives *running* and *available* notifications as *tags*. In Fig. 13, we show the decisions by the worker process, p_i, which is in constant listening mode from any other process and start working when it receives a message containing a task. The series of decisions made when it is doing its job, is expanded in Fig. 14, where the main thread t_0 of this process is in charge of the inter-process communication. Not shown here is the pool threads, $t_i, \dots, t_{|c|-1}$, which receive the initial task from the main thread, then it is partitioned into other tasks and sent accordingly to waiting processes or waiting threads availability, prioritizing processes by default.

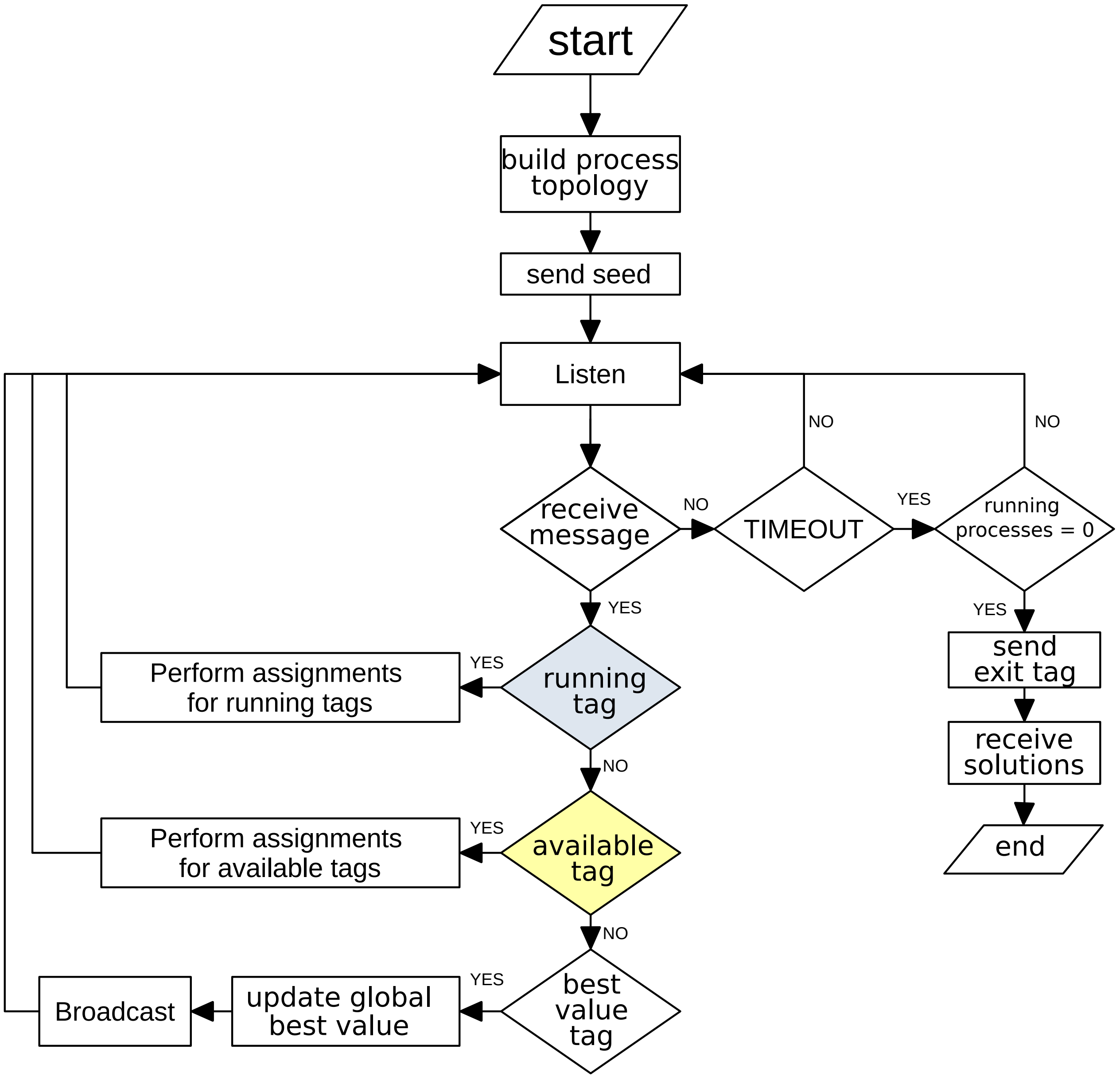

Figure 10 – Center process flowchart

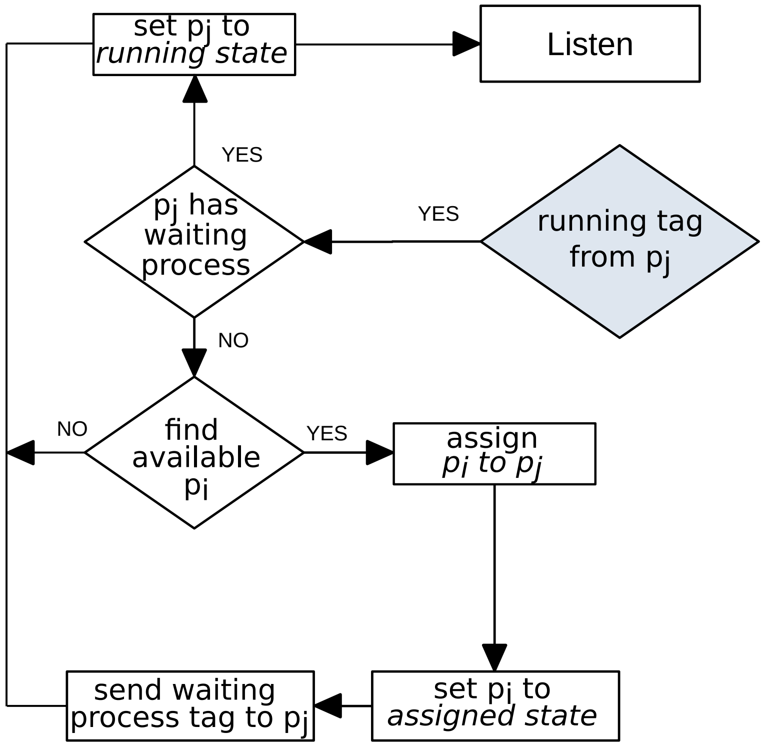

Figure 11 – Perform assignments for running tags.

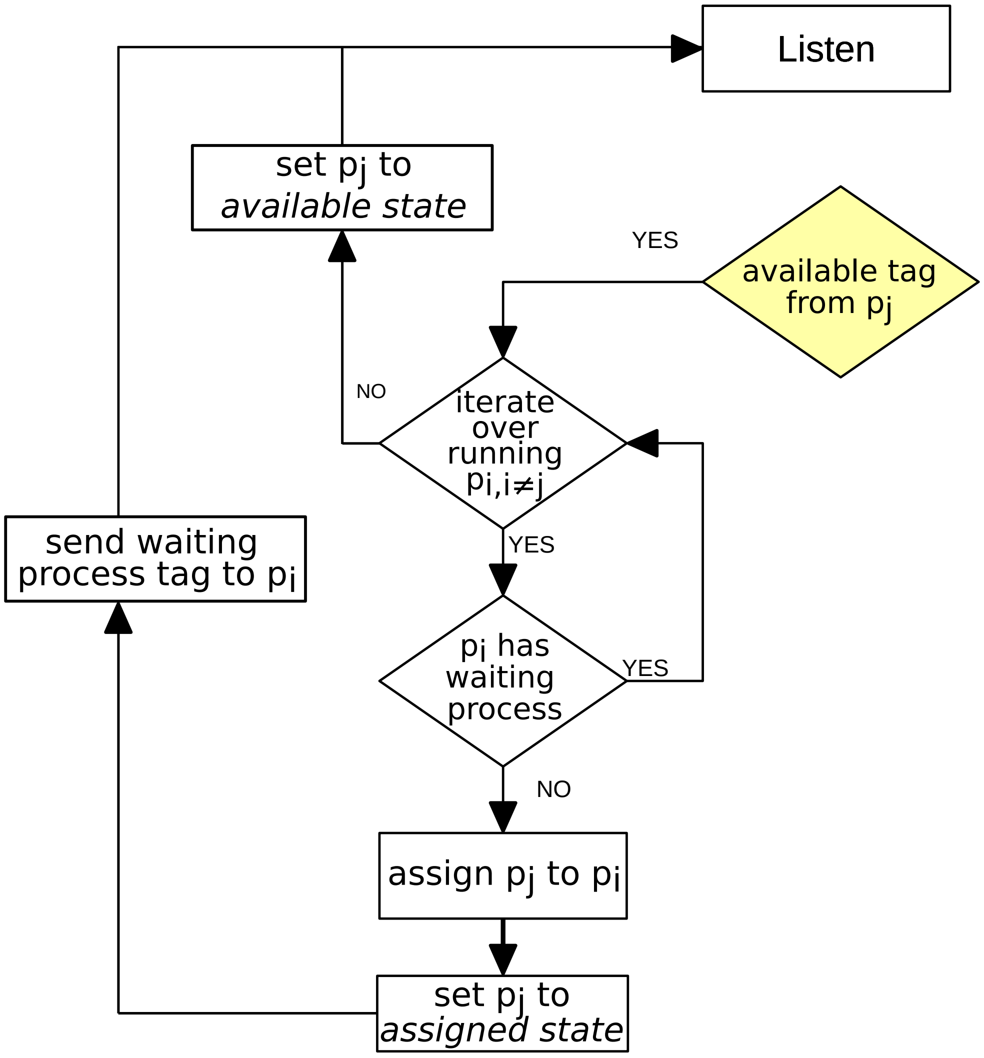
Listen
set p_j to *available state*
YES
available tag from p_j
NO
iterate over running $p_{i,i\neq j}$
YES
send waiting process tag to p_i
p_i has waiting process
YES
NO
assign p_j to p_i
set p_j to *assigned state*

Figure 12 – Perform assignments for available tags.

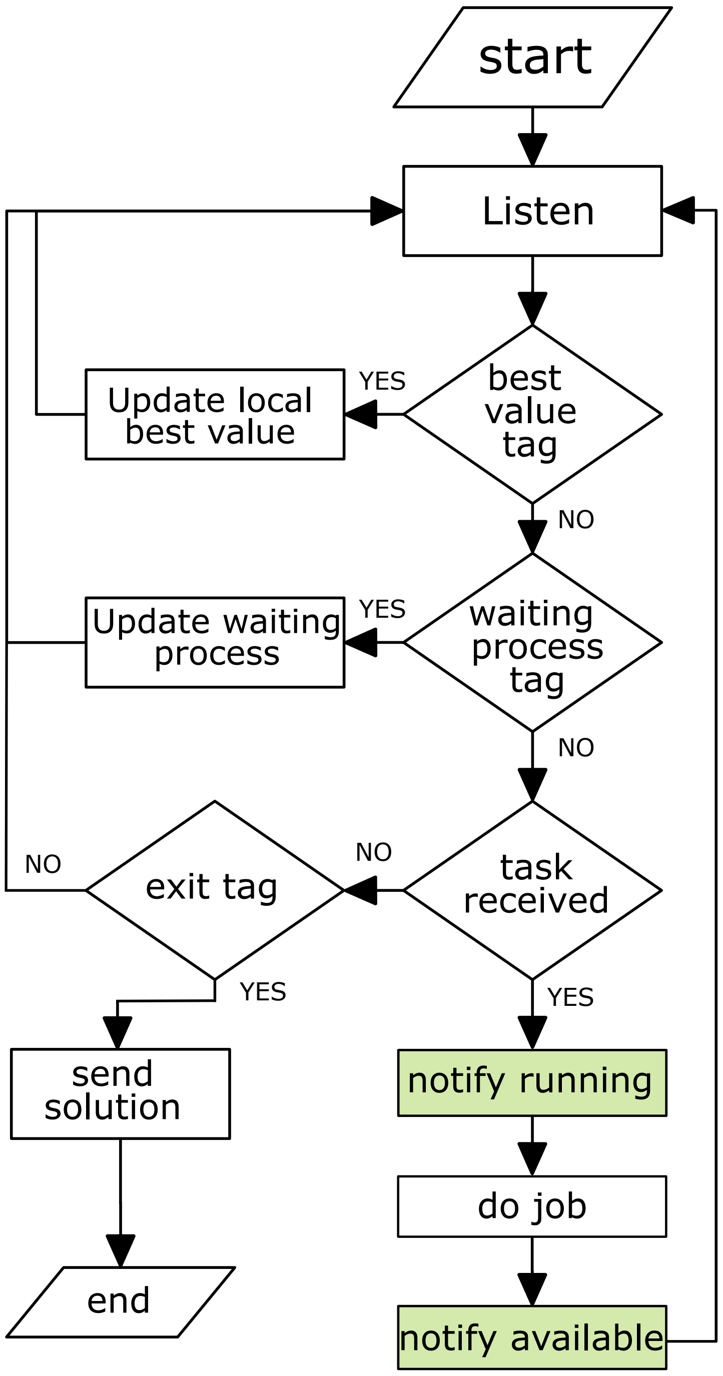

Figure 13 – Worker process

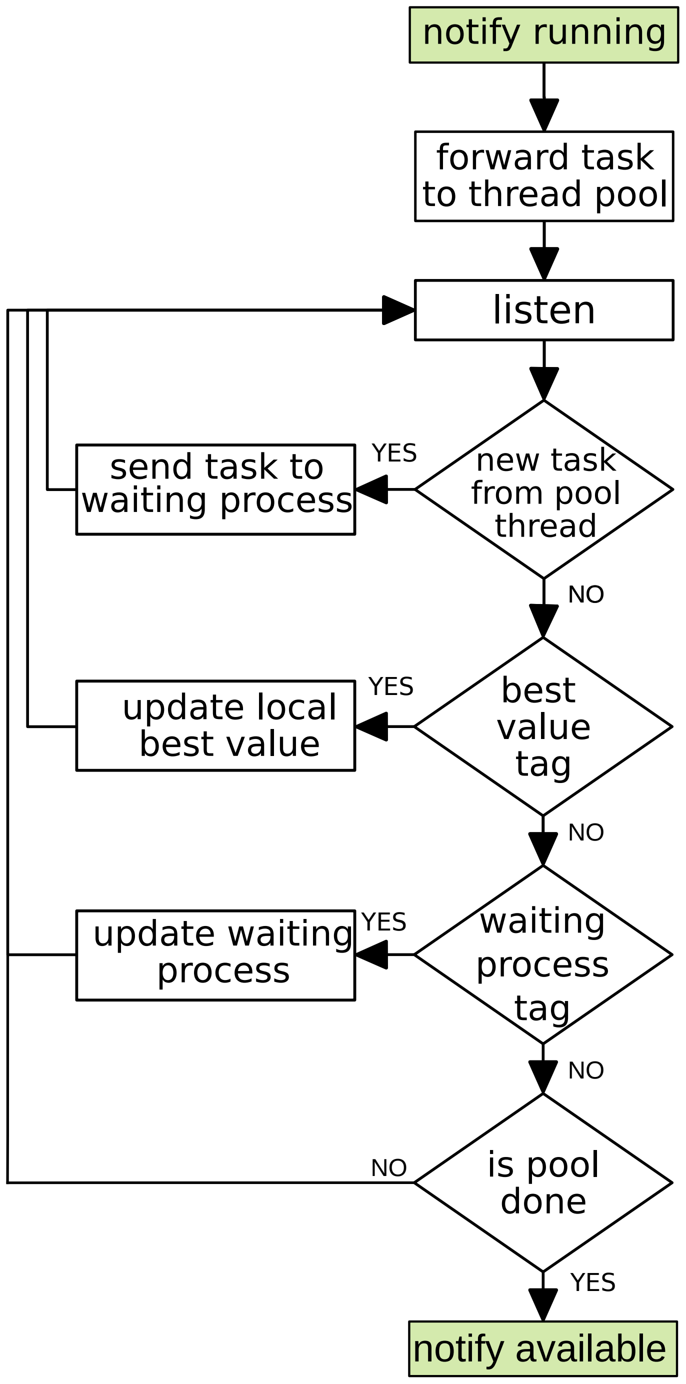

Figure 14 – Worker process, do job details

Chapter 4

Challenges

In this chapter, we address the main challenges that were not discussed in the paper presented in Chapter 3. We discuss the procedures implemented to achieve proper inter-process communication in the context of this research study.

4.1 OpenMPI

We used *openmpi* to handle inter-process communication due to its long-term support and availability on most of the servers. We acknowledge the existence of other capable tools such as UPC++ [9], which has accomplished in some cases better performance than *MPI*. Our implementation is so lightweight and simplistic that migrating from *openmpi* to *UPC++* should be easy, which should be subject to discussion in further research studies.

In *MPI*, there exist four environments to include threads participation, as follows.

— MPI_THREAD_SINGLE meaning that the application is single-threaded and no other thread is spawned within the process domain.

— MPI_THREAD_FUNNELED meaning that the application may be multi-threaded, however only the thread that initialized the MPI environment is allowed to perform MPI calls.

- MPI_THREAD_SERIALIZED meaning that the application may be multi-threaded, yet only one thread can make MPI calls at a time.
- MPI_THREAD_MULTIPLE meaning that the application may be multi-threaded and any thread can make MPI calls. However, this is still experimental for which special care should be considered.

If threads conditions are not applied with the proper environment, it might result in unknown behavior.

4.1.1 Two-sided communication

Inter-process communication is particularly different than multithreading since processes are fully independent by default. In consequence, when a process sends a message to another process, a proper communication channel must be established among the participating workers. This channel is well managed by the *mpi communicators*, which synchronizes and isolates messages, guaranteeing message delivery from the source to the destination, as long as it is properly set up by the user[44].

Therefore, when a message is to be transmitted between two workers, the sender and the receiver must synchronize such that a buffer can be successfully delivered. This type of communication can be achieved either synchronously (blocking) or asynchronously (non-blocking). Asynchronous transmission is not an issue when the current instruction of a running process does not depend on the incoming message. However, even though asynchronous messages are guaranteed to be delivered, the order in which a message is delivered is not, since multiple processes typically share the same communicator. Then, a user must synchronize at some point the incoming message to be certain that it has successfully been received, yet other techniques can be applied to verify that a message was delivered.

Hence, if a message is not immediately required, one should implement non-blocking messaging to avoid the disruption of a task. i.e. when updating the *bestValSoFar* in our implementation, it is safe to allow asynchronous messaging since in the worst case, information would be a few cycles delayed, which would cause to discard a solution that has been found somewhere else. This is not the case when receiving a task, because a process cannot attempt to solve a task that has not been fully received. Most of the time, messages can be sent/received in a non-blocking fashion, though it will depend on the application.

One of the major disadvantages of this technique is that when a process sends a message, the destination process should also participate in the receiving. This could add some complexity to the implementation since the user should ascertain a way to revisit a inter-process communication scope while executing local tasks, or designating a thread only for MPI communication.

4.1.2 One-sided communication

In order to release responsibilities to receiving nodes, there exists the *one-sided communication* model, which supports three synchronization models: Fence, Post-start-complete and passive target. The passive target synchronization is of our interest since it does not require the target to participate in the communication. It consists of allocating a shared memory, which is accessible from any process belonging to the same communicator domain. This shared memory is emulated by the usage of a window, which is essentially an array in which each entry corresponds to the rank id of a process. Each entry points to the remote memory of the process, this memory can be either a single value or another array[44].

This approach is very convenient since a process p_i can read or write data on the domain of a process p_j with $i \neq j$, or any other process part of the window and communicator. Nevertheless, this approach is not yet supported by MPI_THREAD_-MULTIPLE environment, which makes it unsuitable for most multithreaded applications.

One-sided communication has demonstrated to improve performance at large scales [44], however, servers must be properly configured otherwise communication may not be established when attempting passive synchronization.

4.1.3 Implementation

GemPBA was initially developed implementing *one-sided communication*, which demonstrated high performance on Niagara supercomputer, though *infinite waits* were appearing on other servers, even with a minimal working example. This infinite waits consisted of a process p_A trying to write on process p_B but never attaining communication, which were related to the servers configuration. Since our goal is to create a fully generic tool, we opted for migrating this approach to *two-sided communication* by dedicating a single thread for the inter-process communication for which the message passing never failed.

We chose the MPI_THREAD_FUNNELED environment since it is the most convenient approach based on our objectives. Since it isolates the inter-process communication from the multithreading domain and minimizes the *MPI* calls.

It is worth mentioning that, when massive parallelism takes place, there will be an enormous number of processors communicating with each other, in which, more *MPI* calls, make the application prone to errors due to the difficulty of synchronizing threads and processes simultaneously. These errors are easy to track with MPI_THREAD_FUNNELED by localizing a process per computing node, where each process will handle its own threads separately such that only one thread is responsible for inter-process communication, consequently reducing the participating processors in remote communication.

Hence, errors were easily addressed and fixed, which for further studies and extension of the *GemPBA* framework, would be easier to adapt.

Conclusion

With the scientific interests for parallel implementations, non-trivial parallelizable tasks such as branching algorithms are now feasible whilst avoiding a long learning curve in parallelism. Fields like bioinformatics implement complex recursive algorithms, which are constantly evolving to become more efficient, yet certain problems that are impractical to solve sequentially have now become feasible with a few lines of code modifications. Thanks to our generic framework, a user who has no experience in parallelism can now parallelize almost any of his implementations, delegating the hard synchronization to our framework.

Although our implementation was only evaluated on non-waiting algorithms, it is possible to use our implementation on waiting algorithms, which would be the subject of discussion in future studies.

A flexible *dynamic load balancing* was successfully developed, regardless of the search tree topology. This *DLB* showed to impact significantly on the final performance due to the spawning tasks control it offers.

GemPBA demonstrated to be capable of solving the most challenging graphs from DIMACS. The performance shown was more than competitive but outstanding, considering that we implemented one of the most basic algorithms for the *MVC* problem.

Our study showed linear scalability of up to 10,240 physical cores, which is equivalent to 256 computing nodes of the supercomputer cluster Niagara. Even though a blocking factor of 2:1 was considered for a specific scenario, it has demonstrated to attain almost the ideal speedup. Further studies would be required to assess scenarios involving distinct blocking factors, for which we would like to mitigate this communication overhead by the means of constraining tasks passing, such that only the most likely heavy task is sent to another node island and then the majority of process-to-process communication would take place within blocking factors of 1:1.

Bibliography

[1] NP-Completeness, pages 359–392. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-71844-4_15.

[2] Faisal N Abu-Khzam, Khuzaima Daudjee, Amer E Mouawad, and Naomi Nishimura. On scalable parallel recursive backtracking. Journal of Parallel and Distributed Computing, 84:65–75, 2015.

[3] Faisal N Abu-Khzam, DoKyung Kim, Matthew Perry, Kai Wang, and Peter Shaw. Accelerating vertex cover optimization on a gpu architecture. In 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 616–625. IEEE, 2018.

[4] Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable parallel algorithms for fpt problems. Algorithmica, 45(3):269–284, Jul 2006. doi:10.1007/s00453-006-1214-1.

[5] Faisal N Abu-Khzam, Mohamad A Rizk, Deema A Abdallah, and Nagiza F Samatova. The buffered work-pool approach for search-tree based optimization algorithms. In International Conference on Parallel Processing and Applied Mathematics, pages 170–179. Springer, 2007.

[6] Faisal N. Abu-Khzam, Mohamad A. Rizk, Deema A. Abdallah, and Nagiza F. Samatova. The buffered work-pool approach for search-tree based optimization algorithms. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, Parallel Processing and Applied Mathematics, pages 170–179, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] 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), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. doi:10.1145/1465482.1465560.

[8] W.C. Athas and C.L. Seitz. Multicomputers: message-passing concurrent computers. Computer, 21(8):9–24, 1988. doi:10.1109/2.73.

[9] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. Upc++: A high-performance communication framework for asynchronous computation. In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 963–973, 2019. doi:10.1109/IPDPS.2019.00104.

[10] Max Bannach, Malte Skambath, and Till Tantau. Towards work-efficient parallel parameterized algorithms. In International Workshop on Algorithms and Computation, pages 341–353. Springer, 2019.

[11] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W Klau. Exact algorithms for cluster editing: Evaluation and experiments. Algorithmica, 60(2):316–334, 2011.

[12] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. IEEE Transactions on Software Engineering, SE-5(4):341–349, 1979. doi:10.1109/TSE.1979.234201.

[13] Laurent Bulteau and Mathias Weller. Parameterized algorithms in bioinformatics: an overview. Algorithms, 12(12):256, 2019.

[14] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J Taillon. Solving large fpt problems on coarse-grained parallel machines. Journal of Computer and System Sciences, 67(4):691–706, 2003.

[15] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. Journal of Algorithms, 41(2):280–301, 2001. URL: https://www.sciencedirect.com/science/article/pii/S0196677401911861, doi:https://doi.org/10.1006/jagm.2001.1186.

[16] Jianer Chen, Iyad A Kanj, and Ge Xia. Improved upper bounds for vertex cover. Theoretical Computer Science, 411(40-42):3736–3756, 2010.

[17] Jens Clausen. Branch and bound algorithms-principles and examples. Department of Computer Science, University of Copenhagen, pages 1–30, 1999.

[18] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. Parameterized Algorithms. Springer Publishing Company, Incorporated, 1st edition, 2015.

[19] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Parameterized algorithms, volume 5. Springer, 2015.

[20] Rodney G Downey and Michael R Fellows. Fundamentals of parameterized complexity, volume 4. Springer, 2013.

[21] Jörg Flum and Martin Grohe. Parameterized complexity theory. Springer Science & Business Media, 2006.

[22] Fedor V Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. Journal of the ACM (JACM), 56(5):1–32, 2009.

[23] Fedor V Fomin and Petteri Kaski. Exact exponential algorithms. Communications of the ACM, 56(3):80–88, 2013.

[24] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. In Italian Conference on Algorithms and Complexity, pages 108–119. Springer, 2003.

[25] William Gropp and Rajeev Thakur. Issues in developing a thread-safe mpi implementation. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 12–21, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[26] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint 1st Edition. Morgan Kaufmann, Reading, Massachusetts, 2012.

[27] Falk Hüffner, Christian Komusiewicz, Rolf Niedermeier, and Sebastian Wernicke. Parameterized Algorithmics for Finding Exact Solutions of NP-Hard Biological Problems, pages 363–402. Springer New York, New York, NY, 2017. doi: 10.1007/978-1-4939-6613-4_20.

[28] Trey Ideker and Roded Sharan. Protein networks in disease. Genome research, 18(4):644–652, 2008.

[29] J Stephen Judd. Neural network design and the complexity of learning. MIT press, 1990.

[30] Laxmikant V Kalé, Balkrishna Ramkumar, V Saletore, and AB Sinha. Prioritization in parallel symbolic computing. In US/Japan Workshop on Parallel Symbolic Computing, pages 11–41. Springer, 1992.

[31] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. J. ACM, 40(3):765–789, July 1993. doi:10.1145/174130.174145.

[32] Jyrki Katajainen and Jesper Larsson Trgff. A meticulous analysis of mergesort programs. In in Proceedings of the 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science 1203, Springer-Verlag, pages 217–228, 1997.

[33] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 137–146, 2003.

[34] Christopher Kormanyos. Real-Time C : Efficient Object-Oriented and Template Microcontroller Programming. Springer Berlin Heidelberg, 2018.

[35] Chris Loken, Daniel Gruner, Leslie Groer, Richard Peltier, Neil Bunn, Michael Craig, Teresa Henriques, Jillian Dempsey, Ching-Hsing Yu, Joseph Chen, L Jonathan Dursi, Jason Chong, Scott Northrup, Jaime Pinto, Neil Knecht, and Ramses Van Zon. SciNet: Lessons learned from building a power-efficient top-20 system and data centre. Journal of Physics: Conference Series, 256:012026, nov 2010. doi:10.1088/1742-6596/256/1/012026.

[36] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization, 19:79–102, 2016. URL: https://www.sciencedirect.com/science/article/pii/

S1572528616000062, doi:https://doi.org/10.1016/j.disopt.2016.01.005.

[37] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. Discrete Optimization, 19:79–102, 2016.

[38] Rolf Niedermeier. Invitation to fixed-parameter algorithms. 2006.

[39] Rob Oshana. Chapter 1 - principles of parallel computing. In Rob Oshana, editor, Multicore Software Development Techniques, pages 1–30. Newnes, Oxford, 2016. URL: https://www.sciencedirect.com/science/article/pii/B9780128009581000012, doi:https://doi.org/10.1016/B978-0-12-800958-1.00001-2.

[40] Rob Oshana. Chapter 2 - parallelism in all of its forms. In Rob Oshana, editor, Multicore Software Development Techniques, pages 31–38. Newnes, Oxford, 2016. URL: https://www.sciencedirect.com/science/article/pii/B9780128009581000024, doi:https://doi.org/10.1016/B978-0-12-800958-1.00002-4.

[41] Rob Oshana. Chapter 4 - multicore software architectures. In Rob Oshana, editor, Multicore Software Development Techniques, pages 53–66. Newnes, Oxford, 2016. URL: https://www.sciencedirect.com/science/article/pii/B9780128009581000048, doi:https://doi.org/10.1016/B978-0-12-800958-1.00004-8.

[42] Marcelo Ponce, Ramses van Zon, Scott Northrup, Daniel Gruner, Joseph Chen, Fatih Ertinaz, Alexey Fedoseev, Leslie Groer, Fei Mao, Bruno C. Mundim, Mike Nolta, Jaime Pinto, Marco Saldarriaga, Vladimir Slavnic, Erik Spence, Ching-Hsing Yu, and W. Richard Peltier. Deploying a top-100 supercomputer for large parallel workloads: The niagara supercomputer. In Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3332186.3332195.

[43] Vikram A Saletore and Laxmikant V Kale. Consistent linear speedups to a first solution in parallel state-space search. In AAAI, pages 227–233, 1990.

[44] Danish Shehzad and Zeki Bozkuş. Optimizing neuron simulation environment using remote memory access with recursive doubling on distributed memory systems. Computational Intelligence and Neuroscience, 2016:3676582, Jun 2016. doi:10.1155/2016/3676582.

[45] Alexander Shpiner, Zachy Haramaty, Saar Eliad, Vladimir Zdornov, Barak Gafni, and Eitan Zahavi. Dragonfly+: Low cost topology for scaling datacenters. In 2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), pages 1–8, 2017. doi:10.1109/HiPINEB.2017.11.

[46] W. Shu and L. V. Kale. A dynamic scheduling strategy for the chare-kernel system. In Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing '89, page 389–398, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/76263.76306.

[47] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. Operating System Concepts with Java. Wiley Publishing, 8th edition, 2009.

[48] Amitabh B Sinha and Laxmikant V Kalé. A load balancing strategy for prioritized execution of tasks. In [1993] Proceedings Seventh International Parallel Processing Symposium, pages 230–237. IEEE, 1993.

[49] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V Kalé. An adaptive framework for large-scale state space search. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pages 1798–1805. IEEE, 2011.

[50] YANHUA SUN, GENGBIN ZHENG, PRITISH JETLEY, and LAXMIKANT V. KALE. Parsse: An adaptive parallel state space search engine. Parallel Processing Letters, 21(03):319–338, 2011. arXiv:https://doi.org/10.1142/S0129626411000242, doi:10.1142/S0129626411000242.

[51] Luzhi Wang, Shuli Hu, Mingyang Li, and Junping Zhou. An exact algorithm for minimum vertex cover problem. Mathematics, 7(7):603, 2019.

[52] Dinesh P. Weerapurage, John D. Eblen, Gary L. Rogers, and Michael A. Langston. Parallel vertex cover: A case study in dynamic load balancing. In Proceedings of the Ninth Australasian Symposium on Parallel and Distributed

Computing - Volume 118, AusPDC '11, page 25–32, AUS, 2011. Australian Computer Society, Inc.

[53] Barry Wilkinson and C. Michael Allen. Parallel programming. Pearson/Prentice Hall, 2 edition, 2005.

[54] Tobias Wittkop, Jan Baumbach, Francisco P Lobo, and Sven Rahmann. Large scale clustering of protein sequences with force-a layout based heuristic for weighted cluster editing. BMC bioinformatics, 8(1):1–12, 2007.

[55] Gerhard J Woeginger. Exact algorithms for np-hard problems: A survey. In Combinatorial optimization—eureka, you shrink!, pages 185–207. Springer, 2003.

[56] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence, 171(8):514–534, 2007. URL: https://www.sciencedirect.com/science/article/pii/S0004370207000653, doi:https://doi.org/10.1016/j.artint.2007.04.001.

[57] Sinem Çınaroğlu and Sema Bodur. A new hybrid approach based on genetic algorithm for minimum vertex cover. In 2018 Innovations in Intelligent Systems and Applications (INISTA), pages 1–5, 2018. doi:10.1109/INISTA.2018.8466307.