Parallel Branch-and-Bound in Multi-core Multi-CPU Multi-GPU Heterogeneous Environments

Trong-Tuan Vu, Bilel Derbel

Parallel Branch-and-Bound in Multi-core Multi-CPU Multi-GPU Heterogeneous Environments

Trong-Tuan Vu*

INRIA Lille Nord Europe, France

Bilel Derbel†

INRIA, LIFL CNRS UMR-8022, Université Lille 1, France

Abstract

We investigate the design of parallel B&B in large scale heterogeneous compute environments where processing units can be composed of a mixture of multiple shared memory cores, multiple distributed CPUs and multiple GPUs devices. We describe two approaches addressing the critical issue of how to map B&B workload with the different levels of parallelism exposed by the target compute platform. We also contribute a throughout large scale experimental study which allows us to derive a comprehensive and fair analysis of the proposed approaches under different system configurations using up to 16 GPUs and up to 512 distributed cores. Our results shed more light on the main challenges one has to face when tackling B&B algorithms while describing efficient techniques to address them. In particular, we are able to obtain linear speed-ups at moderate scales where adaptive load balancing among the heterogeneous compute resources is shown to have a significant impact on performance. At the largest scales, intra-node parallelism and hybrid decentralized load balancing is shown to have a crucial importance in order to alleviate locking issues among shared memory threads and to scale the distributed resources while optimizing communication costs and minimizing idle times.

Keywords. Parallel B&B, large scale heterogeneous distributed systems, multi-core multi-CPU multi-GPU systems.

1 Introduction

1.1 Context and motivation

Branch and Bound (B&B) is a universal search technique used for solving to optimality NP-hard optimization problems appearing in a large number of computational fields, e.g., logistics, telecommunication, transportation, artificial life, etc. In B&B, the search space can be represented as a tree whose root node is the original unsolved problem and the internal nodes are partially solved subproblems. These subproblems are obtained by decomposing their parents in the tree and using branching and pruning operators. Roughly speaking, branching enables to divide a given subproblem into two or more smaller subproblems which shall be processed recursively in the next iterations. Pruning helps to reduce the size of the exploration space by eliminating the tree branches corresponding to subproblems that surely cannot lead to the best global solution. The main idea of B&B is to recursively identify the subspaces that do not contain the global optimal solution by evaluating an approximated bound for the induced subspaces and discarding the non-promising ones from further exploration.

At a first glance, B&B can be seen as a divide and conqueror algorithm which is straightforward to parallelize — for instance, by exploring each of the generated subproblems (subtrees) independently in

*Electronic address: vttuan84@gmail.com †Electronic address: bilel.derbel@lifl.fr

parallel. However, achieving high performance and scalability in parallel B&B is a difficult issue as witnessed by the amount of literature on the subject during the last decades, see e.g., [24, 27]. Although the sources of parallelism in B&B are rather well identified, the corresponding computations are known to be highly irregular, and the induced workload has an unpredictable and dynamic nature. As a consequence, B&B is a skillful adversary application which pauses several challenges both at the algorithmic and at the system level when attempting to design efficient and effective parallelization for it. In particular, exploring the full power of a large scale heterogeneous compute system and keeping all available processing units busy doing useful works is a major concern.

1.2 Contribution overview

In this paper, we target parallel B&B on large scale heterogeneous distributed platforms harnessing several distributed CPUs and GPUs (Graphical Processing Units). For such compute environments, the heterogeneity and the incompatibility of resources, in terms of compute power or programming models, is a major obstacle that can inherently lead to a substantial gap between the actual and the optimal theoretically attainable performance. For instance, GPUs are well suited for data parallelism and not all B&B operations can be implemented efficiently using a GPU. However, many GPU threads running in parallel can be extremely efficient compared to performing the same computations using CPUs. Moreover, coordinating parallel computations on large scale distributed CPUs comes a communication price that should be minimized as much as possible especially when considering fine-grained parallel applications. Specifically, parallel work should be made available very quickly everywhere in the distributed system in order to keep processing units working and to avoid idle times. In this context, we investigate the design of parallel B&B to maximally utilize the available parallelism on multi-core-CPU-GPU heterogeneous architectures. Our main contribution can be summarized as follows:

- We give a comprehensive description of how to map B&B workload with the different levels of parallelism exposed by the target compute platform. We argue that this is one critical issue when designing parallel B&B for large scale distributed systems. To illustrate this claim, we provide a first approach called 2MBB where the hierarchy exposed by the system is not fully explored. The 2MBB approach is two-fold. First, it allows us to obtain near-linear speed-ups under some configurations which is challenging *per se*. Second, the relatively good performance obtained with the 2MBB approach allows us to point out in a fair and comprehensive manner the main challenges one should face in parallel B&B and the distributed settings where complex convoluted distributed technicalities can be avoided. Actually, the shared memory nature of some of compute nodes is not addressed by the 2MBB approach which can lead to performance loss at the largest scales. This issue is addressed in our second approach, called 3MBB, where a hybrid dynamic workload protocol and a specific data structure to encode B&B work pools, are designed. The 3MBB explores the multi-level parallelism of the compute system, by specifically taking into account the shared memory level (multi-core) and the distributed memory level (multi-CPU and multi-GPU).

- We conduct a throughout experimental study of the proposed approaches for the standard Flowshop problem and we give a detailed analysis under different system configurations. More precisely, we experiment our approaches for a single host-device GPU, as well as for multi-GPUs, multi-CPU multi-GPUs, and multi-core multi-CPU multi-GPUs distributed systems, while considering up to 16 GPUs and 512 cores. For every setting, we report the relative performance with respect to the linear speed-up considered as the ultimate desirable performance. In particular, we are able to show that substantial gains are obtained with the 3MBB approach at the largest scales whereas the 2MBB approach is only able to perform well at the low and relatively moderate scales. In fact, our analysis shows that inter-node parallelism is sufficient to attain good performance when carefully balancing the workload at runtime. However, intra-node parallelism involving multi-core shared memory computations have a significant impact when the application workload is very fine-grained which is likely to be the case for large scale heterogeneous settings.

To gain in generality, we note that this paper does not focus on operations specific to the optimization problem being solved with B&B neither on their specific implementation on a target device or architecture.

To our best, a unique of this work is to provide a complete description, experiments and analysis of the different design and implementation issues one has to face when dealing with B&B on a multi-core multi-CPU multi-GPU heterogeneous platform. The rest of this paper is organized as follows.

1.3 Outline

In Section 2, we sketch the general template of a serial B&B algorithm and we discuss related work dealing with its parallelization in different distributed settings and hardware configurations. In Section 3, we summarize the main B&B challenges addressed by our proposed approaches and give a sketch on how we shall tackle them. In Section 4, we provide a comprehensive description of our first distributed approach (2MBB) for parallel B&B on multi-CPU multi-GPU compute setting. In Section 5, we extend on the 2MBB approach by additionally considering shared memory multi-cores. We thereby describe our second architecture (3MBB) and discuss its main design choices and implementation details. In Section 6, we report and discuss our experimental results. In Section 7, we conclude the paper and raise some open issues.

2 Background and related work

2.1 Serial B&B in a nutshell

As sketched in the introduction, Branch-and-Bound is a universal search algorithm [35] that can be used to find the optimal solution(s) with respect to a given optimization problem. The general idea of B&B is to represent the search space as a tree, where the root of the tree represents the whole search space, and the intermediate nodes represent smaller sub-problems where typically the range of few variables has been restricted. Generally speaking, a sequential B&B algorithm uses four operators as illustrated in the high level template of Algorithm 1. The algorithm maintains a list of subproblems which constitutes the nodes of the search tree. At each iteration, one specific subproblem, that is one intermediate node in the search tree, is selected to be processed. Processing a subproblem first consists in computing its cost. If the subproblem is a leaf in the tree, then this means that all the variables of the subproblem have been determined, and a complete solution is obtained. In this case, the quality of the solution is evaluated using the cost function of the original problem to be optimized. The newly computed solution is retained if its quality is better than the best one found so far in previous iterations. Otherwise, the cost of the subproblem is computed using a problem-dependent method. Without loss of generality, this corresponds to the computation of a lower bound in case minimization is considered. If the computed lower-bound is worst than the quality of the best solution found so far (i.e., the best known upper bound), the subproblem and all its potential descendants are discarded and not expanded further in the tree. This is known as the pruning phase of B&B. If the lower-bound does not allow to prune, the branching operation is activated and the current subproblem is further decomposed into two or more smaller subproblems. The newly computed subproblems represent new intermediate node in the tree, with the current subproblem originating them being their parent, and so on until all the search tree is fully explored.

2.2 Sources of parallelism

Pruning the B&B nodes can significantly reduce the size of the search space by exploring only those parts that exhibit promising costs. However, B&B is a computing intensive algorithm that requires a relatively huge computational effort especially when dealing with large scale and difficult problem instances. Parallel and distributed computing is among the classical alternatives that are used in order to speed up the computations of serial B&B. This has been the object of abundant work and a relatively rich literature can be found about the subject. In particular, the sources of parallelism in B&B are now well-identified and one can find several taxonomies and classifications on the subject B&B [49, 50, 24, 8, 38]. All these classifications share the following simple observation. The problem-dependent bounding operation is many often the most time consuming part of a serial B&B algorithm and much gain can be obtained when parallelizing

Algorithm 1: A simplified template of serial B&B for minimization problems

```
Input: $r$: root node representing the whole problem to solve; f: the objective function to minimize ;
Output: $x^*$: the optimal solution of the problem; $f^*$: the value of the optimal solution $x^*$;
$\mathsf{T} \longleftarrow r$ ; $x^* \longleftarrow \bot$ ; $f^* \longleftarrow \infty$;
while $\mathsf{T} \neq \varnothing$ do
    $P \longleftarrow$ SELECT($\mathsf{T}$) ;
    if $P$ is a leaf then
        $P.cost \longleftarrow f(P)$ ;
        if $P.cost < f^*$ then $f^* \longleftarrow P.cost$; $x^* \longleftarrow P$ ;
    else
        $P.lower\_bound \longleftarrow$ a lower bound for $P$ computed using a problem-dependent method ;
        if $P.lower\_bound \geq f^*$ then PRUNE($P$) ; $\mathsf{T} \longleftarrow \mathsf{T} \setminus P$ ;
        else $(P_1, \ldots, P_k) \longleftarrow$ DECOMPOSE_BRANCH($P$) ; $\mathsf{T} \longleftarrow \mathsf{T} \bigcup (P_1, \ldots, P_k)$ ;
```

this step. This type of parallelism is known as low level or node-based. We can distinguish two basic variants. In the first variant, the bounding operation is parallelized when executed for one single subproblem. In the context of our study, this is of limited interest since the bounding operation is problem-dependent. In the second variant, many bounding operations are carried out for different subproblems which constitutes a much more generic and standard source of parallelism in parallel B&B. Tightly related to this source of parallelism, another important type of parallelism in B&B consists in exploring different subproblems in parallel. This is known as high level or tree-based parallelism. It typically consists in exploring different B&B subtrees in parallel, that is distributing the computed subproblems over the available computing processes and performing the serial B&B in parallel. This type of parallelism enables to implicitly perform the bounding operation in parallel when concurrently exploring different subtrees, but it might also imply different explorations strategies.

2.3 Parallel and distributed B&B

Putting the pieces together, we can view a parallel B&B algorithm as a parallel tree search algorithm where the tree is constructed dynamically at runtime as a consequence of the branching and pruning operators. At every node of the tree, a bound has to be computed before determining whether the tree can be expanded or not. As the parallel exploration is carried out (either to compute bounds or to explore subtrees), the shape of the tree can vary very substantially from a node to another. This constitutes one of the major difficulty when effectively deploying a parallel B&B algorithm and maintaining all computational resources busy doing useful work. In fact, the B&B search tree is known to have a highly unbalanced structure, in such a way scheduling the induced workload evenly and distributively over available processes is a challenging issue. This is especially believed to be difficult when considering a heterogeneous environment with different type of distributed entities having possibly different compute power and compute abilities. In the following, we shall give a brief overview of some existing work on the subject.

2.3.1 Distributed B&B

In message-passing distributed and grid environments, the classical Master-Worker paradigm is many often adopted to parallelize the workload of parallel B&B. In this paradigm, most of the protocols like distribution and scheduling of B&B work units run only on the master which tries to maintain a faithful global view of work being processed. This makes the system easy to deploy and to experiment. Among others [25, 26, 33], we can cite the B&B@GRID approach described in [42], where the authors came with an interval representation of B&B workload in attempt to reduce the communication latency and the cost of synchronizing and updating workers. However, a well-known constraint of the Master-Worker model is its limited scalability, as the master becomes overloaded with the increase in number of workers. In order to deal with scalability and elevate the role of the master, hierarchical approaches have been proposed [1, 2, 53]. In particular, we can cite the recent work of Bendjoudi et al [4, 5], where the authors presented a distributed protocol called AHMW (Adaptive Hierarchical Master Worker) that organizes compute nodes in a hierarchical tree backbone. Every compute node can both play the role of a master and/or a worker, depending on its height in the hierarchy. The global B&B search tree is then decomposed into B&B subtrees which are mapped into the master hierarchy dynamically at runtime. The general idea of AHMW is to adapt the size of the B&B sub-trees being processed by each master/worker in an attempt to balance the load evenly. Despite skillful design, this approach induces a problem-specific B&B tree traversal which is shown to be outperformed by a standard DFS tree search. In fact, continuing the efforts for more efficient and scalable parallel B&B algorithms, the fully distributed approach presented in [51] was proven to substantially outperform the previous master-worker and hierarchical protocols while being fully distributed. The approach there-in is based on an overlay-centric protocol extending on the standard random work stealing protocol [51] used for dynamic load-balancing. Different from other previous works, the extensive and detailed comparative study conducted in [51] demonstrates that parallel branch-and-bound can highly gain both in generality and in efficiency by adequately bridging the gap between the B&B specific operations and the generic protocols available from the high performance computing community for scheduling dynamic and highly irregular applications. Let us remark that the previously mentioned studies are dedicated to completely distributed systems and did not address the issues rising when shared-memory and graphical processing units are additionally available in the system.

2.3.2 Shared memory B&B

In the previous discussion, we highlighted a bench of recent developments of parallel B&B specific to the message passing distributed and grid environments. Adaptations of the generic master-worker and the work-stealing paradigms are actually the corner stone of several other studies targeting different types of computing environments. In particular, one can find an abundant literature dealing with shared memory parallel B&B, see e.g., [46, 47, 20, 3, 32, 9, 21, 44]. With respect to multi-core systems, work pool(s) management is shown to play an important role for the performance of shared-memory parallel B&B. In fact, we can see from the generic template of Algorithm 1 that an iteration of a B&B consists of a procedure to pop a subproblem from a pool, perform a set of operations (branching, bounding and pruning), and then insert one or several generated subproblems into the pool. Generally speaking, work pool management refers to a specific data structure (e.g. array, list, map, stack, queue, etc) placed in a memory location where processing units can find and store generated subproblems during execution. In shared memory parallel B&B, simultaneous I/O operations of several processing units to the same work pool(s) poses several critical challenges. In practice, there are two common strategies with several variants. In single pool based algorithms, only one memory location is used to maintain a single global pool shared among compute units. Synchronization techniques are unavoidable to synchronize among worker threads when popping/pushing a node from/to the single global pool. For instance, a single pool approach is presented in [41], where the authors reported a big gap between their implementation and the ideal linear speed up. There are many reasons behind the scene however synchronization is shown to be the biggest issue. In multiple pool based algorithms, a set of different memory locations are used to handle B&B work units. There are some variants depending on the number of pools used in the system. The three most popular are called collegial, grouped and mixed. In the first case, each processing unit has its own pool. In the second one, all processing units are partitioned in several groups and each group shares the same work pool. The choice of the number of pools depends on the number of processing units as well as their accessing frequency. The last one is a mixed between collegial and grouped. Each processing units is associated with its own pool and share a single global pool with others. The multiple pool B&B algorithms are intended to tackle the bottleneck problem raised by the single pool algorithms.

In this context, we can cite the work of Casado et al. [12] who proposed two multi-threaded schemes for parallelizing B&B. In the first scheme, all threads share a global pool of generated subproblems therefore a synchronization mechanism is used to synchronize the accesses of all threads in a master-worker style. In the second scheme, each thread manages a local pool to avoid a significant overhead of the synchronization caused by the global pool of the first scheme. A dynamic load balancing is proposed to deal with the irregularity of B&B. At each iteration, if a certain condition is satisfied, a thread creates a new one and migrates work from its local pool to the pool of the new one. The condition for new thread creation is described as follows: the number of running threads are less than the total number of available cores and there is more than one subproblem in the local pool of the thread. In the same spirit, Evtushenko et al [22] presented a B&B solver that allows to deal with both shared and distributed memory environments. For shared memory, they use both a global pool and one local pool per compute thread. When a thread becomes idle it picks a subproblem from the shared pool, or it stays blocked unit the shared pool is fulfilled. A thread can process a B&B node from its local pool and add new nodes. After a number of steps, a thread migrates some problems from its local pool to the shared pool which stands for load balancing. In the distributed memory, they use a master-worker like paradigm to coordinate the distributed threads. Similar load-balancing considerations are addressed in [30, 29], where a thread can create a new one if the maximum number of threads is not reached, and a centralized approach harnessing a number of MPI processes is used in an attempt to reduce communications overhead.

Recently, Savadi et al. [46] introduced an approach taking into account the memory hierarchy of multicore systems. The authors proposed to map the task tree of B&B to the memory hierarchy tree of multicore systems. In the memory hierarchy tree, non-leaf nodes and leaf nodes are mapped to cache/memory components and to processor cores respectively. A node of the task tree is mapped on a level of memory tree which has equal or larger memory size than the needed memory for the node execution. Silva et al. [47] proposed the so-called Multicore Cluster Model at the aim of capturing the influence of memory hierarchy and contention in multi-core systems. Three communication models are addressed: i) the communication made through shared memory by intra-chip cache, ii) through inter-chip shared memory and iii) communication between cluster nodes via message passing. Some improvements are reported when the model is used to implement a parallel B&B algorithm for the Set Partition Problem. More recently, a multi-threaded parallel B&B is described in [40, 37] where the work stealing paradigm is combined with the so-called factorial Number system to encode permutational optimization problems, e.g., Flowshop. Only up to 16 CPU-threads are used in the experiments shown there-in. Moreover, the proposed approach focused on the encoding of B&B work units, and unfortunately no comprehensive parallel evaluation with respect to a standard multiple pool work stealing approach was provided.

In contrast to the previously discussed papers, we target large scale heterogeneous systems containing some hundreds of processing's units constituted as a mixture of cores, distributed CPUs and GPUs. We also exploit parallelism with an eye for the whole system and study the scalability of our approaches at different configurations. Moreover, we show that at the lower scales, our approaches are able to attain near-linear speed-ups which suggests that the largest scales constitute the hardest challenges in parallel B&B.

2.3.3 Parallel B&B with GPUs

When turning to more complex compute platforms, parallel B&B is being continuously revised in order to follow the new trends in high performance computing. In this respect, the rapid technological advances in GPUs, considered as highly parallel, multithreaded and many-core architectures, have led to new opportunities in speeding up the B&B search. However, because of the irregularity of the B&B tree search, scheduling the load inside a GPU device is not fully compatible with the underlying SIMD programming model. In [10, 16, 39, 7, 34, 11], implementations on a single GPU device are presented with respect to specific optimization problems, e.g., Flowshop, Knapsack and TSP (Traveling salesman). With respect to this paper, these studies are rather of limited interest since they are problem specific and do not consider a heterogeneous and large scale setting. When aggregating multiple GPUs, only few investigations are known for parallel B&B. In [13], a Master-Slave approach is adopted and only an experimental scale of 2 GPUs is reported, which is clearly not sufficient to express the power of modern compute platforms (sub-optimal speed-up are actually reported). In [17], the authors experienced multi-core pool-based B&B with parallel bounding inside the GPUs devices. Only shared memory threads are studied and small parallel scales (up to 6 threads) are considered there-in. In [14, 15], the master-slave approach of [42] is combined with a GPU-guided implementations similar to [17] in an attempt to tackle large scale environments. The focus there-in is rather on feasibility and implementations issues; but unfortunately, scalability and performance optimality were not addressed in a comprehensive manner.

To our best, the first to address the joint use of multiple distributed CPUs combined with multiple GPUs for B&B was conducted in our previous study presented in [52]. The work stealing paradigm is adopted at the aim of getting rid of the synchronizations overheads inherent to the master-slave architectures. However, the gap between the actual performance and the optimal linear speed-up that one would ultimately like to obtain remains open at large scales. In this paper, we extend the work in [52] and come up with an advanced approach that takes into account the shared-memory level exposed by some of the available compute resources. Our experimental analysis shows that the speed-up gap with respect to the linear one is significantly tightened. Despite the highly irregular nature of B&B and the harsh constraints inherent to heterogeneous large scale multi-core multi-CPU multi-GPU environments, the protocols described in this paper as well as the comprehensive experimental study conducted at different scales and configurations shed more light into how to effectively design and deploy efficient parallel B&B algorithms.

3 Challenges and design principles

In this section, we provide an overview of the general principles guiding our parallel B&B design. Considering that the target platform is heterogeneous with possibly different levels of hierarchy and compute ability of the underlying processing units (PUs), we can identify different but tightly coupled issues for an efficient and effective B&B parallelization. We summarize them in the following. Our main goal is to elicit in a generic manner the main challenges one is facing, and to provide a simplified, but a clear idea, of how we shall address them without going into complicated (yet important) technicalities. A throughout detailed discussion shall then follow according to the target compute platform and its characteristics (multi-core, distributed multi-CPUs or multi-GPUs).

3.1 Mapping B&B parallelism

As discussed in the background section, the main type of parallelism used for generic B&B is to bound several B&B tree nodes in parallel, and, at the tree level, to explore different B&B subtrees in parallel. Therefore, we have to select a proper mapping of B&B parallelism to the underlying hardware for optimal performance. The GPU-many-cores can suffer from thread divergence due to work and execution irregularity, whereas the CPU cores are less sensitive. Hence, we shall restrict the GPU side to deal with only the B&B node-parallelism. Only the bounding operation with respect to several B&B tree nodes is carried out inside the GPU, i.e., by launching a kernel to be executed by the GPU device. The other types of PUs are responsible of the parallel B&B tree exploration, i.e., decomposing and expanding the tree — this is of course in addition to the bounding step. Let us remark that a whole pool of B&B tree nodes has to be prepared by the PU hosting a GPU and uploaded into the GPU memory in order to be processed. This step implies important hardware-dependent technicalities that have a deep impact on performance as will be detailed later.

3.2 PUs compute power

The computational capability of a single CPU against one single GPU is different. According to the optimization problem being tackled, the relative performance can range from few to hundreds orders of magnitude in favor of the GPU device. This is because the most time consuming part in a sequential B&B algorithm is the bounding operation. Depending on how efficiently the problem-dependent bounding operation can be implemented inside the GPU, the many GPU-threads executing the evaluation of several B&B nodes in parallel can allow for substantial speed-ups. Moreover, the compute power of distributed PUs of the same type could be substantially different when dealing with a heterogeneous environment. This is typically the case if we consider different CPUs or different GPUs comings from different clusters. Hence, the relative compute power of PUs is to be carefully taken into account so that workload can be well balanced across different processes.

3.3 Workload irregularity

Besides the difference in PUs compute power, the dynamic and irregular nature of B&B tree constitutes a major source of workload unbalance that can dramatically prevent high performance and scalability. Processing the search tree in distributed fashion might seem trivial, for instance, by iteratively generating and distributing subproblems over the available PUs. Actually, variations of this strategy are often implemented by the existing master-slave approaches. However, this approach fails to achieve optimal performance because (i) the shape of the search tree is unknown in advance, i.e., it is difficult to predict in advance the nodes of the tree that would generate enough work for subsequent iterations, and (ii) the actual explored subtrees differ in shape and have very unbalanced structures, e.g., the depth and the number of nodes that could be attained when following different branches are highly variable. In this respect, we argue that a load balancing mechanism where workload unfolds dynamically at runtime while the different B&B tree explorations are running in parallel is mandatory. The main challenge here is to minimize the time a PUs stays idle waiting for some B&B tasks to process. To deal with this issue, we use a hybrid work-stealing based mechanism that takes into account the hierarchy of the heterogeneous compute environment by including inter-socket communication with message passing through networked PUs, and multi-core intra-socket communication with shared memory. We also use an efficient data structure to handle B&B sharable work pools. For clarity and to better illustrate the impact of load balancing in a fair manner, we shall start describing our approach in the case where the shared memory level is ignored, and then describe how it can be extended in more complex heterogeneous settings.

3.4 Distributed global operations

A B&B process needs to maintain the cost of the best known feasible solution in order to prune subproblems efficiently. This information is hence to be shared distributively between all B&B parallel processes. Moreover, we have to deal with termination detection with minimum disturbance to B&B computations. When a Master-Worker paradigm is considered, this can be handled in a centralized manner using the master as a relay which can introduce further delays and bottleneck issues around the master. In our approach, we shall consider a fully distributed scenario where no master is available. These two global operations are thereby handled distributively by structuring distributed PUs using a tree overlay as it will be detailed later.

4 The 2MBB approach: multi-CPU multi-GPU parallel B&B

In this section, we consider a completely distributed compute setting where all PUs are distributively connected via a network and each PU is considered to be either a single CPU-core or a single CPU-core equipped with a GPU device. In other words, we do not use any sophisticated shared-memory communication mechanisms but stick in a purely message passing distributed scheme that we call 2MBB, i.e., Multi-cpu Multi-gpu B&B. The design components presented within this context shall be extended in the next section by considering more complex compute environments harnessing multi-core shared memory components as well. For the sake of presentation, our 2MBB is described in an a step-by-step incremental fashion.

4.1 B&B work units

Before going into further details, let us define precisely what we shall term a B&B work unit in the rest of this paper. It is with respect to a B&B subproblem, that is a node in the B&B tree which is not yet explored (the branching and the pruning operators are not yet applied). When a PU holds some B&B work units, it can decompose it into several subproblems in order to evaluate the corresponding bounds. This defines how the B&B traversal is actually carried out. In our approach, a combination of DFS and BFS traversals is implemented. This is mainly motivated by the difference between the compute abilities of CPU and GPUs cores. Since a GPU can be efficient only when it can bound *many* tree nodes in parallel, we have to prepare enough work (by decomposing enough subproblems) before activating the device computations. As a consequence, a pool with relatively few subproblems that are pushed inside the GPU for bounding would result in sub-optimal performance. A DFS traversal allows us to quickly go deep in the search tree; However, it does not allow us to infer much parallelism to be handled by the GPU device. In contrast, a BFS traversal makes it possible to generate sufficiently many tree nodes to push into the GPU device, thus gaining much in performance since the serial execution time is often dominating the bounding step. Hence, in the case where a CPU is considered, the traversal is always carried out in a DFS fashion, i.e., branching

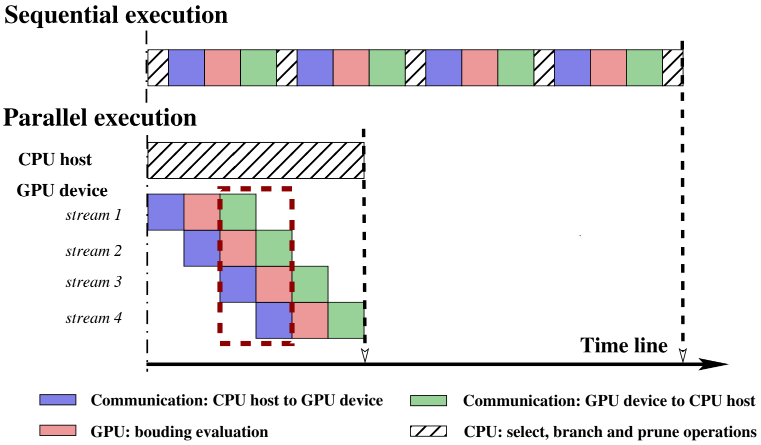

Figure 1: Illustration of host-device parallelism on single CPU-GPU using 4 concurrent streams.

the best candidate sub-problems first. In the case where a GPU is considered, the core hosting the GPU device manages to prepare enough subproblems using a BFS strategy with respect to the subproblems that it owns.

With respect to B&B work units, let us notice that transferring them distributively might induce congestion and communication delays at the network level. This may motivate the use of advanced strategies to encode B&B work units, i.e., to represent them in a compact manner and thus reduce the size of distributed messages. For instance, the so-called interval encoding was proposed in [42] in order to help reducing the cost induced by work units transfers. We do not use any such mechanisms in our approach, but prefer to use a direct encoding of B&B nodes (i.e., full subproblem specification). In fact, our preliminary experiments showed that work encoding can be an issue only when the underlying network is not sufficiently quick with respect to the amount of data being transferred. This is actually very unlikely in nowadays modern high performance interconnect[1].

4.2 Host-device parallelism for single CPU-GPU

Let us zoom on one single PU equipped with a GPU. In this case, the *select*, *branch* and *prune* operations are performed by the CPU host and the bound evaluation is handled by the GPU device. For the bound computations to be run inside the GPU, input data comprising several B&B tree nodes is transferred to the GPU memory, a kernel is executed on the input data and the outputs (the computed bounds) are copied back to the CPU host. In standard CPU/host-GPU/device execution, the previous operations are synchronized sequentially. In other words, while the CPU host is performing *select*, *branch* or *prune* operations, or even while copying data to/from device, the GPU is stalled. Similarly, while the evaluation of B&B tree nodes is running inside the GPU device, the CPU host is stalled. This can significantly slow down computations especially when the CPU host and the GPU device can perform concurrent operations *in parallel*.

In order to address the above issues and minimize the time the host and the device are stalled, we explore the new hardware and software possibilities offered by the recent technology. Instead of synchronizing and waiting for the completion of operations running at the GPU device, the CPU host can still dispatch operations into the GPU device asynchronously and continue its computation. In more details, a sequence of operations (namely: copy data from CPU to GPU, perform parallel bounding operations at GPU device and copy results from GPU to CPU) is wrapped into *a stream* which can be *asynchronously* dispatched to GPU device for execution. In the same spirit, many streams of operations can be *asynchronously* shipped to the GPU device. Let us notice that the operations inside the same stream get automatically synchronized and executed sequentially, but the operations of different streams might be executed in parallel, e.g., execute

[1] For instance, our preliminary experiments showed that the B&B work encoding described in [42] does not improve performance when compared to a direct encoding (for the problem instances under consideration), but it rather makes the overall protocols more complex. Actually, we even observed a loss in performance. This is because to decode work and recover the original B&B subproblems using the encoding of [42], we need extra serial compute operations that are relatively time-consuming compared to a direct transfer of B&B subproblems.

the kernel of stream 1 and retrieve data from stream 2 concurrently in parallel. Figure 1 illustrates a host-device parallelism of a CPU-GPU system where the GPU has one compute engine for executing kernel computations and two copy engines: one for copying data from CPU to GPU and one for copying data from GPU to CPU. Therefore, this CPU system is able to execute up to three operations simultaneously in parallel.

4.3 Adaptive distributed work stealing

In order to keep the available distributed PUs working, we need to design a distributed mechanism to distribute work units (B&B sub-problems) among them. The irregularity of B&B subtrees generated at runtime can eventually lead to very poor performances if many PUs are underloaded and few others are highly overloaded or when the cost of synchronizing PUs and transferring work becomes substantial. We propose to use a dynamic load balancing scheme based on the work stealing paradigm [23]. In the so-called random work stealing scheme (RWS), if a PU runs out of work[2], it acts like a thief and tries to steal work from another PU, called victim, *chosen uniformly at random*. This simple decentralized protocol allows different PUs to acquire work cooperatively in parallel, thus eliminating the time required to synchronize work unit distribution. In this context, the stealing granularity, that is the amount of B&B tree nodes to be offloaded from victims to thieves plays a crucial role. When this amount is very small, the large overhead is observed since many load balancing operations are performed. At the opposite, when it is very large, too few load balancing operations will occur, thereby resulting in large idle times despite the fact that surplus work could be available. In classical RWS approaches, this is a hand-tuned parameter which depends on the distributed system and the application context [43]. In practice, the so called steal-half strategy is often shown to perform efficiently using homogenous computing units [6, 45, 19].

In this paper, we consider a variation of the steal-half strategy in order to cope with the heterogeneous nature of the distributed platform. We make every PU maintain at runtime a measure, denoted by x, reflecting its computing power. Every PU updates this value continuously with respect to the work processed in previous iterations. We simply use the average time needed for processing a B&B subproblem. More precisely, each PU sets its computing power to be $x = N/T$, where T is the time elapsed since the PU has started the computation and N is the number of B&B tree nodes explored locally by that PU. Notice that time T includes, in addition to tree node evaluation (i.e. B&B lower bounding), the time needed for other search operations (i.e. select, branch and prune) but *not* the time when a PU stays idle. When running out of work, a PU attempts to steal work by sending a request message to one other PU chosen at random, while wrapping the value of x in the request. If the victim has some work to serve, then the amount of work (i.e., number of tree nodes) to be transferred is in the proportion of $x/(x + y)$, where y is the computing power maintained locally by the victim. Otherwise, a reject message is sent back to notify the thief and a new stealing round is performed. Initially, the value of x is normalized so that all PUs have the same compute power. Hence, the system starts stealing half and then the stealing granularity is refined for each pairwise PUs.

4.4 Termination and knowledge sharing

One issue missing in the description of previous distributed protocol is how to detect the termination [18] of the parallel B&B algorithm, i.e., when all PUs are idle and no B&B work units are being transferred anywhere in the distributed system. For this purpose, we use a fully distributed technique, where PUs are initially mapped into a tree overlay and termination is detected in an 'Up-Down' distributed fashion. In the up phase, if a PU becomes idle and has not served any stealing request, it will then integrate a positive termination signal to its children signals. If a PU turns to idle and has served at least one stealing request, it will then integrate a negative termination signal to its children signals. Then the termination signal is forwarded to the parent and eventually to the root. In the down phase, if the root receives at least one negative termination signal from its children, it broadcasts a signal to restart a new round of termination detection. Otherwise, if only positive termination signals are received, the root broadcasts a message to announce global termination.

[2] Initially, the B&B root problem is owned by one PU and all other PUs starts in an idle state

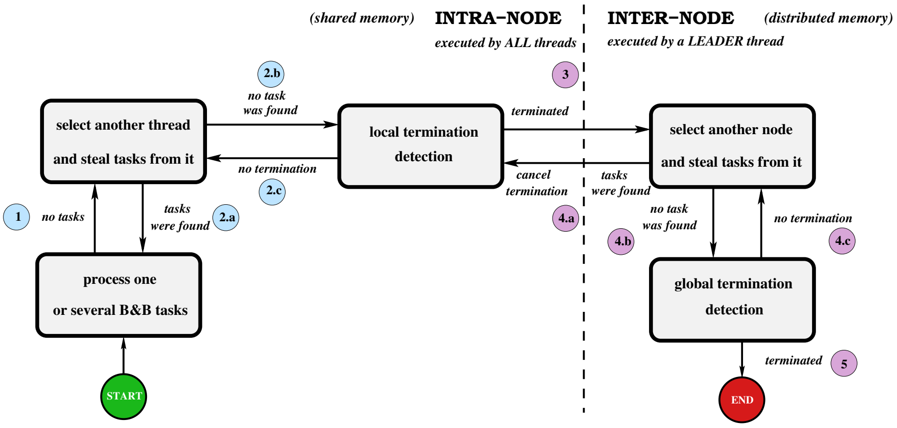

Figure 2: Overview of a thread state in the 3MBB approach.

The tree overlay used in our implementation is a binary one so that the worst case number of simultaneous messages to be handled by a PU (tree degrees) and the time a message spent traveling before flooding all PUs (the diameter of the tree) are kept low. This allows us to scale up the distributed PUs while avoiding communication bottlenecks and performance degradation once a termination phase is initiated. Moreover, the tree overlay spanning the different PUs is used in order to communicate the best upper bound found by any PU during the execution of the parallel B&B. Notice that since the diameter of a binary tree overlay is logarithmic, propagating this knowledge among PUs has a very limited communication cost.

5 The 3MBB approach: multi-core multi-CPU multi-GPU parallel B&B

Heterogeneous platforms usually come with multi-core processors, and a mixture of shared memory and distributed memory components. In order to achieve high performance, this heterogeneity brings further challenges especially when dealing with dynamic load balancing. From the perspective of the hardware environment, there exist two main levels of parallelism: *intra-node* parallelism which refers to shared memory computations among cores of a single compute node and *inter-node* parallelism which refers to distributed memory computations. Since the difference of communication cost between shared memory and distributed memory can be very substantial, balancing workload between cores in a single compute node can be much faster compared with the one in a distributed memory. In this context, we describe the 3MBB approach which extends the previous 2MBB approach.

In the rest of this section, we assume that some compute nodes are identified as multicore hardware components, and some of them can be equipped with GPU devices where a GPU device is hosted by one CPU core. We shall essentially extend the 2MBB architecture so that workload distribution is handled at two different levels. In the first level, work stealing is performed within every multicore shared memory using a specific data structure to encode work pools. In the second one, work stealing is performed using message passing essentially similar to the 2MBB architecture but with some exceptions. In Fig. 2, we provide a general overview of thread states which is to be detailed in the following.

5.1 Intra-node parallelism

5.1.1 Preliminary observations

Let us first zoom in the case of a single shared-memory multi-core component. We use asynchronous multiple work pools in our design. Every single thread runs on a single physical core and manages a separate local work pool where newly generated B&B subproblems are stored. The main challenge is to efficiently share B&B work units among the threads of the shared memory system while minimizing the time a thread stays idle and the disturbance caused to the other B&B computations when attempting to distribute work.

For this purpose, we again consider work stealing in such a way a thread plays a role of either a thief or a victim. Whenever a thread finishes all subproblems of its work pool, it becomes a thief and tries to steal works from the work pools of other threads. Stealing operations and task offloading among shared memory cores are to be converted to read/write operations to/from some data structure, and shall be very fast compared to a message-passing mechanism in a distributed memory setting. Nevertheless, simultaneous read/write operations to common data structures might cause race conditions that can eventually lead to unexpected behaviors. Locking and synchronizing simultaneous accesses of several threads are the popular solution but they come with a price. Overusing these techniques highly reduces the potential parallelism of a multi-core system. Therefore, achieving a good performance requires a well design of the underlying data structures in order to alleviate thread synchronizations as many as possible.

A straightforward solution could be that each thread manages a fully sharable work pool. In other words, a thread can fully access to the work pools of other threads during execution. This approach is simple in design as well as in implementation; but it introduces high overhead as locking is required to synchronize multiple accesses to the work pool of a given thread. For instance, when a thief thread tries to steal from another victim thread, it has to lock the work pool of the victim then offloads tasks from the victim's work pool to the thief's work pool. The vicim is then forced to be stalled during the locking time as it is not granted access to its own work pool to pop/push tasks. A better approach could be to split every single work pool into two different pools: one for the owner thread in order to store new generated tasks, and one for storing sharable tasks exposed to the other threads. In this way, locking can be avoided whenever the owner thread tries to access to its own pool. Nevertheless, tasks have to be copied backward and forward between the two pools whenever one of them is empty. The cost of many copy operations constitutes a critical shortcoming especially when facing the irregular nature of B&B workload.

5.1.2 Split work pools

We found that the most appropriate approach in order to avoid extra-computations and to optimize thread efficiency is to use a single work pool split into private and public part as originally described in [19]. Basically, the private region is exclusively owned by the owner thread and the public one is exposed to other threads. However, they both constitute a single data structure endowed with a *split* pointer in order to identify the frontier between the two regions. The amount of tasks of the private and public regions are then adjusted by simply moving the *split* pointer forward or backward without any memory copies.

In Fig. 3, we describe the general design of a split work pool for B&B. More specifically, the private portion is implemented like a Stack in a LIFO manner and the public portion works like a Queue with a FIFO manner. The LIFO property of the private region allows threads to perform a DFS search on the B&B tree in order to reach a leaf quickly. The FIFO property of the public region allows threads to share coarse grain B&B subproblems that are likely to generate more children subproblems; and thus it encourages the transfer of useful B&B work units. In fact, the B&B subproblems that are close to the root usually contain large amount of works while this amount decreases as the recursion develops more deeply in the tree. Besides, let us remark that steal attempts from the public region of a given thread are handled using a standard lock operation to avoid concurrent thiefs' accesses; but the victim thread is in contrast lock-free with respect to its private region so that it can continue processing B&B subproblems and freely push/pop tasks.

A crucial feature when implementing this split work pool structure is to adjust the amount of work units in the private and public region at runtime. For this purpose, the split pointer is associated with two operations: *release* and *reacquire*. The *release* operation refers to moving the pointer towards the private region by a given amount in order to expose some private tasks to the public region. This operation is handled by the owner thread every time the public part gets empty (because work was stolen so far by other threads). In this case, the release operations enables to expose work in the public part by moving the split pointer to *half* of the private queue. The *reacquire* operation refers to the reverse operation where the *split* pointer must be moved towards the public region in order to move public tasks to the private region. In our architecture, we define two different rules for this operation depending on whether a GPU device is hosted by a core or not:

- **GPU split pointer**: refers to performing the *reacquire* operation in advance before the private region

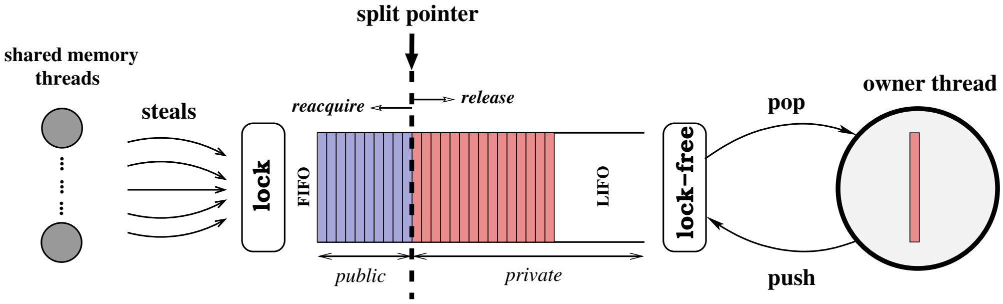

Figure 3: Simplified view of a split work pool

is completely empty. In fact, when a core is also hosting a GPU device, the thread running on the core has the role of preparing B&B subproblems to be bounded inside the GPU as described previously for the 2MBB architecture. The maximum potential compute power of a GPU is only guaranteed when it is provided large enough tasks as inputs. However, it might happen that the private part does not allow a thread to generate enough subproblems to push into the GPU device. Therefore, the *reacquire* operation is performed each time the private part does not allow to reach the maximal size of input data the GPU can handle for optimal performance. More specifically, each time the work maintained in the private part is not enough to keep the GPU busy, the split pointer is moved to *half* of the public queue (if not empty). Again, this allows us to avoid moving back and forth data from the public to the private region, while making full usage of the GPU power.

- **CPU split pointer**: This is with respect to a thread not hosting any GPU device. In this case, the reacquire operation is performed if and only if the private region is empty. The split pointer is then moved to *half* the public region.

To resume, every time a thread in *one single* shared memory component wants to process a subproblem, it takes one from the front of its local queue which actually lies in the private part. Every time a thread generates new subproblems, they are pushed again at the front of the private part. The release and acquire operation are by the mean time executed according to the description given above. A thread becomes idle when its split queue becomes empty, i.e., it has exhausted all its subproblems being in its private and public regions. When a thread runs out of work, it attempts to steal work from the other threads. More specifically, it choose one victim thread uniformly at random and checks the public part of the corresponding queue. Two situations can occur. If the public part of victim's queue is not empty, then the thread is able to steal some work from the tail of the queue after locking it, and hen it can push stolen work at the front of its own private queue. Otherwise, no work is found and the steal operation is renewed (choosing uniformly at random a victim among the shared memory threads) until some work is fetched or global termination is announced.

5.2 Inter-node parallelism

Inter-node parallelism refers to computations among distributed nodes where no shared memory is available. Once again, the main issue is to transfer work efficiently and load balancing operations are to be implemented using message passing among the distributed nodes. One naive approach is to allow every individual thread to perform random work stealing as for the 2MBB and to take into account where the randomly chosen victim lies. In other words, every single thread could use intra-node parallelism to perform a steal attempt whenever the selected victim belongs to the same shared memory system, and message passing whenever the victim is running at a remote node. Although this approach is simple and straightforward, it does not fully take benefit from the hierarchy exposed by the heterogeneous compute system. In fact, several distributed steals performed by different threads can induce high communication delays.

5.2.1 Hybrid distributed stealing

In the 3MBB approach, stealing across distributed nodes is only enabled when all threads detect that there is no work available in any pool of the shared memory component they belong to. This allows the threads of a shared memory compute node to complete all work available locally before attempting to acquire work distributively from other compute nodes. However, if all idle threads in a compute node try to steal works simultaneously, they might generate large amount of messages and cause unnecessary overhead for processing them. This led us to handle the distributed steals by a single thread in every shared memory components as detailed in the following.

Whenever a shared memory component runs out of work, we perform distributed steals using message passing. For this purpose, one thread is initially elected to be a leader at every shared memory system. A leader functions as the other threads with one main difference: he additionally manages the distributed steals when work is no more available. Every leader has to detect when no work is available locally. This is easily solved using a global counter shared by the threads. Every time a thread runs out of work, it increments the counter in order to inform that it is idle. Only when the counter reaches the total number of threads belonging to the shared memory system (i.e., the number of cores), the corresponding leader starts performing distributed steals. Like in the 2MBB architecture, a distributed steal consists in sending a work request message to another remote leader chosen uniformly at random. Two situations can occur at the receiver side. In the first case, the remote victim is also performing distributed steals and thus no work is available within the corresponding shared memory system. Hence, the victim informs the leader originating the distributed steal by sending back a negative work response. When receiving a negative work response, a leader starts a new distributed steal until work is fetched or termination is detected. In the second case, some work is available in the shared memory system, and work is simply transferred distributively using a positive work response. When getting a positive work response, a leader unpack the transferred work into its local split queue and starts processing the received subproblems again. Notice that while a leader is performing distributed steals, the other threads sharing the same memory are waiting for the counter to be updated. As soon as the leader succeeds to acquire work, it resumes to a working state and resets the counter to inform the other waiting threads. Thus, if a distributed steal is successful, then at least one thread is eventually able to get some work from the public queue of the leader which allows work to flow again on the whole shared memory system.

For the above protocol to work correctly, we have to manage termination properly. We simply use the same tree overlay described for the 2MBB while considering only the leader threads. Since distributed steals are initiated if and only if all the threads at the leaders are idle, it is easy to see that whenever termination is detected among leaders, the other threads can be informed immediately using a shared memory variable maintained by every leader. In the same way, the exchange of the best solution needed for the B&B pruning is handled distributively using the tree overlay spanning the different leaders.

5.2.2 Aggregated steal granularity

To complete the description of the 3MBB architecture, we still have to set the amount of work to be transferred. We use essentially the same adaptive work sharing policy used in the 2MBB architecture but with following two differences: (i) The power of a compute node is measured as an aggregated value of all the threads at the shared memory node, and (ii) The amount of work to be considered at victims is the amount of all available tasks in the public regions of all local threads. In more details, a thief leader i collects the aggregated computing power $X = \sum_j x_{i,j}$ (where $x_{i,j}$ is the computing power of thread j at the shared memory component leaded by i) and sends a work request to a randomly selected victim k while wrapping the value of X. Similarly, upon receiving a steal request, the victim leader k also measures its aggregated computing power $Y = \sum_\ell y_{k,\ell}$ (where $y_{k,\ell}$ is the computer power of thread ℓ at the shared memory component leaded by p). The amount of work to be transferred is then in the proportion of $\frac{X}{X+Y}$. Technically speaking, the victim leader k collects $s_{k,\ell} = t_{k,\ell} \cdot \frac{X}{X+Y}$ work units from the public region of every work pool of all the local threads ℓ (where $t_{k,\ell}$ is the total available works at the public region of thread q at victim p). A total amount of $S = \sum_\ell s_{k,\ell}$ work units are then transferred to the requesting thief.

6 Experimental results and analysis

6.1 Experimental setting

To evaluate the performance of our parallel B&B, we consider the permutational FlowShop problem with the objective of minimizing the makespan (C_{max}) of scheduling n jobs over m machines. Flowshop is one the most challenging optimization problem which appears at the bottleneck of many industrial problems, e.g., logistics, manufacturing, etc. It is also used as a benchmark academic problem in order to study the properties of several resolution methods ranging from exact method like B&B in operation research, to heuristics and meta-heuristic in evolutionary computing and computational intelligence. The standard Taillard' instances [48] of the family of 20 jobs and 20 machines are considered. The time required for solving these instances on a standard modern CPU, starting from scratch (that is without any initial solution), can be of several hours which make this class of instances particularly interesting to solve in parallel while fairly evaluating the parallel efficiency of the underlying protocols[3].

The 2MBB approach is implemented using standard low level c++ libraries for all aspects including the underlying distributed protocols and message passing among computing processes. The 3MBB architecture is implemented in a hybrid manner by further using OpenMP for all aspects concerning thread execution within a single multi-core component. Concurrent host device computations needed for both 2MBB and 3MBB in the scenario some GPU devices are available are implemented using standard primitives provided by CUDA. The GPU kernel implementing the bounding operation which is specific to the considered Flowshop problem was taken from [13, 39, 52] and used as a blackbox since we are not interested in problem specific issues but rather in the parallelization of the B&B algorithm considered as a universal technique. The lower bound proposed in [36] which is based on the Johnson's algorithm [31] is used in the bounding step. Notice that this is the only component which is problem specific when implementing and experimenting our approaches.

Three clusters C_1, C_2 and C_3 of the Grid'5000 French national platform [28] were involved in our experiments. Cluster C_1 contains 10 nodes, each equipped with 2 CPUs of 2.26 Ghz Intel Xeon processors with 4 cores per CPU. Besides, each node is coupled with two Tesla T10 GPUs. Each GPU contains 240 CUDA cores, a 4GB global memory, a 16.38 KB shared memory and a warp size of 32 threads. Cluster C_2 (resp. C_3) were equipped with 72 nodes (resp. 34 nodes), each one equipped with 2 CPUs of 2.27 Ghz Intel Xeon processor with 4 cores per CPU (resp. 2 CPUs of 2.5 Ghz Intel Xeon processor having 4 cores) and a network card Infiniband-40G.

In the remainder, we shall run the designed approaches in several configurations at the aim of fairly eliciting their strengths and weaknesses under different scenarios. Two sets of experiments are designed as following.

- We consider the following set of experiments for 2MBB:
  1. Running our approach with a single CPU-GPU.
  2. Running our approach with multiple GPUs and multiple CPUs at different scales.
- We consider the following set of experiments for both 2MBB and 3MBB:
  3. Comparing our approaches when running them with multi-core CPUs and multiple GPUs in large scales.
  4. Comparing our approaches when running them with multi-core CPUs in large scales (without GPUs).

When deploying our experiments, we launch one parallel B&B thread at every CPU core or at every CPU core equipped with a GPU device (taken from cluster C_1). Regarding the second set of experiment (w.r.t 2MBB), the GPUs are taken from cluster C_1 and the CPUs are taken from cluster C_2. Regarding the second set of experiment (w.r.t 3MBB and 2MBB), the CPUs are taken from clusters C_1, C_2, C_3 and the GPUs are taken from cluster C_1. The only difference when deploying 2MBB and 3MBB is in the

[3] Larger instances (resp. smaller) instances would result in a so large (resp. small) workload that would not allow to fully appreciate the efficiency of a parallel algorithm at a reasonable distributed scale.

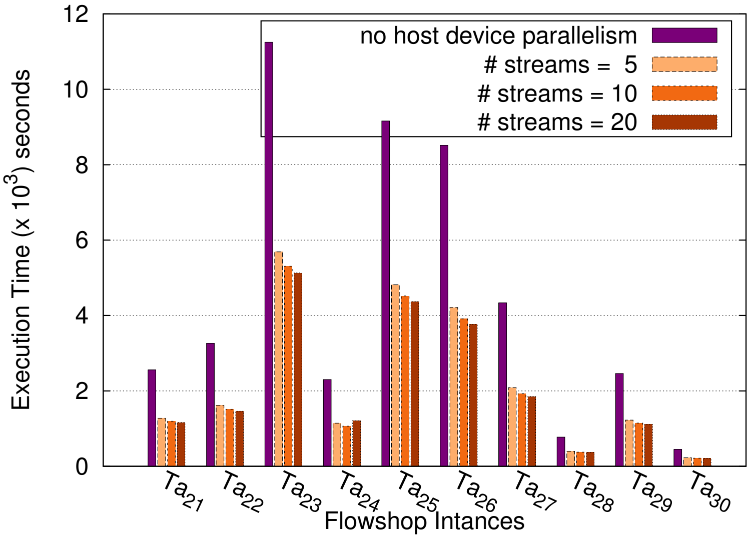

Figure 4: Impact of host device parallelism. Reported results are the execution times for the ten Taillard' instances with different number of concurrent CUDA streams.

communication model: message passing for 2MBB, and mixed shared memory and message passing for 3MBB. For all experiments, we measure T and N, respectively the time needed to complete the B&B tree search and the number of B&B tree nodes that were effectively explored. All reported speedups are relative to the number of B&B tree nodes explored by time units, that is N/T. Since, the clusters involved in our experiments have CPUs with different compute power, all our speedups are normalized accordingly. For that purpose, we proceed as following. Let α_i^j be the speedup obtained by a *single* PU j with respect to PU i. We naturally define the linear (ideal) normalized speedup *with respect to PU* i, to be $\sum_j \alpha_i^j$. For instance, having p *identical* GPUs and q *identical* CPUs, each GPU being β times faster than each CPU, our definition gives a linear speedup of $p + q/\beta$ with respect to *one single GPU*. In the remainder, all reported speed-ups are normalized with respect to one single GPUs unless when only CPU-cores are considered. In the latter case, reported speed-ups are normalized with respect to the compute power of a single CPU-core.

6.2 Impact of host-device parallelism

In Fig 4, we study the impact of concurrent host-device parallelism when running B&B on a single GPU hosted by one CPU-core. For the ten instances available in the Taillard' family 20*20, we report execution time obtained with and without the host-device operations described with the 2MBB architecture. We also report the impact of using an increasing number of concurrent CUDA streams when pushing data concurrently in the GPU device. One can clearly see that substantial gains are obtained with up to one order of magnitude improvement over a baseline implementation where data is pushed sequentially to the device. This tells us that when parallelizing a B&B using a GPU, data transfer is one crucial aspect that can constitute a major bottleneck.

Notice that the number of concurrent streams may affect performance. In our application scenario, this appears to be dependent on the considered instance. However, the impact is relatively marginal compared to the enabling of the host-device parallelism it-self. In the remainder of our experiments, we shall fix the number of streams to 10 since this appears to perform reasonably well on most of the considered instances.

Fig. 4 can also inform about the irregularity exposed by parallel B&B. Although the ten instances are coming form the same problem and have the same size (20 jobs on 20 machines), the execution times are significantly different from one instance to another. In fact, depending on the considered instance, the bounding and the pruning operations can have very different behaviors resulting in different actual shapes and sizes of the B&B tree. From our observations, we also remarked that the more time is needed to solve an instance is small, the more it is difficult to parallelize efficiently. This is because more fine-grain parallelism is needed and the amount of B&B work generated over all the execution is limited and it is

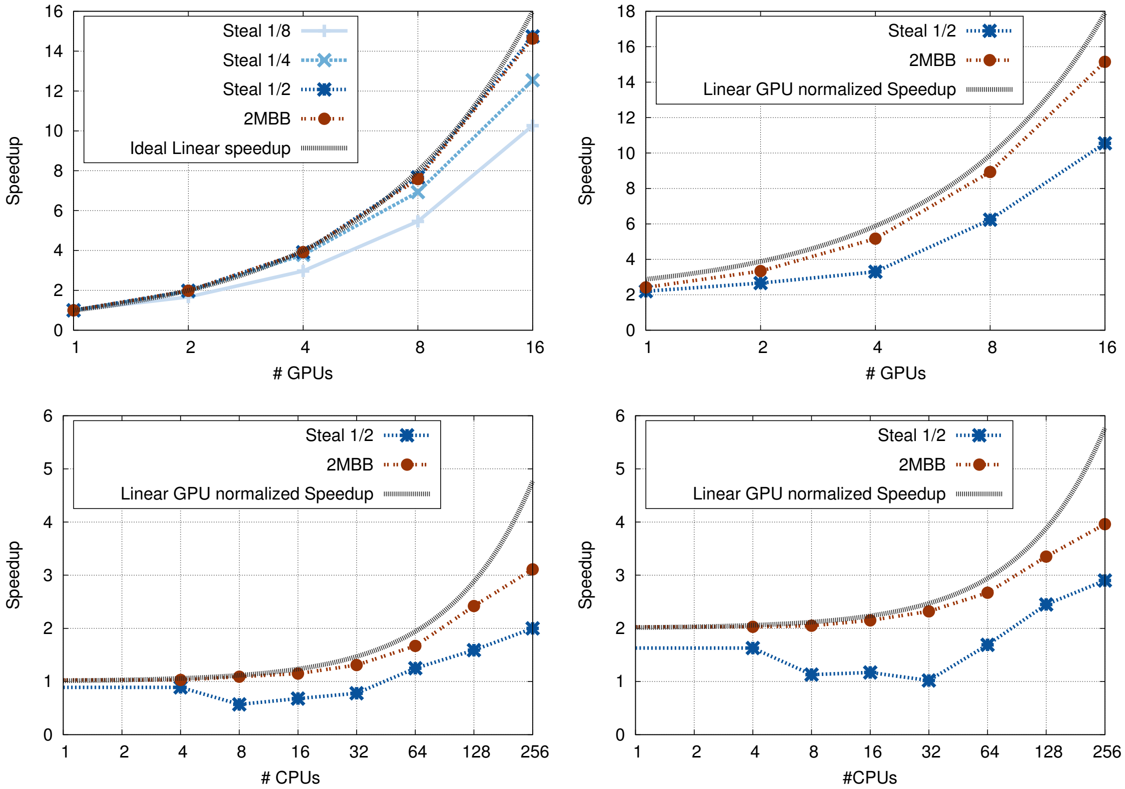

Figure 5: Relative speedup of the 2MBB w.r.t a linear speedup and the static steal-half strategy. Different compute setting with variable heterogeneity are considered. **Top-Left**: scaling GPUs from 1 to 16. **Top-right**: using 128 CPU-cores and scaling the GPUs from 1 to 16. **Bottom-Left**: using 1 GPU and scaling the CPU-cores form 1 to 256. **Bottom-Left**: using 2 GPUs and scaling the CPU-cores form 1 to 256. The X-axis is in the log scale and reported speed-ups are w.r.t. one GPU. The number of concurrent CUDA streams is set to 10.

more likely to be trapped in only few compute nodes. In order to keep our experiments manageable, we shall mostly consider the first instance in the Taillard's Flowshop family since it is observed to have a representative granularity with respect to the other instances.

6.3 Performance and scalability of the 2MBB approach

In this section, we study the scalability of the 2MBB architecture when scaling up the number of GPUs and the number of CPUs. In Fig. 5, we show the results we obtain when running our approach under different configurations.

First, we analyze the speedup of our approach (with respect to one single GPU) when deployed using only homogenous GPU-devices (see Fig. 5 top-left). We also report the speedup obtained when using different static stealing granularities where the amount of work to be transferred is initially set by a fixed parameter. Two main conclusions can be made in this experimental scenario. The 2MBB approach which uses an adaptive stealing mechanism performs similar to the best static stealing, that is the standard steal-half strategy, which we confirm to perform well for B&B. This is without surprise since in this setting the different GPUs have uniform compute powers and the amount of work set adaptively in the 2MBB approach is likely to converge to $1/2$. More importantly, the 2MBB approach is able to scale linearly with the number of GPUs. Nevertheless, we can notice a slight decrease in speedup compared to the linear one at the largest scale of 16 GPUs.

When turning to non-uniform configurations where the ratio between the number of GPUs and CPUs is varied substantially, we are able to report the following findings. In Fig. 5 top-right, we show the speed-ups obtained when fixing the number of CPUs to 128 and scaling the number of GPUs from 1 to 16. We can see that the 2MBB approach is still scaling in a linear manner while being near optimal. This is to contrast to the static steal-half strategy which is substantially outperformed by the 2MBB approach. Similarly, the 2MBB approach is still scaling in a relatively efficient manner when fixing the number of GPUs to 1 (resp. 2) and scaling the number of CPUs from 1 to 256 as depicted in Fig. 5 bottom-left (resp. bottom-right). Surprisingly, the performance of the steal-half strategy gets even worse in the scenario where few CPUs are additionally used with GPUs compared to the scenario where only the GPUs are involved. We can in fact remark that improvements over 1 or 2 GPUs are only observed when the number of CPUs is relatively high (w.r.t GPU power). To understand the issues we are facing when considering such heterogeneous compute configurations, let us recall that a GPU is substantially faster in evaluating B&B tree nodes than a CPU and the B&B tree search is highly irregular. Hence, if GPUs run out of work and stay idle searching for work, the performance of the system can drop dramatically. If only few CPUs are available in the system, work stealing operations from CPUs to GPUs can cause a severe penalty to performance. This is because the few CPUs can only contribute very little to the overall performance but their stealing operations to GPUs can disturb the GPU computations and prevent them from reaching their maximal speed. In contrast, if work is scheduled more on GPUs, then a significant loss in performance can occur when a relatively large number of CPUs are available. From the previous set of experiments, we can conclude that the 2MBB is able to solve these issues relatively well and to take advantage of both the GPU and the CPU components involved in the distributed system.

Nevertheless, at the largest scale of 256 compute cores we remark that the speed-up of 2MBB is drifting from the linear one. We attribute this to the penalty that the 2MBB approach has to pay for the delay to transfer work when the scale of the distributed systems becomes too large. In fact, as we scale the distributed system work start to become more fine grain and it is less likely to fetch work quickly which increases the cost of message passing among the distributed compute units. In the next section, we shall show that the 3MBB architecture can handle this scalability issue in a relatively efficient manner by fully exploring the shared memory parallelism exposed by large scale compute environments.

6.4 Performance of the 3MBB approach

Let us remind that in the large scale setting the application granularity might be broken into very fine-grained which pushes more challenges in order to achieve high performance. Fine-grain parallelism together with the irregularity of the B&B search can make PUs search more frequently for work, thus reducing the utilization of PUs in computations. The objective of our second set of experiments is to evaluate the gain we can expect from the 3MBB design compared to 2MBB when dealing with a large scale heterogeneous and distributed system endowed with shared memory multi-cores.

6.4.1 Relative scalability of 3MBB and 2MBB

In Fig 6, we analyze the scalability of 3MBB compared to 2MBB in different heterogeneous scenarios where the number of GPUs is fixed in the range $\{0, 1, 2, 4\}$ and the CPU-cores are scaled from 1 up to 512. The results clearly show that the 3MBB approach significantly improves the performance of 2MBB. More precisely, the performance of the two approaches is equivalent in small and average scales (up to 128 CPU cores) where they both scale near linearly. In small and average scales, B&B workload is relatively not very fine-grained, hence decreasing the need for PUs to steal work units more frequently. This results in increasing the utilization of PUs in computations and consequently in better speedups. However, the performance of 2MBB starts to drop in the scale of 256 and 512 CPU-cores and it is substantially outperformed by the 3MBB approach. This is clearly attributed to the advanced and hybrid load balancing mechanism used in 3MBB which allows us to take full advantage of the different levels of parallelism exposed by the compute environment. In fact, balancing B&B workload locally at every shared memory component allows PUs to significantly reduce the idle times and the cost of stealing using message passing. We shall analyze this aspect in more details in the next subsection.

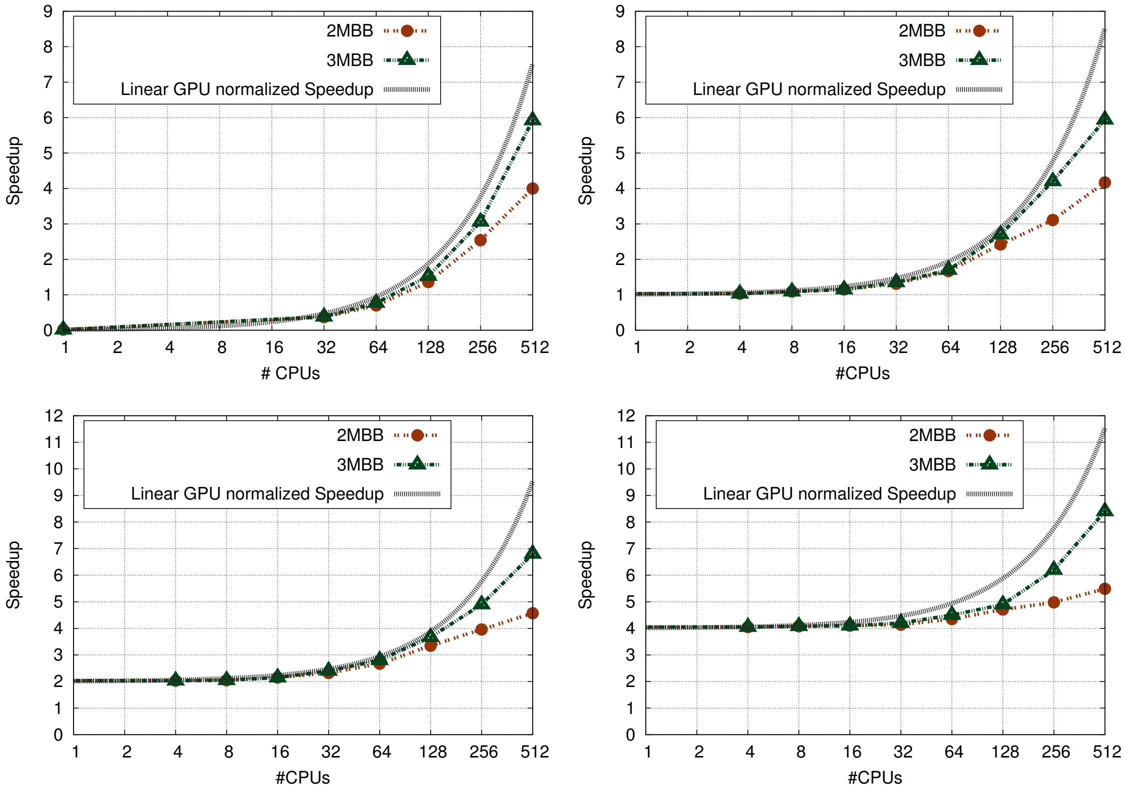

Figure 6: Speedup of the 3MBB approach compared to 2MBB when scaling CPUs and using 0 GPU (**Top-Left**), 1 GPU (**Top-Right**), 2 GPUs (**Bottom-Left**) and 4 GPUs (**Bottom-Right**). The X-axis is in the log scale and reported speed-ups are w.r.t. one GPU. The number of concurrent CUDA streams is set to 10.

6.4.2 Large scale analysis

To appreciate the gain we can obtain when executing parallel B&B with 3MBB, we consider to solve the ten Flowshop instances in the scenario where no GPUs are used but only multiple CPUs with multi-core systems at the largest scale of 512 cores. The results reported in Fig. 7 clearly show that the 3MBB significantly improves the 2MBB by 30% (for instance Ta$_{23}$) up to 50% (for instance Ta$_{30}$). Notice that according to Fig. 4, instance Ta$_{30}$ (resp. Ta$_{23}$) is likely to expose the most (resp. least) fine-grained parallelism.

Table 1 allows us to get more insights on the impact of the shared memory parallelism incorporated in the 3MBB approach. More precisely, we provide different statistics concerning the time where a PU stays idle searching for work. Firstly, we can see (last column of Table 1) that the proportion of time that a PU spend requesting for work over all B&B execution is significantly smaller for 3MBB (6.79 % in average and over all instances) compared to 2MBB (32.88 %). This indicates that the 3MBB allows us to maximize useful B&B computations over work transfer. This also shows that balancing B&B work efficiently among PUs is the critical issue when considering fine-grain large scale scenarios. Secondly, we can see that much less inter-node communications are performed by 3MBB compared to 2MBB (by a factor of 40.5 times). In fact, only the leader threads are allowed to perform inter-node steals in contrast to 2MBB where all individual threads execute inter-node steals. Not only the number of inter-node steals is significantly smaller for 3MBB, but they are also much less time consuming (by a factor of 3.43 times). This is attributed to the fact that a PU in 2MBB is likely to receive more work requests simultaneously, thus inducing more delays when responding them. Secondly, we can see that the intra-node steals are prevailing over the of inter-node ones which illustrates how the 3MBB approach prioritizes intra-node steals over inter-node steals. Since the cost of intra-node communication is relatively low and the design of the split work pool allows us to minimize the cost of thread synchronizations, the shared-memory computations

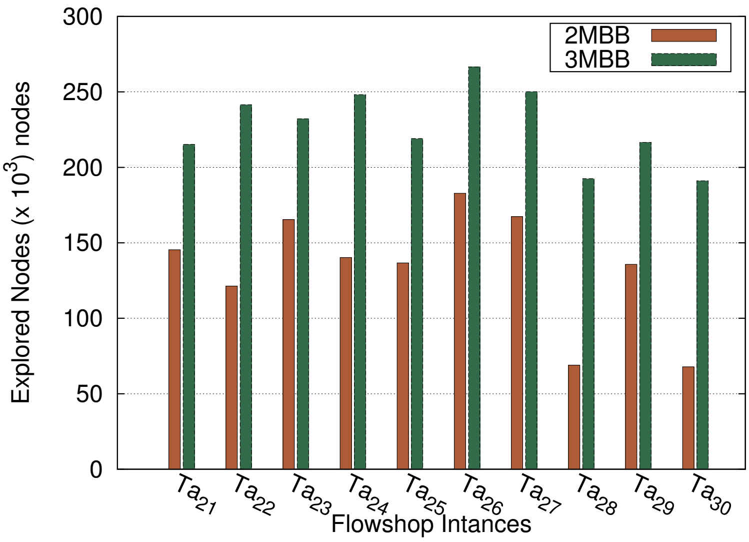

Figure 7: Average number of B&B nodes explored per second for the 3MBB and the 2MBB approaches using 512 CPU-cores.

are optimized and the B&B tasks are balanced and offloaded very efficiently among the shared memory CPU-cores. On the other side, inter-node steals are much slower than intra-node steals; but they are very important to balance work load among distributed compute nodes. Let us remind that 3MBB tries to balance workload adaptively based on the aggregated power of available cores. A leader thread in 3MBB always tries to fetch enough tasks in order to serve all the threads sharing the same memory. Since one single leader is performing remote steals on behalf of many others, the scale of the distributed system is then artificially reduced to relatively few leaders and the inter-node communication is likely to be much more efficient.

The previous observations also hold when considering a large scale scenario incorporating GPUs devices as reported in Table 2 for the first Flowshop instance Ta_{21}. We can see that 3MBB is not sensitive to the number of GPUs devices and it is again able to reduce the overall communication cost; and consequently, the penalty one have to pay when balancing B&B irregular workload distributively. However, notice from Fig. 6 that although the stealing time in 3MBB is minimized, there is still a gap between the actual speedup of 3MBB and the linear one at the largest scale. We attribute this to the combined effect of two facts: (i) a single GPU device can functions at its maximum speed only when there is enough B&B subproblems to push in, and (ii) the available B&B work gets more and more fine-grained and scattered over PUs at such large scales. Hence, although the GPUs devices do not stay idle, it is more likely that they cannot acquire enough B&B subproblems to bound in parallel and thus does not functions at their maximum speed. In other words, since work becomes more fine-grained, the number of B&B subproblems that are pushed inside the GPUs is not optimal to allow the GPUs to attain their maximum speedups. Solving this issue efficiently is left as an open question which would deserve further investigations. Overall, we argue that the 3MBB approach allows us to push the distributed system to its extremes and to take as much benefit as possible from the different levels of parallelism afforded by the compute environment.

7 Conclusion

In this paper, we investigated hybrid shared memory and distributed schemes for parallelizing B&B computations in heterogeneous systems, where multi-core nodes, multiple distributed CPUs and GPUs are used. The following two approaches have been proposed and experimented in various system configurations:

Table 1: Basic steal statistics for the 3MBB and the 2MBB approaches when executing the ten Taillard' instances using 512 CPU-cores. Notice that the intra-node parallelism holds only for 3MBB. The last column gives the percentage of execution time that a PU spends in stealing.

		Inter-node			**Intra-node**			**% of steal time**
		# Steals	**Time (s)**	**Time per steal (s)**	**# Steals**	**Time (s)**	**Time per steal (s)**	
Ta_{21}	2MBB	1831	115.18	0.063	N/A			26.57%
	3MBB	226	3.59	0.016	3351813	5.69	0.0000017	3.96%
Ta_{22}	2MBB	1926	154.36	0.080	N/A			38.01%
	3MBB	193	2.43	0.013	3219090	5.47	0.0000017	2.78%
Ta_{23}	2MBB	3897	263.14	0.067	N/A			22.56%
	3MBB	153	3.02	0.019	3342713	5.68	0.0000017	0.89%
Ta_{24}	2MBB	1336	98.83	0.074	N/A			32.24%
	3MBB	145	6.60	0.045	2939638	5	0.0000017	5.47%
Ta_{25}	2MBB	2921	208.20	0.071	N/A			30.96%
	3MBB	195	4.13	0.021	4519199	7.23	0.0000016	2.23%
Ta_{26}	2MBB	3387	221.17	0.065	N/A			24.95%
	3MBB	168	2.90	0.017	4053746	6.89	0.0000017	2.20%
Ta_{27}	2MBB	3002	187.90	0.062	N/A			28.32%
	3MBB	177	2.62	0.015	3833216	6.516	0.0000017	3.26%
Ta_{28}	2MBB	830	76.54	0.090	N/A			47.93%
	3MBB	177	5.62	0.032	3053185	5.49	0.0000018	15.64%
Ta_{29}	2MBB	1323	95.38	0.072	N/A			28.35%
	3MBB	148	2.88	0.019	2649031	4.77	0.0000018	10.92%
Ta_{30}	2MBB	563	58.35	0.103	N/A			48.94%
	3MBB	127	2.74	0.021	2023738	3.64	0.0000018	20.58%

- The 2MBB approach targets completely distributed CPUs and GPUs. This approach deals with two-level of parallelism allowing for (i) distributed subtree exploration among PUs and (ii) concurrent operations between every single GPU host and device. An adaptive work transfer scheme for sharing works based on PUs' computing power is also proposed. Despite that this approach follows relatively simple design principles, it appears to be extremely effective when parallelize B&B computations and considering relatively low and moderate distributed scales. In particular, near-linear speed-ups can be obtained. At the largest scale however, the 2MBB suffers from distributed communication cost and does not allow to fully use the power offered by the distributed system.

- The 3MBB approach extends the 2MBB approach for multi-cores heterogeneous systems. This approach deals with the hardware hierarchy (shared vs. distributed memory) in order to minimize communication cost. Within a multi-core system, decentralized split work pools are used to share B&B problems, and intra-node stealing operations are performed in order to acquire work efficiently while avoiding shared memory locking and synchronization issues. Prioritizing intra-node communication over inter-node distributed operations is found to be extremely crucial in oder to push the distributed system to its limits. Our experimental results show up to 50% improvement compared to the 2MBB approach .

We also showed that the B&B granularity can have a big impact on performance. While our approaches have near-linear speedups at reasonable scales, there still might be room for improvements when considering fine-grained B&B instances and large scale heterogeneous systems. In particular, designing new distributed protocols in oder to fully take advantage of the power offered by the GPU is worth to be investigated in the future. Furthermore, it is our hope that the lessons learnt from this study help the design of new distributed and parallel B&B algorithms taking into account other types of compute devices such as those integrating many more compute cores within specific shared memory hardwares.

Table 2: Basic steal statistics for the 3MBB and the 2MBB approaches when executing the Ta$_{21}$ instance using 512 CPU-cores and variable number of GPUs. Notice that the intra-node parallelism holds only for 3MBB. The last column gives the percentage of execution time that a PU spends in stealing.

		Inter-node			**Intra-node**			**% of steal time**
		# Steals	**Time (s)**	**Time per steal (s)**	**# Steals**	**Time (s)**	**Time/Steal (s)**	
0 GPU	2MBB	1831	115.18	0.063	N/A			26.57%
	3MBB	226	3.59	0.016	3351813	5.69	0.0000017	3.96%
1 GPU	2MBB	1554	101.81	0.065	N/A			32.02%
	3MBB	151	5.83	0.038	3112522	4.980	0.0000016	4.23%
2 GPUs	2MBB	1459	110.34	0.075	N/A			36.64%
	3MBB	166	2.54	0.015	3845257	6.15	0.0000016	4.75%
4 GPUs	2MBB	1090	96.13	0.088	N/A			40.72%
	3MBB	199	3.43	0.017	3955168	6.72	0.0000017	5.97%

References

[1] K. Aida and Y. Futakata. High-performance parallel and distributed computing for the bmi eigenvalue problem. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 71–78, April 2002.

[2] K. Aida and T. Osumi. A case study in running a parallel branch and bound application on the grid. In *Applications and the Internet, 2005. Proceedings. The 2005 Symposium on*, pages 164–173, Jan 2005.

[3] Lucio Barreto and Michael Bauer. Parallel branch and bound algorithm - a comparison between serial, openmp and mpi implementations. *Journal of Physic: Conference series*, 256(1):012018, 2010.

[4] A. Bendjoudi, N. Melab, and E-G. Talbi. An adaptive hierarchical master-worker framework for grids: Application to B&B algorithms. *J. Parallel Distrib. Comput (JPDC)*, 72(2):120–131, 2012.

[5] A. Bendjoudi, N. Melab, and E. G. Talbi. Hierarchical branch and bound algorithm for computational grids. *Future Gener. Comput. Syst.*, 28(8):1168–1176, October 2012.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, 1999.

[7] A. Boukedjar, M.E. Lalami, and D. El-Baz. Parallel branch and bound on a CPU-GPU system. In 20th *Int. Conf. on Parallel, Distributed and Network-Based Processing*, pages 392–398, 2012.

[8] Benoît Bourbeau, Teodor Gabriel Crainic, and Bernard Gendron. Branch-and-bound parallelization strategies applied to a depot location and container fleet management problem. *Parallel Comput.*, 26(1):27–46, January 2000.

[9] Mihai Budiu, Daniel Delling, and Renato F. Werneck. Dryadopt: Branch-and-bound on distributed data-parallel execution engines. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1278–1289, Washington, DC, USA, 2011.

[10] T Carneiro, A. E. Muritiba, M. Negreiros, L. De Campos, and G. Augusto. A new parallel schema for branch-and-bound algorithms using GPGPU. In 23rd *Symp. on Computer Architecture and High Performance Computing*, pages 41–47, 2011.

[11] Tiago Carneiro, Ricardo Nobre, Marcos Negreiros, and Gustavo Augusto Lima de Campos. Depth-first search versus jurema search on gpu branch-and-bound algorithms: a case study. In *XXXII Congresso da Sociedade Brasileira de Computacao CSBC 2012*, 2012.

[12] L. G. Casado, J. A. Martinez, I. Garcia, and E. M. T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods Software*, 23(5):689–701, October 2008.

[13] I. Chakroun and M. Melab. An adaptative multi-GPU based branch-and-bound. a case study: the flow-shop scheduling problem. In 14^{th} *IEEE Inter. Conf. On High Performance Computing and Communications*, 2012.

[14] I. Chakroun and N. Melab. Towards a heterogeneous and adaptive parallel branch-and-bound algorithm. *Journal of Computer and System Sciences*, 2014.

[15] Imen Chakroun. *Algorithmes Branch and Bound parallèles hétérogènes pour environnements multicoeurs et multi-GPU*. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, June 2013.

[16] Imen Chakroun and Nordine Melab. Operator-level gpu-accelerated branch and bound algorithms. In *International Conference on Computational Science (ICCS)*, pages 280–289, 2013.

[17] Imen Chakroun, Nordine Melab, Mohand-Said Mezmaz, and Daniel Tuyttens. Combining multi-core and GPU computing for solving combinatorial optimization problems. *J. Parallel Distrib. Comput.*, 73(12):1563–1577, 2013.

[18] E. W. Dijkstra. Derivation of a termination detection algorithm for distributed computations. *Control Flow and Data Flow: concepts of distributed programming*, pages 507–512, 1987.

[19] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *ACM Conference on High Performance Computing Networking, Storage and Analysis (SC09)*, pages 53:1–53:11, 2009.

[20] Maciej Drozdowski, Pawe Marciniak, Grzegorz Pawlak, and Maciej Paza. Grid branch-and-bound for permutation flowshop. In *Parallel Processing and Applied Mathematics*, volume 7204 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2012.

[21] J.F.S. Estrada, L. G. Casado, and I Garcia. Adaptive parallel interval global optimization algorithms based on their performance for non-dedicated multicore architectures. In *19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 252–256, Feb 2011.

[22] Yuri Evtushenko, Mikhail Posypkin, and Israel Sigal. A framework for parallel large-scale global optimization. *Computer Science - Research and Development*, 23(3-4):211–215, 2009.

[23] Matteo F, Charles E. L, and Keith H. R. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, 1998.

[24] B. Gendron and T.G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[25] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 43–50, 2000.

[26] J.-P. Goux, Jeff Linderoth, and Michael Yoder. Metacomputing and the master-worker paradigm. In *Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne*, 2000.

[27] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, 1999.

[28] Grid500 French national gird. https://www.grid5000.fr/.

[29] Juan F. R. Herrera, Leocadio G. Casado, Eligius M. T. Hendrix, Remigijus Paulavicius, and Julius Zilinskas. Dynamic and hierarchical load-balancing techniques applied to parallel branch-and-bound methods. In *3PGCIC'13*, pages 497–502, 2013.

[30] Juan F.R. Herrera, Leocadio G. Casado, Remigijus Paulavicius, Julius ilinskas, and Eligius M.T. Hendrix. On a hybrid mpi-pthread approach for simplicial branch-and-bound. *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, 0:1764–1770, 2013.

[31] S.M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[32] S. Kouki, M. Jemni, and T. Ladhari. Scalable distributed branch and bound for the permutation flow shop problem. In *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 503–508, Oct 2013.

[33] Samia Kouki, Mohamed Jemni, and Talel Ladhari. Deployment of solving permutation flow shop scheduling problem on the grid. In Tai-hoon Kim, StephenS. Yau, Osvaldo Gervasi, Byeong-Ho Kang, Adrian Stoica, and Dominik Slezak, editors, *Grid and Distributed Computing, Control and Automation*, volume 121 of *Communications in Computer and Information Science*, pages 95–104. Springer Berlin Heidelberg, 2010.

[34] M. E. Lalami and D. El-Baz. GPU implementation of the branch and bound method for knapsack problems. In *IPDPS Workshops*, pages 1769–1777, 2012.

[35] AilsaH. Land and AlisonG. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer Berlin Heidelberg, 2010.

[36] J.K. Lenstra, B.J. Lageweg, and A.H.G.R. Kan. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[37] Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. Work stealing strategies for multi-core parallel branch-and-bound algorithm using factorial number system. In *Programming Models and Applications on Multicores and Manycores (PMAM)*, pages 111:111–111:119. ACM, 2014.

[38] N. Melab. Contribution a la resolution de problemes dóptimisation combinatoire sur grilles de calcul. Habilitation a diriger des recherches de lúniversité lille1, Université Lille 1, CNRS UMR8022, Inria, Lille, France, 2005.

[39] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens. A GPU-accelerated b&b algorithm for the flow-shop scheduling problem. In 14^{th} *IEEE Conf. on Cluster Computing*, 2012.

[40] M. Mezmaz, R. Leroy, N. Melab, and D. Tuyttens. A multi-core parallel branch-and-bound algorithm using factorial number system. In *IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1203–1212, May 2014.

[41] M. Mezmaz, N. Melab, and D. Tuyttens. A multithreaded branch-and-bound algorithm for solving the flow-shop problem on a multicore environment. In H. Sarbazi-Azad and A. Y. Zomaya, editors, *Large Scale Network-Centric Distributed Systems*, chapter 3. John Wiley & Sons, Inc., Hoboken, New Jersey, 2013.

[42] M.-S. Mezmaz, N. Melab, and E.-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–9, March 2007.

[43] S-J Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In 5^{th} *Conf. on Partitioned Global Address Space Prog. Models*, 2011.

[44] Lars Otten and Rina Dechter. Load balancing for parallel branch and bound. In *10th Workshop on Preferences and Soft Constraints*, pages 51–65, 2010.

[45] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In 16^{th} *ACM Symp. on Principles and practice of parallel programming (PPoPP '11)*, pages 201–212, 2011.

[46] Abdorreza Savadi and Hossein Deldari. A bridging model for branch-and-bound algorithms on multi-core architectures. In *Proceedings of the 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '12, pages 235–241, Washington, DC, USA, 2012. IEEE Computer Society.

[47] J.M.N. Silva, C. Boeres, L.M.A. Drummond, and A.A. Pessoa. Memory aware load balance strategy on a parallel branch-and-bound application. *Concurrency and Computation: Practice and Experience*, April 2014.

[48] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[49] H.W.J.M. Trienekens. Parallel branch and bound on an mimd system. Technical report, Econometric Institute, Erasmus University, Rotterdam, Netherlands, 1986.

[50] H.W.J.M. Trienekens and A. Bruin. Towards a taxonomy of parallel branch and bound algorithms. Technical report, Econometric Institute, Erasmus University, Rotterdam, Netherlands, 1992.

[51] Trong-Tuan Vu, Bilel Derbel, Asim Ali, Ahcene Bendjoudi, and Nouredine Melab. Overlay-centric load balancing: Applications to uts and b&b. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 382–390, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[52] Trong-Tuan Vu, Bilel Derbel, and Nouredine Melab. Adaptive dynamic load balancing in heterogeneous multiple gpus-cpus distributed setting: Case study of b&b tree search. In *7th International Conference on Learning and Intelligent Optimization (LION)*, pages 87–103, 2013.

[53] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Alps: A framework for implementing parallel search algorithms. In *In Proceedings of the Ninth INFORMS Computing Society Conference*, pages 319–334, 2005.