A Chapel-Based Multi-GPU Branch-and-Bound Algorithm

Guillaume Helbecque, Ezhilmathi Krishnasamy, Tiago Carneiro, Nouredine Melab, Pascal Bouvry

A Chapel-based Multi-GPU Branch-and-Bound Algorithm

Application to the Flowshop Scheduling Problem

Guillaume Helbecque[a,b], Ezhilmathi Krishnasamy[a], Tiago Carneiro[c], Nouredine Melab[b], Pascal Bouvry[a]

[a] *Université du Luxembourg, DCS-FSTM/SnT, Luxembourg*
[b] *Université de Lille, CNRS/CRIStAL, Centre Inria de l'Université de Lille, France*
[c] *Interuniversity Microelectronics Centre (IMEC), Belgium*

Abstract

The increasing heterogeneity and diversity of modern supercomputers brings, along with the heterogeneity challenge, both the code and performance portability issues. In this context, the PGAS-based Chapel programming language comes as a solution for both the heterogeneity and portability challenges, as it provides vendor-agnostic GPU support. In this paper, we deal with the design and implementation of a multi-GPU Branch-and-Bound algorithm for solving combinatorial optimization problems. The contribution consists of a generic multi-pool data structure coupled with a dynamic load balancing mechanism based on work stealing. While the CPU cores are used to perform a parallel tree exploration, GPU devices are used to accelerate the bounding phase, which is particularly compute intensive. Extensive experiments on the Permutation Flowshop Scheduling Problem (PFSP) reveal that the proposed Chapel-based approach can achieve a strong scaling efficiency of up to 63% and 75% on average using a GPU-powered processing node including 8 NVIDIA A100 devices and AMD MI50 GPUs, respectively. This demonstrates the efficiency of our approach to solving large PFSP instances, while ensuring code portability.

Keywords: GPU-aware Heterogeneity, GPU-native Support, Chapel, Parallel Branch-and-Bound, Flowshop Problem, PGAS

1. Introduction

The Branch-and-Bound (B&B) tree-based search is one of the most used algorithms for solving Combinatorial Optimization Problems (COPs) instances to the optimality. Given their inherently parallel nature, this class of algorithms has been effectively parallelized for exploiting several different High-Performance Computing (HPC) architectures over the years [1, 2]. In this context, to meet the growing demands for HPC, modern supercomputers are becoming increasingly large and sophisticated.

According to the most recent TOP500 list[1], modern supercomputers include and combine a wide variety of compute devices, such as CPUs, GPUs, and communication networks, which add complexity to software development for these platforms. Indeed, HPC applications have historically been associated with a complex combination of programming models, such as shared memory, distributed memory, and GPU computing models, *e.g.*, OpenMP, MPI, and CUDA, respectively. It is also important to point out that GPU programming Application Programming Interfaces (APIs) tend to be vendor-specific, such as NVIDIA CUDA and AMD ROCm.

In this context, the Chapel programming language [3] comes as an alternative to deal with the complexity in HPC. With one language, one could program all levels of parallelism of a computer node, from the single-threaded level to the multi-core and distributed one. Moreover, Chapel also implements the Partitioned Global Address Space (PGAS) model, which unifies the memory of the compute cluster through a software layer, providing a way of accessing remote data similar to the shared memory model. Finally, Chapel recently added GPU-native and vendor-agnostic support, which tries to solve both the heterogeneity and portability issues in HPC.

In this paper, we propose a PGAS-based multi-GPU B&B algorithm for solving COPs to the optimality that is based on Chapel and its GPU-native support. To the best of our knowledge, this is the first GPU-accelerated B&B approach designed using PGAS and implemented using a single parallel programming language. It consists of a generic multi-pool data structure coupled with a dynamic load-balancing mechanism based on work stealing. While the CPU cores are used to perform a parallel tree exploration, GPU devices are used to accelerate the bounding phase, which is particularly compute intensive. The unified view of the inter-node (distributed) and intra-node (shared) parallel levels provided in Chapel is not investigated in this paper, the focus being put on the GPU-native support to deal with the heterogeneity and portability challenges.

Extensive experiments on the Permutation Flowshop Scheduling Problem reveal that the implementation can achieve a strong scaling efficiency of up to 63% and 75% on average using a GPU-powered processing node including 8 NVIDIA A100 devices and AMD MI50 GPUs, respectively. This demonstrates the efficiency of our approach to solving large COP instances, while ensuring code portability. Moreover, this application serves as a single-node baseline for a future PGAS-

[1] TOP500 ranking of supercomputers worldwide (June 2024), see `https://www.top500.org/lists/top500/2024/06/`.

based multi-node version.

The remainder of this paper is organized as follows. Section 2 provides background on GPU-accelerated B&B algorithms, along with some related work. Section 3 presents the design and implementation of the proposed algorithm in Chapel. Then, Section 4 provides an experimental evaluation of the latter. Finally, Section 5 draws the conclusions and highlights the future works.

2. Background and Related Work

2.1. The Branch-and-Bound method

The Branch-and-Bound (B&B) algorithm systematically explores the solution space by constructing a tree where each node represents a partial solution (subproblem), and edges represent the extension of partial solutions. More precisely, it breaks down the main problem into several smaller pairwise disjoint subproblems (branching) and evaluates their potential to contain an optimal solution (bounding). If a subproblem cannot yield a better solution than the best one found so far, it is pruned, reducing the number of nodes to explore and improving computational efficiency. However, the number of child nodes resulting from a node decomposition varies, making the explored search trees highly irregular and unpredictable.

B&B can employ different strategies to traverse the tree. Depth-First Search (DFS) explores a branch as far down as possible before backtracking, which can be memory efficient but may not find the optimal solution quickly. Breadth-First Search (BFS), on the other hand, explores all nodes at the current depth level before moving on to nodes at the next depth level, ensuring the shortest path to a solution but at a higher memory cost. In this work, we combine both DFS and BFS.

2.2. Parallel GPU-accelerated B&B

The inherently parallel nature of B&B algorithms allows for effective parallelization on modern HPC architectures, including multi-core CPUs and GPU accelerators. The most general and widely used model to design parallel B&B algorithms on multi-core CPUs is the parallel tree-exploration model [4]. Specifically, each CPU core independently explores a sub-tree and determines whether tree nodes should be pruned or expanded. On the other hand, the bounding step, which involves computing upper and lower bounds for subproblems, can be computationally intensive. GPUs, with their massively parallel architecture, are particularly well-suited for the parallel bounding of nodes, enabling the simultaneous processing of numerous independent subproblems.

This work adopts a collegial multi-pool approach, in which each CPU core manages its own pool to store generated but not yet evaluated subproblems [1]. This approach alleviates the bottleneck problem that occurs in single-pool approaches but raises the issue of balancing the workload between multiple pools. A popular and provably efficient approach to dynamic load balancing is the Work Stealing (WS) paradigm [5]. Under WS, each CPU core usually maintains a double-ended queue (deque) of nodes and processes nodes from the tail of its deque or steals work units from the head of another deque when its own work pool is empty.

2.3. Related Works

GPU-accelerated B&B algorithms have demonstrated substantial performance improvements in various applications, including job scheduling and resource allocation problems [6, 7, 8]. By exploiting the parallel processing power of GPUs, these algorithms can handle larger problem instances and provide faster solutions compared to traditional CPU-based implementations. More advanced approaches have further enhanced the efficiency and scalability of B&B algorithms, achieving near-linear speed-ups and optimized performance on large-scale systems [9, 10].

While these approaches allow significant performance improvements, they are implemented using a complex combination of low-level programming environments and vendor-specific GPU programming APIs, such as NVIDIA CUDA and AMD ROCm. In contrast, this work focuses on alternative programming languages, such as the PGAS-based Chapel, that promote vendor-agnostic GPU support. In this context, only few works investigate GPU-accelerated tree-search algorithms.

In [11], the authors implemented a GPU-based backtracking algorithm in Chapel. The proposed algorithm exploits Chapel's iterators by combining a partial search strategy with pre-compiled CUDA kernels for more efficient exploitation of the intra-node parallelism. Helbecque *et al.* [12] revisited the design and implementation of a generic multi-pool GPU-accelerated backtracking algorithm using Chapel. Reported results on the N-Queens problem show that the high GPU abstraction of Chapel results in a performance loss of only 8% (resp. 16%) compared to CUDA (resp. HIP) in a single-GPU setting. Furthermore, we achieve up to 80% (resp. 71%) of the baseline speed-up for coarse-grained instances on NVIDIA (resp. AMD) GPUs.

While [11, 12] investigated proof-of-concept approaches motivating further improvements in solving related COPs, this paper investigates the exact resolution of COPs using a multi-GPU B&B algorithm in Chapel.

3. Multi-GPU B&B using Chapel

This section details the proposed multi-GPU B&B in Chapel. It consists of a generic multi-pool data structure coupled with a dynamic load-balancing mechanism based on WS. While the CPU cores are used to perform a parallel tree exploration, GPU devices are used to accelerate the bounding phase, which is particularly compute intensive. The approach is designed to be problem-independent. The main algorithm can be divided into four main steps: initial partial search, multi-GPU search, dynamic load balancing, and termination criteria, which are detailed as follows.

3.1. Initial partial search

Preliminary to the multi-GPU search, we perform an initial partial exploration of the tree. The aim of this initial search is

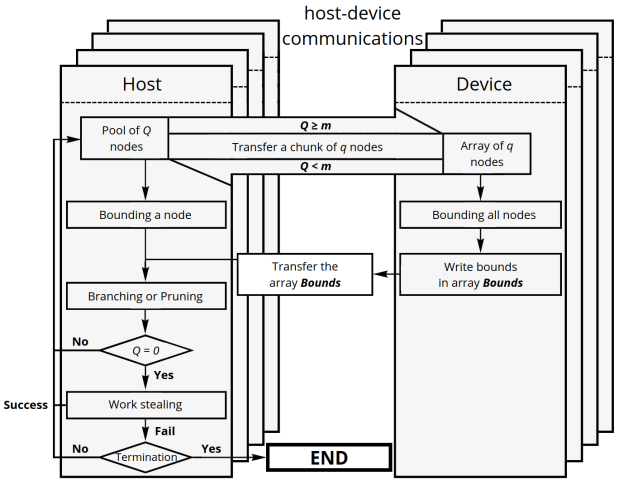

Figure 1: Flowchart of the proposed multi-GPU B&B algorithm.

Algorithm 1: Pseudo-code of the WS mechanism

```
input : multiPool - multi-pool data structure
        D - number of tasks
        trials - maximum number of WS attempts
1 while (trials < D) do
2     randomly select the index of a "victim" task;
       /* called victimID */
3     acquire the lock of multiPool[victimID];
       /* spin-lock */
4     if multiPool[victimID] has enough work then
5         stealf half of the available workload;
6         release lock of multiPool[victimID];
7         exit WS mechanism;
8     release lock of multiPool[victimID];
9     increment trials by 1;
```

to provide a sufficiently large amount of nodes to each GPU to prevent GPU starvation at the beginning of the parallel search. Indeed, GPUs are composed of thousands of cores, which require a substantial number of tasks to maintain high utilization and efficiency. By conducting this initial search, we ensure that each GPU receives enough nodes to process, thereby maximizing their computational capabilities and avoiding idle time during the parallel search. The search is performed using BFS, which allows for a broad and uniform distribution of nodes. At the end of the initial search, the workload is evenly spread among pools in a Round-Robin fashion.

3.2. Multi-GPU search

Figure 1 shows the flowchart of the multi-GPU search. Each CPU core (host) maintains a pool of nodes and successively: (1) get a node from the pool, (2) bound it, (3) branch or prune it according to the best solution found so far, and (4) evaluate the termination of the algorithm. Pools are implemented as double-ended queues and are maintained in a DFS order. After several iterations, pools may contain a large amount of nodes and the GPU devices allow their efficient parallel bounding. More precisely, when a pool contains at least m nodes, then a chunk of $q = \min(Q, M)$ nodes is copied from the host to the device, where Q is the pool size. The GPU device then performs the concurrent bounding of subproblems and sends back the results to the host. The m and M parameters are used to determine the minimum number of nodes required for efficient GPU processing and to set the maximum chunk size for data transfer, respectively. This ensures optimal use of GPU resources and minimizes communication overhead. Because of the irregularity of B&B, a pool may become empty during parallel execution, even if the tree exploration is not completely done. In that case, a dynamic WS mechanism is triggered.

3.3. Dynamic load balancing

Algorithm 1 shows the pseudo-code of the WS mechanism. First of all, we randomly select a victim task (line 2) and acquire the lock of its pool (line 3). Indeed, to ensure the parallel-safety of the approach, each pool is equipped with a spin-lock. This synchronization mechanism repeatedly checks a lock variable in a tight loop (spins) until the lock becomes available, making it efficient for short wait times but CPU-intensive. A common implementation uses atomic operations like `compare-and-swap` to manage the lock state. Then, if the pool contains enough nodes (line 4), we steal half of the available workload and release the lock (lines 5-6). More precisely, to avoid stealing too many nodes and creating implicit load imbalance, we ensure that the victim pool contains at least $2m$ nodes. This way, the steal-half strategy ensures that both the thief and victim pools contain at least m nodes after stealing, which is sufficient to further process the tree exploration on GPU. If WS cannot be done, another victim task is chosen. In particular, we choose another victim task that has not been visited in previous iterations. The WS mechanism ends when a victim is found (line 7), or all pools have been visited (line 1). In the latter case, the termination detection is checked.

3.4. Termination detection

The termination of the algorithm is reached when there is a consensus between tasks. Algorithm 2 illustrates the detection mechanism. Each task owns a globally visible `state` atomic variable containing its current status: busy or idle. This variable is updated each iteration (lines 2-3,5-6) and its atomicity ensures that checks and updates prevent race conditions. When the WS mechanism fails during execution, the termination is checked (line 4). In that case, the calling task checks the status of the other tasks: if they are all idle the algorithm ends (line 8), otherwise the tree exploration continue (line 9).

4. Performance Evaluation

Section 4.1 presents the experimental protocol as well as the experimental testbed. Then, Section 4.2 describes the application context. Finally, experimental results are discussed in Section 4.3.

Algorithm 2: Pseudo-code of the termination detection mechanism

```
input : states - array of task states
        taskID - global index of the calling task
1 if task taskID is busy or work stealing succeeded then
2     if (states[taskID] is idle) then
3         set states[taskID] to busy;
4 else
5     if (states[taskID] is busy) then
6         set states[taskID] to idle;
7     if allTasksIdle(states) then
8         break tree exploration;
9 continue tree exploration;
```

4.1. Experimental protocol and testbed

The Grid'5000 large-scale testbed[2] is used and both NVIDIA and AMD GPU architectures are considered:

- *AMD MI50*: one 48-core AMD EPYC 7642 (Zen 2) @ 2.3 GHz and 512 GiB memory, equipped with eight AMD Radeon Instinct MI50 (32 GB HBM2 memory and 3840 stream processors @ 1200 MHz);
- *NVIDIA A100*: two 64-core AMD EPYC 7742 (Zen 2) @ 2.25 GHz and 1.0 TiB memory, equipped with eight NVIDIA A100 SXM4 (40 GB HBM2 memory and 6912 stream processors @ 1095 MHz).

The code is based on Chapel 2.0.1 in a fine-tuned configuration environment, along with CUDA 12.0.0 and ROCm 5.2.0 for GPU support on the NVIDIA and AMD architectures, respectively. The following B&B implementations are considered: (1) the multi-GPU B&B algorithm in Chapel outlined in Section 3, and (2) an optimized version of the latter algorithm for single-GPU execution. To encourage reproducibility, the code can be found in the open-source GitHub repository of the authors [13]. However, given the scarcity of open-source counterpart implementations in the current literature, a direct comparison with other existing approaches has not been explored in this paper.

In the experimental evaluation, the performance of the multi-GPU B&B implementation is measured in terms of processing rate, that is, decomposed nodes per second (kn/s, *i.e.*, 10^3 nodes/second), and speed-up. In order to study the performance of the algorithm in the absence of speed-up anomalies, we always initialize the algorithm with the optimal solution of the instance to be solved. With this initialization, the algorithm proves the optimality of the initial upper bound and it is ensured that all parallel executions explore exactly the same critical tree, *i.e.*, all nodes for which the bounding operator gives a lower bound smaller than the optimal solution.

[2]Grid'5000 documentation: `https://www.grid5000.fr/w/Grid5000:Home`.

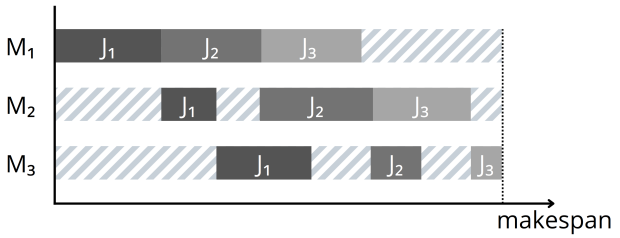

Figure 2: Solution of a PFSP instance consisting of 3 jobs and 3 machines.

4.2. The Permutation Flowshop Scheduling Problem (PFSP)

The PFSP consists of finding an optimal processing order (a permutation) for n jobs $\{J_1, \ldots, J_n\}$ on r machines $\{M_1, \ldots, M_r\}$, such that the total completion time (makespan) is minimized. Obeying a chain production principle, the processing of a job J_j on machine M_k can only start if the processing of J_j is completed on all upstream machines $M_1, \ldots, M_{k-1}$. Processing job J_j on machine M_k takes a given indivisible amount of time p_{jk}, and all jobs are to be processed in the same order on all machines. Figure 2 shows an example of a solution of a PFSP instance consisting of $n = 3$ jobs and $r = 3$ machines. For $r \geq 3$, this problem is shown to be NP-hard [14]. The lower bound proposed by Lageweg *et al.* [15] is used in our bounding operator. This bound is known for its good results and has a complexity of $O(r^2 n \log(n))$.

The widely used Taillard's class of instances is considered [16]. Those instances are indexed from `ta001` to `ta120` and belong to sub-categories according to their size ($n \times r$): 500×20, 200×20, 200×10, 100×20, 100×10, 100×5, 50×20, 50×10, 50×5, 20×20, 20×10, and 20×5. When $r \leq 10$, the instances can be solved in a few seconds using a sequential B&B and do not justify the need for GPU accelerators. In contrast, instances where $r = 20$ and $n \geq 50$ are very hard to solve. For instance, proving the optimality of the 50×20 `ta058` required over 13 hours of processing on 256 GPUs, and 339×10^{12} node decompositions [2]. Therefore, in this work, the focus is on the ten instances belonging to the 20×20 category.

4.3. Experimental results

Table 1 shows the speed-up achieved by the Chapel multi-GPU version compared to the optimized single-GPU one, on both NVIDIA and AMD GPU-powered systems. Instances are sorted in ascending order based on the number of nodes to better highlight the effect of the tree size on the results. Tree sizes range from 9.5×10^6 to 3944.5×10^6.

One can first see in Table 1a the results obtained using the NVIDIA A100 GPUs. Solving the smallest instances, *i.e.*, `ta029` to `ta022`, we achieve a strong scaling efficiency of up to 77% using 2 GPUs, but the performance remains limited when the number of GPUs increases. For example, we achieve only 50% using 8 GPUs. For these instances, the workload is insufficiently large to maximize the computational capabilities of GPUs. However, when the instance size is large, *e.g.*, for `ta021`, up to 73% of strong scaling efficiency is achieved using 8 GPUs, which represents 17127.7×10^3 nodes processed

Table 1: Node processing rates (in kn/s) and speed-up achieved by the multi-GPU version compared to the optimized single-GPU one, on both NVIDIA and AMD systems. Instances are sorted by the number of nodes.

(a) Results on the NVIDIA A100 GPU architecture.

Inst.	#nodes	GPU×1	GPU×2		GPU×4		GPU×8	
	(10^6)	kn/s	kn/s	speed-up	kn/s	speed-up	kn/s	speed-up
ta029	9.5	2553.7	3909.5	1.54	6168.8	2.42	9047.6	3.55
ta030	13.2	2875.8	4177.2	1.45	6734.7	2.33	10819.6	3.76
ta022	14.6	2754.7	4066.9	1.48	6666.6	2.41	11060.6	4.01
ta027	54.6	2810.1	4187.1	1.49	7388.4	2.63	14521.3	5.16
ta023	115.5	2817.1	4169.7	1.48	7664.2	2.72	15671.6	5.56
ta028	196.9	2804.0	4173.4	1.49	7599.4	2.71	15764.6	5.62
ta025	235.0	2842.3	4187.5	1.47	7612.6	2.68	15953.8	5.61
ta026	514.4	2954.6	4326.3	1.46	7864.2	2.66	16609.6	5.62
ta024	2173.0	2973.5	4296.2	1.44	7986.0	2.69	16989.8	5.71
ta021	3944.5	2965.3	4250.5	1.43	8056.6	2.72	17127.7	5.78
AVG	**727.1**	**2835.1**	**4174.4**	**1.47**	**7374.2**	**2.60**	**14356.6**	**5.04**

(b) Results on the AMD MI50 GPU architecture.

Inst.	#nodes	GPU×1	GPU×2		GPU×4		GPU×8	
	(10^6)	kn/s	kn/s	speed-up	kn/s	speed-up	kn/s	speed-up
ta029	9.5	447.7	784.5	1.75	1332.4	2.98	2311.4	5.16
ta030	13.2	509.7	866.7	1.70	1468.3	2.88	2677.5	5.25
ta022	14.6	473.6	817.5	1.73	1359.4	2.87	2530.3	5.34
ta027	54.6	479.9	838.9	1.75	1538.9	3.21	2902.7	6.05
ta023	115.5	479.7	837.6	1.75	1588.7	3.31	3015.7	6.29
ta028	196.9	481.8	841.8	1.75	1589.2	3.30	3024.6	6.28
ta025	235.0	484.6	845.9	1.75	1594.3	3.29	3047.9	6.29
ta026	514.4	516.3	896.3	1.74	1697.7	3.29	3245.4	6.29
ta024	2173.0	521.2	901.9	1.73	1730.1	3.32	3308.9	6.35
ta021	3944.5	514.7	889.1	1.73	1719.9	3.34	3288.2	6.39
AVG	**727.1**	**490.9**	**852.0**	**1.74**	**1561.9**	**3.18**	**2935.3**	**5.97**

per second. On average, 63% of strong scaling efficiency is achieved using 8 GPUs.

Similarly, Table 1b shows the results obtained on the AMD MI50 GPU architecture. Consistently with the observations done on the NVIDIA A100 GPUs, the strong scaling efficiency increases with the instance size, and up to 80% is achieved using 8 GPUs, which represents 3288.2×10^3 nodes processed per second. On average, 75% of strong scaling efficiency is achieved using 8 GPUs. The difference in the generation and stream processors count of both NVIDIA A100 and AMD MI50 GPU architectures explains the large difference in the maximum processing rate between them. For instance, as described in Section 4.1, NVIDIA A100 has 6912 cores, which is 80% more than AMD MI50.

Another observation that we can make from Table 1 is that the performance scalability does not necessarily increase when the instance size increases. For instance, using 2 GPUs, we note that the strong scaling efficiency is always ±3% the average one. This limited scalability can be attributed to several factors, including communication overhead between GPUs. Additionally, the full utilization of the GPUs' thousands of cores may not be achieved, further limiting performance gains. Indeed, the PFSP involves a bounding function that is irregular and can lead to thread divergence [17]. For instance, if subproblems evaluated in parallel by a warp of threads are located at different levels of the search tree, the threads may diverge. These factors collectively explain why larger instances do not always lead to improved performance scalability for a given number of GPUs.

5. Conclusions and Future Works

This work deals with the design and implementation of a multi-GPU B&B algorithm in Chapel. This PGAS-based programming language stands out as an alternative to traditional CUDA/HIP+X, promoting code performance and portability in addition to expressiveness. The proposed approach consists of a generic multi-pool data structure coupled with a dynamic load balancing mechanism based on work stealing. While the CPU cores are used to perform a parallel tree exploration, GPU devices are used to accelerate the bounding phase, which is particularly compute intensive. Extensive experiments on the PFSP reveal that the implementation can achieve a strong scaling efficiency of up to 63% and 75% on average using a GPU-powered processing node including 8 NVIDIA A100 devices and AMD MI50 GPUs, respectively. This demonstrates the efficiency of our approach to solving large COP instances, while ensuring code portability.

In the future, we plan to extend our contribution to distributed multi-GPU supercomputers, involving many more GPU devices across many compute nodes. The unified view of the inter-node (distributed) and intra-node (shared) parallel levels provided in Chapel will be exploited in addition to its GPU-native support. Our objective is to provide a pioneering tree-search approach unifying the multi-core, distributed, and GPU levels of parallelism for Combinatorial Optimization. Such an approach would pave the way toward the exact resolution of hard and open COP instances. We also plan to implement our algorithm using CUDA/HIP to address the absence of open-source baselines in the literature and to enable more accurate performance comparisons. Another future perspective is to extend the experimental evaluation to Intel GPUs, and also heterogeneous clusters with GPUs from various vendors (NVIDIA, AMD and Intel).

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria, which includes CNRS, RENATER, several universities, and other organizations. This work is supported by the Agence Nationale de la Recherche [grant number ANR-22-CE46-0011] and the Luxembourg National Research Fund (FNR) [grant number INTER/ANR/22/17133848], under the UltraBO project; and by the FNR POLLUX program under the SERENITY project [grant number C22/IS/17395419]. The authors gratefully acknowledge the Chapel's development team, particularly Engin Kayraklioglu, for their expert support.

Data availability statement

All code written in support of this publication is publicly available on GitHub at `https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel`.

Disclosure of interests

The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] B. Gendron and T. G. Crainic, "Parallel Branch-and-Branch Algorithms: Survey and Synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[2] J. Gmys, "Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers," *INFORMS Journal on Computing*, vol. 34, no. 5, pp. 2502–2522, 2022.

[3] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, and G. Titus, "Chapel Comes of Age : Making Scalable Programming Productive," 2018.

[4] A. Grama and V. Kumar, "Parallel Search Algorithms for Discrete Optimization Problems," *ORSA Journal on Computing*, vol. 7, no. 4, pp. 365–385, 1995.

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, 1999.

[6] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens, "A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem," in *2012 IEEE International Conference on Cluster Computing*, 2012, pp. 10–17.

[7] M. E. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 1769–1777.

[8] A. Dabah, A. Bendjoudi, D. El-Baz, and A. Aitzai, "GPU-Based Two Level Parallel B&B for the Blocking Job Shop Scheduling Problem," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016, pp. 747–755.

[9] T.-T. Vu and B. Derbel, "Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments," *Future Generation Computer Systems*, vol. 56, pp. 95–109, 2016.

[10] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens, "IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 9, p. e4019, 2017.

[11] T. Carneiro, N. Melab, A. Hayashi, and V. Sarkar, "Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators," in *18th International Conference on High Performance Computing & Simulation*, 2021.

[12] G. Helbecque, E. Krishnasamy, N. Melab, and P. Bouvry, "GPU-Accelerated Tree-Search in Chapel versus CUDA and HIP," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2024.

[13] G. Helbecque, E. Krishnasamy, T. Carneiro, N. Melab, and P. Bouvry, "GPU-accelerated tree search in Chapel," 2024. [Online]. Available: https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel

[14] M. R. Garey, D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop Scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976.

[15] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem," *Operations Research*, vol. 26, no. 1, pp. 53–67, 1978.

[16] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[17] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1121–1136, 2013.