Concurrency and Computation: Practice and Experience

RESEARCH ARTICLE OPEN ACCESS

Portable PGAS-Based GPU-Accelerated Branch-And-Bound Algorithms at Scale

Guillaume Helbecque[1,2] | Ezhilmathi Krishnasamy[1] | Tiago Carneiro[3] | Nouredine Melab[2] | Pascal Bouvry[1]

[1]University of Luxembourg, DCS-FSTM/SnT, Esch-sur-Alzette, Luxembourg | [2]Université de Lille, CNRS/CRIStAL UMR 9189, Centre Inria de l'Université de Lille, Lille, France | [3]Interuniversity Microelectronics Centre (IMEC), Leuven, Belgium

Correspondence: Guillaume Helbecque (guillaume.helbecque@univ-lille.fr)

Received: 30 November 2024 | **Revised:** 13 August 2025 | **Accepted:** 17 September 2025

Funding: This work was supported by the Agence Nationale de la Recherche [grant number ANR-22-CE46-0011] and the Fonds National de la Recherche Luxembourg (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].

Keywords: branch-and-bound | chapel | graphics processing unit computing | parallel programming | partitioned global address space | portability

ABSTRACT

The Branch-and-Bound (B&B) technique plays a key role in solving many combinatorial optimization problems, enabling efficient problem-solving and decision-making in a wide range of applications. It incrementally constructs a tree by building candidates to the solutions and abandoning a candidate as soon as it determines that it cannot lead to an optimal solution. With modern problems growing increasingly large, accelerating B&B algorithms through parallelization has become a critical challenge for handling large solution spaces. At the same time, modern parallel computing systems themselves are becoming larger, more heterogeneous, and more diverse, requiring programming approaches capable of effectively exploiting such complexity. To address these challenges, this work presents a GPU-accelerated B&B algorithm based on the Partitioned Global Address Space (PGAS) programming model, implemented using the Chapel language. The PGAS-based design is motivated by the high-level abstraction provided by this programming model, which favors programmability, whereas vendor-neutral GPU features of the Chapel language favor GPU portability. The algorithm uses a pool-based approach for generality and exploits a dynamic load balancing mechanism for performance scalability. Extensive experimentation on the N-Queens and permutation flowshop scheduling problems demonstrated both code performance and code portability of the proposed algorithm on several GPU architectures compared to optimized CUDA-based implementations. Moreover, the strong scaling efficiency of the proposed algorithm is investigated on a TOP500 pre-exascale supercomputer up to 1024 GPUs.

1 | Introduction

Branch-and-Bound (B&B) techniques play a key role in solving many combinatorial optimization problems (COPs), enabling efficient problem-solving and decision-making in a wide range of applications [1]. These methods are widely used in fields such as Machine Learning, Operations Research, and Artificial Intelligence, where they help to find optimal solutions by systematically exploring and pruning large search spaces. However, as modern applications grow in size and complexity, search spaces become exponentially larger, making traditional B&B methods inefficient. The need for accelerating these algorithms

Abbreviations: API, application programming interface; B&B, branch-and-bound; BFS, breadth-first search; COP, combinatorial optimization problem; CPU, central processing unit; DFS, depth-first search; GPU, graphics processing unit; HPC, high-performance computing; LICM, loop-invariant code motion; NUMA, non-uniform memory access; PFSP, permutation flowshop scheduling problem; PGAS, partitioned global address space; WS, work stealing.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2025 The Author(s). *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

through parallelization techniques has thus become a significant challenge, as managing large, irregular solution spaces requires advanced parallel architectures and load-balancing strategies to maintain efficiency and scalability.

Graphics Processing Units (GPUs) have emerged as building blocks in modern supercomputers, reshaping the landscape of High-Performance Computing (HPC) [2]. Their parallel processing capabilities accelerate computations, making them a valuable asset in solving large and complex problems. However, the design of GPU-accelerated tree search raises multiple challenges related to irregular workloads, dynamic memory requirements, and data exchanges of those methods present in B&B applications. In addition to GPU heterogeneity, modern systems are becoming more diverse. For instance, among the top five systems in the latest TOP500 ranking (June 2025), five different GPU architectures are used, including three different vendors: AMD, Intel, and NVIDIA [2]. Therefore, in addition to design challenges related to GPU-accelerated tree search, particular attention must also be paid to code portability and performance portability.

The Partitioned Global Address Space (PGAS) paradigm contrasts with traditional shared-memory and message-passing models by providing a global memory space that is logically partitioned among multiple nodes, allowing for global addressing, which avoids the use of lower-level explicit communication, for example, such as those used in MPI [3]. PGAS balances the ease of shared-memory programming with the scalability of message-passing by abstracting the complexity of data distribution and inter-node communication. However, GPU programming in this context is in its infancy, and studies are still needed to investigate the suitability of these approaches, for instance, for accelerating tree search algorithms.

In this paper, we present a GPU-accelerated B&B algorithm based on the PGAS programming model, implemented using the Chapel language. The PGAS-based design is motivated by the high-level abstraction provided by this programming model, which favors programmability, whereas vendor-neutral GPU features of the Chapel language favor GPU portability. It extends our previous work [4], which introduced an intra-node pool-based GPU-accelerated B&B algorithm. Our algorithm is generic with respect to the problem being solved and can be executed on both NVIDIA and AMD GPU architectures without any code modification. To handle the irregularity of the explored tree, we introduce a new locality-aware work-stealing mechanism that dynamically balances the workload across computing nodes. The main contributions of this paper are as follows:

- The design and implementation of a GPU-accelerated B&B algorithm based on the PGAS programming model, that is portable in terms of GPU vendor, and also achieving code performance, as low performance loss compared to its optimized CUDA-based counterparts is observed;
- The extension of a previously proposed intra-node GPU-accelerated algorithm [4] with a novel, locality-aware work-stealing mechanism to handle load imbalance in distributed-memory environments;
- A comprehensive portability analysis across six different GPU architectures, covering multiple generations and both major vendors (NVIDIA and AMD);
- A strong scalability evaluation on a Top 10 pre-exascale supercomputer, demonstrating the algorithm's ability to scale up to 1024 GPUs;
- An empirical validation of the algorithm on two representative benchmark problems—N-Queens and permutation flowshop scheduling—highlighting its robustness and general applicability;
- An in-depth exploration of parameter tuning and work-stealing efficiency, including a discussion of current limitations and performance trade-offs.

By addressing both the algorithmic and system-level challenges of portable GPU-accelerated B&B at scale, this work provides a foundation for future developments in high-performance combinatorial optimization.

The remainder of this paper is organized as follows. Section 2 presents a background on the B&B method and existing parallel models, along with some related works. Section 3 provides the design and implementation of our PGAS-based multi-GPU B&B algorithm. Crucial aspects for scalable performance, such as load balancing and termination detection, are discussed. Section 4 reports the experimental evaluation of the proposed algorithm. Then, Section 5 discusses the role and boundaries of PGAS in simplifying distributed-memory parallelism and also brings some concluding remarks concerning code portability and performance portability. Finally, Section 6 draws the conclusions of this work and highlights some future directions.

2 | Background and Related Works

2.1 | The Branch-And-Bound (B&B) Method

Branch-and-Bound (B&B) is a general problem-solving method that implicitly enumerates the solution space by incrementally building a solution and abandoning a path as soon as it determines that this path cannot improve the best solution found so far. In B&B, the root node corresponds to the initial problem, the internal nodes represent partially constructed solutions or intermediate *subproblems*, and the leaf nodes represent complete solutions. As illustrated in Algorithm 1, the exploration of the solution tree relies on four main operators: *branching*, which divides a given subproblem into several smaller, pairwise disjoint subproblems by generating new branches in the tree; *bounding*, which computes a bound on the best possible solution within a subproblem (lower bound for minimization problems and upper bound for maximization problems); *pruning*, which discards subproblems that cannot lead to an optimal solution because their bound is worse than the cost of the current best solution; and *selection*, which chooses the next node to explore based on predefined criteria.

The B&B method typically stores generated but not yet evaluated subproblems in a *work pool*, and different enumeration

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ALGORITHM 1 | Pseudo-code of a sequential B&B algorithm.

```
Require: root - problem-specific root node
Require: pool - pool of nodes (initially empty)
1: pool.insert(root);
2: while (pool.size > 0) do
3:   node ← pool.remove();                          ▷ contain selection rule
4:   if (node is a leaf) then
5:     cost ← evaluate(node);
6:     if (cost better than best known cost) then
7:       update best known cost and solution;
8:     end if
9:   else
10:     bound ← compute_bound(node);
11:     if (bound better than best known cost) then     ▷ else prune node
12:       child_nodes ← branch(node);
13:       pool.insert(child_nodes);
14:     end if
15:   end if
16: end while
```

strategies exist to expand tree nodes. In Depth-First Search (DFS), tree branches are explored as far as possible before backtracking. It is easily implemented by storing subproblems in a stack (last-in, first-out). In contrast, Breadth-First Search (BFS) explores all tree nodes at the present depth prior to moving on to the nodes at the next depth level, and is generally implemented using a queue (first-in, first-out). In this work, both exploration strategies are combined.

Due to its enumerative nature and pruning operator, B&B typically generates large and irregular trees when tackling large problem instances. To accelerate the search process, advanced B&B operators like dynamic branching or selection can be employed, though these often leverage problem-specific characteristics [5]. A more general and widely adopted strategy is parallel processing, which distributes the workload across multiple processing units, such as Central Processing Units (CPUs) or GPUs.

2.2 | Parallel Models for Accelerating B&B Algorithms

The parallelization of B&B has been particularly studied, and four parallel methods can be outlined from the literature [1, 6]:

- The *parallel tree exploration model*, which consists of exploring disjoint sub-trees concurrently and asynchronously in parallel, as shown in Figure 1. This is undoubtedly the model that has been studied the most, since it is generic and has a high degree of parallelism on big problem instances.
- The *parallel evaluation of bounds model*, which consists of bounding several subproblems concurrently in parallel, as illustrated in Figure 2. This model has the advantage of being generic, but its synchronous nature can make it costly in terms of CPU time. Moreover, its granularity (cost of the bounding function) can be fine-grained, and therefore penalize it in a large-scale environment.

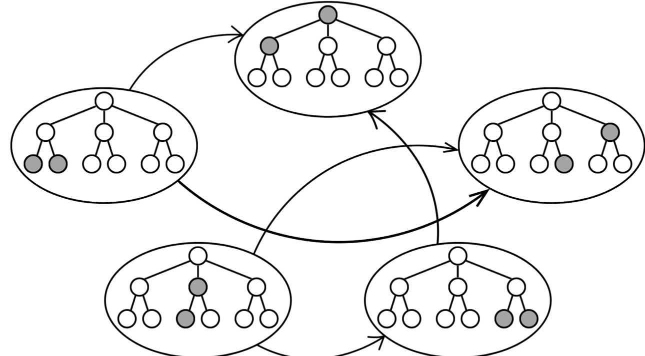

FIGURE 1 | Illustration of the parallel tree exploration model.

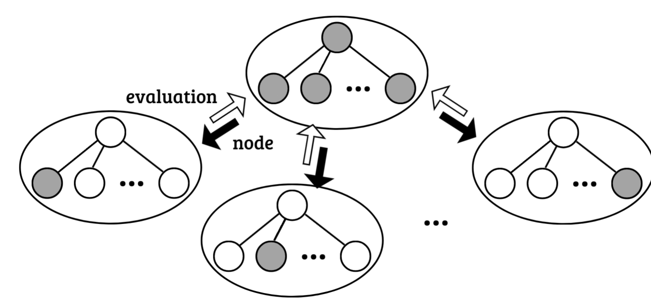

FIGURE 2 | Illustration of the parallel evaluation of bounds model.

- The *parallel evaluation of a bound model*, which consists of parallelizing the bounding function itself. This model is therefore problem-specific, and is only beneficial when the node bounding function is coarse-grained.
- The *parallel multi-parametric model*, which consists of launching several B&B algorithms with different parameterization (e.g., exploration order, branching strategy) concurrently in parallel. The main advantage of this model is its genericity, but it has the disadvantage of generating

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

additional exploration costs as many nodes are explored redundantly.

In this work, the first two parallelization models are considered. More precisely, both models are combined together as they are known to generate massive parallelism on GPU-powered clusters.

2.3 | Related Work

Historically, GPUs are specialized electronic circuits designed for digital image processing and accelerating computer graphics. However, the rise of General-Purpose computing on GPU (GPGPU) in the 2000s enabled their use for applications involving compute-intensive, highly parallel processing tasks [7]. The CUDA platform was one of the first to offer a GPGPU framework, allowing the use of the C programming language to code algorithms to execute on the GPU. Subsequently, many other frameworks followed, such as OpenCL, SYCL, and ROCm HIP, as shown in Figure 3.

B&B algorithms, due to their highly parallel nature, have been the focus of numerous studies investigating their acceleration on GPUs. These studies covered a variety of applications, including COPs such as scheduling [8–10] and knapsack problems [11, 12], cryptographic analysis like block cipher problems [13], industrial process optimization in multi-product batch plants [14], as well as game search [15, 16] and linear programming test problems [17].

For fine-grained problems, the parallel tree exploration model is often employed directly on the GPU [15, 16, 18]. These approaches typically begin with a sequential or weakly parallel search on the CPU to generate a sufficiently large set of pending subproblems. The GPU then performs a parallel exploration of the resulting pool of subproblems. Although varying thread granularity can lead to load imbalance, these methods rely exclusively on the static work distribution produced by the CPU, assuming that it is sufficiently uniform to avoid major performance degradation.

In the case of medium-grained problems, the bounding operator represents the dominant computational cost, whereas the evaluation of individual subproblems remains lightweight enough to be handled efficiently by single GPU threads. Most approaches in this category delegate the evaluation of multiple subproblems in parallel to the GPU, whereas retaining control of the tree exploration logic on the CPU [11, 19]. Considerable effort has been devoted to reducing CPU–GPU data transfer overheads and offloading more of the algorithm's components to the GPU. In some cases, researchers have introduced problem-specific data structures to enable fully GPU-resident B&B implementations [20]. To address irregular workloads, several strategies have been proposed to allow work redistribution among processing units, typically performed during explicit balancing phases.

For coarse-grained problems, the bounding function is computationally expensive. Therefore, acceleration efforts often focus on optimizing the bounding kernel itself [8, 17]. In such cases, the high computational cost per subproblem means that relatively few tasks are required to saturate the GPU's compute capacity, distinguishing them from medium-grained workloads that benefit more from fine-grained parallelism.

Despite the multitude of different approaches in the literature and the continuous development of GPGPU frameworks (see Figure 3), implementations are largely based on CUDA, which raises significant portability issues since this platform is specific to NVIDIA GPUs. Moreover, this low-level Application Programming Interface (API) is often combined with other

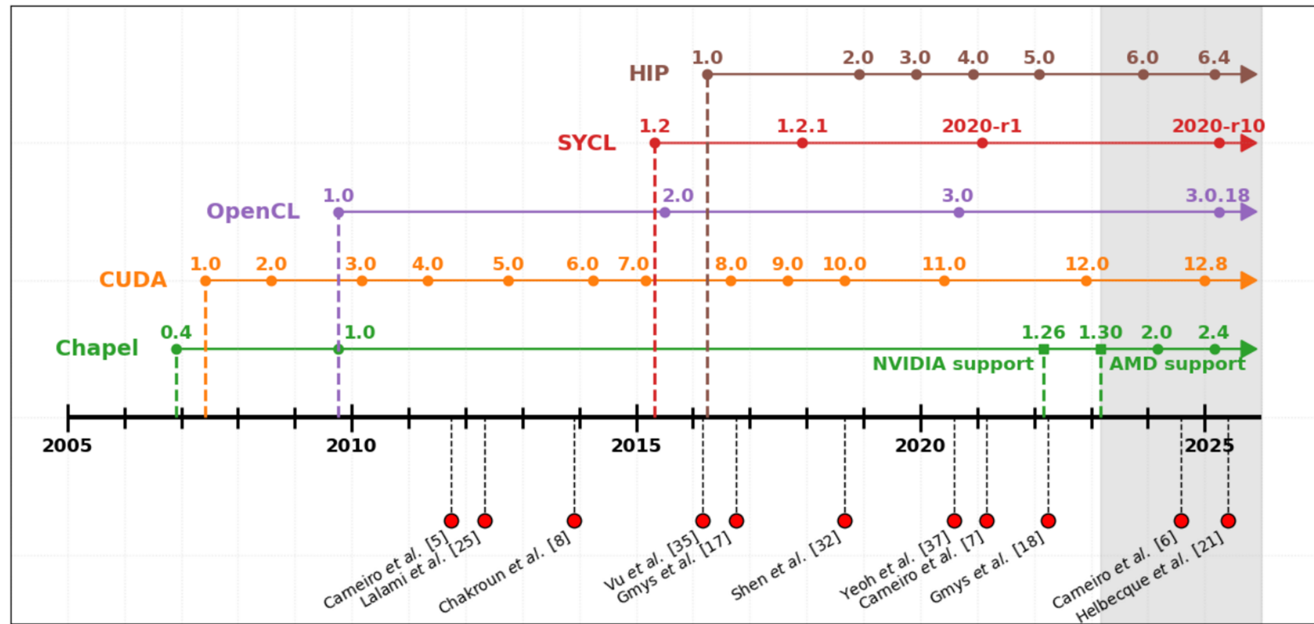

FIGURE 3 | A retrospective on GPU-oriented programming environments and GPU-accelerated B&B algorithms (2005–2025). The list of references is not exhaustive. The gray-shaded range indicates the period during which vendor-neutral GPU support for PGAS Chapel became available.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

parallel programming environments, such as OpenMP or POSIX Threads for shared-memory models, and MPI for distributed memory. In a context where modern supercomputers are becoming increasingly large, GPU-heterogeneous, and diverse, this introduces critical challenges in terms of programmability and maintainability [2]. Additionally, most existing implementations are tightly coupled to problem-specific design, limiting their reusability across domains.

This work revisits the design of GPU-accelerated B&B algorithms in the context of the PGAS model, offering an alternative to the traditional MPI + X approach. In PGAS, memory is logically shared but physically distributed across multiple compute nodes [3]. Each compute node has local memory, and all compute nodes can access both local and remote memory with an emphasis on locality to optimize performance in distributed systems. In this context, GPU computing is in its infancy. Some independent research initiatives have been developed to take advantage of GPU clusters within PGAS-based languages [21–23], mostly using compiler extensions and dedicated libraries. In [24], Carneiro et al. introduced a proof-of-concept GPU-accelerated backtracking algorithm that integrates Chapel—a PGAS-based programming language designed for productive large-scale parallel computing [25]—with pre-compiled CUDA kernels, leveraging Chapel's distributed iterators and a partial search strategy to more effectively exploit intra-node parallelism.

More recently, GPU-native supports became available, as in the Chapel language [26]. It provides high-level abstraction for data and task parallelism, vendor-neutral GPU programming features, distributed arrays leveraging thousands of nodes' memories and cores, and so forth. In [27], Carneiro et al. extended their previous work by replacing the CUDA layer with Chapel's native features and investigated the code performance and portability on GPU-powered clusters. They demonstrated that their approach scales on two different systems using the same Chapel-based tree search, with the high-level abstraction incurring less than a 10% performance penalty on the largest problem instances compared to baseline implementations. Similarly, we proposed a pool-based GPU-accelerated backtracking in Chapel targeting the N-Queens problem [28]. This alternative approach is designed to be generic with regard to the problem being solved, allowing extensibility to other problems. In [4], the algorithm is extended to a B&B algorithm to solve hard instances of the permutation flowshop scheduling COP. On average, a strong scaling efficiency of up to 63% and 75% is achieved using systems equipped with 8 NVIDIA A100 and 8 AMD MI50 GPUs, respectively.

Although [4] opens the way to PGAS-based GPU-accelerated B&B algorithms for large-scale GPU-powered clusters, some strong limitations remain. First of all, we designed and experimented with the proposed algorithm only considering the intra-node parallel level on shared-memory systems. An extension to the inter-node level is therefore necessary to investigate further the performance scalability and leverage distributed-memory architectures. In addition, the algorithm is evaluated on a limited number of GPU architectures, some of which are outdated and no longer used in modern supercomputers. Consequently, the performance evaluation should also be extended to incorporate modern GPUs, in particular those used by the supercomputers dominating the TOP500 rankings (e.g., the AMD Instinct MI250X GPU architecture used by the pre-exascale LUMI supercomputer, 9th of the TOP500 list of June 2025).

3 | Design and Implementation

This section outlines the design and implementation of our PGAS-based multi-GPU B&B algorithm. We describe the parallel CPU–GPU coordination, dynamic load balancing via work stealing, and global termination detection. Key implementation aspects using the Chapel language are also presented.

3.1 | PGAS-Based Multi-GPU B&B Algorithm

The design of the proposed algorithm combines both the parallel tree exploration on CPU and the parallel evaluation of bounds on GPU, as illustrated in Figure 4. In this work, we assume that all compute nodes have the same amount of computing resources (e.g., CPU and GPU cores).

At the intra-node level, the algorithm maintains a multi-pool of tree nodes. More precisely, each GPU device is assigned to a CPU core host managing a pool of tree nodes. In that configuration, the CPU manages all the algorithm's dynamic aspects, such as data structure management and irregular workload. GPU devices operate independently, with any necessary communication or collective operations handled by the hosts. The tree exploration takes place in parallel on the hosts according to the following steps: (1) a tree node is first taken from the pool; (2) this tree node is then bounded using the problem-specific bounding function; (3) depending on the bound, it is either branched or pruned; and (4) the previous steps are repeated until the global termination of the algorithm is triggered.

In practice, the pool size Q increases as iterations progress, leading to CPU over-occupancy. At this point, GPU devices are used to accelerate the bounding of pending subproblems, doing this massively in parallel. Specifically, when a pool contains at least m nodes, a chunk of $q = \max(Q, M)$ nodes is transferred from the host to the device. Then, the device performs the bounding of all tree nodes in parallel and sends the results back to the host. The latter is then able to proceed with the tree exploration, applying branching or pruning based on the pre-computed bounds. The parameters m and M define the minimum number of nodes needed for efficient GPU processing and the maximum number of nodes to transfer, respectively. The optimal values for m and M must be determined by experimentation.

The performance of such a multi-pool algorithm targeting irregular applications lies in the load balancing between local and remote pools. Therefore, we assume that compute nodes are interconnected through a high-performance network at the inter-node level. The latter enables remote data exchange, which is essential for the design of efficient load balancing mechanisms.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

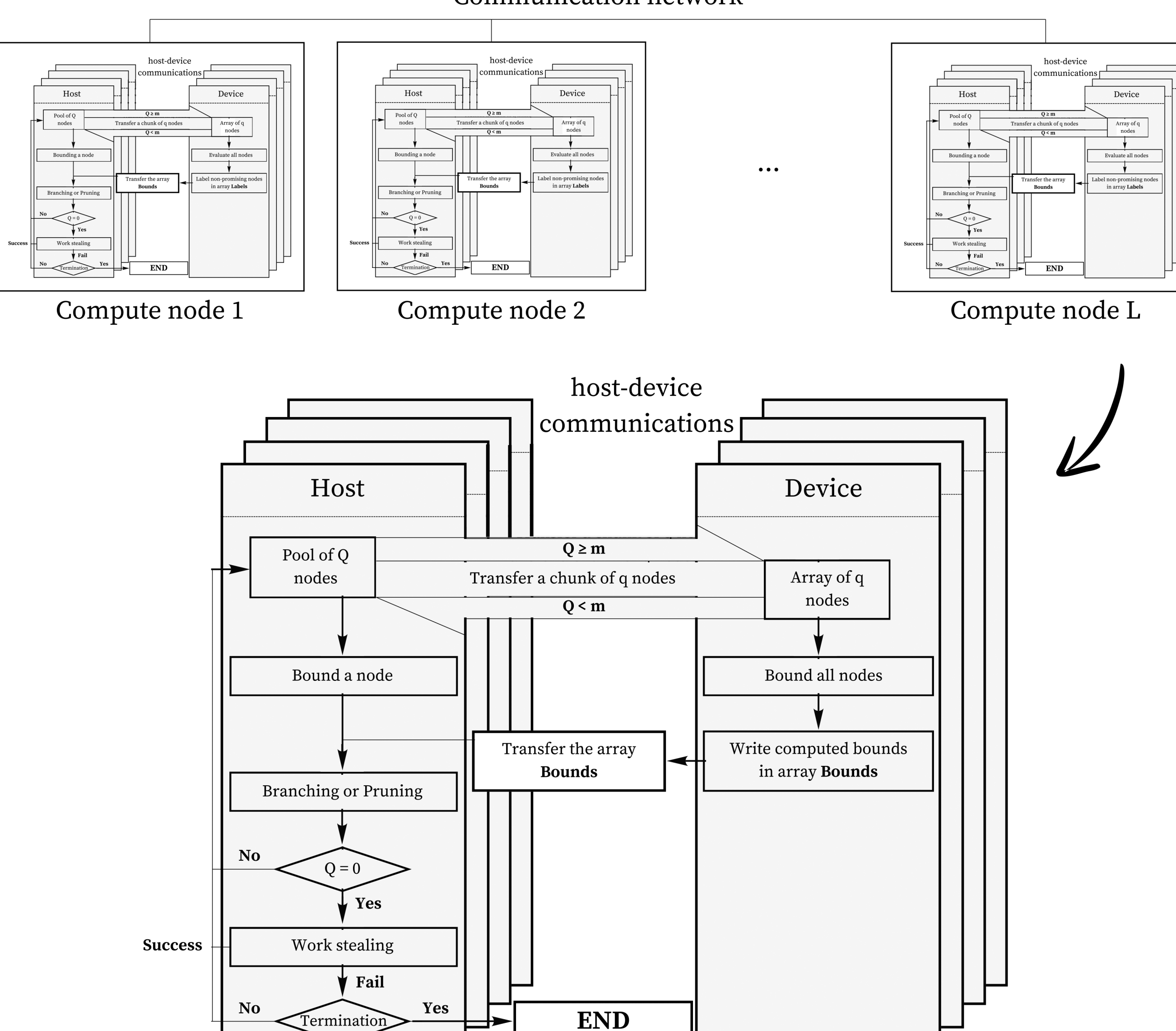

FIGURE 4 | Illustration of our parallel multi-GPU B&B algorithm.

3.1.1 | Initial Search and Static Workload Distribution

The beginning of the algorithm is particularly challenging, as only the root node is available to all processes [1]. In that case, a start-up phase in which parallelism is not fully utilized seems difficult to avoid. In this work, we address this issue by performing a sequential B&B algorithm up to a point where a sufficient number of pending subproblems are available. This strategy aims to provide a sufficiently large number of subproblems to each pool 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 subproblems to maintain high utilization and efficiency. The initial search is performed in a BFS manner, allowing for a broad and uniform distribution of subproblems. The latter ends when at least $m \times \text{numGPUs} \times \text{numComputeNodes}$ pending subproblems are in the pool, ensuring that all processes start the parallel exploration with at least m subproblems each.

The workload distribution at the end of the initial search is done in a cyclic fashion, as shown in Figure 5. More precisely, two cyclic distributions occur successively. The first one distributes all the workload evenly among all the available compute nodes, whereas the second one distributes all the workload on each compute node evenly among the processing cores. Although the mapping is static, it is generally more practical to perform two successive distributions. On a distributed architecture with Non-Uniform Memory Access (NUMA), relying on a single master task to traverse each thread on every compute node for element transfers would result in increased communication overhead.

3.1.2 | Dynamic Work Stealing Mechanism

To ensure that the workload remains balanced during parallel execution, a load balancing mechanism based on Work Stealing (WS) is dynamically triggered when a pool becomes empty, as

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

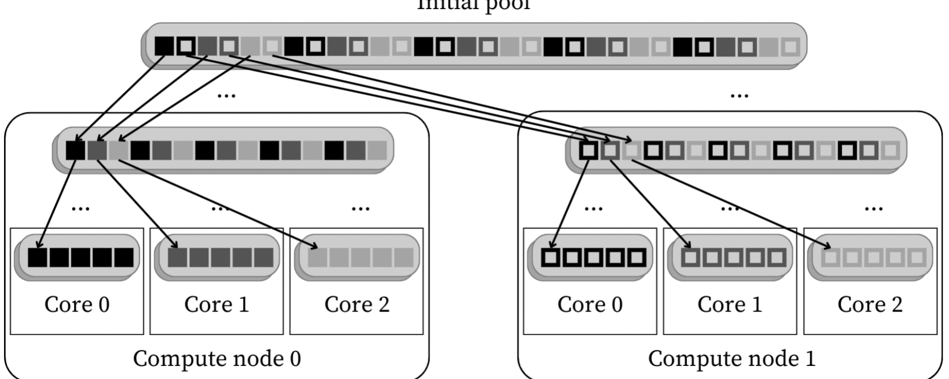

FIGURE 5 | Illustration of the static workload distribution, assuming that 2 compute nodes are used, each with 3 CPU cores, and $m = 5$.

shown in Figure 4. This mechanism is composed of three main components: a victim selection strategy, a work sharing policy, and a synchronization mechanism.

3.1.2.1 | Victim Selection Strategy. The victim selection strategy determines how victim pools are chosen among all the available ones. One of the most commonly used and provably efficient policies in the literature is the random selection policy [29]. It assumes that a victim thread is selected at random when a thread initiates a WS operation. However, since most distributed systems have a NUMA architecture, threads are not selected uniformly as the cost of accessing a remote thread is higher compared to a local one. Therefore, our victim selection strategy exploits locality awareness. When a thread becomes empty during execution and triggers a WS operation, it first uniformly selects a victim thread from its own compute node. Indeed, at the intra-node level, memory access can be considered as uniform. If all WS attempts fail, the initiator then searches from another compute node, also selected randomly. In this work, the mechanism allowing the random selection of victim threads/locales is implemented as an iterator yielding victim indexes, as shown in Algorithm 2.

ALGORITHM 2 | Pseudo-code of the random victim selection iterator.

```
Require: thiefId - index of the calling thread
   or compute node
Require: n - number of threads or compute
   nodes
1: id ← 0;
2: victims ← permuteRand([0, ..., n-1]);
3: while (id < n) do
4:    if (victims[id] ≠ thiefId) then
5:       yield victims[id];
6:    end if
7:    id ← id + 1;
8: end while
```

3.1.2.2 | Work Sharing Policy. The work sharing policy controls how many subproblems are transferred during a WS operation and plays a critical role in achieving good load balance. Stealing too few subproblems may lead to frequent, low-yield stealing attempts, increasing synchronization overhead and contention. However, stealing too many subproblems risks overloading the thief thread and potentially creating new imbalances. Additionally, since WS may involve remote communication—particularly in inter-node scenarios—the cost of transferring a small number of subproblems can easily outweigh the benefits of improved parallelism.

To address these trade-offs, this work adopts a policy in which the initiator thread steals half of the subproblems available in the victim's pool, but only if the victim holds more than $2 \times m$. This threshold ensures that, after the stealing operation, both the initiator and the victim retain at least m subproblems, allowing them to proceed efficiently with the exploration. This policy strikes a balance between communication efficiency and work granularity, and prevents excessive draining of any one pool.

Regarding idle periods or potential starvation, we do not enable CPU fallback execution. Bounding subproblems on the CPU would significantly degrade performance, as GPU bounding is orders of magnitude faster for both problem classes considered. Moreover, dynamically offloading execution from GPU to CPU would require complex memory consistency management in a distributed PGAS environment. Instead, the WS mechanism is locality-aware and hierarchically structured: it first attempts to steal work from other threads within the same locale, where memory access is uniform, before escalating to inter-node stealing. This hierarchical approach mitigates starvation while keeping computation confined to the most efficient resources.

3.1.2.3 | Synchronization Mechanism. The synchronization mechanism determines how initiator and victim threads coordinate themselves to ensure the parallel safety of pools. In this algorithm, we adopt a spin-lock synchronization mechanism. When a thread tries to acquire a spin-lock, it continuously checks (or "spins") to see if the lock is available, rather than yielding control or putting itself to sleep. This is achieved by repeatedly polling a flag or memory location until the lock is free. Spin-locks are typically lightweight and provide low-latency access to locks, making them suitable for short critical sections where the lock is held briefly. However, they can cause high CPU usage if contention is significant, as waiting threads consume processor cycles while spinning.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3.1.3 | Global Termination Detection

In asynchronous environments, where B&B processes operate independently and at different speeds, detecting global termination becomes particularly challenging. To address this, Figure 6 shows the flowchart of our global termination detection. The method involves maintaining a record of active threads and their states, enabling the system to determine if any ongoing computation is still in progress. It assumes that each thread maintains a state variable in the global address space, either set to `BUSY` or `IDLE`. A state variable is also assigned to each locale (i.e., compute node) and is set to `BUSY` if at least one of its threads is busy; `IDLE` otherwise. To ensure that the states are consistent with the actual states of the thread/locale, we implement them as atomic variables. The algorithm is initiated when a thread becomes idle during execution, meaning that its work pool is empty and all WS attempts fail. In that case, the initiator first checks the states of the other threads from its locality (Step 1), after setting its own state to `IDLE`. If at least one thread is busy, then the termination detection ends, and the initiator continues the B&B algorithm. Otherwise, the locality state is set to `IDLE`, and the states of the other localities are checked (Step 2). If all localities are found to be idle, the algorithm triggers the global termination of the B&B algorithm.

In this PGAS-based global termination detection algorithm, the initiator reads the state of each other thread from the global address space in a one-sided manner. Since some of the state variables are located in remote memory areas, we implement the global termination detection mechanism in a locality-aware manner. The first step allows checking the state of each thread from the same locality at a relatively low cost, performing uniform memory access at the intra-node level. Then, the second step, involving remote memory accesses at the inter-node level, is triggered only if the first step does not allow the initiator to make a decision. This method aims to minimize the number of remote communications, hence reducing the associated overhead.

3.2 | Implementation Aspects in Chapel

In Chapel, GPUs are considered as *locales*, which are Chapel abstractions for parts of a target architecture that have processing and storage capabilities (such as a compute node). Generally speaking, a thread running within a locale has roughly uniform memory access to the data stored in the locale's local memory and longer latencies for accessing the memory of another locale. The migration of a thread from one locale to another (potentially) remote one is done using Chapel's `on`-clause, which prefixes another statement, specifying where it should be executed. Within a locale targeting a GPU, Chapel stores by default array data directly on the GPU device memory and stores other data on the CPU host in a page-locked manner.

Chapel provides several data-parallel loops, such as `foreach` and `forall` loops. The first one asserts that the loop meets the order-independent and unsynchronized properties of the iterations. It specifies that its iterations should be implemented using hardware parallelism if possible. In contrast, `forall` loops are similar to `foreach` loops, except that they have the potential to be implemented using multiple Chapel tasks. This permits them to use multiple cores and/or compute nodes to execute the loop's iterations. The Chapel compiler will generate GPU kernels for certain `forall` and `foreach` loops and launch these onto a GPU when the current locale is assigned to a locale representing a GPU. More precisely, loops are said to be *GPU-eligible* when: (1) they are order-independent, (2) they only make use of known compiler primitives that are fast and local, (3) they do not call out to extern functions, and (4) they are free of any call to a function that fails to meet the above criteria or accesses outer variables. Any code in an on statement for a GPU locale that is not within an eligible loop will be executed on the CPU.

Chapel's GPU-native support is designed in a vendor-neutral manner. Specifically, Chapel supports compilation from a single source file to multiple target architectures, including CPUs and GPUs. This is achieved through the LLVM compiler framework, which allows Chapel code generation to target a diverse range of back-ends, such as PTX for NVIDIA GPUs and AMDGCN for AMD GPUs.

4 | Experimental Evaluation

Section 4.1 introduces the benchmark problems considered for the experimental evaluation, followed by a description of

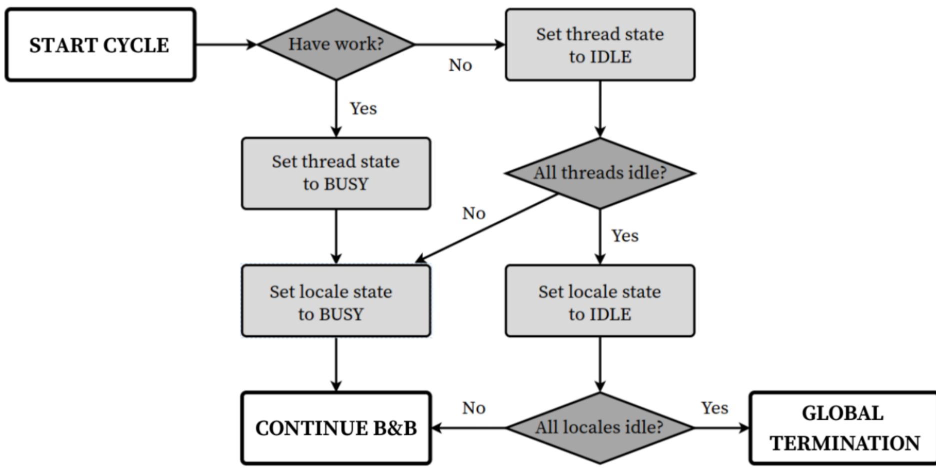

FIGURE 6 | Flowchart of the global termination detection.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the experimental protocol and testbed in Section 4.2. Then, Section 4.3 examines the code performance and code portability of the algorithm on several GPU architectures, and compares the results to an optimized CUDA-based baseline implementation. In Section 4.4, we present a calibration of the algorithm's parameters, whereas Section 4.5 evaluates its strong scaling efficiency on up to 1024 GPUs. Finally, Section 4.6 analyzes the benefits of the proposed WS mechanism at both intra- and inter-node parallel levels.

4.1 | Benchmark Problems

4.1.1 | The Permutation Flowshop Scheduling Problem

The Permutation Flowshop Scheduling problem (PFSP) is a well-known NP-hard COP in which the aim is to determine an optimal processing order (a permutation) of n jobs $\{J_1, \dots, J_n\}$ on r machines $\{M_1, \dots, M_r\}$ minimizing the total completion time, or makespan [30]. Adhering to a chain production principle, a job J_j can only start processing on machine M_k once it has been completed on all preceding machines M_i, $i < k$ The processing time of job J_j on machine M_k is a fixed, indivisible duration p_{jk}, and all jobs must be processed in the same sequence on all machines. Figure 7 illustrates an example solution for a PFSP instance with $n = 3$ jobs and $r = 3$ machines. Our B&B bounding operator employs the lower bound proposed by Lageweg et al. [31], which is renowned for its effectiveness and has a computational complexity of $\mathcal{O}\left(r^2 n \log(n)\right)$.

The most used benchmark instances considered in the literature are the ones defined by Taillard [32]. The latter are indexed from `ta001` to `ta120` and are divided into 12 groups of 10 instances according to their size ($n \times r$): 20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5, 100×10, 100×20, 200×10, 200×20, and 500×20. When $r \leq 10$, the instances can be solved in a few seconds using a sequential B&B. However, instances where $r = 20$ and $n \geq 50$ are very hard to solve. For example, proving the optimality of the 50×20 `ta058` required over 13 h of processing on 256 NVIDIA V100 GPUs, and 339×10^{12} node decompositions [9]. In this paper, the class of 20×20 instances is considered, and Table 1 provides some statistics for each instance.

4.1.2 | The N-Queens Problem

The N-Queens problem has been studied for over 170 years and remains a classic challenge in Computer Science, particularly as a benchmark for tree-based algorithms [33]. It consists of placing N queens on a N by N chessboard, so that no queen captures any other. That is, the board configuration in which there exists at most one queen on the same row, column, and diagonals. In this work, the proposed algorithm is used to determine the exact solution count for a given N. N-Queens instances of sizes $N = 15$ to 19 are considered. Instances are denoted "NQ", where N is the number of queens. Table 2 summarizes their solution counts, along with the size of the explored tree to obtain them.

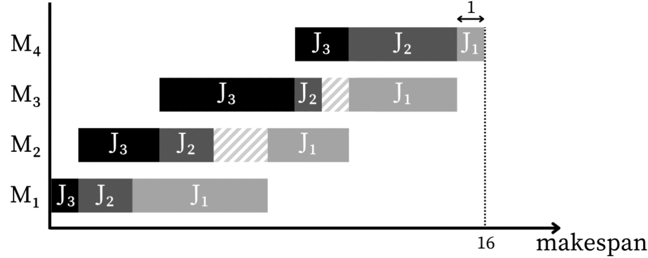

FIGURE 7 | Solution of a PFSP instance consisting of $n = 3$ jobs and $r = 4$ machines, corresponding to the permutation (3,2,1).

TABLE 1 | Summary of the PFSP instances solved in this paper.

Instance	Number of explored nodes	Optimum
`ta029`	9,499,307	2237
`ta030`	13,228,600	2178
`ta022`	14,561,974	2099
`ta027`	54,588,346	2273
`ta023`	115,511,993	2326
`ta028`	196,916,205	2200
`ta025`	234,988,695	2291
`ta026`	514,453,278	2226
`ta024`	2,173,092,255	2223
`ta021`	3,944,527,267	2297

Note: Instances are sorted by their number of explored nodes.

TABLE 2 | Summary of the N-Queens instances solved in this paper.

Instance	Number of explored nodes	Solution count
15Q	171,129,071	2,279,184
16Q	1,141,190,302	14,772,512
17Q	8,017,021,931	95,815,104
18Q	59,365,844,490	666,090,624
19Q	461,939,618,823	4,968,057,848

The N-Queens problem can be modeled as a permutation problem, in which the value $x_j \in \{1, \dots, N\}$ at index $j \in \{1, \dots, N\}$ indicates that a queen is placed in column x_j of row j. As an example, Figure 8 shows a feasible solution for $N = 8$. The encoding of a solution as a permutation of size N ensures that exactly N queens are placed on the board and that the "exactly-one" constraints on rows and columns are satisfied. Therefore, to evaluate the feasibility of a (partial) solution, it is enough to check for diagonal conflicts among the already placed queens.

Formally, N-Queens is not an optimization problem, but a constraint satisfaction problem. However, B&B can easily be adapted to solve such problems. Instead of searching for optimal solutions, the goal is to find all valid solutions. Instead of a lower bound on the optimal cost of a subproblem, it is enough to use a node evaluation function that assigns the Value 1 to feasible (partial) solutions and 0 to infeasible (partial) solutions. Initializing the algorithm at the upper bound 0 and pruning only in the case of strict inequality, the number of leaf nodes visited by B&B equals the number of valid board configurations.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

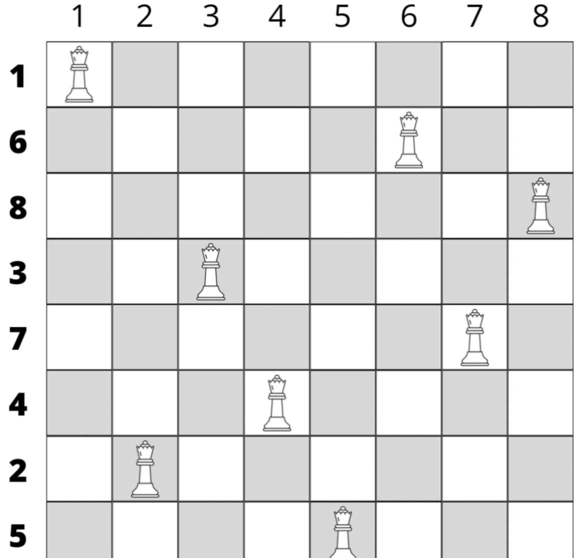

FIGURE 8 | Solution of the 8Q instance. The permutation representing the board configuration is displayed on the left side of the board.

4.2 | Experimental Protocol and Testbed

Three variants of the algorithm have been implemented in Chapel: single-GPU, single-node multi-GPU, and multi-node multi-GPU. Moreover, for comparison purposes, we also implemented optimized single-GPU and single-node multi-GPU baseline implementations based on the low-level CUDA API. In order to target AMD GPU architectures, the latter CUDA codes have been "translated" to HIP using the `hipify-perl` tool provided by AMD/ROCm. The complete source code is available at https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel.

In the following, Section 4.3 focuses on code performance and code portability in a single-GPU setting. For that purpose, the following six configurations, including GPU architectures from different vendors, are considered:

- *NVIDIA P100*: 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60 GHz CPU, equipped with a NVIDIA Tesla P100 PCIE 16GB GPU (3584 CUDA cores, released in June 2016);
- *NVIDIA V100*: 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60 GHz CPU, equipped with a NVIDIA Tesla V100 PCIE 32GB GPU (5120 CUDA cores, released in March 2018);
- *NVIDIA A100*: 32-core AMD EPYC 7513 (Zen 3) @ 2.60 GHz CPU, equipped with a NVIDIA A100 SXM4 40GB GPU (6912 CUDA cores, released in May 2020);
- *AMD MI50*: 48-core AMD EPYC 7642 (Zen 2) @ 2.40 GHz CPU, equipped with a AMD Radeon Instinct MI50 32GB GPU (3840 stream processors, released in November 2018);
- *AMD MI250X*: 64-core AMD EPYC 7A53 "Trento" (Zen 3) @ 2.0 GHz CPU, equipped with a AMD Instinct MI250X GPU (7040 stream processors, released in November 2021);
- *AMD MI300X*: 64-core AMD EPYC 7A53 "Trento" (Zen 3) @ 2.0 GHz CPU, equipped with a AMD Instinct MI300X GPU (9728 stream processors, released in June 2023).

Later, we investigate performance scalability in multi-GPU configuration, considering both intra- and inter-node parallel levels. Only the system equipped with AMD MI250X GPUs is considered. It corresponds to the HPE Cray EX LUMI European pre-exascale supercomputer.[1] Compute nodes are equipped with a single 64-core AMD EPYC 7A53 "Trento" CPU and four AMD Instinct MI250X accelerators based on the 2nd Gen AMD CDNA architecture. An MI250X is a multi-chip module with two GPU dies, each featuring 110 compute units (CU) and 64 GB slice of HBM2e memory for a total of 220 CUs and 128 GB of total memory per MI250X module. From a software perspective, an MI250X module is considered as two GPUs, meaning that nodes can be considered as 8 GPU nodes. The compute nodes use the HPE Cray Slingshot-11 network interconnect and are equipped with four endpoints, one for each AMD MI250X GPU module. Each endpoint provides up to 50 GB/s of bidirectional bandwidth. The contribution is implemented using Chapel 2.1.0 built with the LLVM 15.0.0 back-end compiler, and HIP-based codes rely on ROCm 5.4.6.

4.3 | Code and Performance Portability

Figures 9 and 10 show the normalized execution time of the Chapel implementation compared to the CUDA/HIP baseline, solving instances of the N-Queens problem and PFSP, respectively. Single-GPU experiments are conducted to isolate and evaluate the core performance on individual GPUs without the added complexity of multi-GPU coordination or communication overhead. It is important to note that no direct comparison can be made between the performance and capabilities of the different GPU architectures.

Solving the N-Queens instances, one can see that the Chapel single-GPU implementation is between 87% faster and 43% slower than the baselines. Actually, it can be observed that the performance gap strongly depends on the GPU architecture. Considering NVIDIA GPUs, the performance improves with more modern architectures. Specifically, the Chapel-based implementation is on average 13% slower on the NVIDIA P100 (released in 2016), 7% slower on the NVIDIA V100 (released in 2018), and 3% faster on the NVIDIA A100 (released in 2020). This is largely due to the fact that modern architectures offer greater capabilities in terms of core count, memory storage, and bandwidth, and so forth. Furthermore, optimizing compiler performance on older GPU architectures, such as the NVIDIA P100, is of limited interest to language developers targeting exascale, like Chapel, as these architectures are no longer (or only minimally) used in modern systems [2]. Similar observations are made considering the AMD GPUs, where the best average performance is achieved using the AMD MI300X GPU.

The experiments on PFSP instances reveal similar behaviors, but the performance gaps between the PGAS approach and the low-level baselines are now much more pronounced. For

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

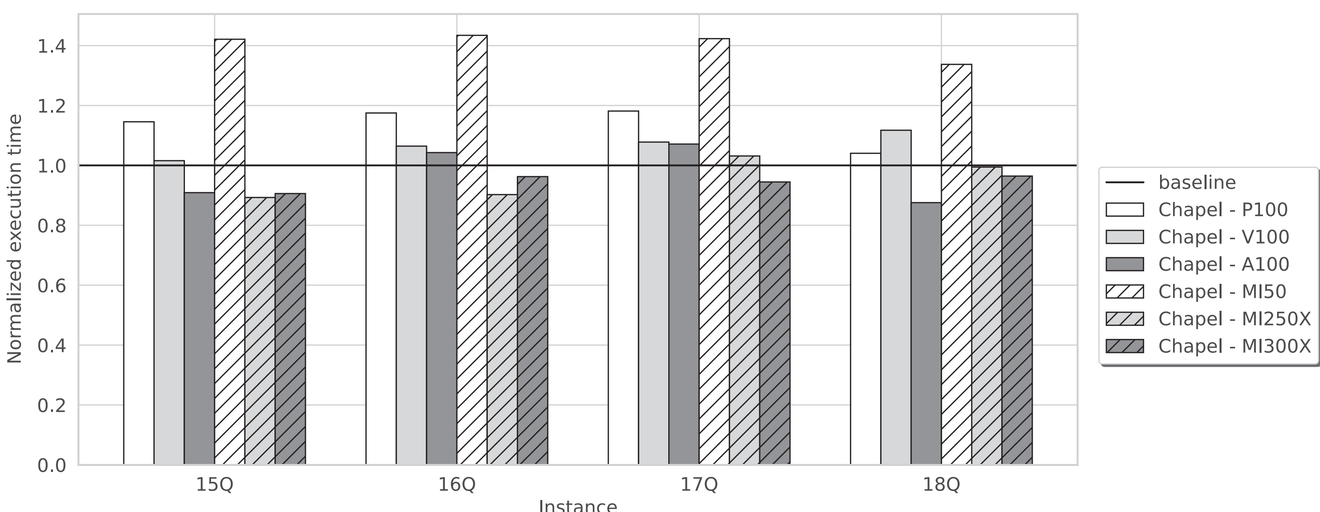

FIGURE 9 | Normalized execution time of the single-GPU Chapel code solving N-Queens instances on different NVIDIA and AMD GPU architectures, compared to CUDA/HIP-based baseline implementations.

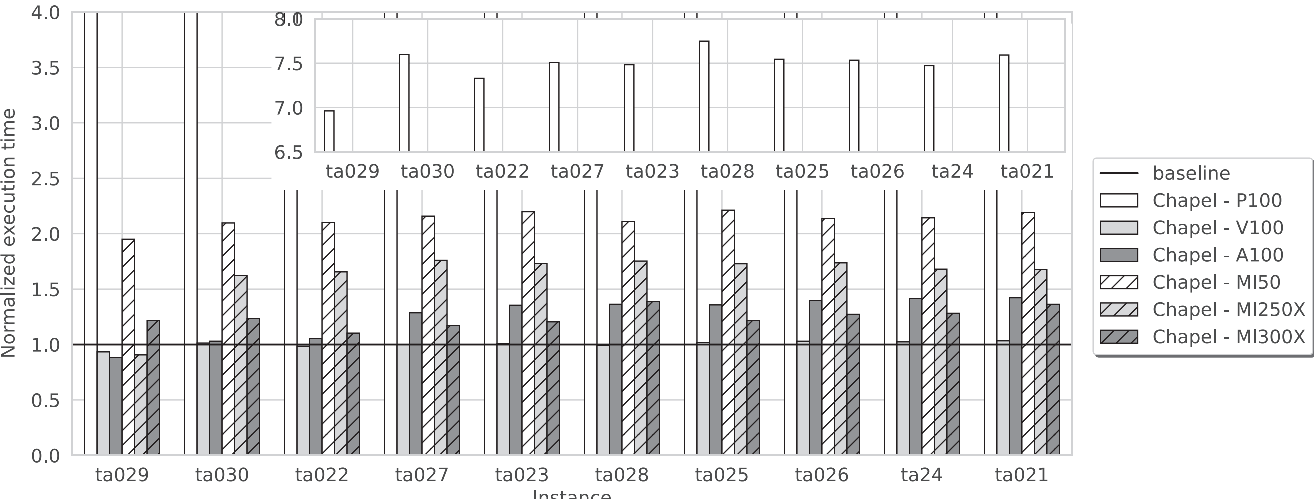

FIGURE 10 | Normalized execution time of the single-GPU Chapel code solving PFSP instances on different NVIDIA and AMD GPU architectures, compared to CUDA/HIP-based baseline implementations.

example, the Chapel approach is on average 62% slower than the HIP baseline on the AMD MI250X GPU, whereas it was 4% faster on the N-Queens problem. The key difference between the two problems lies in the bounding function of the B&B algorithm. For the N-Queens problem, the bounding function is relatively simple, relying only on the depth of the tree node and the permutation that represents the associated subproblem to determine satisfiability. In contrast, the PFSP requires a more complex bounding function, as it must account for cumulative processing times across multiple machines and jobs to compute the lower bound. Storing data, such as constant matrices, on the GPU device reduces memory transfer overhead and provides faster access during computation. However, this may increase the demands on memory usage and management, especially on GPUs with limited memory capacity, leading to a severe degradation in performance.

In investigating the performance gap between the Chapel and the baseline codes, we detected that the Chapel compiler may throw off Loop-Invariant Code Motion (LICM) in a few specific cases. LICM is a compiler optimization that performs automatically the movement of statements or expressions that can be moved outside the body of a loop without affecting the semantics of the program [34]. In programs implemented using high-level PGAS languages, such as Chapel, LICM typically occurs when arrays, which are not merely simple arrays but also embed additional data or metadata, are used within loops. For instance, in contrast to C arrays that are essentially contiguous blocks of memory that store elements of a specific data type, Chapel arrays also include metadata such as their size, domain (index space), distribution, and so forth. As a result, accessing redundant metadata repeatedly within the loop may result in performance degradation in the absence of LICM, compared to simpler baseline codes where such operations are either absent or easily optimized away.

Algorithm **3** illustrates the LICM optimization that is usually performed by compilers to deal with this issue. It consists of isolating array pointers towards contiguous blocks of memory

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

ALGORITHM 3 | Example of LICM optimization for array access in PGAS programs.

```
Require: n - number of iterations
Require: A - array of constant data
```

Without LICM:

```
1: result ← 0;
2: for i from 1 to n do
3:    result ← result + A[i] * i;
4: end for
```

With LICM:

```
1: result ← 0;
2: Aptr ← constPtr(A[0]);
3: for i from 1 to n do
4:    result ← result + Aptr[i] * i;
5: end for
```

TABLE 3 | Performance improvement (%) with manual LICM for solving the `ta023` PFSP instance in a single-GPU configuration.

GPU architecture	NVIDIA P100	NVIDIA V100	NVIDIA A100	AMD MI50	AMD MI250X	AMD MI300X
Performance gain	10%	17%	26%	26%	11%	7%

in dedicated variables (here *Aptr*) and using the latter inside the loop, thus avoiding any metadata access. In practice, we observed that the Chapel compiler may fail LICM in certain scenarios, thus degrading performance especially by solving PFSP instances, where the problem data are stored in constant arrays on the device. Although it is not possible to identify the root cause of the non-triggering of the optimization in the Chapel compiler without a low-level analysis of it (which goes beyond the scope of this paper), doing LICM manually in inner-most parts of our code allows reducing the execution time from 7% on the MI300X GPU to 26% on the MI50 GPU on the tested instance, as shown in Table 3.

LICM is not the only factor that can explain the difference in performance between Chapel and its baseline counterparts. Related works investigating the performance portability of Chapel in a different context also reported that the Chapel compiler may not take advantage of register coalescing optimizations provided by LLVM, which are available to the GPU back-end code generation [26]. Register coalescing is another compiler optimization technique aimed at improving the efficiency of register usage by reducing the number of move instructions between registers. This optimization identifies situations where multiple registers hold values that could be stored in a single register without affecting the program's correctness, and then "coalesces" (merges) them into one register, eliminating unnecessary data transfers between registers. Failing this optimization may result in a higher register pressure and, thus, lower warp occupancy, reducing parallel execution.

4.4 | Calibration of Parameters for AMD MI250X GPUs

The experiments presented in this section aim to determine the pair of parameters (m, M) that optimize the execution time of the proposed GPU-accelerated B&B algorithm on AMD MI250X. These parameters influence different parts of the algorithm, and careful tuning is essential to fully exploit the capabilities of the target GPU architecture.

In particular, during the initial search phase described in Section 3.1.1, the process terminates when $m \times \text{numGPUs} \times \text{numComputeNodes}$ tree nodes are pending evaluation. As this phase is sequential, increasing m raises the termination threshold, thereby reducing parallelism and overall efficiency. Conversely, reducing m shortens the sequential phase but may leave GPUs with insufficient initial work, hindering their ability to begin traversal effectively. This can lead to an increased number of WS operations and increased contention on pool 0.

The parameters m and M also govern the volume and frequency of host-device communication during the multi-GPU exploration. If m is set too low, the cost of frequent data transfers may overcome the benefits of parallel execution, resulting in sub-optimal performance. On the other hand, a large M increases the chunk size sent to the GPU, which can exceed the available memory or lead to prolonged transfer times, both of which negatively impact performance.

Table 4 shows the execution time of the Chapel single-GPU B&B implementation solving the `ta030` PFSP instance on AMD MI250X, and varying the parameters m and M. The range of values shown in the table corresponds to the neighborhood of the optimal solution found. It can first be observed that, for nearly all tested values of m, setting $M = 500{,}000$ yields the lowest execution time. Furthermore, for this value of M, choosing $m \in [50; 60]$ tends to produce the best performance. Overall, this configuration leads to at least a 5% improvement compared to the worst-case execution time on this instance. The tuning of (m, M) has been repeated across several PFSP and N-Queens instances, and in most cases, setting $m = 50$ and $M = 500{,}000$ consistently delivers good performance. These parameter values are therefore adopted in the remainder of this study.

4.5 | Strong Scaling Analysis

Tables 5 and 6 present the strong scaling results of our PGAS-based GPU-accelerated B&B algorithm at the intra- and inter-node levels, respectively. Performance is reported in terms of thousands of nodes explored per second (kn/s) and the corresponding speed-up relative to the optimized single-GPU or single-node multi-GPU baseline.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 | Execution times (seconds) of the Chapel single-GPU B&B implementation for different pairs of values (m, M) and solving the `ta030` PFSP instance.

m	*M*						
	50,000	**100,000**	**200,000**	**300,000**	**400,000**	**500,000**	**600,000**
10	9.78	9.55	9.41	9.42	9.36	9.36	9.42
20	9.78	9.52	9.4	9.39	9.38	9.36	9.38
30	9.78	9.56	9.41	9.4	9.36	9.36	9.38
40	9.78	9.56	9.4	9.38	9.37	9.37	9.37
50	9.78	9.55	9.41	9.38	9.37	9.36	9.36
60	9.79	9.56	9.4	9.38	9.37	9.36	9.36
70	9.78	9.56	9.42	9.4	9.36	9.36	9.37
80	9.78	9.55	9.41	9.38	9.36	9.37	9.38
90	9.79	9.56	9.41	9.37	9.37	9.36	9.38
100	9.80	9.56	9.41	9.39	9.36	9.36	9.39

Note: The colour in each cell indicates the quality of the value in that cell. Green marks the minimum (best) value, red marks the maximum (worst) value.

TABLE 5 | Strong scaling efficiency achieved by the GPU-accelerated B&B considering the intra-node level.

Instance	**GPU × 1**	**GPU × 2**		**GPU × 4**		**GPU × 8**	
	kn/s	***kn/s***	**speed-up**	***kn/s***	**speed-up**	***kn/s***	**speed-up**
15Q	22,564.1	43,228.1	1.91	79,630.7	3.53	119,030.2	5.28
16Q	21,875.8	43,795.1	1.99	83,537.2	3.82	145,488.1	6.65
17Q	21,819.7	43,147.4	1.98	82,898.6	3.80	149,695.9	6.86
AVG	22,086.5	43,390.2	1.96	82,022.1	3.71	138,071.4	6.26
`ta029`	2059.3	2749.4	1.33	3760.0	1.82	5564.8	2.70
`ta030`	2120.9	3018.5	1.42	4531.1	2.13	5728.1	2.70
`ta022`	2115.7	2894.2	1.36	5189.3	2.45	5963.0	2.81
`ta027`	2152.0	3603.1	1.67	6195.8	2.87	10,427.7	4.84
`ta023`	2186.6	3862.7	1.76	6764.6	3.09	12,685.3	5.80
`ta028`	2146.5	3951.9	1.84	6719.4	3.13	12,732.7	5.93
`ta025`	2195.4	4020.3	1.83	6974.8	3.17	13,742.5	6.25
`ta026`	2153.5	4032.9	1.87	7180.1	3.33	14,326.8	6.65
`ta024`	2160.4	4121.0	1.90	7542.9	3.49	15,015.2	6.95
`ta021`	2218.2	4232.2	1.90	7739.5	3.48	15,437.0	6.95
AVG	2150.8	3648.6	1.68	6259.7	2.89	11,162.3	5.15

Note: Instances are sorted by the number of nodes.

TABLE 6 | Strong scaling efficiency achieved by the GPU-accelerated B&B considering the inter-node level.

Instance	**Node × 1**	**Node × 8**		**Node × 16**		**Node × 32**		**Node × 64**		**Node × 128**	
	kn/s	***kn/s***	**speed-up**	***kn/s***	**speed-up**	***kn/s***	**speed-up**	***kn/s***	**speed-up**	***kn/s***	**speed-up**
17Q	185,947.3	749,772.9	4.03	1,479,164.4	7.95	2,886,915.4	15.52	5,514,769.0	29.66	8,189,439.6	44.04
18Q	187,958.1	754,585.6	4.01	1,502,010.0	7.99	2,967,609.6	15.78	5,905,864.1	31.42	11,386,895.3	60.58
19Q	186,002.2	751,300.9	4.04	1,504,256.4	8.08	3,002,350.3	16.14	6,008,376.6	32.30	11,961,181.1	64.30
AVG	186,635.8	751,886.4	4.02	1,495,143.6	8.00	2,952,291.7	15.81	5,809,669.9	31.12	10,512,505.3	56.30
`ta026`	14,326.8	69,657.3	4.86	94,946.3	6.62	94,499.8	6.59	69,287.3	4.83	34,278.4	2.39
`ta024`	15,015.2	91,847.8	6.11	153,999.1	10.25	213,169.4	14.19	263,612.7	17.55	141,007.9	9.39
`ta021`	15,437.0	98,370.2	6.37	178,555.1	11.56	296,992.6	19.23	363,863.9	23.57	259,160.5	16.78
AVG	14,926.3	86,625.1	5.78	142,500.1	9.47	201,553.9	13.33	232,254.6	15.31	145,815.6	9.52

Note: Instances are sorted by the number of nodes.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 7 | Comparison of execution times (seconds) with intra-node WS enabled/disabled using 8 GPUs at intra-node level.

	15Q	16Q	17Q	ta029	ta030	ta022	ta027	ta023	ta028	ta025	ta026	ta024	ta021
WS disabled	1.46	8.02	65.73	**3.62**	**4.59**	8.07	12.97	21.47	81.86	43.70	112.24	381.50	725.91
WS enabled	**1.44**	**7.84**	**53.56**	3.69	4.71	**5.20**	**9.71**	**17.03**	**27.20**	**30.33**	**62.94**	**254.49**	**448.53**

Note: Best results are shown in bold.

At the intra-node level, the B&B algorithm demonstrates nearly ideal scaling for the N-Queens instances. Speed-ups reach up to 6.86× with 8 GPUs for instance 17Q, corresponding to a strong scaling efficiency of 85%. The average efficiency across the three tested N-Queens instances is approximately 78%, highlighting the ability of the algorithm to efficiently leverage multiple GPUs within a compute node when the search space is sufficiently large. On the other hand, for PFSP instances, scaling is more moderate. The average speed-up at 8 GPUs is 5.15×, translating to an average efficiency of 64%. This reduced efficiency can be attributed to the more irregular and imbalanced nature of the search trees in PFSP instances, which introduces greater overhead in WS and synchronization. Indeed, in the N-Queens problem, the search tree exhibits some regularity due to board symmetries [33], whereas the PFSP problem lacks such patterns. Additionally, the problem granularity—reflected by the cost of evaluating tree nodes—is noticeably coarser for PFSP compared to N-Queens, as evidenced by average node evaluation rates of approximately 11×10^6 and 138×10^6 nodes per second on 8 GPUs, respectively.

Inter-node scaling results further confirm the robustness of the proposed parallelization strategy for the N-Queens problem. For the larger N-Queens instances, speed-ups increase almost linearly up to 128 compute nodes, with 64.3× acceleration achieved for 19Q. The average efficiency across instances remains reasonably high on the largest scale tested, staying above 50%. In contrast, PFSP instances again exhibit less favorable scaling. Although notable gains are observed up to 32 compute nodes (e.g., 19.23× for `ta021`), performance begins to plateau or even degrade beyond this point. The average efficiency drops to 7.4% at 128 nodes, mainly due to increasing communication costs and work imbalance across distributed resources. This level of scalability indicates that the PGAS-based design can leverage substantial parallelism. However, the observed efficiency loss at larger scales reveals limitations in the current approach, motivating a closer look at the WS mechanism and its impact on performance.

4.6 | Evaluation of the Work-Stealing Mechanism

We evaluate the benefits of our WS mechanism by comparing runs with WS enabled and disabled. The analysis considers both execution time and load balancing at intra- and inter-node parallel levels.

Table 7 compares the execution times when intra-node WS is either enabled or disabled, using a single compute node with 8 GPUs. In almost all cases, the data reveal that enabling intra-node WS consistently reduces execution times, especially as the number of explored nodes increases. An average speed-up of 24% is observed, along with a best-case speed-up of 66% solving `ta028`. For the two smallest instances of PFSP (`ta029` and `ta030`), where WS may not be justified, we observe a performance degradation of only 2%–2.5%.

Figure 11 provides deeper insight into the efficiency of the intra-node WS mechanism by showing statistics on the overall distribution of workload between GPUs. When solving N-Queens instances, we observe that the initial search and static workload distribution described in Section 3.1.1 allow good load balancing, even in the absence of intra-WS. This can be attributed to the inherent symmetries of the chessboard introducing regularities in the structure of the B&B tree. However, in the case of the PFSP problem, there is significant variation in both the results across instances and the workload distribution among GPUs within a single instance. For example, the maximum observed workload share reaches approximately 53% solving `ta028`, meaning that a single GPU explored more than half of the B&B tree alone—greatly reducing the effective parallelism for the remaining 7 GPUs. Enabling intra-node WS in that case significantly rebalanced the workload, with each GPU now handling between 10% and 15% of the total number of tree nodes, compared to the ideal uniform distribution of 12.5%. More generally, the intra-node WS mechanism consistently and significantly improves load balance, producing near-equal distributions for the largest PFSP instances, such as `ta024` and `ta021`.

A similar analysis is carried out at the inter-node level, with intra-node WS kept enabled throughout. As shown in Table 8, the results are more nuanced this time for the PFSP problem. For the three instances considered, our algorithm performs better without inter-node WS when using 128 compute nodes. This clearly indicates that the communication overhead and spin-lock synchronizations associated with inter-node WS are too high. This explains the limitation in strong scaling efficiency observed when solving PFSP instances at scale in the previous section. We also note that the larger the instance in terms of tree nodes, the smaller these overheads become—with the performance drop decreasing from 95% for instance `ta026` to just 10% for `ta021`. This suggests that our approach could become more effective on even larger PFSP instances. In the case of the N-Queens problem, execution times are largely unaffected by the inter-node WS mechanism, again due to the regularity of the problem, which drastically reduces stealing activity. As a result, speed-ups of up to 8% are observed for the largest instances.

The load balancing analysis shown in Figure 12 reveals trends similar to those observed at the intra-node level. First, we note the near-perfect balance achieved for the N-Queens instances, made possible by the initial search mechanism and static distribution strategy. For the PFSP instances, we observe an improved balance when inter-node WS is enabled, particularly through the narrowing of the minimum and maximum workload values toward the average (approximately 0.78%). For instance, the minimum

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

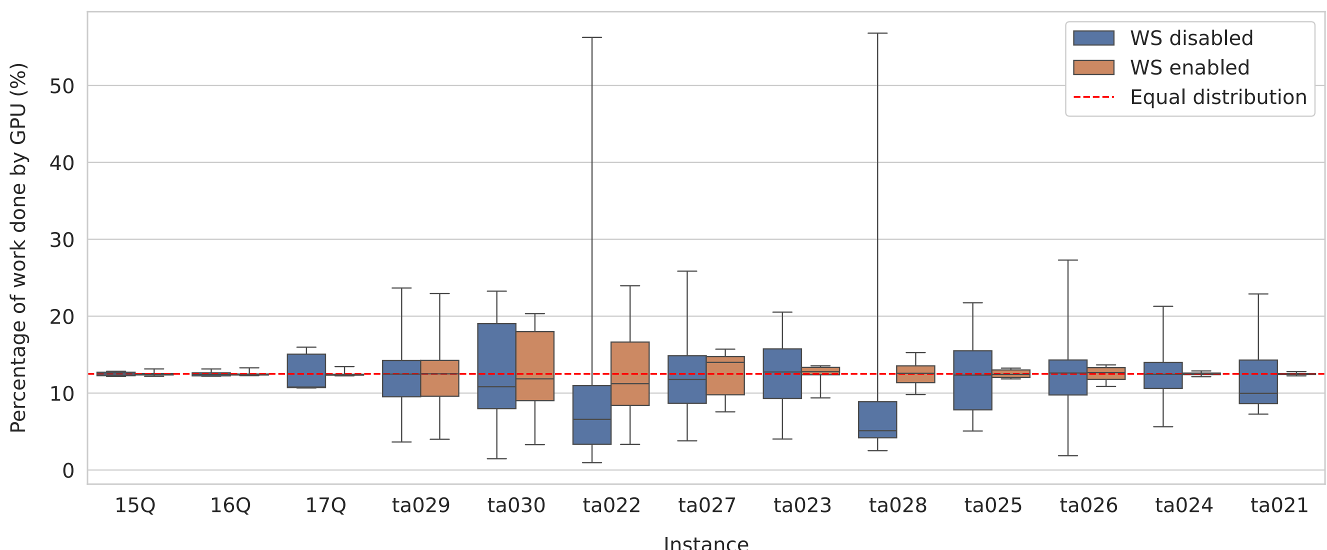

FIGURE 11 | Boxplot illustrating the distribution of workload across GPUs for all instances at the intra-node level, with intra-node WS enabled/disabled. The box represents the interquartile range, with the central line indicating the median workload. Whiskers extend from the minimum to the maximum workload observed across GPUs, including all values without outlier exclusion. A horizontal red dashed line at 12.5% represents the ideal workload per GPU, assuming equal workload distribution across 8 GPUs.

TABLE 8 | Comparison of execution times (seconds) with inter-node WS enabled/disabled using 128 compute nodes (each including 8 GPUs) at the inter-node level.

	17Q	18Q	19Q	ta026	ta024	ta021
WS disabled	**0.79**	5.48	41.93	**7.68**	**8.42**	**13.73**
WS enabled	0.98	**5.21**	**38.62**	15.00	15.41	15.22

Note: Best results are shown in bold.

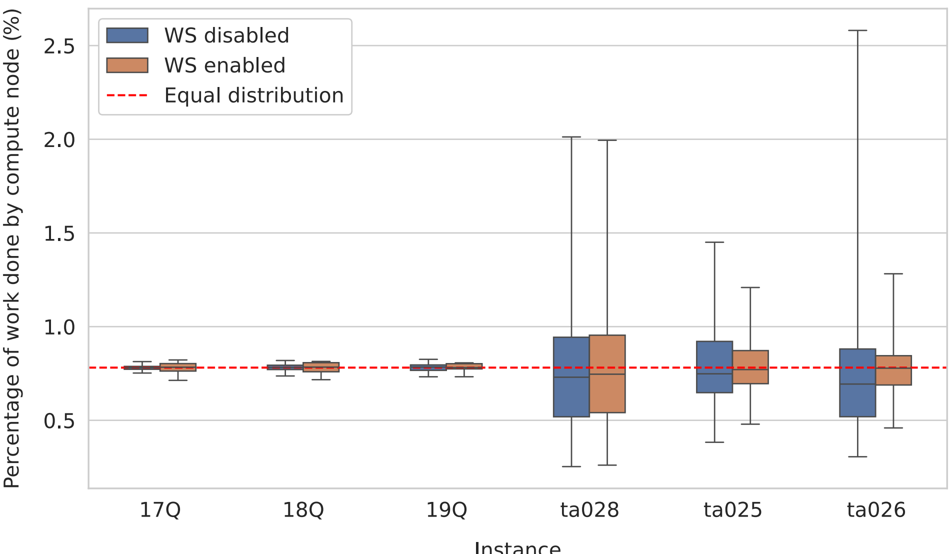

FIGURE 12 | Boxplot illustrating the distribution of workload across compute nodes for all instances at the inter-node level, with inter-node WS enabled/disabled. The box represents the interquartile range, with the central line indicating the median workload. Whiskers extend from the minimum to the maximum workload observed across compute nodes, including all values without outlier exclusion. A horizontal red dashed line at 0.78125% represents the ideal workload per compute node assuming equal workload distribution across 128 nodes.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

workload increases from 0.30% to 0.46%, whereas the maximum decreases from 2.58% to 1.28%, respectively, solving `ta026`.

5 | Discussion

One of the key advantages of PGAS lies in its ability to unify multiple levels of parallelism under a single high-level programming model. In the context of this work, this feature allows for seamless coordination between CPU-based tree exploration, GPU-based bounding, and distributed-memory parallelism across compute nodes. Chapel, as a PGAS language, provides abstractions, such as locales and distributed arrays, that simplify the management of data and tasks across heterogeneous resources. This eliminates the need for separate parallel programming models (e.g., OpenMP for CPU, CUDA for GPU, MPI for inter-node), reducing code complexity and improving maintainability. With a single source program, developers can target massively parallel systems while expressing parallelism at different levels declaratively and coherently.

Another major benefit of the PGAS-based Chapel language is its support for performance portability across diverse hardware platforms. Thanks to its vendor-neutral GPU back end, the same code can be executed on NVIDIA and AMD GPUs without requiring architecture-specific modifications, also achieving performance portability. The Chapel implementation achieves equivalent performance across six distinct GPU architectures, with acceptable overheads relative to optimized CUDA/HIP baselines. However, performance disparities were observed, particularly on coarse-grained problems like PFSP, due to missed compiler optimizations such as LICM and register coalescing. The manual application of these optimizations resulted in noticeable performance improvements. Nevertheless, this portability is especially valuable in the evolving HPC landscape, where supercomputers often combine GPUs from different vendors, and long-term maintainability requires minimizing dependence on hardware-specific features.

Despite its high-level abstractions, PGAS does not relieve the programmer from handling complex algorithmic mechanisms required by irregular parallel workloads. In our implementation of WS, for example, we must still design and orchestrate critical components manually, including global termination detection, concurrent access control, and dynamic victim selection strategies. These challenges are not addressed by the PGAS memory model itself, but instead require thoughtful algorithmic engineering. For example, our algorithm's performance was found to be sensitive to the tuning of its parameters, particularly the values of m and M, which implicitly govern the frequency and size of host-device data transfers. On AMD MI250X GPUs, the configuration $m = 50$, $M = 500{,}000$ consistently delivered good performance across a range of instances. Moreover, while the strong scaling analysis showed that the fine-grained N-Queens problem achieved speed-ups of up to 64× on 128 nodes, the PFSP instances experienced reduced inter-node scaling efficiency at larger scales due to synchronization bottlenecks. Although PGAS simplifies how data is accessed, for example using `remotePool[i]` rather than sending explicit messages, the decisions of *when* to steal work, *whom* to steal it from, and *how* to avoid oversubscription must still be addressed at the application level. Thus, PGAS eases communication but not coordination, and effective use still depends on parallel algorithm design.

6 | Conclusions and Future Works

We provided the design and implementation of a GPU-accelerated B&B algorithm based on PGAS. This choice was motivated by the high-level abstraction of the programming model, which enhances programmability, and by the vendor-neutral GPU features of the PGAS-based Chapel language, which favor GPU portability. The algorithm combines parallel tree exploration on the CPU with parallel evaluation of bounds on the GPU to accelerate the compute-intensive bounding operator of the B&B algorithm. The workload irregularity is managed by a multi-level dynamic load-balancing mechanism based on WS. From the implementation aspect, the use of Chapel allows high-level abstractions that seamlessly integrate multiple levels of parallelism—CPU, GPU, and inter-node—within a unified programming language. In addition, the portability challenge is addressed by the vendor-neutral GPU support of the language. The algorithm is generic with regard to the tackled optimization problem and has been tested on two different challenging problems: PFSP and N-Queens. For comparison purposes, an optimized CUDA-based baseline implementation has also been implemented. We investigated the code performance and code portability of the algorithms on several GPU architectures and also performed large-scale experimentation on a pre-exascale supercomputer.

The experimental evaluation revealed several important findings. First, intra-node WS proved highly effective in reducing execution times and improving load balance for highly irregular problems such as PFSP. In contrast, inter-node WS introduced significant communication and synchronization overheads, which outweighed its benefits for small to medium problem sizes. Nonetheless, as the number of tree nodes increased, these overheads diminished, suggesting potential benefits for larger-scale instances. The GPU-accelerated B&B algorithm showed portable performance across six Nvidia and AMD architectures, with Chapel enabling unified multi-level parallelism. Performance gaps on coarse-grained problems like PFSP were linked to missed compiler optimizations, that, when applied manually, improved results. Tuning parameters m and M was critical, with $m = 50$, $M = 500{,}000$ performing well on AMD MI250X GPUs. Strong scaling was observed for N-Queens, whereas the PFSP scaling efficiency declined at large scales.

The results revealed that inter-node communications and spin-lock synchronization are responsible for the performance limitation observed at scale. One future direction is therefore to explore advanced data structures specifically designed for exascale computing, incorporating a sophisticated locking mechanism. For example, the PGAS-based `distBag` data structure specifically designed for the parallel depth-first traversal of unbalanced trees will be experimented [35]. Another factor limiting portability is the tuning of algorithm parameters, such as m and M, which are highly dependent on the target hardware and are set manually in this work. Consequently, future efforts will focus on designing and implementing adaptive methods

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

that hybridize B&B with heuristic approaches for automatic parameter tuning [36].

Another important aspect not addressed in this work is fault tolerance. As HPC platforms scale to thousands of nodes and beyond, the probability of hardware or software failures during execution becomes non-negligible [37]. Traditional message-passing models like MPI have begun incorporating features for resilience, such as User-Level Failure Mitigation. In contrast, PGAS models still assume failure-free execution and do not provide standard mechanisms for detecting, recovering from, or adapting to node or device failures. As such, building fault-tolerant PGAS-based applications remains a manual and largely unexplored process. Addressing this gap will be essential to making PGAS viable for exascale systems and long-running applications subject to hardware volatility.

By combining a high level of abstraction, code portability, and code performance, the proposed approach offers a promising foundation for tackling combinatorial optimization at a large scale. Its generality makes it suitable for a wide range of B&B applications, whereas its ability to run on diverse supercomputing platforms positions it as a valuable tool for large-scale scientific and industrial computations.

Acknowledgments

This work was supported by the Agence Nationale de la Recherche [grant number ANR-22-CE46-0011] and the Fonds National de la Recherche Luxembourg (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 acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI (EHPC-DEV-2024D03-066). In addition to LUMI, some of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations; and using the HPC facilities of the University of Luxembourg. The authors gratefully acknowledge the Chapel's development team for their expert support.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

All code written in support of this publication is publicly available at https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel and archived on Zenodo (DOI: 10.5281/zenodo.10786275).

Endnotes

[1] LUMI is ranked 9th in the latest TOP500 ranking (June 2025).

References

1. B. Gendron and T. G. Crainic, "Parallel Branch-and-Branch Algorithms: Survey and Synthesis," *Operations Research* 42, no. 6 (1994): 1042–1066, https://doi.org/10.1287/opre.42.6.1042.

2. TOP500.org, "TOP500 the List (June 2025)," https://www.top500.org/lists/top500/2025/06/.

3. G. Almasi, "PGAS (Partitioned Global Address Space) Languages," in *Encyclopedia of Parallel Computing* (Springer, 2011), 1539–1545, https://doi.org/10.1007/978-0-387-09766-4_210.

4. G. Helbecque, E. Krishnasamy, T. Carneiro, N. Melab, and P. Bouvry, "A Chapel-Based Multi-GPU Branch-and-Bound Algorithm," in *Euro-Par 2024: Parallel Processing Workshops* (Springer Nature Switzerland, 2025), 463–474, https://doi.org/10.1007/978-3-031-90200-0_37.

5. J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens, "A Computationally Efficient Branch-And-Bound Algorithm for the Permutation Flow-Shop Scheduling Problem," *European Journal of Operational Research* 284, no. 3 (2020): 814–833, https://doi.org/10.1016/j.ejor.2020.01.039.

6. N. Melab, "Contributions à la Résolution de Problèmes D'optimisation Combinatoire Sur Grilles de Calcul. Université Des Sciences et Technologies de Lille. Thèse HDR," (2005).

7. E. H. Wu and Y. Q. Liu, "Emerging Technology About GPGPU," in *APCCAS 2008–2008 IEEE Asia Pacific Conference on Circuits and Systems* (IEEE, 2008), 618–622, https://doi.org/10.1109/APCCAS.2008.4746099.

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 *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2016), 747–755, https://doi.org/10.1109/IPDPSW.2016.14.

9. J. Gmys, "Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers," *INFORMS Journal on Computing* 34, no. 5 (2022): 2502–2522, https://doi.org/10.1287/ijoc.2022.1193.

10. T. T. Vu and B. Derbel, "Parallel Branch-And-Bound in Multi-Core Multi-CPU Multi-GPU Heterogeneous Environments," *Future Generation Computer Systems* 56 (2016): 95–109, https://doi.org/10.1016/j.future.2015.10.009.

11. M. E. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (IEEE, 2012), 1769–1777, https://doi.org/10.1109/IPDPSW.2012.219.

12. J. Shen, K. Shigeoka, F. Ino, and K. Hagihara, "GPU-Based Branch-And-Bound Method to Solve Large 0-1 Knapsack Problems With Data-Centric Strategies," *Concurrency and Computation: Practice and Experience* 31 (2019): e4954, https://doi.org/10.1002/cpe.4954.

13. W. Z. Yeoh, J. S. Teh, and J. Chen, "Automated Search for Block Cipher Differentials: A GPU-Accelerated Branch-and-Bound Algorithm," in *In Australasian Conference on Information Security and Privacy* (Springer International Publishing, 2020), 160–179, https://doi.org/10.1007/978-3-030-55304-3_9.

14. A. Borisenko, M. Haidl, and S. Gorlatch, "A GPU Parallelization of Branch-And-Bound for Multiproduct Batch Plants Optimization," *Journal of Supercomputing* 73, no. 2 (2017): 639–651, https://doi.org/10.1007/s11227-016-1784-x.

15. L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A Parallel Algorithm for Game Tree Search Using GPGPU," *IEEE Transactions on Parallel and Distributed Systems* 26, no. 8 (2015): 2114–2127, https://doi.org/10.1109/TPDS.2014.2345054.

16. K. Rocki and R. Suda, "Parallel Minimax Tree Searching on GPU," in *In International Conference on Parallel Processing and Applied Mathematics* (Springer Berlin Heidelberg, 2010), 449–456, https://doi.org/10.1007/978-3-642-14390-8_47.

17. X. Meyer, B. Chopard, and P. Albuquerque, "A Branch-And-Bound Algorithm Using Multiple GPU-Based LP Solvers," in *20th Annual International Conference on High Performance Computing* (IEEE, 2013), 129–138, https://doi.org/10.1109/HiPC.2013.6799105.

18. T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. Lima de Campos, "A New Parallel Schema for Branch-And-Bound Algorithms Using GPGPU," in *Proceedings of the 2011 23rd International Symposium on Computer Architecture and High Performance Computing* (IEEE, 2011), 41–47, https://doi.org/10.1109/SBAC-PAD.2011.20.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

19. I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens, "Combining Multi-Core and GPU Computing for Solving Combinatorial Optimization Problems," *Journal of Parallel and Distributed Computing* 73, no. 12 (2013): 1563–1577, https://doi.org/10.1016/j.jpdc.2013.07.023.

20. 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* 29, no. 9 (2017): e4019, https://doi.org/10.1002/cpe.4019.

21. L. Chen, L. Liu, S. Tang, et al., "Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation," in *Languages and Compilers for Parallel Computing* (Springer Berlin Heidelberg, 2011), 151–165, https://doi.org/10.1007/978-3-642-19595-2_11.

22. D. Cunningham, R. Bordawekar, and V. Saraswat, "GPU Programming in a High Level Language: Compiling X10 to CUDA," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* (ACM, 2011), 1–10, https://doi.org/10.1145/2212736.2212744.

23. A. Hayashi, S. R. Paul, and V. Sarkar, "A Multi-Level Platform-Independent GPU API for High-Level Programming Models," in *High Performance Computing. ISC High Performance 2022 International Workshops* (Springer International Publishing, 2022), 90–107, https://doi.org/10.1007/978-3-031-23220-6_7.

24. T. Carneiro, N. Melab, A. Hayashi, and V. Sarkar, "Towards Chapel-Based Exascale Tree Search Algorithms: Dealing With Multiple GPU Accelerators," (2021), 18th International Conference on High Performance Computing & Simulation.

25. B. L. Chamberlain, E. Ronaghan, B. Albrecht, et al., "Chapel Comes of Age: Making Scalable Programming Productive," (2018), Cray User Group.

26. J. Milthorpe, X. Wang, and A. Azizi, "Performance Portability of the Chapel Language on Heterogeneous Architectures," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2024), 6–13, https://doi.org/10.1109/IPDPSW63119.2024.00011.

27. T. Carneiro, E. Kayraklioglu, G. Helbecque, and N. Melab, "Investigating Portability in Chapel for Tree-Based Optimization on GPU-Powered Clusters," in *Euro-Par 2024: Parallel Processing* (Springer Nature, 2024), 386–399, https://doi.org/10.1007/978-3-031-69583-4_27.

28. 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 (IPDPSW)* (IEEE, 2024), 872–879, https://doi.org/10.1109/IPDPSW63119.2024.00156.

29. R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM* 46, no. 5 (1999): 720–748, https://doi.org/10.1145/324133.324234.

30. M. R. Garey, D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop Scheduling," *Mathematics of Operations Research* 1, no. 2 (1976): 117–129, https://doi.org/10.1287/moor.1.2.117.

31. B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem," *Operations Research* 26, no. 1 (1978): 53–67, https://doi.org/10.1287/opre.26.1.53.

32. E. Taillard, "Benchmarks for Basic Scheduling Problems," *European Journal of Operational Research* 64, no. 2 (1993): 278–285, https://doi.org/10.1016/0377-2217(93)90182-M.

33. C. Erbas, S. Sarkeshik, and M. M. Tanik, "Different Perspectives of the N-Queens Problem," in *Proceedings of the 1992 ACM Annual Conference on Communications* (ACM, 1992), 99–108, https://doi.org/10.1145/131214.131227.

34. S. Muchnick, *Advanced Compiler Design and Implementation* (Morgan Kaufmann Publishers Inc., 1998).

35. G. Helbecque, T. Carneiro, N. Melab, J. Gmys, and P. Bouvry, "PGAS Data Structure for Unbalanced Tree-Based Algorithms at Scale," in *Computational Science—ICCS 2024*, vol. 14834 (Springer Nature Switzerland, 2024), https://doi.org/10.1007/978-3-031-63759-9_13.

36. I. Chakroun and N. Melab, "An Adaptative Multi-GPU Based Branch-and-Bound. A Case Study: The Flow-Shop Scheduling Problem," in *IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (IEEE, 2012), 389–395, https://doi.org/10.1109/HPCC.2012.59.

37. F. Cappello, "Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications* 23, no. 3 (2009): 212–226, https://doi.org/10.1177/1094342009106189.

15320634, 2025, 25-26, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.70321 by IMEC, Wiley Online Library on [27/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License