Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators

Tiago Carneiro, Nouredine Melab, Akihiro Hayashi, Vivek Sarkar

Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators

Tiago Carneiro[†], Nouredine Melab[*], Akihiro Hayashi[‡], Vivek Sarkar[§]
University of Luxembourg[†], Luxembourg
INRIA Lille - Nord Europe, Université de Lille, CNRS/CRIStAL[*], France
Georgia Institute of Technology[‡§], USA
tiago.carneiropessoa@uni.lu, nouredine.melab@univ-lille.fr, {ahayashi,vsarkar}@gatech.edu

***Abstract*—Tree-based search algorithms applied to combinatorial optimization problems are highly irregular and time-consuming when it comes to solving big instances. Solving such instances efficiently requires the use of massively parallel distributed-memory supercomputers. According to recent Top500 trends, the degree of parallelism in these supercomputers continues to increase in size and complexity, with millions of heterogeneous (mainly CPU-GPU) cores. Harnessing this scale of computing resources raises at least three challenging issues which are described and addressed in this paper. Indeed, as a step towards exascale computing, we revisit the design and implementation of tree search algorithms dealing with multiple GPUs, in addition to scalability and productivity-awareness using Chapel. The proposed algorithm exploits Chapel's distributed iterators by combining a partial search strategy with pre-compiled CUDA kernels for more efficient exploitation of the intra-node parallelism. Extensive experimentation on big N-Queens problem instances using 24 GPUs shows that up to 90% of the linear speedup can be achieved.**

***Index Terms*—Backtracking, GPU, PGAS, Chapel.**

I. Introduction

Tree search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree. Algorithms from this class, such as backtracking and branch-and-bound (B&B), are popular to efficiently solve permutation combinatorial optimization problems (PCOPs), especially when combined with parallel/distributed computing. As the decision version of PCOP is often NP-complete, the size of problems that can be solved to optimality is limited, even if large-scale distributed computing is employed [1]. In this sense, the use of GPUs became crucial because it enables solving to the optimality instances having prohibitive execution times on CPUs [2], [3]. Also in the context of large-scale distributed computing, the use of accelerators such as GPUs and FPGAs plays a special role, as the energy-efficiency of such devices helps to break the power barrier towards exascale [4].

It is important to point out that such large-scale heterogeneous systems are going to be complex to program, and efforts towards productivity are crucial for better exploiting the future generation of supercomputers [5]. The DARPA High Productivity Computing System Program (HPCS) represents an effort for creating high-productivity languages for the next generation of supercomputers [6]. Among the HPCS high-productivity languages, Chapel stands out, as it is competitive to both C-OpenMP and MPI+X in terms of performance and scalability [7]. Existing work on distributed tree search [8] show that it is possible to achieve parallel efficiency and performance, but also high productivity, by using the Chapel language. However, due to Chapel's lack of official GPU support, related works do not employ GPUs, which is critical in the exact optimization field [3].

The objective of the present research is to deal with multiple GPU accelerators as a major step towards exascale using a high-productivity language. In this sense, we revisit the design and implementation of tree search algorithms taking into account GPU heterogeneity, in addition to scalability and productivity-awareness using Chapel. The proposed algorithm is a distributed GPU-based tree search algorithm for solving PCOPs. Chapel's lack of official GPU support is overcome by combining the language's features with pre-compiled CUDA kernels. Chapel's distributed iterators are used for work distribution and load balancing between computer nodes. For more efficient exploitation of the intra-node parallelism, each subproblem received by a computer node is processed locally on CPU and then offloaded to the local GPUs.

Extensive experimentation on big N-Queens problem instances using up to 24 GPUs shows that 90% of the linear speedup can be achieved compared to an optimized CUDA-C baseline. We also compare the proposed algorithm to another implementation using an existing approach from the literature that enables GPU support in Chapel [9]. Results show that the proposed algorithm is up to $1.77\times$ faster and almost $2\times$ more efficient than its counterpart, mainly due to its lack of distributed load balancing. It is worth to mention that we use the N-Queens problem as a proof-of-concept that motivates further improvements in solving related combinatorial optimization problems.

The remainder of this document is structured as follows. Section II brings background information and the related work. Section III details the distributed GPU-based tree search algorithm in Chapel. In turn, Section IV presents the performance evaluation of the proposed algorithm. Next, Section V brings a discussion of the reported experimental results. Finally, conclusions are outlined in Section VI.

II. Background and Related Work

A. Tree-based Search Algorithms

Tree search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree [10]. Algorithms that belong to this class start with an initial (root) node, which represents the initial state of the problem to be solved. Nodes are branched during the search process, generating children nodes more constrained than their father node. The generated nodes are evaluated, and then, the valid and feasible ones are stored in a pool-like data structure called *Active Set*. The search generates and evaluates nodes until the data structure is empty or another termination criterion is satisfied.

During the search, if an undesirable state is reached, the algorithm discards this node and then chooses an unexplored (frontier) node in the active set. This action prunes some regions of the solution space, preventing the algorithm from unnecessary computations. However, the pruning of subproblems makes the shape of the tree irregular, which might result in severe load imbalance when parallel computing is used. In this sense, load balancing schemes are crucial for achieving parallel efficiency in tree-based search algorithms.

B. The Chapel Language

Chapel is an open-source parallel programming language designed to improve productivity in high-performance computing. In Chapel, the program is started with a single task, and parallelism is *added* through data or task-parallel features [7]. Furthermore, as Chapel belongs to the partitioned global address space languages (PGAS), the application has a global memory addressing space, and each segment of this space is assigned to a different locale [11]. In Chapel, the term *locale* usually refers to a symmetric multiprocessing computer in a parallel system. Due to the PGAS, a task can refer to any variable lexically visible, whether this variable is placed on the same locale on which the task is running, or on the memory of another one. Moreover, indexes of data structures can be globally expressed, even in the case where the implementation of such data structures distributes the indexes across several locales.

Although Chapel is a parallel programming language, there is no official support for GPUs. There are solutions that translate the body of a `forall` loop into GPU code [12], [13]. However, these solutions may not always deliver the best possible performance on GPU, particularly when advanced GPU optimization techniques such as warp-shuffles and shared memory utilization are required.

C. Chapel's Parallel Iterators

Iterators in Chapel are similar to procedures that can be used to isolate iterations from the loop body. Each value yielded by the iterator corresponds to an iteration of the loop. Chapel provides different parallel and distributed iterators that implement load balancing between computer cores and locales. Related work on distributed tree search shows that the iterators provided by Chapel are the key feature to achieve a trade-off between productivity and parallel efficiency [8]. The complex communication pattern of a distributed master-worker algorithm is encapsulated by the iterators. As a consequence, there is no need for explicitly dealing with work distribution, load balancing, termination criteria, or metrics reduction. Due to Chapel's lack of official GPU support, related work do not deal with the CPU-GPU heterogeneity, which is critical in the exact optimization field.

As Chapel does not provide any iterator dedicated to GPUs, Hayashi *et al.* [9] propose the `GPUIterator` module that allows Chapel programmers to perform `forall` loops on both CPU and multi-GPU with minor additions to the code. The `GPUIterator` automatically divides a given iteration space into CPU and GPU portions by looking at a user-specified parameter, namely `CPUPercent`. A `forall` loop that is responsible for the CPU portion is executed on CPUs. On the other hand, the user needs to prepare a callback function that is expected to invoke a GPU program that is responsible for the GPU portion. It is worth noting that the `GPUiterator` also facilitates distributed execution. Also, in the current implementation, the `GPUIterator` does not dynamically update the CPU-GPU percentage to mitigate load-imbalance during the execution, neither provides distributed load balancing.

III. The Proposed Algorithm

This section presents a productivity-aware distributed CPU-GPU backtracking to solve permutation combinatorial problems. The algorithm follows the master-worker model and is implemented for enumerating *all* complete and feasible solutions to the N-Queens problem.

Backtracking is a fundamental tree-based search that dynamically enumerates a solution space in a depth-first fashion. Due to its low memory requirements and its ability to quickly find new solutions, depth-first search (DFS) is often preferred as a search strategy for branch-and-bound (B&B) search algorithms [14]. In turn, the N-Queens problem consists of placing N non-attacking queens on a $N \times N$ chessboard, and it is often used as a benchmark for novel tree-based search algorithms [15]. N-Queens is easily modeled as a permutation problem: position r of a permutation of size N designates the column in which a queen is placed in row r.

The concepts herein presented are similar to any PCOP and can be adapted for solving other problems with straightforward modifications [16], [17]. In this paper, we use the N-Queens problem as a proof-of-concept that motivates further improvements in solving related combinatorial optimization problems.

A. Initial Premises

One can see in Section II-B that a couple of approaches try to mitigate the gap between Chapel and GPUs. Those solutions are not feasible for the implementation of distributed tree-based search algorithms, as they require problem-specific data structures and several advanced programming features from the CUDA API [16], [17]. In turn, the `GPUIterator` module fulfills this requirement since it is designed to facilitate the

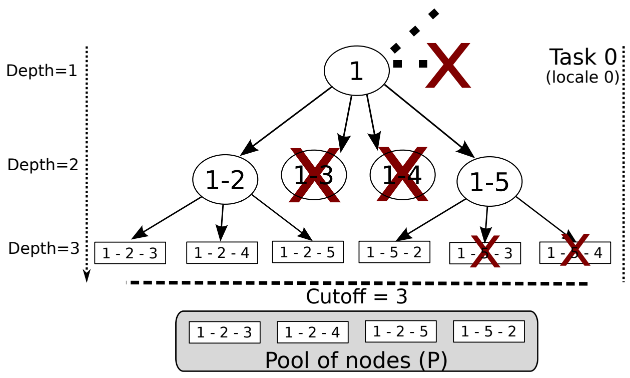


Fig. 1. Schematic representation of an initial search on *locale 0 - task 0* that generates the pool P for a problem size $N = 4$ and $cutoff = 3$. The figure depicts the branch that has the element 1 of the permutation as the root and generated 4 valid and feasible incomplete solutions at depth $cutoff = 3$.

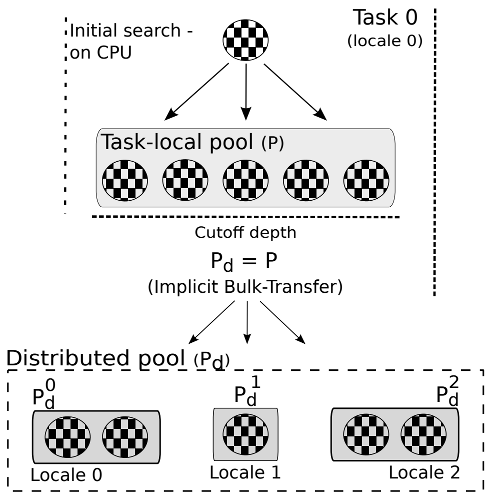


Fig. 2. The distributed pool P_d consists of several sets $P_d^i, i \in \{0, ..., L-1\}$, where L is the number of locales on which the application runs (based on [8]).

use of pre-compiled optimized kernels. However, its lack of distributed load balancing makes it unfeasible for the purpose of this work, since the irregular loads produced by tree search algorithms demand load balancing for efficient use of the computational resources [8]. In turn, Chapel's iterators are not well suited for the GPU programming model, as the loop body is executed for *each* element e of an iteration space. However, one element is not enough to use efficiently a massively parallel device such as a modern GPU. In order to cope with the challenge, we proceed as follows.

Firstly, the master locale generates initial load through a partial search on CPU, filling a pool of nodes. Chapel's distributed iterators are used for work distribution and load balancing among locales. Then, for more efficient exploitation of the intra-node parallelism, each subproblem received by a worker locale is processed locally on CPU, by another partial search, generating a task-local pool. Then, this set of subproblems is divided by the number of existing GPUs and then offloaded to the local devices. Details concerning the master locale and the initial search on CPU are presented in the next subsection.

Algorithm 1: The Master locale.

```
1  $N \leftarrow get\_problem(\,)$
2  $cutoff \leftarrow get\_cutoff\_depth(\,)$
3  $second\_cutoff \leftarrow get\_scnd\_cutoff\_depth(\,)$
4  $P \leftarrow \{\}\ Node$
5  $metrics \leftarrow (0, 0)$
6  $metrics \mathrel{+}= initial\_search(N, cutoff, P)$
7  $Size \leftarrow \{0..(|P|-1)\}$ // Domain
8  $D \leftarrow Size$ mapped onto locales to a standard distribution
9  $P_d \leftarrow [D] : Node$
10 $P_d = P$ // Using implicit bulk-transfer
11 forall node in $P_d$ following a distributed iterator with(+ reduce metrics) do
12     $metrics \mathrel{+}= Algorithm\_2(N, node, cutoff,$
13         $second\_cutoff)$
14 end
15 $present\_results(metrics)$
```

B. The Master Locale and the Initial Search

The algorithm starts with *task* 0 running on *locale* 0. As one can see in Algorithm 1, *task* 0 initially receives the size N of the problem, the first cutoff depth, and the second one (*lines* $1-3$). As illustrated in Figure 1, the initial pool of nodes P is generated though a partial search (*line* 6), called *initial search*. This procedure is performed by task 0 and implicitly enumerates all *feasible* and *valid* incomplete solutions containing $cutoff$ elements of the permutation, keeping them into the pool P.

Lines 7 to 9 are responsible for defining the distributed pool of nodes P_d. As one can see at line 7, initially the domain D $(0..|P|-1)$ is mapped onto the locales according to a distribution. Then, the distributed pool P_d is defined over the domain D. Finally, as depicted in Figure 2, P_d is initialized via implicit bulk transfer, represented by the assignment operator at line 10, which performs $L-1$ transfers through the network, where L is the number of locales on which the application runs, instead of $|P_d|$ small ones.

The parallel distributed search takes place in line 11, *adding* parallelism by using the `forall` statement along with distributed iterators (`DistributedIters`), which are responsible for the assignment of nodes in P_d to locales in a *master-worker* manner (distributed load balancing). Finally, there is no need for programming termination criteria or a reduction of the search metrics. The search finishes when the distributed active set P_d is empty, and metrics are reduced by using the *reduction intents* provided by Chapel (+ `reduce`).

C. Exploiting Intra-node Parallelism

As commented in Section III-A, one node yielded by the distributed iterator used in Algorithm 1 (*line* 11) is not enough for the efficient use of all GPUs a locale has. For the efficient use of the GPUs the locale is equipped with, we proceed as outlined in Algorithm 2.

Before the intra-locale search starts, a task-local pool P_l of type $Node$ is defined (*line* 2). Next, a second partial search is performed from depth $cutoff$ until $second\_cutoff$, also storing into P_l all feasible and valid incomplete solutions found at $second\_cutoff$ (*line* 3). Finally, the multi-GPU search takes place (*line* 4), searching for *valid* and *complete* solutions, i.e., from depth $second\_cutoff$ until the problem size (N). The algorithm returns a tuple (*metrics*) containing the explored tree size and the number of valid and complete solutions found by Algorithm 3.

Algorithm 2: Generating the task-local pool.

```
Input: $N$, $node$, $cutoff$, $scnd\_cutoff$
Output: A tuple containing the explored tree size and the number of
        complete and valid solutions found on GPU.
1 $metrics \leftarrow (0, 0)$
2 $P_l \leftarrow \{\}$ $Node$ // Task-local pool
3 $metrics += partial\_search(N, node, cutoff, scnd\_cutoff, P_l)$
4 $metrics += Algorithm\_3(N, P_l, scnd\_cutoff)$
5 return $metrics$
```

The Intra-locale Multi-GPU Search: The GPUs of the locale are exploited as shown in Algorithm 3. Initially, the vectors used by the CUDA kernel to retrieve the metrics from the GPU are defined in lines 1 – 2, and the variable γ receives the number of GPUs the computer node on which the task is running has (*line* 3). In turn, lines 5 to 14 are responsible for launching the searches on GPU. Initially, one task *gpu_id* is created for each GPU. Next, the id of the GPU onto which the task *gpu_id* is going to offload is set though function *cuda_set_gpu*(), which calls the the CUDA function `cudaSetDevice()` though the C-interoperability layer. At line 7, the number of subproblems each GPU processes is calculated. The function `get_load()` simply

Algorithm 3: Exploiting multiple GPUs.

```
Input: $N$, $P$, the second cutoff $depth$
Output: A tuple containing the explored tree size and the number of
        complete and valid solutions found on GPU.
1  $tree\_h \leftarrow [0..|P|-1]$ $int$
2  $sols\_h \leftarrow [0..|P|-1]$ $int$
3  $\gamma \leftarrow cuda\_get\_num\_devices(\,)$
4  $GPU\_load \leftarrow |P|$
5  forall $gpu\_id$ in $0..\gamma-1$ do
6      $cuda\_set\_gpu(gpu\_id)$
7      $device\_load \leftarrow get\_load(gpu\_id, GPU\_load, \gamma)$
8      $stride \leftarrow get\_starting\_point(GPU\_load, gpu\_id, \gamma)$
9      $sols\_ptr \leftarrow sols\_h + stride$
10     $tree\_ptr \leftarrow tree\_h + stride$
11     $pool\_ptr \leftarrow P + stride$
12     $call\_GPU\_search(N, depth, device\_load, pool\_ptr,$
13         $tree\_ptr, sols\_ptr)$
14 end
15 $redTree \leftarrow (+ reduce\ tree\_size\_h)$
16 $redSols \leftarrow (+ reduce\ sols\_h)$
17 $metrics += (redTree, redSol)$
18 return $(redTree, redSols)$
```

verifies whether the division $\frac{GPU\_load}{\gamma}$ is exact. If it is not exact, the GPU with its id gpu_id equals to $\gamma - 1$ receives the remainder of the division. Next, it is required to specify which slices of P, $sols\_h$ and $tree\_h$ each one of the γ kernels are going to work on (*lines* 8–11). The task-local variable $stride$ represents the beginning of the range belonging to each GPU and is calculated as shown in (1).

$$gpu\_id * (\lfloor GPU\_load/\gamma \rfloor) \quad (1)$$

Once the starting positions on the vectors are known, the task-local variables $sols\_ptr$, $tree\_ptr$, and $pool\_pts$ are defined pointing to the beginning of the range the GPU gpu_id is responsible for working on. Then, the function written in C responsible for launching the kernel is called (*line* 12). It is important to point out that $host-to-device$, $device-to-host$, error checking, and other CUDA-specific functions are performed in this function. Finally, a parallel reduction of the metrics is performed (*lines* 15–16), and then, the metrics are returned (*line* 17–18). One can see in Figure 3 an overview of the proposed algorithm.

D. Implementation Aspects

The master-worker behavior is obtained by setting the `coordinated` flag of the distributed iterator to `true`. This way, *locale 0* does not search but coordinates the search process. To avoid losing the computational resources of the computer on which locale 0 resides, one locale more than the number of computer nodes available is launched by using the `-nl` launching option. Thus, the first computer of the reservation hosts two locales: *locale 0* and *locale 1*.

Concerning the intra-node parallelism, the pointers arithmetic of Algorithm 3 are performed by using Chapel's `c_ptrTo()` function. The CUDA-C kernel is based on a serial and hand-optimized backtracking for solving permutation combinatorial problems [17], and it is a non-recursive backtracking that does not use dynamic data structures, such as stacks.

The semantics of a stack is obtained by using a variable *depth* and by trying to increment the value of the vector *board* at position $depth$. If this increment results in a feasible and valid incomplete solution, the $depth$ variable is then incremented, and the search proceeds to the next depth. After trying all configurations for a given depth, the search backtracks to the previous one.

It is important to point out that the `Node` data structure is similar to any permutation-based combinatorial/combinatorial optimization problem. It contains *two* integer vectors of size $cutoff$. The first one, identified by *board*, stores the feasible and valid incomplete solution. The second one keeps track of board lines by setting position n to 1 each time a queen is placed in the n-th line.

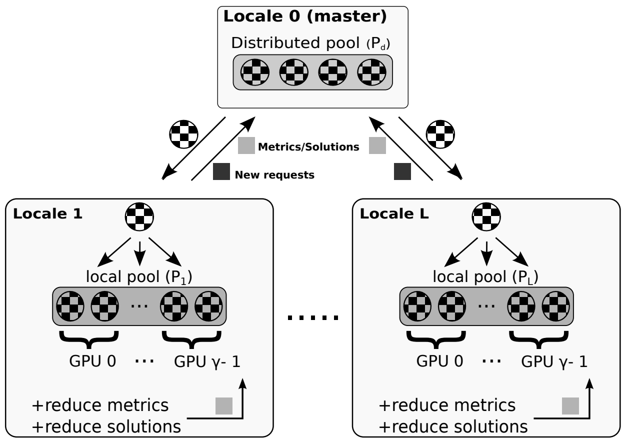


Fig. 3. The locale 0 (master) is responsible for generating the distribute pool P_d and controlling the search. Each worker locale receives nodes from the master and generates a local pool (P_l) that is partitioned into γ subsets. As said in Section III-D, L locales are launched on $L-1$ computer nodes (Own representation adapted from [1]).

IV. Performance Evaluation

A. Experimental Protocol

In our experimental study, the following applications for enumerating all valid and complete configurations of the N-Queens problem are considered.

- **Baseline (CUDA-C)**: multi-GPU implementation optimized for single-locale execution written in CUDA-C. It consists of an initial search on CPU until a cutoff depth, which generates a pool of nodes P. Then, this set is divided into γ subsets, where γ is the number of GPUs the computer node is equipped with. Finally, after the kernel execution, the CPU part of the code reduces the metrics (tree size and number of found solutions).
- **GPUIterator**: distributed version of the baseline implementation written in Chapel. The main difference between the GPUIterator and the baseline is that this variant uses the `GPUIterator` module [9], which has been introduced in Section II. Differently from the algorithm proposed in this paper, the GPUIterator implementation performs no distributed load balancing. In turn, it first divides P_d into L subsets, where L is the number of locales on which the application runs. Then, on each locale, the iterator further divides each set $L^i, i \in \{0, ..., L-1\}$ into γ subsets to be offloaded to the GPUs.
- **ChplGPU**: implementation of the productivity-aware distributed search detailed in the last section, which employs Chapel's distributed iterators for work distribution and load balancing.

It is important to point out that all implementations introduced above employ the same kernel written in CUDA-C [1]. Finally, it is worth mentioning that it is not the objective of this work to compare Chapel *vs.* MPI+X for implementing distributed tree search in terms of productivity, performance, and scalability, as such a study has already been performed [8]. Instead, the objective of this evaluation is to investigate how the Chapel-based applications scale as the number of GPUs increases.

[1]All implementations can be found on the Chapel-based Optimization Project (ChOp) repository: https://github.com/tcarneirop/ChOp

TABLE I
Summary of the environment configuration for multi-locale execution and compilation.

Variable	Value
`CHPL_RT_NUM_THREADS_PER_LOCALE`	48
`CHPL_TARGET_ARCH`	*native*
`CHPL_COMM`	*gasnet*
`CHPL_GASNET_SEGMENT`	*everything*
`CHPL_COMM_SUBSTRATE`	*ofi*
`GASNET_PSM_SPAWNER`	*ssh*

B. Parameters Settings

N-Queens problems of size (N) ranging from 17 to 21 are considered. The experiments take from a few seconds ($N=17$) to *ten hours* of parallel processing on *two* GPUs ($N=21$). The number of computer nodes considered in the experiments ranges from 1 to 12. Computer nodes operate under Debian 4.9.0, 64 bits. They are equipped with *two* Intel Xeon E5-2650 v4 @ 2.00 GHz (a total of 24 cores/48 threads per node) and 128 *GB* RAM. Each computer node is equipped with *two* NVIDIA GeForce GTX 1080 Ti – Pascal generation (11 GB RAM and 3584 CUDA cores @ 1582 Mhz). Thus, the maximum number of GPUs used in the experiments is 24 ($86,016$ CUDA cores). The computer nodes are interconnected through a 100 *Gbps* Intel Omni-Path network.

The ChplGPU and GPUIterator implementations are programmed with Chapel 1.22, and the *default* task layer (qthreads) is the one employed. The GPU kernel is programmed in CUDA and compiled with NVCC 10.1, whereas the function that calls the kernel is programmed in C and compiled with *gcc* 8.3. Both implementations written in Chapel are executed on top of GASNet, and appropriate environment variables should be set to fully utilize the capability of the target platform. Table I shows a summary of the runtime

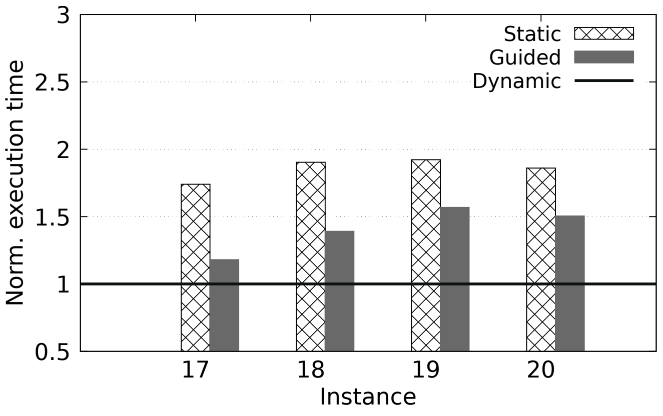


Fig. 4. Performance comparison of the *ChplGPU* implementation using two different distributed iterator and static distribution of P_d. Results are shown for 24 GPUs (12 computer nodes).

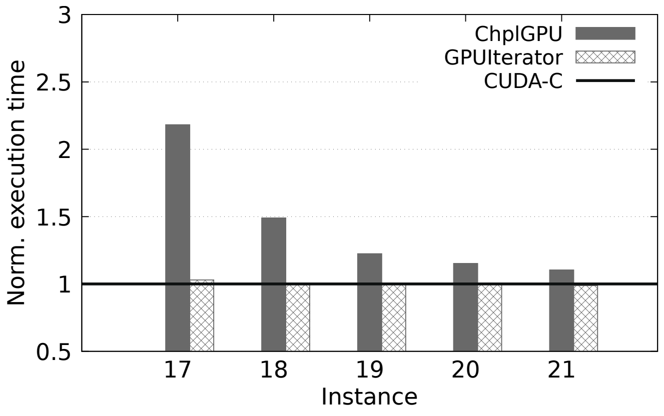


Fig. 5. Normalized execution time of the optimized CUDA-C implementation compared to its Chapel-based counterparts. Results are for one *one* computer node – *two* GPUs.

configurations for multi-locale execution. The OpenFabrics GASNet implementation is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with SSH, which is responsible for getting the executables running on different locales (`GASNET_PSM_SPAWNER`).

Chapel provides two different distributed load balancing iterators: *guided* and *dynamic*, which are also similar to OpenMP's schedules of the same name. Experiments were also carried out to identify the best *chunk* for both load balancing strategies. As depicted in Figure 4, using the dynamic distributed iterator results in the best overall performance for ChplGPU. Therefore, it is the distributed iterator to be considered hereafter in the results presented for the ChplGPU implementation.

Experiments were also carried out to choose suitable cutoff depths for all implementations. This parameter directly influences the size of the pool of nodes, the granularity of the nodes, the GPU occupancy and the capacity of load balancing of the iterators [8], [17], [18]. One can see in Table II the best parameters experimentally found for all implementations introduced in Section IV-A

TABLE II
BEST PARAMETERS EXPERIMENTALLY FOUND FOR THE CHPLGPU, GPUITERATOR AND THE OPTIMIZED CUDA-C IMPLEMENTATION.

Parameter	ChplGPU	GPUIterator	CUDA-C
$cutoff$	2	7	7
$second\_cutoff$	8	–	–
Distributed iterator	*dynamic*	–	–
Distributed chunk	1	–	–

C. Performance Results

1) Single-locale Execution: Figure 5 shows the single-locale performance of both implementations written in Chapel compared to the CUDA-C baseline. Firstly, the GPUIterator implementation is equivalent to the baseline one in terms of performance. While it is 3% slower than the baseline when $N = 17$, the performance difference is not significant as N increases. This indicates that the use of the `GPUIterator` module does not incur significant overhead. However, this is not the case with ChplGPU. For example, ChplGPU is more than twice as slow as the baseline when $N = 17$.

The CUDA-C implementation is straightforward and optimized for single-locale execution. The initial search on CPU creates the active set and divides it into γ sets, one for each GPU, and *only* γ kernel calls are performed. The same is true for GPUIterator. In turn, ChplGPU has several sources of overhead for a single-locale execution. It consists of *two* partial searches on CPU before starting the search on GPU. The iterator distributes nodes to the worker locales, which perform another partial search on CPU for generating a task-local pool of nodes (P_t). In turn, local active sets are much smaller than the single one generated by CUDA-C, which results in less efficient use of the GPUs for the smallest sizes [18].

It is also worth to point out that, unlike the CUDA-C baseline and the GPUIterator implementation that only launch γ kernels, ChplGPU launches $|P_d| * \gamma$ kernels. For instance,

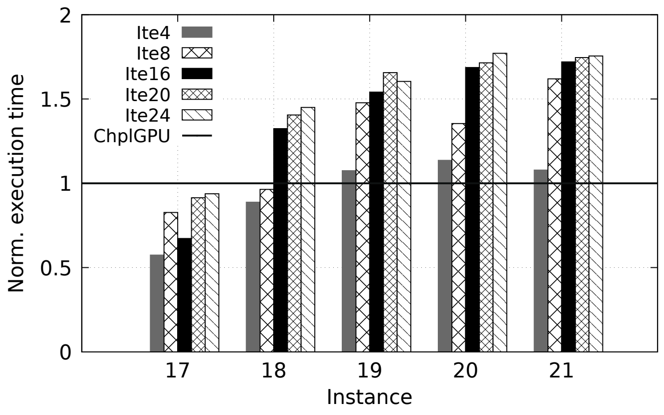
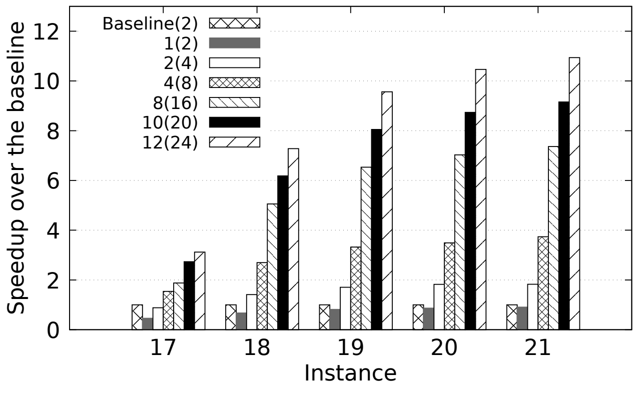
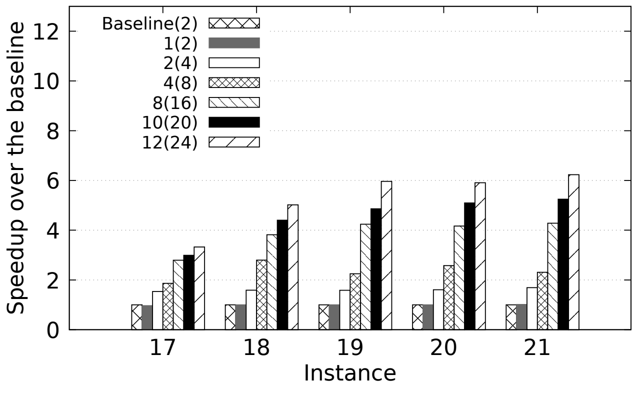


Fig. 6. Normalized execution time of GPUIterator and ChplGPU. Results are shown for 4 GPUs (2 computer nodes) to 24 GPUs (12 computer nodes).



(a) ChplGPU *vs.* Baseline (CUDA-C)



(b) GPUiterator *vs.* Baseline (CUDA-C)

Fig. 7. Speedup achieved by (a) *ChplGPU* and (b) *GPUIterator* implementations compared to the optimized CUDA-C baseline executed on *one* computer node (*two* GPUs). Results are shown for 1 computer node (2 GPUs) to 12 computer nodes (24 GPUs). In the graph, the keys present the number of computer nodes followed by the number of GPUs in parenthesis – e.g., 12(24) means that the results are shown for 12 computer nodes and a total of 24 GPUs.

ChplGPU launches $240 * 2$ kernels for $N = 17$. Additionally, each kernel launch requires *host-to-device*, *device-to-host* transfers, and metrics reductions on CPU, which is also a source of overhead. As the problem size grows, the overhead becomes less significant compared to the solution space to be evaluated. Therefore, ChplGPU, which is $2.15\times$ slower than the baseline for $N = 17$, is only 15% and 10% slower for sizes $N = 20$ and 21, respectively.

2) Multi-locale Execution: Figure 6 presents the normalized execution times of ChplGPU and GPUIterator solving the N-Queens problems using from 4 to 24 GPUs. Firstly, consider the results for the smallest instance ($N = 17$). As solving this instance takes only a few seconds, GPUIterator is faster than ChplGPU for 4 to 20 GPUs. As more computer nodes are added, GPUIterator performs poorly due to the lack of distributed load balancing. In turn, the performance difference between ChplGPU and GPUIterator grows as more computer nodes are added. For $N = 18$ and 16 – 24 GPUs, ChplGPU is from $1.32\times$ – $1.45\times$ faster than its counterpart. As the problem size grows, GPUIterator becomes slower than ChplGPU even on 2 computer nodes – 4 GPUs. For $N = 20$, ChplGPU is from $1.13\times$ (4 GPUs) to $1.77\times$ (24 GPUs) faster than the GPUIterator implementation.

Figure 7 shows the strong scaling of ChplGPU and GPUIterator on 2 GPUs (1 computer node) to 24 GPUs (12 computer nodes) over the CUDA-C baseline on 2 GPUs (1 computer node). Note that for each instance, the performance numbers are normalized to *Baseline(2)*. In general, as the number of GPUs increases, the performance of both ChplGPU and GPUIterator improves except for the ChplGPU on 2 GPUs as with Figure 5. Also, ChplGPU achieves further performance improvements over GPUIterator as N, and the number of GPUs increases, due to the dynamic scheduling feature, which is not supported in GPUIterator.

To quantify these performance trends, we compare the performance numbers with the linear speedup – i.e., the CUDA-C baseline on $nGPUs$ gives $n\times$ speedup. Initially, consider the results for ChplGPU (Figure 7a). When $N \geq 19$, the speedups achieved for all $< problem, \#GPUS >$ configurations are at least 80% of the linear speedup. For $N \geq 20$, results are even better, ranging from 87% to 91% of the linear speedup. In turn, due to its static load distribution scheme, the `GPUIterator`-based implementation reaches around 80% of the linear speedup for all sizes when only 4 GPUs (2 computer nodes) are used (refer to Figure 7b). For $N \geq 18$, the speedups achieved by the GPUIterator implementation are around 50% of the linear speedup.

V. Discussion

Initially, if only the single-locale programming is taken into account, the `GPUIterator` module is indeed an option in terms of *time to a first implementation*. Error-prone details such as division of the pool among locales and GPUs, pointers arithmetic, and dealing with the CUDA API are hidden to the programmer. As a consequence, the size of the single-locale version of the GPUIterator implementation has almost the same number of source lines of code (SLOCS) of the CUDA-C baseline implementation. Moreover, it also enables one to conceive a GPU-based distributed implementation by adding a few lines of code. For instance, an addition of 8 SLOCS is required to make GPUIterator a distributed application. However, the lack of load balancing of the `GPUIterator` module results in poor scalability, which justifies the more complex implementation of ChplGPU.

In terms of SLOCS, ChplGPU is $1.5\times$ longer than the GPUIterator implementation. The partial searches necessary to exploit the distributed iterators, along with the C-interoperability code related to the use of GPUs, amount for more than 50% of the SLOCS of ChplGPU. In the scope of this work, the higher programming effort of coding two levels of partial search and the functions responsible for offloading and getting data from the GPU pays off, as ChplGPU achieves parallel efficiency up to $2\times$ higher and it is up to $1.77\times$ faster than its Chapel-written counterpart. In this sense, a further necessary improvement to the `GPUIterator` module is implementing distributed GPU-oriented load balancing iterators. This effort would make it possible to program a distributed tree search that scales but having only a fraction of the size of ChplGPU.

Finally, it is important to mention that some of the challenges concerning employing GPUs for irregular tree search remain when using a high-productivity language. On the one hand, we can exploit high-level features provided by Chapel for several aspects of the search, such as work distribution and termination criteria. On the other hand, mixing programming models also brings GPU's challenges along with it. For instance, it is required to tune the chunk size of the distributed iterator taking into account that a subproblem yielded by the iterator must provide load enough for multiple GPUs. The performance of ChplGPU can be drastically decreased due to a bad parameter choice. This is also true for GPUIterator. Despite its straightforward implementation, it is still required to tune the depth parameters to generate a distributed pool that, after divided into $L * \gamma$ subsets also can provide load enough to the GPUs, so they can be used efficiently.

VI. Conclusions and Future Work

In this work, we revisited the design and implementation of distributed tree search algorithms dealing with multiple GPU accelerators, in addition to scalability and productivity-awareness, using Chapel. A distributed backtracking for enumerating all valid solutions of the N-Queens problem was conceived. The proposed algorithm exploits Chapel's distributed iterators combined with pre-compiled CUDA kernels through Chapel's C-interoperability layer.

According to the performance results, the implementation of the proposed algorithm is slower than a hand-optimized CUDA-C implementation, when taking into account single-locale execution. However, in distributed scenarios, the proposed algorithm scales, achieving more than 90% of the linear speedup in the biggest test-cases.

Permutation combinatorial optimization problems are commonly solved to optimality by using branch-and-bound search algorithms [1]. Therefore, the first future research direction is to extend the proposed multi-locale backtracking into a distributed B&B. This way, it will be possible to solve challenging optimization problems, such as the Quadratic Assignment and the Flow-shop Scheduling Problem.

Going one step further towards exascale, the scalability should be increased considering more GPU-powered processing nodes. Also, heterogeneity can be improved to consider other accelerators such as FPGAs thanks to Chapel's interoperability capabilities. Finally, checkpointing-based fault tolerance should be addressed in addition to scalability, productivity-awareness, and CPU-GPU heterogeneity issues to achieve a holistic exascale-aware design and implementation.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [19], hosted by INRIA and including several other organizations. We thank Bradford Chamberlain, Elliot Ronaghan, Vassily Litvinov, Michael Ferguson, and Lydia Duncan (Hewlett Packard Enterprise – HPE) for their support on Chapel's Gitter platform [2].

References

[1] T. Crainic, B. Le Cun, and C. Roucairol, "Parallel branch-and-bound algorithms," *Parallel combinatorial optimization*, pp. 1–28, 2006.

[2] M. Mezmaz, N. Melab, and E.-G. Talbi, "A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems," in *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.* IEEE, 2007, pp. 1–9.

[3] 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*, 2020.

[4] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, and R. V. Van Nieuwpoort, "The landscape of exascale research: A data-driven literature analysis," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–43, 2020.

[5] S. Fiore, M. Bakhouya, and W. W. Smari, "On the road to exascale: Advances in high performance computing and simulations—an overview and editorial," *Future Generation Computer Systems*, vol. 82, pp. 450 – 458, 2018.

[6] E. Lusk and K. Yelick, "Languages for high-productivity computing: the darpa hpcs language project," *Parallel Processing Letters*, vol. 17, no. 01, pp. 89–102, 2007.

[7] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu *et al.*, "Chapel comes of age: Making scalable programming productive," in *Cray User Group*, 2018.

[8] T. Carneiro, J. Gmys, N. Melab, and D. Tuyttens, "Towards ultra-scale branch-and-bound using a high-productivity language," *Future Generation Computer Systems*, vol. 105, pp. 196 – 209, 2020.

[9] A. Hayashi, S. R. Paul, and V. Sarkar, "GPUIterator: Bridging the gap between Chapel and GPU platforms," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, ser. CHIUW 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 2–11.

[10] A. Y. Grama and V. Kumar, "A survey of parallel search algorithms for discrete optimization problems," *ORSA Journal on Computing*, vol. 7, 1993.

[11] G. Almasi, "PGAS (partitioned global address space) languages," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1539–1545.

[12] A. S. et al., "Performance portability with the chapel language," ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 582–594. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.60

[13] M. L. C. et al., "GPGPU support in chapel with the radeon open compute platform (extended abstract)," ser. CHIUW'17, 2017.

[14] W. Zhang, "Branch-and-bound search algorithms and their computational complexity," DTIC Document, Tech. Rep., 1996.

[15] J. Bell and B. Stevens, "A survey of known results and research areas for n-queens," *Discrete Mathematics*, vol. 309, no. 1, pp. 1–31, 2009.

[16] 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.

[17] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Junior, N. Melab, and D. Tuyttens, "GPU-accelerated backtracking using CUDA dynamic parallelism," *Concurrency and Computation: Practice and Experience*, pp. e4374–n/a, 2017.

[18] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the GPU," in *European Conference on Parallel Processing*. Springer, 2011, pp. 425–437.

[19] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[2]https://gitter.im/chapel-lang/chapel