Towards Ultra-scale Exact Optimization Using Chapel

Tiago Carneiro[†], Nouredine Melab[*]
University of Luxembourg[†], Luxembourg, INRIA Lille - Nord Europe[†][*], France
Université de Lille, CNRS/CRIStAL[*], France
tiago.carneiropessoa@uni.lu, nouredine.melab@univ-lille.fr

***Abstract*—Tree-based search algorithms applied to combinatorial optimization problems are highly irregular and time-consuming when it comes to solving big instances. Due to their parallel nature, algorithms of this class have been revisited for different architectures over the years, and these parallelization efforts have always been guided by the performance objective setting aside productivity. However, dealing with scalability implicitly induces the heterogeneity issue, which means that different programming models/languages, runtimes and libraries need to be employed together for efficiently exploiting all levels of parallelism of large-scale systems. As a consequence, efforts towards productivity are crucial for harnessing the future generation of supercomputers. In this talk proposal, we present our efforts towards productivity-aware ultra-scale tree search using the Chapel language. Four topics are covered in this document: the design and implementation of tree-search using Chapel, improving intra-node efficiency, the use of GPUs and future perspectives.**

***Index Terms*—Branch-and-bound, Backtracking, PGAS, Chapel**

I. Introduction

Combinatorial optimization problems (COPs) are present in different areas of knowledge, such as operations research, bioinformatics, artificial intelligence, and machine learning [1]. Algorithms for solving COPs can be divided into exact (complete) or approximate methods [2]. The exact ones guarantee to return a proven optimal solution for any instance of the problem in a finite amount of time. Among the complete algorithms, the tree-based enumerative strategies, such as backtracking and branch-and-bound (B&B), are the most widely used methods for solving instances of COPs to optimality.

As the decision version of COPs is usually NP-Complete, the size of problems that can be solved to optimality is limited, even if large-scale distributed computing is employed [3], [4]. It is expected that the use of exascale computers will result in a significant decrease in the execution time required to solve COP instances to optimality. However, dealing with scalability implicitly induces the heterogeneity issue [5], which means that different programming models/languages, runtimes and libraries need to be employed together for efficiently exploiting all levels of parallelism of the system [6]. As a consequence, efforts towards productivity are crucial for harnessing the future generation of supercomputers [7], [8].

The study of a feasible high-productivity language for the design and implementation of tree-based search led us to the Chapel parallel programming language [9]. In the context of this work, Chapel stands out as it is a compiled language that allows us to hand-optimize the data structures for performance and also provides high-level features for dealing with the irregularity of the solution space, such as distributed iterators. In this work, we deal with the challenge of redesigning a tree-based search from a performance-oriented perspective to a productivity-aware one by finding a trade-off between both approaches.

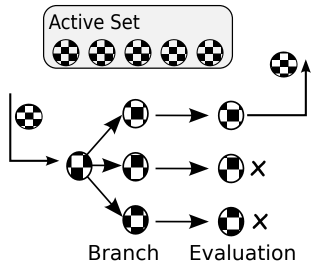

Fig. 1. Visual representation of a tree-based search algorithm (Own representation based on [3]).

In what follows, we present our efforts towards productivity-aware ultra-scale tree search using the Chapel language. Four topics are covered in this talk proposal: the design and implementation of tree-search algorithms using Chapel, improving intra-node efficiency, the use of GPUs and future perspectives.

II. Background

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 (subproblems) more constrained than their parent 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

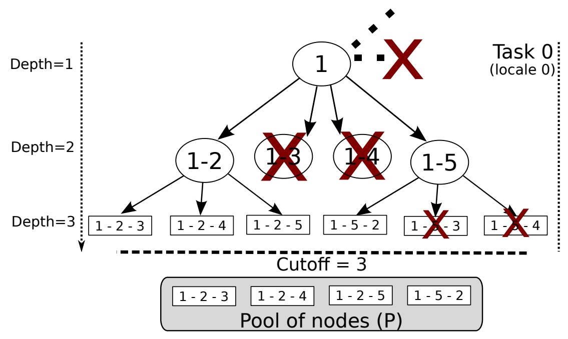

Fig. 2. Schematic representation of an initial search on *locale 0 - task 0* that generates the pool P for a problem size $N = 4$ and $cutoff = 3$. The figure depicts the branch that has the element 1 of the permutation as the root and generated 4 valid and feasible incomplete solutions at depth $cutoff = 3$.

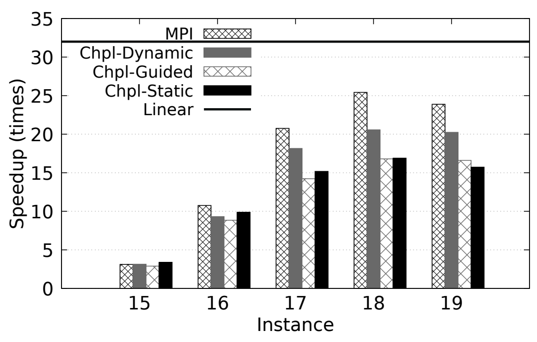

Fig. 3. Speedup achieved by the Chapel-base implementation compared to its MPI-C counterpart on 2 (25 cores) to 32 computer nodes (324 cores) compared to the execution on *one* computer node. Results are shown for all distributed iterators. The N-Queens sizes range from 15 to 19. Values are given in percent of the linear speedup.

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.

III. A Productivity-aware Distributed Tree-based Search

In this section, we present a productivity-aware distributed tree-search algorithm for solving permutation combinatorial problems. All parallel/distributed tree-search algorithms presented in this talk proposal follow the master-worker scheme described as follows.

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 2, the initial pool of nodes P is generated though a partial search (*line* 6), called *initial search*. This work focuses on permutation combinatorial problems, for which an N-sized permutation represents a valid and complete solution. Therefore, task 0 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.

Algorithm 1: The Master-worker scheme.

```
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  $metrics\ += initial\_search(N, cutoff, P)$
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     $metrics\ += Search(N, node, cutoff,$
13         $second\_cutoff)$
14 end
15 $present\_results(metrics)$
```

The parallel 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). There is no need for programming a termination criterion 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`). Finally, *both* intra- and inter-locale levels of parallelism are exploited by using the distributed iterators.

Implementation: We conceived two backtracking algorithms for enumerating *all* complete and feasible solutions of the N-Queens: a single-locale and a distributed one. It is worth mentioning that we use the N-Queens problem as a proof-of-concept that motivates further improvements in solving related combinatorial optimization problems. Refer to [11], [12] for more about both implementations.

Evaluation: The experimental results show that Chapel is a suitable language for the design and implementation of parallel and distributed tree search algorithms. The single-threaded search in Chapel is on average only 7% slower than its counterpart written in C. Whereas programming a serial and multicore tree search in Chapel is equivalent to C-OpenMP in terms of performance and SLOC, its productivity-aware features for distributed programming stand out.

Thanks to Chapel's global view of the control flow and data structures, the main difference between the multi- and single-locale versions lies mainly in the use of the PGAS data structures and distributed iterators for load balancing. There is no need for explicitly dealing with communication, metrics reduction, distributed load balancing and intra-locale parallelism. As a consequence, the multi-locale version is only 8 *lines longer* than its single-locale counterpart, which results in a code 33% bigger. In contrast, it is required to add 24 lines to the backtracking written in C-OpenMP to use MPI, which almost doubles the program size. Despite the high level of its features, the distributed tree search in Chapel is on average 16% slower and reaches up to 80% of the efficiency of its C-MPI+OpenMP counterpart.

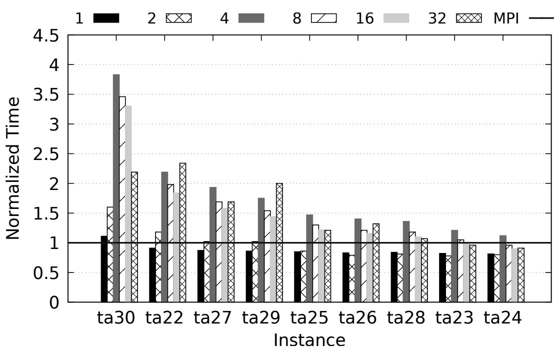

Fig. 4. Execution times of the Chapel-based B&B compared to a MPI-based state-of-the-art implementation (MPI-PBB) [13] solving flow-shop scheduling instances ($ta21$-$ta30$) [14] to optimality. Results are shown for 1 computer node (32 cores) to 32 computer nodes (1024 cores).

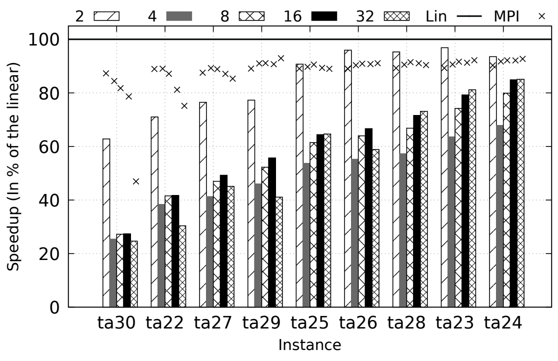

Fig. 5. Speedup achieved by Chapel-BB and MPI-PBB on 2 (64 cores) to 32 computer nodes (1024 cores) compared to the execution on *one* locale. Values are given in percent of the linear speedup ($Lin - 100\%$).

IV. Improving Intra-locale Efficiency

In this section, we detail the extension of the distributed backtracking into a branch-and-bound (B&B) search for solving permutation *combinatorial optimization* problems. This way, we added to the backtracking bounding operations for subproblem evaluation and also means to keep the coherency of the best solution found so far.

In Algorithm 1, a task receives a chunk of subproblems from the master, and then the search implicitly enumerates the feasible region rooted by a given subproblem from depth $cutoff$ until the depth N. However, solving instances of combinatorial optimization problems to optimality is much more challenging, as the loads produced are much more irregular than the ones produced by the N-Queens. The number of feasible and valid incomplete solutions at depth $cutoff$ might be insufficient to efficiently use all CPU cores of several locales at once. To cope with this situation, we cannot rely only on the distributed iterator for exploiting intra-node parallelism.

For each subproblem in a chunk, the worker task executes once more the partial search described in Algorithm 1 for generating a task-local pool (P_l). This search is performed from depth $cutoff$ until $second\_cutoff$, also storing into P_l all feasible and valid incomplete solutions found at $second\_cutoff$. Next, an intra-locale iterator is used to manage the pool. Then the metrics are reduced and returned to the master locale. Refer to [15] for more details about this B&B algorithm.

Evaluation: We compare the improved implementation to a master-worker state-of-the-art MPI-C++ B&B [13] in terms of parallel performance and efficiency. The benchmark instances used in our experiments are the flow-shop scheduling problem (FSP) instances defined by Taillard [14] where $M = N = 20$.

As one can see in Figure 4, Chapel-BB is faster or at least equivalent to MPI-PBB on 32 locales (1024 cores) for the three biggest instances ($ta23$, $ta24$ and $ta28$). In turn, as the number of computer nodes increases, load balancing becomes crucial, and Chapel's distributed iterators cannot deliver to the 4 smallest instances ($ta22$, $ta27$, $ta29$ and $ta30$) regular loads among locales, resulting in poor parallel performance and scalability (Figure 5).

According to the productivity-oriented results (see [15] for more details), the overall software cost of MPI-PBB in terms of SLOC is $5.6\times$ higher than the one of Chapel-BB. The most expensive parts of the MPI-PBB code are the load balancing and termination criteria, which are $35\times$ and $18\times$ more costly than the Chapel-BB ones, respectively. It is important to point out that this programming effort pays off, as the state-of-the-art load balancing mechanism of MPI-PBB provides 90% of the linear speedups for the majority of the instances.

V. Using Multiple GPUs

In combinatorial optimization, the use of GPUs became crucial because it enables solving to the optimality instances having prohibitive execution times on CPUs [13]. 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 [16].

Although Chapel is a parallel programming language, it does not support the use of GPUs. The `GPUIterator` module [17] fills this gap, as it allows GPUs and concurrent CPU/GPU execution in Chapel. However, its lack of

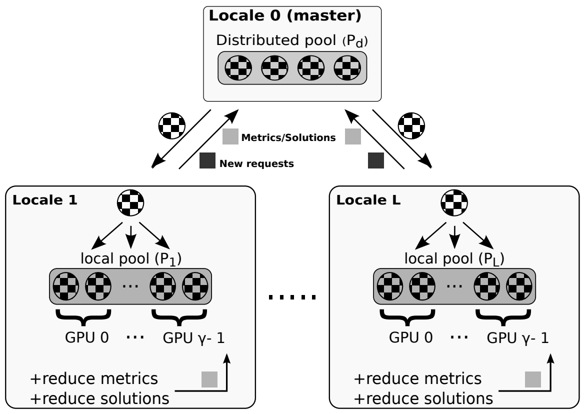

Fig. 6. The locale 0 (master) is responsible for generating the distributed pool P_d and controlling the search. Each worker locale receives nodes from the master and generates a local pool (P_l) that is partitioned into γ subsets. L locales are launched on $L - 1$ computer nodes (Own representation adapted from [3]).

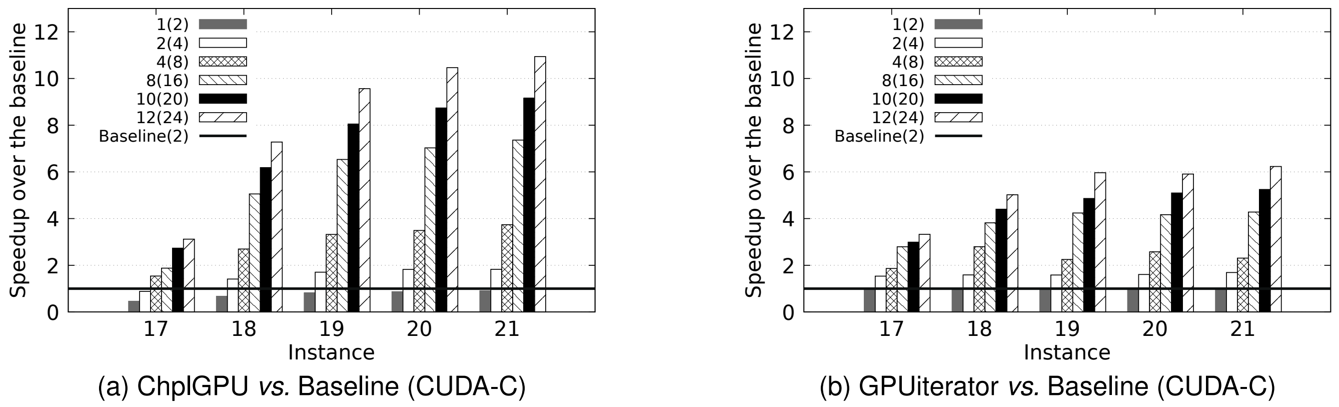

Fig. 7. Speedup achieved by (a) *ChplGPU* and (b) *GPUIterator* implementations compared to the optimized CUDA-C baseline executed on *one* computer node (*two* GPUs). Results are shown for 1 computer node (2 GPUs) to 12 computer nodes (24 GPUs). In the graph, the keys present the number of computer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

distributed load balancing makes it unfeasible for this work. To cope with this challenge, we adapted the procedure for exploiting intra-node parallelism, introduced in the previous section, for taking advantage of GPUs through the Chapel C-interoperability layer.

As in the B&B algorithm detailed in the last section, the second partial search also generates a task-local pool P_l for each subproblem in the chunk. Then, the $|P_l|$ nodes are divided among the γ GPUs of the system. Next, for each GPU, a CPU task is created for managing data and launching the kernel. All GPU-related code is accessed through the C-interoperability layer. One can see in Figure 6 an overview of the distributed GPU-based master-worker search. For more details concerning the GPU-based algorithm see [18].

Implementation: We conceived three backtracking algorithms for enumerating *all* complete and feasible solutions of the N-Queens: a CUDA-C baseline, a GPU-based version of the search introduced in the last section (ChplGPU) and a distributed version of the CUDA-C application using the `GPUIterator` module for distributed execution. As for the implementation detailed in Section III, the N-Queens is used as a proof-of-concept aiming at further improvements in solving related combinatorial optimization problems. Refer to [19] for more details about the implementations.

Evaluation: Figure 7 shows the strong scaling of both ChplGPU and the GPUIterator-based implementations. Due to its static load distribution scheme, the `GPUIterator`-based implementation reaches around 80% of the linear speedup only when 4 GPUs (2 computer nodes) are used (refer to Figure 7b). In the scope of this work, the higher programming effort of combining two levels of partial searches with the distributed iterators pays off, as ChplGPU achieves parallel efficiency up to $2\times$ higher and it is up to $1.77\times$ faster than its `GPUIterator-based` counterpart for $N \geq 19$.

VI. Conclusions and Future Perspectives

The most difficult challenge of revisiting a tree-based search algorithm from a performance-oriented approach to a productivity one is achieving a trade-off between both approaches. In this talk, we show that such a trade-off is possible by using Chapel's features for distributed programming. As Chapel is a compiled language, it is possible to hand-optimize for performance the data structures used in the enumeration process. Moreover, the distributed iterators are a key feature for achieving high productivity in the context of this project, preventing us from implementing the master-worker model for load balancing and metrics reduction.

Concerning the use of accelerators, it is important to mention that some of the challenges concerning employing GPUs for irregular tree search remain when using Chapel. 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-related 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.

Despite the obtained promising results, there are challenges not yet addressed *in the context of productivity-awareness*. The most important are: **1)** improving scalability **2)** the heterogeneity issue and **3)** fault tolerance.

1) **Scalability:** we show that the OpenMP-like work distribution provided by Chapel is not effective in the most challenging scenarios. In this case, the objective is to improve Chapel's iterators by including locality-aware work-stealing mechanisms for load balancing taking into account both intra- and inter-node levels of parallelism.
2) **Heterogeneity:** a challenge that remains open concerning heterogeneity is how to harness all CPUs and GPUs of the system, but also keeping productivity. Thus, we plan to incorporate work-stealing features into the `GPUIterator` module.
3) **Fault tolerance**: in exact optimization, the execution time of an algorithm cannot be accurately predicted. Thus, it is necessary to provide means for the program to recover from a previous execution state after a failure. Initially, the fault tolerance aspects of the algorithms are going to be focused on checkpointing.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [20], 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. We also would like to thank Akihiro Hayashi and Prof. Vivek Sarkar (Georgia Tech.) for helping us to implement the `GPUIterator`-based tree search.

References

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

[2] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

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

[4] 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.

[5] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," *IEEE Micro*, vol. 35, no. 4, pp. 26–36, 2015.

[6] W. Gropp and M. Snir, "Programming for exascale computers," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 27–35, 2013.

[7] 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.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of . . . , Tech. Rep., 2006.

[9] J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, and D. Tuyttens, "A comparative study of high-productivity high-performance programming languages for parallel metaheuristics," *Swarm and Evolutionary Computation*, vol. 57, p. 100720, 2020.

[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] T. Carneiro and N. Melab, "An incremental parallel PGAS-based tree search algorithm," in *The 2019 International Conference on High Performance Computing & Simulation (HPCS 2019)*, 2019.

[12] ——, "Productivity-aware design and implementation of distributed tree-based search algorithms," in *International Conference on Computational Science*. Springer, 2019, pp. 253–266.

[13] 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.

[14] E. Taillard, "Benchmarks for basic scheduling problems," *European journal of operational research*, vol. 64, no. 2, pp. 278–285, 1993.

[15] 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.

[16] 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.

[17] 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.

[18] T. Carneiro, N. Melab, A. Hayashi, and V. Sarkar, "Towards chapel-based exascale tree search algorithms: dealing with multiple gpu accelerators," in *The 2020 International Conference on High Performance Computing & Simulation (HPCS 2020)*, 2021.

[19] T. Carneiro and N. Melab, "Chapel-based optimization," https://github.com/tcarneirop/ChOp, 2021.

[20] 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.