Productivity-aware Design and Implementation of Distributed Tree-based Search Algorithms

Tiago Carneiro, Nouredine Melab

Productivity-aware Design and Implementation of Distributed Tree-based Search Algorithms

Tiago Carneiro[1] and Nouredine Melab[1,2]

[1] INRIA Lille - Nord Europe, Lille, France
[2] Université de Lille, CNRS/CRIStAL, Lille, France
tiago.carneiro-pessoa@inria.fr, nouredine.melab@univ-lille.fr

Abstract. Parallel tree search algorithms offer viable solutions to problems in different areas, such as operations research, machine learning and artificial intelligence. This class of algorithms is highly compute-intensive, irregular and usually relies on context-specific data structures and hand-made code optimizations. Therefore, C and C++ are the languages often employed, due to their low-level features and performance. In this work, we investigate the use of Chapel high-productivity language for the design and implementation of distributed tree search algorithms for solving combinatorial problems. The experimental results show that Chapel is a suitable language for this purpose, both in terms of performance and productivity. Despite the use of high-level features, the distributed tree search in Chapel is on average 16% slower and reaches up to 85% of the scalability observed for its MPI+OpenMP counterpart.

Keywords: Tree search algorithms · High productivity · PGAS · Chapel · MPI+OpenMP.

1 Introduction

Tree-based search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree. This class of algorithms is often used for the exact resolution of permutation combinatorial optimization problems (COP) and offers viable solutions to problems in different areas, such as operations research, artificial intelligence, bioinformatics and machine learning [15, 19]. As the decision version of permutation COPs are usually NP-Complete, the size of problems that can be solved to optimality is limited, even if large-scale distributed computing is used [9, 16]. In this sense, it is expected that exascale computers are willing to allow a significant decrease in the execution time required to solve COP instances to optimality. However, such large scale systems are going to be complex to program, and efforts towards programmability are crucial for better exploiting this future generation of computers [2, 13].

Tree-based search algorithms are compute-intensive and highly irregular, which demands hand-optimized data structures for efficient search and load balancing [6, 14, 17]. Thus, high-productivity languages are not often employed

within the scope of tree search, as they historically suffer from severe performance penalties [11]. Instead, this kind of application is frequently written in either C or C++, due to low-level features present in both languages [5, 9].

Chapel is a productivity-aware programming language for high-performance computing that is competitive to both C-OpenMP and C-MPI+OpenMP in terms of performance, considering different benchmarks [8]. The objective of the present research is to investigate Chapel's features to design and implement distributed tree search algorithms for solving permutation combinatorial problems. To the best of our knowledge, the present research is the first one that investigates the use of a high-productivity language for this purpose.

The experimental results show that Chapel is a suitable language for the design and implementation of distributed tree search algorithms, both in terms of performance and productivity. It is possible to conceive a distributed tree search algorithm starting from its multicore counterpart by adding few modifications. The distributed algorithm performs load balancing among different computer nodes and also uses all CPU cores that a computer node has. Despite the high level of its features, the distributed tree search in Chapel is on average 16% slower and reaches up to 85% of the scalability achieved by its C-MPI+OpenMP counterpart. Finally, the distributed load balancing strategies provided are effective: the dynamic load balancing version is up to 1.5× faster than its static counterpart.

The remainder of this paper is structured as follows. Section 2 brings background information and related works. The distributed tree-based search in Chapel is detailed in Section 3. Section 4 presents a performance evaluation. Then, Section 5 brings a discussion in terms of performance, programmability, and limitations of Chapel for programming distributed tree search algorithms. Finally, conclusions are outlined in Section 6.

2 Background and Related Works

2.1 The Chapel Programming Language

Chapel is an open-source parallel programming language designed to improve the programmability for high-performance computing. It incorporates features from compiled languages such as C, C++, and Fortran, as well as high-level elements related to Python and Matlab. The parallelism is expressed in terms of lightweight tasks, which can run on several locales or a single one. In this work, the term *locale* refers to a symmetric multiprocessing computer in a parallel system [10].

In Chapel, both *global view of control flow* and *global view of data structures* are present [8]. Concerning the first one, the program is started with a single task and parallelism is added through data or task parallel features. Moreover, a task can refer to any variable lexically visible, whether this variable is placed in the same locale on which task is running, or in the memory of another one. Concerning the second one, indexes of data structures are globally expressed,

even in case the implementation of such data structures distributes them across several locales. Thus, Chapel is a language that realizes the Partitioned Global Address Space (PGAS) programming model [1].

Finally, indexes of data structures are mapped to different locales using *distributions*. Contrasting to other PGAS-based languages, such as UPC and Fortran, Chapel also supports user-defined distributions [7].

2.2 Tree-based Search Algorithms

Tree-based search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree [15]. The internal nodes of the tree are incomplete solutions, whereas the leaves are solutions. Algorithms that belong to this class start with an initial node, which represents the root of the tree, i.e., the initial state of the problem to be solved. Nodes are branched during the search process, which generates children nodes more restricted than their parent node. Generated nodes are evaluated, and then, the valid and feasible ones are stored in a data structure called *Active Set*.

At each iteration, a node is removed from the active set according to the employed search strategy [19]. The search generates and evaluates nodes until the data structure is empty or another termination criterion is reached. 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, keeping the algorithm from unnecessary computation. The degree of parallelism of tree-based search algorithms is potentially very high, as the solution space can be partitioned into a large number of disjoint portions, which can be explored in parallel.

As these algorithms are compute-intensive, diverse strategies have been used for improving performance, such as instruction-level parallelism, architecture-specific code optimizations and problem-specific data structures [6, 12, 14, 17]. Thus, parallel tree-based search algorithms are frequently written in C/C++, due to their low-level features and supported parallel computing libraries [5]. In the context of distributed algorithms, the same performance-aware strategies are combined with distributed programming libraries for implementing load balancing and explicit communication between processing nodes [9, 16, 18]. As a consequence, programming distributed tree search algorithms can be challenging and time-consuming.

3 Distributed Tree-based Search Algorithms in Chapel

A major objective of Chapel concerning productivity is allowing distributed programming using concepts close to the ones of shared-memory programming [8]. In this section, a multicore single-locale tree-search algorithm is initially proposed. Then, it is extended using Chapel's productivity-aware features for distributed programming.

3.1 Algorithm Overview

This work focuses on permutation combinatorial problems, for which an N-sized permutation represents a valid and complete solution. Permutation combinatorial problems are used to model diverse real-world situations, and their decision versions are often NP-Complete [16, 19].

This section presents two backtracking algorithms for enumerating *all* complete and feasible solutions of the N-Queens problem. Backtracking is a fundamental problem-solving paradigm that consists in dynamically enumerating 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 [19].

The N-Queens problem consists in 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 [3, 12]. The 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. Furthermore, the concepts herein presented are similar to any permutation combinatorial problem and can be adapted for solving other problems of this class with straightforward modifications [6, 14].

3.2 The Single-Locale Multicore Implementation

Algorithm 1 presents a pseudocode for the single-locale backtracking in Chapel. The algorithm starts receiving the problem to be solved (*line 1*) and the cutoff depth (*line 2*). Then, it is required to generate an initial load for the parallel search. For this purpose, *task* 0 performs backtracking from depth 1 (initial problem configuration) until the cutoff depth $cutoff$, storing all feasible, valid, and incomplete solutions at depth $cutoff$ in the active set A (*line 4*). After generating the initial load, the parallel search strategy begins through a `forall` statement (*line 5*).

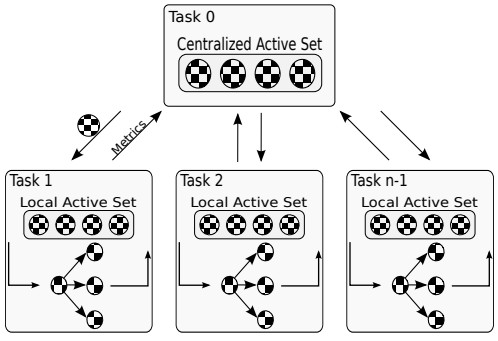

Fig. 1. Task 0 is responsible for managing the centralized active set A and performing load balancing. The searches are independent, and metrics are reduced using the *Reduce Intents* of Chapel (Own representation adapted from [9]).

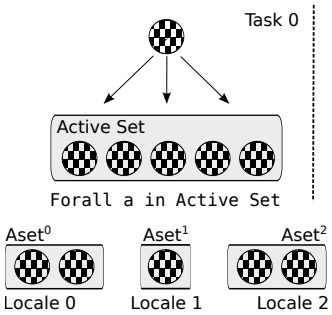

Fig. 2. Task 0 is responsible for distributing the active set across several locales. The distributed active set A_d consists of several sets $A_d^i, i \in \{0, ..., l-1\}$, where l is the number of locales on which the application is going to run.

As one can see in Fig. 1, nodes in the centralized active set A are assigned to tasks in chunks. Each task has its active set and executes a backtracking search strategy. In turn, nodes are used to initialize the backtracking, which enumerates the solution space rooted by a node. The load balancing is done through the iterator (`DynamicIters`) used to assign indexes of A to tasks, like in OpenMP. Metrics are reduced through *Reduce Intents*. In Chapel, it is possible to use the `Tuple` data type (equivalent to C-structs) and reduce all metrics at once (*line 6*). Differently from OpenMP, it is not required to define a tuple reduction. Finally, the parallel search finishes when the active set A is empty.

Algorithm 1: The multicore tree-based search algorithm.

```
1 I ← get_problem()
2 cutoff ← get_cutoff_depth()
3 A ← ∅
4 A ← generate_initial_active_set(cutoff, I)
5 forall node in A with(+ reduce metrics) do
6 |   metrics+ = tree_search(node, cutoff, I)
7 end
```

3.3 The Multi-locale Distributed Implementation

One can see in Algorithm 2 a pseudocode for the distributed tree-based search algorithm in Chapel. Thanks to Chapel's global view of control flow, the search also starts serially, with *task* 0 generating the initial load to populate the active set A (*line 4*). To make it possible to distribute A, it is required to define a domain (*line 5*) and define how this domain it is going to be distributed across different locales (*line 6*) [7]. In this work, only *standard distributions* are used [3]. Finally, the distributed active set A_d of type `Node` is defined over the domain D (*line 8*).

After the initial load generation, the nodes of A are distributed across several locales by using a parallel `forall` (*line 9*), which generates the distributed active

[3] https://chapel-lang.org/docs/modules/layoutdist.html

set A_d. Thanks to Chapel's global view of A_d, the indexes of both active sets are directly accessed in *line 9*. The compiler is responsible for the communication code. Moreover, as shown in Fig. 2, A_d is an abstraction. The distributed active set A_d consists of several sets $A_d^i, i \in \{0, ..., l-1\}$, where l is the number of locales on which the application is going to run.

The parallel search takes place in *line* 12. As one can see in Algorithm 2, its `forall` is similar to the one of Algorithm 1. However, distributed iterators are used instead (`DistributedIters`). Additionally, the distributed search exploits two levels of parallelism, and the compiler is also responsible for generating the code that exploits all CPU cores a locale has. Finally, the metrics are reduced in the same way as in the single-locale algorithm.

Algorithm 2: The multilocale tree-based search algorithm.

```
1  I ← get_problem()
2  cutoff ← get_cpu_cutoff_depth()
3  A ← ∅
4  A ← generate_initial_active_set(cutoff, I)
5  Range ← 0..(|A| − 1)
6  D ← Range mapped according to a standard distribution
7  A_d ← ∅
8  A_d ← [D] : Node
9  forall s in Range do
10 |   A_d[s] ← A[s]
11 end
12 forall node in A_d following the iterator with(+ reduce metrics) do
13 |   metrics+ = tree_search(node, cutoff, I)
14 end
```

3.4 Search Procedure and Data Structures

The kernel of both parallel algorithms previously presented is based on a serial and hand optimized backtracking for solving permutation combinatorial problems, originally written in C [6]. The serial backtracking was then adapted to Chapel, obeying the handmade optimizations, instruction-level parallelism, data structures, and C-types. The data structure `Node` is similar to any permutation combinatorial problem. It contains an unsigned 8-bit integer vector of size $cutoff$, identified by *board*, and an unsigned integer variable. The vector *board* stores the feasible and valid incomplete solution. In turn, the integer variable, identified by *bitset*, keeps track of board lines by setting its bit n to 1 each time a queen is placed in the n-th line.

The search performed by the kernel is a non-recursive backtracking that does not use dynamic data structures, such as stacks. Initially *depth* receives the value of $cutoff$. Next, *board* and *bitset* are initialized with `Node[i].board` and `Node[i].bitset`, respectively. The semantics of a stack is obtained 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.

4 Performance Evaluation of a Multi-locale Backtracking

The objective of this section is to show that it is possible to use a high-productivity language for programming distributed tree-based search algorithms and achieve metrics similar to MPI+X.

4.1 Protocol

The following programs were conceived for enumerating all valid and complete configurations of the N-Queens problem.

- **Chapel**: implementation of the multi-locale backtracking search algorithm described in Algorithm 2, written in Chapel.
- **MPI+X**: single program - multiple data (SPMD) counterpart written in C of the program above introduced. This program uses MPI for communication and X means the use of OpenMP for exploiting all cores/threads a locale has.

Both implementations use the data structures and search procedure detailed in Section 3.4. In this section, it is investigated how the application scales according to the number of locales. Furthermore, the influence of the PGAS data structure distribution on the application execution time is also studied. Tree search algorithms for solving combinatorial problems are usually highly irregular applications. Therefore, the influence of the distributed load balancing strategies on the overall performance of the application is also investigated. Finally, all metrics collected for the implementation in Chapel are compared to the ones achieved by its MPI+X counterpart.

4.2 Parameters Settings

Problems of size (N) ranging from 15 to 20 are considered. The experiments take from few seconds to several hours of parallel processing. The number of locales ranges from 1 to 32, and the application is the same for either one or more than one computer node(s). The number of locales is passed to the application using Chapel's built-in command line parameter `-nl l` (`-np l` for MPI), where l is the number of locales on which the application is executed.

All computer nodes are symmetric and operate under Debian $4.9.130 - 2$, 64 bits. They are equipped with *two* Intel Xeon X5670 @ 2.93 GHz (a total of 12 cores/24 threads), and 96 *GB* RAM. Thus, up to 384 cores/768 threads are used in the experiments. All locales are interconnected through an Infiniband network: Mellanox Technologies MT26428 (ConnectX VPI PCIe 2.0 5GT/s - IB QDR / 10GigE).

Concerning the MPI+X implementation, OpenRTE 2.0.2 along with *gcc* 6.3.0 and OpenMP 4.5 were used for compilation and execution. The Chapel implementation was programmed in its current version (1.18), and the *default* task layer (qthreads) is the one employed. Chapel's multi-locale code runs on top of

Table 1. Summary of the environment configuration for multi-locale execution and compilation.

Variable	Value
CHPL_RT_NUM_THREADS_PER_LOCALE	24
CHPL_TARGET_ARCH	*native*
CHPL_COMM	*gasnet*
CHPL_COMM_SUBSTRATE	*ibv*
GASNET_IBV_SPAWNER	*mpi*

GASNet, and several environment variables should be set with the characteristics of the system the multi-locale code is supposed to run. One can see in Table 1 a summarization of the runtime configurations for multi-locale execution. The Infiniband GASNet implementation is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with MPI, which is responsible for getting the executables running on different locales (`GASNET_IBV_SPAWNER`).

Chapel provides several standard distributions to map data structures onto locales. Different tests were also carried out to identify the best option in the context of this work. The one chosen was the one-dimension *BlockDist*, which horizontally maps elements across locales. For instance, in case $l = 3$ and $|A_d| = 8$, elements $0, ..., 2$ are on locale l_0, $3, .., 5$ on locale l_1, and $6, 7$ on locale l_2. In the scope of the present research, choosing a different standard distribution does not lead to performance improvements.

Experiments were carried out to choose a suitable cutoff depth. This parameter directly influences the size of A_d, and therefore the time spent in distributing the active set across locales. As observed in Fig. 3, the fastest data structure distribution is observed for $cutoff = 3$. However, such a cutoff value limits parallelism, resulting in a slow distributed search. In contrast, when the cutoff is set to 6, the distribution of A_d becomes $10\times$ slower than the search procedure itself. This behavior happens due to the combinatorial nature of N-Queens: a cutoff depth twice deeper results in an active set $725\times$ bigger. When choosing $cutoff = 5$, the search takes the same time as for $cutoff = 4$. Despite that, the distribution of A_d is on average $9\times$ slower for $cutoff = 5$. Thus, the cutoff depth chosen is 4. Preliminary experiments also show that $cutoff = 4$ is the best value for the MPI+X implementation.

Chapel also provides two different distributed load balancing iterators: *guided* and *dynamic*, which are also similar to OpenMP's schedules of the same name. Experiments were carried out to identify the best *chunk* for both guided and dynamic multi-locale load balancing strategies. Both strategies present the best performance using the *default* chunk size.

4.3 Results

First of all, concerning the distributed load balancing strategies provided by Chapel, using the *dynamic* iterator is from $1.17\times$ to $1.51\times$ times faster than using no load balancing (static version). Moreover, the *guided* iterator does not seem a suitable load balancing in the scope of this work: it shows benefits compared to

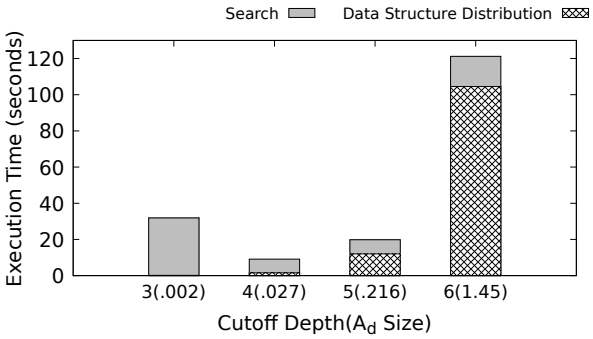

Fig. 3. Influence of the $cutoff$ choice on the execution time. Values are for the N-Queens of size $N = 17$ and 32 locales (384 cores). On the X axis: the cutoff depth and the distributed active set size in parentheses (in 10^6 nodes).

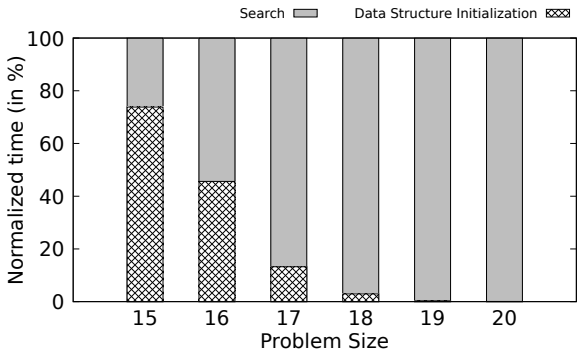

Fig. 4. Proportion of the initialization and distribution of A_d compared to the whole execution time. Results are for the N-Queens of sizes (N) ranging from 15 to 20 and executed on 32 locales (384 cores).

the static version only for sizes ranging from 18 to 20. For these problem sizes, using the guided iterator makes the search up to 1.21× faster than its static counterpart. In turn, using the dynamic distributed iterator results in a search from 1.21× to 1.25× faster than using the guided one.

The benefits of using load balancing are not observed for the smallest solution space, i.e., for the problem of size $N = 15$. In such a situation, the static search performs better because there is no communication among locales during the search. As shown in Fig. 4, the overhead of data structure initialization and distribution becomes less detrimental as the solution space grows, and the benefits of using distributed load balancing can be observed.

It is shown in Fig. 5 how the distributed search scales according to the number of locales. The worst scalability is observed for the smallest size ($N = 15$). In such a situation, the initialization and distribution of A_d amount for almost the whole execution time (see Fig. 4). For problem sizes ranging from 17 to 20, the

dynamic version scales up to 20.5× ($N = 19$), whereas guided and static scale up to 16.8× and 16.9×, respectively (also for $N = 19$). The MPI+X version scales up to 25.4× ($N = 18$). Therefore, the distributed search in Chapel achieves up to 80% of the scalability observed for its MPI+X counterpart.

As shown in Fig. 4, running the Chapel program on multiple locales comes with the overhead of distributing A_d across several nodes. However, the scalability results previously discussed take into account the search running on 1 locale (`-nl 1`). In such a situation, the active set is not distributed across different nodes, and the search works similarly to a multicore and single locale one. One can see in Fig. 6 the best speedup reached for 4 to 32 locales compared to the search running on 2. Results closer to the linear speedup are observed: for 32 locales (16× more nodes) all three variations of the search written in Chapel and the MPI implementation reach a speedup of almost 13×, which corresponds to 81% of the linear speedup.

It is worth to mention that the time spent on distributing A_d does not grow linearly according to the number of locales. As one can see in Fig. 7, the time required to distribute A_d grows up to size $N = 16$, then it becomes almost constant. This behavior comes from the fact that the size of A_d is the same for one or more locale(s). Thus, as the number of locales grows, the number of messages sent grows as well, but their size decreases. Moreover, the A_d distribution is performed in parallel (Algorithm 2, *line 9*), and the Infiniband GASNet implementation supports one-sided communication.

In terms of wall-clock time, Chapel is equivalent to MPI+OpenMP when running on one locale. For the smaller solution space (i.e., $N = 15$), Chapel stands out and it is up to 25% faster than MPI+X. In such a situation, A_d is not distributed, and the program behaves like a single-locale and multicore one. Moreover, MPI implements the SPMD programming model. This way, MPI is started, and its functions are called even for one locale. Additionally, it is worth to mention that Chapel is a compiled language and it is possible to program in

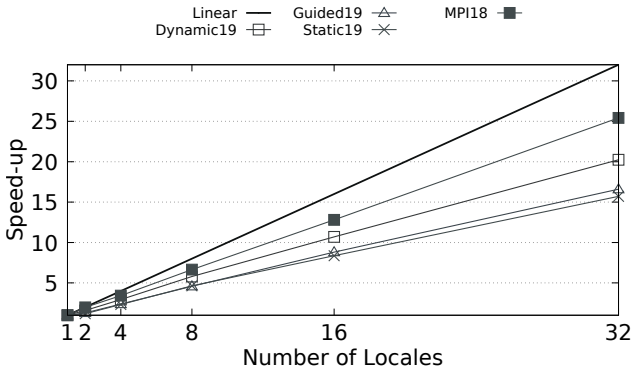

Fig. 5. The *highest* speedup achieved by Chapel and MPI+X implementations when executed on 2 (24) to 32 locales (384 cores).

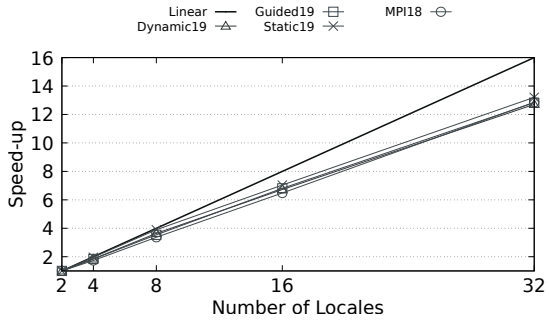

Fig. 6. The *highest* speedup achieved by Chapel and MPI+X implementations when executed on 4 (48) to 32 locales (384 cores) and compared to the execution on 2 (24).

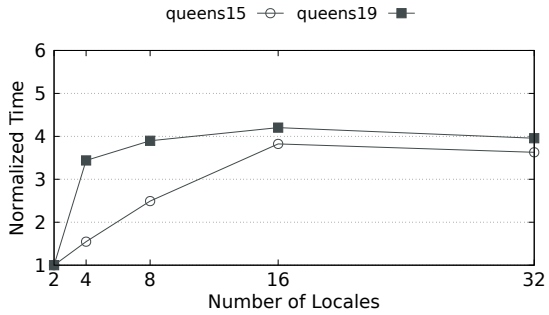

Fig. 7. Normalized time required to initialize and distribute A_d. Results are for 2 (24) to 32 locales (384 cores).

Chapel both search strategy and data structures equivalent to the ones present in its counterpart written in C. In contrast, for multiple locales and bigger problem sizes, the Chapel distributed search is on average 16% slower than its MPI+X counterpart.

5 Discussion

Chapel's high-productivity features make it straightforward to design and implement a distributed backtracking based on a multicore and single-locale version. There is no need for dealing with communication, metrics reduction, or distributed load balancing. Moreover, differently from the classic MPI+X, there is no need for using a different parallel library for each level of parallelism.

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. 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.

Chapel also shows to be competitive to MPI+OpenMP regarding performance. It is possible to program the search and data structures in a way equivalent to C, resulting in efficient single-core use. Additionally, the compiler also generates code for exploiting all cores a locale has, and the load balancing provided by Chapel is effective. One would argue that it could be possible to program an MPI+X version faster than the one used, and the results of the last section could have been more advantageous for MPI+X. However, that is also the case for Chapel.

The multicore portion of the MPI+X implementation was programmed by hand, differently from its counterpart written in Chapel. Therefore, a first improvement in the Chapel implementation is programming the code for using all CPU cores a locale has. Additionally, It is worth to mention that Chapel also supports MPI and ZeroMQ libraries for programming inter-locale communication and the distributed load balancing. However, it does not seem necessary, as the use of high-productivity features resulted in performance competitive to MPI+OpenMP.

Concerning limitations, it took much more time to configure the GASNet library for running on a cluster than programming the multi-locale code itself. Moreover, Chapel's documentation is restricted to a few system configurations, e.g., Infiniband network with Slurm for job spawning. In our case, GASNet could not run on an MXM network with a non-default partition key, and a modification in the GASNet code was necessary. This problem would keep a not so enthusiastic user from Chapel. The bright side is that it was not a Chapel-only effort, as other PGAS libraries, such as UPC, Fortran, SHMEM use GASNet as communication layer.

Finally, graphics processing units are crucial for solving big and challenging combinatorial optimization problems [5, 14]. The adoption of Chapel by the parallel optimization community, besides performance and productivity, also depends on the support of GPUs. According to Chapel's official documentation, the Xeon Phi accelerator is supported. However, there is no information concerning the support of GPUs.

5.1 Future Works

Permutation combinatorial optimization problems are commonly solved to optimality by using Branch-and-Bound search algorithms (B&B) [9]. Therefore, a 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. Additionally, this future work will aim at larger scale clusters. Thus, it will be possible to investigate the limits of the productivity-aware features of Chapel concerning performance and scalability. A final future work is to compare Chapel to other PGAS-based libraries and high-productivity languages, such as SHMEM, UPC, and Julia.

6 Conclusion

This work investigated the use of Chapel high-productivity language for the design and implementation of distributed tree search algorithms. A distributed backtracking for enumerating all valid solutions of the N-Queens problem was conceived. The concepts herein presented can be adapted for solving other permutation combinatorial problems by performing straightforward modifications.

Programmers familiarized with OpenMP can easily conceive a distributed tree-based search in Chapel. Despite the high level of its features, the distributed search written in Chapel scales well and the distributed load balancing schemes are effective. Experimental results show that Chapel is competitive to C-MPI+OpenMP in terms of performance and scalability. The most significant drawbacks found do not concern performance nor scalability. Instead, they are related to the configuration of the communication layer, which can be more time consuming than programming the distributed application itself.

It is worth to point out that the parallel optimization community already possesses legacy code mainly written in C, C++. Therefore, programmers may be resistant to learn another language and translate programs to Chapel. The capacity of Chapel to include C code can be a partial solution for this situation. One can reuse C code for node bounding and search procedure, whereas Chapel distributed programming features are employed for load balancing and communication. Finally, the lack of support for accelerators may also limit the adoption of Chapel by the parallel optimization community.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [4], hosted by INRIA and including several other organizations [4]. We thank Bradford Chamberlain, Elliot Ronaghan (from Cray inc.) and Paul Hargrove (Berkeley lab.) for helping us to run GASNet on GRID5000. Moreover, we also thank Paul Hargrove for the modifications in GASNet InfiniBand implementation necessary to run GASNet on GRID'5000 MXM InfiniBand networks.

References

1. Almasi, G.: Pgas (partitioned global address space) languages. In: Encyclopedia of Parallel Computing, pp. 1539–1545. Springer (2011)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al.: The landscape of parallel computing research: A view from berkeley. Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of (2006)
3. Bell, J., Stevens, B.: A survey of known results and research areas for n-queens. Discrete Mathematics **309**(1), 1–31 (2009)

[4] http://www.grid5000.fr

4. Bolze, R., Cappello, F., Caron, E., Dayde, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quétier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. International Journal of High Performance Computing Applications **20**(4), 481–494 (2006)
5. Carneiro, T., de Carvalho Júnior, F.H., Arruda, N.G.P.B., Pinheiro, A.B.: Um levantamento na literatura sobre a resolução de problemas de otimização combinatória através do uso de aceleradores gráficos. In: Proceedings of the XXXV Ibero-Latin American Congress on Computational Methods in Engineering (CILAMCE), Fortaleza-CE, Brasil (2014)
6. Carneiro Pessoa, T., Gmys, J., de Carvalho Junior, F.H., Melab, N., Tuyttens, D.: GPU-accelerated backtracking using CUDA dynamic parallelism. Concurrency and Computation: Practice and Experience pp. e4374–n/a (2017). https://doi.org/10.1002/cpe.4374
7. Chamberlain, B.L., Choi, S.E., Deitz, S.J., Navarro, A.: User-defined parallel zippered iterators in chapel. In: Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models. pp. 1–11 (2011)
8. Chamberlain, B.L., Ronaghan, E., Albrecht, B., Duncan, L., Ferguson, M., Harshbarger, B., Iten, D., Keaton, D., Litvinov, V., Sahabu, P., et al.: Chapel comes of age: Making scalable programming productive. In: Cray User Group (2018)
9. Crainic, T., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. Parallel combinatorial optimization pp. 1–28 (2006)
10. Cray Inc.: Chapel language specification v.986. Cray Inc. (2018)
11. Da Costa, G., Fahringer, T., Gallego, J.A.R., Grasso, I., Hristov, A., Karatza, H.D., Lastovetsky, A., Marozzo, F., Petcu, D., Stavrinides, G.L., et al.: Exascale machines require new programming paradigms and runtimes. Supercomputing frontiers and innovations **2**(2), 6–27 (2015)
12. Feinbube, F., Rabe, B., von Löwis, M., Polze, A.: Nqueens on cuda: Optimization issues. In: Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on. pp. 63–70. IEEE (2010)
13. Fiore, S., Bakhouya, M., Smari, W.W.: On the road to exascale: Advances in high performance computing and simulationsan overview and editorial. Future Generation Computer Systems **82**, 450 – 458 (2018)
14. Gmys, J., Mezmaz, M., Melab, N., Tuyttens, D.: Ivm-based parallel branch-and-bound using hierarchical work stealing on multi-gpu systems. Concurrency and Computation: Practice and Experience **29**(9), e4019 (2017)
15. Grama, A.Y., Kumar, V.: A survey of parallel search algorithms for discrete optimization problems. ORSA Journal on Computing **7** (1993)
16. Mezmaz, M., Melab, N., Talbi, E.G.: A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In: IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. pp. 1–9. IEEE (2007)
17. San Segundo, P., Rossi, C., Rodriguez-Losada, D.: Recent developments in bit-parallel algorithms. INTECH Open Access Publisher (2008)
18. Tschoke, S., Lubling, R., Monien, B.: Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In: 9th International Parallel Processing Symposium, 1995. Proceedings. pp. 182–189. IEEE (1995)
19. Zhang, W.: Branch-and-bound search algorithms and their computational complexity. Tech. rep., DTIC Document (1996)