# An Incremental Parallel PGAS-based Tree Search Algorithm

Tiago Carneiro, Nouredine Melab

# An Incremental Parallel PGAS-based Tree Search Algorithm

Tiago Carneiro
*INRIA Lille - Nord Europe*
Lille, France
tiago.carneiro-pessoa@inria.fr

Nouredine Melab
*Université de Lille, CNRS/CRIStAL*
*INRIA Lille - Nord Europe*
Lille, France
nouredine.melab@univ-lille.fr

***Abstract*—In this work, we show that the Chapel high-productivity language is suitable for the design and implementation of all aspects involved in the conception of parallel tree search algorithms for solving combinatorial problems. Initially, it is possible to hand-optimize the data structures involved in the search process in a way equivalent to C. As a consequence, the single-threaded search in Chapel is on average only $7\%$ slower than its counterpart written in C. Whereas programming a multicore tree search in Chapel is equivalent to C-OpenMP in terms of performance and programmability, its productivity-aware features for distributed programming stand out. It is possible to incrementally conceive a distributed tree search algorithm starting from its multicore counterpart by adding few lines of code. The distributed implementation performs load balancing among different computer nodes and also exploits all CPU cores of the system. Chapel presents an interesting trade-off between programmability and performance 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 scalability achieved by its C-MPI+OpenMP counterpart.**

***Index Terms*—High-productivity Language, PGAS, Chapel, MPI+OpenMP, Tree Search Algorithms.**

## I. 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 it is present in many areas, such as operations research, artificial intelligence, bioinformatics, and machine learning [1], [2]. Algorithms that belong to this class are compute-intensive and highly irregular, which demands hand-optimized data structures for efficient single-core utilization and load balancing between processes [3]–[5].

High-productivity languages historically suffer from severe performance penalties, do not provide low-level features, and are not suited to parallelism [6], [7]. Therefore, they are not often employed within the scope of parallel tree search. Instead, this kind of algorithm is frequently coded in either C or C++, and different libraries and programming models are combined for exploiting parallelism [8], [9].

Among the high-productivity languages, Chapel is one that stands out. It was designed for high-performance computing, and it is competitive to both C-OpenMP and C-MPI+OpenMP in terms of performance, considering different benchmarks [10]. The objective of the present research is to investigate whether Chapel is suitable for the design and implementation of all aspects involved in the conception of a parallel tree search algorithm for solving 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 parallel tree search algorithms. It is possible to hand-optimize the data structures involved in the search process. As a consequence, the single-threaded search in Chapel is on average only $7\%$ slower than its counterpart written in C. Whereas programming a tree search in Chapel is equivalent to C-OpenMP in terms of performance and programmability, its productivity-aware features for distributed programming stand out.

Thanks to Chapel's global view of the control flow and data structures, it is possible to conceive a distributed tree search starting from its multicore counterpart by incrementally adding few lines of code. The distributed implementation performs load balancing among different processes 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 reaches $80\%$ of the scalability reached by its C-MPI+OpenMP counterpart. Finally, the distributed load balancing strategies provided are effective: the dynamic load balancing version is up to $1.5\times$ faster than its static counterpart.

The remainder of this paper is structured as follows. Section II brings background information and the related works. Section III presents the incremental and PGAS-based distributed tree search algorithm. In turn, Section IV and Section V present the multicore and distributed evaluations, respectively. Next, Section VI brings a discussion of the results obtained in Sections IV and VI. Finally, conclusions are outlined in Section VII.

## II. Background and Related Works

### A. 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 [11].

In Chapel, both *global view of control flow* and *global view of data structures* are present [10]. 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 [12].

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 [13].

### B. Tree-based Search Algorithms

Tree-based search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree [2]. 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. As shown in Fig. 1, 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 [1]. 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.

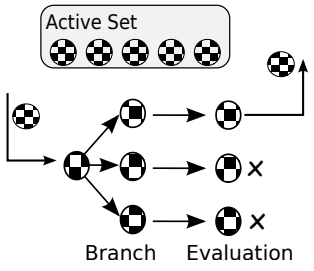



Fig. 1. Visual representation of a tree-based search algorithm (Own representation based on [9]).

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 [3]–[5], [14]. Thus, parallel tree-based search algorithms are frequently written in C/C++, due to their low-level features and supported parallel computing libraries [8]. In the context of distributed algorithms, the performance-aware strategies above mentioned are combined with distributed programming libraries for implementing load balancing and explicit communication between processes [9], [15], [16]. As a consequence, programming distributed tree search algorithms can be challenging and time-consuming.

## III. Parallel 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 [10]. In this section, a multicore and single-locale tree search algorithm is initially proposed. Then, it is incrementally extended using Chapel's productivity-aware features for distributed programming.

### A. 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 [1], [16].

This section presents two backtracking algorithms for enumerating $all$ complete and feasible solutions of the N-Queens. 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 [1].

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 [14], [17]. 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 [4], [5].

### B. 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).

As one can see in Fig. 2, 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 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
```

### C. The Multi-locale 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 distributed the nodes of $A$ across several locales, it is required to define a domain (*line* 5) and to indicate how the indexes of this domain are mapped across different locales (*line* 6). In this work, only *standard distributions* are used [1]. Finally, the distributed active set $A_d$ of type `Node` is defined over the mapped domain $D$ (*line* 8).

After the initial load generation, the nodes of $A$ are distributed by using a parallel `forall` (*line* 9), which generates the distributed active set $A_d$. Thanks to Chapel's global view of $A_d$, the indexes of both active sets are directly accessed

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

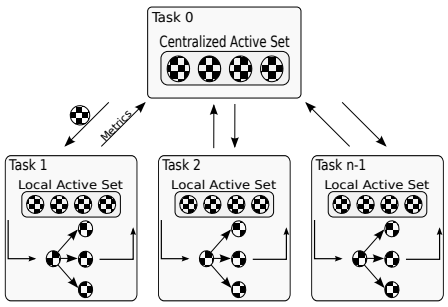



Fig. 2. Task 0 is responsible for managing the centralized active set $A$ and performing load balancing. The searches are independent, and metrics are reduced using the *Reduce Intents* of Chapel (Own representation adapted from [9]).

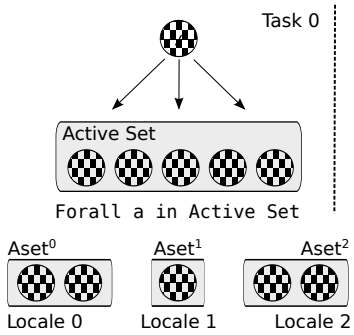



Fig. 3. Task 0 is responsible for distributing the active set across several locales. The distributed active set $A_d$ consists of several sets $A_d^i, i \in \{0, ..., l-1\}$, where $l$ is the number of locales on which the application is going to run.

in *line* 10. Moreover, as shown in Fig. 3, $A_d$ 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 multi-locale tree search algorithm.

```
1  I ← get_problem()
2  cutoff ← get_cutoff_depth()
3  A ← ∅
4  A ← generate_initial_active_set(cutoff, I)
5  Space ← {0..(|A| − 1)}
6  D ← Space mapped according to a standard distribution
7  A_d ← ∅
8  A_d ← [D] : Node
9  forall s in Space 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
```

### D. 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 [4]. 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.

TABLE I
LIST OF BEST PARAMETERS FOUND EXPERIMENTALLY FOR THE CHAPEL AND C-OPENMP IMPLEMENTATIONS.

| | *Parameters Settings* | | | |
|---|---|---|---|---|
| **Implementation** | Load Balancing | Chunk | Optimization | $cutoff$ |
| $Chapel$ | Dynamic | $default$ | `--fast` | 4 |
| $C-OpenMP$ | Dynamic | $default$ | `-O3` | 4 |

The search performed by the kernel (Algorithm 1, *line* 6 and Algorithm 2, *line* 13) 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 the incomplete solution that `Node[i]` contains.

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.

## IV. A SINGLE-LOCALE PERFORMANCE EVALUATION OF CHAPEL

The primary objective of this section is to investigate the single-locale programming features and performance of Chapel.

### A. Protocol

For this evaluation, the following programs were conceived for enumerating all valid and complete solutions of the N-Queens problem.

- **Multicore**: Chapel and C-OpenMP implementations of the backtracking search algorithm described in Section III-B.
- **Serial**: Chapel and C implementations corresponding to the kernel of the multicore programs above listed (refer to Section III-D).

All implementations apply the data structures and search procedure detailed in Section III-D.

### B. Parameters Settings

In the experiments, both multicore and serial implementations enumerate all complete and valid solutions of the N-Queens problem, for which sizes ($N$) range from 10 to 19. The experiments take from few milliseconds to several hours of parallel processing.

The testbed Operates under SMP Debian 4.9.65 64 bits, and it is composed of $two$ AMD EPYC 7301, with 32 cores @2.7 GHz, 64 threads, and 128 GB RAM. All C programs were compiled with `gcc 6.3.0`. The Chapel version used was 1.8.0.

Chapel provides three task layer implementations: *qthreads* (default), Tokio's University *Massive Threads*, and POSIX Threads (Pthreads). A preliminary experiment was performed

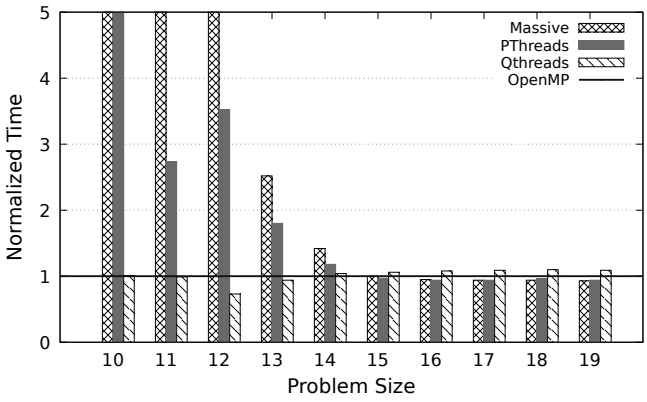



Fig. 4. Average Chapel execution time of compared to its C-OpenMP counterpart for silving the N-Queens problem. Both implementations use the parameters of Table I Results are shown for all three Chapel task layer implementations: qthreads, massive threads and Pthreads. Problem sizes (N) range from 10 to 19.

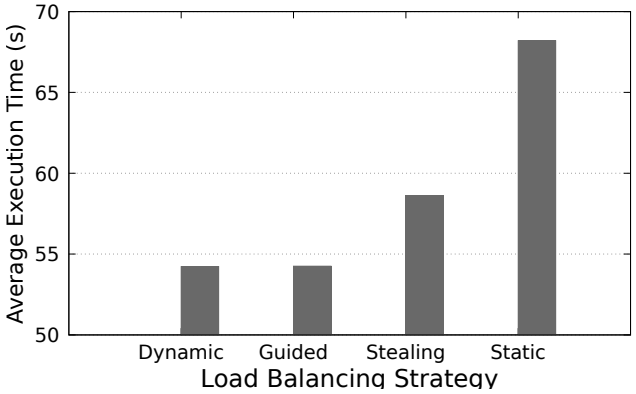



Fig. 5. Average execution time required by the multicore implementation written in Chapel to solve the N-Queens problem of size 17. Results are for the different built-in load balancing strategies provided by Chapel and *qthreads* task implementation.

to verify which thread implementation is the most advantageous in the context of this work. It is important to point out that the task layer is chosen in terms of environment variables and this action means no coding efforts. Fig. 4 shows that both massive threads and Pthreads are much heavier than qthreads for the smaller tree sizes. All task layer implementations perform similarly as the size of the solution space grows.

As OpenMP, Chapel makes available various load balancing strategies, which are implemented as built-in iterators used in `forall` statements. They are close to OpenMP's scheduling policies, such as *guided* and *dynamic*. A preliminary experiment was carried on to figure it out the best Chapel's built-in load balance strategy for solving the N-Queens. It is shown in Fig. 5 the average execution time required by the multicore Chapel backtracking to solve the N-Queens problem, taking into account different built-in load balance strategies. According to the results, the *dynamic* approach is the fastest one.

Experiments were also carried out to choose a suitable

cutoff depth for both Chapel and OpenMP implementations. One can see in Table I the best parameters experimentally found for the Chapel and C-OpenMP implementations.

### C. Results

One can see in Fig. 6 a comparison between Chapel and C serial implementations. As previously pointed out, it is possible to write in Chapel a hand-optimized code similar to C. For sizes ranging from 10 to 11, the serial implementation in Chapel is on average $1.4\times$ and $1.21\times$ slower than its counterpart written in C, respectively. However, as the size of the problem grows, this difference becomes much smaller. In turn, taking into account problem sizes ranging from 12 to 18, the Chapel serial implementation is on average $7\%$ slower than its C counterpart.

It is shown in Fig. 4 the average execution time of spent by Chapel for solving the N-Queens compared to its C-OpenMP counterpart. The results also consider all task implementations of Chapel and use the best parameters found, summarized in Table I. The version running over the qthreads task layer is comparable to C-OpenMP even for the smallest sizes (10 to 13). For sizes ranging from 14 to 18, the version over qthreads is on average $8\%$ slower than the search in C-OpenMP.

Both massive threads and Pthreads task layer implementations contrast to qthreads. For these two task layers, the overhead of managing threads amounts negatively when enumerating small solution spaces, and they perform poorly for sizes ranging from 10 to 13. The massive threads version is from $29\times$ ($N = 10$) to $2.52\times$ ($N = 13$) slower than its C-OpenMP counterpart. In turn, the implementation over Pthreads is from $16\times$ ($N = 10$) to $1.8\times$ ($N = 13$) slower than its counterpart written in C-OpenMP. As the size of the solution space grows, both Pthreads and massive threads version stand out. From sizes ranging from 15 to 19, both implementations are on average on average $5\%$ faster than C-OpenMP and $13\%$ faster than its counterpart over qthreads.

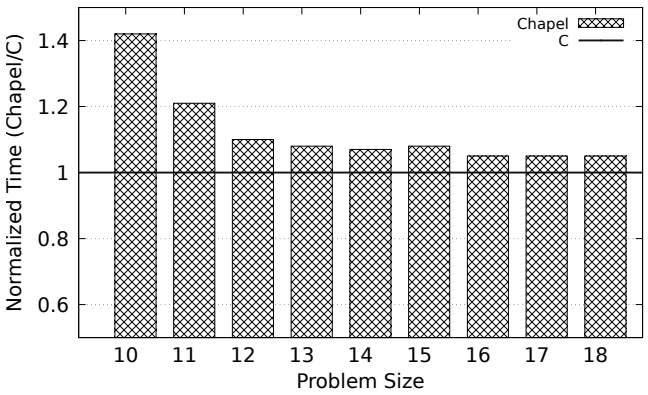



Fig. 6. Average execution time of the serial backtracking in Chapel compared to its counterpart written in C. Problem sizes (N) range from 10 to 18.

## V. A Multi-locale Performance Evaluation of Chapel

In this section, the incrementally conceived distributed algorithm presented in Section III-C is evaluated. The primary goal of this section is to show that it is possible to use a high-productivity language for programming distributed tree search algorithms and achieve metrics similar to MPI+X.

### A. Protocol

The following applications were programmed 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. In this case, MPI is applied for communication, and $X$ means the use of OpenMP for exploiting all CPU cores/threads a locale has.

Both applications implement the data structures and search procedure detailed in Section III-D.

In this evaluation, it is investigated how the applications scale according to the number of locales. Furthermore, the influence of the PGAS data structure distribution on the application execution time is also studied. Moreover, the impact 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.

### B. 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).

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 GASNet, and several environment variables should be set with the characteristics of the system the multi-locale code is supposed to run. 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.

TABLE II
SUMMARIZATION 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* |

One can see in Table II 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 (Algorithm 2, *line 2*). 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. 7, 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 load balancing strategies. They present the best performance when using the *default* chunk size.

### C. Results

First of all, the benefits of using distributed 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 slightly better because there is no communication among locales during the search. As shown in Fig. 8, 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.

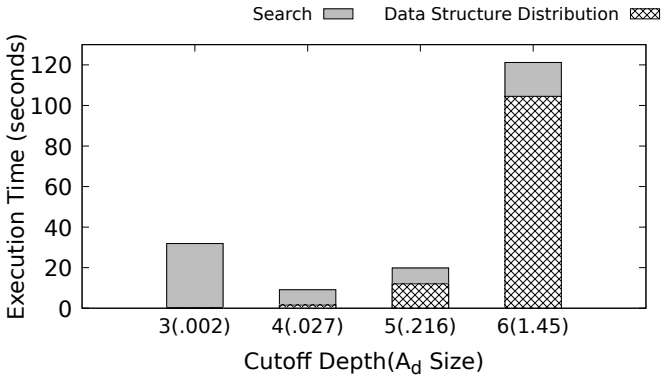



Fig. 7. Influence of the $cutoff$ choice on the execution time. Values are for the N-Queens of size $N = 17$ and 32 locales. On the $X$ axis: the cutoff depth and the distributed active set size in parentheses (in $10^6$ nodes).

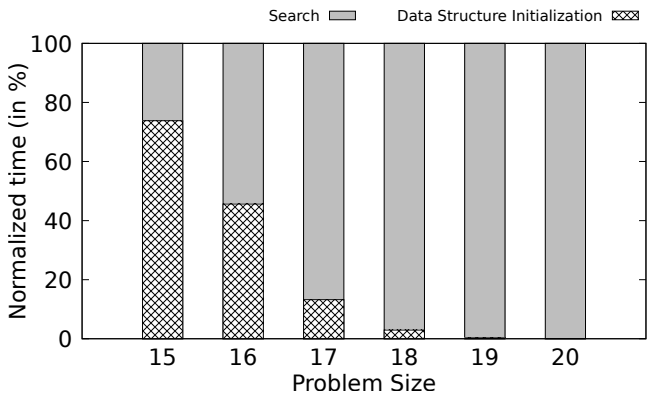



Fig. 8. Proportion of the initialization and distribution of $A_d$ compared to the whole execution time. Results are for the N-Queens of sizes ($N$) ranging from 15 to 20 and executed on 32 locales.

For sizes bigger than 15, 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 the static version only for sizes ranging from 18 to 20. For these problem sizes, using the guided iterator makes the search up to $1.21\times$ faster than its static counterpart. In turn, using the dynamic distributed iterator results in a search from $1.21\times$ to $1.25\times$ faster than using the guided one.

It is shown in Fig. 9 how the distributed searches in Chapel and MPI+X 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. 8). For problem sizes ranging from 17 to 20, the dynamic version scales up to $20.5\times$ ($N = 19$), whereas guided and static scale up to $16.8\times$ and $16.9\times$, respectively (also $N = 19$). The MPI+X version scales up to $25.4\times$ ($N = 18$). Therefore, the distributed search in Chapel achieves up to $80\%$ of the scalability observed for its MPI+X counterpart.

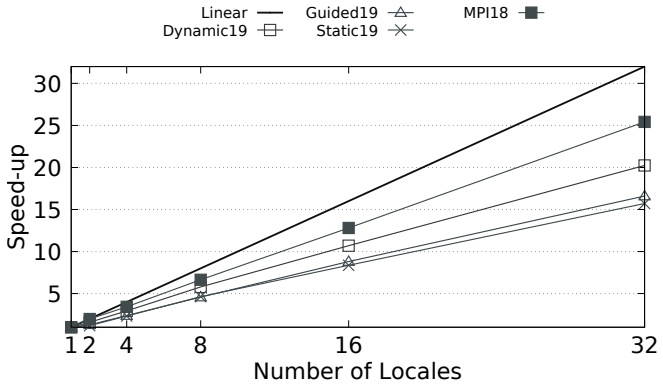



Fig. 9. The highest speedup achieved by Chapel and MPI+X implementations when executed on 2 to 32 locales.

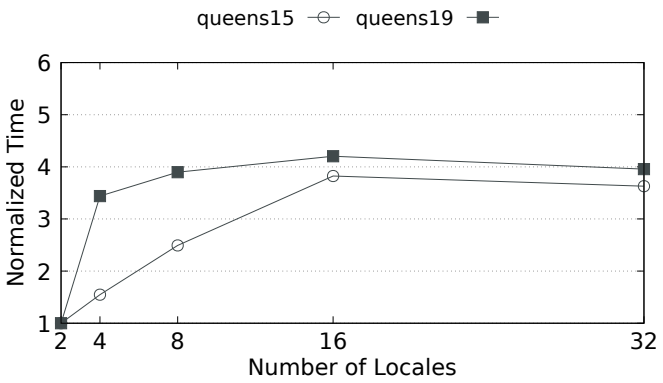



Fig. 10. Normalized time required to initialize and distribute the PGAS-based active set ($A_d$). Results are for 2 to 32 locales.

It is worth to mention that the time spent on distributing $A_d$ does not grow linearly according to the number of locales, as shown in Fig. 10. 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 ($N = 15$), 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 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.

## VI. Discussion

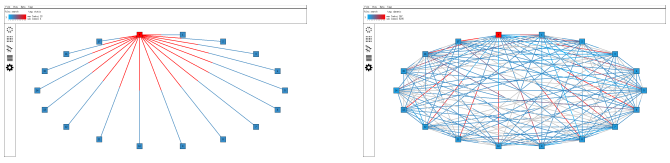
In this work, all aspects of the search process were programmed in Chapel, even though C code can be incorporated into a Chapel program. It was possible to hand-optimize the search kernel in a way similar to C. Both codes are equivalent in terms of types, data structures and code size, which resulted in a single-threaded performance competitive to C. This fact is essential in the context of this work, otherwise using the parallel features of Chapel would result in low-performance.

Programming a multicore search in Chapel involves almost the same effort as using C-OpenMP. Both provide built-in load balancing features and reduction of variables. However, Chapel presents some advantages, as it provides more load balancing strategies, and it is possible to reduce all metrics at once. Moreover, there are several task layer implementations, which may be advantageous for some users. Concerning performance, the multicore Chapel implementation using the default task layer is competitive to C-OpenMP even when solving the smallest problems.

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, or distributed load balancing. Furthermore, the compiler generates code for exploiting all CPU cores a locale has. Differently from the classic MPI+X, there is no need for an additional library to exploit each level of parallelism.

Concerning the program size, Chapel's multi-locale implementation is only $8$ *lines longer* than its single-locale counterpart, which results in a code $33\%$ bigger. Consequently, the two communication behaviors presented in Fig. 11 are achieved by the same program, but different parameters. 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 and also incorporates the SPMD programming model. Therefore, Chapel presents an interesting trade-off between programmability and performance.

The most significant limitations found concern neither programmability nor performance. Instead, they are related to



(a) (b)

Fig. 11. Output of Chplvis [18] for (a) static load distribution (b) Dynamic distributed iterator. Results are for size $N = 18$ running on 16 locales. In the figure, when the color of an edge $\{x, y\}$ is red closer to $x$ means that much more communication happen from $x$ to $y$ rather than $y$ to $x$.

technical issues. For instance, it took much more time to configure the GASNet library for running on a cluster than programming the multi-locale backtracking itself. In our case, a modification in the GASNet source code was necessary to run the Chapel distributed search on an MXM network with a non-default partition key. 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.

## VII. Conclusion

This work has investigated the use of Chapel high-productivity language for the design and implementation of all aspects involved in the conception of parallel tree search algorithms. This research covered from instruction-level parallelism used to improve the single-threaded search to the distributed and multi-level parallelism. According to the results, Chapel is a suitable language for programming such a complex and compute-intensive application. It is possible to hand-optimize the data structures involved in the search process in a way equivalent to C. Moreover, Chapel's multicore features are similar to OpenMP. Additionally, programmers familiarized with shared memory programming can incrementally conceive a multi-level and distributed tree search.

Chapel presented an interesting trade-off between performance and programmability, despite the high level of its features for distributed programming. One would argue that it could be possible to program an MPI+X version faster than the one used; however, that is also the case for Chapel. For instance, the code for exploiting all CPU cores a locale has could be programmed by hand, as well as the communication and load balancing among locales. However, the latter does not seem necessary, as the use of high-productivity features resulted in performance competitive to MPI+OpenMP.

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 [7]. The capacity of Chapel to include C code can be a partial solution for this situation. One could use C could along with Chapel's high-productivity features for distributed programming. Finally, graphics processing units are crucial for solving big and challenging combinatorial optimization problems [5]. The adoption of Chapel by the parallel optimization community, besides performance and productivity, also may also depend on the support of GPUs.

## Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [19], hosted by INRIA and including several unother organizations [2]. We thank Bradford Chamberlain, Elliot Ronaghan (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.

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

## References

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

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

[3] P. San Segundo, C. Rossi, and D. Rodriguez-Losada, *Recent developments in bit-parallel algorithms*. INTECH Open Access Publisher, 2008.

[4] 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.

[5] 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.

[6] G. Da Costa, T. Fahringer, J. A. R. Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides *et al.*, "Exascale machines require new programming paradigms and runtimes," *Supercomputing frontiers and innovations*, vol. 2, no. 2, pp. 6–27, 2015.

[7] 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.

[8] T. Carneiro, F. H. de Carvalho Júnior, N. G. P. B. Arruda, and A. B. Pinheiro, "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.

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

[10] 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.

[11] Cray Inc., "Chapel language specification v.986," *Cray Inc.*, 2018.

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

[13] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-defined parallel zippered iterators in chapel," in *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, 2011, pp. 1–11.

[14] F. Feinbube, B. Rabe, M. von Löwis, and A. Polze, "Nqueens on cuda: Optimization issues," in *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*. IEEE, 2010, pp. 63–70.

[15] S. Tschoke, R. Lubling, and B. Monien, "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.* IEEE, 1995, pp. 182–189.

[16] 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.

[17] 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.

[18] P. A. Nelson and G. Titus, "Chplvis: A communication and task visualization tool for chapel," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International.* IEEE, 2016, pp. 1578–1585.

[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.