A Local Search for Automatic Parameterization of Distributed Tree Search Algorithms

Tiago Carneiro, Loizos Koutsantonis, Nouredine Melab, Emmanuel Kieffer, Pascal Bouvry

A Local Search for Automatic Parameterization of Distributed Tree Search Algorithms

Tiago Carneiro[†], Loizos Koutsantonis[†§], Nouredine Melab[*], Emmanuel Kieffer[†], Pascal Bouvry[‡]
FSTM, University of Luxembourg[†], Luxembourg
DCS-FSTM/SnT, University of Luxembourg[‡], Luxembourg
INRIA Lille - Nord Europe[*], France
Université de Lille, CNRS/CRIStAL[*], France
Wings ICT Solutions[§], Athens, Greece
{tiago.carneiropessoa, emmanuel.kieffer, pascal.bouvry}@uni.lu, nouredine.melab@univ-lille.fr
lkoutsantonis@wings-ict-solutions.eu

***Abstract*—Tree-based search algorithms applied to combinatorial optimization problems are highly irregular and time-consuming when solving instances of NP-Hard problems. Due to their parallel nature, algorithms for this class of complexity have been revisited for different architectures over the years. However, parallelization efforts have always been guided by the performance objective setting aside productivity. Using Chapel's high productivity for the design and implementation of distributed tree search algorithms keeps the programmer from lower-level details, such as communication and load balancing. However, the parameterization of such parallel applications is complex, consisting of several parameters, even if a high-productivity language is used in their conception. This work presents a local search-based heuristic for automatic parameterization of ChapelBB, a distributed tree search application for solving combinatorial optimization problems written in Chapel. The main objective of the proposed heuristic is to overcome the limitation of manual parameterization, which covers a limited feasible space. The reported results show that the heuristic-based parameterization increases up to 30% the performance of ChapelBB on 2048 cores (4096 threads) solving the N-Queens problem and up to 31% solving instances of the Flow-shop scheduling problem to the optimality.**

***Index Terms*—Parallel tree-based search, Combinatorial optimization, Parameter configuration, High-productivity languages, Chapel**

I. Introduction

Algorithms for solving combinatorial optimization problems (COPs) can be divided into exact (complete) or approximate methods [1]. 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 [2].

Tree-based search algorithms applied to combinatorial optimization problems are highly irregular and time-consuming when solving instances of NP-Hard problems. Due to their parallel nature, algorithms for this class of complexity have been revisited for different architectures over the years [3]. However, parallelization efforts have always been guided by the performance objective setting aside productivity. The study of a feasible high-productivity language for the design and implementation of tree-based search led us to the Chapel language [4]. In the context of this work, Chapel stands out. It is a compiled language that allows one to hand-optimize data structures for performance and also provides high-level features for dealing with the irregularity of the solution space, such as iterators, which are responsible for load balancing at both distributed and intra-node levels.

Using Chapel for the design and implementation of distributed tree-search algorithms hides from the programmer perspective details, such as lower-level communication messages, load balancing, metrics reduction and distributed coherency of the incumbent solution. However, the parameterization of such parallel applications is complex, consisting of several parameters, even if a high-productivity language is used in their conception. Usually, the parameterization of the productivity-aware tree search is performed manually by using parameters from previous experiments. However, it is difficult to improve these parameters manually, as a misconfiguration might lead to poor parallel performance. Moreover, there are cases where the previous experience comes from different problems or different system architectures, which might result in poor parameterization.

This work presents a local search-based heuristic for automatic parameterization of ChapelBB, a distributed tree search application for solving combinatorial optimization problems written in Chapel. The main objective of the proposed heuristic is to overcome the limitation of hand-based parameterization, which strongly relies on executions on different systems and covers a limited solution space. Results show that the heuristic-based parameterization increases up to 30% the performance of ChapelBB on 2048 cores (4096 threads) solving the N-Queens problem and up to 31% solving instances of the Flow-shop scheduling problem to the optimality.

The remainder of this document is structured as follows. Section II brings background information introduces ChapelBB, the productivity-aware tree-based search to be parameterized. In turn, Section III presents the heuristic-based approach for parameterization of ChapelBB. The evaluation of the heuristic-based parameterization is shown in Section IV. Finally, conclusions are outlined in Section V.

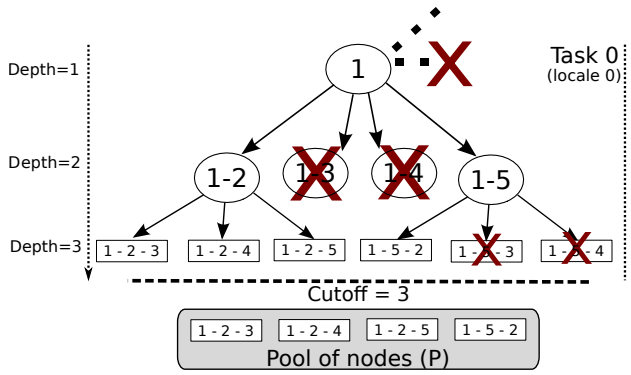

Fig. 1. Schematic representation of an initial search on *locale 0 - task 0* that generates the pool P for a permutation combinatorial problem size $N = 4$ and $cutoff = 3$. The figure depicts the branch that has the element 1 of the permutation as the root and generated 4 valid and feasible incomplete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

II. The Productivity-aware Distributed Tree-based Search

This section introduces ChapelBB, a productivity-aware distributed tree search designed and implemented in Chapel. The application solves permutation-based combinatorial optimization problems to the optimality and implements the master-worker scheme detailed in Section II-B.

A. The Chapel High-productivity Language

Chapel is an open-source parallel programming language designed to improve productivity in high-performance computing. In Chapel, the program is started with a single task, and parallelism is *added* through data or task-parallel features [5]. Furthermore, as Chapel belongs to the partitioned global address space (PGAS) languages, the application has a global memory addressing space, and each segment of this space is assigned to a different locale [6]. In Chapel, a *locale* is similar to a MPI process, and a computer node can host one or more locale. Due PGAS, a task can refer to any variable lexically visible, whether this variable is placed on the same locale on which the task is running, or on the memory space of another one. Moreover, indexes of data structures can be globally expressed, even in the case where the implementation of such data structures distributes the indexes across several locales.

Iterators in Chapel: are similar to procedures that can be used to isolate iterations from the loop body. Each value yielded by the iterator corresponds to an iteration of the loop. Chapel provides different parallel and distributed iterators that implement load balancing between computer cores and processes. Related works on distributed tree search show that the iterators provided by Chapel are the key feature to achieve a trade-off between productivity and parallel efficiency [4], [7]. The complex communication pattern of a distributed master-worker algorithm is encapsulated by the iterators. As a consequence, there is no need for explicitly dealing with work distribution, load balancing, termination criteria, or metrics reduction.

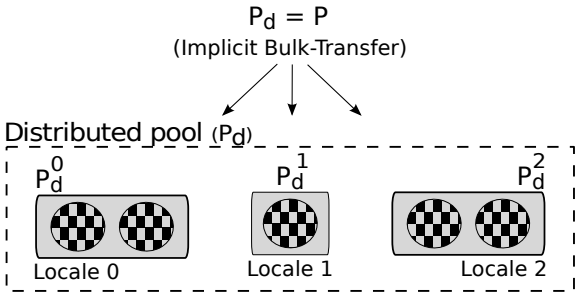

Fig. 2. The distributed pool P_d consists of several sets $P_d^i, i \in \{0, ..., L-1\}$, where L is the number of locales on which the application runs. This distribution is performed through an attribution operator, which invokes an implicit bulk transfer.

B. The Master Locale and the Initial Search

ChapelBB (Algorithm 1) starts with *task* 0 running serially on *locale* 0. As ChapelBB is an algorithm that follows the master-worker load distribution scheme, it is necessary to generate an initial load which should be kept into the distributed pool of nodes P_d. As illustrated in Figure 1, this initial load is generated through a partial search (*line* 5) referred to as the *initial search*. The latter is *partial* as it just evaluates sequentially a small portion of the solution space with the sole objective to generate valid, feasible, and incomplete solutions (subproblems) to be kept into P_d.

As one can see in Algorithm 1, *task* 0 initially receives the size N of the problem and the cutoff depth (*lines* $1-2$). This work focuses on permutation-based combinatorial problems, for which an N-sized permutation represents a valid and complete solution. Therefore, in the partial search, task 0 implicitly enumerates all *feasible* and *valid* incomplete solutions containing $cutoff$ elements of the permutation, keeping them into the pool P (*line* 3), which is local to *task* 0. Lines 6 to 8 are responsible for defining the PGAS-based pool of nodes P_d. In line 9, P_d receives via implicit bulk transfer the nodes of P.

The parallel search takes place in line 10, *adding* parallelism by using the `forall` statement along with distributed iterators (`DistributedIters`), which are responsible for the as-

Algorithm 1: The Master-worker scheme.

```
1  N ← get_problem( )
2  cutoff ← get_cutoff_depth( )
3  P ← {} Node
4  metrics ← (0, 0)
5  metrics += initial_search(N, cutoff, P)
6  Size ← {0..(|P| − 1)} // Domain
7  D ← Size mapped onto locales to a standard distribution
8  Pd ← [D] : Node
9  Pd = P // Using implicit bulk-transfer
10 forall node in Pd following a distributed iterator with(+ reduce
     metrics) do
11 |  metrics += Search(N, node, cutoff)
12 end
13 present_results(metrics)
```

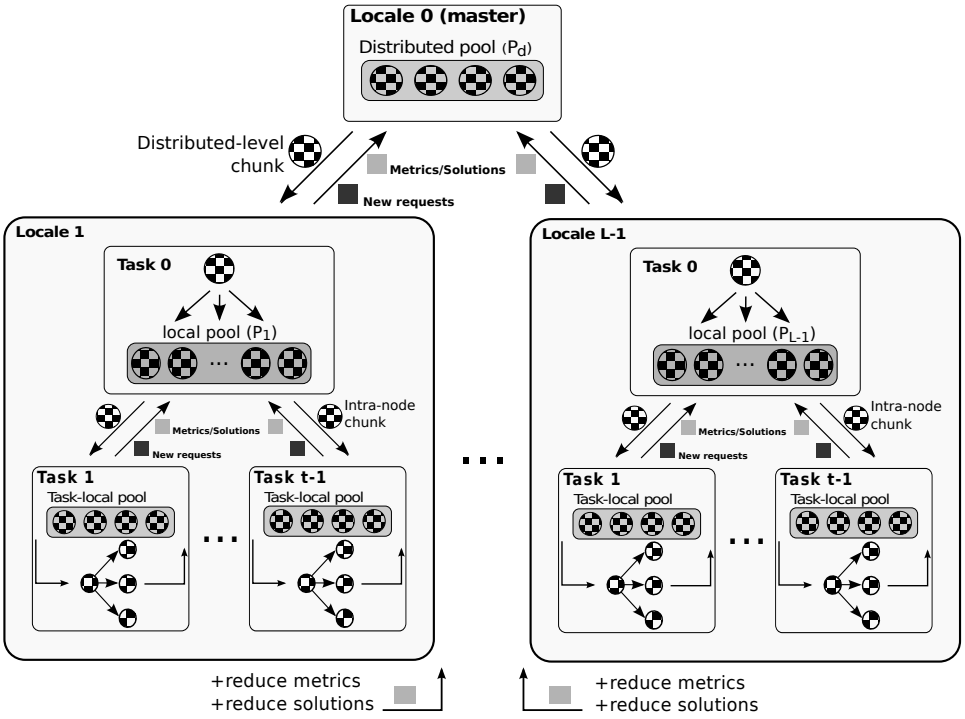

Fig. 3. In the master-worker scheme followed by ChapelBB, locale 0 (master) is responsible for generating the pool P_d and controlling the search, distributing subproblems for the worker locales according to a distributed iterator. In the intra-node level, each worker locale receives nodes from the master and task 0 generates a local pool (P_l). In turn, subproblems are distributed to tasks according to an intra-node iterator. Figure adapted from [8]).

signment 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`).

In the context of this work, iterators can be considered similar to OpenMP schedulers. Finally, *both* intra- and inter-locale levels of parallelism are exploited by using Chapel parallel iterators. For more details about ChapelBB, the interested reader may refer to [4], [9].

C. Aspects of Implementation

So far, two implementations of ChapelBB have been developed. The first one corresponds to a distributed backtracking for enumerating all complete and valid solutions of the N-Queens problem. This latter consists of placing N non-attacking queens on a $N \times N$ chessboard, and it is often used as a benchmark for novel tree-based search algorithms [10]. The N-Queens is a proof-of-concept that motivates further improvements in solving related combinatorial optimization problems. The second implementation is a distributed branch-and-bound (BB) for solving instances of the Flow-shop scheduling problem (FSP), a classical NP-Hard problem, to the optimality. In short, FSP aims at scheduling N jobs on M machines $\{m_1, m_2, \ldots, m_M\}$ in that order. For more details, refer to [4].

The implementation of *ChapelBB* is openly available at the *Chapel-based Optimization* repository [11].

III. A Local Search-based Approach for the Parameterization of ChapelBB

A. Challenging Issues

In order to maximize parallel performance and efficiency, on *each* computer ChapelBB is executed, it is required to provide to the application configuration parameters. These parameters are found manually and based on previous experience. It is important to point out that it is required to find good parameters for *each system* on which ChapelBB runs, as a good parameters configuration for a given cluster might not work for another one. Additionally, a parameters configuration found for c computer nodes on the same system might not be good for a different number of computer nodes. The same goes for multiple problems: each implementation requires a specific parametrization.

Consider the tuple $\mathcal{T} = (S, c, \pi, \Pi)$, where S is the system on which ChapelBB is executed, c is a system configuration in terms of computer nodes, and π is the instance of the problem Π to be solved to the optimality. This way, the objective is to find a parameters vector $\mathcal{P}$ which minimizes the execution time of $ChapelBB(\mathcal{T}, \mathcal{P})$. In this context, the challenge is to find a parameters vector $\mathcal{P}$ which minimizes the execution time of $ChapelBB(\mathcal{T}, \mathcal{P})$ and also presents good performance for another configuration c' and another instance $\pi' \in I$, where I is the set of instances of Π used in the experiments.

Before introducing the algorithm to find by hand feasible and valid parameter configuration for ChapelBB, the parame-

TABLE I
PARAMETERS USED TO CONFIGURE THE DISTRIBUTED SEARCHES AND THE VALUES THEY CAN RECEIVE.

Parameter	Alias	Description	Possible values
c_1	$First\ Cutoff$	Initial cutoff depth	3,4,5
c_2	$Second\ Cutoff$	Second cutoff depth	6,7,8
c_3	$Load\ balancing$	Distributed load balancing	$static$, $dynamic$, $guided$
s_1	$Distributed\ chunk$	Distributed load balancing chunk size	1,4,8,16,32
s_2	$Intra-node\ Chunk$	Intra-node load balancing ($dynamic$) chunk size	8,16,32,64,128
s_3	$Hyperthreading$	Use hyper-threading?	$true$, $false$
t_1	$Coordinated$	Locale 0 acts only as the coordinator?	$true$, $false$
t_2	$PGAS$	PGAS-based active set?	$true$, $false$

ters used to configure the distributed search are introduced as follows.

B. Parameters

As one can see in Table I, an implementation ChapelBB for solving to optimality instances of the problem Π receives eight parameters for execution, which are divided into three classes that reflect their importance: $Critical$, $Secondary$ and $Tertiary$.

The parameters for which a misconfiguration drastically affects the parallel performance and efficiency of the search are classified as $Critical$. There are three parameters in this class: c_1 – the initial cutoff depth; c_2 – the second cutoff depth, and c_3 – the distributed load balancing scheme. In turn, the $Secondary$ class consists of the chunk size of the distributed and intra-node load balancing schemes – s_1 and s_2, respectively, and whether to use hyperthreading for improving intra-node parallelism – s_3. Finally, the least critical parameters, which belong to the $Tertiary$ class are the binary variables t_1 and t_2, which corresponds to the decision of removing the master locale (id 0) from the search process, using it just to coordinate the search, and whether to distribute or not the centralized pool of nodes P_d (See Figure 2), which could improve locality. Therefore, $\mathcal{P} = [c_1, c_2, c_3, s_1, s_2, t_1, t_2]$.

C. The Manual Algorithm

An initial parameters configuration $\mathcal{P}$ is chosen according to previous experience, from results returned from executions of ChapelBB on another system S'. Thus, the initial step of the manual algorithm is to execute $ChapelBB(\mathcal{T}, \mathcal{P})$ to get its execution time e. Then, we perturb the critical parameters of $\mathcal{P}$, usually one at a time, generating the neighboring vector $\mathcal{P}'$ to verify whether this perturbation results in execution time improvement. If P' leads to a better performance, a perturbation is performed on P', resulting in P''.

If a local optimum is found in the region of the critical parameter, the next step is to explore less critical parameters. However, in the manual algorithm, the search for better secondary and tertiary parameters can be neglected, as a wrong configuration might not degrade considerably the parallel performance and efficiency of *ChapelBB*. The last step of the manual algorithm is to run experiments on different system configurations and for a reduced set of instances. The parameters configuration which returns the best overall results is then chosen.

D. The Hill-climbing Algorithm

Explicit enumeration of all possible configurations for a given $\mathcal{T}$ corresponds to $16,200$ possibilities, and this parameter choice would only be the proven best for $\mathcal{T}$. Based on the manual algorithm, we propose a hill climbing approach for automatic parallelization of *ChapelBB*. Our objective is not to rely only on previous experience and overcome the limitation of manual parameterization, which neglects the secondary and tertiary parameters.

As it can be observed in Algorithm 2, the hill climber starts with an arbitrary parameters vector $\mathcal{P}_{hill}$ and executes $ChapelBB(T, \mathcal{P}_{hill})$, to get its cost $hill\_cost$ in terms of execution time. Then, the algorithm performs a local search on $\mathcal{P}_{hill}$, returning the best parameters configuration that can be found in its neighborhood. In case an improvement is found, $\mathcal{P}$ and $best\_cost$ are updated and the local search is called until no further improvement is possible.

E. The Local Search

Due to the huge amount of time required for the search to complete when bad parameters are chosen, the local search does not evaluate a large solutions space, only neighbors with distance *one* from P_{hill}. For each parameter p of $\mathcal{P}_{hill}$, the search generates the neighboring vectors $\mathcal{P}_{left}$ and $\mathcal{P}_{right}$. For instance, if c_3 (See Table I) in $\mathcal{P}_{hill}$ is $dynamic$, $\mathcal{P}_{left}$ has all elements equal to the ones of $\mathcal{P}_{hill}$ but c_3, which is

Algorithm 2: A hill climbing algorithm for automatic parameterization of ChapelBB.

Input: The tuple $\mathcal{T}$.
Output: A parameters vector $\mathcal{P}$ and the cost of $ChapelBB(\mathcal{T}, \mathcal{P})$.

```
1  P_hill ← get_random_parameters( )
2  P ← ∅
3  best_cost ← 0
4  hill_cost ← ChapelBB(T, P_hill)
5  while hill_cost < best_cost do
6  |   best_cost ← hill_cost
7  |   P ← P_hill
8  |   < P_hill, hill_cost > ← local_search(T, P_hill, hill_cost)
9  end
10 return < P, hill_cost>
```

Algorithm 3: The local search used by the hill climber.

Input: The tuple $\mathcal{T}$, the parameters vector $\mathcal{P}_{hill}$, and $hill\_cost$.
Output: A tuple $< \mathcal{P}_{best}, search\_cost >$, which corresponds to the best parameters found by the local search and the cost of $ChapelBB(\mathcal{T}, \mathcal{P}_{best})$, $search\_cost$.

```
P_left ← ∅, P_right ← ∅
P_best ← P_hill
left_cost ← 0, right_cost ← 0
best_cost ← 0
search_cost ← hill_cost
forall parameter p in P_hill do
    P_left ← get_left_neighbor(P_hill, p)
    P_right ← get_right_neighbor(P_hill, p)
    left_cost ← ChapelBB(T, P_left)
    right_cost ← ChapelBB(T, P_right)
    best_cost ← min(left_cost, right_cost)
    if best_cost < search_cost then
        search_cost ← best_cost
        if left_cost < right_cost then
            P_best ← P_left
        else
            P_best ← P_right
        end
    end
end
return < P_best, search_cost >
```

equals to $static$. In turn, $\mathcal{P}_{right}$ has c_3 equals to $guided$. The concepts of $left$ and $right$ are similar to the *next* and the *previous* elements of a circular doubly linked. If the local search finds a cost improvement, the search returns the best parameters vector $\mathcal{P}_{best}$ and the cost of $ChapelBB(\mathcal{T}, \mathcal{P}_{best})$, $search\_cost$.

F. Aspects of Implementation

As parallelism is added in Chapel, it is possible to call the ChapelBB itself as the cost function of the parameterization heuristic, as shown in both pseudocodes previously presented. Moreover, the heuristic can be used also as an initial parameters generation, so the search can be started without parameters, only receiving $\mathcal{T}$. In the current implementation, the parameterization is only performed for a given tuple $\mathcal{T}$. If the user wants to get a parameter for another system configuration or instance, it is required to start the search receiving a different $\mathcal{T}$.

IV. Evaluation

A. Experimental Protocol

In this evaluation, the hill climbing algorithm detailed in Section III-D is used to find feasible parameters for two implementations of ChapelBB: the first one solves instances of the FSP to the optimality, whereas the second one enumerates all feasible and complete solutions of the N-Queens problem. We compare the results obtained by the hill climber to the ones obtained manually, based on previous executions on another system, which has a different processor architecture and high-performance network.

As pointed out in Section III-C, in the manual parameterization, we run a set of experiments, for a reduced set of instances, on a number of nodes smaller than the one we want to run the final experiments. The reason for such a choice is that a reservation containing several nodes is usually difficult to get. Moreover, good parameters for a high number of computer nodes differ considerably from the parameters for a small number of nodes and single-node execution. In this way, both the manual and the heuristic-based parameterization are performed on 20 computer nodes. Then, we compare the execution times for 32 computer nodes.

In the experiments, N-Queens problems of size (N) ranging from 17 to 21 are considered. The instance chosen for parameterization is the N-Queens of size $N = 19$, which can be considered as a big instance. In turn, the FSP benchmark instances used in our experiments are the FSP instances defined by Taillard [12]. We use 9 instances where $M = N = 20$. For most instances where $M = 5$ or 10, the bounding operator gives such good lower bounds that it is possible to solve them in few seconds using a sequential B&B. Instances with $M = 20$ and $N = 50, 100, 200$ or 500 are very hard to solve. For example, the resolution of the instance *Ta056* ($N = 50$, $M = 20$), performed in [13], lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years. For the parameterization, the instance chosen is $ta28$, which, among the 9 selected instances for the experiments, is a medium-sized one.

To compare the performance of two tree-based search algorithms, both should explore the same search space. When an instance is solved twice using a parallel tree search algorithm, the number of explored nodes varies between two resolutions. Therefore, for all FSP instances, the initial upper bound (cost of the best found solution) is set to the optimal value, and the search proves the optimality of this solution. This initialization ensures that precisely the critical sub-tree is explored, *i.e.*, the nodes visited are exactly those nodes which have a lower bound smaller than the optimal solution [14].

B. Experimental Testbed

The experiments were executed on the new Aion Cluster, hosted at the University of Luxembourg. All computer nodes are symmetric and operate under Red Hat Enterprise Linux 8.3, 64 bits. The nodes are equipped with two AMD Epyc

TABLE II
SUMMARY OF THE ENVIRONMENT CONFIGURATION FOR MULTI-LOCALE EXECUTION AND COMPILATION.

Variable	Value
CHPL_RT_NUM_THREADS_PER_LOCALE	128
CHPL_TARGET_CPU	*native*
CHPL_HOST_PLATFORM	*linux*64
CHPL_LLVM	*none*
CHPL_COMM	*gasnet*
CHPL_GASNET_SEGMENT	*everything*
CHPL_COMM_SUBSTRATE	*ibv*
GASNET_PSM_SPAWNER	*ssh*

ROME 7H12 @ 2.6 GHz, a total of 64 cores (128 threads) per node, and 256 GB ram. In the experiments, up to 32 nodes have been reserved, total of 2048 cores, hosting 4096 threads. All computer nodes are interconnected through a InfiniBand (IB) HDR100 network, configured over a Fat-Tree Topology. The number of locales (Chapel processes) is passed to the application using Chapel's built-in command line parameter `-nl L`, where L is the number of locales on which the application is executed. In these experiments, each computer node hosts *one* Chapel locale.

Both the hill climber and ChapelBB implementations were programmed for Chapel version 1.25.0, 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. One can see in Table II a summary of the runtime configurations for multi-locale execution. The Infiniband GASNet implementation is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with SSH, which is responsible for getting the executables running on different locales (`GASNET_PSM_SPAWNER`).

C. Experimental Results

One can see in Table III parameters found manually and *via* the hill climbing heuristic for the N-Queens and the FSP problems. A first detail that is it worth to mention concerns the parameters found by the heuristic for the N-Queens problem. The previous experience shows us that the $dynamic$ distributed load balancing scheme presents the best performance for both the N-Queens and the FSP problems [4], [9]. In turn, on the Aion cluster, which is AMD-based, ChapelBB shows the best performance enumerating the solutions of the N-Queens using $guided$ as the load balancing scheme. For all other scenarios, the best configuration uses the $dynamic$ iterator. It is important to point out the limitation of relying on previous experience without proper evaluation of all classes of parameters. Without using the heuristic, we would continue to use the $dynamic$ iterator instead of exploring other configurations for the load distribution scheme.

For the manual parameters choice, the best initial and secondary cutoff depths found are 4 and 8, respectively. In turn, the use of a heuristic made it possible to find a initial cutoff depth deeper for the N-Queens and a shallower depth for the FSP. A deeper initial cutoff might result in fine-grained subproblems, and a shallower depth might result in subproblems with a coarser granularity, which might result in load imbalance. First, sub-tress that would became unfeasible in deeper depths are not pruned yet in the enumeration process. Second, with coarse-grained subproblems, computer nodes might starve more easily, as the pool gets empty while other computer nodes did not finish their enumeration.

TABLE III
PARAMETERS FOUND MANUALLY AND VIA HEURISTIC TO EXECUTE CHAPELBB. THE ORDER OF THE PARAMETERS FOLLOWS TABLE I.

	Parameters
Heuristic–Queens	$(guided, 5, 8, 1, 8, 128, false, false)$
Manual–Queens	$(dynamic, 4, 8, 16, 32, 128, false, false)$
Heuristic–FSP	$(dynamic, 3, 7, 8, 8, 128, true, false)$
Manual–FSP	$(dynamic, 4, 8, 64, 32, 128, false, false)$

For the N-Queens, the size of the centralized pool using the heuristic, for $N = 18$, is $8.6\times$ bigger than the one using the manual parameters. On the other hand, there is no time spent in distributing the active set as the $PGAS$ flag is set to `false`. The time is only spent when the master sends tree nodes to other locales. However, in the current implementation of Chapel, this data transfer for the distribution of the active set is residual.

Concerning the parameters in the Tertiary class, for the manual parameterization, it is not worth to make several experiments with the $PGAS$ flag as it did not affect significantly the performance of the application. Only the heuristically chosen parameters for the FSP problem are using the $PGAS$ flag set to `true`. Furthermore, the heuristic found the same intra-node chunk size for both problems – 128. The choice of such parameter was not performed manually. The value of 128 is based on previous experience.

Performance Results: One can see in Figure 4 the comparison of ChapelBB's parallel performance configured with the manually and heuristically chosen parameters. Concerning the N-Queens problem, and for the smaller size ($N = 17$), ChapelBB with the heuristically chosen parameters can be almost twice as slow as its counterpart using the manually chosen parameters. As the solution space grows, it is possible to improve the performance of ChapelBB solving the N-Queens from 24% ($N = 19$) to 30% ($N = 21$) on 32 computer nodes (2048 cores). The main reason for this big difference can be explained by primary parameters. Using the deeper initial depth allied to the $guided$ distributed iterator made the search more efficient. Using the $dynamic$ iterator with the hand-chosen parameters results in load imbalance. For $N = 21$, the biggest load processed by a locale is $1,47\times$ higher than the smallest load. In turn, the heuristic-based parameters, which use the $guided$ iterator and a deeper initial depth, show the biggest tree processed by a locale only 5% bigger than the smallest tree.

Finding a good parameters configuration is easier for the N-Queens: a bigger problem keeps the characteristics of the smaller ones. Thus, the results are similar for $N > 18$. However, when it comes to the FSP, the parameterization is much more difficult, even though the instances belong to the class for which $N = M = 20$. Each instance has its characteristic in terms of solution space size and irregularity. Despite these challenging issues, ChapelBB configured with the heuristically-chosen parameters shows performance inferior to its manually-configured one only for the instances $ta22$ (-22%), $ta28$ (-30%) and $ta30$ (-33%). It is important to point out that $ta22$ and $ta30$ are the smallest instances. This way, this performance difference, which might look big in percent, corresponds to about one second of execution time.

Equivalent performance is observed for both parameteriza-

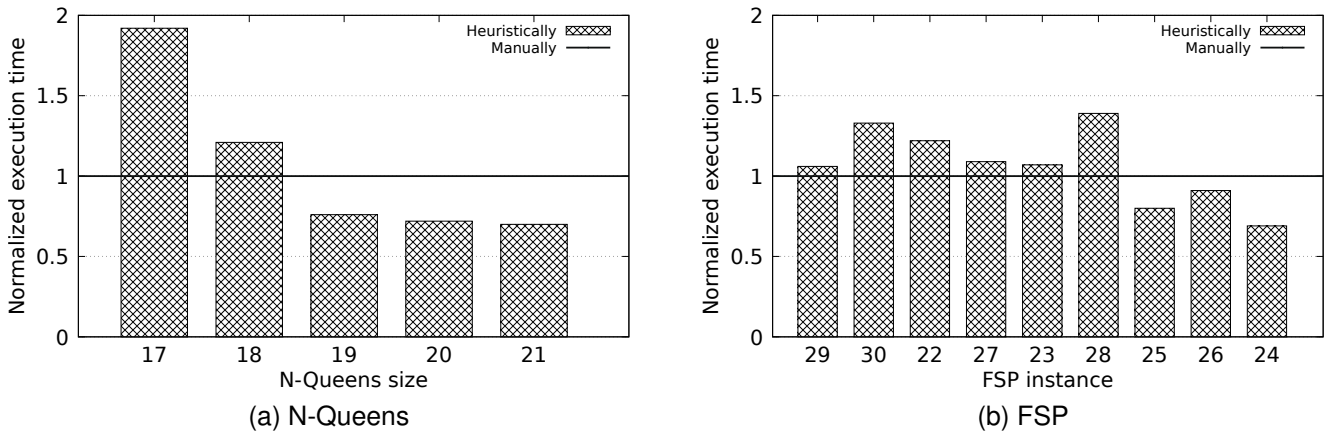

Fig. 4. Relative execution time achieved by ChapelBB configured with the heuristically-chosen parameters compared to its manually configured counterpart (a) enumerating all feasible and valid solutions of the N-Queens problem and (b) solving instances of the FSP to the optimality. The FSP instances are ordered according to their tree sizes. Results are shown for 32 computer nodes – 2048 cores (4096 threads).

tion strategies for three instances: $ta23$, $ta27$ and $ta29$. By using the heuristic, it is possible to improve the performance of ChapelBB for the three biggest instances $ta24$, $ta25$ and $ta26$ in 31%, 20% and 11%, respectively. As these instances are the biggest, e.g., solving $ta24$ to the optimality takes 110 seconds on 2048 cores – 4096 threads and using the heuristically-chosen parameters, the average execution time of ChapelBB using the heuristically-chosen parameters is 13% lower than its version initialized with the hand-chosen parameters.

V. Conclusions and Future Research Directions

This work presented a heuristic-based approach for the parameterization of ChapelBB, a productivity-aware tree-based search for solving combinatorial problems. The proposed heuristic is a hill-climbing approach based on the manual algorithm previously used for the parameterization of ChapelBB. A first concluding remark is to point out the limitation of manual parameterization. For instance, relying on previous experience and performing a poor search for new secondary and tertiary parameters limited the performance of ChapelBB, mainly when solving the N-Queens or bigger FSP instances.

The reported results show that the heuristic-based approach improves the performance of ChapelBB in around 30% for $N > 18$ solving the N-Queens. Solving instances of the FSP is a more complex scenario. It is valid to mention that both ways of parameterization result in equivalent results for the small and medium cases. In turn, the heuristic-based parameterization is more efficient for the biggest instances, for which improvements result in significant execution time gains. In such a scenario, it was possible to decrease the execution time of ChapelBB from 13% to 31%.

A first improvement to the hill-climbing search implementation is to use a solution found by hand as the initial solution. This way, it would be possible to overcome the main flaw of the hand-made parameterization, which is the limited search of better secondary and tertiary parameters. Finally, ChapelBB also is implemented to harness all GPUs and CPUs of the system altogether. The parameterization of such a scenario is more challenging, as parameters from different architectures need to be taken into account. Therefore, we also consider implementing a version of the hill-climber for the parameterization of this complex scenario.

References

[1] 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.

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

[3] N. Melab, J. Gmys, M.-S. Mezmaz, and D. Tuyttens, "Multi-core *versus* many-core computing for many-task branch-and-bound applied to big optimization problems," *Future Gener. Comput. Syst.*, vol. 82, pp. 472–481, 2018.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] T. Carneiro, "Chapel-based optimization," https://github.com/tcarneirop/ChOp, 2022.

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

[13] 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.

[14] G. Karypis and V. Kumar, "Unstructured tree search on SIMD parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1057–1072, 1994.