PGAS Data Structure for Unbalanced Tree-Based Algorithms at Scale

Guillaume Helbecque[a,b], Tiago Carneiro[c], Nouredine Melab[b], Jan Gmys[b], Pascal Bouvry[a]

[a] *Université du Luxembourg, DCS-FSTM/SnT, Luxembourg*
[b] *Université de Lille, CNRS/CRIStAL, Centre Inria de l'Université de Lille, France*
[c] *Interuniversity Microelectronics Centre, Belgium*

Abstract

The design and implementation of algorithms for increasingly large and complex modern supercomputers requires the definition of data structures and workload distribution mechanisms in a productive and scalable way. In this paper, we propose a PGAS data structure along with a Work-Stealing mechanism for the class of parallel tree-based algorithms that explore unbalanced trees using the depth-first search strategy. The contribution has been implemented and packaged as an open-source module in the Chapel PGAS language. The experimentation of the contribution in a single-node setting using backtracking applied to fine-grained Unbalanced Tree-Search benchmark shows that 68% of the linear speed-up can be achieved. In addition, the scalability of the contribution has been evaluated using the Branch-and-Bound algorithm to solve big instances of the Flowshop Scheduling problem on a large cluster. The reported results reveal that 50% of strong scaling efficiency is achieved using 400 computer nodes (51,200 processing cores).

Keywords: Depth-first Tree Search, PGAS, Scalable Data structures, Work Stealing

1. Introduction

In the landscape of modern programming environments, the definition of efficient and versatile data structures is a fundamental requirement. This need becomes even more pronounced in Partitioned Global Address Space (PGAS) environments that are inherently tailored for distributed computing, where the ability to effectively manage data structures across clusters is pivotal. In this work, the focus is on tree-based algorithms that explore unbalanced solution spaces. This class of algorithms has garnered significant attention due to their capacity to offer viable solutions to problems in different areas, such as Operations Research, Artificial Intelligence, Bio-informatics, and Machine Learning [1; 2].

These algorithms, such as backtracking and Branch-and-Bound (B&B), are able to efficiently explore solution spaces. However, they exhibit large irregular trees making their design in a parallel distributed context raising multiple challenges. The major of

them is the design of efficient and scalable data structures and dynamic adaptive load balancing mechanisms.

To raise that challenge, we introduced a PGAS data structure, called `DistBag_DFS`, along with a Work Stealing (WS) mechanism for the design and implementation of unbalanced Depth-First (DFS) tree-search at scale [3]. The implementation is based on the Chapel programming language [4] and packaged in the open-source `DistributedBag_DFS` module [5].

In this paper, we extend this previous work addressing its limitations. First, we provide a comprehensive description of `DistBag_DFS` and associated WS, along with a performance evaluation of this latter mechanism on the backtracking fine-grained Unbalanced Tree-Search (UTS) benchmark. Then, we investigate its performance in a large-scale distributed-memory setting using B&B applied to the Permutation Flowshop Scheduling problem. Up to 400 computer nodes (51,200 processing cores) are used to evaluate the scalability of the WS mechanism in solving large instances. The results demonstrate the high performance of the PGAS data structure and WS, even in highly-demanding scenarios.

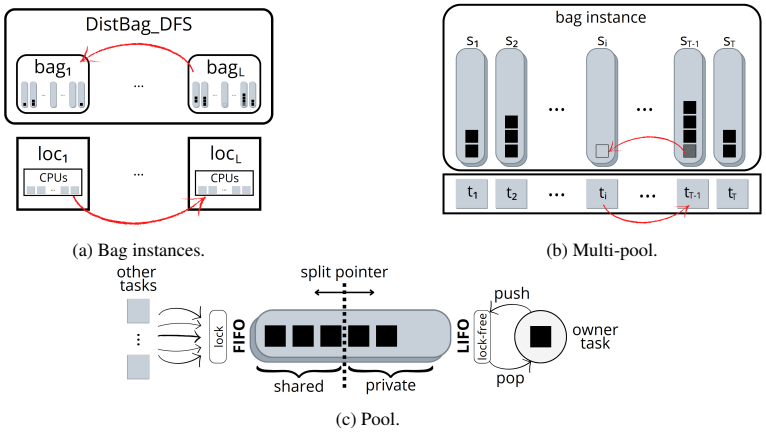
In the following, we first give a short background and some related work. Then, we present `DistBag_DFS` and associated WS. After a performance evaluation, we outline some conclusions and future directions.

2. Background and related work

Tree-based search algorithms are powerful techniques that have the ability to efficiently explore solution spaces. The exploration is often guided by the principles of backtracking and B&B involving a systematic search through the decision tree, incrementally building and evaluating potential solutions. Those algorithms generally involve highly irregular and unpredictable search trees, and explore the tree in a DFS manner. Implemented using a last-in, first-out (LIFO) stack to store generated but not yet visited nodes, DFS is favored in combinatorial algorithms due to its memory efficiency, especially when compared to memory-intensive strategies such as breadth-first.

Tree-search algorithms are inherently recursive, making them well-suited for parallelization. The most general and frequently used approach is the parallel tree exploration, which consists in exploring several disjoint subspaces in parallel [6]. In asynchronous mode, adopted in this paper, the search processes communicate in an unpredictable way making non trivial the sharing of knowledge among workers. Therefore, defining an efficient data structure, to store the workload, and its associated management policy is highly crucial for performance.

In this work, the multi-pool strategy is used. Each worker manages its own pool of generated, but not yet evaluated, nodes and maintains them in a DFS order. This strategy requires a sophisticated communication model since it raises the issue of balancing the workload between multiple pools. A popular and provably efficient dynamic load balancing approach is the WS paradigm [7]. Under WS, each process usually maintains a double-ended queue (deque) of nodes. Each worker processes nodes from the tail of the deque and steals work items from the head of another deque when its pool is empty.



(a) Bag instances.

(b) Multi-pool.

(c) Pool.

Figure 1: Illustration of the `DistBag_DFS` components: (a) bag instances, (b) multi-pool, and (c) pool based on non-blocking split deque.

In this context, some data structures based on traditional programming environments have been proposed [8; 9]. The latter allow high-performance and often benefit from problem-specific optimizations. In contrast, several papers discuss scalable dynamic load balancing techniques for unbalanced tree-search using PGAS-based environments [10; 11; 12; 13]. Similarly, we introduced in [3] the `DistBag_DFS` distributed data structure designed for unbalanced tree-search at scale. The latter is implemented using Chapel and has been exploited for a generic parallel distributed tree-search. We demonstrated the competitiveness of our approach compared to OpenMP and MPI+X baselines, considering both intra- and inter-node performance, and productivity-awareness. This work extends [3] with a comprehensive description of the data structure and its WS mechanism while investigating its performance at scale.

3. The `DistBag_DFS` data structure and work stealing

3.1. Design.

Figure 1 illustrates the hierarchical structure of the data structure. First of all, `DistBag_DFS` maintains one *bag* instance (multi-pool) per locale, as shown in Figure 1a. In Chapel, a *locale* is a subset of the target architecture that can be used to control and reason about affinity for the sake of performance and scalability. For most target architectures, a locale is equivalent to a computer node. In that sense, this component of the data structure handles the inter-node level of parallelism. While the data structure is safe to use in a distributed manner, it provides a mean to obtain a privatized instance of it for maximized performance, and each locale always operates on its privatized instance.

As shown in Figure 1b, each bag instance contains multiple pools, called *segments*. More precisely, one segment is maintained per parallel task (T in the figure). Each task has a unique identifier $1, \ldots, T$, used to map it to a segment. The latter is used in the

`DistBag_DFS`'s insertion (resp. retrieval) procedure to specify the segment into (resp. from) which an element node gets inserted (resp. retrieved). This is required by the DFS, because when a node is evaluated, the entire subtree below it must be explored before another sibling node is processed. However, when children nodes are inserted into a different segment than the one from which the parent was taken, that necessary condition cannot be ensured. In addition, it is worth to mention that while each segment locally guarantees a DFS order, the multi-pool do not.

The implementation of segments is based on non-blocking split deques [14]. Under this scheme, each segment is logically split into a "shared" and "private" region using an atomic *split pointer*, as shown in Figure 1c. This scheme allows lock-free local access to the private portion of the deque and copy-free transfer of work between the shared and private portions. Work transfer is done by moving the split pointer in either directions using appropriate operators. The other tasks access the shared region for load balancing, and synchronize themselves using an atomic lock. Segments are dynamic-sized and have an initial capacity of 1,024 elements. When a segment is full, we extend its capacity by a power of two.

Finally, `DistBag_DFS` is equipped with a WS mechanism, as shown in Figure 1 by the red arrows. This mechanism intervenes at the levels of both segments (intra-node) and bag instances (inter-node). The latter is locality-aware and, depending on the state of the data structure, different scenarios may occur:

- When the private region of a segment is empty, the associated task will first try to steal a work item locally. It iterates randomly over all the eligible segments from the same bag instance, and only one node is stolen, if applicable. Indeed, the WS is performed at the head of the deque and thanks to DFS the stolen node is the shallowest one in the search tree. Thus, it is expected to generate a large number of children nodes.

- When the local WS attempts fail, a global WS is triggered. Similarly, it iterates over all the eligible bag instances, and then over all the eligible segments on it. Since remote accesses generate high overheads, multiple nodes are stolen at once. In addition, only one global steal attempt is allowed per bag instance, meaning that when a task is performing a global WS, the other tasks can not.

- When all the local and global WS attempts fail, nothing is returned and one can be sure that the whole `DistBag_DFS` is either empty or few elements remain inside.

3.2. Implementation aspects.

`DistBag_DFS` is designed to be as simple as possible for the user. It implements a multi-pool and encapsulates a load balancing mechanism transparently to the user, thanks to the PGAS paradigm. Indeed, the latter includes a unified global address space, implicit communications, better data locality, and expressive memory models. Furthermore, the data structure is generic and can contain any types, even user-defined or external ones.

Regarding the user's interface, the data structure is composed of a set of two initialization variables (`eltType` and `targetLocales`) and seven methods. The operators

`add`, `addBulk`, and `remove` allow the user to insert an element, insert elements in bulk, and remove an element from a given segment, respectively. Each of these procedures applies to the bag instance of the locale it is called from. In addition, the data structure contains four global methods that apply to the whole `DistBag_DFS`. To avoid holding onto locks, we take a snapshot approach, increasing memory consumption but also increasing parallelism. This allows other concurrent, even mutating, operations while iterating, but opens the possibility to iterating over duplicated or missing elements from concurrent operations. These methods are `clear`, `these`, `contains`, and `getSize`, and allow the user to clear `DistBag_DFS`, iterate over it, search for a specific element in it, and get its global size, respectively. Finally, the data structure owns some configuration parameters that can be used to fine-tune its capacity and WS. The latter allow the user to set the initial and maximum capacities of the segments and also the minimum number of elements a segment must have to become eligible to be stolen from. This may be useful if some segments contain less elements than others and should not be stolen from.

One aspect that requires further investigation is the required "task id" in the insertion/retrieval operations of the data structure. In the current version of `DistBag_DFS`, it is required to ensure the local DFS ordering. We could implement an automated way to deal with this index, but *a priori* Chapel's design intentionally avoids supporting a standard language-level way to query a task's id. One can still exploit the internal `chpl_task_ID_t` opaque type, that refers to the task id that the runtime uses, but this could raise portability issues since Chapel includes different runtime tasking options, and the support is not guaranteed to continue across future versions of the language.

4. Performance evaluation

4.1. Experimental protocol.

We first evaluate the performance of the WS mechanism in a shared-memory setting on the UTS benchmark [15]. The latter is widely used to evaluate dynamic load balancing of fine-grained applications and is solved using backtracking. Moreover, we assess the scalability of our implementation in a large scale distributed-memory setting. As test-cases, large Permutation Flowshop Scheduling Problem (PFSP) instances proposed by E. Taillard in [16] are solved using the B&B technique. They consist in finding an optimal processing order for n jobs on m machines, such that the completion time of the last job on the last machine (makespan) is minimized. The so-called two-machine bound [17] and the dynamic *minBranch* branching technique [18] are used.

The Luxembourg national petascale MeluXina - Cluster module is used for the experiments. Each computer node has 2 AMD EPYC Rome 7H12 64 cores @ 2.6 GHz CPUs and 512 GB of RAM. In addition, the nodes are interconnected *via* the InfiniBand HDR high-speed fabric and operate under Rocky Linux 8.7. Chapel 1.31.0 is used in a fine-tuned configuration environment, along with the gcc 11.3.0 back-end compiler.

4.2. Evaluation of work-stealing.

In this section, two synthetic UTS trees with different types, binomial and geometric, are solved. A binomial tree is an optimal adversary for load balancing strategies,

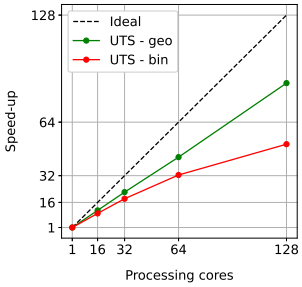


Figure 2: Speed-up achieved solving geometrical and binomial synthetic UTS trees, compared to a sequential version.

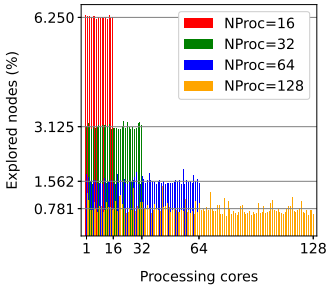


Figure 3: Percentage of explored nodes per processing cores solving the `UTS-bin` instance. The grey lines represent the ideal percentage, i.e., $100/NProc$.

Table 1: Summary of the instances used, along with some execution statistics.

Inst.	**Nb. of nodes** (10^6)	**Time (s)**	**nodes/s** (10^3)	**WS attempts (% success)**
`UTS-geo`	171.1	37.38	4,577	48,433 (99.0%)
`UTS-bin`	131.7	37.11	3,548	1,473,048 (96.8%)

since there is no advantage to be gained by choosing to move one node over another for WS: the expected work at all nodes is identical (i.e., at most two children nodes). In contrast, in a geometric tree the expected size of the subtree rooted at a node increases with proximity to the root. One can see on Figure 2 that for the best results, 68% of the ideal speed-up is achieved using 128 processing cores and solving the `UTS-geo` instance. This represents 40% more than the `UTS-bin` instance, which is directly related to the branching factor, as explained above.

Table 1 provides some execution statistics of the solved instances. In order to allow a fair comparison between instances, we make sure that the sequential times are approximately the same. One can see that for each instance, the percentage of WS attempts failed is less than 9%, which demonstrate the relevance of the WS. Moreover, Figure 3 shows the percentage of explored nodes per processing cores, solving the `UTS-bin` instance. One can see that for each experiment the total workload is almost evenly balanced among all the processing units, meaning that all the allocated resources are fully exploited.

These experiments show that our WS mechanism is able to achieve good performance, as well as a good workload distribution between all the allocated resources, even at low-granularity. The latter takes advantage of the fact that the shallowest nodes are stolen first, and that those nodes generally have a higher branching factor than the others. Nevertheless, it was observed that the performance may be impacted when it is not the case, like solving `UTS-bin`.

4.3. Performance at scale.

Figure 4 shows the speed-up reached solving the `ta056` PFSP instance up to 400 nodes, compared to a multi-core version. The latter exhibits 173×10^9 nodes and requires 1.26 node-hour. The experimental results revealed that up to 70% of the ideal

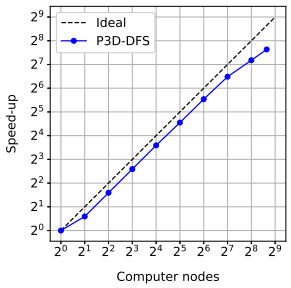


Figure 4: Speed-up achieved solving `ta056`, compared to a multi-core version.

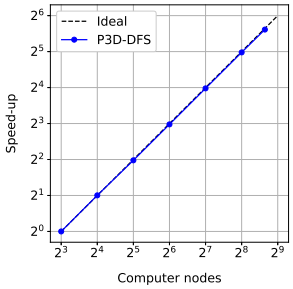


Figure 5: Relative speed-up achieved solving `ta057`.

speed-up can be achieved using up to 128 computer nodes, and around 50% using 400 nodes. In the latter experiment, 51,200 processing cores are used, and therefore as many segments are maintained in parallel. This leads to a large number of potentially remote communications, which can explain this limit in performance.

As a preliminary experiment towards the resolution of open instances, `ta057` is also solved. The latter exhibits a tree composed of $28{,}340 \times 10^9$ nodes, requires on average 220 node-hour, and was first solved to the optimality in 2022 [19], exploiting on average 384 GPUs during 1h11. For time limitation reasons, we consider the processing time on 8 nodes as the reference time for the speed-up computation. Figure 5 shows a high relative scalability, with 98% of the ideal speed-up reached using 400 nodes.

5. Conclusions and future works

We investigated a PGAS data structure and WS for the class of unbalanced tree-based algorithms, focusing on DFS. According to the experimental results, it is shown that the data structure and WS allow to achieve 68% of the linear speed-up on a fine-grain backtracking application in single-node setting. Furthermore, large scale experiments revealed that 50% of strong scaling efficiency is achieved using 400 computer nodes (51,200 processing cores) solving large PFSP instances using the B&B technique.

This work opens the road toward the resolution of open combinatorial optimization problems instances. To that end, we plan to extend our approach with a fault-tolerance mechanism in order to face Mean-Time-Between-Failure that are ever smaller.

Acknowledgments.

The experiments presented in this paper have been performed on the Luxembourg national supercomputer MeluXina. The authors gratefully acknowledge the LuxProvide teams for their expert support. This work is supported by the Agence Nationale de la Recherche (ref. ANR-22-CE46-0011) and the Luxembourg National Research Fund (ref. INTER/ANR/22/17133848), under the UltraBO project.

References

[1] A. Grama and V. Kumar, "Parallel Search Algorithms for Discrete Optimization Problems," *ORSA Journal on Computing*, vol. 7, no. 4, pp. 365–385, 1995.

[2] W. Zhang, "Branch-and-Bound Search Algorithms and Their Computational Complexity," Defence Technical Information Center, Tech. Rep., 1996.

[3] G. Helbecque, J. Gmys, N. Melab, and et al., "Parallel distributed productivity-aware tree-search using Chapel," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 27, p. e7874, 2023.

[4] B. L. Chamberlain, E. Ronaghan, B. Albrecht, and et al., "Chapel Comes of Age : Making Scalable Programming Productive," 2018.

[5] G. Helbecque, J. Gmys, T. Carneiro, and et al., "Productivity- and Performance-aware Parallel Distributed Depth-First Search," Oct. 2023. [Online]. Available: https://github.com/Guillaume-Helbecque/P3D-DFS

[6] B. Gendron and T. G. Crainic, "Parallel Branch-and-Branch Algorithms: Survey and Synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[7] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, 1999.

[8] J. Gmys, R. Leroy, M. Mezmaz, and et al., "Work stealing with private integer–vector–matrix data structure for multi-core branch-and-bound algorithms," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 18, pp. 4463–4484, 2016.

[9] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Júnior, and et al., "GPU-accelerated backtracking using CUDA Dynamic Parallelism," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4374, 2018.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, and et al., "Scalable Work Stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Association for Computing Machinery, 2009.

[11] R. Machado, C. Lojewski, S. Abreu, and et al., "Unbalanced tree search on a manycore system using the GPI programming model," *Computer Science - Research and Development*, vol. 26, no. 3, pp. 229–236, 2011.

[12] G. Cong, S. Kodali, S. Krishnamoorthy, and et al., "Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing," in *37th International Conference on Parallel Processing*, 2008, pp. 536–545.

[13] S. Olivier and J. Prins, "Scalable Dynamic Load Balancing Using UPC," in *37th International Conference on Parallel Processing*, 2008, pp. 123–131.

[14] T. van Dijk and J. C. van de Pol, "Lace: Non-blocking Split Deque for Work-Stealing," in *Euro-Par 2014: Parallel Processing Workshops*, 2014, pp. 206–217.

[15] S. Olivier, J. Huan, J. Liu, and et al., "UTS: An Unbalanced Tree Search Benchmark," in *Languages and Compilers for Parallel Computing*, 2007, pp. 235–250.

[16] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[17] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem," *Operations Research*, vol. 26, no. 1, pp. 53–67, 1978.

[18] J. Gmys, M. Mezmaz, N. Melab, and et al., "A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem," *European Journal of Operational Research*, vol. 284, no. 3, pp. 814–833, 2020.

[19] J. Gmys, "Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers," *INFORMS Journal on Computing*, vol. 34, no. 5, pp. 2502–2522, 2022.