Towards a scalable load balancing for productivity-aware tree-search

Guillaume Helbecque[†], Jan Gmys[†], Tiago Carneiro[*], Nouredine Melab[†], Pascal Bouvry[*]
[†]Université de Lille, CNRS/CRIStAL UMR 9189, Centre Inria de l'Université de Lille, France
[*]University of Luxembourg, DCS-FSTM/SnT, Luxembourg
guillaume.helbecque@univ-lille.fr

ABSTRACT

In the context of exascale programming, we investigate a parallel distributed productivity-aware tree-search for exact optimization in Chapel. To this end, we present the `DistBag-DFS` distributed data structure, which is our revisited version of the Chapel's `DistBag` data structure for depth-first search. The latter implements a distributed multi-pool, as well as an underlying locality-aware load balancing mechanism. Extensive experiments on large unbalanced tree-based problems are performed, and the competitiveness of our approach is reported against MPI+X implementations in terms of performance. For our best results, we achieve 94% of the ideal speed-up, using up to 64 computer nodes (8192 cores).

KEYWORDS

Distributed programming, Load balancing, Depth-first search, Productivity-awareness, Chapel.

1 INTRODUCTION

In the context of exascale programming[1], we investigate a parallel distributed productivity-aware tree-search for exact optimization in Chapel. The focus is put on the backtracking/Branch-and-Bound (B&B) algorithms, which explore implicitly constructed trees. The efficient parallel design and implementation of these methods is challenging because the pattern of computation and communication captured by these latter is highly irregular. In this work, we consider the commonly used parallel tree exploration model, which consists in exploring several disjoint subspaces in parallel (*i.e.* multiple tree-searches are performed in parallel). Depth-First Search (DFS) is used as it provides a good memory efficiency, unlike the other search strategies (*e.g.* breadth-first). It is usually implemented as a stack, operating in a last-in first-out (LIFO) order. In the asynchronous mode adopted in this work, the search processes communicate in an unpredictable manner and the sharing of knowledge among workers becomes non-trivial. Therefore, defining a data structure to store the work pool and an associated management policy is highly important for performance. In this paper, we investigate a multi-pool strategy, in which each search process maintains a work pool, and dynamic load balancing is required to balance workload. For that purpose, we consider the Work Stealing (WS) paradigm, in which idle processes steal work items from another work pool.

To the best of our knowledge, the few existing works in the context of productivity-aware parallel exact optimization in Chapel are from some of the co-authors of this paper. In [1], we proposed an incremental parallel PGAS-based backtracking algorithm. Reported results on the N-queens problem show that the distributed search achieves up to 80% of the scalability of its MPI+OpenMP counterpart. The approach is based on the high-productivity iterator features of Chapel. In addition, we propose in [2] an extension of the previous algorithm to a more difficult problem, which is the resolution of large Permutation Flowshop Scheduling Problem (PFSP) instances. The Chapel-based search presents performance equivalent to MPI+pthreads for its best results on 1024 cores and reaches up to 84% of the linear speed-up.

Both works mentioned above used data structure specific to permutation problems. The search strategy consist in non-recursive backtracking that does not use dynamic data structures. Unlike these papers, we propose in this work a more generic approach based on the task-parallel features of Chapel. The implementation relies on the `DistBag-DFS` distributed data structure, which provides a distributed multi-pool as well as an underlying locality-aware WS mechanism. The latter is our revisited version of the Chapel's `DistBag` data structure for DFS.

2 DESIGN AND IMPLEMENTATION

2.1 The `DistBag-DFS` distributed data structure

2.1.1 Limitations of the `DistBag` *data structure.* The Chapel's `DistributedBag` package module[2] implements a parallel-safe distributed multi-pool implementation, called `DistBag`. This data structure is unordered and incorporates a WS mechanism to balance workload across multiple locales, transparently to the user. While the bag is safe to use in a distributed manner, each node always operates on its privatized instance. `DistBag` can contain either predefined-Chapel types, user-defined types or external ones (*e.g.* C structures). Internally, the `DistBag` container is composed of multiple pools (called *segments*) implemented as unrolled linked-lists. In the following, we refer to *pool* and *segment* without distinction. By default, there are as many segments per locale as threads. To ensure correctness, operations on pools (insertion, retrieval, *etc.*) are lock-protected.

Preliminary experiments and source-code inspection revealed that pools do not necessarily operate in LIFO-order and, in addition, they are not explicitly mapped onto threads. In fact, multiple pools are maintained to reduce lock contention, but threads remove and insert elements from any (not necessarily the same) unlocked pool. Although this may be acceptable for some applications, this behaviour is not suitable for parallel DFS. In DFS, 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 pool than the one from which the parent was taken, that necessary condition cannot be ensured. As a

[1]Top500 ranking (November 2022): https://www.top500.org/lists/top500/2022/11/.

[2]The `DistributedBag` module (Chapel 1.29.0): https://chapel-lang.org/docs/modules/packages/DistributedBag.html.

direct consequence, memory requirements may rapidly grow out of control. This has been observed in our preliminary experiments, and is consistent with the Chapel documentation[2] which states that important memory consumption may appear in single-locale experiments.

The parallel distributed aspect of the `DistBag` data structure, as well as the underlying WS mechanism make it particularly attractive in the context of parallel distributed productivity-aware tree-search. However, as explained above, its segments' scheduling policy does not allow us to use it for parallel DFS, nor to have any control over the order of insertion/retrieval of nodes. This motivates our redefinition of the data structure and underlying mechanisms.

2.1.2 Our revisited version for DFS. In this work, we revisit the `DistBag` data structure in two different ways: (1) we redesign the work pools implementation, and thus provide a new synchronization scheme using non-blocking split deques, and (2) we redefine the underlying WS mechanism. The resulting data structure is refered to as `DistBag-DFS` and its implementation is publicly available [3].

Non-blocking split deques. We extend the segments' scheduling policy to support insertion and retrieval from both ends, effectively supporting both first-in first-out (FIFO) and LIFO orders, like a deque. This allows threads to perform the local exploration of nodes in a DFS, whereas the oldest (*i.e.* shallowest) nodes are stolen in WS operation.

WS operations require synchronization between *thieves* and *victim* threads. In the `DistBag` data structure, segments are lock-protected using one atomic synchronization variable per segment, *i.e.* when a thread operates on a segment, the latter is locked until the end of the operation. In `DistBag-DFS`, we redesign this synchronization scheme using non-blocking split deques [4, 5]. This method consists in splitting deques (segments) into a *public* and a *private* portion using an atomic *split pointer*, as shown in Figure 1. Under this scheme, all processes push new tasks on the tail of the queue and pop tasks from the tail to get the next task to execute in LIFO order, while WS is done at the head in a FIFO manner. This synchronization scheme allows lock-free local access to the private portion of the deque and copy-free transfer of work between the public and private portions of the deque. Work transfer is done by moving the split pointer in either directions. Thieves synchronize using a lock and the local process only needs to take the lock when transferring work from a portion to the other of the deque. Some authors demonstrate the efficiency of such synchronization scheme at scale using several benchmarks, including the Unbalanced Tree-Search benchmark (UTS) [5]. While each segment acts like a deque, the overall `DistBag-DFS` semantics does not guarantee a strict deque semantics, in contrast to the Chapel's `DistDeque` data structure[3].

Work stealing mechanism. When a thread's segment becomes empty during execution, the retrieval operation of the bag transparently becomes a WS operation, *i.e.* an attempt is made to steal work items from another segment. The latter is thus embedded in the retrieval operation (called `remove`) of the `DistBag-DFS` data structure, which is illustrated in Figure 2. For readability, only one bag

[3]The `DistributedDeque` module (Chapel 1.29.0):https://chapel-lang.org/docs/modules/packages/DistributedDeque.html.

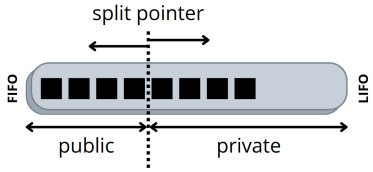


Figure 1: Simplified view of a non-blocking split deque.

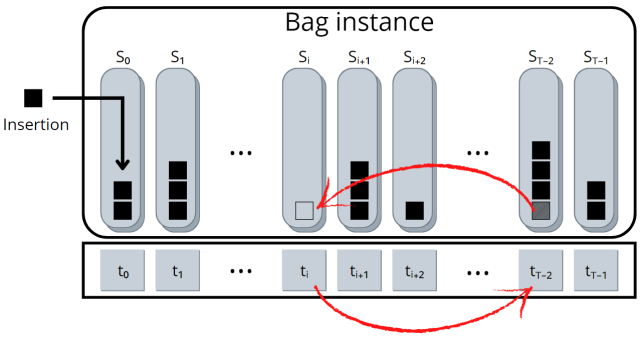


Figure 2: Illustration of the `DistBag-DFS` data structure.

instance is shown (one per Chapel's locale in practice). In this figure, we assume that `DistBag-DFS` is used by T threads (per locale) and that bag instances are initialized with T segments (this is the default behavior). Each thread has a unique identifier $0, \ldots, T-1$, mapping threads to segments. The identifier is used in the bag's insertion (resp. retrieval) procedure, to specify the segment into (resp. from) which an element node gets inserted (resp. retrieved). When a segment s_i is empty, thread t_i first tries to steal workload from another thread's segment of its locale. If a victim thread fulfills the stealing request (t_{T-2} in the figure), t_i gets its lock and steals work (this case if referred to as `SUCCESS` in the `remove` procedure). However, if all local attempts fail, the thief tries a global steal. It means that a victim bag instance is chosen, and its segments are visited. Since global WS operations generate high parallel overheads, the thief is expected to steal more work items than it needs. Then, these extra work items provide work for potential local WS. Thus, when a thread performs a global steal, other global steal requests issued by threads in its locale are ignored (`FAST_FAIL`). Finally, the `remove` operation fails if both local and global steals fail (`FAIL`). The random victim selection strategy is used for both local and global WS, and half of the shared region of the victim segment is stolen.

2.2 Parallel distributed DFS

In this work, `DistBag-DFS` is employed to implement a generic parallel distributed DFS (*i.e.* non problem-specific). The problem as well as its data are passed to the algorithm via a `problem` class instantiation. Algorithm 1 shows a simplified view of the implementation in Chapel. Termination detection as well as the sharing of global knowledge are omitted to favor readability. First of all, the bag is initialized and the root node is created and inserted in it (lines 1-3). Note that `DistBag-DFS` here contains user-defined `Node` data type. In addition, we arbitrarily chose to insert the root in task 0's segment. Then, nested concurrent tasks are created using

the two `coforall` statements, and synchronized before the search (lines 5-7). Note that tasks are distributed using the on clause. At this stage, the parallel exploration starts, and each task is indefinitely performing:

(1) the retrieval of a parent node into the bag (lines 9-10). As explain in Section 2.1.2, this procedure also invokes a WS mechanism if needed.
(2) the checking of the termination detection (line 11). In few words, it consist of a bi-level mechanism, where the local tasks' statuses are first checked, and then globally, if needed.
(3) the sequential decomposition of the parent node into a set (possibly empty) of children nodes (line 13). This function contains all the backtracking/B&B logic (*e.g.* branching, bounding, pruning).
(4) the insertion of the children nodes in the bag (line 14);
(5) the sharing of global knowledge if any, *e.g.* the best solution found so far.

Note that the only difference between `DistBag` and `DistBag-DFS` from a user point of view is that the calling task's id is now explicitly gave to the insertion and retrieval procedures to ensure correctness of the DFS, as discussed in Section 2.1.2. In the following, this algorithm is referred to as P3D-DFS [3] (which stands for Performance- and Productivity-aware Parallel Distributed Depth-First Search).

Algorithm 1: Simplified view of our parallel distributed productivity-aware DFS in Chapel

```
1  var bag = new DistBag_DFS(Node, Locales);
2  var root = new Node(problem);
3  bag.add(root, 0);
4
5  coforall loc in Locales do on loc {
6    coforall taskId in 0..#here.maxTaskPar {
7      allLocalesBarrier.barrier();
8      while true {
9        var (hasWork, parent): (int, Node);
10       (hasWork, parent) = bag.remove(taskId);
11       /* Check termination condition */
12       var children: list(Node);
13       children = problem.decompose(parent);
14       bag.addBulk(children, taskId);
15       /* Share the global knowledge */
16     }
17   }
18 }
```

3 EXPERIMENTAL EVALUATION

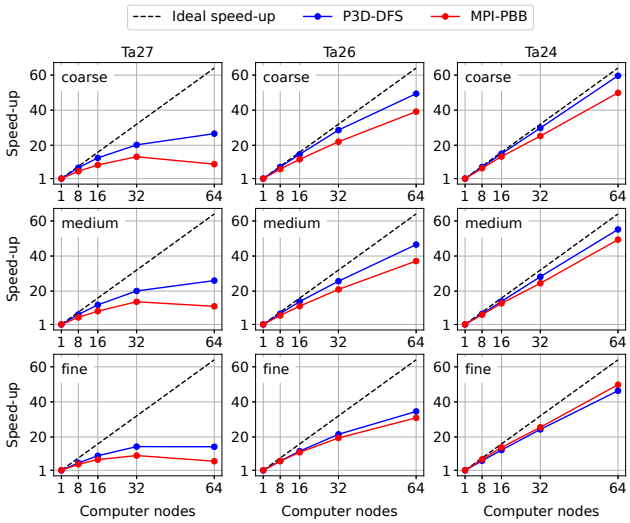
In this section, our Chapel-based parallel distributed tree-search is evaluated and compared to MPI+X baseline implementations in terms of performance and scalability. As test-cases, we consider the B&B method and its application to the PFSP, and the UTS benchmark. PFSP consists in finding an optimal processing order for n jobs on m machines, such that the completion time of the last job on the last machine is minimized, while UTS consists in counting the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size and imbalance. More particularly, some of the well-known PFSP Taillard's instances [6] are solved (*e.g.* `Ta27`, `Ta26` and `Ta24`) as well as synthetic UTS trees. Different granularities (*e.g.* B&B lower bound functions) are also considered. The experiments are conducted on a cluster equipped with 2 AMD Epyc ROME 7H12 @ 2.6 GHz processors, including a total of 128 cores and 256 GB RAM per computer nodes. All nodes are interconnected through a Fast InfiniBand HDR100 network, and operate under Red Hat Enterprise Linux 8.3, 64 bits. Finally, Chapel 1.29.0 is used with a well-tuned environment for execution. The following implementations are experimented:

- **P3D-DFS**: Chapel implementation of backtracking/B&B [3], described in Section 2.2. It relies on the `DistBag-DFS` data structure and is instantiated on the B&B method and its application to the PFSP, and the UTS benchmark.
- **MPI-PBB**: MPI+pthread implementation of B&B [7]. It is a Master-Worker approach using an interval-based encoding of work units and the IVM data structure [8] (specifically designed for permutation-based problems) for the implementation of DFS. Each MPI worker consists of multiple worker threads performing local WS operations for load balancing on the intra-node level. Inter-node work load balancing is performed by the intermediate of the centralized coordinator process.
- **MPI-PUTS**: two-sided MPI+MPI implementation of UTS [9, 10]. In this approach, each MPI process maintains a private deque, and dynamic WS is done using an explicit polling *progress engine*. A working process must periodically invoke the progress engine in order to observe and service any incoming steal requests.

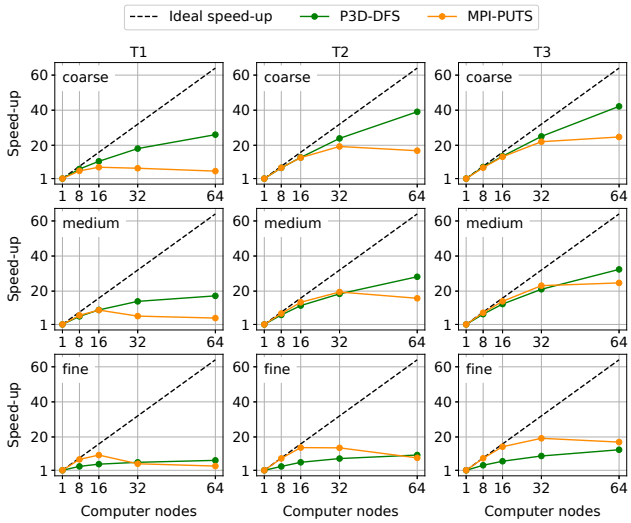
Figure 3a shows the absolute speed-up reached by P3D-DFS and MPI-PBB in distributed setting, using up to 64 computer nodes (8192 cores). First of all, we can see that P3D-DFS outperforms its counterpart in almost all cases. Indeed, the presence of a master process in MPI-PBB makes termination detection trivial, however, it becomes a sequential bottleneck at large scale. Nevertheless, solving the largest `Ta24` instance with the finest granularity, MPI-PBB achieves better results than P3D-DFS. This can be explained by the fact that in this experiment, the management overheads of the data structure and load balancing mechanism are particularly visible, and thus `DistBag-DFS` seems to be less efficient than its counterpart. For its best results, P3D-DFS achieves 94% of the ideal speed-up. Similar experiments on the UTS are shown in Figure 3b. On the one hand, one can see that for medium- and coarse-grained experiments, P3D-DFS outperforms its counterpart. Limited scalability of MPI-PUTS can be explained by the fact that the implementation does not have a locality-aware WS mechanism, as P3D-DFS does, meaning that a thief thread steals a local/remote victim one, without distinction. On the other hand, P3D-DFS is outperformed at fine grain. In fact, this is explained by the poor scalability of the latter at the intra-node parallel level, that leads to limited absolute speed-up. This is consistent with the observation done in Figure 3a. For its best results, P3D-DFS reached up to 66% of the linear speed-up.

4 DISCUSSION AND FUTURE PERSPECTIVES

In this section, we discuss the productivity-awareness of P3D-DFS, as well as the future perspectives. Some authors characterize HPC



(a) P3D-DFS *vs.* MPI-PBB on B&B applied to PFSP.



(b) P3D-DFS *vs.* MPI-PUTS on UTS.

Figure 3: Absolute speed-up achieved by P3D-DFS and MPI+X baseline implementations on large unbalanced tree-based problems, considering different granularities (fine, medium, coarse). For each figure, most coarse-grained is top-right, most fine-grained is bottom-left. Computer nodes vary from 1 to 64.

productivity as a trade-off between performance and programming effort [11]. Regarding the latter aspect, Chapel's global view of control flow and data structures make it straightforward to design and implement a distributed backtracking/B&B based on a sequential version. This requires few more lines of code in contrast to MPI+X, where we have to deal explicitly with inter-node communications or command line launch parameters for example. On the other hand, P3D-DFS is generic and general. It is instantiable on numerous optimization problems. This is facilitated by the object-oriented programming supported by Chapel, as well as the generic `DistBag-DFS` data structure. In addition, as shown in Section 2.2, the latter hides to the user the multi-pool implementation as well as the underlying WS mechanism, which has the benefit of making the parallel distributed implementation readable and easy to maintain. Nevertheless, in contrast with `DistBag`, we have seen in Section 2.2 that our revisited version introduces a new argument (the calling task's id), and it is not ideal since the burden belongs to the user. *A priori*, Chapel's design intentionally avoids supporting a standard language-level way to query a task's id. Nevertheless, it could be possible to exploit the internal `chpl_task_ID_t` opaque type, that refers to the task ids that the runtime uses. This will raise portability issues since Chapel includes different runtime tasking options (qthreads and fifo), and the support is not guaranteed to continue across future versions of the language.

From a performance point-of-view, we have seen in Section 3 that `DistBag-DFS` suffers from high management overheads, especially visible on fine-grained applications. A future work is thus to optimize its low-level mechanisms, and to benchmark and compare its performance with well-established Chapel or C data structures. In addition, we plan to investigate the performance and scalability of P3D-DFS at a larger scale, and consider other optimization problems (*e.g.* Knapsack problems). Finally, we hope that Chapel will integrate our efforts over the coming months for the benefit of the community.

5 CONCLUSION

In this paper, we presented the design and implementation of a productivity-aware parallel distributed DFS in Chapel, namely P3D-DFS. The latter is based on the `DistBag-DFS` distributed data structure, which is our revisited version of the Chapel's `DistBag` data structure for DFS. It implements a distributed multi-pool, as well as an underlying locality-aware WS mechanism to balance workload. Reported results on large unbalanced tree-based problems revealed that P3D-DFS is competitive against MPI+X baseline implementations for coarse-grained applications. More precisely, it achieves up to 94% and 66% of the ideal speed-up using 64 computer nodes (8192 cores), on the B&B applied to PFSP and the UTS, respectively. Nevertheless, we note that `DistBag-DFS` seems to suffer from high management overheads, especially visible at fine grain, and we plan to further investigate its design and implementation.

ACKNOWLEDGEMENT

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg. Moreover, the code has been developed using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations. 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] T. Carneiro and N. Melab. An Incremental Parallel PGAS-based Tree Search Algorithm. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 19–26, 2019. URL https://doi.org/10.1109/HPCS48598.2019.9188106.

[2] T. Carneiro, J. Gmys, N. Melab, and D. Tuyttens. Towards ultra-scale Branch-and-Bound using a high-productivity language. *Future Generation Computer Systems*, 105:196–209, 2020. URL https://doi.org/10.1016/j.future.2019.11.011.

[3] G. Helbecque, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. Productivity- and Performance-aware Parallel Distributed Depth-First Search (P3D-DFS), 2023. URL https://doi.org/10.5281/zenodo.7674860.

[4] T. van Dijk and J. C. van de Pol. Lace: Non-blocking Split Deque for Work-Stealing. In *Euro-Par 2014: Parallel Processing Workshops*, pages 206–217. Lecture Notes in Computer Science, vol 8806. Springer, Cham., 2014. URL https://doi.org/10.1007/978-3-319-14313-2_18.

[5] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, New York, USA, 2009. Association for Computing Machinery. URL https://doi.org/10.1145/1654059.1654113.

[6] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. URL https://doi.org/10.1016/0377-2217(93)90182-M.

[7] J. Gmys. Parallel Branch-and-Bound for permutation-based optimization, 2023. URL https://doi.org/10.5281/zenodo.7674826.

[8] J. Gmys, R. Leroy, M. Mezmaz, N. Melab, and D. Tuyttens. Work stealing with private integer–vector–matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience*, 28(18):4463–4484, 2016. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3771.

[9] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007. URL https://doi.org/10.1109/IPDPS.2007.370581.

[10] J. Dinan and S. Olivier. The Unbalanced Tree-Search benchmark, 2022. URL https://doi.org/10.5281/zenodo.7328332.

[11] K. Kennedy, C. Koelbel, and R. Schreiber. Defining and Measuring the Productivity of Programming Languages. *The International Journal of High Performance Computing Applications*, 18(4):441–448, 2004. URL https://doi.org/10.1177/1094342004048537.