Parallel Distributed Productivity-aware Tree-Search using Chapel

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

[a] *Université de Lille, CNRS/CRIStAL UMR 9189, Centre Inria de l'Université de Lille, France*
[b] *Université du Luxembourg, FSTM, Luxembourg*
[c] *Université du Luxembourg, DCS-FSTM/SnT, Luxembourg*

Abstract

With the recent arrival of the exascale era, modern supercomputers are increasingly big making their programming much more complex. In addition to performance, software productivity is a major concern to choose a programming language, such as Chapel, designed for exascale computing. In this paper, we investigate the design of a parallel distributed tree-search algorithm, namely P3D-DFS, and its implementation using Chapel. The design is based on the Chapel's `DistBag` data structure, revisited by: (1) redefining the data structure for Depth-First tree-Search (DFS), henceforth renamed `DistBag-DFS`; (2) redesigning the underlying load balancing mechanism. In addition, we propose two instantiations of P3D-DFS considering the Branch-and-Bound (B&B) and Unbalanced Tree Search (UTS) algorithms. In order to evaluate how much performance is traded for productivity, we compare the Chapel-based implementations of B&B and UTS to their best-known counterparts based on traditional OpenMP (intra-node) and MPI+X (inter-node). For experimental validation using 4096 processing cores, we consider the permutation flow-shop scheduling problem for B&B and synthetic literature benchmarks for UTS. The reported results show that P3D-DFS competes with its OpenMP baselines for coarser-grained shared-memory scenarios, and with its MPI+X counterparts for distributed-memory settings, considering both performance and productivity-awareness. In the context of this work, this makes Chapel an alternative to OpenMP/MPI+X for exascale programming.

Keywords: Distributed programming, Productivity-awareness, Chapel, Branch-and-Bound, Unbalanced Tree-Search

1. Introduction

On June 2022, one can consider that we officially entered the exascale *era* as the 59^{th} edition of the Top500 international ranking[1] revealed the Frontier system to be the first true exascale supercomputer at Oak Ridge National Laboratory. According to Top500, modern supercomputers are composed of thousands of processing nodes, allowing high-level inter-node parallelism, each of them including hundreds of cores coupled with accelerators, supplying many-core low-level intra-node parallelism. Such a complex hierarchical organization supplying multi-level parallelism in heterogeneous resources (multi-core processors, accelerators, *etc.*) makes exascale programming much more challenging. The problem is that there is no standard language for exascale programming. Even if the exascale *era* is already there as announced several years ago, there is still a debate between two schools of thought: evolutionary and revolutionary. The first one defends the adaptation to exascale of traditional bare-metal programming environments (*e.g.* OpenMP, MPI+X) while the second one is disruptive suggesting to design new ones (*e.g.* Chapel).

In this paper, we contribute to this debate focusing on the intra- and inter-node parallel levels. We investigate representative programming languages/libraries of each approach: OpenMP (intra-node level) and MPI+X (inter-node or both level(s)) for the evolutionary approach, and Chapel for the other one. OpenMP and MPI are standards well known to be efficient for parallel shared- and distributed-memory programming, respectively. Chapel is a productivity-aware language, based on Partitioned Global Address Space (PGAS), that unifies both intra- and inter-node parallel levels. The choice of Chapel is motivated by the fact that it is an open-source language representative of the PGAS paradigm, having an active community[2] and maintained by a well-established High-Performance Computing (HPC) builder (HPE/Cray). As a basis of our study, we consider parallel distributed tree-search with a focus on backtracking/Branch-and-Bound (B&B) algorithms. The efficient parallel design and implementation of these latter is challenging, mainly because the pattern of computation and communication captured by these methods (named a "computational dwarfs" [1]) is highly irregular. For comparison purpose, as test-cases we consider two different unbalanced tree-search applications: B&B applied to the Permutation Flow-shop Scheduling Problem (PFSP) [], and the Unbalanced Tree-Search benchmark (UTS) []. The research question we address in this paper is the following: is Chapel (representing the revolutionary approach) representing an alternative to OpenMP and MPI+X (representing the evolutionary approach) at the intra-

[1] Top500 (Edition of June 2022): `https://www.top500.org/lists/2022/06/`

[2] *e.g.* Chapel Implementers and Users Workshop (CHIUW), `https://chapel-lang.org/CHIUW.html`

and inter-node levels, in the context of parallel distributed tree-search targeting exascale systems?

The major contributions of this paper are the following:

- We revisit the design and implementation of the `DistBag` distributed data structure supplied in the Chapel's `DistributedBag` module [2]. This data structure is attractive as it is a parallel-safe distributed multi-set implementation that employs an underlying dynamic load balancing mechanism among multiple nodes. However, we demonstrate its limitations in a Depth-First tree-Search (DFS) context. Therefore, we redesigned it (henceforth renamed `DistBag-DFS`) as follows: (1) we propose a new scheduling policy of its elements as well as a new synchronization mechanism using non-blocking split deques, and (2) we redefine the underlying dynamic load balancing mechanism.
- We propose a generic Chapel-based parallel distributed tree-search implementation, namely P3D-DFS, based on our `DistBag-DFS` distributed data structure and the underlying load balancing mechanism. P3D-DFS is then instantiated on B&B and UTS.
- The above Chapel-based implementations are experimented using up to 32 computer nodes (4096 cores). They are compared to their state-of-the-art counterparts using OpenMP and MPI+X. The reported results allow us to answer the research question stated above.

The rest of the paper is organized as the following. In Section 2, we first give a background on parallel distributed tree-search algorithms, followed by some related works. Then, the Chapel and baseline implementations are described in Section 3, and experimented, compared and discussed in Section 4. Finally, conclusions are drawn in Section 5 followed by some future directions.

2. Background and related Works

2.1. Parallel distributed tree-search

2.1.1. Sequential DFS

Both UTS and B&B explore implicitly constructed trees – implicit construction meaning that each node contains all information necessary to construct its children nodes. In UTS, a synthetic benchmark designed to evaluate the performance and ease of programming for parallel applications requiring dynamic load balancing, each node has a 20-byte descriptor which allows to generate a variable, but deterministic number of children. In B&B, an exact optimization algorithm, each tree node corresponds to a subproblem (the initial problem defined on a restricted domain) and children nodes are obtained by further restricting the search space. As B&B computes lower bounds on the optimal cost of each subproblem and prunes (eliminates) subproblems that cannot lead to an improvement of the best solution found so far, the number of children resulting from a node decomposition is variable. In both cases, the explored search trees are highly irregular and unpredictable.

The goal of the UTS benchmark is to count the total number of nodes in a tree, while B&B finds an optimal solution to an optimization problem and proves its optimality by complete enumeration. While the search order in UTS is irrelevant, it may have a strong impact on B&B's capability of finding optimal solutions. However, once B&B has found an optimal solution, the search order becomes irrelevant as well.

In this work, we consider DFS, meaning that all children of an internal node are explored before expanding the next sibling node. DFS is easily implemented by storing generated, but not yet visited nodes in a stack (last-in first-out, LIFO). DFS is frequently used in combinatorial B&B algorithms because memory requirements of other search strategies, such as breadth-first or best-first search, often become excessive. For the PFSP, B&B search trees containing up to 339×10^{12} nodes have been reported [3], meaning that breadth-first exploration would require peta-bytes of memory to store the tree.

Despite the differences between UTS and B&B, in the following we refer to the tree-search algorithm considered in this paper simply as B&B and use the B&B terminology (*e.g.* subproblems) – unless the distiction is meaningful.

2.1.2. Parallel tree-search

The parallelization of B&B is well-studied and different classifications have been proposed [4; 5]. The most general and most frequently used approach is the parallel tree exploration model (also called ”type 2” or ”high-level” parallelization). It consists in exploring several disjoint subspaces in parallel, meaning that multiple DFS, rooted in different tree nodes, are performed in parallel (the global search order is no longer depth-first, in general). This can be done either synchronously or asynchronously.

In asynchronous mode, adopted in this paper, 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.

2.1.3. Work pool management

We call work pool the data structure that is used to store generated but not yet evaluated subproblems/nodes. A node is usually implemented as a structure containing all information necessary for its evaluation. The role of the work pool management policy is mainly to allow the insertion/retrieval of subproblems. Moreover, the work pool may maintain subproblems in a certain order, facilitating the implementation of a search strategy. For instance, DFS corresponds to processing nodes in LIFO order and is therefore naturally implemented by a stack.

In parallel B&B, there are different strategies for implementing the work pool. One approach is to use a single centralized work pool, concurrently accessed by all workers to pick subproblems for branching/evaluation. In single-pool approaches synchronization between workers to gain exclusive access to the pool for the insertion/retrieval of nodes is inevitable. These concurrent accesses may create a bottleneck and memory contention, limiting the scalability of the approach.

Multi-pool approaches aim at solving this issue by using several pools. There are variants of multi-pool B&B algorithms. The most popular are collegial, grouped and mixed [4]. In the collegial variant, used in this work, each worker manages its own pool. The grouped approach uses multiple shared pools for groups of workers and the mixed variant combines both approaches using a hierarchy of pools. Multi-pool strategies alleviate the bottleneck problem that occurs in single-pool approaches, but raise the issue of balancing the workload between multiple pools. Also, termination detection may become challenging. Generally speaking, multi-pool approaches require more sophisticated communication models than single-pool ones.

While the latter implicitly balance the workload among workers, multi-pool approaches require explicit dynamic load balancing. A popular and provably efficient approach to dynamic load balancing is the Work Stealing (WS) paradigm [6]. Under WS, each process usually maintains a double-ended queue (deque) of subproblems/nodes. Each worker processes nodes from the tail of the deque and steals work units from the head of another worker's deque when its own work pool is empty.

2.2. The Chapel programming language

Chapel is a programming language arising from the DARPA High Productivity Computing System program [7]. It is designed to improve productivity in HPC, and this is achieved mainly by unifying both intra- and inter-node parallel levels supplying a *global view of control flow* as well as a *global view of data structures* [8].

Concerning the first aspect, the program is started with a single task, and parallelism is added though intra- or inter-node programming features. Regarding the global view of data structures, indexes are globally expressed, even in case the implementation of such data structures distributes them across several Chapel processes, denominated *locales*. Thus, Chapel is a language that applies the PGAS programming model [9]. Moreover, thanks to 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 remote memory.

2.3. Related Works

There exist few studies analyzing Chapel's performance along with other modern or conventional programming languages. In [10], the authors studied several optimizations such as reducing redundant memory allocation and manipulation. On the one hand, the reported results show that these optimizations increase Chapel's intra-node performance up to 7.9× on well-known application benchmarks such as LULESH and SSCA2. On the other hand, it is shown that with these optimizations Chapel can perform better than OpenMP. Similarly, a performance analysis of Chapel is proposed in [11] in comparison to OpenMP using multi- and many-core architectures. Several optimizations are proposed in different layers of the Chapel software stack, and their impact on performance is discussed. It is observed that base performance of Chapel ranges from 41% to 184% that of OpenMP. The optimizations show performance improvements ranging from 1.4× to 2× in Chapel. There exist also other related works in which the inter-node level is considered as well. For instance, in [12], Chapel- and MPI+OpenMP-based implementations are compared using Breadth-First Search (BFS) and PageRank applications. Chapel is proven 3.7× faster on average for BFS, but 1.3× slower for PageRank. In addition, we investigated in [13] the parallel generation of Mandelbrot's fractals on shared- and distributed-memory platforms. We demonstrate the competitiveness of the Chapel implementation against its OpenMP and MPI+X counterparts.

In this paper, we propose a similar investigation, but in the context of parallel distributed tree-based exact optimization. To the best of our knowledge, the few existing studies proposed in this context are from some of the co-authors of this paper. In [14], we proposed an incremental parallel PGAS-based backtracking algorithm. The methodology starts with a serial Chapel-based implementation, continues with an intra-node one and ends with an inter-node one. The implementations have been applied to the N-Queens problem, experimented and compared to their respective counterparts. The reported results show that the serial Chapel implementation is on average 7% slower than its C counterpart. At the intra-node level, the results show that if Chapel is well parameterized (*e.g.* using qthreads as a low-level threading library) it could be up to 13% faster than OpenMP for large problem instances. At the inter-node level, the distributed search in Chapel achieves up to 80% of the scalability compared to its MPI+OpenMP counterpart. Unlike the present paper, in [14] the focus is put on the iterator feature that supplies productivity in Chapel, considering the backtracking as a tree-search algorithm. In this paper, instead of iterators we investigate the task parallelism using a generic DFS algorithm, instantiated to the B&B method and the UTS benchmark. Finally, we proposed in [15] a Chapel design and implementation of distributed B&B for solving large PFSP instances. The results show that Chapel is much more expressive and up to 7.8× more productive than MPI+pthreads. In addition, 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.

On the other hand, some studies investigated productivity-aware design and implementation of tree-search considering other programming languages and the UTS benchmark. A Unified Parallel C (UPC) [16] implementation is presented and evaluated in [17]. The latter achieves better scaling and parallel efficiency in both shared- and distributed-memory settings than using MPI. In [18], the authors investigate the use of the Global Address Space Programming Interface (GPI) PGAS API [19] for UTS on a many-core system. According to the reported experimental results, GPI outperforms the MPI version by a maximum factor of 2.5 in terms of raw performance (number of nodes processed per second). In terms of speed-up and efficiency, the authors observed promising results.

As mentioned in the introduction, we just entered the exascale *era* making urgent the need to answer the software programming issue of modern supercomputers. Getting a feedback on the suitability and limits of conventional and new languages targeting exascale supercomputers becomes critical. In the same spirit as [10; 11; 12; 13; 14; 15], this paper aims at providing a useful data point to help practitioners gauge the difficult question of whether or not to invest time and effort into learning and using the Chapel programming language.

3. Implementations

In Section 3.1, we first present the `DistBag-DFS` distributed data structure used in our Chapel implementation. The latter is our revised version of the Chapel's `DistBag` data structure supplied in the `DistributedBag` module. Then, in Section 3.2, our Chapel-based parallel distributed DFS algorithm is presented. Finally, the baseline implementations considered for comparison purpose are described in Section 3.3.

3.1. Distributed data structure

3.1.1. The `DistBag` data structure

The `DistributedBag` module [2] developed by the Chapel's community provides a parallel-safe distributed multi-set implementation, called `DistBag`. This data structure is unordered and incorporates a WS mechanism that balances 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, 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 direct consequence, memory requirements may rapidly grow out of control (see Section 4.4.1). This observation 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 management mechanisms.

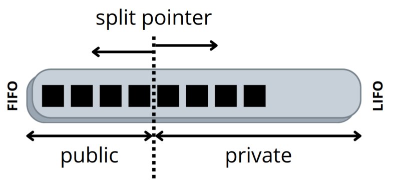

Figure 1: Simplified view of a non-blocking split deque.

3.1.2. `DistBag-DFS`: our redesigned version of `DistBag`

The `DistBag` data structure has been redefined in two different ways: (1) we propose a new scheduling policy of its elements as well as a new synchronization mechanism using non-blocking split deques, and (2) we redefine the underlying WS mechanism. The revised data structure is referred to as `DistBag-DFS` [20]. Experimental comparisons between `DistBag` and `DistBag-DFS` are done in Section 4.4.1.

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 and re-implement this synchronization scheme using non-blocking split deques [21; 22]. 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 deque 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. Work transfer is done by moving the split pointer in either directions using appropriate `release` or `acquire` operations. 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 UTS one [22].

Work stealing mechanism. When a thread's segment becomes empty during execution, the retrieval operation of the bag (called `remove`) 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 of the `DistBag-DFS` data structure, which is illustrated in Figure 2. For readability, only one bag 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 therefore bag instances are initialized with T segments. Each thread has

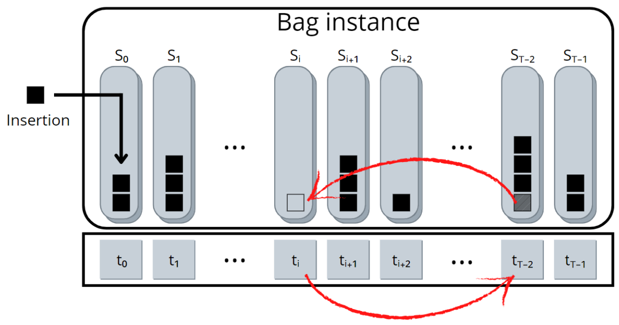

Figure 2: Illustration of the DistBag-DFS data structure.

a unique identifier $0, \dots, T-1$, 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 items. In that case, `remove` returns one of the stolen items and also the `SUCCESS` status of the operation. 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 needed. Then, these extra work items provide work for potential future 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.

3.2. Parallel distributed DFS in Chapel

Algorithm 1 shows the pseudo-code of our Chapel-based Productivity- and Performance-aware Parallel Distributed DFS algorithm, namely P3D-DFS [20]. The algorithm starts by creating the root node, using the problem-specific data/parameters (line 5). Here, `problem` represents an inherited class instance containing all the data related to the problem. Then, the `DistBag-DFS` distributed data structure (hereinafter referred to as *bag*) is initialized (line 6). The initialization procedure constructs on each specified locale (stored in the built-in `Locales` variable) a bag instance with as many segments as threads per locale, as described in Section 3.1.2. Moreover, `DistBag-DFS` is generic and requires the `Node` type of the elements. Furthermore, the root node is inserted inside the bag (line 7)—note that we must specify the segment into which the root is inserted (here segment 0 is chosen arbitrarily).

At this point, the parallel exploration of the tree starts with the creation of concurrent homogeneous tasks (one per loop-iteration, *i.e.* one per locale), using the `coforall` statement (line 9). This statement can be seen as a concurrent *for*-loop, creating distinct tasks, each of which executes a copy of the loop body. In addition, each of these tasks is going to execute on a remote locale using the `on` clause. Then, locale-specific variables are created, such as termination detection flags and *state vector* (lines 10-12). A second `coforall` loop-based tasking construct is then used to exploit the intra-node parallel level, creating as many tasks as threads per locale (line 13). Then, each created task indefinitely performs the following:

- Retrieve an node (line 17). Each task tries to take a parent node (*i.e.* a subproblem) from the bag using the `remove` procedure, that returns also the final state of the operation (`SUCCESS`, `FAST_FAIL` or `FAIL`). As explained in Section 3.1, this procedure contains an underlying WS mechanism.
- Check the termination (line 18). To favor readability, the pseudo-code of the distributed termination detection is discussed separately.
- Decompose the node (line 20). Each parent node is decomposed according to the problem-specific decomposition function. In B&B, this function consists in the branching, bounding and pruning operators, while in UTS it consists in generating a random number of children to create. We suppose that `decompose` is a thread-safe procedure that creates a (possibly empty) set of children nodes from an input parent node. Then, the resulting children nodes are inserted in bulk into the bag—more precisely, into the segment from which the parent was retrieved (line 21).
- Share the global knowledge (line 22). Some applications require continuous exchange of knowledge between tasks/locales, *e.g.* the cost of the best solution found so far in B&B. This is typically managed using global atomic variables that are only read periodically to avoid high parallel overheads.

Algorithm 1: Pseudo-code of P3D-DFS in Chapel

```
1  var PrivSpace: domain(1) dmapped Private();
2  var localeState: [PrivSpace] atomic bool = BUSY;
3  var Tasks: domain(1) = 0..#here.maxTaskPar;
4
5  var root = new Node(problem);
6  var bag = new DistBag_DFS(Node, Locales);
7  bag.add(root, 0);
8
9  coforall loc in Locales do on loc {
10   var allTasksIdle: atomic bool = false;
11   var allLocalesIdle: atomic bool = false;
12   var taskState: [Tasks] atomic bool = BUSY;
13   coforall taskId in Tasks {
14     allLocalesBarrier.barrier();
15     while true {
16       var (hasWork, parent): (int, Node);
17       (hasWork, parent) = bag.remove(taskId);
18       /* Check termination condition */
19       var childList: list(Node);
20       childList = problem.decompose(parent);
21       bag.addBulk(childList, taskId);
22       /* Share the global knowledge */
23     }
24   }
25 }
```

Algorithm 2 shows our implementation of the distributed termination detection. The latter is based on a bi-level detection using *state vectors*. At the intra-node level, we maintain a vector of Boolean atomics `taskState` (one atomic per task), indicating tasks' status (`IDLE` or `BUSY`). The status of each task is only accessible by tasks of the same locale, and local variables are used in practice to avoid "unnecessary" coherence operations when accessing `taskState`, also known as *false sharing* [23]. Similarly, at the inter-node level, we maintain a vector `localeState`, for which the set of indices is distributed across locales, *i.e.* index i to locale i (lines 1-2 in Algorithm 1). The status of each locale is accessible by any task of any locale. According to Chapel, the writing and reading of atomic variables is done by calling the convenient `write` and `read` procedures, respectively. It is worth to mention that local variables are used in practice to avoid unnecessary global atomic accesses. Then, according to the status of the `remove` procedure (here denoted `hasWork`), three cases are encountered:

- If `SUCCESS`, the calling task still has work and sets its status as well as its locale's one to `BUSY` (lines 2-3).
- If `FAST_FAIL`, the calling task is idle and failed to steal work locally. However, there is an ongoing global steal request and we assume that it is not necessary to check the termination detection in that case. Thus, it simply puts its status on `IDLE` (line 6) before continuing the search.
- If `FAIL`, the calling task is idle and failed to steal work locally and globally. In that case, there is a high probability so that each locale is almost empty. Therefore, it first checks the intra-node detection (line 11), and if necessary, the inter-node one (line 13). If the termination is detected, the calling task breaks the exploration (line 14). Termination flags are used (lines 11,13) to allow fast checking for the remaining tasks.

Algorithm 2: Pseudo-code of P3D-DFS's termination detection in Chapel

```
 1 if (hasWork == SUCCESS) {
 2     taskState[taskId].write(BUSY);
 3     localeState[localeId].write(BUSY);
 4 }
 5 else if (hasWork == FAST_FAIL) {
 6     taskState[taskId].write(IDLE);
 7     continue;
 8 }
 9 else if (hasWork == FAIL) {
10     taskState[taskId].write(IDLE);
11     if all_idle(taskState, allTasksIdle) {
12         localeState[localeId].write(IDLE);
13         if all_idle(localeState, allLocalesIdle) {
14             break; /* End of the exploration */
15         }
16     } else {
17         localeState[localeId].write(BUSY);
18     }
19     continue;
20 }
```

3.3. Baseline implementations

In this section, the state-of-the-art baseline implementations considered for comparison purpose are presented.

OMP-PBB. We implemented a C++/OpenMP version of an asynchronous parallel multi-core B&B algorithm [24]. It aims at mirroring the behavior of P3D-DFS at the intra-node level as closely as possible. A shared multi-pool is implemented as a vector of deques containing (pointers to) node elements (whose type is specified as a template parameter). The OpenMP `atomic` construct and OpenMP lock variables are used for synchronization. The shared multi-pool operates in the same way as `DistBag-DFS`. However, instead of using custom split deque implementations, OMP-PBB relies on lock-protected containers from the C++ standard library. For further details, the pseudo-code of the implementation is described in Appendix A.

OMP-PUTS. This implementation of UTS based on C+OpenMP is available in the public repository [25] of Dinan *et al.* In this latter, each OpenMP thread maintains a *stealStack*, which is a stack of nodes with sharing at the bottom of the stack and exclusive access at the top for the "owning" thread, which has affinity to the stack's address space. Elements move between the shared and exclusive portion of the stack solely under control of the owning thread (`stealStack_release` and `stealStack_acquire`). Moreover, all operations on the shared portion of the stack are guarded using a stack-specific lock, based on the `omp_(un)set_lock` procedures. When a thread has no more work, it tries to steal work from another thread's stack. The victim thread is selected in a ring fashion, and the amount of work to steal is parameterized by `chunkSize`. When a thread is unabled to steal work from shared portion of other stacks, it enters quiescent state waiting for termination, unless some thread has made work available. The parallel search ends when all threads are in the waiting state.

MPI-PBB. Single program - multiple data (SPMD) B&B written in C++/MPI+pthreads [24]. MPI-PBB is a hierarchical Master-Worker approach where workers (MPI processes) can be multi-threaded or GPU-based. An interval-based encoding of work units and the IVM data structure [26], specifically designed for permutation-based problems (*e.g.* PFSP), are used 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. A dedicated thread is used for communication with the master process, allowing to overlap work progress and communication efficiently. To overlap computation and communication, redundant exploration of some parts of the search space is tolerated. Inter-node work load balancing is performed by the intermediate of the centralized coordinator process. The presence of a master process centralizing the work-pool makes termination detection trivial and simplifies checkpointing. However, at large scale, the master process becomes a sequential bottleneck.

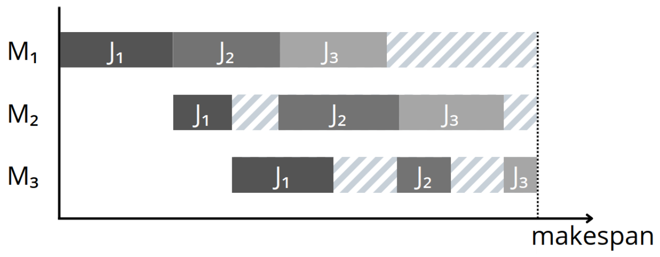

Figure 3: Solution of a PFSP instance consisting of 3 jobs and 3 machines.

MPI-PUTS. We consider the C+MPI-based implementation proposed by Dinan *et al.* in their UTS benchmark [25], that allows us to characterize the performance of unbalanced computations on distributed-memory systems [27]. In this algorithm, 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. The frequency with which a processor enters the progress engine is adaptive and the amount of work to be stolen is parameterized by `chunkSize`. Under this approach, processes with no work constantly search for work to become available until termination is detected. More precisely, the global termination is managed using a colored token which is circulated around the processes in a ring in order to reach a consensus. In this implementation, MPI two-sided communications occur through the WS mechanism.

4. Experimental results and discussion

We first present in Section 4.1 the application context. Then, Sections 4.2 and 4.3 describe the experimental protocol and the parameter settings, respectively. Furthermore, Section 4.4 presents the experimental results. In the latter, we first compare both `DistBag` and `DistBag-DFS` data structures in terms of memory requirements in the context of DFS. Then, we evaluate and compare in terms of performance the Chapel and baseline implementations, on both shared- and distributed-memory systems. Up to 32 computer nodes (4096 cores) are used. Finally, we discuss the productivity-awareness of P3D-DFS in Section 4.5.

4.1. Application context

P3D-DFS is instantiated on the B&B method and the UTS benchmark. These unbalanced tree-based applications provide a means to study the expression and performance of algorithms requiring continuous dynamic load balancing. The PFSP is chosen as a test-case for the B&B.

4.1.1. The Permutation Flow-shop Scheduling Problem

The PFSP consists in finding an optimal processing order (a permutation) for n jobs $\{J_1, \dots, J_n\}$ on m machines $\{M_1, \dots, M_m\}$, such that the completion time of the last job on the last machine (makespan) is minimized. Obeying a chain production principle, the processing of a job J_j on machine M_k can only start if processing of J_j is completed on all upstream machines $M_1, \dots, M_{k-1}$. Processing job J_j on machine M_k takes a given indivisible amount of time p_{jk} and all jobs are to be processed in the same order on all machines. Figure 3 shows an example of a solution of a PFSP instance consisting of $n = 3$ jobs and $m = 3$ machines. For $m = 2$, the problem can be solved in $O(n \log n)$ steps by sorting the jobs according to the Johnson's rule [28]; for $m \geq 3$ it is shown to be NP-hard [29].

To evaluate a subproblem (bounding), different lower bound functions (LBs) exist. In this paper, three different ones are considered:

- LB1: the so-called one-machine bound [30] which can be computed in $O(mn)$ steps for one supproblem;
- LB1$_\Delta$: a fast implementation of LB1, that takes advantage of the fact that the LB1 value of a child node can be obtained incrementally from the parent in $O(m)$ steps, *i.e. all* children are evaluated in $O(mn)$ steps [31];
- LB2: the so-called two-machine bound which relies on the exact resolution of two-machine problems obtained by relaxing capacity constraints on all machines, with the exception of a pair of machines $(M_u, M_v)_{1 \leq u < v \leq m}$. Taking the maximum over all $\frac{m(m-1)}{2}$ machine-pairs, LB2 gives the sharpest known LB (of polynomial complexity) [31]. LB2 can be computed in $O(m^2 n)$ steps per subproblem.

Using LB1$_\Delta$, the algorithm explores identical search trees as with LB1, but the latter yields a more fine-grained workload (lower computational cost for evaluating one subproblem). The sharper LB2 bound results in a more coarse-grained, but also more irregular workload, due to the improved efficiency of the pruning operator.

In the PFSP, a tree node (subproblem) is typically a structure that contains all necessary information for its evaluation, *i.e.* its depth, the number of scheduled jobs and its permutation. Moreover, the `decompose` function mentioned in P3D-DFS (see Section 3.2) is composed of the sequential branching, bounding and pruning operators.

4.1.2. The Unbalanced Tree Search benchmark

The UTS benchmark [32] consists in counting the number of nodes in an implicitly constructed tree that is parameterized in shape, depth, size and imbalance. Implicit construction means that each node contains all information necessary to construct its children. Thus, starting from the root, the tree can be traversed in parallel in any order as long as each parent is visited before its children.

UTS trees are generated using a process, in which the number of children of a node is a random variable with a given distribution. To create deterministic results, each node is described by a 20-byte descriptor. The child node descriptor is obtained by application of the SHA-1 cryptographic hash [33] on the pair: parent descriptor / child index. The node descriptor is also the random variable used to determine the number of children of the node. Consequently the work in generating a tree with N nodes is N SHA-1 evaluations. Carefully validated implementations of SHA-1 exist which ensure that identical trees are generated from the same parameters on different architectures. One of the parameters of a tree is the value r of the root node. Multiple instances of a tree type can be generated by varying this parameter, hence providing a check on the validity of an implementation.

In this paper, we focus on binomial trees, *i.e.* each node has q children with probability p and has no children with probability $1 - p$, where $p \in [0, 1]$. When $pq < 1$, this process generates a finite tree, and the variation of subtree sizes increases dramatically as pq approaches 1 [32]. This is the source of the tree's imbalance. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be gained by choosing to move one node over another for load balance—the expected work at all nodes is identical. The root-specific branching factor b_0 can be set sufficiently high to generate an interesting variety of subtree sizes below the root. Finally, to vary the granularity in our experiments, we introduced the g parameter which controls the number of SHA-1 evaluation(s) per decomposed node.

A significant advantage of the UTS benchmark compared to B&B is that it allows us to vary the instance size and granularity independently. Indeed, in general, the more coarse-grained is the B&B lower bound and the smaller the critical tree; and *vice versa*. From an implementation point of view, an UTS tree node contains all necessary information for the generation of its children, *e.g.* the distribution governing the number of children (p and q). Moreover, the `decompose` function contains the SHA-1 evaluation(s).

4.2. Experimental protocol

In this evaluation, P3D-DFS is instantiated on the B&B method and the UTS benchmark. Moreover, for experimental validation, we consider the PFSP for B&B and synthetic literature benchmarks for UTS. Then, our algorithm is experimented at both intra- and inter-node parallel levels, and compared to the best-known counterparts, based on traditional OpenMP and MPI+X (see Section 3.3). It is worth to mentioned that each implementation uses the DFS strategy as selection rule, and that the decomposition functions are the same. Moreover, for a fair comparison, all implementations enumerate equivalent search spaces. Nevertheless, the implementations are still very different from each other (data structure, WS mechanism, termination detection, *etc.*), and it is therefore not a purpose here to evaluate the programming languages/libraries. We investigate the intra- and inter-node performances of P3D-DFS, its scalability according to the number of computer nodes, and the benefits of our `DistBag-DFS` distributed data structure.

Three series of experiments are performed: (1) we evaluate the memory requirements of both `DistBag` and `DistBag-DFS` data structures in the context of DFS, (2) we evaluate and compare the scalability of the P3D-DFS and OpenMP baseline implementations on shared-memory setting, and (3) we evaluate and compare the performance and scalability of the P3D-DFS and MPI+X baselines implementations on distributed-memory settings.

Table 1: Sample UTS trees with parameters and their resulting size in millions of nodes.

Tree	b_0	p	q	r	NNodes (10^6)
T1	2000	0.499995	2	38	5
T2	2000	0.499995	2	30	51
T3	2000	0.499995	2	55	514
T4	2000	0.499999995	2	0	10612
T5	2000	0.4999975	2	559	52990
T6	2000	0.499999	2	559	94795

Table 2: PFSP Taillard's instances with parameters and tree size in millions of nodes, according to the different LBs.

Instance	m	n	Optimal makespan	NNodes (10^6)	
				LB1$_{(\Delta)}$	LB2
Ta10	5	20	1108	83	8
Ta20	10	20	1591	859	4
Ta30	20	20	2178	111	13
Ta27	20	20	2273	1854	54
Ta26	20	20	2226	11392	514
Ta24	20	20	2223	71876	2173

4.3. Parameter settings

Tables 1 and 2 summarize the UTS and PFSP instances considered as test-cases with their parameters and sizes in million of nodes, respectively. These PFSP instances are from the benchmark proposed by Taillard [34], which is the mostly used in the literature. For larger instances ($n > 20$) the resolution time becomes excessive—unless advanced branching techniques are used [35]—and those instances are therefore not considered.

To compare the performance of B&B algorithms, they should explore the same search space. When a problem instance is solved repeatedly using parallel B&B, the number of explored subproblems is often different between independent resolutions, because tree nodes are expanded in a different order. In order to study the performance of the parallel algorithm in the absence of speed-up anomalies, we always initialize the algorithm with the optimal solution of the instance to be solved. With this initialization, the algorithm proves the optimality of the initial upper bound and it is ensured that both the sequential and parallel algorithms explore exactly the critical tree (*i.e.* all nodes for which the bounding operator gives a lower bound smaller than the optimal makespan). Corresponding Taillard's optimal solutions are reported in [36].

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. The Chapel implementation is based on version 1.29.0 and uses the *default* task layer (qthreads). Chapel's multi-locale code runs on top of GASNet, and several environment variables are set with the characteristics of the system the multi-locale code is running. One can see in Table 3 a summary of the runtime configuration for multi-locale execu-

Table 3: Summary of the Chapel environment configuration for multi-locale execution and compilation.

Variable	Value
CHPL_RT_NUM_THREADS_PER_LOCALE	128
CHPL_TARGET_CPU	*native*
CHPL_COMM	*gasnet*
CHPL_COMM_SUBSTRATE	*ibv*
CHPL_LAUNCHER	*slurm-gasnetrun_ibv*
GASNET_IBV_SPAWNER	*ssh*

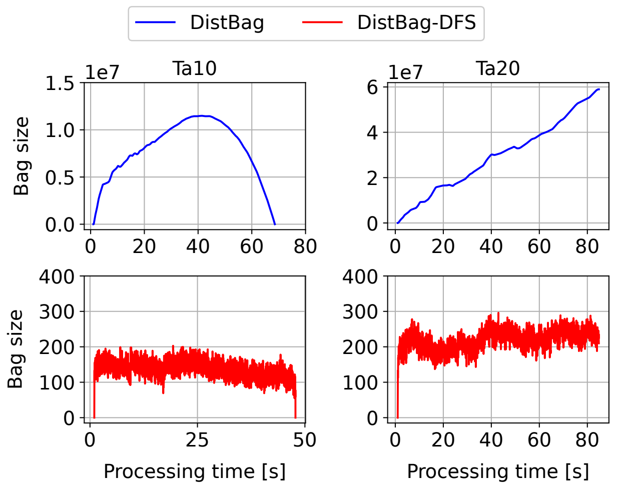

Figure 4: Bag size (in subproblems) according to the processing time when solving the `Ta10` (left) and `Ta20` (right) PFSP instances.

tion and compilation. The ibv-conduit implementing GASNet over InfiniBand networks is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with SSH, which is responsible for getting the executables running on different locales (`GASNET_IBV_SPAWNER`). Concerning the OpenMP and MPI+X implementations, we used OpenMP 4.5, Open MPI 4.0.5 as well as the `gcc` 10.2.0 compiler. Both Chapel and C-based implementations are compiled using the `--fast` and `-O3` optimization flags, respectively.

In MPI-PBB, each worker can be configured to use any number of worker threads. We chose to use 16 threads (plus one additional communication thread) per MPI process. Therefore, a total of $8L$ MPI processes is launched and mapped evenly across the L compute nodes. Node 0 runs the master process using the same configuration, meaning that node 0 hosts at most 7 worker processes (112 threads).

4.4. *Experimental results*

4.4.1. *`DistBag` vs. `DistBag-DFS`*

In this section, both `DistBag` and `DistBag-DFS` data structures are compared in terms of memory usage, in the context of DFS. As a test-case, we consider the B&B instantiation of P3D-DFS applied to the PFSP. `Ta10` and `Ta20` are solved, using LB1 and 4 processing threads.

Figure 4 shows the evolution of the bag size (in number of subproblems) over time, which is roughly proportional to the overall memory usage of the program. When solving `Ta10` using `DistBag`, we first note a monotonic growth phase during 40s. At its maximum size, the bag holds up to 1.2×10^7 nodes. Then, the bag size is progressively decreasing to 0, signifying the end of the exploration. This change in behavior can be explained by the fact that in the upper parts of the search tree the average branching factor is higher than in the bottom parts. Indeed, all considered LBs increase monotonically according to the depth of a subproblem, meaning that for all children c of a parent p, LB$(c) \geq$ LB(p). Therefore, the average branching factor decreases as the search moves deeper into the tree, causing the drain of the bag. Considering `Ta20`, the size of the bag grows in a quasi-linear way until 6×10^7 nodes, and we had to manually interrupt the experiment before the memory consumption becomes too excessive. To get an idea of the latter, let's multiply the number of nodes inside the bag by the bit size of each node (composed of 23 64-bit integers): it gives $6 \times 10^7 \times (23 \times 64) = 88 \times 10^9$ bits, or 11 GB. The same experiments using the `DistBag-DFS` gives us very different results. We observe that for both instances, the size of the bag, and thus the memory consumption, remains bounded over time. Theoretically, in a DFS B&B algorithm applied to PFSP, the maximum size that a pool can contain is $\sum_{i=1}^{n-1} i$, *i.e.* 190 in our experiments ($n = 20$). Indeed, by definition, the subtree of a node must be completely explored before decomposing another node of the same depth. Therefore, for each level l, at most $n - l$ nodes are kept in memory (the generated but not yet evaluated subproblems). We note that this is satisfied, since the bag size never exceeds $4 \times 190 = 760$ at a time.

These preliminary experiments show that our `DistBag-DFS` data structure is more reliable than `DistBag` in the context of DFS. Indeed, it allows a better control of the memory management, in particular *via* the redesign of the scheduling policy of its elements (see Section 3.1).

4.4.2. *Intra-node performance*

In this section, we compare the performance of our Chapel-based P3D-DFS algorithm to the OpenMP-based baselines, at the intra-node parallel level.

Figure 5 shows the absolute speed-up achieved by P3D-DFS and OMP-PBB on the B&B method and its application to the PFSP. Different granularities are considered (LBs: LB1$_\Delta$, LB1 and LB2), and the `Ta10`, `Ta20`, and `Ta30` instances are solved. First of all, one can see that for a given instance, coarser is the LB, better is the speed-up. Indeed, the computation time of the LB overlays the parallel overheads. The best results are obtained using LB2, where the speed-up ranges from 85% to 95% of the ideal speed-up for P3D-DFS, and from 60% to 85% for OMP-PBB. Then, we can note that P3D-DFS always achieves better or comparable performance to OMP-PBB. The difference in absolute speed-up is explained by the faster single-threaded execution time for P3D-DFS. Indeed, as shown in Table 4 of Appendix B, OMP-PBB's single-threaded execution time is up to 42% larger considering LB1$_\Delta$, 45% for LB1, and 18% for LB2. Although both approaches implement the same

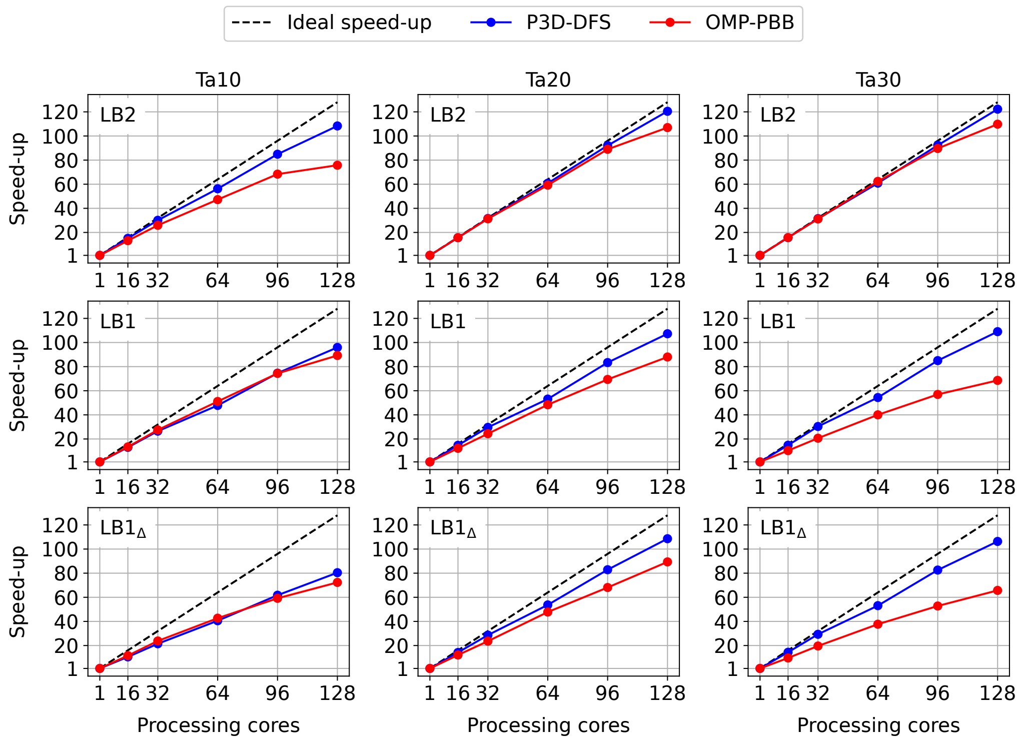

Figure 5: Absolute speed-up achieved by P3D-DFS and OMP-PBB on B&B applied to PFSP, considering different granularities (LBs: $LB1_\Delta$, LB1 and LB2) and instances (Ta10, Ta20, Ta30). Most coarse-grained is top-right, most fine-grained is bottom-left. Processing cores varies from 1 to 128.

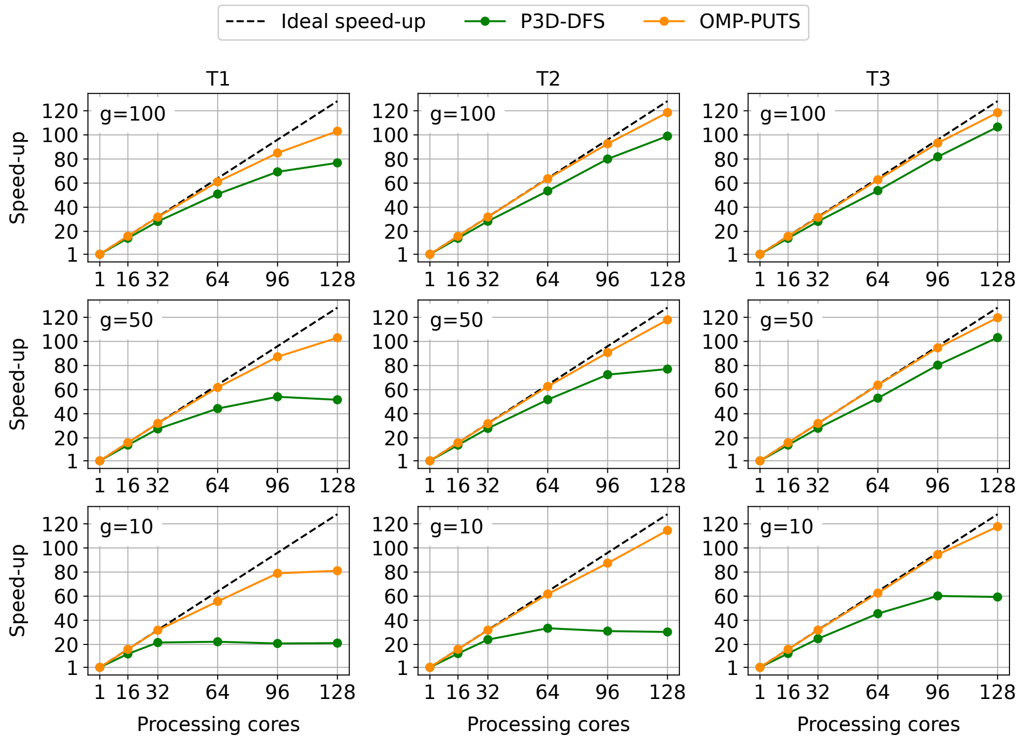

Figure 6: Absolute speed-up achieved by P3D-DFS and OMP-PUTS on UTS, considering different granularities g and tree sizes (T1, T2, T3). Most coarse-grained is top-right, most fine-grained is bottom-left. Processing cores varies from 1 to 128.

termination detection, as well as the same LBs calculation, the difference in performance can be explain by the use of different data structures, relying on different locking mechanisms for example.

Similarly to the previous comparison, Figure 6 shows the absolute speed-up achieved by P3D-DFS and its OMP-PUTS counterpart, considering the UTS benchmark. Different granularities g are considered, as well as different tree sizes (`T1`, `T2` and `T3`). In all cases, our P3D-DFS implementation is outperformed by its counterpart, especially at low granularity where OMP-PUTS achieves performance up to 3× better. In addition, one can see on Table 4 that P3D-DFS's single-threaded execution times are between 10 – 20% larger than those of OMP-PUTS. At this point, it is important to note that both implementations are quite different, in contrast to the previous comparison. Mainly, they implement different data structures, different WS mechanism, as well as different termination detection mechanisms. We still have to investigate the origin of the performance pitfall(s) that occur(s) in our Chapel-based implementation, but a first explanation could come from one of the aforementioned mechanisms which would be less effective in P3D-DFS than in OMP-PUTS. When the granularity is high ($g = 100$ for instance), the gap between the two implementations narrows. For its best results, P3D-DFS achieves 82% of the ideal speed-up using 128 processing cores.

Based on the reported results, it appears that our Chapel-based P3D-DFS implementation achieves high performance at the intra-node level in the context of the B&B method and its application to the PFSP, and outperforms its OpenMP-based counterpart. Therefore, we can expect the resolution of large PFSP instances ($n > 20$) with P3D-DFS to be highly efficient. Nevertheless, on UTS, which is a synthetic benchmark with a fine-grained decomposition operator, P3D-DFS struggles to achieve good scalability, while the OpenMP-based baseline does.

4.4.3. Inter-node performance

In this section, we compare the performance of P3D-DFS to the MPI+X baselines, at the inter-node level of parallelism.

Figure 7 shows the absolute speed-up achieved by P3D-DFS and MPI-PBB on B&B applied to PFSP, considering different granularities and the `Ta27`, `Ta26`, and `Ta24` instances. We first note that P3D-DFS outperforms MPI-PBB in almost all cases, and more particularly solving the smallest `Ta27` instance. As shown in Table 5 of Appendix B, the single-node execution time for MPI-PBB is between 6% and 30% larger than P3D-DFS, in all cases. In addition, the lack of performance of MPI-PBB can be explained by the presence of a centralized coordinator-process in the inter-node load balancing mechanism that can lead to sequential bottleneck at large scale, as explained in Section 3.3. Nevertheless, one can note that larger is the critical tree size of the instance, smaller is the gap between the implementations. Especially solving `Ta24` with $\mathrm{LB1}_{\Delta}$, which is the largest fine-grained instance solved, MPI-PBB performs better than P3D-DFS. This can be related to the data structures, and suggests that `DistBag-DFS` could be less efficient than its counterpart in MPI-PBB. For its best results, P3D-DFS achieves 93% of the ideal speed-up, using 32 computer nodes.

Similarly to the previous comparison, one can see in Figure 8 the absolute speed-up achieved by P3D-DFS and MPI-PUTS, on the UTS benchmark. We can see that MPI-PUTS outperforms P3D-DFS at low granularity ($g = 10$). More precisely, P3D-DFS is up to 50% less efficient than the baseline. This can be explained by the limited scalability of our algorithm at fine granularity at the intra-node level, as observed in Figure 6. Moreover, this is consistent with the single-node execution time which is up to 43% higher for P3D-DFS, as shown in Table 5. Nevertheless, when g increases, P3D-DFS provides comparable or better performances than MPI-PUTS. More particularly, we observe that P3D-DFS outperforms its MPI counterparts on the smallest instance `T4`. P3D-DFS speed-up is more than 2× larger than MPI-PUTS, considering 32 computer nodes. Limited scalability of MPI-PUTS can be explained by the fact that the implementation does not have a bi-level WS mechanism, as the P3D-DFS does, meaning that a thief thread steals a local/remote victim one, without distinction. For its best results, P3D-DFS achieves 78% of the ideal speed-up, using 32 computer nodes.

Based on the reported results, it appears that P3D-DFS outperforms its MPI+X counterparts, especially on the B&B and its application to the PFSP. Its bi-level WS mechanism has proven to be particularly effective up to 32 computer nodes. Consering fine-grained UTS trees, P3D-DFS's inter-node performance is limited by its low intra-node scalability, observed in Section 4.4.2.

4.5. Discussion on productivity

In this section, the productivity-awareness of P3D-DFS is discussed. Some authors characterized HPC productivity as a trade-off between performance and programming effort [37]. While the first aspect has been discussed in Section 4.4, we will now give some highlights of the second one.

First of all, P3D-DFS is generic and general. It is designed to be applicable to any DFS algorithm, *e.g.* for solving combinatorial problems. This is facilitated by the object-oriented programming supported by Chapel, as well as the generic `DistBag-DFS` data structure. As explained in Section 3.1, the latter hides to the user the multi-pool implementation as well as the underlying WS, which has the benefit of making the multi-core implementation close to the sequential one, in terms of lines of code. In turn, Chapel's global view of control flow and data structures make it straightforward to design and implement a distributed B&B/backtracking based on a multi-core version. Actually, as shown in Section 3.2, programming a distributed version starting from a multi-core one requires the addition of a few lines of codes, and there is no need to deal explicitly with inter-node communications or command line launch parameters, as it is the case in MPI+X. In the latter, considering only the core of the master process, multiple synchronization points at the inter-node level (WS, termination detection, update of the current best solution) need to be handled, which is challenging and error-prone, as it induces hard-to-detect race conditions [15].

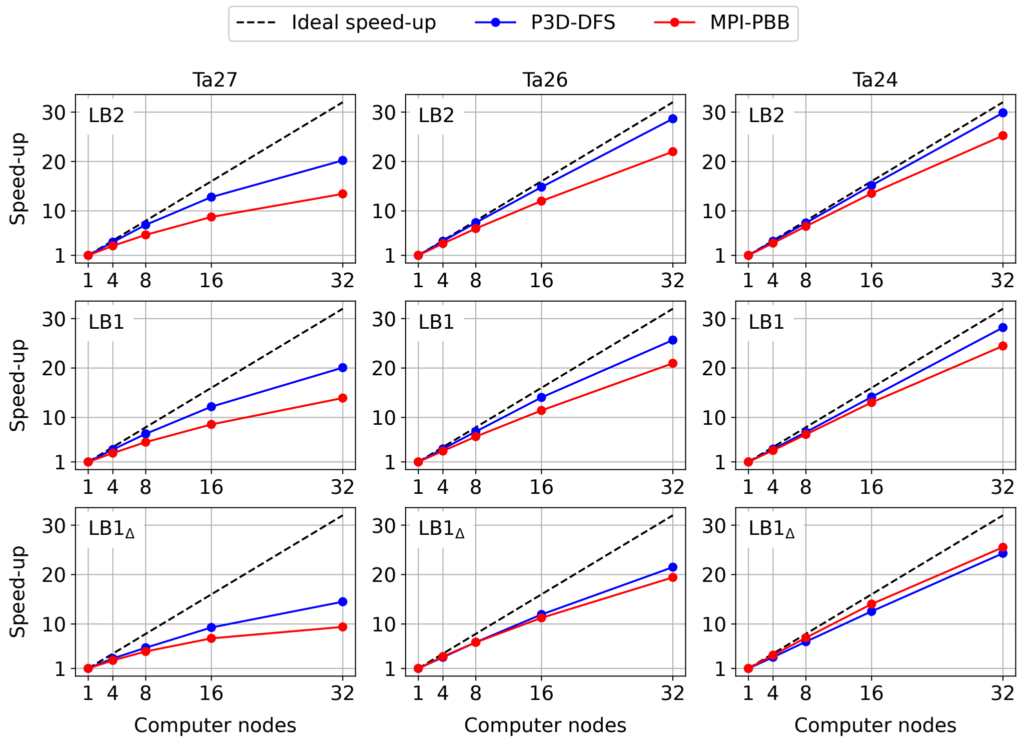

Figure 7: Absolute speed-up achieved by P3D-DFS and MPI-PBB on B&B-PFSP, considering different granularities (LBs: $LB1_\Delta$, LB1 and LB2) and instances (Ta27, Ta26, Ta24). Most coarse-grained is top-right, most fine-grained is bottom-left. Computer nodes vary from 1 to 32.

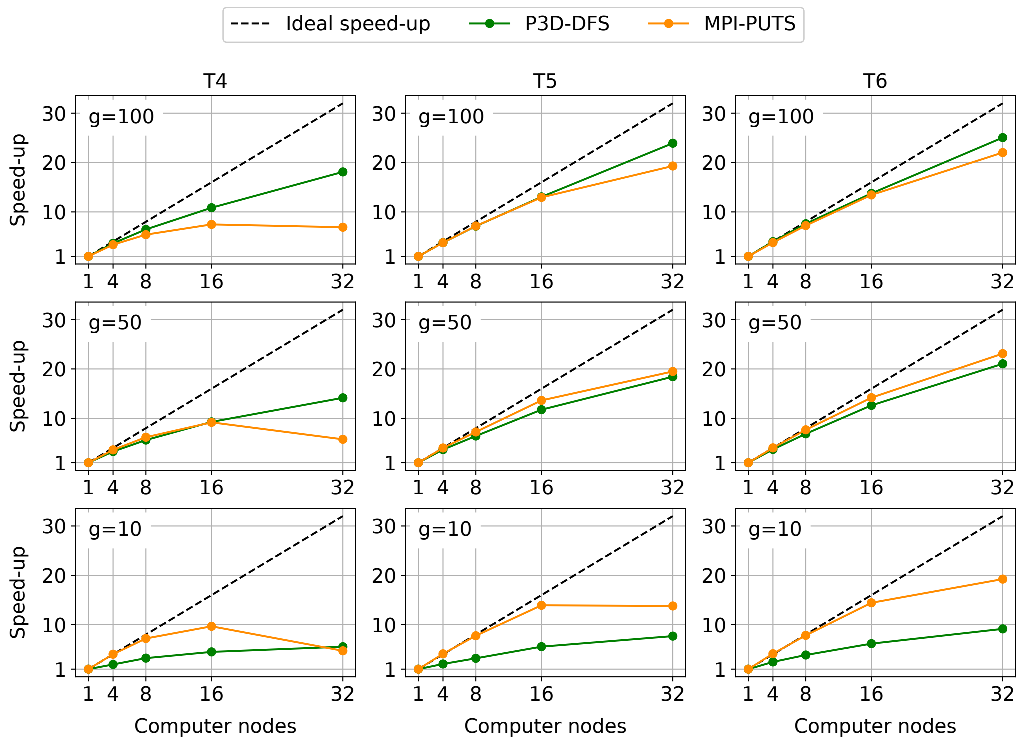

Figure 8: Absolute speed-up achieved by P3D-DFS and MPI-PUTS on UTS, considering different granularities g and tree sizes (T4, T5, T6). Most coarse-grained is top-right, most fine-grained is bottom-left. Computer nodes vary from 1 to 32.

In contrast, the use of Chapel global shared atomics comes in handy, but needs to be handled with extreme care to avoid performance pitfalls. Similarly, the programmer must be aware of the underlying communications. Finally, to achieve the best performances of Chapel programs, one has to install and highly tune its execution environment, in contrast to the ubiquitous OpenMP and MPI. Depending on the hardware, this can be a daunting task, especially since Chapel's documentation is restricted to a few system configurations, *e.g.* Infiniband network with Slurm for job spawning.

5. Conclusions and future works

In this paper, we have investigated Chapel's performance within the context of parallel DFS on both shared- and distributed-memory systems. We designed and implemented a general PGAS-based parallel distributed tree-search algorithm unifying both intra- and inter-node parallel levels, and evaluated it using large unbalanced tree-based problems, which are the B&B applied to PFSP and UTS benchmarks. The implementation is based on the `DistBag-DFS` distributed data structure, which is our revisited version of the Chapel's `DistBag` data structure for DFS. `DistBag-DFS` enables a better memory management in a DFS context, as well as a new synchronization mechanism between threads. Our implementation has been compared to state-of-the-art baseline approaches in terms of performance, using up to 32 computer nodes including each 128 cores. In a shared-memory setting, the experimental results revealed that P3D-DFS is competitive with its OpenMP counterparts for coarser-grained instances, while it suffers from parallel overheads at low granularity. Moreover, by using our `DistBag-DFS` data structure, one can achieve performance equivalent to MPI+X in a distributed-memory setting, but with a programming effort much smaller. Therefore, Chapel stands out as an alternative to OpenMP and MPI+X at the intra- and inter-node parallel levels for parallel distributed tree-search, in the context of exascale programming.

In the future, we expect `DistBag-DFS` to be integrated in the Chapel language. Actually, we have demonstrated in this paper that this distributed data structure is now fully operable in a DFS context. It could now be employed in a productive manner by the Chapel community. In this paper, we have considered both intra- and inter-node parallel levels. However, we still have to consider heterogeneous systems, including GPU accelerators. Chapel 1.29.0 includes preliminary work to target NVidia GPUs using the CUDA driver API at runtime. Such an approach will be compared to MPI+X+CUDA implementations, still on challenging unbalanced tree-based problems. From the implementation point of view, we have seen that P3D-DFS still suffers from some management overheads at the intra-node level, and a series of low-level optimizations remains to be performed. Doing so, we will contribute a bit more to the debate between the two schools of thought: evolutionary or reuse what we have (MPI+X) *vs.* revolutionary or using new languages designed for exascale (*e.g.* Chapel).

Acknowledgments

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg (see `https://hpc.uni.lu`). Moreover, the code has been developed and debugged 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 (see `https://www.grid5000.fr`). 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.

Data availability statement

All Chapel code written in support of this publication are publicly available at `https://github.com/Guillaume-Helbecque/P3D-DFS`, and archived on Zenodo (DOI: 10.5281/zenodo.7674860). Moreover, the parallel B&B and UTS implementations considered as baselines are available at `https://github.com/jangmys/pbb` and `https://sourceforge.net/projects/uts-benchmark/`, respectively, and archived on Zenodo (DOI: 10.5281/zenodo.7674826 and DOI: 10.5281/zenodo.437497, respectively).

Appendix

A. OMP-PBB implementation

In this section, we provide the pseudo-code of the parallel multi-core C++/OpenMP implementation described in Section 3.3. It follows the same flow as the Chapel one. Nevertheless, some notable distinctions are present. In Chapel, the element-type `Node` is passed to the bag's initializer, whereas in C++ this is typically achieved by specializing the `sharedPool` template with the `Node` template parameter (line 2).

Another point is that we do not use `for-loop` in our OpenMP implementation to launch concurrent tasks. Although this is quite possible, OpenMP's fork-join mechanism (`#pragma omp parallel`) in combination with the built-in `omp_get_thread_num()` function seems to be the simpler solution. Moreover, the way to manage atomicity is different according to both implementations. Indeed, in P3D-DFS, we declared atomic variables, while in OpenMP we have to explicitly declare atomic memory location through specific compiler directives to ensure that race conditions are avoided through direct control of concurrent threads. Thus, we introduce the `set_stop_flags` and `atomic_read` functions that include `omp atomic write`, `omp atomic read` and `omp flush` pragmas to ensure atomicity and consistency of memory (lines 20, 27, 30).

Algorithm 3: Pseudo-code of OMP-PBB

```
Node root(instance);
sharedPool<Node> pool;
pool.insert(root, 0);

int global_best_ub = ub_init();
int stop_flags[omp_get_max_threads()];

#pragma omp parallel
{
  int threadId = omp_get_thread_num();
  int local_best = global_best_ub;

  stop_flags[threadId] = BUSY;

  while true {
    local_best = global_best_ub;
    Node n = pool.take(threadId);
    // if locally empty and steal failed
    if !n {
      set_stop_flags(threadId, IDLE);
      // if globaly empty
      if all_idle(stop_flags) {
        break;
      }
      continue;
    } else {
      set_stop_flags(threadId, BUSY);
      if n->is_leaf() {
        int ub = evalSolution(n);
        if (ub < local_best && ub < atomic_read(
    global_best_ub)) {
          #pragma omp critical
          {
            global_best_ub = ub;
            // updates best solution ...
          }
        }
      } else {
        std::vector<Node> ns;
        ns = decompose(n, local_best);
        pool.insert(ns, threadId);
      }
    }
  }
}
```

B. Reference times for absolute speed-up calculation

Tables 4 and 5 summarize the single-threaded and single-node execution times of our Chapel-based implementation and its baselines in shared- and distributed-memory settings, respectively.

References

[1] K. Asanovic, R. Bodik, J. Demmel, *et al.* (2009) A view of the parallel computing landscape. *Commun. ACM* 52, 10, 56–67. DOI: 10.1145/1562764.1562783.

[2] The Chapel Parallel Programming Language. The DistributedBag module. URL: https://chapel-lang.org/docs/modules/packages/DistributedBag.html, version 1.29.0.

[3] J. Gmys. (2022) Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. *INFORMS Journal on Computing* 0(0). DOI: 10.1287/ijoc.2022.1193.

[4] B. Gendron, and T. G. Crainic. (1994) Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research* 42(6):1042–1066. DOI: 10.1287/opre.42.6.1042.

[5] H. W. Trienekens, and A. de Bruin. (1992) Towards a taxonomy of parallel branch and bound algorithms. Technical report. URL: http://hdl.handle.net/1765/1491.

[6] R. D. Blumofe, and C. E. Leiserson. (1999) Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46(5):720–748. DOI: 10.1145/324133.324234.

[7] E. Lusk, and K. Yelick. (2007) Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters* 17:89–102. DOI: 10.1142/S0129626407002892.

[8] B. Chamberlain, E. Ronaghan, B. Albrecht, *et al.* (2018) Chapel comes of age: Making scalable programming productive. Cray User Group. URL: https://chapel-lang.org/publications/cug2018-chapel.pdf.

[9] G. Almasi. (2011) PGAS (partitioned global address space) Languages, *Encyclopedia of Parallel Computing*, pp. 1539-1547. DOI: 10.1007/978-0-387-09766-4_210.

[10] R. B. Johnson, and J. Hollingsworth. (2016) Optimizing Chapel for Single-Node Environments, *IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1558-1567. DOI: 10.1109/IPDPSW.2016.181.

[11] E. Kayraklioglu, W. Chang, and T. A. El-Ghazawi. (2017) Comparative Performance and Optimization of Chapel in Modern Manycore Architectures, *IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. DOI: 10.1109/IPDPSW.2017.126.

[12] T. B. Rolinger, J. Craft, C. D. Krieger, and A. Sussman. (2021) Towards High Productivity and Performance for Irregular Applications in Chapel. *SC Workshops Supplementary Proceedings* (SCWS), pp. 1-11. DOI: 10.1109/SCWS55283.2021.00012.

[13] G. Helbecque, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. (2022) A performance-oriented comparative study of the Chapel high-productivity language to conventional programming environments. *13th International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM), pp. 21–29. DOI: 10.1145/3528425.3529104.

[14] T. Carneiro, and N. Melab. (2019) An Incremental Parallel PGAS-based Tree Search Algorithm, *17th International Conference on High Performance Computing & Simulation* (HPCS). DOI: 10.1109/HPCS48598.2019.9188106.

[15] T. Carneiro, J. Gmys, N. Melab, and D. Tuyttens. (2020) Towards ultra-scale Branch-and-Bound using a High-productivity Language, *Future Generation Computer Systems* 105:196-209. DOI: 10.1016/j.future.2019.11.011.

[16] UPC Consortium. (2005) UPC language specifications, v1.3. Technical Report LBNL-6623E, Lawrence Berkeley National Lab. URL: https://upc.lbl.gov/publications/upc-spec-1.3.pdf.

[17] S. Olivier, and J. Prins. (2008) Scalable Dynamic Load Balancing Using UPC. *37th International Conference on Parallel Processing*, pp. 123-131. DOI: 10.1109/ICPP.2008.19.

[18] R. Machado, C. Lojewski, S. Abreu, and F.-J. Pfreundt. (2011) Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science*. 26, 3–4, 229–236. DOI: 10.1007/s00450-011-0163-3.

[19] D. Grünewald, and C. Simmendinger. (2013) The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*, 243, p. 52.

[20] G. Helbecque, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. (2022) Productivity- and Performance-aware Parallel Distributed Depth-First Search repository. URL: https://github.com/Guillaume-Helbecque/P3D-DFS. DOI: 10.5281/zenodo.7674860.

[21] T. van Dijk, and J. C. van de Pol. (2014) Lace: Non-blocking Split Deque for Work-Stealing. *Euro-Par 2014: Parallel Processing Workshops*. Lecture Notes in Computer Science, vol 8806. DOI: 10.1007/978-3-319-14313-2_18.

[22] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. (2009) Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*

Table 4: Single-threaded execution time (in seconds) for P3D-DFS (bold) and the OMP-PBB and OMP-PUTS baselines on the PFSP and UTS, respectively, considering different granularities (*i.e.* LBs and values of g, respectively).

LB	PFSP						UTS						g
	`Ta10`		`Ta20`		`Ta30`		`T1`		`T2`		`T3`		
$LB1_\Delta$	**41.86**	32.77	**547.46**	570.48	**88.91**	126.25	**7.61**	6.31	**78.91**	65.19	**785.39**	647.22	10
LB1	**64.49**	61.72	**1292.36**	1529.17	**257.02**	372.50	**34.25**	30.62	**354.41**	313.98	**3527.66**	3132.47	50
LB2	**18.29**	21.63	**88.74**	89.73	**706.88**	695.77	**67.42**	60.96	**698.55**	626.52	**6953.12**	6205.90	100

Table 5: Single-node execution time (in seconds) for P3D-DFS (bold) and the MPI-PBB and MPI-PUTS baselines on the PFSP and UTS, respectively, considering different granularities (*i.e.* LBs and values of g, respectively).

LB	PFSP						UTS						g
	`Ta27`		`Ta26`		`Ta24`		`T4`		`T5`		`T6`		
$LB1_\Delta$	**18.99**	23.69	**114.45**	126.46	**720.74**	769.11	**200.02**	149.53	**994.57**	691.52	**1739.93**	1234.39	10
LB1	**48.60**	63.31	**267.26**	328.75	**1717.75**	2053.32	**649.49**	731.10	**3274.58**	3371.43	**5856.14**	6023.25	50
LB2	**34.29**	41.93	**244.56**	285.95	**1060.53**	1217.34	**1219.07**	1453.14	**6153.05**	6623.77	**11006.25**	11658.42	100

(SC '09). Association for Computing Machinery, Article 53, 1–11. DOI: 10.1145/1654059.1654113.

[23] W. J. Bolosky, and M. L. Scott. (1993) False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4* (Sedms'93). USENIX Association, USA, 3. DOI: 10.5555/1295480.1295483.

[24] J. Gmys. (2022) Parallel Branch-and-Bound for permutation-based optimization repository. URL: `https://github.com/jangmys/pbb`. DOI: 10.5281/zenodo.7674826.

[25] J. Dinan, and S. Olivier. (2012) The Unbalanced Tree-Search benchmark repository. URL: `https://sourceforge.net/projects/uts-benchmark/`. DOI: 10.5281/zenodo.7328332.

[26] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. (2017) IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems. *Concurrency and Computation: Practice and Experience*, 29:9. DOI: 10.1002/cpe.4019.

[27] J. Dinan, S. Olivier, J. Prins, G. Sabin, P. Sadayappan, and C.-W. Tseng. (2007) Dynamic Load Balancing of Unbalanced Computations Using Message Passing. *6th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems* (PMEO-PDS 2007). DOI: 10.1109/IPDPS.2007.370581.

[28] S. M. Johnson. (1954) Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* 1:61–68. DOI: 10.1002/nav.3800010110.

[29] M. R. Garey, D. S. Johnson, and R. Sethi. (1976) The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research* 1:117–129. DOI: 10.1287/moor.1.2.117.

[30] A. P. G. Brown, and Z. A. Lomnicki. (1966) Some applications of the "branch-and-bound" algorithm to the machine scheduling problem. *Journal of the Operational Research Society* 17:173–186. DOI: 10.1057/jors.1966.25.

[31] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. (1978) A General Bounding Scheme for the Permutation Flow-Shop Problem. *Operations Research* 26(1):53-67. DOI: 10.1287/opre.26.1.53.

[32] S. Olivier, J. Huan, J. Liu, *et al.* (2007) UTS: An Unbalanced Tree Search Benchmark. *19th International Workshop on Languages and Compilers for Parallel Computing* (LCPC). DOI: 10.1007/978-3-540-72521-3_18.

[33] D. Eastlake 3rd, and P. Jones. (2001) US Secure Hash Algorithm 1 (SHA1). DOI: 10.17487/RFC3174.

[34] E. Taillard. (1993) Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64(2):278-285. DOI: 10.1016/0377-2217(93)90182-M.

[35] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. (2020) A computationally efficient Branch-and-Bound algorithm for the permutation flowshop scheduling problem, *European Journal of Operational Research* 284(3):814-833. DOI: 10.1016/j.ejor.2020.01.039.

[36] Eric Taillard's page, Summary of best known lower and upper bounds of Taillard's instances, URL: `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/flowshop.dir/best_lb_up.txt`, accessed June 2022.

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