Parallel best-first search algorithms for planning problems on multi-core processors

Didier El Baz, Bilal Fakih, Romeo Sanchez Nigenda, Vincent Boyer

Parallel best-first search algorithms for planning problems on multi-core processors

Didier El Baz[1] · Bilal Fakih[1] · Romeo Sanchez Nigenda[2] · Vincent Boyer[3]

Abstract
The multiplication of computing cores in modern processor units permits revisiting the design of classical algorithms to improve computational performance in complex application domains. Artificial Intelligence planning is one of those applications where large search spaces require intelligent and more exhaustive search control. In this paper, parallel planning algorithms, derived from best-first search, are proposed for shared memory architectures. The parallel algorithms, based on the asynchronous work pool paradigm, maintain good thread occupancy in multi-core CPUs. All algorithms use one ordered global list of states stored in shared memory from where they select nodes for expansion. A parallel best-first search algorithm that develops new states with depth equal to one is proposed first. Then, we propose an extension of this parallel algorithm that features a diversification strategy in order to escape local minima. We study and analyse a set of computational experiments for problems that come from the International Planning Competition and real-world industry applications. The empirical evaluation shows that the parallel algorithms solve most of the domains efficiently without incurring higher solutions costs. In those problems with partial results, we highlight the potential shortcomings of the proposed approaches for promising future directions.

Keywords Artificial intelligence planning · Parallel computing · Parallel best-first search · Multi-threading · Asynchronous work pool parallel model

1 Introduction

Artificial intelligence (AI) planning is the problem of finding sequences of actions that, if executed from an initial state of the world, satisfy a goal state [?]. Planning problems occur in many real-world domains like planning tasks for satellites,

airport task planning, airline crew scheduling, autonomous robot navigation, intelligent transportation, and logistics, among others. In general, planning algorithms solve planning problems through action and goal model representations, and heuristics and metrics to control search.

Nowadays, many planning instances continue to be very difficult to solve via sequential algorithms. Classical Planning, the simplest case for AI Planning where actions are fully deterministic and instantaneous, is PSPACE-Complete [4]. Therefore, planning problems search spaces are large and complex. One way to explore different regions of the search space, while looking for a solution, is through the development of heuristics [1, 5, 22, 28]. A second alternative, which has drawn recent attention, is to parallelize search algorithms [16]. Modern multi-core CPUs and computing nodes can provide both the computing resources, i.e., the many computing cores and the memory resources required to solve efficiently hard planning problem instances.

In consequence, parallel best-first search methods, implemented on shared or distributed memory architectures, have been recently proposed [3, 16, 17]. An adaptive K-parallel best-first search algorithm, designed specifically for multi-core domain-independent planning [30], was implemented on the top of the YAHSP planning system [29]. The adaptive parallel algorithm solves 57 more problems than the serial counterpart out of the 2042 instances evaluated from 54 International Planning Competition domains, shedding a promising path for the application of parallel search methods in AI Planning. Nevertheless, some authors (see [19]) have pointed recently that parallel Best-first search algorithms that are suited for both multi-core and multi-machine clusters have not been previously evaluated in depth.

In this paper, we leverage that the many improvements introduced in modern multi-core CPUs as well as computing nodes, e.g., number of cores, memory bandwidth, threads scheduling techniques, and we propose two parallel best-first search algorithms implemented on the top of the LPG-td planning system [9]. The parallel algorithms exploit all the computing resources of modern multi-core CPUs and computing nodes while using an ordered global list of states. States are stored in the ordered list according to the value of the evaluation function. The proposed parallel algorithms are based on the asynchronous work pool parallel model that is convenient for planning applications.

Each parallel thread fetches a state with the best value of the evaluation function in the ordered list. They generate children states that are stored in the ordered global list via mutual exclusion techniques to preserve data consistency. The parallel algorithms keep selecting and expanding states until a goal state is found. We evaluate in depth the proposed parallel algorithms. In particular, we analyse their solutions, i.e., the number of actions as well as the computing time in function of the number of threads used for resolution. The proposed parallel algorithms solve more planning problems from the evaluation set, having better performance in the majority of the problems.

The next section introduces related work on parallel algorithms in the context of AI planning problems. Then, Sect. 3 presents general background on best-first search, which LPG-td considers. After that, Sect. 4 presents in detail the principles of the proposed parallel algorithms. Furthermore, Sect. 5 provides an empirical

analysis of the algorithms in a set of problems from the International Planning Competition (IPC)[1] and problems from real-world industry applications. Finally, we conclude the paper and provide future research directions.

2 Related work

In this section, we give a brief survey on several parallel algorithms. First, we consider parallel methods applied in the context of general search. Then, we present parallel algorithms in the context of planning problems and systems.

One of the most well-known algorithms is Parallel Retracting A* (PRA*), which is a parallelization version of RA* [7]. PRA* distributes work among processors using a state hash function. In particular, the hash function maps each state generated to a corresponding processor. Notice that in PRA*, each processor maintains its own (local) open and closed lists. The open list stores the states that have been generated but not yet expanded, while the closed list keeps the expanded states to detect duplicates. PRA* has a significant synchronization overhead since some processors have to wait for others to reach the synchronization point. For example, when a processor P generates and sends a new state to processor R, P is blocked until it receives a confirmation message from R. This mechanism is required since PRA* is implemented on a connection machine with a limited amount of local memory. PRA* uses a retraction mechanism to remove nodes from memory when needed.

Transposition-table driven work scheduling (TDS) [23, 24] is a hash-based parallelization of IDA* [18] with a distributed transposition table. Like PRA*, TDS distributes work using a state hash function. In particular, TDS distributes a transposition table among processors instead of open and closed lists like PRA*. Transposition tables detect and prune duplicate states in TDS. Unlike PRA*, TDS has no synchronization overhead since it is a fully asynchronous algorithm.

Massively parallel heuristic search (MR-search) [26] is a parallel heuristic framework based on the MapReduce paradigm. MR-search uses all available computing resources (processors, memory, and disks) with its search strategies, breadth-first frontier search, and breadth-first iterative deepening A*. The first search strategy builds large pattern databases to guide the second search strategy.

Notice that PRA*, TDS, as well as MR-search, are parallel algorithms that consider problem-dependent heuristics to guide their search strategies. For example, they have been evaluated in the 15-puzzle and 24-puzzle problems using distance heuristics derivatives. On the contrary, the proposed parallel algorithms of this paper are implemented on the top of a domain-independent planning system, which solves any problem specified with the Planning Domain Definition Language (PDDL) standard [8] using heuristic abstractions. The following works consider the same design.

Hash-Distributed A* (HDA*) [17] is a parallelization of the A* algorithm [10] that asynchronously distributes and schedules work among processors based on a hash function of the search state. In particular, HDA* is an algorithm that combines the hash-based work distribution strategy of PRA* and the asynchronous communication

[1] http://www.icaps-conference.org/index.php/Main/Competitions.

of TDS. The algorithm is implemented on top of the Fast Downward domain-independent planner [11]. As opposed to PRA*, HDA* does not incorporate a node retraction mechanism. More recent works on HDA* have focused on improving the hash function of HDA* that asynchronously distributes work [14–16]. They concentrate on increasing the speedups of the HDA* algorithm by reducing node transfers and by mitigating communication overhead using abstract Zobrist hashing methods.

Parallel Best-NBlock-First (PBNF) [3] is a parallel algorithm that uses abstractions to partition the search space, detecting duplicate information when threads expand the most promising nodes. Threads in the algorithm can also perform speculative expansions by continuing the expansion of their current search spaces. With this technique, threads will always be busy. PBNF is implemented on top of a regression state-space planner.

3 Best first search preliminaries

Best-first search is an instance of the general graph and tree search algorithms that selects the next node for expansion based on an evaluation function, so it is an informed search strategy [25]. The algorithm uses a priority queue to store the search nodes because it needs access to the best node, given the evaluation function, for expansion. A priority queue is usually implemented with Heaps, a complex but efficient tree-based data structure that provides access to the object with the highest (or lowest) priority in $O(1)$ time. Once the best node is selected, each applicable operator, e.g., action, generates children nodes, which are inserted back into the priority queue, insertions take $O(\log n)$ time where n is the size of queue. The algorithm keeps selecting and expanding search nodes until a goal state is found.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
1: PriorityQueue q;
2: Table d;
3: // First state of Global ordered list q
4: State s;
5: q.insert(i);
6: While True
7:    s = q.Remove();
8:    d.insert(s);
9:    Foreach child v of state s
10:      v_h = EVALFN(v, g);
11:      If v_h == 0
12:         return v;
13:      If v ∉ d (not a duplicate)
14:         q.insertorderly(v);
15: End
```

Best-first search has been widely used in the context of planning. Planning, in its classical definition, involves satisficing a goal state from an initial state through a series of action refinements. Notice the direct correspondence to the general description of best-first. Many of the most awarded planning systems base their implementations on best-first search derivatives [9, 11, 13, 27], introducing differences in how they compute heuristic estimates to guide the search process.

Algorithm 1 summarizes best-first search. The algorithm requires the initial state i of the problem as well as the goal state g that needs to be satisfied. The algorithm creates an empty priority queue q and an empty table d of visited nodes. The priority queue is used to select nodes for expansion, while table d is checked for duplicates. Initially, the priority queue only contains the initial state i.

The main loop of the general best-first algorithm keeps removing the best node s from the priority queue until a child state that satisfies the goal is found. The current selected state s is inserted in table d to avoid revisiting it later during the search process. The algorithm uses the available planning actions to generate children states v for s and inserts them in ascending order in the priority queue q according to the value v_h of the evaluation function **EVALFN**(v, g), only if these children are not duplicates in d.

Notice that the evaluation function **EVALFN**(v, g) is given by a complex iterative procedure that evaluates the cost of future actions to reach goal state g from any state v during the search. Given that we construct our proposed parallel solutions on the top of LPG-td, we are bound to the heuristic estimates the planner computes. The heuristic, described in [9], considers reachability information from temporal action graphs to weigh the elements of the search space. The general design process to construct the proposed parallel solutions of this work defines the methods to access asynchronously, via mutual exclusion, both data structures q and d to control search. The following section describes the details of the proposed parallel algorithms.

4 Parallel best-first search algorithms

During more than 50 years, the improvements in technology have led to a sustained increase in the capacity of integration of dense digital circuits. This was put forward by the famous Moore law. This fact, combined with the race for high-performance computing, has led to the predominance of parallel architectures in modern computing. Memory bandwidth has also increased, pushing the limits of memory bound applications. Today, processors like IBM Power9 are multi-core systems with up to 48 computing cores. One can also find computing nodes whereby memory is shared

by two to four multi-core processors. We believe that such multi-core processors and parallel systems are excellent platforms that permit one to revisit classical algorithms for planning problems.

In this section, we propose two parallel algorithms that rely on the same principle. First, we introduce the general principle of the proposed parallel best-first search algorithms, and then we present their details.

4.1 Principle of the proposed parallel best-first search algorithms

We propose a family of asynchronous parallel best-first search algorithms that leverage modern multi-core processors as well as computing nodes with shared memory architectures. The proposed parallel best-first search algorithms maintain a global list and a global table of states. The global list stores the set of states that have been generated but not yet expanded; while the global table stores the expanded states to detect possible duplications.

The parallel best-first search algorithms are multi-threaded methods that generate as many threads as there are computing cores in the system, i.e., one thread per core. Threads expand states asynchronously from the ordered global list. The parallel algorithms are based on the asynchronous work pool paradigm; they maintain good thread occupancy in multi-core CPUs. The approach solves elegantly and efficiently multi-threaded computations in terms of execution time and memory use. The strategy aims at achieving good load balancing; in particular, it tends to keep all threads busy. When a thread T generates a new state v for expansion, it places v in the global list if v is not a duplicate. When T has no work, it retrieves a state from the global list. The accesses to the ordered global list are made via mutual exclusion techniques to avoid the simultaneous use of shared resources by different threads in order to maintain data consistency and efficiency. A good thread occupancy is maintained in our parallel multithreaded algorithms due to the complexity of the planning problems and large number of states to develop before finding a solution. The ordered list of states is sometimes huge, and there is enough work for the parallel threads that run on the computing platform.

In the next subsections, we propose two parallel best-first search algorithms. The first parallel algorithm relies on best-first search with depth equal to one. The second parallel algorithm features a diversification strategy.

4.2 Parallel best-first search algorithm *PBFSD1*

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
1: Global PriorityQueue q;
2: Global Table d;
3: // initial phase
4: Foreach child v of state i
5:    v_h = EVALFN(v, g);
6:    If v_h == 0
7:       return v;
8:    q.insertorderly(v);
9: // First state of Global ordered list q
10: State s;
11: //Start parallel region
12: While True
13:    Mutual exclusion{
14:       s = q.Remove();
15:       d.insert(s);
16:    }
17:    // Produce one generation children v of state s
18:    Foreach child v of state s
19:       v_h = EVALFN(v, g);
20:       If v_h == 0
21:          return v;
22:       Mutual exclusion{
23:          If v ∉ d (not a duplicate)
24:             q.insertorderly(v);
25:       }
26: End
```

The proposed asynchronous parallel algorithms, derived from the general principle given in the previous subsection, differ in the depth of the best-first procedure implemented by each thread and the possibility to introduce some diversification in the search.

In the first method, parallel best-first search and depth equal to one (*PBFSD1*), each thread performs a best-first search with a depth equal to one. Newly created states, resulting from the best-first search procedure, are stored at one time in the ordered global list via mutual exclusion if they are not duplicated. Algorithm 2 displays the pseudo-code for the overall *PBFSD1* algorithm. It begins by the expansion of the initial state i at the head processor after creating the empty global list and global table denoted by q and d, respectively.

Each parallel thread t executes the following loop:

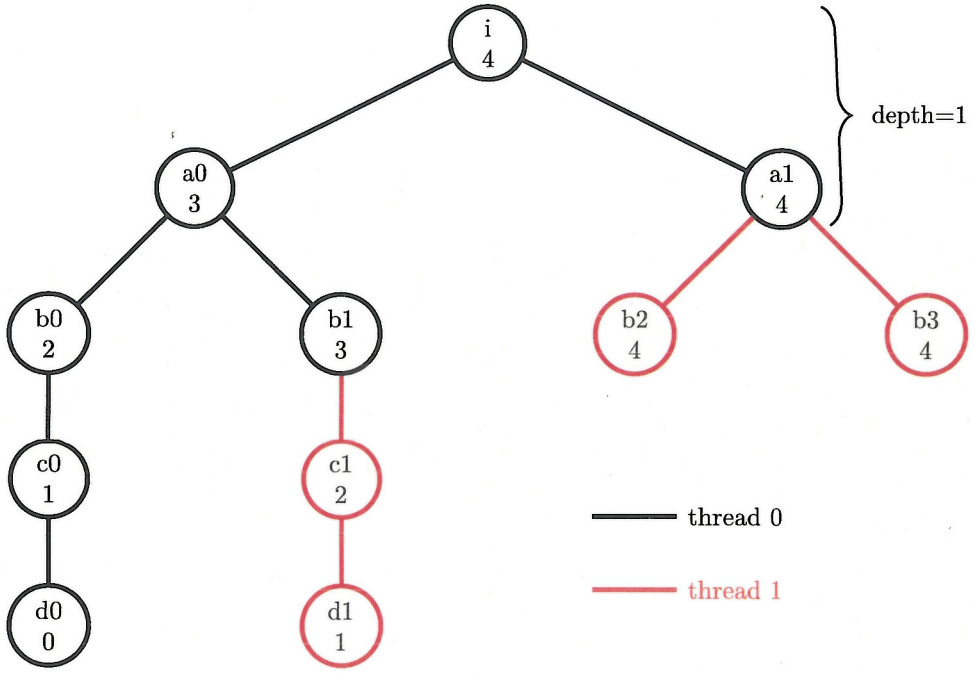


Fig. 1 Example of behavior of parallel best-first search algorithm *PBFSD1*

- Thread t waits until the global list is available, i.e., the list is not locked by another thread.
- Thread t checks if a state is available from the global list. If so, then t retrieves in mutual exclusion the first state s of the global list q, i.e., the state with the smallest value of evaluation function. Then, the following actions are performed in sequence:
  - Store the new state s in the global table d of expanded states.
  - Expand state s, i.e., produce one generation of children of state s. Then, thread t checks the global table d for each of the newly generated child state v to determine whether v is a duplicate or not. If v is not a duplicate, then insert v in ascending order of the evaluation function in the global list q. The writing operation is performed after obtaining a lock on the global list. This lock is released when the writing operation completes. If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and returns it.

To illustrate the parallel best-first search algorithm *PBFSD1*, we consider the example displayed in Fig. 1. For simplicity of presentation, we assume that only two threads are running and that we have a synchronous behavior (which is a special case of the general asynchronous work pool paradigm); moreover, the computing

time of tasks performed by all threads have the same duration, and time spent in the critical sections are negligible.

The parallel algorithm starts with initial state i, whose value of evaluation function is $h = 4$. The global list q initially contains one element, i.e., state i. The algorithm uses the applicable operators of state i to produce one generation of children states. The newly generated children are inserted in the global list q in ascending order based on the value of the evaluation function of each child. At this point, the global list q is $\{a0, a1\}$. State $a0$ is before state $a1$ in the global list since the value of the evaluation function of $a0$ is $h = 3$ that is smaller than the value of the evaluation function of $a1$, which is $h = 4$.

Then, parallel section begins. At each step, each thread retrieves the first state from the global list q, i.e., the state with minimal value of evaluation function h. Each thread performs a best-first search with depth equal to one and process the children of the retrieved state. The children that are not duplicates are stored into the ordered global list q in mutual exclusion. Here, thread0, in black, retrieves state $a0$ and thread1, in red, retrieves state $a1$ from the global list. Then, both threads process children and store them into the ordered list q after checking they are not duplicates. The global list is now $\{b0, b1, b2, b3\}$. Then, thread0 retrieves state $b0$ from the global list q and thread1 retrieves state $b1$ from the global list q. Threads process children and store them back into q, i.e., thread0 adds state $c0$ to q, and thread1 adds state $c1$ to q. The global list q is now $\{c0, c1, b2, b3\}$. Then, thread0 retrieves state $c0$ from the global list q, and thread1 retrieves state $c1$ from the global list q. Once again, threads process children and store them back into q, i.e., thread0 adds state $d0$ to q, and thread1 adds state $d1$ to q. The global list q is now $\{d0, d1, b2, b3\}$. Thread0 which has generated child $d0$ whose value of evaluation function $h = 0$ reaches the goal and returns.

4.3 The parallel search algorithm *PS*

Domain-independent planning systems use heuristics, computed from abstractions and relaxations of the original problem, to traverse their large search spaces. LPG-Td best-first search procedure considers reachability information from relaxed temporal action graphs [9]. One of the effects of the heuristic is that it might localize search. In other words, search exploration using parallel threads might not be diverse enough.

The proposed parallel algorithm *PBFSD1* always picks nodes from the top of the global ordered list. In other words, it selects the best nodes given the heuristics. Such a strategy, which seems reasonable, might not be sufficient if the heuristic estimates are conservative. In this subsection, we propose *PS*, the Parallel Search algorithm that performs a best-first search with diversification. We keep the data

structure of algorithm *PBFSD1*, i.e., the global ordered list of states q, and table d of visited states.

The multiple threads of *PS* algorithm perform randomly either a best-first search with a depth equal to one, like with *PBFSD1* algorithm, or they develop a state situated at 30% of the global list of states q that was generated but not yet expanded. In the latter case, the selected state is expanded along twenty generations according to best-first search principle. This way, we take benefit of another advantage of parallelism, i. e., the possibility to diversify the search concurrently.

Picking by random states that are not the best in the ordered list and developing these states during a convenient number of generations according to best-first search principle has the potential to generate a new best state. Parallel algorithm *PS* is a combination of best-first search and diversification. The probabilities to perform either a best-first search with depth equal to one from the best state or a best-first search with depth equal to twenty from a state situated at thirty percent of the global list q are identical and equal to 0.5. We note that this value, as well as the value of the search depth and the position of the state to expand have been determined empirically. The algorithm *PS* is also implemented according to the asynchronous work pool parallel paradigm.

Algorithm 3 displays the pseudo-code for the *PS* algorithm. It starts with the expansion of the initial state i at the head processor. Then, parallel threads develop either one generation children starting from state s at the beginning of the global ordered list q or twenty generations of children according to best-first search scheme starting from a state t, that is situated at thirty percent of the global ordered list.

To illustrate the parallel best-first algorithm PS, we consider the example displayed in Fig. 2. We assume now that four threads are running. For simplicity of presentation, we assume that we have a synchronous behavior like in the previous example. Moreover, time spent in the critical sections are negligible, and the computing time of tasks performed by all threads have the same duration, but in the case where a thread produces 20 successive generations according to best-first search and starting from a state that is situated at 30% of the global list q. Then, the computing time of that particular thread is more important than in the case where a thread produces only one generation of children states. For clarity of presentation, Fig. 2 displays only one case where a thread produces 20 successive generations.

Algorithm 3: Parallel Search Algorithm *PS*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
1: Global PriorityQueue q;
2: Global Table d;
3: // Initial phase
4: Foreach child v of state i
5:    v_h = EVALFN(v, g);
6:    If v_h == 0
7:       return v;
8:    q.insertorderly(v);
9: // First state of Global ordered list q
10: State s;
11: // State at 30% of Global ordered list q
12: State t;
13: //Start parallel region
14: While True
15:    x_random := generateRandomInteger[1, 100]
16:    If x_random < 50{
17:       Mutual exclusion{
18:          s = q.Remove();
19:          d.insert(s);
20:       }
21:       // Produce one generation children v of state s
22:       Foreach child v of state s
23:          v_h = EVALFN(v, g);
24:          If v_h == 0
25:             return v;
26:          Mutual exclusion{
27:             If v ∉ d (not a duplicate)
28:                q.insertorderly(v);
29:          }
30:    }
31:    else{
32:       Mutual exclusion{
33:          // Point to state t at 30% of the global list q
34:          t = q.Remove();
35:          d.insert(t);
36:       }
37:       //Produce twenty generations descendants v of state t according to
      best-first search
38:       Foreach v
39:          If EVALFN(v, g) == 0
40:             return v;
41:          Mutual exclusion{
42:             If v ∉ d (not a duplicate)
43:                q.insertorderly(v);
44:          }
45:    }
46: End
```

The algorithm *PS* starts with initial state i, whose value of evaluation function is $h = 26$. The global list q initially contains state i. The algorithm uses the applicable operators of state i to produce a global list $q = \{a0, a1\}$, with two states and a list $q = \{b0, b1, b2, b3\}$, with four states, whose associated values of evaluation function are 24, 25, 26, and 26, respectively. Then, parallel section begins. Each thread performs a random test; as a result of the test, thread0, in black, thread1, in red, thread2, in green and thread3, in blue, retrieve in mutual exclusion states $b0$, $b1$, $b2$, and $b3$, respectively, and carry out a best-first search with depth equal to one.

Threads process children states and store them in mutual exclusion into the ordered list q after checking they are not duplicates. The global ordered list q is now $\{c0, c1, c2, \ldots c10, c11\}$. Threads continue to process states in function of the result of the test. We note that after having performed a test, thread3 begins to carry out a best-first search with depth equal to twenty from state *ff* that is situated at 30% of the global list q and whose value of evaluation is equal to 15. In that specific case, the diversification produced by performing a best-first search with depth equal to 20 from a state that is not a state with smallest value of evaluation function permits one to obtain a solution (state 3t0) after less than 20 generations, while thread0, thread1, and thread2 are still stucked in a local minima. Thread3, which generates state 3t0, whose associated value of evaluation function $h = 0$, reaches the goal and returns.

We note that thread3 that performs best-first search with depth equal to 20 produces children states from state *ff* without interfering with other threads, i.e., without having access to global list of states q before twentieth generation.

5 Empirical evaluation

The proposed parallel algorithms were implemented on top of the LPG-td planning system [9]. LPG-td is a domain-independent planner based on stochastic local search, best-first search, and planning graphs. LPG-td won the best-automated planner award in the 2003 IPC and the best performance award in domains with timed literals in 2004.

We present experimental results on selected problems from the International Planning Competition (IPC) benchmark domains[2] and real-world applications from the literature. There are 824 problem instances in total for evaluation from 11 different planning domains. In summary, the results show that the LPG-td sequential algorithm solved 687 problems from the evaluation set (83%), while the *PBFSD1* and the *PS* parallel versions solved 732 (88%) and 737 (89%), respectively.

We calculate the computing time of parallel algorithms via the *gettimeofday()* function, i.e., time spent from the beginning to the end of the computation. We halt the algorithms if no plan is found after 10 min of execution. Note that all parallel algorithms proposed in the paper are asynchronous; therefore, the computing time and the number of actions of solutions for some given difficult instances may differ

[2] https://www.icaps-conference.org/competitions/.

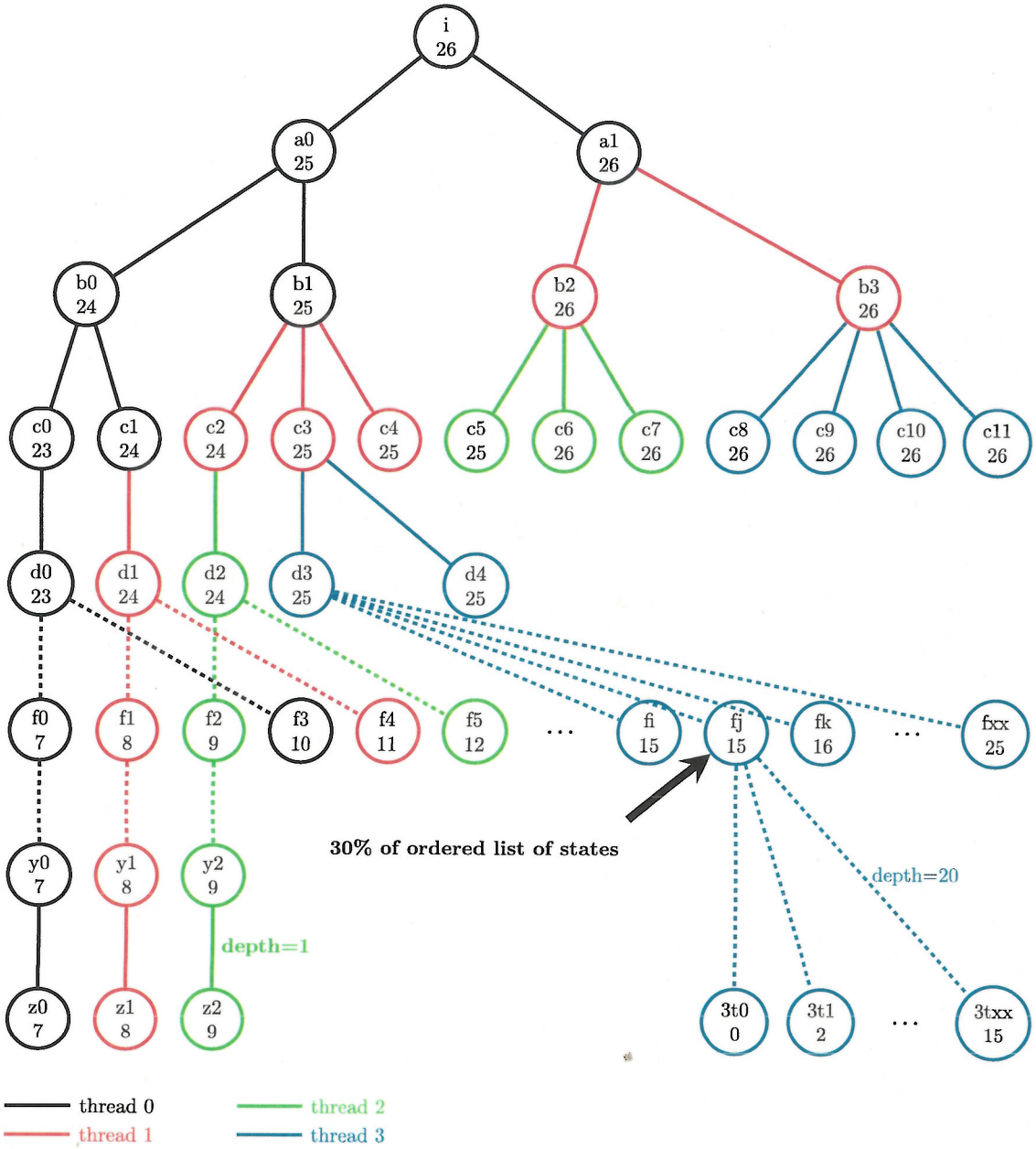


Fig. 2 Example of behavior of parallel best-first search algorithm *PS*

from one run to another. Nevertheless, we display in the sequel results for single runs due to the number of problems treated.

Computational experiments were carried out via OpenMP on a computing node with two CPUs Intel Xeon Gold 6130, with 16 cores, clock 2.10 GHz, and 192 GB of RAM (product collection: Intel Xeon Scalable Processors). The computing node has a total of 32 computing cores. We note that the implementation of our parallel algorithms does not pin threads to given cores; the scheduler of the CPU assigns threads to cores, and assignment can change during a run.

The next subsection presents the analysis and results of the different algorithms in real-world problems. After that, we introduce the evaluation on the IPC benchmark planning domains.

5.1 Real-world application problems

We present three different planning domains from real-world applications. The first couple of problems come from manufacturing production plants. These problems showcase single machine scheduling scenarios that consider maintenance operations. Maintenance operations have sequence-dependent setup costs that have to respect the availability constraints of the production line [2]. The main difference between these two manufacturing domains is the planning of the maintenance operations. In the first model, called *Model-EOL* (*End of Line*), each planning horizon has to end with a maintenance job, while in the second model, such tasks are *External Events* in the production plan (*Model-ExE)*.

The third planning domain solves Public Transportation Network problems (*Model-PTN*). Such problems consider the engineering of travel plans taking into account the public transport and user preferences [6]. Travel plans in public transit networks are time and space-dependent, given that the location of transportation units and users change over time. This characteristic increases the complexity of solving the problem.

5.1.1 Manufacturing models results

Figures 3, 4, and 5 show execution times and solution quality (in terms of number of actions) for the LPG-td and the parallel algorithms (*PBFSD1* and *PS*) for the Model-EOL problems. There are 225 evaluation instances in total, of which LPG-td solves only 46% of them. On the other hand, *PBFSD1* and *PS* return solutions to every problem. Notice that LPG-td does not scale up to problems with 30 tasks (see Fig. 3a). Furthermore, even for the simplest case with ten tasks, the parallel versions are 99% faster than LPG-td. Between the parallel algorithms, *PS* is slightly slower than *PBFSD1* on average. The time difference is more significant in the largest instances (i.e., problems with 30 tasks) where *PBFSD1* is 37% faster than *PS* (Fig. 3c). In addition, LPG-td generates 28 times longer solution plans on average for the simplest case. On the other hand, *PS* returns slightly shorter plans on average for the largest scenarios (Fig. 5).

Figures 6 and 7 show the results of the algorithms, in terms of execution time and number of actions, on the second manufacturing model (i.e., ModelExe with external events). This time, the algorithms solved each instance of the evaluation set. The parallel versions were more efficient than LPG-td, returning globally better quality solutions. While LPG-td can return a solution on 23.21 s on average in the evaluation set, *PBFSD1* takes 14.65 s and *PS* 19.08 s. Therefore, *PBFSD1* is 36% faster than LPG-td, while *PS* is 17% faster as well. Concerning number of actions, the overall solutions returned by *PBFSD1* are 6% shorter than those of LPG-td, while *PS* solutions are also 5% better.

5.1.2 Public transportation model results

We got mixed results in the Public Transportation Domain (see Fig. 8). Although the *PBFSD1* algorithm returns on average shorter plans, both parallel versions are

less efficient than LPG-td. Furthermore, LPG-td solves 99% of the 105 problem instances, while *PBFSD1* solves 88% and *PS* 89% of them. Among both parallel algorithms, *PBFSD1* is 65% faster than *PS*.

5.2 International planning competition (IPC) benchmarks

The next set of problem instances corresponds to problem benchmarks from the international planning competition. There are 269 problem instances from eight different planning domains: OpenStacks, Satellite, Pipes, PSR, Airport, Rovers, Promela, and Pathway.

Figure 9 shows results from the *OpenStacks* planning domain. The OpenStacks domain is based on the simultaneous min–max open stack combinatorial optimization problem. A manufacturer has several orders, each for a combination of different products and can only make one product at a time. In this domain, the parallel algorithms perform consistently better than LPG-td being more than 200 times faster in several scenarios (see Fig. 9a) without losing solution quality (i.e., without increasing the number of actions in their solutions as seen in Fig. 9b).

The next evaluation set corresponds to the *Satellite* domain. This domain considers a set of satellites equipped with different devices that operate under various modes. The objective is to acquire images. Satellites divide the observation tasks considering the capabilities of their instruments. Figure 10a and b show execution times and number of actions, respectively, by the different approaches. Notice that only *PS* can scale up to more problems from the set. *PBFSD1* solves the same first 13 problems as LPG-td and remains competitive in terms of number of actions and execution time.

In the case of the *Pipes-World,* planners control the flow of oil derivatives through a pipeline network, obeying various constraints such as product compatibility, tankage restrictions, and (in the most complex domain version) goal deadlines. LPG-td solves two and three more problems than *PBFSD1* and *PS,* respectively. However, the parallel algorithms perform generally better than LPG-td. *PBFSD1* returns better quality solutions at a fraction of the time taken by the other approaches (see Fig. 11).

The *PSR* domain considers resupplying lines in a faulty electricity network. A transitive closure over the network connections determines its flow of electricity. This flow is subject to the states of the electric supply devices. Figure 12 displays the PSR results. Notice that, in this domain, the parallel algorithms generally return better quality solutions. However, they solve 12% fewer problems than LPG-td on average.

One of the most complex domains for the parallel methods is *Airport* (see Fig. 13). This domain considers the problem of planning ground traffic operations at an airport. The airport scenarios illustrate traffic situations arising during simulation runs in the airport simulation tool Astras [12]. The largest instances in the test suites are realistic encodings of the Munich airport. In this domain, *PBFSD1* and *PS* found plans for 69% of the scenarios, while LPG-td solved 98% of them. However, for difficult instances, parallel algorithm *PS* tends to give better solutions in terms of time and number of actions than LPG-td.

The final three benchmark domains, *Rovers* (Fig. 14), *Promela* (Fig. 15), and *Pathway* (Fig. 16) show inferior results for the parallel algorithms. The Rovers domain models a collection of rovers that must navigate a planet's surface. Rovers must collect samples and communicate data about them to the lander. *PBFSD1* solves 63% and *PS* 55% of the problems, while LPG-td solves all of them. Although the parallel algorithms return equivalent quality solutions like those provided by LPG-td, the time performance for LPG-td is significantly superior.

Promela models deadlocks in communication protocols. Processes that emulate finite-state transition diagrams model the deadlocks. The communication protocols used in Promela consider the dining philosophers problem and the telegraph routing problem. In this set, LPG-td solves 14 scenarios, while *PBFSD1* solves only three (21%) and *PS* nine of them (64%). Again, LPG-td is consistently more efficient than the parallel methods, but all the algorithms return the same quality solutions in terms of number of actions.

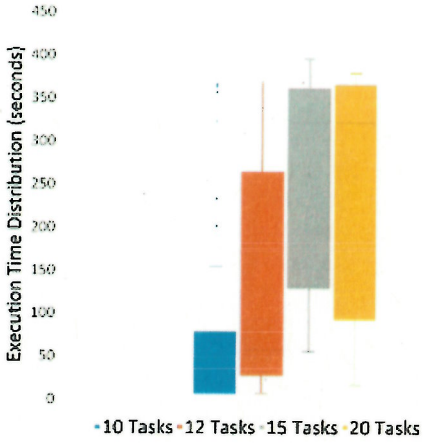
The Pathway domain models biochemical pathways; that is, the chemical reactions in a biological organism that explain cell behavior. The goal consists of synthesizing specific substances in the pathway. This model is the most complex domain evaluated for the parallel algorithms. *PBFSD1* solves only 26% of the problems in the evaluation set, while *PS* returns solutions in 23% of them. LPG-td finds a solution in 93% of the scenarios.

5.3 Analysis and discussion

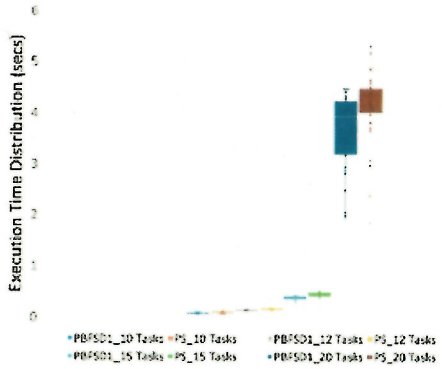
In summary, the parallel algorithms solved more problems from the evaluation set. They returned solutions close to 90% of the evaluation scenarios, while LPG-td found solutions in 83% of the instances. Furthermore, *PBFSD1* needed 224 s sum of averages times to complete almost 90% of the evaluation set, while *PS* required 486 s. In the meantime, LPG-td registered 463 s sum of average times to cover 83% of the scenarios. The good results of parallel algorithms in terms of number of problems solved, computing time and solution quality rely in part on the good thread occupancy. The planning problems are complex (sometimes very complex), and the proposed methods have to develop a large number of states before finding a solution. As a consequence, the ordered list of states is sometimes huge, and there is enough work for the parallel threads that run on the computing platform.

Concerning the real-world problems, the parallel algorithms solved 98% of the instances in this set, while LPG-td returned solutions to 78% of them. A comparison between the parallel methods shows that *PBFSD1* is slightly better than *PS*. *PBFSD1* returns shorter plans on average than *PS* and more efficiently. For example, *PBFSD1* returns plans with 550 actions, while *PS* returns plans with 710 actions on average.

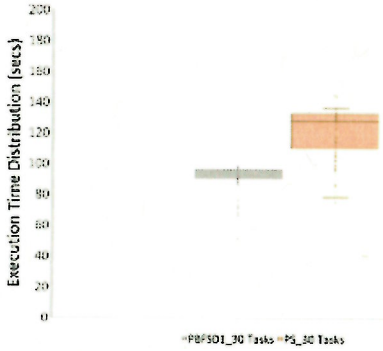
Parallel algorithms got mixed results in the IPC evaluation set. The algorithms performed strongly in 50% of the IPC domains; that is, Pipes (Fig. 11), OpenStacks (Fig. 9), Satellite (Fig. 10), and PSR (Fig. 12). *PBFSD1* solved 89% and *PS* 90% out of the 140 instances. LPG-td solved 93% of the scenarios. Notice that the parallel algorithms tend to return shorter plans more efficiently. *PBFSD1* returns plans with



(a) LPG-td Times for 10 to 20 Tasks



(b) Parallel Execution Times for 10 to 20 Tasks Problems



(c) Parallel Execution Times for 30 Tasks Problems

Fig. 3 Execution times for manufacturing model with EOL maintenance tasks (Model-EOL)

130 actions on average within 55 s of average time. *PS* increases these numbers by delivering solutions with 142 actions on average within 110 s of execution time. On the other hand, LPG-td takes 130 s average time to compute plans with 171 actions.

LPG-td outperforms the proposed methods in the rest of the IPC domains, constituted by Airport, Rovers, Promela, and Pathway. LPG-td solves 94% of the 129 instances in these domains, while *PBFSD1* completes 50% and *PS* 52% of the problems. By looking at the results, it appears that all the algorithms return resembled solutions. For example, we can observe that the algorithms returned plans with the same number of actions in the Promela set (Fig. 15b), which is an indication that the domain is over-constrained. In other words, although the space of the problem appears to grow exponentially (see Fig. 15a), there are not enough solutions to justify parallel search diversification. In this case, exploring alternative branches of the search space might deviate the algorithms from the solutions set, which increases the computation time returning no solutions given the time limits of the experimental design.

Diversification poses a challenge for the parallel methods. We expect that increasing the number of threads helps to explore more of the search space. Tables 1, 2, and 3 display such behavior. The efficiency of parallel algorithms *PBFSD1* and *PS* relies

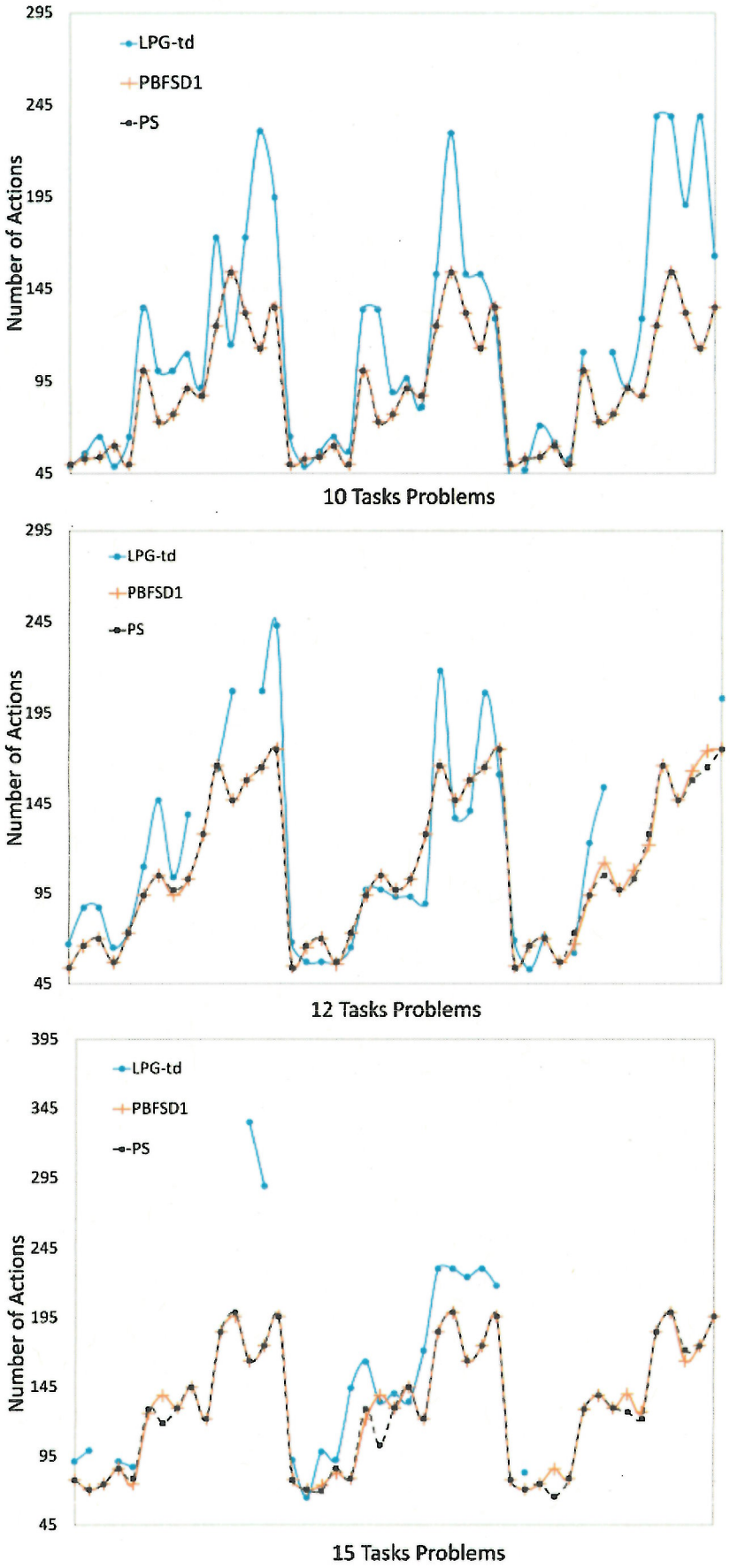


Fig. 4 Number of actions for medium sized instances problems in manufacturing model with EOL maintenance tasks (Model-EOL)

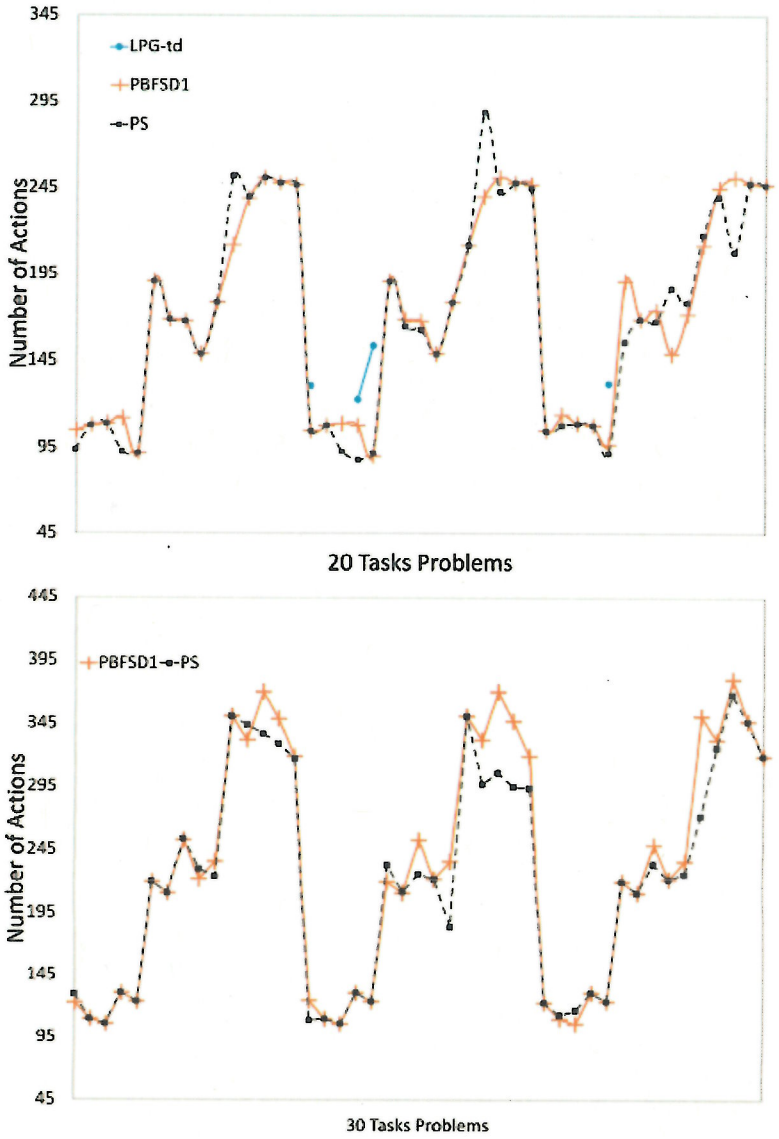
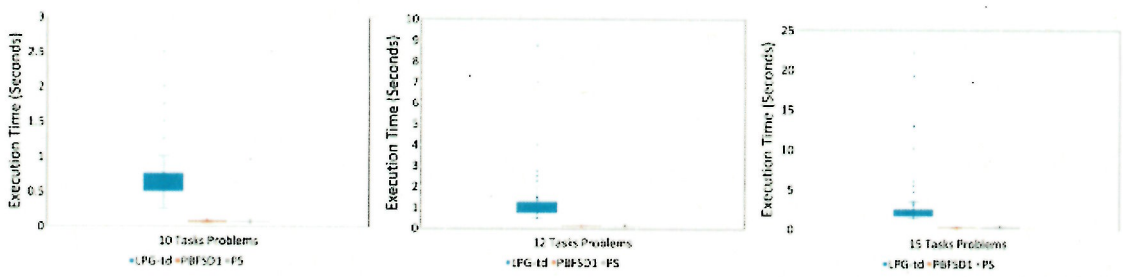
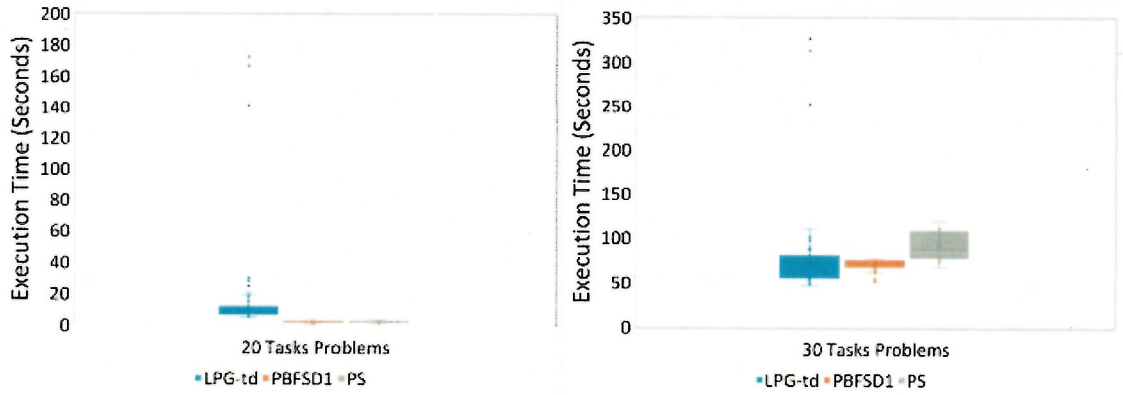


Fig. 5 Number of actions for large sized instances problems in manufacturing model with EOL maintenance tasks (Model-EOL)

mainly on the following features: the asynchronous behavior of the parallel methods; a shared memory implementation with a limited number of cores (a few dozen). As a consequence, waiting time/delays due to atomic operations remain in small in general. Algorithms return better solutions as the number of threads increases. However, as we mentioned for Promela, that might not always be the case. Tables 4 and 5 display the effect of the number of threads on computing time and solution quality on the Airport domain for *PBFSD1* and *PS*. We can point to two observations. It appears that increasing the number of threads generally decreases the computing time. Notice the runs of *PBFSD1* for the P16 problem in Table 4. The algorithm takes 40 s using four threads, reducing its computing time down to 3.8 s when using 32. However, the second observation is that when the solution space is constrained, increasing the number of threads might hurt. That is the case for *PS* in Table 5, where its computation time increases without improving the quality of the solutions

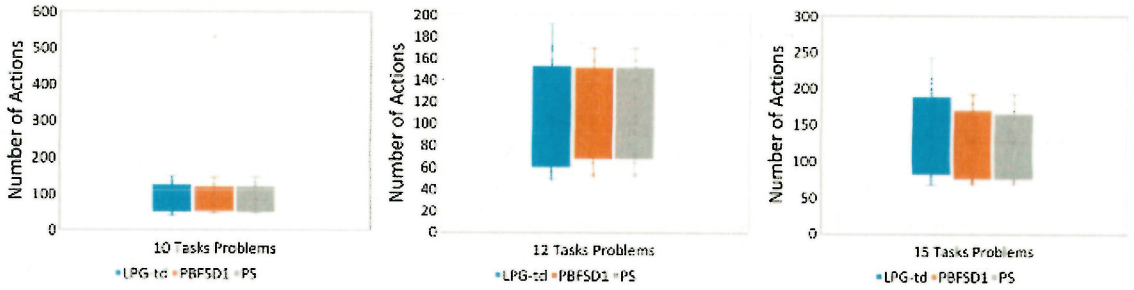


(a) Medium Sized Instances with 10, 12 and 15 Tasks

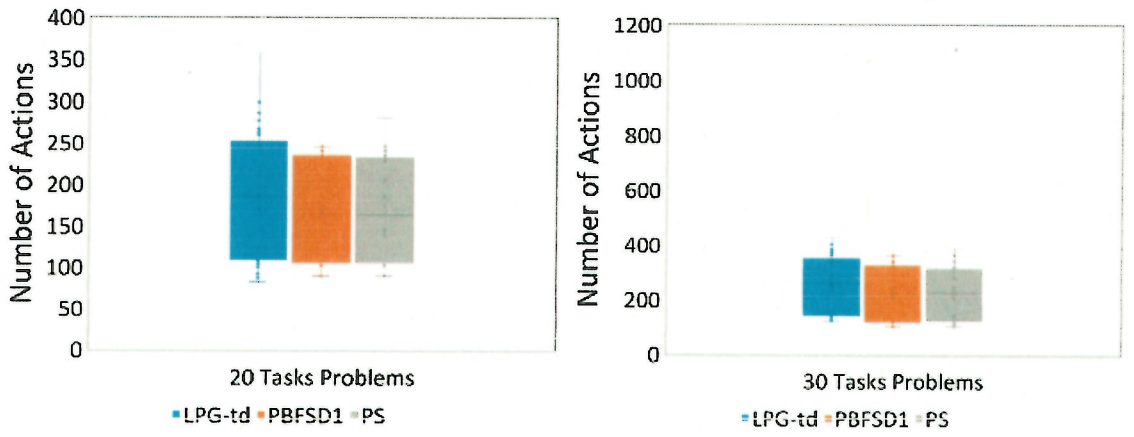


(b) Large Sized instances with 20 and 30 Tasks

Fig. 6 Execution times for model with external events (Model-ExE)

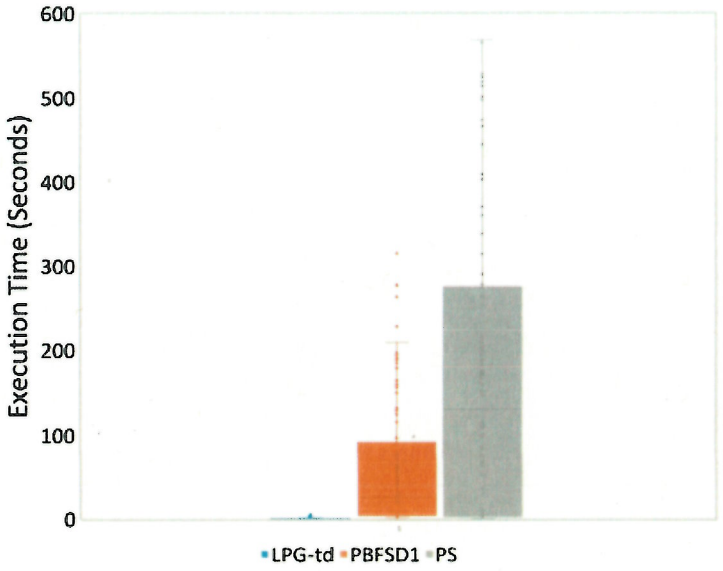


(a) Medium Sized Instances with 10, 12 and 15 Tasks

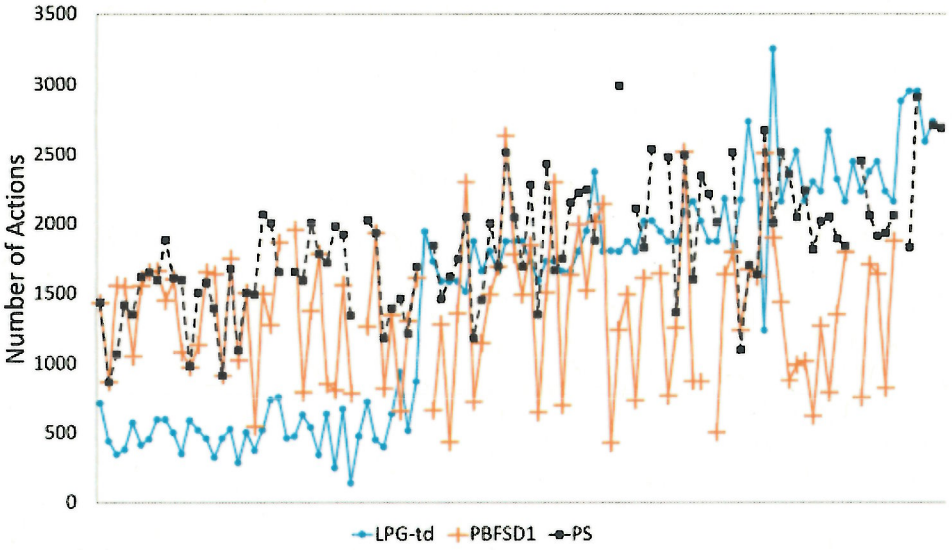


(b) Large Sized instances with 20 and 30 Tasks

Fig. 7 Number of actions for model with external events (Model-ExE)



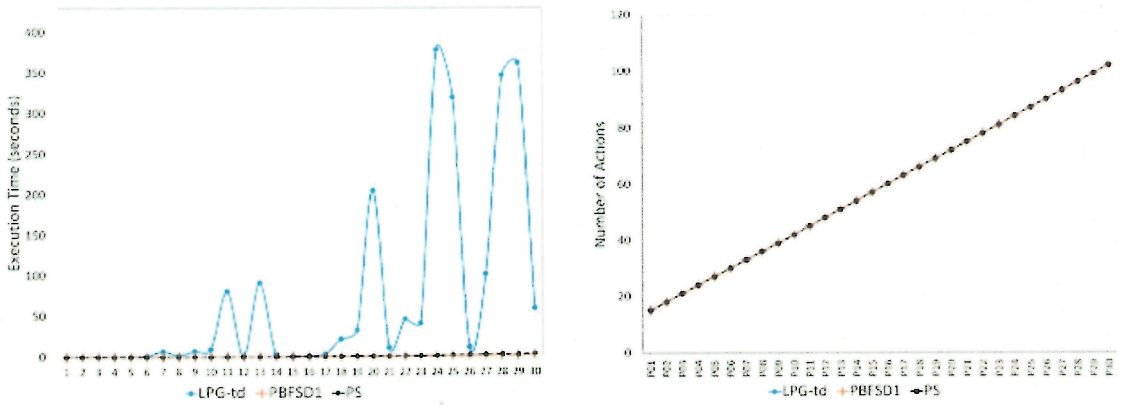
(a) Execution Times in the Public Transportation Domain



(b) Number of Actions in the Public Transportation Domain

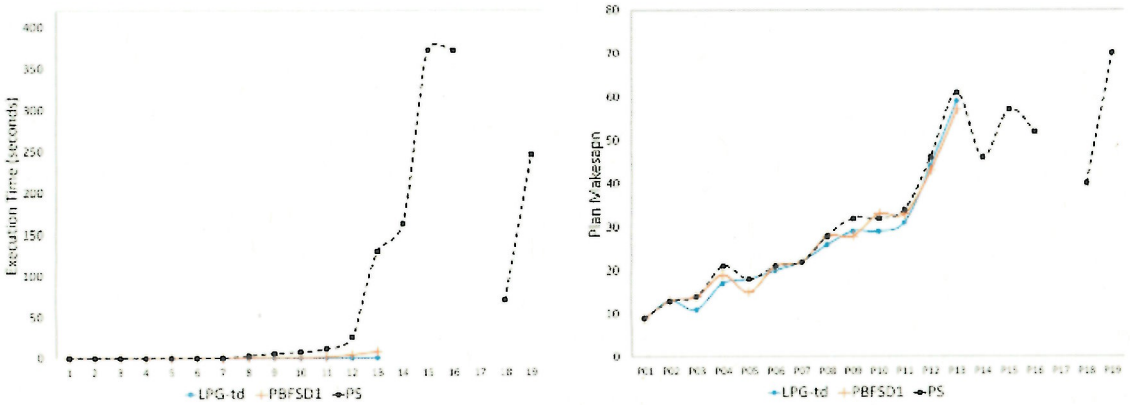
Fig. 8 Public transportation domain results

returned. Therefore, performance is application dependent since the granularity of tasks changes according to the type of planning problem. It is an open research area, for parallel algorithms, to dynamically adjust the resources employed given the plateau of the search space. Recent theoretical studies on analyzing pathological behavior of parallel best-first search could shed some light on how to improve the diversification of such algorithms without incurring higher computational costs [20].



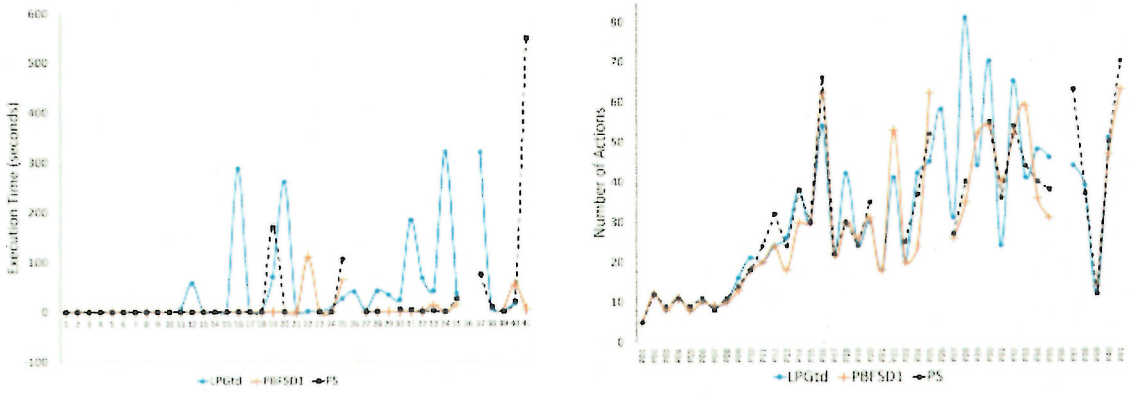
(a) Execution Times in Openstacks (b) Number of Actions in Openstacks

Fig. 9 Openstacks domain results



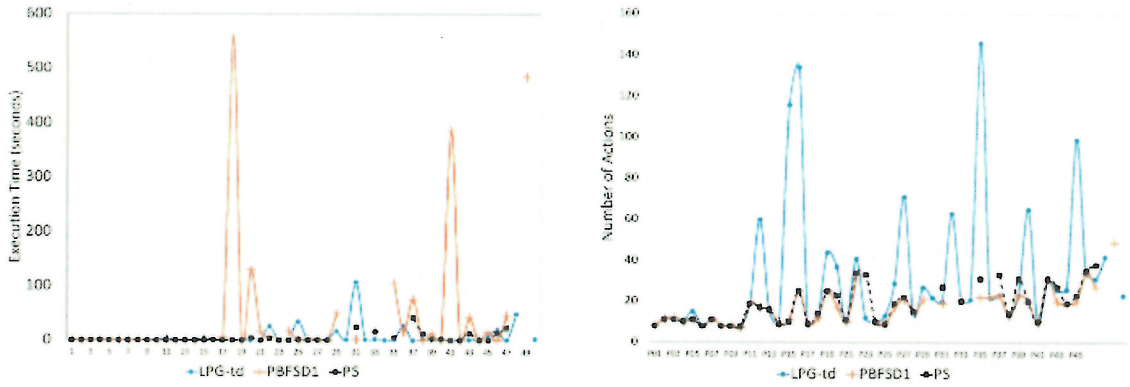
(a) Execution Times in Satellite (b) Number of Actions in Satellite

Fig. 10 Satellite domain results



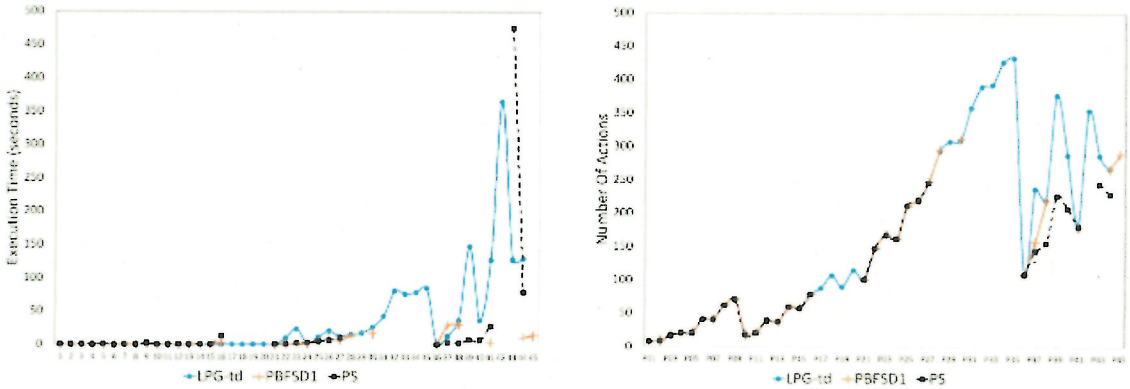
(a) Execution Times in Pipes (b) Number of Actions in Pipes

Fig. 11 Pipes domain results



(a) Execution Times in PSR (b) Number of Actions in PSR

Fig. 12 PSR domain results



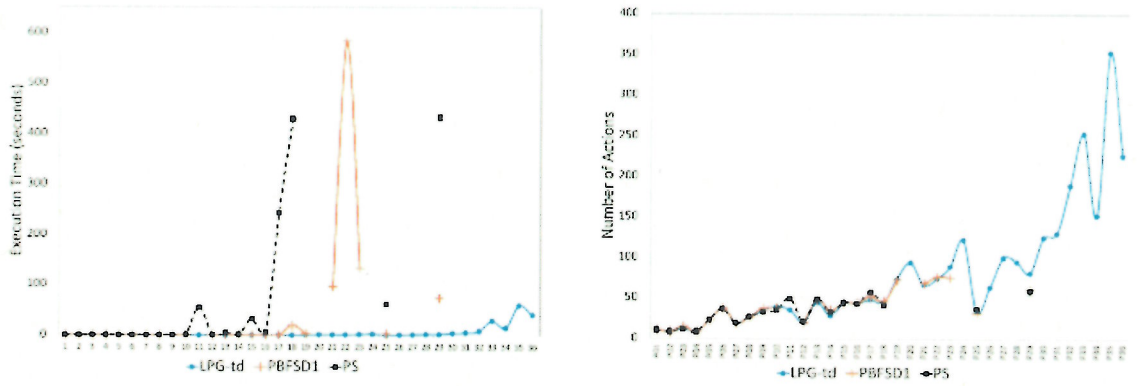
(a) Execution Times in Airport (b) Number of Actions in Airport

Fig. 13 Airport domain results

5.4 Comparisons with other parallel methods

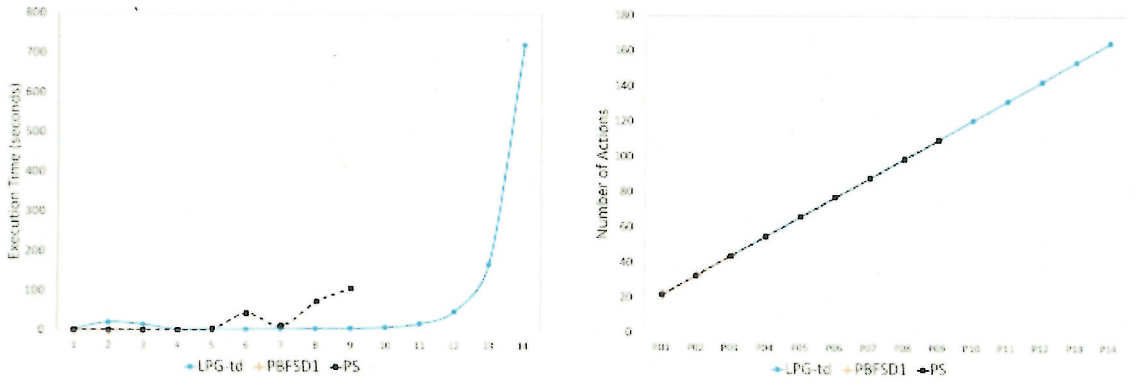
To conclude this section, we note that the computing times obtained with the proposed parallel algorithms *PBFSD1* and *PS* for instances of the Airport and Pipes world problems are fast as compared with those obtained for the same scenarios with other parallel methods like HDA* and PRA* in the literature.

In particular, Kishimoto et al. [17] have considered several implementations of HDA* and PRA* in a multi-core machine with up to 8 cores (2.33 GHz 2× 4-cores Xeon L5410) and a cluster of computing nodes that consists of 2.93 GHz 2× 6-core Xeon X5670. For example, instance P24 of Pipes world problem is solved in 194.96 and 217.21 s by heuristics HDA* and PRA*, respectively, on a multi-core machine (8 cores). The same instance is solved in 5.55 s (best computing time) with HDA* on a cluster with 25 nodes and a total of 300 computing cores. (we recall that we obtain the same solution in 0.54 s with algorithm *PBFSD1* and that parallel algorithm *PS* gives a solution in 0.12). Similarly, instance P27 of Pipes world problem is solved in 103.62 s (best computing time) with HDA* on a cluster with 200 computing nodes and a total of 2400 computing cores (we recall that we obtain the same



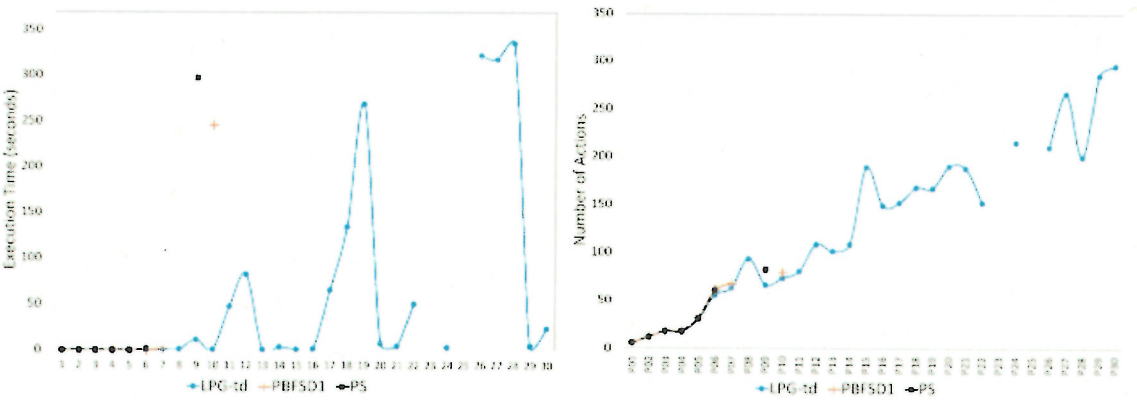
(a) Execution Times in Rovers (b) Number of Actions in Rovers

Fig. 14 Rovers domain results



(a) Execution Times in Promela (b) Number of Actions in Promela

Fig. 15 Promela domain results



(a) Execution Times in Pathway (b) Number of Actions in Pathway

Fig. 16 Pathway domain results

solution in 0.40 s with algorithm *PBFSD1* and that parallel algorithm *PS* gives a solution with 27 actions in 0.26 s).

As mentioned earlier, work by Jinnai and Fukunaga [14–16] concentrate on improving the speedups of the HDA* algorithm by using a set of abstract hashing

Table 1 solution (time secs and number of actions) of instances P24, 27, P31, and P34 of pipes world problem by parallel algorithm *PBFSD1* for 4, 8, 16, and 32 threads and sequential counterpart

Instances	Pipes world problem									
	Sequential		4 threads		8 threads		16 threads		32 threads	
	Time	Actions	Time	Actions	Time	Actions	Time	Actions	Time	Actions
P24	0.04	38	0.10	45	0.51	38	0.054	38	0.54	24
P27	–	–	0.16	52	0.04	28	–	–	0.40	26
P31	–	–	–	–	130.29	65	–	–	0.92	40
P34	0.18	56	0.07	42	0.06	38	0.09	42	0.45	36

Table 2 Solution (time secs and number of actions) of instances P24, 27, P31, P34, and P38 of pipes world problem by algorithm *PS* for 4, 8, 16, and 32 threads and sequential counterpart

Instances	Pipes world problem									
	Sequential		4 threads		8 threads		16 threads		32 threads	
	Time	Actions	Time	Actions	Time	Actions	Time	Actions	Time	Actions
P24	0.40	36	0.05	36	0.11	32	0.34	41	0.12	37
P27	0.17	28	0.31	26	0.61	27	0.15	26	0.26	27
P31	16.09	43	2.24	48	0.67	38	4.2	43	3.57	36
P34	0.43	44	1.73	52	0.28	44	0.94	42	0.58	40
P38	–	–	–	–	–	–	–	–	74.81	63

Table 3 Solution (time secs and number of actions) of several instances of Public Transportation problem by parallel algorithm *PBFSD1* for 4, 8, 16, and 32 threads and sequential counterpart

Instances	Public transportation									
	Sequential		4 threads		8 threads		16 threads		32 threads	
	Time	Quality	Time	Quality	Time	Quality	Time	Quality	Time	Quality
P144-1	0.02	1296	0.69	1033	0.88	1042	1.97	1033	6.47	547
P225-1	0.05	1800	4.19	1751	6.03	1753	10.41	1753	39.17	1609

methods that reduce node transfers and communication overhead. In summary, these works do not present a complete evaluation of planning domains, instead, a set of planning problems are selected for analysis. The evaluation selects the hardest instance in which the general A* algorithm finds a solution. Notice, however, that such an instance might not be the hardest of the evaluation set, as pointed out by the evidence presented in this paper. Authors do not report solution quality; therefore, in the following, we compare only those scenarios solved by them and our approach in terms of solution time.

Table 4 Solution (time secs and number of actions) of instances P15 and P16 of airport problem by parallel algorithm *PBFSD1* for 4, 8, 16, and 32 threads and sequential counterpart

Instances	Airport problem									
	Sequential		4 threads		8 threads		16 threads		32 threads	
	Time	Actions	Time	Actions	Time	Actions	Time	Actions	Time	Actions
P15	0.016	58	0.80	58	0.82	58	0.61	58	0.61	58
P16	110.42	83	40.88	83	17.46	83	7.24	83	3.80	79

Table 5 Solution (time secs and number of actions) of instances P39 and P40 of airport problem by parallel algorithm *PS* for 4, 8, 16, and 32 threads and sequential counterpart

Instances	Airport problem									
	Sequential		4 threads		8 threads		16 threads		32 threads	
	Time	Actions	Time	Actions	Time	Actions	Time	Actions	Time	Actions
P39	–	–	9.93	282	7.42	226	4.74	226	8.02	226
P40	21.06	265	–	–	–	–	5.92	207	7.55	207

In [15], the authors implemented the HDA* variants on the top of the Fast Downward planner [11], contemporary of LPG-td, the planner in which we base our extensions. They solved with the variant FAZHDA*, using a commodity cluster with six nodes and 48 cores, scenario 10 of the Pipes World domain in 9.3s. Our evaluation shows that *PBFSD1* solves the scenario in 0.12s and *PS* in 0.05s using one node and 32 cores. The same problem reports 10s of solution time with GRAZHDA* in [16], under the same hardware configuration. They also report 106.28s and 120.64s solution time for scenario 16 for FAZHDA* and GRAZHDA*, respectively, using a cloud cluster with 128 virtual cores. On the other hand, *PBFSD1* takes 0.46s and *PS* 0.19s for this problem using one node with 32 cores.

Table 6 displays the running time of HDA* variants from Jinnai and Fukunaga in comparison with *PBFSD1* and *PS*. In [15], authors implemented the HDA* variants on the top of the Fast Downward planner [11], contemporary of LPG-td, the planner in which we base our extensions. The hardware configuration consists of a cluster with six nodes and 48 cores, while *PBFSD1* and *PS* use one node with 32 cores. Notice that the three HDA* variants from [15] for solving scenario 10 of the Pipes domain are significantly slower than *PBFSD1* and *PS*. The table also presents scenarios 10, 12, and 15 from Pipes and problem six from the rover domain under the same hardware configuration. These problems are solved by abstract feature generation methods from [16]. Again, *PBFSD1* and *PS* outperform the HDA* variants. The authors also provide a platform based on a cloud cluster with 128 virtual cores. This platform shows the only problem instance that *PBFSD1* and *PS* did not solve, Airport 18. Finally, work [14] introduces pattern database heuristics to solve scenario 14 from Pipes and scenario nine from Airport. *PBFSD1* as well as *PS* are faster than their counterparts.

Table 6 Solution time (seconds) comparison of *PBFSD1* and *PS* with HDA* variants introduced by Jinnai and Fukunaga [14–16]

Configuration							
Cluster with 6 nodes, 48 cores						One node, 32 cores	
Work [15]	Pipes NT 10	A*	FAZHDA*	OZHDA*	AHDA*	PBFSD1	PS
		147.79	9.3	8.19	7.98	0.12	0.05
Work [16]		A*	FAZHDA*	GAZHDA*	GRAZHDA*		
		157.31	10.6	10.1	10		
	Pipes NT 12	A*	DAHDA	ZHDA*	GRAZHDA*	0.21	0.07
		201.07	6.11	9.12	7.71		
	Pipes NT 15	323.59	16.56	15.33	12.85	0.11	0.1
	Rover 6	A	FAZHDA*	GAZHDA*	GRAZHDA*	0.04	0.12
		1042.69	25.76	31.13	25.32		
Work [14]	Pipes NT 14	A*	AZHDA*	AHDA*	ZHDA*	0.12	0.09
		231.5	34.66	28.48	32.77		
	Airport 9	156.59	24.5	27.19	24.22	1.01	2.65
Cloud cluster with 128 virtual cores						0.46	0.19
Work [16]	Pipes NT 16	A*	FAZHDA*	GAZHDA*	GRAZHDA*		
		—	106.28	108.28	120.64		
	Airport 18	—	95.48	128.22	102.34	—	—

Contrarily to these works, we do not distribute states among the different threads/processes, i.e., we do not have distributed local lists of states. In our parallel methods, threads asynchronously and concurrently retrieve states from the same global ordered list, in mutual exclusion. As a consequence, we do not have communication and search overhead. Nevertheless, the access in mutual exclusion to the same global list in shared memory may lead to idle times resulting from shared memory management. These idle times remain negligible if we have dozen of threads and the benefit of parallelism is quite obvious.

6 Conclusions and future work

This paper proposes two original parallel algorithms based on best-first search. The target machines are modern multi-core CPUs like Intel Xeon scalable processors and computing nodes with shared memory architectures. The basic principle of the parallel methods relies on asynchronous updating of an ordered global list of states that is accessed concurrently by multiple threads in mutual exclusion according to the asynchronous work pool paradigm.

The empirical evaluation considers a set of 824 planning problems from real-world applications and the International Planning Competition (IPC). Experiments

are carried out on a node with two processors with a total of 32 computing cores. The experimental results show that parallel algorithms solve up to 7% more problems from the evaluation set than the sequential counterpart in less time. The parallel methods perform strongly in real-world application domains, solving up to 98% of the scenarios while LPG-td finds a solution in 78% of them. Furthermore, parallel methods return shorter plans at a fraction of the time taken by the sequential algorithm.

We got mixed results in the IPC evaluation set. Parallel methods perform strongly in 50% of the IPC set, where they return on average shorter plans more efficiently. On the other hand, LPG-td outperforms the proposed methods in the remaining 129 problems, solving 92% of them versus the overall 50% resolution rate of the parallel algorithms. We notice, given the results, that these instances are over-constrained; that is, the solution space might not be large enough to justify parallelization. This observation is an important finding also because it might help in the future to design pruning techniques on parallel branches to reduce computational overhead.

AI Planning is a hard problem, where large search landscapes are neither categorized nor understood. Planning search spaces greatly vary among applications. This study clearly shows the benefits that can be derived from parallelism and in particular multithreading on modern multi-core processors for solving planning problems. The results are promising. It will enable further research on the potential application of parallel algorithms to domain-independent planning. In particular, the design and development of diversification and pruning techniques for parallel search exploration. In future work, we intend to develop alternative strategies for inserting and removing states from the global processing queues of our algorithms. We require greedy strategies that consider worse quality solutions to escape from local optima. We also plan to port our parallel methods to clusters of multi-core machines as well as Graphics Processing Units (GPUs), considered massively parallel devices.

Funding Part of this study has been made possible via a funding of project BigGraphs of LABEX CIMI 2020, Toulouse and UANL PAICYT Grant IT1838-21.

22. Nguyen X, Kambhampati S, Nigenda RS (2002) Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. Artif Intell 135(1):73–123. https://doi.org/10.1016/S0004-3702(01)00158-6
23. Romein JW, Bal HE, Schaeffer J, Plaat A (2002) A performance analysis of transposition-table-driven work scheduling in distributed search. IEEE Trans Parallel Distrib Syst 13(5):447–459. https://doi.org/10.1109/TPDS.2002.1003855
24. Romein JW, Plaat A, Bal HE, Schaeffer J (1999) Transposition table driven work scheduling in distributed search. In: Proceedings of AAAI/IAAI. pp 725–731. https://www.aaai.org/Papers/AAAI/1999/AAAI99-103.pdf
25. Russell S, Norvig P (2020) Artificial intelligence: a modern approach, 4th edn. Pearson, London
26. Schütt T, Reinefeld A, Maier R (2013) Mr-search: massively parallel heuristic search. Concurr Comput Pract Exp 25(1):40–54. https://doi.org/10.1002/cpe.1833
27. Seipp J, Röger G (2018) Fast downward stone soup 2018 (planner abstract). In: Ninth International Planning Competition (IPC 2018). https://ipc2018-classical.bitbucket.io/planner-abstracts/team45.pdf
28. Shleyfman A, Katz M, Helmert M, Sievers S, Wehrle M (2015) Heuristics and symmetries in classical planning. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015). pp 3371–3377. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/download/9635/9767
29. Vidal V (2004) A lookahead strategy for heuristic search planning. In: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling. pp 150–160. https://aaai.org/Papers/ICAPS/2004/ICAPS04-020.pdf
30. Vidal V, Bordeaux L, Hamadi Y (2010) Adaptive k-parallel best-first search: a simple but efficient algorithm for multi-core domain-independent planning. In: Proceedings of the Third Annual Symposium on Combinatorial Seach (SOC-10). pp 100–107

Authors and Affiliations

Didier El Baz[1] · Bilal Fakih[1] · Romeo Sanchez Nigenda[2] · Vincent Boyer[3]

Didier El Baz
elbaz@laas.fr

Bilal Fakih
bfakih@laas.fr

Vincent Boyer
vincent.boyer@uanl.edu.mx

1 LAAS-CNRS, CNRS, Université de Toulouse, Toulouse, France

2 Facultad de Ingeniería Mecánica y Eléctrica, Universidad Autónoma de Nuevo León, San Nicolás de los Garza, Nuevo León, Mexico

3 Universidad Autónoma de Nuevo León, San Nicolás de los Garza, Mexico

References

1. Arredondo RLG, Sanchez R, Berrones A (2014) Introducing simulated annealing in partial order planning. In: 2014 13th Mexican International Conference on Artificial Intelligence. pp 190–196. https://doi.org/10.1109/MICAI.2014.35
2. Ángel Bello F, Álvarez A, Pacheco J, Martínez I (2011) A single machine scheduling problem with availability constraints and sequence-dependent setup costs. Appl Math Model 35(4):2041–2050. https://doi.org/10.1016/j.apm.2010.11.017
3. Burns E, Lemons S, Ruml W, Zhou R (2010) Best-first heuristic search for multicore machines. Artif Intell Res 39:689–743. https://doi.org/10.1613/jair.3094
4. Bylander T (1994) The computational complexity of propositional strips planning. Artif Intell 69(1):165–204. https://doi.org/10.1016/0004-3702(94)90081-7
5. Coles AI (2007) Heuristics and metaheuristics in forward-chaining planning. Ph.D. thesis, University of Strathclyde
6. Elizalde-Ramírez F, Nigenda RS, Martínez-Salazar IA, Ríos-Solís YA (2019) Travel plans in public transit networks using artificial intelligence planning models. Appl Artif Intell 33(5):440–461. https://doi.org/10.1080/08839514.2019.1582859
7. Evett M, Hendler J, Mahanti A, Nau D (1995) Pra*: Massively parallel heuristic search. J Parallel Distrib Comput 25(2):133–143. https://doi.org/10.1006/jpdc.1995.1036
8. Fox M, Long D (2003) Pddl2.1: An extension to PDDL for expressing temporal planning domains. J Artif Int Res 20(1):61–124. https://doi.org/10.1613/jair.1129
9. Gerevini A, Saetti A, Serina I (2003) Planning through stochastic local search and temporal action graphs in LPG. Artif Intell Res 20:239–290. https://doi.org/10.1613/jair.1183
10. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 4(2):100–107. https://doi.org/10.1109/TSSC.1968.300136
11. Helmert M (2006) The fast downward planning system. J Artif Intell Res 26:191–246. https://doi.org/10.1613/jair.1705
12. Hoffmann J, Edelkamp S, Thiébaux S, Englert R, dos Santos Liporace F, Trüg S (2006) Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. J Artif Intell Res 26(1):453–541. https://doi.org/10.1613/jair.1982
13. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14:253–302. https://doi.org/10.1613/jair.855
14. Jinnai Y, Fukunaga A (2016) Abstract zobrist hashing: an efficient work distribution method for parallel best-first search. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI). pp 717–723. http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13081
15. Jinnai Y, Fukunaga A (2016) Automated creation of efficient work distribution functions for parallel best-first search. In: Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS). pp 184–192. http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13081
16. Jinnai Y, Fukunaga A (2017) On hash-based work distribution methods for parallel best-first search. J Artif Intell Res 60:491–548. https://doi.org/10.1613/jair.5225
17. Kishimoto A, Fukunaga A, Botea A (2013) Evaluation of a simple, scalable, parallel best-first search strategy. Artif Intell 195:222–248. https://doi.org/10.1016/j.artint.2012.10.007
18. Korf RE (1985) Depth-first iterative-deepening: an optimal admissible tree search. Artif Intell 27(1):97–109. https://doi.org/10.1016/0004-3702(85)90084-0
19. Kuroiwa R, Fukunaga A (2019) On the pathological search behavior of distributed greedy best-first search. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling. pp 255–263. https://www.aaai.org/ojs/index.php/ICAPS/article/download/3485/3353
20. Kuroiwa R, Fukunaga A (2020) Analyzing and avoiding pathological behavior in parallel best-first search. In: Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling. pp 175–183. https://ojs.aaai.org//index.php/ICAPS/article/view/6659
21. Nau D, Ghallab M, Traverso P (2004) Automated planning: theory and practice. Morgan Kaufmann Publishers Inc., San Francisco