Parallel AI Planning on the SCC

Vincent Vidal, Simon Vernhes, and Guillaume Infantes

***Abstract*—We present in this paper a parallelized version of an existing Artificial Intelligence automated planner, implemented with standard programming models and tools (hybrid OpenMP/MPI). We then evaluate this planner with respect to its sequential version through extensive experiments over a wide range of academic benchmarks, on two different target architectures: a small standard cluster and the research processor SCC ("Single-chip Cloud Computer") developed by Intel Labs and made available to the research community through the MARC program ("Many-core Applications Research Community"). We obtain interesting speedups (super-linear in some cases) on both architectures. Interestingly enough, these experiments also reveal different behaviors between the cluster and the SCC.**

I. Introduction

Automated Planning in Artificial Intelligence [1] is a general problem solving framework which aims at finding solutions to combinatorial problems formulated with concepts such as actions, states of the world, and goals. For more than 50 years, research in Automated Planning has provided mathematical models, description languages and algorithms to solve this kind of problems. We focus in this paper on Classical Planning, which is one of the simplest model but has seen spectacular improvements in algorithm efficiency during the last decade.

The sequential planning algorithm that will form the basis of our parallel algorithm has been implemented in the YAHSP2 planner [2][3] which participated to the 4th and 7th International Planning Competitions[1] (IPC) in 2004 and 2011. It uses a forward state-space heuristic search algorithm with relaxed plan extraction inspired by the FF planner [4]. The main differences with FF are that (1) the search algorithm is a complete weighted-A* algorithm [5] (while FF first tries an incomplete one), (2) the heuristic function is based on h^{add} [6] instead the length of the relaxed plan length and (3) at each node of the search, a lookahead strategy is performed before classical node expansion in order to try to reach a node closer to a goal state, in a computationally easy way by using actions from the relaxed plan.

The parallelization scheme we propose is based on the principle already used in TDS [7] and HDA* [8]: to distribute search nodes among the processing units (PUs) based on a hash key computed from planning states. In this way, the list of nodes to be expanded (the open list) owned by each PU are disjoint: computations made on a given state (applicable actions, heuristic function, lookaheads, etc.) are performed only once, by the PU the node belongs to. Another important aspect is that communication between PUs can be performed in an asynchronous way: a PU expands nodes from its open list, sends sons to the PUs they belong to, and periodically checks its incoming messages to incorporate new nodes into its open list (between OpenMP threads, this last step is seamlessly performed by writing to shared memory).

We evaluate the performance of the parallel algorithm with respect to its sequential version over a wide range of academic benchmarks issued from the IPCs, on two architectures: a small standard cluster composed of four 12-core servers (48 cores in total), and the research processor SCC ("Single-chip Cloud Computer") embedding 48 cores on a single chip developed by Intel Labs. These experiments show that interesting speedups, sometimes super-linear, are obtained thanks to the parallelization. Unfortunately, some super-linear speed-downs are also observed, which suggests some improvements to the parallelized algorithm that could combine the advantages of both. This behavior was not unexpected, as the parallelized algorithm does not perform the same computations as the sequential version: the order of node evaluation being modified, the search space is not explored the same way, which can help or deserve the parallel algorithm.

The paper is outlined as follows. After introducing the research domain of Classical Planning in Artificial Intelligence and the mathematical STRIPS model of planning, we present the sequential algorithm implemented into the YAHSP2 planner. We then briefly explain the principles of the parallelization we propose, and the main modifications of the sequential algorithm. After having described the experimental evaluation, we conclude and draw some future works.

II. Background on Classical Planning in AI

Classical Planning is about finding a sequence of actions (possibly optimal) leading from an initial state towards a defined goal. We make some assumptions about the world:

- finite number of possible states of the world,
- full observability: one always know the state of the world,
- determinism: the result of applying an action to a state s is always a single state s'.

An example of an Automated Planning problem is described in Figure 1. There, a robot arm can move a single block at a time. It is able to `unstack` two blocks by taking the upper one; `stack` a block on another; `pick-up` a block from the table or `put-down` a block on the table. A planning algorithm should find a plan (a sequence of defined actions) that the robot can execute to reach the goal state from the initial one.

Planning is hard, in our case it has been shown to be PSPACE-complete [9]. The major problem for planning algorithms is to deal with the combinatorial explosion of the number of states during search.

All authors are working at Onera, the French Aerospace Lab, in the DCSD department, Toulouse center. Email addresses: first-name.last-name@onera.fr.

This work has been funded by the Onera research program PR-SCC and supported by Intel Labs through a research proposal for working with Intel SCC and the Many-core Applications Research Community (MARC).

[1]See http://ipc.icaps-conference.org/ for more information about the IPCs.

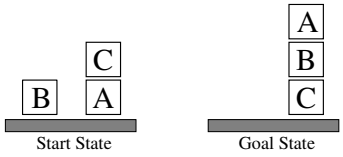

Fig. 1. An Automated Planning classic domain: BlocksWorld

a) PDDL (Planning Domain Definition Language): PDDL [10] is a language commonly used to represent planning problems, as for instance in IPCs. It helps to compare planners with well-established benchmarks[2] (over 40 different application domains and several thousand instances).

The operator `stack` of the previous domain (Figure 1) written using PDDL syntax is shown below:

```
(:action stack
 :parameters  (?ob ?underob)
 :precondition (and (clear ?underob) (holding ?ob))
 :effect (and (arm-empty) (clear ?ob)
          (on ?ob ?underob) (not (clear ?underob))
          (not (holding ?ob))))
```

After parsing a PDDL problem, planners transform the PDDL first-order language into a set-theoretic representation (sets of propositions) like STRIPS (see below). To do so, PDDL operators, like `(stack ?ob ?underob)`, are instantiated into ground actions {`(stack A B)`, `(stack A C)`, ...}.

b) The STRIPS model of Classical Planning: Planning problems can be expressed into the STRIPS model defined as follows. A *state* of the world is represented by a set of ground atoms. A *ground action* a built from a set of atoms A is a tuple $\langle pre(a), add(a), del(a)\rangle$ where $pre(a) \subseteq A$, $add(a) \subseteq A$ and $del(a) \subseteq A$ represent the preconditions, add effects and del effects of a respectively. A *planning problem* can be defined as a tuple $\Pi = \langle A, O, I, G\rangle$, where A is a finite set of *atoms*, O is a finite set of ground actions built from A, $I \subseteq A$ represents the *initial state*, and $G \subseteq A$ represents the *goal* of the problem. The *application* of an action a to a state s is possible if and only if $pre(a) \subseteq s$ and the resulting state is $s' = (s \setminus del(a)) \cup add(a)$. A *solution plan* is a sequence of actions $\langle a_1, \ldots, a_n\rangle$ such that for $s_0 = I$ and for all $i \in \{1, \ldots, n\}$, the intermediate states $s_i = (s_{i-1} \setminus del(a_i)) \cup add(a_i)$ are such that $pre(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$.

c) Prior work on Automated Planning: Different approaches exist [1]. One of the most successful for suboptimal planning is state-space search where each node corresponds to a state of the world and edges between nodes are applicable actions which allow to move from a state s to a state s' (state transition). Finding a path from the initial state I (node) to the goal state G provides a plan for a problem. Heuristic search algorithms like A* are mainly used to find such a path. Various domain-independent heuristics have been developed to guide search. Many successful state-of-the-art sequential planners are based on Fast Downward [11].

Several approaches to parallel planning have been proposed in recent years. Most of them are modifications of the A* algorithm, trying to transform sequential planning techniques into parallel ones. Some algorithms use a distributed hash function to allocate generated states to a unique processing unit and avoid unnecessary state duplications, like HDA* [8]. Parallel Frontier A* with Delayed Duplicate Detection [12] uses a strategy based on intervals computed by sampling to distribute the workload among several workstations, targeting distributed-memory systems. The Adaptive K-Parallel Best-First Search [13] algorithm presents an asynchronous parallel search for multi-core architectures. This paper also provides a recent bibliography about parallel planning. In the IPC 2011 competition, a multi-core track has been started. The most efficient planners were the ones using a portfolio-based approach, meaning they run different planners (or the same planner with different configurations) on each processor (or core) like ArvandHerd [14] and ay-Also-Plan Threaded [15].

[2]The benchmark problems used in past planning competitions are all available on the IPC webpages

Algorithm 1: plan-search

input : a planning problem $\Pi = \langle A, O, I, G\rangle$ and a weight ω for the heuristic function
output : a plan if search succeeds, $\perp$ otherwise

```
1  open ← closed ← ∅
2  create a new node n:
3    n.state ← I
4    n.parent ← ⊥
5    n.steps ← ⟨⟩
6    n.length ← 0
7  n' ← compute-node(Π, ω, n, open, closed)
8  if n' ≠ ⊥ then return extract-plan(n')
9  else
10   while open ≠ ∅ do
11     n ← arg min_{n∈open} n.heuristic
12     open ← open \ {n}
13     foreach a ∈ n.applicable do
14       create a new node n':
15         n'.state ← (n.state \ del(a)) ∪ add(a)
16         n'.parent ← n
17         n'.steps ← ⟨a⟩
18         n'.length ← n.length + 1
19       n'' ← compute-node(Π, ω, n', open, closed)
20       if n'' ≠ ⊥ then return extract-plan(n'')
21   return ⊥
```

III. THE SEQUENTIAL PLANNING ALGORITHM

Algorithm 1 (`plan-search`) constitutes the core of the best-first search algorithm (a weighted-A* here). The first call to `compute-node` may find a solution to the problem without search, by recursive calls to the lookahead process. If not, nodes are extracted from the open list following their heuristic evaluation and are expanded with the applicable actions (already computed and stored in nodes inserted into the open list), and a solution plan is returned as soon as possible. Search can be pursued in an anytime way, in order to improve the solution, with pruning of partial plans whose quality is lower than that of the best plan found so far. In our experiments, the weight ω has been set to 3.

Algorithm 2 (`compute-node`) first performs duplicate state detection. It then computes the heuristic, checks if the goal is obtained or cannot be reached, and updates the node with the heuristic and the applicable actions given by `compute-hadd`. The node is then stored in the open list and a lookahead

Algorithm 2: compute-node

input : a planning problem $\Pi = \langle A, O, I, G \rangle$, a weight ω for the heuristic function, a node n, the *open* and *closed* lists
output : a goal node if search succeeds, $\perp$ otherwise; *open* and *closed* are updated

```
1  if ∃ n' ∈ closed | n'.state = n.state then return ⊥
2  else
3      closed ← closed ∪ {n}
4      ⟨cost, app⟩ ← compute-hadd(Π, n.state)
5      gcost ← Σ_{g∈G} cost[g]
6      if gcost = 0 then return n
7      else if gcost = ∞ then return ⊥
8      else
9          n.applicable ← app
10         n.heuristic ← n.length + ω × gcost
11         open ← open ∪ {n}
12         ⟨state, plan⟩ ← lookahead(Π, n.state, cost)
13         create a new node n':
14             n'.state ← state
15             n'.parent ← n
16             n'.steps ← plan
17             n'.length ← n.length + length(plan)
18         return compute-node(Π, ω, n', open, closed)
```

state/plan is computed by a call to `lookahead`. A new node corresponding to the lookahead state is then created and `compute-node` is recursively called. Recursion is stopped when a goal, duplicate or a dead-end state is reached.

The other algorithms are not shown here due to lack of space (more details can be found in [3]), but their role can be summarized as follows. Algorithm `compute-hadd` computes h^{add} and returns a vector of costs for all atoms and actions, as well as actions applicable in the state for which h^{add} is computed obtained as a side-effect. Algorithm `lookahead` computes a lookahead state/plan from a relaxed plan given by a call to `extract-relaxed-plan`. Once a first applicable action of the relaxed plan is encountered, it is appended to the lookahead plan and the lookahead state is updated. A second applicable action is then sought from the beginning of the relaxed plan, and so on. When no applicable action is found, a repair strategy tries to find an applicable action of minimum cost from the whole set of actions, in order to replace an action of the relaxed plan which produces an unsatisfied precondition of another action of the relaxed plan, and the process loops. Algorithm `extract-relaxed-plan` computes a relaxed plan from a vector of action costs. A sequence of goals to produce is maintained, starting from the goals of the problem. The first one is extracted, and an action which produces it with the lowest cost is selected and stored in the relaxed plan. Its preconditions are appended to the sequence of goals, and the process loops until the sequence of goals is empty. An atom already satisfied, i.e. produced by an action of the relaxed plan, is not considered twice. The relaxed plan is finally sorted before being returned, by increasing costs first, and for equal costs by trying to order first an action which does not delete a precondition of the next action.

IV. An hybrid OpenMP/MPI Parallel Planning Algorithm

The main idea for parallelizing YAHSP2 is based on the same principle than in TDS [7] and HDA* [8]: to distribute search nodes among the PUs based on a hash key computed from planning states. A PU can either be an MPI process running a single thread, or an OpenMP thread started with several others by an MPI process.

One important consequence of the hash-based distribution principle is that several occurrences of a given state, encountered in any PU, will be sent to the same PU that will either discard it if it has already been encountered, or expand it in the opposite case. Another consequence is that this communication scheme can be performed in an asynchronous way: PUs send nodes that do not belong to them (i.e. the state hash key identifies another PU), and receive nodes from any other PUs, while expanding nodes they currently own in their open list.

The main differences with respect to TDS and HDA* are that (1) we focus on suboptimal planning, while TDS and HDA* search optimal plans, (2) we have integrated the lookahead strategy into this framework, and (3) we implemented this principle as an hybrid OpenMP/MPI algorithm (while TDS and HDA* only use MPI). The advantages of using OpenMP are that problem parsing, instantiation, and all other preprocessing tasks are performed only once (thus saving memory), and communication between threads by shared memory is much more efficient than communication between MPI processes. The main drawback of using OpenMP is that memory locks are sometimes necessary; but fortunately, this does not happen often because the algorithm spends most of its time in computing the h^{add} heuristic (Algorithm 2 line 4).

Each PU (either an MPI process running a single thread, or an OpenMP thread inside an MPI process) runs the search algorithm described in Algorithm 1, with its own open and closed lists, with several modifications:

The first initial node (lines 2–6) is created only by the master thread of the first MPI process.

The main loop condition (line 10) is modified in order for the loop to be executed even if the PU has no node yet (i.e., it is waiting for states sent by other PUs). This loop is stopped when a PU finds a solution, which will be handled by special messages. We have not yet implemented a distributed termination algorithm in the case where the search space is completely explored without finding a solution (which happens extremely rarely on academic benchmarks). In HDA*, a termination algorithm from [16] has been used.

Calls to `compute-node` (lines 7 and 20) are performed only if the corresponding nodes belong to the current PU; in the opposite case, they are sent to their associated PU. This is performed by either sending a message to another MPI process, or by incorporating the node into the open list of another thread within the same MPI process. In the latter case, it is required to lock the open list of the destination thread.

Before choosing the next node to be expanded (line 11), incoming MPI messages are checked and all new nodes are incorporated into the open list: either the open list of the current PU, or into the open list of another thread of the same MPI process –which requires once again a lock on this list.

In order to completely follow the hash-based distribution principle, Algorithm 2 should also be modified in order to send nodes to their appropriate PUs (as in the third point above): line 18 should be executed only in the case where the obtained

node belongs to the current PU, and if not, this node should be sent to its correct destination. However, after some preliminary experiments, we observed that the strategy of performing full lookaheads –i.e. the full recursive calls of `compute-node` inside a single PU– was more efficient than distributing them. One consequence is that a given node may appear in different PUs, thus duplicating the work of expanding it. Many other variations and strategies can be imagined, and the description and comparison of various node distribution policies will be the subject of a more extensive study.

The last main modification of the sequential algorithm is about the reconstruction of the solution, which is distributed among the different PUs. Indeed, when nodes are communicated between PUs, the actions attached to it (which represent the path from a node to its son) are kept in the PU they are computed, in order to minimize the traffic. All messages thus have the same size, as states are represented with bit arrays whose size is the number of ground atoms of the problem, which is determined during the planning problem instantiation. When a PU finds a solution, it sends a special message to all nodes meaning that a synchronization step is required. These messages are checked in the same place than MPI incoming messages are treated (before line 11 of algorithm 1). In the case of synchronization, a function is called by all PUs at this place, into which they exchange messages to build the solution plan and aggregate statistics on the current run (everything being controlled and owned by the master thread of the first MPI process). This procedure was a bit tricky to implement, but do not deserves more details in this paper. One important remark though is that all PUs play exactly the same role in the algorithm, except in two minor cases where the master thread of the first MPI process plays a special role: when search starts (initial node of the problem) and when a solution is built.

V. Experimental Evaluation

In order to evaluate the different parallel implementations of the algorithm, we conducted a set of experiments with 1171 benchmarks from the 3rd to the 7th IPCs (all sequential problems from these IPCs). The cluster is composed of 4 servers with two 6-core Intel Xeon X5670 running at 2.93GHz and 24GB of RAM. The different configurations are:

- c1: 1 process with the sequential algorithm;
- c48: 48 MPI processes uniformly distributed;
- c4x12: 4 MPI processes, each one of them including 12 OpenMP threads, uniformly distributed (cluster only);
- c48-I and c4x12-I: 48 independent MPI processes or 4 MPI processes including 12 independent threads (cluster only) executing the sequential algorithm in exactly the same way with no communication (all PUs are equivalent to configuration c1 and perform identical computations on the same data), in order to assess the impact of memory contention.

In the following, we compare absolute (wall-clock) time used for finding a solution with a timeout of 600 sec., and the ratio of search times used by compared versions. This will be shown as *speedup* in the figures. Thus, the shown speedup will be the amount of time used by the quickest implementation divided by the amount of time used by the slowest one.

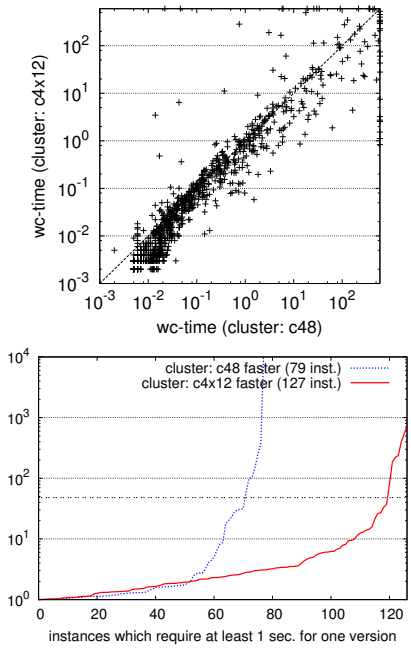

Fig. 2. Comparison of the wall-clock time in seconds between cluster versions (either 48 MPI processes, or 4 MPI processes of 12 threads each) and speedup of both versions (Blue curve –dotted lines– represent the speedup of the 48 processes version when it is faster, while red curves –plain lines– represent the speedup of the 4x12 version when it is faster.).

a) Cluster versions compared: Figure 2 compares c48 and c4x12 on the cluster. On the top figure, the solving time is compared, problem by problem. As most of the points are on the bottom right part, we can deduce that c48 performs worse. While the number of threads is the same, the overhead caused by the MPI message passing mechanism makes this version generally worse than c4x12. In further experiments, we then will only compare c4x12 to c48 on the SCC. The bottom figure shows the speedup of both versions, for the problems were the particular version performs better. Again, it can be seen that c4x12 performs better in a larger number of cases than c48. Interestingly, when problems become harder, the speedup can become extremely large: one of the versions typically go around 10000 times faster than the other one.

b) Parallel vs. sequential: On Figure 3 can be seen the comparison between the sequential version and the parallel implementation. Hopefully, the parallel version performs better than the sequential one as soon as the problem is complex enough to take more time to solve than the overhead induced by communication mechanisms. Another reason for the parallel version not to always perform better is that the order in which nodes are explored is not the same, and the aim of the heuristic used is to make the sequential version use a very good order, while in parallel version there is much more variation around the order implied by the use of the heuristic value. One can also remark a very large number of problems unsolved before timeout for the sequential version, especially on the SCC: they are the many points on the right frame.

c) Detailed speedup analysis: Figure 4 shows the comparison between c4x12 and c1 on the cluster. We show different curves in order to emphasize the effect of the com-

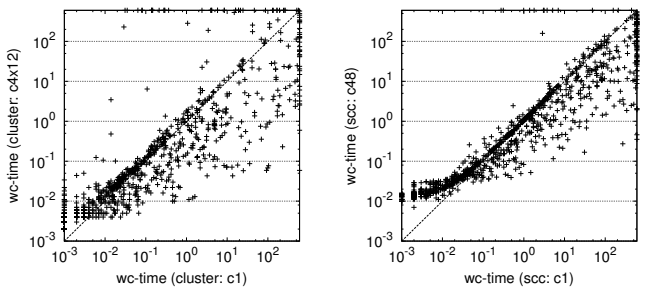

Fig. 3. Comparison of the wall-clock time in second between parallel (4x12 on the cluster and 48 processes on the SCC) and sequential versions.

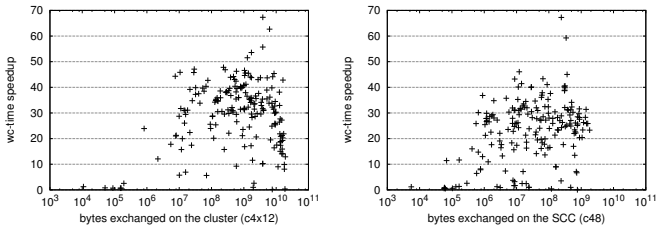

Fig. 6. Wall-clock time speedup of parallel algorithms vs. the sequential version in function of the total number of bytes exchanged between all processes (between the 4 MPI processes) for anytime runs of 100 seconds.

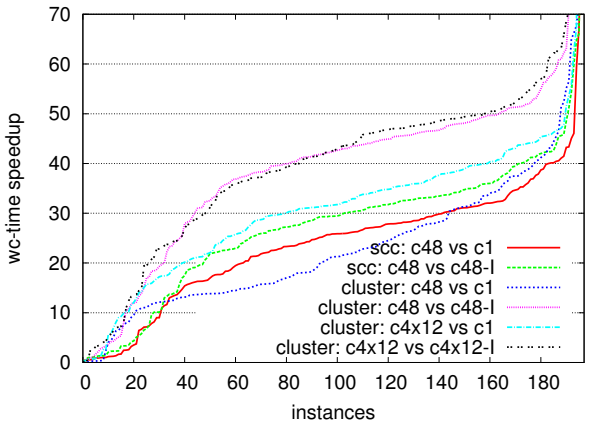

Fig. 7. Wall-clock time speedup in node generation of parallel algorithms vs. sequential versions for anytime runs of 100 seconds, on the cluster and on the SCC. Curves labelled by "architecture: x vs y" compare on the given architecture the speedup of running x processes versus running y processes (single process or 48 non-communicating processes running the same instance in the same way, or 4 non-communicating MPI processes of 12 OpenMP threads each also running the same way).

munication overhead, independently of the parallelism itself; this because communication overhead has a large influence for very simple problems, and becomes less important for larger ones. So we show results for problems taking at least 0.001 second to solve for one of the version, 0.1 second to solve and so on. As expected, the sequential version performs better for problems that can be very quickly solved, but the parallel version becomes generally better as soon as the problems need at least 0.1 second to be solved. On the other hand, this trend becomes less obvious when the problems are very complex (more than 100 seconds to be solved for one version). We think that this is because both versions get trapped into long useless explorations that do not lead to find the goal.

We conducted the very same comparison on the SCC, as shown on Figure 5. Interestingly, the trend observed on the cluster for the larger problems (that the parallel version does not perform better and better compared to the sequential one) is not present here (even with comparable complexity obtained by comparing 30 sec. of cluster time with 600 sec. of SCC time –not shown here–). So the SCC parallel version performs better and better with the problem complexity, whereas the cluster version just performs better, but not better and better.

At this point, we are unsure why this occurs. One explanation is that the amount of data exchanged increases super-linearly with the problem complexity, thus the SCC implementation would be less sensitive to the problem complexity. It may also be the case that all threads being trapped into bad explorations may occur only for a larger timeout…

d) Influence of the amount of data exchanged: In order to figure this out, we present Figure 6, where one can see the speedups related to the amount of data exchanged. This is performed on a selection of 5 problems in each planning domain (210 instances in total), for anytime runs of 100 seconds (search continues after a solution is found, producing solutions of increased quality). In the cluster version, there is a clear trend of worse speedup when the amount of data exchanged increases, whereas there is no correlation for the SCC. Indeed on the cluster, for the instances where the exchanges are about 10GB (nearly 800 Mb/sec) we seems to reach the I/O capacities (1Gib/sec). This seems to be a good explanation of the "less-sensitivity" to the problem complexity of the SCC implementation compared to the 4x12 cluster one.

e) Influence of concurrent resources access: Finally, we present as Figure 7 the speedups in node generation of parallel implementations relative to one sequential process, but also to the same number of sequential processes, in order to see the speedup obtained with a comparable bottleneck for memory access. This is performed in the same experimental conditions as in the previous experiment (anytime search on 210 problems during 100 seconds).

For small instances, the speedup can be small due to the overhead of message passing, while for larger instances, the complexity of problems causes the sequential algorithm to get trapped into exploring non-interesting states for a very long time, making very large speedups. This shows that for complex problems the sequential algorithm would perform better simply by avoiding such traps. More interestingly, the "center" part of the curves, for average instances show very large differences between the SCC and the cluster implementations. More precisely, there is a large difference between the "cluster: c48 vs c1" and the "cluster: c48 vs c48-I" curves (same for "c4x12" versions) meaning that on the cluster there is a lot of memory contention (c48-I is a lot less efficient than c1: only one process). This is less the case for the SCC versions: on the SCC, the sequential non-communicating processes almost do not slow down each other. Several conclusions can thus be stated: on a cluster implementation, good cooperation is mandatory in order to achieve large speedups, in order to reduce memory usage of each core. On the other hand, our

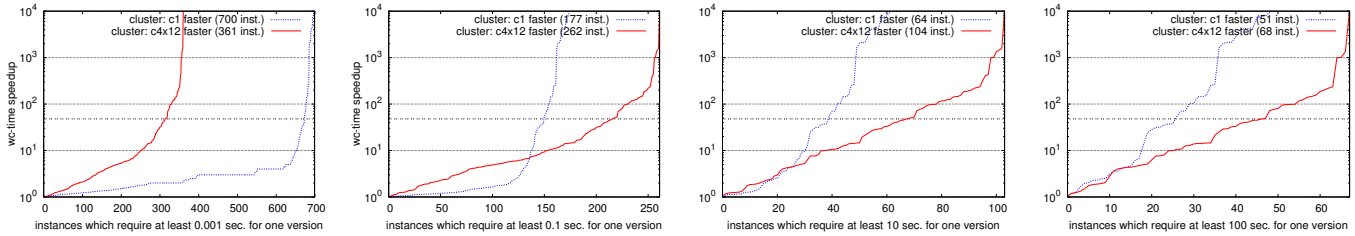

Fig. 4. Wall-clock time speedup of the parallel algorithm with 4 MPI threads of 12 threads each vs. the sequential version running on the cluster, for all instances which require at least a given number of seconds (see x-axis) for one version.

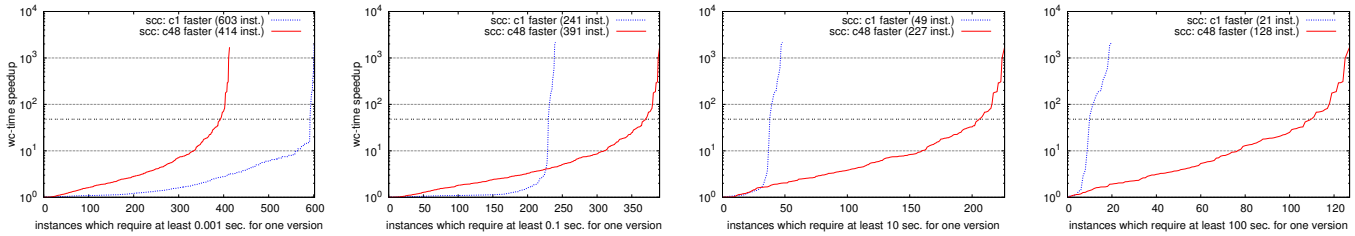

Fig. 5. Wall-clock time speedup of the parallel algorithm with 48 MPI threads vs. the sequential version running on the SCC, for all instances which require at least a given number of seconds (see x-axis) for one version.

implementation achieves a good speedup between the "c48 vs c48-I", meaning that the main bottleneck for improving it is the memory contention problem itself (which will be hard to avoid for our algorithm). On the SCC, this is not a problem, so we can either try to improve the communication scheme for more complementarity, or try a very different "portfolio" approach, where the cores are more independent, and try to solve the problem in different ways.

VI. Conclusion

We described in this paper the parallelization of an automated planner based on forward heuristic search and lookaheads for suboptimal sequential classical planning. It is based on a hash-based node distribution, implemented in hybrid OpenMP/MPI. Experiments show performance improvements with respect to the sequential version, especially for difficult problems. As the search space is not explored the same way in the sequential and parallel versions, super-linear speedups are observed, but also super-linear speed-downs. This suggests trivial improvements of the parallel version, for example by running the sequential version on a single processing unit and the parallel algorithm on the remaining processing units. More elaborate strategies can be imagined, that will make the subject of further studies. The experiments also revealed some differences in the behavior of the parallel algorithm on a standard cluster and on the SCC. These differences suggest that improvements of the parallel version may be more beneficial to an execution on the SCC (which suffers less from memory contention and benefits from faster communications), but clearly more in-depth studies are needed to understand these differences in order to better take advantage of the capabilities of the SCC.

Acknowledgment

The authors would like to thank Intel Labs for providing access to the SCC, and for their reactivity in solving all problems that arose during the SCC exploitation. They also thank Eric Noulard from Onera for insightful discussions.

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning, theory and practice*. Morgan-Kaufmann, 2004.
[2] V. Vidal, "A lookahead strategy for heuristic search planning," in *Proc. ICAPS*, 2004, pp. 150–159.
[3] ——, "YAHSP2: Keep it simple, stupid," in *Proc. of the 7th International Planning Competition (IPC'11)*, 2011.
[4] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *JAIR*, vol. 14, pp. 253–302, 2001.
[5] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1, no. 3, pp. 193–204, 1970.
[6] B. Bonet, G. Loerincs, and H. Geffner, "A robust and fast action selection mechanism for planning," in *Proc. AAAI*, 1997, pp. 714–719.
[7] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer, "Transposition table driven work scheduling in distributed search," in *Proc. AAAI*, 1999.
[8] A. Kishimoto, A. S. Fukunaga, and A. Botea, "Scalable, parallel best-first search for optimal sequential planning," in *Proc. ICAPS*, 2009.
[9] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1-2, pp. 165–204, 1994.
[10] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL – The Planning Domain Definition Language," Yale Center for Computational Vision and Control, New Haven, CI, USA, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.
[11] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, no. 1, pp. 191–246, 2006.
[12] R. Niewiadomski, J. N. Amaral, and R. C. Holte, "Sequential and parallel algorithms for frontier a* with delayed duplicate detection," in *Proc. AAAI*, 2006.
[13] V. Vidal, L. Bordeaux, and Y. Hamadi, "Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning," in *Proc. 3rd Symposium on Combinatorial Search (SOCS'10)*, 2010.
[14] R. Valenzano, H. Nakhost, M. Muller, and J. Schaeffer, "Arvandherd: Parallel planning with a portfolio," in *Proc. 7th International Planning Competition (IPC'11)*, 2011.
[15] J. Ernits, C. Gretton, and R. Dearden, "Ay also plan: Bitstate pruning for state-based planning on massively parallel compute clusters," in *Proc. 7th International Planning Competition (IPC'11)*, 2011.
[16] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, 1987.