ECAI 2012
Luc De Raedt et al. (Eds.)

doi:10.3233/978-1-61499-098-7-786

ArvandHerd: Parallel Planning with a Portfolio

Richard Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer[1] and **Nathan Sturtevant**[2]

Abstract. ArvandHerd is a parallel planner that won the multi-core sequential satisficing track of the 2011 International Planning Competition (IPC 2011). It assigns processors to run different members of an algorithm portfolio which contains several configurations of each of two different planners: LAMA-2008 and Arvand. In this paper, we demonstrate that simple techniques for using different planner configurations can significantly improve the coverage of both of these planners. We then show that these two planners, when using multiple configurations, can be combined to construct a high performance parallel planner. In particular, we will show that ArvandHerd can solve more IPC benchmark problems than even a perfect parallelization of LAMA-2011, which won the satisficing track at IPC 2011. We will also show that the coverage of ArvandHerd can be further improved if LAMA-2008 is replaced by LAMA-2011 in the portfolio.

1 Introduction

In recent years, the rate at which processor speed is increasing has curtailed, while the proliferation and availability of multi-core technology has substantially increased. This development suggests that to best utilize modern hardware when constructing automated satisficing planning systems, it is necessary to consider parallel approaches.

Past work on building parallel planning systems, such as PBNF [2] and HDA* [12], has generally focused on parallelizing a single heuristic search algorithm. While these approaches have successfully improved run-time, satisficing planners that use these or similar techniques on shared memory machines should not be expected to solve many more problems than their single-core counterparts. This would be true for even a perfect parallelization of LAMA-2011 [18], the winner of the single-core sequential satisficing track of IPC 2011, that runs exactly k times faster than the single-core version when run on k cores. Given a time limit T, the performance of such a k-core system can be simulated by running LAMA-2011 for $k \cdot T$ time and counting any problem solved within this time limit as having been solved by the k-core parallelization in time T. This simulation indicates that even with such a speedup, coverage only increases slightly. For example, when given a 6 GB memory limit and a 30 minute time limit, even the 8-core version of this perfect parallelization of LAMA-2011 would solve only 6 more problems than the 721 solved by the standard single-core version when tested on all 790 problems from the 2006, 2008, and 2011 IPC competitions.

An alternative to parallelizing a single algorithm is to run members of an *algorithm portfolio* in parallel. This involves tackling each problem using a set of strategies that differ in either their *configuration* (*ie.* different parameter values or other settings) or in the underlying algorithm, and running these strategies simultaneously on different cores. This technique is inspired by two considerations. First, planners are expected to solve problems from a diverse set of domains, and no single algorithm can be expected to dominate all others on all domains. Second, this approach offers a simple alternative to the difficult process of parallelizing a single-core algorithm and it mostly avoids overhead from communication and synchronization.

Parallelizing a single memory-heavy algorithm in a shared-memory environment can also be problematic as it is often the available memory that limits coverage. In these cases, any speedup seen only causes memory to be exhausted more quickly. This behaviour is seen in the simulated LAMA-2011 parallelization, as the 8-core simulation ran out of memory on 52 problems. This means that regardless of how many more cores are used, at most 738 of the 790 problems can be solved using LAMA-2011 without an increase in memory. However, running a low-memory planner alongside a high-memory planner in a parallel portfolio can increase the coverage if the portfolio is selected properly.

ArvandHerd is the first parallel planning system to successfully combine disparate planning approaches to create a state-of-the-art parallel planner for shared memory machines. It won the multi-core sequential satisficing track of IPC 2011 [4] and was designed specifically to avoid the inherent limitations of parallelizing a single memory-heavy planning algorithm that were described above. Planners competing in this track were run on a 4-core machine with a maximum of 30 minutes of run-time and 6 GB of memory. In ArvandHerd, three cores were used to run a set of configurations of the linear-space random-walk-based planner Arvand, and the final processor was used to run the WA*-based LAMA-2008 planner.

This paper extends the description of ArvandHerd that was given in [20] by providing full documentation of the design choices made with respect to coverage, and then by analyzing the effectiveness of these choices. We will demonstrate that each of the Arvand and LAMA-2008 planners can be enhanced through the use of multiple configurations and restarts. While these techniques have been successfully applied in the satisfiability community, we demonstrate that they are similarly successful in planning. In Arvand, we will also describe an online learning configuration selection system which effectively speeds up the search. Finally, we will show that combining these two planners in a parallel portfolio solves more IPC benchmark problems than several state-of-the-art planners, even if they could be effectively parallelized.

2 Related Work

Work in the area of parallel planning has typically focused on the parallelization of heuristic search algorithms. This includes HDA* [12] and PBNF [2], two recent and successful parallelizations of A* which exhibit impressive speedups in distributed and shared memory

[1] University of Alberta, Canada, email: {valenzan, nakhost, mmueller, jonathan}@cs.ualberta.ca
[2] University of Denver, USA, email: sturtevant@cs.du.edu

systems, respectively. As both algorithms involve parallelizing A*, they will also have the same memory limitations as A* on shared-memory machines. As such, these algorithms cannot be expected to improve the coverage of A* except where search time is limited.

Note, there is nothing precluding the use of the parallel algorithms in a portfolio. If a parallelized algorithm is included, it can be allotted several cores on which to run while the remaining cores will run the rest of the portfolio. Alternatively, the parallelized algorithm can be run until it hits some resource limit, at which point the remainder of the portfolio will be run. This approach would benefit from both the speedups seen with the parallelized algorithm and the coverage improvements seen with a portfolio. As such, research into parallelizing individual search algorithms remains important work.

Using multiple configurations has also previously been shown to effectively improve the coverage and run-time of several single-agent search algorithms [21]. These ideas were central in the construction of ArvandHerd which takes the idea a step further by using multiple planners in addition to multiple configurations.

The portfolio approach was initially shown to effectively trade-off run-time and risk for heuristic algorithms for computationally hard problems [11]. At IPC 2011, this approach was also used successfully in the single-core satisficing track by Fast-Downward Stone Soup (FDSS) [9], which solved the second most problems of all competing planners [4]. This planner employs a variety of planning configurations in sequence, by giving each a time limit and restarting upon failure with a new configuration. The time limit assigned to each configuration is determined offline based upon the coverage that these configurations collectively achieved on a training set. However, while FDSS configurations differ in the heuristics used and the search enhancements employed, they do not differ in their operator ordering, which will be shown to improve coverage in Section 5.1. All of the configurations are also either WA* or hill-climbing based and so this portfolio is not as diverse as that of ArvandHerd. As a result, it lags behind ArvandHerd in coverage as we will show in Section 6.1.

Below we also discuss the use of restarts in WA*-based planners as a way to improve coverage. While LAMA-2008 already uses restarts, it does so in a very different way: it restarts with a less greedy configuration whenever a solution is found in an effort to find better quality solutions [16]. The restarts we consider have been shown to significantly improve coverage in SAT-solvers due to the long-tailed distribution of the problem-solving time [5]. The results below suggest that planning domains exhibit a similar property.

In IPC 2011, ArvandHerd solved 236 out of 280 problems [4]. As our focus is on coverage and no other multi-core planner solved more than 184 problems (by AyAlsoPlan [3]), we do not compare with any other competition planners below.

3 Parallel Planning with a Portfolio

In order to maximize the coverage of a planning portfolio, the portfolio members should be complementary in terms of their strengths and weaknesses. This requires diversity in the set of planners selected. If the portfolio is to be used on a shared-memory machine, then another important design decision relates to the way in which memory is partitioned between the planners. For example, if a two-planner portfolio contains two approaches for which lots of memory is essential, the collective coverage may suffer if each planner is assigned only half of the available memory. Avoiding this behaviour is therefore integral for properly selecting a portfolio and was an important consideration when building ArvandHerd.

3.1 The ArvandHerd Portfolio

LAMA-2008 [17], the winner of the sequential satisficing track of IPC 2008, was a state-of-the-art planner prior to IPC 2011 making it a natural selection as a member of the ArvandHerd portfolio. LAMA-2008 is WA*-based and can be memory-heavy. As such, although the ArvandHerd portfolio contains several configurations of LAMA-2008, it avoids the memory-partitioning issues mentioned above by running only a single LAMA-2008 configuration at a time. The additional LAMA-2008 configurations are only used if the first runs out of memory, in which case the planner *restarts* with another configuration. In Section 5, this planner will be described in more detail and this restarting procedure will be evaluated.

The ArvandHerd portfolio also contains several configurations of Arvand [15]. This planner uses a random-walk-based search which makes it ideal for use alongside LAMA-2008 in a portfolio for several reasons. First, this approach is very different from WA* and it can solve some problems that the systematic search of WA* is unable to handle. Secondly, domains in which Arvand exhibits poor behaviour are often successfully tackled by WA*-based approaches. Finally, Arvand has low memory requirements, and so when it is run alongside LAMA-2008 in a shared-memory system, the majority of the memory can be assigned to LAMA-2008, thereby avoiding the memory-partitioning issue described above. Arvand will be described in further detail in Section 4.

3.2 ArvandHerd Architecture

As both Arvand and LAMA-2008 are built on top of Fast Downward [6], ArvandHerd is run from a single binary. ArvandHerd uses Fast Downward's preprocessor to translate the PDDL problem description to an *SAS+*-like representation [7]. This preprocessor has not been parallelized. While doing so would speed up ArvandHerd, we consider this an orthogonal problem to that of parallelizing the search component and leave it as future work. As all planners tested in this paper require this preprocessing step, it is not counted against the time limits used in the experiments below.

When ArvandHerd begins its search, separate threads are spawned to run different portfolio members. In a k-core machine setting, $k-1$ threads will be running a parallelization of Arvand while the remaining thread runs LAMA-2008.

4 The Arvand Planner

Arvand is a sequential satisficing planner that uses heuristically-evaluated random walks as the basis for its search [15]. The execution of Arvand consists of a series of *search episodes*. In the simplest version of Arvand, each search episode begins with n random walks, with each walk being a sequence of m legal random actions originating from the initial state (s_i), where n and m are parameters. The heuristic value of the final state reached by each random walk is also computed using the FF heuristic function [10]. Once all n walks have been performed, the search *jumps* to the end of the walk whose final state, s, has the lowest heuristic value. Arvand then runs a new set of n random walks, only this time the walks originate from state s. This is followed by another jump to the end of the most promising walk from this new set of walks. This process repeats until either a goal state is found, or some number of jumps are made without any improvement in the heuristic values being seen. In the latter case, the current search episode is terminated, and the planner restarts with a new episode that begins with random walks originating from s_i.

Arvand has also been enhanced with *smart restarts* [14]. For this technique, the trajectories found during the most effective search episodes are stored in a *walk pool*. When a new episode begins, it then starts from a state randomly selected from a trajectory that itself was randomly selected from the walk pool, instead of from the initial state. This technique has been shown to improve Arvand's coverage [14]. Note, smart restarts have been disabled in the experiments in Sections 4.1 and 4.3 when experimenting with different ways to use multiple configurations. This was done so as to evaluate these techniques in isolation of the communication between configurations that a walk pool allows. However, smart restarts are employed in the more complete systems evaluated in Sections 4.4 and 6.

4.1 Arvand Configurations

There are a number of parameters in Arvand that can greatly affect its performance on a domain-by-domain basis. Perhaps the most important of these relates to the biasing of the random action selection. Arvand allows for random action selection to either be biased to avoid actions that have previously led to dead-ends (referred to as *MDA*) or to be biased towards using preferred operators (referred to as *MHA*). These different biasing strategies have been shown to each be useful for different domains [15].

A set of parameters related to the random walk length can also greatly affect performance. In Arvand, this length is adjusted online if little progress is being made in the heuristic values seen during a set of random walks. The initial walk length, the frequency with which walks are lengthened, and the factor by which they are lengthened (called the *extending rate*) are all parameters affecting this process.

The average performance of six different configurations over 5 runs on each of 480 problems is shown in Table 1. Configurations are given a maximum of 30 minutes and 2 GB per run. Configurations 1 and 4 correspond to the default configurations that use one of MDA or MHA. The remaining four configurations are constructed by modifying these default configurations by either its initial walk length or its extending rate, but not both, simultaneously.

Table 1. Performance of different Arvand configurations.

Config	Bias Type	Initial Walk Length	Extending Rate	Av. Num Solved
1	MDA	1	1.5	400.8
2	MDA	1	2.0	414.2
3	MDA	10	1.5	397.8
4	MHA	1	1.5	338.8
5	MHA	1	2.0	348.0
6	MHA	10	1.5	386.0

These experiments were performed on a cluster of machines each with two 2.19 GHz AMD Opteron 248 processors with 1 MB of L2 cache[3]. The problem set consists of all problems in the 2006 and 2008 planning competitions, except for sokoban from IPC 2008 which was omitted as previous testing has indicated that Arvand performs poorly in this domain regardless of how it is configured.

While the MDA configurations outperform the MHA configurations, this is not true in all domains [15]. For example, configuration 1 (MDA) only solves an average of 16.2 problems in the IPC 2006 pathways domain while configuration 4 (MHA) solves all 30 problems. This suggests that a combination of configurations is needed.

[3] While three different clusters were used in the experiments presented in this paper, any comparisons made are between planners/techniques that were tested on the same cluster

4.2 Combining Different Arvand Configurations

A simple way to combine k configurations in a single-core version of Arvand is to run each for $1800/k$ seconds. This technique will be referred to as *uniform time partitioning*. We can evaluate this approach by calculating the expected coverage based on the run-time of each configuration on its own. To do so, let $P(p, c, t)$ denote the probability that Arvand with configuration c will solve problem p in at most t seconds. Now let $P_k(p, C)$ denote the probability that p is solved when using uniform time partitioning with a set of configurations $C = \{c_1, c_2, ..., c_k\}$. As the searches performed by the different configurations are independent, the following holds:

$$P_k(p, C) = 1 - \overline{P(p, c_0, 1800/k)} * ... * \overline{P(p, c_k, 1800/k)}$$

where $\overline{P(p, c_i, 1800/k)} = 1 - P(p, c_i, 1800/k)$. Given a problem set, the expected number of problems solved is then the sum of these probabilities over all problems. For the values of $P(p, c, t)$, we will use the empirically determined probability that p is solved in time limit t as seen during the experiments summarized in Table 1.

Given the 6 configurations tested in Table 1, there are $\binom{6}{k}$ possible portfolios for any portfolio size k such that $1 \leq k \leq 6$. When $k = 2$, the best configuration set of all 15 possible sets is expected to solve 436.3 problems, an increase of 22.1 over the average number solved by the single best configuration alone. In fact, all but 2 of these 15 configuration sets improved over the best configuration in its own set. For $k = 4$ and $k = 6$, the expected coverage of the best sets are 434.4 and 431.4, respectively. These diminishing returns are to be expected since an increase in k decreases the amount of time any individual configuration will run. In contrast, the coverage of the worst set for any k reaches its highest point at $k = 6$, and surpasses the performance of the single best configuration alone when $k = 4$.

In practice, instead of starting with a new configuration every $1800/k$ seconds, we alternate amongst the configurations in a round-robin fashion. For each of $k = 2$, $k = 4$, and $k = 6$, we tested this approach with the configuration set of size k with the best expected uniform time partitioning performance. The coverage of alternation is slightly better, as it averages 435.4, 439.8, and 439.6 for k values of 2, 4, and 6, respectively, over 5 runs per problems. This occurs because alternation spends more time using the best configuration on any problem. For example, if two configurations, c_1 and c_2, are used on a single problem p, and c_1 is less effective on p than c_2, then search episodes using c_1 will stop making progress and restart more quickly than those using c_2. The available run-time will therefore skew more towards the longer, more effective c_2 configurations, than to the shorter, quickly-restarting c_1 configurations.

4.3 Configuration Selection as a Bandit Problem

Arvand was also enhanced through the use of an online configuration selection system which, while not increasing coverage, did decrease run-time. Given a set of configurations C, the system selects a configuration for the next search episode from C based on the performance of the configurations during previous episodes. This system views configuration selection as an instance of the *multi-armed bandit problem*. This paradigm requires the definition of a payoff function for search episodes. For this system, the reward given to a search episode e performed with configuration c is given as follows: where $h(v)$ is the heuristic value of state v, s is the state on the trajectory of e that achieved the lowest heuristic value, and s_i is the initial state, the reward given to c is $\max(0, 1 - h(s)/h(s_i))$.

Using this reformulation of configuration selection, configurations can be selected online using any multi-armed bandit algorithm. Arvand uses UCB [1], which begins by performing a single search episode with every configuration. After this stage, the configuration selected for the next episode is given by

$$\arg\max_{c \in C} r(c) + q \cdot \sqrt{\ln t(c)/t}$$

where $r(c)$ is the average reward seen thus far for configuration c, $t(c)$ is the number of search episodes performed with configuration c so far, t is the total number of search episodes, and q is a parameter called the *UCB constant value*. This algorithm has been shown to have strong theoretical guarantees on its ability to balance between focusing on effective selections and exploring the alternatives [1].

In order to give the UCB system some quick search episodes so as to more quickly identify the more useful configurations, the frequency with which episodes restarted was initially set high and then gradually decreased. The resulting system was then tested on a selection of sets of the configurations in Table 1. In general, the UCB system did not significantly change the coverage of Arvand when compared to the use of round-robin configuration selection but it did improve run-time. For example, four different values of the UCB constant value were tested on the configuration set of size 2 with the best expected uniform cost partitioning performance. Recall that round-robin selection solved 435.4 problems when applied to this same set. Of the UCB constant values tested (0.1, 0.5, 1.0, and 5.0), the most problems solved when using the UCB selector was 439.4 and the least was 437.2. However, if we only consider the 399 problems solved on all five runs per problem by either a UCB system or round-robin, we see that even the UCB constant resulting in the longest run-time results in a 2.75 times speedup, while the value with the shortest run-time sees a 3.90 times speedup.

4.4 Parallelizing Arvand

For ArvandHerd, a simple parallelization of Arvand was developed in which each core runs an independent search episode. The only communication between cores is through the use of a shared walk pool and a shared UCB configuration selector. When a core has completed a search episode, it submits the corresponding trajectory to the shared walk pool, and gets a trajectory in return. The core also submits the reward for its current configuration to the shared UCB system and in return is given a configuration to use in its next search episode. The correctness of the walk pool and UCB learner are maintained by limiting access to each to only one thread at a time. As the search episodes dominate execution time, there is little synchronization or contention overhead caused by sharing these resources.

Parallel Arvand was tested with different numbers of cores on the 790 problems from IPCs 2006, 2008, and 2011. These experiments were performed on a cluster of machines, each with two 4-core 2.8 GHz Intel Xeon E546s processors with 6 MB of L2 cache. The configuration set used is identical to the set used in IPC 2011. It includes configurations 1, 4, and 6 from Table 1 and another MDA configuration with an extending rate of 1.5 and an initial walk length of 3. This configuration set was selected manually prior to IPC 2011 based on familiarity with Arvand, and also before the expected coverage analysis described above had been performed. We use this set in our experiments below so as to evaluate how parallel Arvand contributed to ArvandHerd's success.

Table 2 shows the average number of problems solved over five runs per problem when using different numbers of cores. In addition, the table shows how much faster the multi-core versions were in comparison to the single-core version on the 639 problems that were solved on all five runs regardless of the number of cores used.

Table 2. The performance of parallel Arvand.

	Number of Cores				
	1	2	3	4	8
Coverage	660.4	668.0	671.4	677.8	679.6
Speedup Factor	1.0	1.9	2.5	3.0	3.4

While 8-core Arvand solved 19.2 more problems on average than the 1-core version, parallel Arvand is still unable to solve as many as the 721 problems that LAMA-2011 solved. Domain-by-domain analysis also indicates that domains in which the single-core version exhibits poor performance are often also difficult for the multi-core versions. For example, neither the single-core nor the 8-core version of Arvand can solve even one of the 20 barman problems from IPC 2011. This suggests that there is a limit in the coverage that can be achieved in this domain through parallelizing Arvand. However, LAMA-2008 can solve 15 of these problems, thus making the case for the use of a portfolio.

5 The LAMA-2008 Planner

LAMA-2008 is a WA*-based planner that won the sequential satisficing track of IPC 2008 [8]. It uses a number of planning techniques including *multiple heuristic functions*, *preferred operators*, and *deferred heuristic evaluation* [17]. Below, we briefly describe the set of heuristics that we used in this system and show that LAMA-2008's coverage can be improved through the use of restarts.

LAMA-2008 can use several heuristics to guide search through a process called *multi-heuristic best-first search* [6]. Previous analysis of this planner has shown that strong performance can be achieved if the set of heuristics used includes both the landmark-count heuristic (*LM*) and a version of the FF Heuristic which ignores action costs (*FFs*) [17]. For details on these heuristics, see [10] and [17]. We found that if we also included a third heuristic function, a version of FF which did adjust for action costs (*FFc*), then LAMA-2008 could solve 464 of the 550 problems taken from the 2008 and 2011 competitions, as opposed to just 449 when this heuristic is omitted.

Note, before IPC 2011 the LAMA-2008 code was re-factored so as to make it thread-safe. Subsequent experimentation has shown that these changes do not significantly impact the coverage of the system when run on a single core. As such, though the experiments use our newer version of LAMA-2008, the results shown below are expected to hold if the techniques considered were implemented in the original. Also of note, as *FFs* and *FFc* are identical in unit cost domains, we only use one of these heuristics in the IPC 2006 domains (which have no action costs).

5.1 Randomizing Operator Order

The use of multiple operator operator orderings has previously been shown to yield an effective WA*-based parallel planning system [21]. Below, we will show that this is also true when using a complete planning system like LAMA-2008 even though it already uses multiple heuristics to introduce diversity.

Recall that LAMA-2008 uses *deferred heuristic evaluation*. When using this technique, the heuristic value of a state s is not calculated until s is expanded. When s is generated, the heuristic value used for s is actually the heuristic value of the parent of s. This technique can often improve search time by decreasing the number of expensive

heuristic evaluations. It will also increase the number of ties, as any two children c_1 and c_2 with the same parent p that are achieved with equal cost actions will have the same f-cost. A unit cost domain represents an extreme case of this phenomenon; all children of the same state will be assigned the same f-cost. This increase in the number of ties can increase the variance in coverage found with different operator orderings.

This variance can be seen when experimenting with *random operator ordering* in LAMA-2008. This technique involves randomly re-ordering the list of children of an expanded state before those new states are added to any open list. For this experiment, LAMA-2008 was configured to use random operator ordering in greedy best-first search (GBFS) with the *FFs* and *LM* heuristics and preferred operators. This system was tested on all 510 problems from the 2006 and 2008 competitions for 5 runs per problem with a 30 minute time limit and a 2 GB memory limit. These experiments were run on a cluster of machines each with two AMD Opteron 250 2.4 GHz processors, each with 1 MB of L2 cache. While the average number of problems solved is 431.8, if the best random seed had been selected on a problem-by-problem basis, 448 problems would have been solved. If the worst seed had been selected on a problem-by-problem basis, only 415 problems would have been solved.

This variance suggests the use of *restarts*, whereby if some resource limit has been reached without a solution having been found, the search starts fresh with a new random seed. The expected effectiveness of using restarts can be calculated using the technique described in Section 4.2. Table 3 shows this data for different types of search and different numbers of restarts. In these experiments, LAMA-2008 uses *FFs* and *LM* when running GBFS, and *FFc* and *LM* when using WA* (*ie.* $w = 7$ represents a weight 7 search).

Table 3. Expected performance of LAMA-2008 using restarts.

Search Type	Number of Restarts 0	1	2	4	8	16
GBFS	431.8	437.0	438.5	440.3	437.3	427.8
$w = 7$	403.6	408.2	409.1	409.0	405.4	397.7
$w = 1$	207.2	209.1	209.8	207.3	205.4	194.9

Table 3 shows that a small number of restarts can help improve the expected number of problems solved, though too many restarts can degrade performance. This is true for all configurations tested including weights 10 and 5 which are not shown in the table. The estimation technique also indicates that if LAMA-2008 is set to restart not just with a new random seed but also with a different configuration, the additional diversity would help to further improve coverage. For example, when restarting 4 times such that each of GBFS, $w = 10$, $w = 7$, $w = 5$, and $w = 1$ are run for a maximum of six minutes, the expected coverage is 448.4. This result motivates the inclusion of several LAMA-2008 configurations in the portfolio.

The version of LAMA-2008 used in the competition was set to restart on a memory limit instead of on a time limit. This limit was enforced through the use of an internal memory estimator. The closed list was also saved in between restarts so as to avoid recomputing the heuristic values of states seen in previous iterations. Subsequent experiments suggest that this was not necessarily the best approach. When restarting with a 2 GB memory limit and using the 5 configurations from above, an average of 440.2 problems was solved. While this is competitive with restarting on a time limit with GBFS alone, it trails behind the expected performance of restarting with different configurations. This is at least partially due to inaccuracies in our memory estimator which could only provide rough estimates. A more in-depth consideration is beyond the scope of this paper.

6 Putting the Portfolio Together

In this section, we evaluate ArvandHerd on IPC benchmark domains and show that it outperforms several state-of-the-art planners, and would do so even if these planners could be efficiently parallelized. Before doing so, we begin by describing the Arvand and LAMA-2008 configurations used in the ArvandHerd portfolio.

ArvandHerd runs multiple configurations of Arvand using the Arvand parallelization described in Section 4.4, and a single instance of LAMA-2008. The configurations of Arvand used in both the IPC competition and in the experiments below are the same as those used in that section.

The version of LAMA-2008 used below differs slightly from that used in IPC 2011 in terms of its restart-inducing memory limit. In the competition, the memory limit was set at 2.7 GB even though the systems were allowed a maximum of 6 GB. This limit was selected so as to accommodate the memory needs of the plan improvement system used. As the focus of this paper is coverage and the plan improvement system (a description of which is beyond the scope of this paper) has been updated to allow for a larger limit, this restart-inducing limit has been increased to 4 GB in the experiments below.

A second difference between the competition version of LAMA-2008 and that used in the experiments below relates to the configurations used. In the experiments, the first iteration performed is GBFS, which is followed by a set of WA* searches that use the following weights in the order given: 10, 5, 2, and 1. If the weight 1 search restarts due to the memory limit being reached, this cycle of searches is repeated indefinitely (starting back at GBFS) until the time limit is hit or a solution is found. In the competition version, the weight 1 search was followed by several more low-weight searches which were included for plan improvement, and have therefore been removed. A final difference is that in the competition version, once all LAMA-2008 configurations were tried once, the thread running it would join the other 3 in running parallel Arvand. This is not done in the experiments below in the interest of evaluating the general portfolio technique, as the switch from LAMA-2008 to Arvand assumes the portfolio members can all be run from a single binary.

6.1 ArvandHerd on IPC Benchmarks

ArvandHerd was run 5 times on each of the 790 problems in the 2006, 2008, and 2011 planning competitions on the same cluster described in Section 4.4 (as were all planners considered in this section). The performance of this system can be seen in Table 4, which shows that ArvandHerd 's coverage is significantly better than that of the Arvand parallelization and the perfectly linear parallelizations of LAMA-2011 and FDSS. ArvandHerd achieves its high coverage in the expected way, with Arvand and LAMA-2008 cancelling out each others weaknesses. For example, recall that Arvand is unable to solve even a single barman problem. With LAMA-2008 in the portfolio, 2-core ArvandHerd solves an average of 15.4 of the 20 problems (similar to the 16 solved by LAMA-2008 when run on its own). Similarly, while LAMA-2008 only solves 19 of 30 problems in storage (IPC 2006), 2-core ArvandHerd solves an average of 29.4 (similar to the 30 that Arvand solves when run on its own). In this way, ArvandHerd combines two planners in LAMA-2008 and Arvand whose performance lag significantly behind LAMA-2011 when used on their own to surpass even a perfectly linear parallelization of LAMA-2011.

Table 4. Performance of parallel planners.

Planner	Number of Cores			
	1	2	4	8
LAMA-2008 Simulation	639.0	641.0	643.0	*NA*
LAMA-2011 Simulation	721.0	724.0	726.0	727.0
FDSS-1 Simulation	720.0	724.0	726.0	727.0
Parallel Arvand	660.4	668.0	677.8	679.6
ArvandHerd	*NA*	737.2	743.2	741.8
ArvandHerd +LAMA-2011	*NA*	750.4	754.2	755.2

6.2 Using LAMA-2011 in ArvandHerd

LAMA-2008 was included in the portfolio instead of LAMA-2011 because Arvand had originally been built into the LAMA-2008 framework. Table 4 shows that performance would further improve if LAMA-2011 had been used instead (see the row labelled "ArvandHerd +LAMA-2011"). For testing the k-core performance of this portfolio, parallel Arvand was run with $k-1$ cores on all problems that LAMA-2011 could not solve with a 4 GB memory limit. The table shows the sum of the number of problems solved by LAMA-2011 and the average number of problems solved by $(k-1)$-cores running parallel Arvand.

The fact that the new portfolio successfully solves even more problems than LAMA-2011 by itself reflects the importance of Arvand in the ArvandHerd portfolio. Arvand is not simply covering the weaknesses in LAMA-2008 that have been addressed with the release of LAMA-2011. It is also handling problems that this state-of-the art planner cannot.

While coverage does not increase substantially with additional cores regardless of whether LAMA-2008 or LAMA-2011 is used in the portfolio, this is largely due to a "glass ceiling" effect. For example, the 2-core portfolio containing LAMA-2011 is already solving 95% of the test problems.

7 Conclusion

In this paper, we began by demonstrating that parallelizing a single planning algorithm is not necessarily the best way to use a multi-core shared memory machine if the goal is to maximize coverage. This occurs because while the parallelized algorithm may be faster, it will have similar limitations as the original single-core algorithm in terms of both resource usage and the domains it handles well. Instead of parallelizing a single algorithm, we used an algorithm portfolio approach to parallel planning in the development of ArvandHerd, which won the multi-core sequential satisficing track at IPC 2011.

This paper contains a full description of ArvandHerd and an analysis of how several design decisions contributed to its success. In particular, we have shown that the use of multiple configurations and restarts can improve the coverage of each of the two planners used in the portfolio, namely LAMA-2008 and Arvand, even when used on only a single core. While these techniques have previously been used in the SAT-solving community, we have shown that their success extends into automated planning. The combination of these planners in ArvandHerd is then shown to outperform even the simulated performance of perfect parallelizations of two state-of-the-art single-core planners. It is also shown that the coverage can be further improved by replacing LAMA-2008 with LAMA-2011in the ArvandHerd portfolio.

More generally, we have demonstrated through the construction of ArvandHerd that the use of a portfolio is a powerful approach for building general parallel planners due to its ability to combine the strengths of different planners. While the ArvandHerd portfolio only contains two planners, others may be included as well. In particular, we suspect that planners that use approaches other than the WA*-based and random-walk-based approaches already included will offer the most potential for further improving coverage. These may include SAT-based planners [19] and probe-based planners [13]. However, an evaluation of portfolios containing these approaches on top of those already in ArvandHerd is left as future work.

Acknowledgments

This research was supported by NSERC, iCore, and Alberta Ingenuity.

REFERENCES

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer, 'Finite-time Analysis of the Multiarmed Bandit Problem', *Machine Learning*, **47**(2-3), 235–256, (2002).
[2] Ethan Burns, Wheeler Ruml, Sofia Lemons, and Rong Zhou, 'Best-First Heuristic Search for Multicore Machines', *JAIR*, **39**, 689–743, (2010).
[3] Juhan Ernits, Charles Gretton, and Richard Deardon, 'AyAlsoPlan: Bitstate Pruning for State-Based Planning on Massively Parallel Compute Clusters', in *IPC 2011 Deterministic Track*, pp. 117–124, (2011).
[4] Ángel García-Olaya, Sergio Jiménez, and Carlos Linares López. IPC 2011 Deterministic Track, 2011. http://ipc.icaps-conference.org.
[5] Carla Gomes, Bart Selman, and Nuno Crato, 'Heavy-tailed distributions in combinatorial search', in *CP*, pp. 121–135, (1997).
[6] Malte Helmert, 'The Fast Downward Planning System', *JAIR*, **26**, 191–246, (2006).
[7] Malte Helmert, 'Concise finite-domain representations for pddl planning tasks', *Artificial Intelligence*, **173**(5-6), 503–535, (2009).
[8] Malte Helmert, Minh Do, and Ioannis Refanidis. IPC 2008 Deterministic Track, 2008. http://ipc.informatik.uni-freiburg.de.
[9] Malte Helmert and Gabriele Röger, 'Fast Downward Stone Soup: A Baseline for Building Planner Portfolios.', in *ICAPS-2011 Workshop on Planning and Learning*, pp. 28–35, (2011).
[10] Jörg Hoffmann and Bernhard Nebel, 'The FF Planning System: Fast Plan Generation Through Heuristic Search', *JAIR*, **14**, 253–302, (2001).
[11] Bernardo Huberman, Rajan Lukose, and Tad Hogg, 'An Economics Approach to Hard Computational Problems', *Science*, **275**, 51–54, (January 3 1997).
[12] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea, 'Scalable, Parallel Best-First Search for Optimal Sequential Planning', in *ICAPS*, (2009).
[13] Nir Lipovetzky and Hector Geffner, 'Searching for Plans with Carefully Designed Probes', in *ICAPS*, (2011).
[14] Hootan Nakhost, Jörg Hoffmann, and Martin Müller, 'Resource-Constrained Planning: A Monte Carlo Random Walk Approach', in *ICAPS*, (2012).
[15] Hootan Nakhost and Martin Müller, 'Monte-Carlo Exploration for Deterministic Planning', in *IJCAI*, pp. 1766–1771, (2009).
[16] Silvia Richter, Jordan Thayer, and Wheeler Ruml, 'The Joy of Forgetting: Faster Anytime Search via Restarting', in *ICAPS*, pp. 137–144, (2010).
[17] Silvia Richter and Matthias Westphal, 'The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks', *JAIR*, **39**, 127–177, (2010).
[18] Sylvia Richter, Matthias Westphal, and Malte Helmert, 'LAMA 2008 and 2011', in *IPC 2011 Deterministic Track*, pp. 117–124, (2011).
[19] Jussi Rintanen, 'Heuristics for Planning with SAT and Expressive Action Definitions', in *ICAPS*, (2011).
[20] Richard Valenzano, Hootan Nakhost, Martin Müller, Nathan Sturtevant, and Jonathan Schaeffer, 'ArvandHerd: Parallel Planning with a Portfolio', in *IPC 2011 Deterministic Track*, pp. 113–116, (2011).
[21] Richard Valenzano, Nathan Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto, 'Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms', in *ICAPS*, pp. 177–184, (2010).