Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Lucas Bordeaux[†] and **Youssef Hamadi**[†‡]
[†]Microsoft Research, Cambridge, United Kingdom
[‡]LIX École Polytechnique, Palaiseau, France
{lucasb, youssefh}@microsoft.com

Abstract

Motivated by the recent hardware evolution towards multi-core machines, we investigate parallel planning techniques in a shared-memory environment. We consider, more specifically, parallel versions of a best-first search algorithm that run K threads, each expanding the next best node from the open list. We show that the proposed technique has a number of advantages. First, it is (reasonably) simple: we show how the algorithm can be obtained from a sequential version mostly by adding parallel annotations. Second, we conduct an extensive empirical study that shows that this approach is quite effective. It is also *dynamic* in the sense that the number of nodes expanded in parallel is adapted during the search. Overall we show that the approach is promising for parallel domain-independent, suboptimal planning.

Introduction

The advent of multi-core computers poses challenges as well as opportunities for compute-intensive methods such as automated planning. Planning is a prime candidate of technology that can potentially benefit from parallel hardware: the exploration of very large search spaces can consume enormous computational resources and is on the face of it "embarrassingly parallel". The challenge, however, is to rethink the state-of-the-art algorithms so that they efficiently exploit this parallelism. There are well-known pitfalls: a clever strategy has to be used to split the workload among processors, synchronization costs need to be restricted, etc.

This paper contributes to the emerging field of parallel planning. It studies a simple approach to shared-memory parallel planning that bears some similarities with the K-Best-First Search (KBFS) approach proposed in (Felner, Kraus, and Korf 2003). The basic idea of KBFS is that instead of expanding at every step a single node, optimal w.r.t. the heuristic function, the K best nodes are expanded. This is a more robust approach, because it takes into account that heuristic functions are rarely perfect: by introducing diversification it reduces the risk of being guided for too long towards wrong directions. This approach is also a natural candidate for parallelism, since we have at any time K independent node expansions to perform.

We propose an approach in which a shared state space is expanded in parallel by several threads concurrently. This approach does *not* explicitly mimic the original, sequential KBFS algorithm, but we explain that it essentially implements a similar behavior, while additionally exploiting the computational power of multiple cores. The approach is, deliberately, relatively simple: in fact our presentation shows how, starting from a sequential algorithm, annotations on the code suffice to move to a full-fledged parallel version. The advantage of such an approach is that the correctness of the parallel algorithm naturally follows from reasoning on the sequential one. In terms of implementation, the annotations are very close to actual constructs used in parallel programming libraries. This makes our results easy to understand and reproduce by other interested researchers.

Starting from a basic version of the parallel algorithm, we show how a number of improvements are smoothly integrated. First, we show that the algorithm can be made *adaptive*, in that the number of threads automatically adjusts to the problem. Second, we show that a *restart* policy brings complementary benefits. We conduct an empirical evaluation of our algorithms against a large set of planning problems. For the sake of generality we targeted *domain-independent* (PDDL) planning problems. This is another difference with the original KBFS work, which focussed on domain-dependent planning. Our focus in this paper is, also, on *suboptimal* planning: the approach is used to compute a plan without consideration of optimizing a quality measure. The main conclusion of this experimental work is that the proposed approach to parallelism, although simple, actually provides substantial performance improvements over the sequential algorithm; this is good news w.r.t. the bad performance of the parallel implementation of KBFS made in (Burns et al. 2009).

Outline. First we present related work on parallel planning. We then present our basic approach, explaining in particular how we gradually move from a sequential algorithm to a parallel one. Next comes a first empirical study used to evaluate and improve the basic algorithm. From the conclusions of this experimental study, we propose improvements of the algorithm. A second empirical section analyzes in details the performance of the resulting optimized algorithm. Our main findings are then summarized in the conclusion.

Prior Work on Parallel Planning

Several approaches to parallel planning have been proposed in recent years. Parallel Retracting A* (Evett et al. 1995), was implemented on a Connection Machine and had to deal with very severe memory limitations. In that algorithm, a distributed hash function is used to allocate generated states to a unique processing unit and avoid unnecessary state duplications. PRA* dealt with the memory limitation through a retraction mechanism which allowed a processor to free its memory by dropping states. In order to confirm the transfer of a state, synchronous communication channels had to be used, which seriously slowed down the search process. Transposition-table driven work scheduling (Romein et al. 1999), similarly to PRA* uses a hash function to avoid duplication. It is based on IDA* and, running on a standard architecture, does not necessitate any retraction mechanism and can efficiently exploit asynchronous communication channels. Parallel Frontier A* with Delayed Duplicate Detection (Niewiadomski, Amaral, and Holte 2006) uses a strategy based on intervals computed by sampling to distribute the workload among several workstations, targetting distributed-memory systems as opposed to previous approaches. In (Kishimoto, Fukunaga, and Botea 2009), the authors introduce Hash Distributed A* (HDA*) which combines successful ideas from previous parallel algorithms. HDA* uses a hash function which assigns each generated state to a unique processing unit in order to avoid the duplication of the search efforts. This mechanism was introduced in PRA*, which unfortunately combined it with synchronous communication channels which caused a lot of overhead. This problem was addressed in HDA* by the use of non-blocking communication (as in (Romein et al. 1999)). In (Burns et al. 2009) the authors present Parallel Best-NBlock-First (PBNF). It uses an abstraction to partition the state space. PBNF allows each thread to expand the most promising nodes while detecting duplicate states. Rather than sleeping if a lock cannot be acquired, a thread can perform "speculative" expansions by continuing the expansion of its current part of the space. This technique keeps cores busy at the expense of duplicate work. Finally, (Valenzano et al. 2010) adapts to planning a technique called dovetailing, in which several instances of a search algorithm with different parameter settings are run in parallel. We use this approach in this paper as a bottom-line, and show how our approach improves over a simple dovetailing strategy.

K-Parallel best-first search

We now introduce the main features of the parallel version of best-first search we propose. We then show a first round of empirical results that helped us propose some further improvements.

Overview

At a high-level, the approach consists in sharing the state-space between the K threads that are created. Note that K does not need to be equal to the number of cores available, nor does it need to stay constant. The threads continuously expand nodes in the shared data-structure, with the main differences w.r.t. KBFS being that the threads do not wait until the K best nodes are expanded before picking a new one in the open list. As soon as a thread has finished expanding a node, it extracts the actual best node from the open list and expands it, whatever the state in which the other threads are. As described below, the only bottleneck that requires locking concerns concurrent access to the *open* and *closed* list. For several reasons that will be discussed and experimentally assessed, this problem remains limited.

From Sequential to Parallel

To give more details on the algorithm we show how it can be obtained from a sequential version mostly by adding parallelization annotations to specific parts of a classical best-first search algorithm. These annotations are based on the OpenMP framework (OpenMP 1997), which is an API specification for parallel programming integrated into several C, C++ and Fortran compilers. OpenMP provides two kinds of tools: *directives* that control parallelism (thread creation, synchronization, critical parts, etc.) and *runtime library routines* that permit to get or set several parameters at run time and manually control synchronization locks.

This approach is interesting for the following reasons: first reasoning on the correctness of the algorithm is simple —essentially it comes from the fact that the sequential version is correct. Second, our approach only uses directives, which yields minimal changes to the code and can be removed at compile time, making it identical to the sequential version. A classical best-first search algorithm can be modified as follows.

Declaration of global thread-private data. By default, all variables and data structures created in the sequential part of the code are shared among all threads. By using the `threadprivate(`*`variable_list`*`)` directive, we make the listed global variables private to each thread. These variables could have been declared locally in a parallel section of the code, and would have then been private to each thread; but as the actual code uses some global variables, this directive permits to keep them global but private at the thread level. These variables essentially concern some structures used to compute the heuristics.

Initialization of global thread-private data. After the sequential code which parses the PDDL files, instantiates the operators, creates the data structures, and performs some preprocessing intended to reduce the problem size (removing irrelevant actions and so on), a first `parallel num_threads(`K`)` directive defines a parallel block starting K threads. The global variables private to each thread described above are then initialized. When the `parallel` block ends, all threads are synchronized through an implicit *barrier* in the OpenMP terminology, and the code continues in a sequential way in the master thread. All global thread-private variables created in the parallel block will retrieve their value in the next parallel block, the data belonging to the master thread being the only one accessible in the sequential part that follows.

Initialization of shared data. Now in the sequential part, the *open* and *closed* lists are created. As all threads use these lists, they are not declared in the initial `threadprivate` directive. The initial state of the problem is then evaluated by the heuristic function and put in *open*, using the global variables, private to the master thread, previously initialized.

Parallelizing the main loop. A second `parallel num_threads(`K`)` directive is then encountered, where all threads retrieve the values of the global thread-private variables initialized earlier, surrounding the main loop of the best-first search algorithm. This loop extracts and expands the best node from *open*, puts it in *closed* and adds its children to *open* after evaluating them. This operation is performed by all threads in a concurrent way, so each variable or data structure used to expand and evaluate a node must be private to each thread. Also, all access to *open* and *closed* must be performed in a safe way, as we did not use lock-free structures (actually, they are implemented by way of red-black trees). Consequently, all operations on these lists are enclosed in blocks preceded by the `critical` directive, which ensures that only one thread at a time can perform them. This is the only bottleneck of our approach: several threads trying to simultaneously access *open* and *closed* must wait for the implicit lock of the `critical` section to be freed.

There is one difference in the main loop with a classical sequential implementation. This loop must not end when *open* is empty, as usually made in the case where the state space is completely explored without finding a solution. Indeed, it may happen (particularly in the first iterations) that a thread tries to extract a node from the empty *open* list, while one or several threads are still expanding some other nodes and will later put their children into the list. To overcome this problem, a shared counter is used to enumerate the threads that are actually expanding a node. When a thread extracts a node from the open list, it increments this counter; and just before demanding a new node, it decrements the counter. When a thread tries to extract a node, if the counter is equal to 0 and *open* is empty, the search can be terminated and no solution reported. It must be noted that simple operations on shared counters can be performed in a thread-safe and non-blocking way with the `atomic` directive.

Ending the parallel work and returning the solution. Once a solution is found by a given thread, the main loop of this thread is terminated (which can this time be assessed in the loop condition). Then, several alternatives exist to properly finish the work, the problem being that no possibility exists in OpenMP to explicitly kill a thread. One alternative is to change the value of a shared Boolean from false to true, indicating to all threads that a solution is found and letting them terminate as soon as possible, in order to reach the implicit synchronization barrier of the `parallel` directive surrounding the main loop. Another possibility is to simply output the solution in the successful thread and exit the program, without waiting the other threads to synchronize. In both cases, a `single` directive just after the main loop, surrounding a block entered only by the first successful thread that reaches it, ensures that either the plan solution recording operation in a data structure accessible by the master thread (for the next sequential part) or the solution output and exit is performed by only a single successful thread.

Analysis

Coarse-grained parallelism. A first observation is that we focus on domain-independent planning with fairly costly heuristic functions. We think this is the right assumption for parallel planning: with such heuristic functions a substantial amount of work needs to be done when expanding a node, which guarantees a sufficiently coarse-grained approach to parallelism. We think that our approach is not as appropriate for cheap heuristic functions because in this setting the various threads would spend a lower portion of their time actually performing computation —which means that a bigger share of the time would be spent in concurrent access to the search frontier. Such fine-grained approaches to parallelism seem more prone to synchronization overhead.

Dynamic aspects of K-Parallel BFS. A second observation concerns the dynamic aspects of K-Parallel BFS. Although this algorithm relies on the same basic principle than KBFS —expanding several nodes instead of the best one only may guide search towards a good direction by correcting heuristic mistakes—, it has the fundamental difference that the number of nodes simultaneously extracted from the *open* list to be expanded is not exactly K, but less than or equal to K. It may vary following several external parameters, such as the number of cores, the time needed to expand a node and compute the heuristic, the behavior of the operating system thread scheduler, etc. The main advantage is that as previously seen, this approach is extremely simple to implement on top of any best-first search algorithm without really modifying the algorithm itself. The main drawback is that we have no control on the number of nodes simultaneously expanded, and that this number depends on external parameters as seen above, out of control inside the algorithm. Furthermore, the K parameter is limited to the number of threads that the system can reasonably handle. It could be possible to mimic the behavior of KBFS by distributing the K nodes over the cores with one thread by core, but we preferred to stick to the simplest alternative that requires minimal changes to an existing implementation.

Optimality and solution quality. A third observation is about optimality and plan quality. Although our approach is designed with suboptimal planning in mind, where reaching a goal state is enough to return a solution, it could be worth the effort trying it on optimal planning too with costly heuristic functions. It is claimed in (Felner, Kraus, and Korf 2003) that KBFS should not be effective for optimal planning with admissible and monotonic heuristic functions, because "all nodes whose cost is less than the optimal solution cost must be expanded". But in that case, distributing the work among several computation units could still be beneficial. Finally in that work, we do not care about plan quality, as such an algorithm could be for example embedded into an optimization algorithm such as the evolutionary metaheuristic proposed in (Bibaï, et al. 2010). Solution optimization could be performed directly by the planning algorithm itself (see e.g. (Richter, Thayer, and Ruml 2010)), but we chose once again to evaluate the simplest objective of finding one solution as quickly as possible.

Empirical Study of Basic Parallelization

Experimental Setting

We implemented our approach on top of a new implementation in C of the YAHSP planner (Vidal 2004), a state-of-the-art PDDL planner—see (Bibaï, et al. 2010) for a recent comparison with other planners . Essentially[1], YAHSP computes lookahead states by applying as many actions as possible from the relaxed plan computed in each state of a forward search in a FF manner (Hoffmann and Nebel 2001), and adds these states to the *open* list. The plan repair algorithm introduced in YAHSP is used, in order to maximize the number of actions used from relaxed plans. This new implementation is simpler than the original one, as some features (optimistic BFS with helpful actions, goal-preferred actions) revealed to be less useful on modern benchmarks than they used to be for old ones. In order to diversify the lookahead strategy, stochastic tie-breaking is performed in various choice points in the relaxed plan computation and repair. The search algorithm is WA*, with a weight on the heuristic equal to 3, modified with the OpenMP directives for parallelization. This version is able to solve classical, cost-based and temporal planning problems (with a simple deordering algorithm to parallelize the plan).

The instances used for the experiments come from the 2nd to the 6th International Planning Competitions, and include a total of 2042 problems from 54 domains, which are all the instances from these IPCs that YAHSP can handle. Many of these instances are solved very quickly, which explains why the difference in the number of solved problems between all the versions we tested is not so high w.r.t. the total number of instances. The tests were run on a Xeon 5160 processor (4 cores, 3GHz). The time-out was set to 30 minutes of wall-clock (WC) time. All reported timings are WC times.

We should also note that experiments with parallel programming involve a great deal of non-determinism: running the same algorithm twice on the same instance, with identical number of threads and parameters, may result in different solutions, and sometimes in very different runtimes. The runtimes in our figures have *not* been averaged over a series of run: our figures are based on one run for each instance/configuration. However the figures *are* averaged-out by the very fact that we run each algorithm over a large number of problems. We checked, by re-running some of the experiments, that our *overall* results are not affected by the non-determinism, in the following sense: if we run a particular algorithm/configuration over all instances several times, the runtimes for a few specific instances may vary, but the overall picture is not affected: we get comparable cumulated numbers of instances solved, and consistent pairwise comparisons between algorithm configurations.

[1]Due to lack of space, we cannot give much details about YAHSP: the interested reader may find a more comprehensive description in (Vidal 2004).

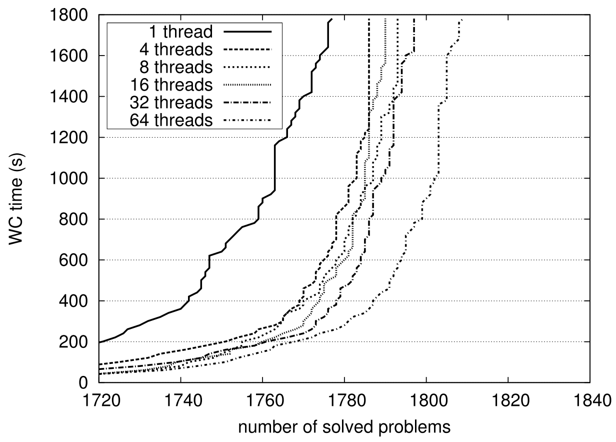

Figure 1: Number of instances solved for various number of threads ranging from 1 to 64. For each WC time t on the y axis, the corresponding value on the x axis gives the cumulated number of instances solved in under t seconds.

Results

Impact of multi-threading. We first report in Fig. 1 the overall number of instances solved as we increase the number of threads. A first observation is that multi-threading effectively increases the number of solved instances: from 1777 for the 1-thread version up to 1809 for the 64-thread one (25 more). Note, again, that in all cases the same 4-core computer is used. In other words, when the number of threads is greater than 4 we have more threads than cores. Importantly, such configurations pays off, with our best performance observed for the highest number of threads, 64 (experiments with over 64 threads showed a stabilization of the performance at around this number and are not reported). The reason why a number of threads greater than the actual number of cores pays off is that extra diversification is obtained by the multi-threading, which allows the algorithm to solve some difficult instances. This is consistent with the observations made with KBFS.

Correlating the benefits of multi-threading to the instance hardness. An in-depth analysis of the naive version of our algorithm shows in which cases a high-level of multi-threading pays off, in function of the hardness of the instance considered. The intuition is the following. Many problems in our benchmark suite are easy, and we assume that such would be the case in a practical deployment scenario as well: for such instances the heuristic function is well-informed and guides the search effectively towards the goal. In this case multi-threading only causes overhead since we lose time exploring nodes that deviate from a good heuristic score. We expect multi-threading to pay mostly for instances that are beyond a certain threshold of hardness, where it can really unblock an imperfect heuristic function and improve the scalability.

To validate this intuition experimentally we measure the *winning ratio* between various pairs of parallel strategies. The winning ratio is the percentage of times the first strat-

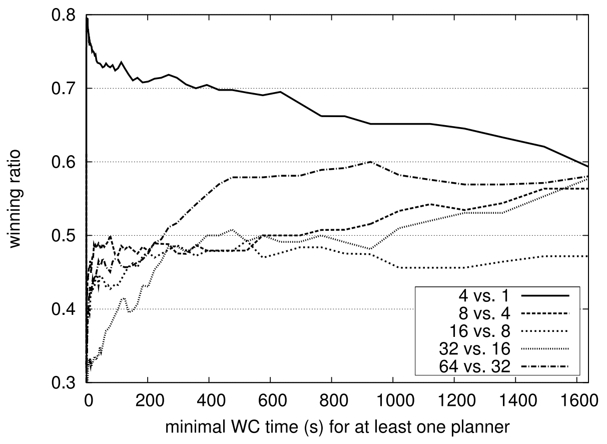

Figure 2: Winning ratios between various pairs of parallel strategies, in percentage (y axis), and expressed in function of the problem hardness (x axis).

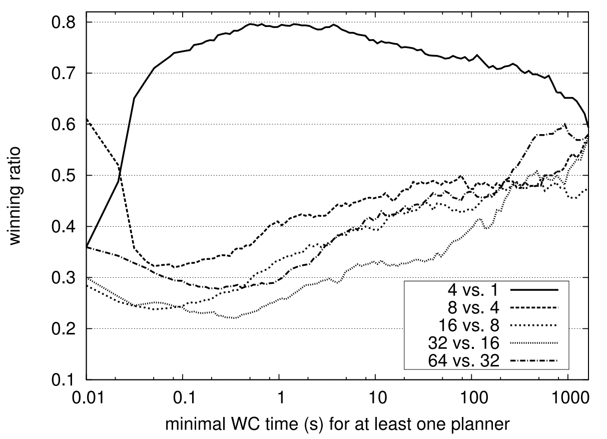

Figure 3: Winning ratios between various pairs of parallel strategies; the x axis uses a logarithmic scale to zoom on the small runtimes.

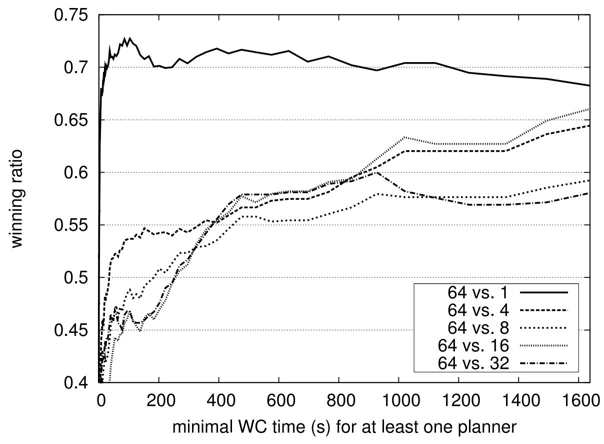

Figure 4: Winning ratios between the 64-thread algorithm versus lower numbers of threads.

egy is faster than the second. We measure how this ratio evolves in function of a "problem hardness" estimate, defined as the WC time (in seconds) used by the slowest of the two compared planners: in the x axis of Fig. 2 a time of t indicates that one of the planners took at least t seconds to solve the problem, and the other took less than t seconds. We therefore identify hardness, in this Figure, with "hard for one of the approaches", which typically means that diversifying away from the heuristic function is more likely to pay. Fig. 3 shows the same ratios but with a logarithmic time line that better shows what happens for small runtimes.

What these data confirm is that when an instance is solved easily by any method, multi-threading does not pay. As the hardness increases, the winning ratio clearly favors the more highly multi-threaded algorithms. The winning ratios of the 64-thread version versus all other versions is shown more specifically on Fig. 4. For problems whose hardness is up to 200 sec., we see for instance that the 8-thread version is winning against the 64-thread one; beyond 200 sec. however the winning ratio clearly favors the 64-thread version. These curves illustrate clearly that the optimal number of threads increases smoothly: for instances that are easily solved we should perhaps use a single-threaded version; as time goes by we should use increasingly many threads, reaching the highest number of threads after a certain time which, judging from the empirical data, should be shortly after 200 sec.

This clearly suggests that some improvements of the basic version of our algorithm are possible; in particular a number of threads adapted dynamically should give us the scalability of multi-threading without penalizing the execution for easier instances. Next section explores this and other improvements of our baseline algorithm.

Improving K-Parallel BFS

In this section we address two questions raised by the previous experimental section. First: is it possible to adjust dynamically the number of threads to the observed instance hardness? In other words: how do we benefit from the scalability brought by a high number of threads while avoiding the overhead this induces for easy instances.

Second, we have explained that multi-threading is powerful because it diversifies the search. There are other, well-known diversification techniques, among which *restarts*. How do we compare the benefits of restarts with multi-threading? Can the two techniques complement each other.

Dynamically Adapting the Number of Threads

The previous analysis shows that the more difficult an instance is for two compared planners, the more successful the use of many threads is; while for easier problems, the overhead induced by too many threads does not pay off. Then naturally comes the idea to dynamically adapt the number of threads to the difficulty of an instance, by gradually increasing the number of threads simultaneously expanding nodes from the *open* list, as time goes by.

We experimented that strategy by increasing the number of threads starting from one thread up to a maximum of 64 threads, following the number of expanded nodes. The bounds on these numbers that decide the increase in the number of threads has been set up by correlating the timing information of Fig. 2, 3 and 4 with the number of nodes effectively expanded by the algorithm.

However, we did not obtain interesting results; this policy being most often, for difficult problems, worse than the initial 64-thread strategy. We think that this can be explained by the fact that doing this way, early mistakes made on top of the search tree cannot be corrected by the use of more threads, as these additional threads only expand the actual best nodes, and not top-level nodes that could orientate search towards completely different directions.

Combining Adaptive Strategy and Restarts

Restart strategies have been proven to be successful techniques in combinatorial search to diversify the way a search space is explored, by increasing the chance to obtain faster a solution. They have been recently successfully used in (Richter, Thayer, and Ruml 2010), which presents an anytime approach that restarts the search from the initial state every time a new solution is found. The authors show that this gives better performance than trying to improve the current best solution starting from the end of the solution path. Restarting has the ability to escape the early mistakes performed by a poorly informed heuristic function. Their technique is applied to PDDL planning and to other domains, with good performance on hard optimization problems.

In order to combine the facts that (i) easy problems are faster solved with few threads, (ii) difficult problems benefit from a larger number of threads, and (iii) the use of many threads should be made at the root of the search tree to better diversify the search from the initial state, we propose to restart the search from scratch each time the number of threads is increased, by simply emptying *open* and *closed*.

This can be made by enclosing the main loop of the parallel best-first search algorithm described earlier, including the `parallel num_threads(`K`)` directive, into another loop which executes the inner loop, and when that inner loop stops due to a predefined bound limit on the number of nodes, flushes the *open* and *closed* lists, increments K and starts again. A Boolean shared between all threads, whose value can be changed by any thread, indicates that a bound limit has been reached. As there is an implicit synchronization point at the end of the block surrounded by the `parallel` directive, all threads can escape the inner loop before the *open* and *closed* lists are emptied.

The simple and completely ad-hoc restart policy based on the number of expanded nodes that we used is the following: 1 thread up to 50 nodes, 4 threads up to 400 nodes, 8 threads up to 3,000 nodes, 16 threads up to 20,000 nodes, 32 threads up to 100,000 nodes, and then 64 threads until the end.

Empirical Study of Adaptive+Restarts

We now report on further experiments that analyze the benefits of the improved strategy.

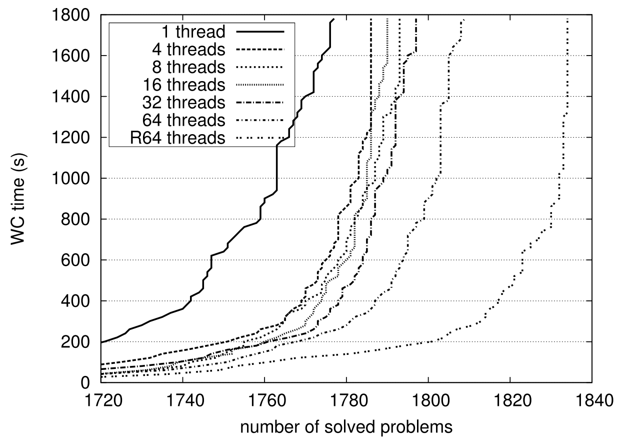

Figure 5: Number of instances solved for various versions of the algorithm. R64 denotes our best algorithm, which includes the adaptive Restart strategy described in the previous section, tuned to gradually reach up to 64 threads.

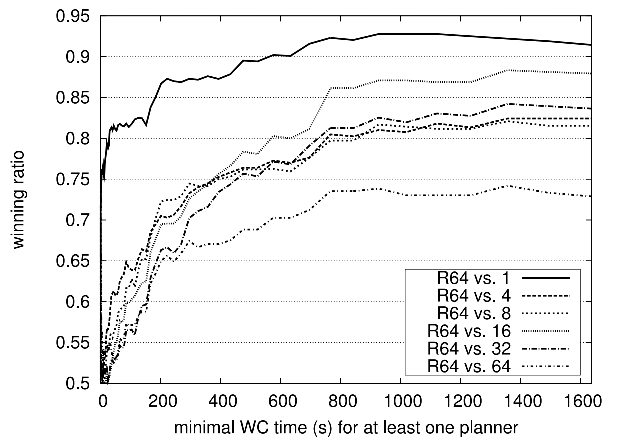

Figure 6: Winning ratios between the adaptive+restarts algorithm versus basic multi-threaded strategies.

Overall performance. We first compare the global performance of the new algorithm with our previous parallel versions without adaptive restarts. Fig. 5 shows that the parallel strategy with adaptive restarts improves significantly the cumulated number of solved problems over the original 64-thread version. Fig. 6 shows that the winning ratio for the new algorithm versus basic multi-threaded versions is improved a lot and mostly avoids poor performance for the easiest instances. Fig. 7 shows a scatter-plot comparison of the WC time between the adaptive restarts strategy versus the basic single-thread approach, showing that we very often obtain *super linear speedups* (when the running times are improved by a factor greater than the number of cores, see (Rao and Kumar 1988)) with the new approach.

A distinction should be made between two improvements. First, the adaptive strategy, i.e. the fact that we start with only one thread, and introduce more threads only gradually: this essentially pays off for instances solved under a certain time threshold, and explains why the curve for the improved

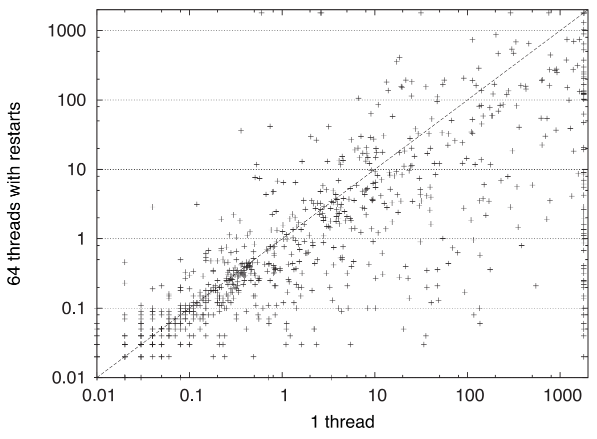

Figure 7: Our best strategy with 64 threads and adaptive restarts versus the basic single-thread strategy (WC time).

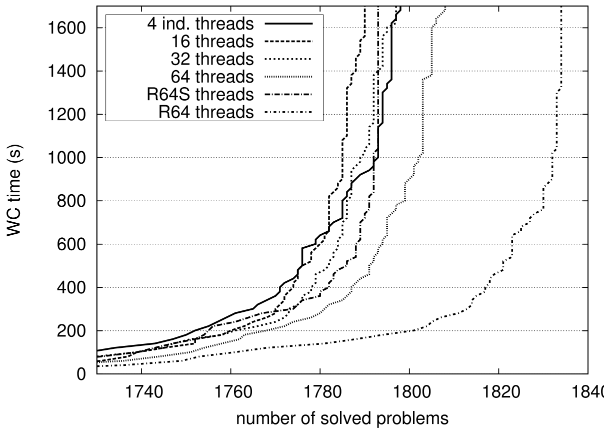

Figure 8: Number of instances solved for various versions of the algorithm. The "4 ind. threads" curve corresponds to our bottom-line 4-core strategy. The "R64S" curve corresponds to our best strategy forced to run on a single core (rather than 4 cores, shown in the curve 'R64').

version is slightly flatter for small runtimes. For instance about 1800 instances are solved in under 200 sec. by the method with adaptive restarts, against 1765 for the initial 64-thread version.

Second, the restarts: they contribute to solving more instances. An important finding is that the source of diversification introduced by restarts is indeed *complementary* to the diversification already introduced by K-Parallel BFS: the version with restarts is able to solve 25 more instances than the version with 64 threads without adaptive restarts (1834 instances versus 1809), and 57 more than the 1-thread version (1777 instances). This is interesting because it shows that the two sources of diversification are not, as could have been conjectured, redundant; instead they mix well together.

Break-down of the improvements. We now provide data that analyze further the various aspects of our algorithms and

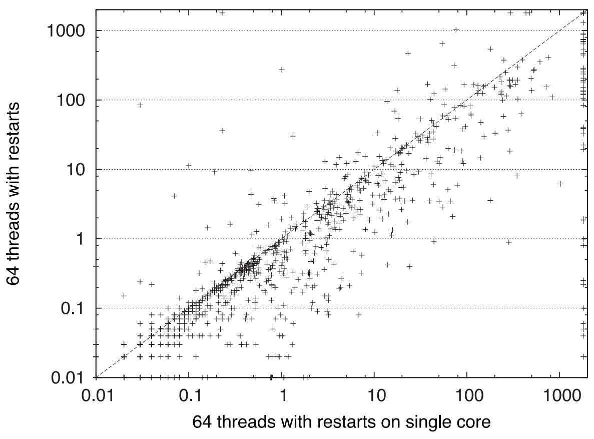

Figure 9: Our best strategy with 64 threads and adaptive restarts run on 1 core versus 4 cores (WC time).

how the gains break-down. Fig. 8 super-imposes to the plot of solved instances the results of two extra experiments.

To estimate how our proposal compares to a truly naive parallel strategy we implemented a "bottom-line" parallel approach working as follows. The algorithm uses a portfolio of four isolate planning algorithms (having different behavior thanks to the stochastic tie-breaking mentioned earlier), each using its own copy of the search space and of all data-structures (similar to the dovetailing strategy of (Valenzano et al. 2010)). The advantage of this extremely simple approach is that no locking or communication of any sort is needed: each search goes ahead independently and we stop as soon as one of them completes. Fig. 8 shows that this approach is significantly less effective than our best adaptive strategy. The bottom-line approach is roughly as effective as our 16-thread version without adaptive restarts. The issue is that it is difficult to see how to improve the naive algorithm further without fundamentally changing its properties, for instance without introducing communications. In contrast our "slightly less naive approach" allows us to smoothly integrate the improvements we suggested in the previous section and which greatly increase its scalability.

To estimate the benefits of parallel hardware we ran our best algorithm in a single-core setting (i.e. by forcing the use of one core, the hardware remaining the same). Again we see that the use of 4 cores allows a better scalability: exploiting them all, we solve 41 more instances than with the same algorithm using one core (1834 versus 1793). Fig. 9 gives a scatter-plot comparison of the execution of our best algorithm on a single core versus 4 cores.

Core utilization. To get a full understanding of the costs of our algorithms we also carefully measured the *core utilization* of all versions. This measures the percentage of the time that each core spends doing effective computation (the rest of the time is the overhead caused by synchronization, system calls, ...) for problems solved by all versions. It is worth distinguishing, here again, between the easy instances and those that take longer to solve. For easy instances the utilization is lower, which is consistent to our observation

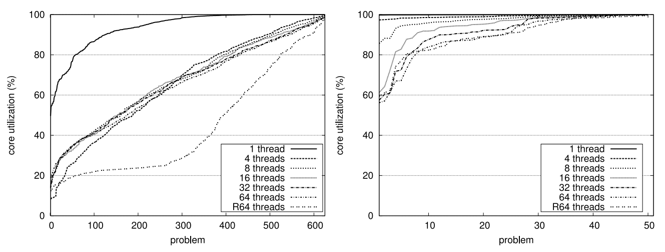

Figure 10: Core utilization of several versions of the algorithm for the 625 instances solved by all versions in less than 10 sec (left) and the 51 instances solved by all versions in more than 10 sec (right).

that the high-level of multi-threading pays mostly when the instance is reasonably difficult, but has overheads for easy instances. Fig. 10 (left-hand side) reports on the utilization measured for the 625 easiest instances with meaningful numbers (search time greater than 0, etc.), solved by all versions in less than 10 sec. (problems on the x axis are ordered differently for each version, by increasing utilization). For the new algorithm, it begins much lower with a plateau at around 25% for the 300 first problems, as core utilization is measured w.r.t. 4 cores, even if 3 of them are unused for the 50 first expanded nodes. Fig. 10 (right-hand side) reports on the utilization for the 51 most difficult instances, solved by all versions in more than 10 sec. The core utilization is high, indicating an efficient use of locking. As explained earlier, part of it is due to our focus on reasonably costly heuristic functions: once a core starts expanding a node, it has a substantial amount of work to do, which guarantees a high utilization.

Summary of the experiments. Compared to a basic sequential version, our algorithm cumulates effects from four improvements: (1) the use of a number of threads that starts low and increases as a higher runtime is needed to solve the problem; (2) the use of a K-Parallel strategy as opposed to the expansion of a single best node; (3) the use of restarts; (4) the use of a 4-core machine as opposed to a single-core one. Overall the experiments reported in this section show the following:

1. The benefits of adapting the number of threads dynamically are mainly in terms of solving faster some (reasonably) easy instances. For instance in Fig. 5 our interpretation is that this improvement helps solving more instances under a given time threshold, say 200 sec. However the adaptive strategy is not the explanation for the resolution of harder instances, where a high level of multi-threading pays off. In our view the adaptive strategy should be used in cases where many instances, including a high percentage of easy ones, need to be solved sequentially and the cumulated runtime matters: in such cases avoiding overhead for small instances is desirable.
2. Part of the gains in our approach are due to the fact that multi-threading simulates a KBFS approach. We have noted that running our approach on a single core, notably, proves to pay off to some extent.
3. An important conclusion is that restarts work well in addition to the K-Parallel strategy.
4. Last, our strategy proves to make effective use of a deployment on a 4-core computer compared to single-core, as evidenced by Fig. 8 and 9.

Conclusion

We demonstrated in this paper that a simple parallelization of a best-first search algorithm, targeted to shared memory multi-core computers, yields nice empirical results. This algorithm is then improved by combining an adaptive strategy for increasing the number of threads concurrently expanding nodes with a restart policy, in order to correct early mistakes when diversifying search with more threads. We shown that this algorithm improves the cumulated number of solved problems, and solves many problems faster than the baseline approaches; especially the instances that present some difficulty, and often with super-linear speedups. We plan for future works to provide more accurate adaptive and restart strategies, which actually sound a bit ad-hoc, and to combine this approach with portfolios of different search algorithms and planning heuristics.

References

Bibaï, J; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proc. ICAPS*, 18–25.

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first heuristic search for multi-core machines. In *Proc. IJCAI*, 449–455.

Evett, M. P.; Hendler, J. A.; Mahanti, A.; and Nau, D. S. 1995. PRA*: Massively parallel heuristic search. *J. Parallel Distrib. Comput.* 25(2):133–143.

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: K-best-first search. *AMAI* 39(1-2):19–39.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. ICAPS*, 10–17.

Niewiadomski, R.; Amaral, J. N.; and Holte, R. C. 2006. Sequential and parallel algorithms for frontier a* with delayed duplicate detection. In *Proc. AAAI*, 1039–1044.

OpenMP. 1997. OpenMP: A proposed standard API for shared memory programming (`http:www.openmp.org`).

Rao, N. and Kumar, V. 1988. Superlinear speedup in parallel state-space search In *Proc. Conf. on FSTTCS*, 161–174.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *Proc. ICAPS*, 137–144.

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition table driven work scheduling in distributed search. In *Proc. AAAI*, 725–731.

Valenzano, R.; Sturtevant, N.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proc. ICAPS*, 177–184.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proc. ICAPS*, 150–160.