Scalable Local Search on Multicore Computers

Amit Kumar* Alexander Nareyek*

Department of Electrical and Computer Engineering
National University of Singapore, 4 Engineering Drive 3, Singapore 117576
amit_kumar@nus.edu.sg, alex@ai-center.com

1 Introduction

Practical computing is undergoing a major shift towards parallelism due to exhaustion of performance gains of other approaches to computer design like increasing clock speed [4]. In a few years, parallel *multicore* processors with hundreds of concurrent processing threads will be prevalent in common desktop and laptop computers, while web servers having *manycore* processors will feature even thousands of concurrent processing threads [4]. In this paper, we discuss the question of how to effectively utilize the resources of multicore computers for local search. Local-search methods have been effectively used for large combinatorial problems [22] including complex structures, dynamic changes and anytime computations [1, 14].

A problem for local search is often stated in terms of minimization of a *cost function* $C : S \rightarrow \mathbb{N} \cup \{0\}$ defined on the solution space S. In the local-search procedure [21] for a minimization problem, a problem-specific *neighborhood* $N : S \rightarrow P(S)$ where $P(S)$ is the power set of S, is defined for search. Neighborhood $N(s)$ of one or more solutions s is searched and each s is *selectively* replaced by another solution $s^{'} \in N(s)$. This process of replacement is called a *move*. The goal of a local-search procedure is to quickly reach the global minimum of the objective function. Quick exploration of the neighborhood and avoidance of getting stuck within one of the *local minima* (solutions s_m such that for all $s^{'} \in N(s_m)$, $C(s^{'}) > C(s_m)$) are important. To get out of a local minimum, the search procedure may selectively move to a worse-off solution $s^{'} \in N(s_m)$ in the hope that $s^{'}$ will later lead to a better value of cost function.

In Section 2 we discuss the benefits and challenges of multicore parallelization. In the context of parallel local search, broadly there have been three major approaches [2, 7]: speeding up each single move, several moves in parallel and several independent neighborhood searches. We discuss how these approaches fit for implementation on multicore computers in section 3. In large practical problems, often the number of neighbors $\#N(s)$ is so large as to rule out an exact evaluation of the cost function of all the solutions in the neighborhood within reasonable time. In such cases, the search mechanism may evaluate only a limited number of neighborhood solutions, or perform quick

*This research was supported by a joint grant of Singapore's Ministry of Education / Academic Research Fund and the National University of Singapore (RG-263-001-148).

but inaccurate approximation of cost function of the neighborhood solutions. We discuss these choices further in Section 3. In Section 4 we describe a case-study of using local search for action planning. Our experiments described in Section 4.2 compare the behavior of each parallelization approach for action planning with varying parameters.

2 The Benefits and Challenges of Multicore Computing

Using more processors may help reduce *latency* of processing a computational task by distribution of subtasks and their concurrent processing. This reduction in latency is measured by *speedup* – the ratio of the time taken to complete a given sequential computational task on a uniprocessor divided by the time taken to complete the equivalent parallelized computational task on a parallel computer [10] (see Figure 1). Sometimes superlinear speedups are also possible. *Scalability* is the ability of the system to increase speedup when computational resources are added. Higher *dependability* is achieved by utilizing computational resources to explore multiple options. Even on a uniprocessor, using multiple threads yields anytime solutions to multiple options.

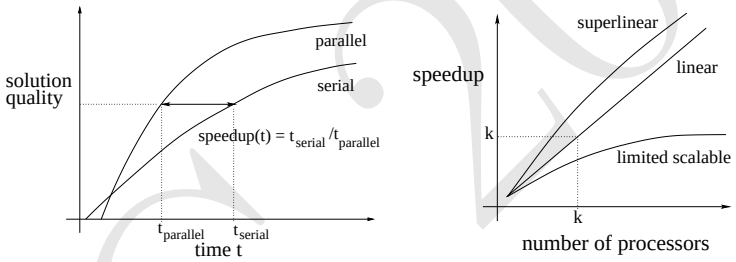

Figure 1: Speedup and scalability of parallel computation

However, achieving the above targets offers challenges unseen in sequential computation. *Amdahl's Law* [10] says that even a small part of sequential code can bound the maximum speedup achievable by the multiple processors. Hence to achieve a high speedup, a balanced distribution of computational tasks and minimization of sequential parts is needed. Speedup and scalability are bounded by the number of implemented concurrent computational tasks if it is less than the available physical concurrency. The overhead of communication and synchronization may increase with problem size if the decomposition into (independent) computational tasks is poor. This may bound scalability. *Granularity* is used to denote the size of a computational task. A computational task of higher granularity may afford a higher overhead on communication.

Caching. A primary aspect of modern computer architectures is that the processors and the main memory elements are far apart. A main memory access takes about a hundred clock cycles while a floating point multiplication takes about four [4]. Therefore, most computers implement automatic *caching* to a temporary storage close to processors. If the data or instructions needed for computation are not frequently found within the cache, the processor parallelism is not effective for speedup as a lot of time is lost in fetching data from main memory. To enhance performance, cache locality may be exploited by performing related computations on the same processor.

3 Parallelization Strategies

In this section, we first discuss parallelization strategies for local search and then some important aspects of efficient implementation on multicores. The parallelization strategies can be classified into three categories: *parallel multistart*, *parallel moves* and *move acceleration*. In parallel multistart approach, multiple searches (with or without cooperation) are performed in parallel. In parallel moves approach, the solution search space is decomposed, moves are performed in parallel and then the moves are composed into one solution. Move acceleration is used to speed up a move by performing cost function evaluations faster in parallel. In this section, we analyze these parallelization strategies in the context of multicores. For illustration, we take an example of an integer programming problem of minimizing $z = 5x_1 + 3x_2$ subject to constraints c_1: $x_1 + x_2 + x_3 \geq 4$, c_2: $5x_1 + 2x_3 + x_4 \geq 10$, c_3: $x_2 + 2x_4 \geq 5$, c_4: $x_i \geq 0, i = 1, 2, 3, 4$ using local search. Let $V = x_i$, the set of variables.

Parallel Multistart. In parallel multistart, the interaction between the multiple searches is limited only to sharing of the learnt knowledge about good solutions, which is an efficient mechanism to generate good quality solutions from several moderate quality solutions. For small and moderately sized problems, this approach is a promising candidate for generating parallel tasks needed for scalability. However, due to duplication of all data structures related to search, such approaches do not scale to very large problems as they consume a lot of memory.

Parallel Moves. One major limitation of this approach is that since the improvement iterations do not a priori know the effect of each other, *interferences* between moves which worsen the effects of the parallel moves may arise. In our example, suppose $x_1 = x_2 = 1, x_3 = x_4 = 0$. Then $z = 8$ and constraints 1, 2 and 3 are violated. Suppose the moves $x_1 = 3$ (to satisfy constraint 1) and $x_2 = 0$ (to reduce z) are made in parallel. After the moves, constraint 1 is still violated and the value of z increased. To reduce the effect of interferences, depending on the context, one of the following strategies may be used: 1) reduce the degree of parallelism of moves, 2) only independent moves may be scheduled concurrently (selection of independent moves is problem-dependent), or 3) interferences may be fixed by the local-search procedure after they happen, that is, accept the overhead of interferences.

A scheme to handle interferences may combine more than one strategy. For example, we define a measure $closeness : V \times V \rightarrow \mathbb{N} \cup \{0\}$ between every pair of variables as the number of constraints both occur together in. For example, $closeness(x_1, x_2) = 2$ as x_1 and x_2 occur together in the objective function and in c_1, and $closeness(x_1, x_1) = 4$ as x_1 occurs in the objective function and three constraints. We also define a measure of closeness between every constraint, for example the maximum of the closeness of each pair of variables in the constraints, $closeness(c_1, c_2) = 4$ due to x_1. A central scheduler then maintains a list of available moves (we assume that each move changes the value of only one variable), and selects the next move based on the least closeness of the affected constraints from the affected constraints of currently processed moves.

Move Acceleration. Cost function evaluations before making a move are one of the most frequent function calls during local search. Parallel cost function evaluation offers a rich source of parallelism, but at a low granularity level. This implies that using this parallelization alone may not yield much speedup.

For large neighborhoods and for real-time requirements, there might not be enough time to

exactly evaluate *all* neighborhood solutions before making a move (see Figure 2). In our example, suppose there are additional constraints $x_i < 100$. These constraints do not affect the problem much as they are always satisfied for the small values of variables relevant here. In a large problem, there may be many such constraints. The search mechanism may infer that such constraints are almost always satisfied for whatever solutions it has searched before. It may then reduce the frequency of checking such constraints for newer solutions. The other dimension of limiting cost function evaluations is: evaluating few versus many neighborhood solutions.

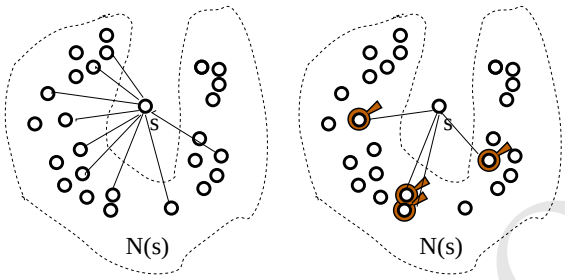

Figure 2: Choices in limiting cost function evaluations for large neighborhoods: (Left) Few versus many neighborhood solutions, (Right) More verus less exact evaluation of cost function.

Distribution of Computational Tasks. The distribution of computational tasks should exploit the locality of communication requirements of tasks (see Section 2). For parallel moves, the distribution based on *closeness* of variables and constraints (variables and constraints with high *closeness* to close-by or same processors) is favored by two factors: first, the data dependency between a pair of moves is high for moves related to constraints with high *closeness* value, and second, to reduce interferences, two moves with high *closeness* should not be scheduled in parallel. Cache-affinity between computational tasks (see [18]) may be based on *closeness*.

4 Case Study

In this section, we first describe the architecture of an action planning [16] system based on parallel local search. Action planning studies how an intelligent agent can formulate a plan of *actions* to achieve specified goals. More than scheduling, a planning system must also determine which actions out of a given set should be taken. Later in the section, we compare the various parallelizations in the context of this system.

We use an example of the Logistics Domain [5] to describe our planning system. The goal is to produce a plan to transport some packages from their current depot location in a city to another depot – possibly in another city – using trucks and airplanes (only airplanes can transport packages between cities). Suppose one goal G is to deliver a package $pkg1$ initially at depot $dpt1$ in city $cty1$ to another depot $dpt2$ in city $cty2$ within a day (both $dpt1$ and $dpt2$ are not airports). Actions like $loadPackageIntoTruck$ are given and can be inserted into the plan by the planning system in order to achieve the goal. Each action may have conditions, for example for execution of $loadPackageIntoTruck$ action, a condition is that the package must be at the same location as the truck.

We first describe a sequential algorithm to solve the above problem based on [14], and then discuss our framework to parallelize the sequential algorithm in section 4.1. The sequential Al-

gorithm is based on iterative repair of the plan: in each iteration, a repair of an imperfection in plan is carried out. An imperfection in plan is called a *Cost*. For example, a cost may be due to an unsatisfied condition of an action in plan. *CostManager* "manages" Costs. We define several *CostCenter*s, each of which has the responsibility to repair Costs related to it. The PlanManager has the knowledge about the objects and maintains the plan by receiving requests to repair the plan from CostCenters and forwarding the resulting changes to CostManager and CostCenters.

Initially, the plan is empty and the unsatisfied goal G results in a Cost (let's call it $cst1$). We describe one iteration of the algorithm. First, CostManager selects one of CostCenters and asks it to repair one of its Costs. CostCenters are defined for each attribute of each object; therefore, the CostCenter corresponding to location $lctn$ attribute of the object $pkg1$ (henceforth referred to as $pkg1.lctn$) has the responsibility to repair the cost $cst1$. Suppose the CostManager asked $pkg1.lctn$ to repair $cst1$, and the CostCenter decides to repair $cst1$. Each CostCenter has a set of heuristics (like insert, change or remove an action from the current plan). The CostCenter selects one heuristic to repair each kind of Cost. For example, suppose the CostCenter $pkg1.lctn$ selects the heuristic "insert an action" into the plan to satisfy $cst1$. One among the actions that the CostCenter evaluates is $loadPackageIntoTruck$. The CostCenter evaluates various options for the parameters of the action, like the start time, and the truck to load the package into. To evaluate, for example, one of these options, the CostCenter consults the PlanManager requesting it to return a close-by truck at the time the action starts, since the PlanManager has the knowledge about the objects in the domain. In other cases, the CostCenter also consults other CostCenters for evaluation of moves.

After several such evaluations, the CostCenter forwards a change, for example "insert into plan $loadPackageIntoTruck$ action, the package $pkg1$ into truck $t1$, at location $dpt1$ and time $9am$" to the PlanManager. The PlanManager sends updates to all the concerned CostCenters and the CostManager about this update in plan – in our case, for example, an update is sent to $pkg1.lctn$ that its value after $9am$ is now $t1$. Addition of the action may give rise to new costs, for example due to an unsatisfied condition of the $loadPackageIntoTruck$ action. In the current case, it leads to another cost $cst2$ corresponding to $t1$ being at $dpt1$ at $9am$. These cost updates are forwarded to the CostManager. This completes one iteration. Such iterations are repeated to repair the imperfections (Costs) in the plan.

4.1 Applying Parallelization

Parallelization may be used at plan level, repair level or within-repair level. Based on the discussion in Section 3, at plan level, parallel multistarts for independent plan searches may be used. At repair level, parallel moves (or repairs) may be used. Move acceleration is used at within-repair level.

The algorithm of parallel moves at repair level is more involved than the other strategies: The CostManager selects some CostCenters for repairs. The selected CostCenters concurrently analyze the plan and select changes, and forward to the PlanManager. The PlanManager applies the changes received from the CostCenters one by one into the plan. Sometimes the received repairs are based on outdated information and may not be applicable to the plan. One option here is whether the changes should be applied into the plan after synchronization. We don't use synchronization since the time taken by the heuristics have a huge variance. As in section 3, the CostManager uses a heuristic measure of *closeness* between CostCenters for scheduling repairs; if there is an action within the plan with precondition and effects on two attributes, the *closeness* of the two attribute

CostCenters is incremented by 1.

Our implementation is in C++ and uses the Intel Threading Building Blocks library. The library implements cache-efficient load balancing (using work stealing) and affinity-based task scheduling [18].

4.2 Experiments and Discussion

In this section, we first describe the setup of our experiments on various parallelization strategies, and then observations and discussion of the results. We ran 80 runs for each test configuration with different random seeds on computers with Intel Core 2 Quad 2.4GHz CPU (4 processors), 3.5GB RAM and 4MB L2 Cache running Windows Vista operating system. Each run was stopped after 40,000 iterations (i.e., with 2 threads, there will be 2 sets of independent 40,000 iterations). For each set of experiments we use 2 problems (4-0 with 4 packages and 20-0 with 20 packages) from Track 2 of the ICAPS 2000 Logistics Domain [5], enhanced with durative actions.

Parallel Multistarts. In Figure 3, we compare the wall-clock time taken by our implementation in order to reach a certain solution quality by runs with different number of threads. We note that in both problems, the performance of 8 and 10 threads is weak at the start, but catches up as search proceeds. The performance of using 4 to 10 threads is comparable near the end of the test runs for the smaller problem. For the larger problem, near the end, 10 threads perform much worse than 4 to 8 threads. Towards the end, the memory usage increases a lot due to a high space complexity. We verified that this results in increased page faults. Towards the end, high memory usage offsets the benefit of diversification.

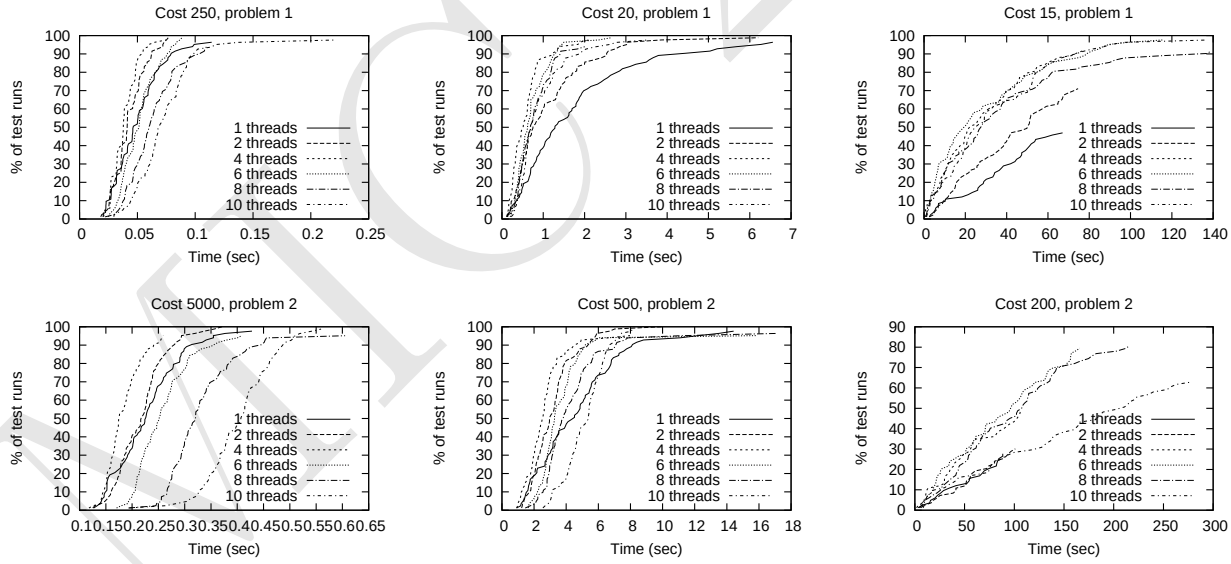

Figure 3: Percentage of parallel multistart test runs that achieve a cost/minimization value of 250, 20 and 15 for problem 1, and 5,000, 500 and 200 for problem 2.

Limiting Cost Function Evaluation. In the following experiment, we compare searches that examine 20, 50, 80, and 110 neighbors for problem 1 using 1, 2 or 4 threads (Figure 4) and 80, 110, 140 and 170 neighbors for problem 2 (Figure 5) in each iteration, before making a move. For the

smaller problem, towards the end 20-80 neighbors perform almost the same for 4 threads. In the larger problem, the larger number of neighbors perform worse at the start but better towards the end. In both cases, higher number of neighbors is favored while employing more threads.

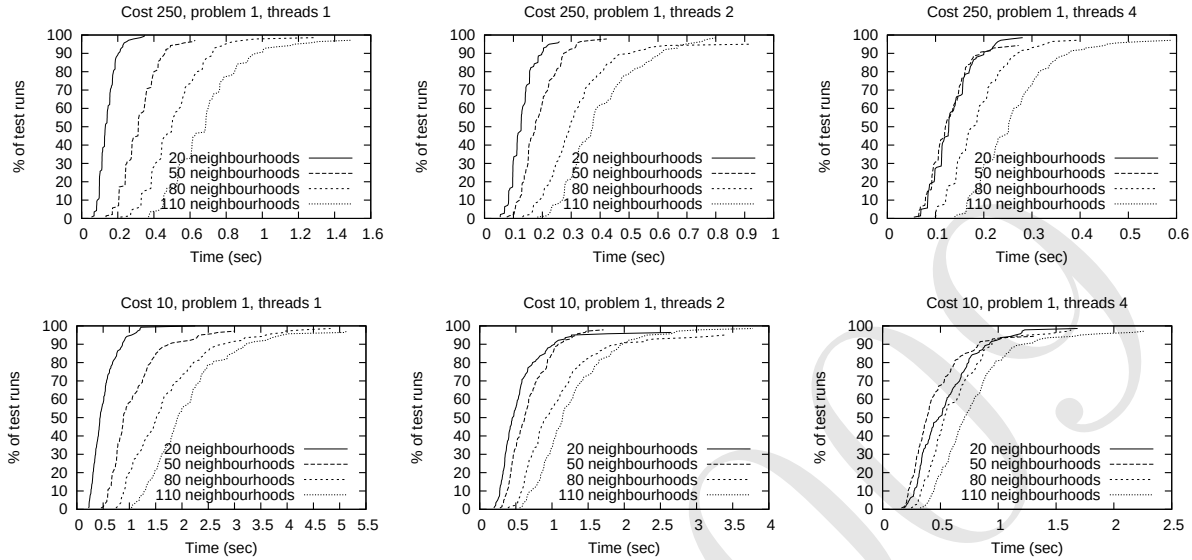

Figure 4: Percentage of "number of neighbors" test runs that achieve a cost/minimization value of 250, and 10 for problem 1 using 1, 2 and 4 threads.

We note that during the plan construction, initially a lower number of neighbor evaluations perform better. As the construction proceeds, the memory usage increases too. Therefore, initially parallel multistarts are more useful than towards the very end. Towards the end, more threads may be allocated towards an increasing number of neighbor evaluations needed to improve the solution.

5 Related and Future Work

[7, 2] describe various parallelization strategies for metaheuristics. Our strategies correspond to type 3, type 2 and type 1 strategy in the terminology of [7]. In the context of parallel moves, [20, 19] analyze the effect of connectiveness of problem structure on optimal amount of parallelism. Our centralized scheme for handling inferences is likely to be more efficient than the distributed asynchronous search approaches [24] as the scheduler has the complete picture of the available moves.

Parallel planning has previously been studied mostly in the context of inherently distributed sensing or execution [8]. Our paper discusses parallelization in the context of lower communication times within a multicore system, and the control of distribution is in the hands of the designer. Previous local-search-based planning approaches include planning as satisfiability [9], genetic programming [12], planning by rewriting [3] and the ASPEN system for space missions [17]. However, none of these focus on parallelization of search for planning.

Several approaches [11, 6, 23] to parallel planning divide the search space into heuristically independent regions based on actions, states, or subgoals. When the complete subplans are formed

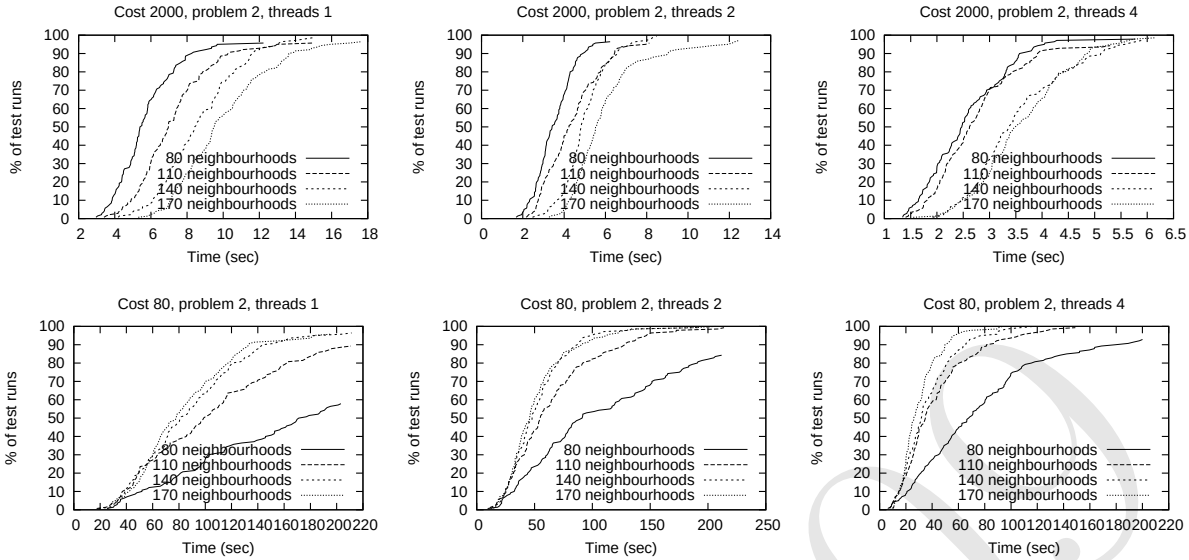

Figure 5: Percentage of "number of neighbors" test runs that achieve a cost/minimization value of 2,000 and 80 for problem 2 using 1, 2 and 4 threads.

for each of the regions, the subplans are merged. The problem of merging large subplans has a huge run-time complexity. To save this complexity, in our current implementation, concurrent repairs are made by CostCenters into a single integrated plan. The intermediate approach of formation of small subplans by the CostCenters and then merging into the overall plan also looks promising.

We noted that the best parallelization in our plan-construction experiments depends on the stage of the search. Many choices during search can be improved if based on the history of the search. How much parallelization should be devoted to each of the parallelization strategies? How many and which neighborhood solutions should be evaluated before making a move? These questions can benefit from machine learning techniques that can dynamically adjust the goodness of each choice (see, for example [15]) during runtime. In this context, [7] ask: How can learning interactions be used to globally direct search? Our future work will be devoted to these questions in the context of multicore.

Parallelization is affected by other aspects of real-time planning as well. For example, often plan execution must start quite soon – in this case, parallel evaluation of the plan options allows us to explore more options or details. Further, dynamic external changes affect all the parallel multistarts and they must therefore be updated. This is a redundant effort. We will explore such issues in the future.

In large problems, there may be a huge number of constraints that do not affect the current moves. In such cases, inexact faster cost function evaluations before making a move would be useful.

6 Conclusion

In this paper, we analyzed various parallelization options to solve large problems on the now ubiquitous multicore computers. We presented ways to efficiently implement parallel local search on

multicores. We also evaluated how the search evolves as it proceeds. In our case study, we present the architecture of an action planning system amenable to parallelization. Results of our experiments show that different strategies yield a good or not so good performance depending on the stage of the search. A search strategy that dynamically adjusts to the current stage of the search would fare better. We have implemented our parallelization in a planning system called Crackpot, which evolved out of the Excalibur system [13].

References

[1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[2] E. Alba, E.-G. Talbi, G. Luque, and N. Melab. *Parallel Metaheuristics: A New Class of Algorithms*, pages 79–103. Wiley-Interscience, 2005.

[3] J. L. Ambite and C. A. Knoblock. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 706–713, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[4] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[5] C. F. Bacchus. AIPS 2000 planning competition. *AI Magazine*, 22(3):47–56, 2001.

[6] R. Berlanga. Parallel planning in structured domains. In *Proceedings of the 16th Workshop of the UK planning and scheduling SIG '97*, 1997.

[7] T. G. Crainic and M. Toulouse. In *Handbook of Metaheuristics*, pages 475–513. Springer New York, 2003.

[8] E. H. Durfee. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, chapter Distributed Problem Solving and Planning. MIT Press, 2000.

[9] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*. Addition Wesley, 2nd edition, Boston, MA, USA, 2003.

[11] A. L. Lansky. Localized planning with action-based constraints. *Artif. Intell.*, 98(1-2):49–136, 1998.

[12] I. Muslea. A general-purpose AI planning system based on the genetic programming paradigm. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 157–164, Stanford University, CA, USA, 13–16 1997. Stanford Bookstore.

[13] A. Nareyek. *Constraint-Based Agents - An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds*. LNAI 2062. Springer, 2001.

[14] A. Nareyek. Using global constraints for local search. In E. C. Freuder and R. J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS*, pages 9–28, Boston, MA, USA, 2001. American Mathematical Society.

[15] A. Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In M. G. C. Resende and J. P. de Sousa, editors, *Metaheuristics: computer decision-making*, pages 523–544, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[16] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[17] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. Iterative repair planning for spacecraft operations using the ASPEN system. In M. Perry, editor, *In Proceedings of the Fifth International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS)*, volume 440 of *ESA Special Publication*, pages 99–, Aug. 1999.

[18] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *IPDPS 2008*, pages 1–8, 2008.

[19] A. Roli. Impact of structure in parallel local search for SAT. In *SAT 2002 – Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, Cincinnati, Ohio, USA, 2002.

[20] A. Roli and C. Blum. Critical parallelization of local search for max-sat. In *AI*IA 01: Proceedings of the 7th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, pages 147–158, London, UK, 2001. Springer-Verlag.

[21] R. J. M. Vaessens and E. H. L. Aarts. A local search template. Technical report, Computers and Operations Research, 1995.

[22] S. Voß. Meta-heuristics: The state of the art. In *Local Search for Planning and Scheduling*, volume 2148/2001 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2001.

[23] Q. Yang. *Intelligent Planning: A Decomposition and Abstraction Based Approach to Classical Planning*. Springer-Verlag, Berlin, 1997.

[24] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In U. Montanari and F. Rossi, editors, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 88–102, London, UK, 1995. Springer-Verlag.