Lessons Learned from Prototyping Parallel Computer Architectures for AI Algorithms

Bernhard Rinner
Institute for Technical Informatics
Technical University Graz
A-8010 Graz, AUSTRIA
b.rinner@computer.org

Reinhard Schneider
Institute for Technical Informatics
Technical University Graz
A-8010 Graz, AUSTRIA
schneider@iti.tu-graz.ac.at

Abstract

For many years algorithms from the field of artificial intelligence (AI) have been targeted for parallelization. This paper reports on our experience in parallelizing and distributing AI algorithms, i.e., the design and prototype implementation of parallel computer architectures for AI algorithms. While the presented prototypes have been designed for specific applications, the underlying algorithms are quite general and frequently-used AI search methods, i.e., constraint satisfaction and simulated annealing. Our prototypes achieve speedup factors of several orders of magnitude.

keywords: parallel and distributed AI; specialized computer architecture; constraint satisfaction; simulated annealing

Introduction

For many years algorithms from the field of artificial intelligence (AI) have been targeted for parallelization. The general parallelization approach is to partition the search problem of the AI algorithm and distribute the subproblems among multiple processing nodes. Over the years significant progress has been achieved in the parallelization and the development of specialized architectures for AI applications (Moldovan *et al.* 1992) (IEEE Computer 1992) (Higuchi et al. 1994). In most cases, the need for increased performance was the driving factor for the parallelization effort.

Parallelization has also been applied as a key technique for performance improvement in many different areas, such as ordinary and partial differential equations, linear algebra and digital signal processing. AI algorithms, however, have a different characteristic than the numerical algorithms in the areas listed above. This difference results in a complex parallelization process which is probably more complicated than used for numerical algorithms.

This paper reports on our experience in parallelizing and distributing AI algorithms, i.e., the design and prototype implementation of parallel computer architectures for AI algorithms. Our approach extends the parallelization of AI algorithms using general-purpose multiprocessors and results in a higher performance and an improved scalability than the general-purpose multiprocessor approach. While the presented prototypes have been designed for specific applications, the underlying algorithms are quite general and frequently-used AI search methods, i.e., *constraint satisfaction* and *simulated annealing*. Therefore, our results might be of interest for a broad community within the fields of AI, parallel processing and computer architecture.

Design of Specialized Computer Architectures

This section presents the key steps in the design of specialized computer architectures with emphasis on the design differences between AI and numerical algorithms.

Algorithm Analysis

The goal of the algorithm analysis is to detect runtime-dominant portions which can be easily distributed among several processing nodes. Thus, the analysis deals with (i) profiling, (ii) identifying the hierarchical structure and the control flow of the algorithm, and (iii) determining the dependencies between algorithmic units, such as modules, functions and statements.

AI algorithms can be typically characterized as follows: First, AI algorithms offer only low to medium inherent data parallelism. Second, they have a high algorithmic complexity, e.g., NP-complete. Finally, the control structures of AI algorithms are often complex and data-dependent which means that the order of execution is only determined at runtime. This characteristic is different to numerical algorithms and has a great influence on the remaining steps in the design of parallel computer architectures.

Partitioning

The goal of partitioning is to identify tasks, i.e., parts of the overall algorithm which can be easily distributed among the processing nodes. To ease the distribution the tasks should have no or only a few dependencies between them. Due to the characteristic of AI algorithms, i.e., low data parallelism and data-dependent control structures, the partitioning may not be defined

before runtime. Therefore, dynamic partitioning may be required.

Architecture Design

The goal of architecture design is to find processing elements that are suitable for executing the tasks and a topology that optimally matches the structure of the partitioned algorithm. Furthermore, the bandwidth and latency of the communication network should correspond to the task structure and granularity. Another goal of the architecture design is scalability, i.e., that the performance of the computer architecture can be adapted to the problem complexity by adding more processing elements.

A problem of the architecture design is that the partitioning may not be known before runtime and the architecture cannot be modified at runtime.[1] In most cases, the (static) computer architecture, therefore, is only a compromise over all partitionings generated during runtime.

Mapping and Scheduling

The goal of mapping and scheduling is to determine where and when each task is to be executed. Mapping and scheduling is in general NP-complete. Since the tasks may not be known before runtime, dynamic mapping and scheduling is necessary. To keep the runtime required for mapping and scheduling small, approximation methods or heuristics can be applied resulting in a suboptimal mapping and scheduling.

Prototypes

This section presents two parallel computer architectures for AI algorithms prototyped at our research institution.

Multiprocessor Architecture for Qualitative Simulation

In this project, a special-purpose computer architecture for the qualitative simulator QSim (Kuipers 1994) has been developed. Qualitative simulation involves deriving a dynamic system's behavior given only weak and incomplete information about it. This technique excels at predicting all physically behaviors derivable from this incomplete knowledge.

The generation of all possible behaviors basically requires the solution of two different problems: First, given a qualitative model and partial information about the initial state, determine all complete, consistent qualitative states. Second, given a qualitative model and a complete qualitative state, determine its immediate successor states. QSim solves both problems by applying well-known AI techniques, i.e., constraint propagation and constraint satisfaction (Mackworth 1992). During a simulation run, many different constraint satisfaction problems (CSP) must be solved. The CSPs are defined by the qualitative model and the values of the current state.

The design of this application-specific computer architecture is based on an analysis of the QSim algorithm including extensive runtime measurements taken from a QSim implementation (Platzner, Rinner, & Weiss 2000). We apply two strategies for performance improvements: (i) parallelizing and mapping more complex QSim functions onto a multiprocessor and (ii) supporting small but runtime-intensive functions with specialized processing elements. The first strategy partitions the CSPs into tasks that can be executed in parallel using a master/slave topology. Since the CSPs are only known at runtime, dynamic partitioning is necessary. Our dynamic partitioning method VBP (Platzner & Rinner 1998) divides the search space into an arbitrary number of independent subspaces. Dynamic partitioning enforces dynamic mapping and scheduling. To keep the overhead small, we apply an online list scheduling strategy which is simple and is able to guarantee a worst-case deterioration from the optimal schedule. The second strategy accelerates low-level functions by implementing them in hardware, i.e., in specialized coprocessors. The main features of the coprocessor design are (i) exploitation of parallelism at the instruction- and operation-level, (ii) use of optimized data types, (iii) and use of customized on-chip memory architectures.

By combining both design strategies, we achieve a heterogeneous multiprocessor architecture (see Figure 1). We chose the digital signal processor TMS320C40 as the processing element because of its six independent communication channels and its high I/O performance. We prototyped the coprocessors on field-programmable gate arrays (FPGAs). Figure 2 compares the speedup factors of the QSim computer architecture to a standard QSim implementation in Lisp using three different simulation models (STLG, RCS and QSEA) (Platzner, Rinner, & Weiss 1997). The speedup factors of three different architectures are shown to demonstrate the scalability of our QSim computer architecture: the QSim computer architecture consisting of a single processing element (single PE), 7 processing elements (7 PEs), and 7 processing elements with coprocessor support (7 PEs + coprocessors).

A Parallel Computer Architecture for Simulated Annealing

In this project, a special-purpose computer architecture for simulated annealing (SA) has been developed. SA is an optimization algorithm based on local search (Aarts & Lenstra 1997). This technique is able to solve NP–complete combinatorial optimization (CO) problems. SA searches the solution space by local search within the neighborhood of the current solution. The generation of a new solution as well as the possible acceptance of deteriorations are performed randomly. The sequence of solutions generated during the search forms

[1] Only if the architecture is capable of *dynamic reconfiguration* may the functionality and interconnection structure be modified at runtime.

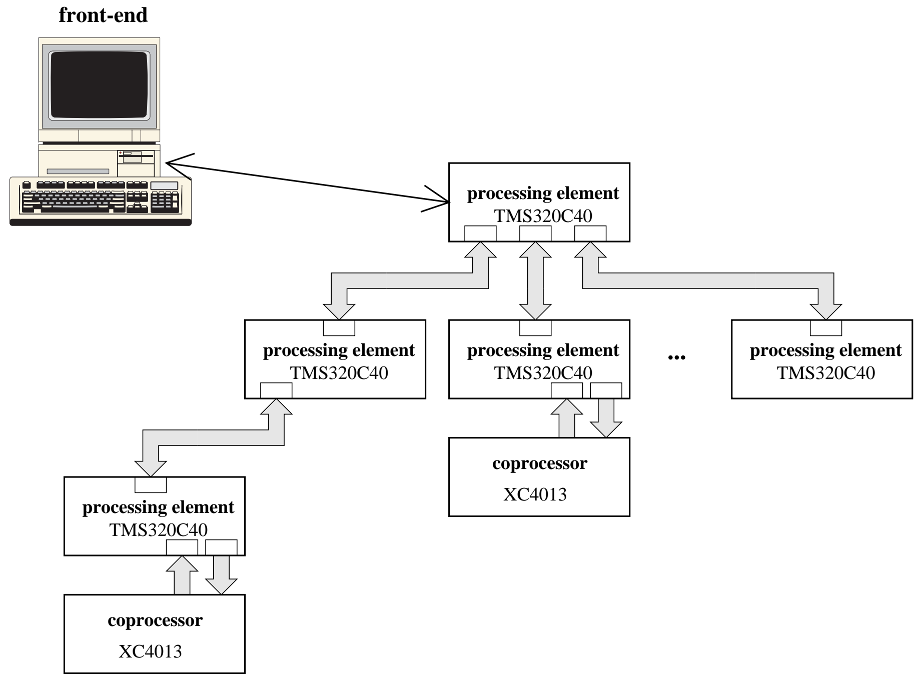

Figure 1: Example of the multiprocessor architecture for the qualitative simulator QSim.

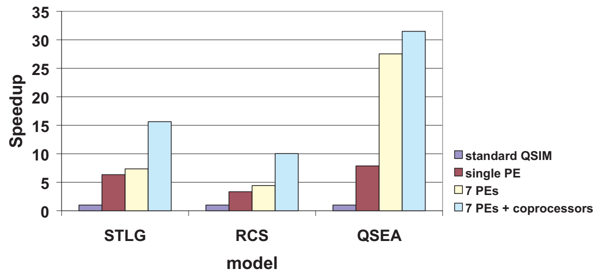

Figure 2: Speedup factors of the QSim computer architecture compared to a standard QSim implementation in Lisp.

a Markov chain. The convergence of the algorithm is proved by Markov theory. The quality of the solution is rated by a cost function, which assigns a scalar value to each solution.

The inner loop of SA performs one step in the Markov–chain, consisting of (i) the random selection of a new solution in the neighborhood, (ii) the generation of the selected solution, (iii) the computation of the cost changes, and (iv) the acceptance decision. Typically, billions of iterations are necessary to reach the optimal solution.

The special computer architecture for SA shortens the computation time by (i) parallel search on multiple processing nodes and (ii) by instruction set extensions of a standard processor core supporting fast movement in the solution space.

We investigated several SA parallelization techniques (Schmid & Schneider 1999) which either distribute parts of a single step, i.e., selection (S), generation (G), cost computation (C), and acceptance (A), a complete step, or a sequence of steps among the available processing elements (see Figure 3).

- The *decision tree decomposition* technique uses speculative computation to overlap parts of successive iteration steps. If the solution i of iteration n (i_n) is rejected, the starting solution of both iterations n and $n+1$ is the same; both iterations can start in parallel. If i_n is accepted, iteration $n+1$ must start from the new solution generated in n. In this case, the calculation for i_{n+1} starts immediately after step (G) of iteration n.
- In the *one–chain* technique all slave processors work on the same solution i_n. The master processor selects either the first, the best or a random solution out of all accepted ones as new starting solution for $n+1$. The use of multiple processors increases the number of solutions checked in the neighborhood of i_n without changing the behavior of sequential SA.
- The *division* technique works similar to the one–chain technique, but synchronization is done only after a rather long sequence of steps computed individually at each slave. The master either selects the first or the best solution as the new starting solution. This results in less communication and a higher speedup.

The processor core extensions (Schneider & Weiss 2000) for fast CO basically consist of an internal memory which efficiently represents one solution of a CO

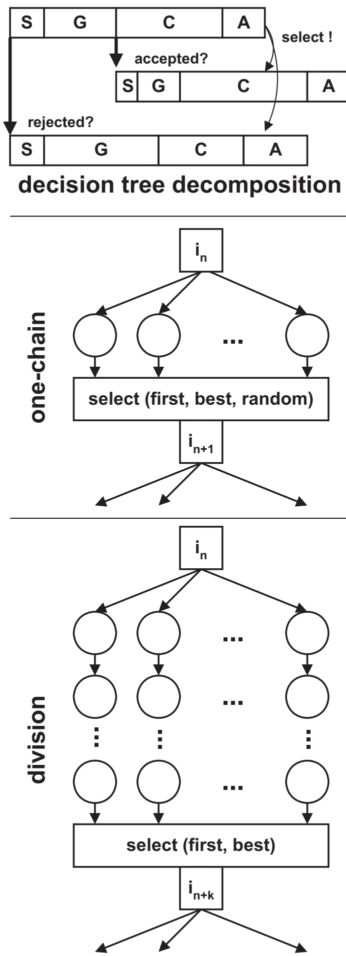

Figure 3: SA parallelization techniques either distribute parts of a single step (top), complete steps (middle) or sequences of steps (bottom).

problem and instructions for memory manipulation. Other hardware modules, such as a pseudo random number generator, additionally, exploit instruction level parallelism. Speculative parallel SA techniques are supported by synchronization and communication extensions, which allow to reconfigure dynamically the computer architecture, as well as by an acceptance prediction unit.

A prototype of the architecture is implemented and evaluated in parts on different platforms. The parallelization techniques are implemented on a multiprocessor system consisting of 8 TMS320C40 digital signal processors by Texas Instruments. The hardware extension modules are implemented and emulated on programmable hardware (FPGA). The experimental evaluation shows that the special–purpose computer architecture is capable of reducing the run–time by up to three orders of magnitude (Fig. 4).

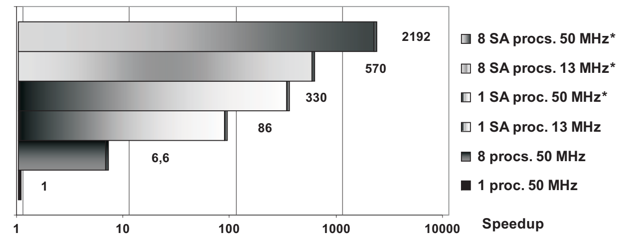

Figure 4: Speedup factors of the SA computer architecture using the division technique compared to a software solution on a single processor. As benchmark the traveling salesman problem was used. Extrapolated speedup factors are marked with an asterisk.

Lessons Learned

This section presents our key lessons learned from prototyping parallel computer architectures for AI search algorithms.

#1 Keep Overhead Introduced by Dynamic Partitioning, Mapping and Scheduling Small

Due to the high data-dependency of AI algorithms, dynamic partitioning, mapping and scheduling is often required in order to parallelize the algorithm. Then, these functions have to be executed at runtime as individual sequential tasks. From the viewpoint of performance improvement, simple but fast methods for partitioning, mapping and scheduling are in most cases preferable over more complex methods. The deteriorations in partitioning, mapping and scheduling are outweighed by the shorter runtime of the simpler methods.

Furthermore, the overhead introduced by the dynamic methods may even be the limiting factor for the speedup. If the reduction of the execution time in the parallelized search space is smaller than the time required for dynamic partitioning, mapping and scheduling, there will be no performance gain by parallelization.

#2 Migrate Software to Hardware to Leverage Performance

Analysis of AI algorithms reveals that they often consist of short and simple functions which are called very often and thus dominate the overall runtime. Even a small acceleration of these functions may speed up the algorithm significantly. As our prototypes demonstrate, a significant improvement can be achieved by migrating these low-level functions from software to dedicated hardware. An important issue in software to hardware migration is the interface between software and hardware components.

#3 Deploy Specialized Computer Architectures

A good match between the parallel/distributed structure of the search problem and the multiprocessor architecture is a precondition for high performance. General-purpose architectures are not flexible enough to achieve a perfect matching between problem structure, multiprocessor topology and inter-processor communication structure. By designing specialized computer architectures a weak match can be overcome. Additionally, dynamic logic structures can be supported efficiently (Schneider 2000). Our prototypes demonstrate how specialized computer architectures can be tuned for scalability. Thus, the performance can be adapted to the problem complexity by adding more processing elements and dedicated hardware.

#4 Exploit Parallelism at Multiple Levels

Obviously, the speedup can be improved if the parallelism is exploited at multiple levels. However, by using general-purpose multiprocessor or multicomputer architectures, parallelism can only be exploited at a single level, i.e., at the task or application level. By deploying a specialized computer architecture with dedicated hardware, i.e., combining lessons #2 and #3, high-level parallelism can be exploited among the processing elements and low-level parallelism can be exploited within the dedicated hardware.

#5 Be Aware of Differences in Sequential and Parallel Search

Partitioning of the search space into smaller problems may change the behavior of the search. This is exemplified by our prototypes: The dynamic partitioning method VBP (Platzner & Rinner 1998) in prototype 1, not only divides the search space of the CSP into independent sub-spaces but may also prune genuinely inconsistent parts. Thus, the overall space for the parallel search may be significantly smaller than for the sequential search which may result in a super-linear speedup. Due to the stochastic nature of the search in prototype 2, the parallel search may converge faster than the sequential search. This is because (i) spreading the search increases the probability of finding better solutions in the neighborhood of the current solution, and (ii) merging (synchronizing) helps to overcome the problem of being trapped in a local minimum.

Conclusion

In this paper we have presented two prototypes of parallel computer architectures for the general AI search methods constraint satisfaction and simulated annealing. Due to differences in the algorithmic characteristic, parallelization of AI algorithms is more complex than parallelization of numerical algorithms. Dynamic partitioning, mapping and scheduling especially complicate the parallelization process.

Our approach of designing specialized computer architectures results in runtime improvements of several orders of magnitude.

Acknowledgments

This work has taken place at the Institute for Technical Informatics, Technical University Graz and has been supported in part by the Austrian Science Fund under grant number P10411-MAT.

References

Aarts, E., and Lenstra, K. 1997. *Local Search in Combinatorial Optimization*. Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons.

Higuchi et al., T. 1994. The IMX2 Parallel Associative Processor for AI. *IEEE Computer* 27(11):53–63.

1992. IEEE Computer. Computer Architectures for Intelligent Systems.

Kuipers, B. 1994. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. Artificial Intelligence. MIT Press.

Mackworth, A. K. 1992. Constraint Satisfaction. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*, volume 1. John Wiley & Sons, Inc. 285–293.

Moldovan, D.; Lee, W.; Lin, C.; and Chung, M. 1992. SNAP Parallel Processing Applied to AI. *IEEE Computer* 25(5):39–49.

Platzner, M., and Rinner, B. 1998. Design and Implementation of a Parallel Constraint Satisfaction Algorithm. *International Journal of Computers and Their Applications* 5(2):106–116. International Society of Computers and Their Applications (ISCA).

Platzner, M.; Rinner, B.; and Weiss, R. 1997. Parallel Qualitative Simulation. *Simulation Practice and Theory — International Journal of the Federation of European Simulation Societies* 5(7-8):623–638. Elsevier Science Publishers B.V.

Platzner, M.; Rinner, B.; and Weiss, R. 2000. Toward Embedded Qualitative Simulation: A Specialized Computer Architecture for QSim. *IEEE Intelligent Systems* 15(2).

Schmid, M., and Schneider, R. 1999. Parallel Simulated Annealing Techniques for Scheduling and Mapping DSP-Applications onto Multi-DSP Platforms. In *Proceedings of the International Conference on Signal Processing Applications & Technology*. Orlando, U.S.A.: Miller Freeman, Inc.

Schneider, R., and Weiss, R. 2000. Hardware Support for Simulated Annealing and Tabu Search. In *Workshop on Biologically Inspired Solutions to Parallel Processing Problems at the International Parallel and Distribute Processing Symposium 2000*. Cancun, Mexico: Springer Verlag LNCS.

Schneider, R. 2000. *Ein Spezialprozessor für kombinatorische Optimierung*. Ph.D. Dissertation, Institut für Technische Informatik, Technische Universität Graz.