Design and Implementation of an Efficient Thread Partitioning Algorithm

José Nelson Amaral, Guang Gao, Erturk Dogan Kocalar,
Patrick O'Neill, Xinan Tang

Computer Architecture and Parallel Systems Laboratory,
University of Delaware, Newark, DE, USA, http://www.capsl.udel.edu
Dep. of Comp. Science, Univ. of Alberta, Canada, http://www.cs.ualberta.ca

Abstract. The development of fine-grain multi-threaded program execution models has created an interesting challenge: how to partition a program into threads that can exploit machine parallelism, achieve latency tolerance, and maintain reasonable locality of reference? A successful algorithm must produce a thread partition that best utilizes multiple execution units on a single processing node and handles long and unpredictable latencies.
In this paper, we introduce a new thread partitioning algorithm that can meet the above challenge for a range of machine architecture models. A quantitative affinity heuristic is introduced to guide the placement of operations into threads. This heuristic addresses the trade-off between exploiting parallelism and preserving locality. The algorithm is surprisingly simple due to the use of a time-ordered event list to account for the multiple execution unit activities. We have implemented the proposed algorithm and our experiments, performed on a wide range of examples, have demonstrated its efficiency and effectiveness.

1 Introduction

This paper is a contribution to the development of high-performance computer systems based on a fine-grain multi-threaded program execution and architecture model. A key to the success of multi-threading is the development of compilation methods that can efficiently exploit fine-grain parallelism in application programs and match them with the parallelism of the underlying hardware architecture. In particular, partitioning programs into fine-grain threads is a new challenge that is not dealt with in conventional compiler code generation and optimization.

Our thread partitioning algorithm was developed for the Efficient Architecture for Running THreads (EARTH), a multi-threaded execution and architecture model [8, 4]. Under the EARTH model, a thread becomes *enabled* for execution if and only if it has received signals from all the split-phase operations that it depends on. Furthermore, threads are non-preemptive: once a thread is scheduled for execution, it holds the execution unit until its completion. Therefore whenever an operation may involve long and/or unpredictable latencies,

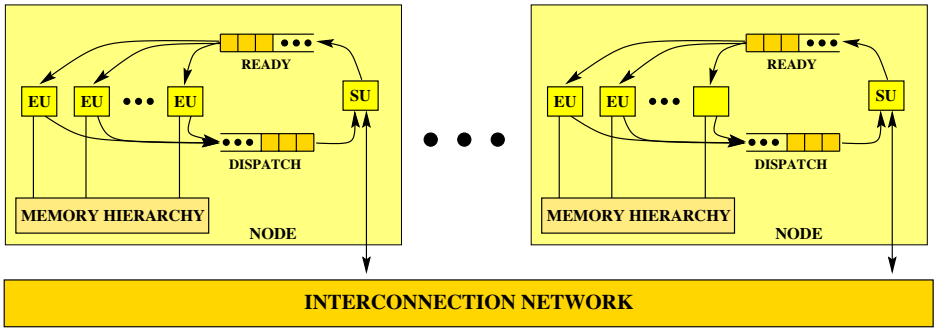

Fig. 1. Architecture abstract model.

the role of a compiler (or programmer) is to make the operation "split-phase". We call this requirement the *split-phase constraint*, and assume that such constraints are explicitly identified and presented to the thread partitioner. The thread partitioning problem studied in this paper can be stated as follows.

> *Given a machine model M and a weighed data dependence graph G with some nodes labeled as split-phase nodes, partition G into threads such that the total execution time of G is minimized subject to the split-phase constraints.*

The main contribution of this paper is the development of an heuristic thread partitioning algorithm suitable for a machine model that allows multiple execution units in each processing node.[1] Unlike previous related thread partitioning algorithms, ours faces a new challenge: the existence of more than one thread execution unit per node implies a trade-off between the need to generate enough parallel threads per node to utilize these execution units, and the need to assign related operations to the same thread to enhance locality of access.

2 Machine Model

Our architecture model is presented in Figure 1. Each processing node has N execution units (EU) and one synchronization unit (SU). Both the EU and the SU perform the functions specified in the EARTH Virtual Machine [8]. Threads that are ready for execution are placed in the ready queue that is serviced by the EUs. When an active thread performs a synchronization operation or requests a long latency data transfer, the request for such a service is placed in the dispatch queue. The SU is responsible for the communication with all other processing nodes and for the synchronization between threads within the node.

[1] Notice that the algorithm presented in [7] collapses as much local computation as possible into a single thread, thus making it inadequate for the machine model studied in this paper that has multiple execution units per processing node.

In the model of Figure 1 each processing node has its own memory hierarchy, but an EU can access memory locations in any node of the machine.[2] An access to a location in the local memory hierarchy, a local access, has a lower latency and higher bandwidth than a remote access. As in [7], we assume that a cost model is provided that allows for an estimation of the cost of all local and all remote operations required in a program statement. We use δ to represent the cost associated with the termination of a thread and the start of the execution of another thread. Although we assume that a ready thread can be executed by any one of the local processors, to favor data locality and benefit from the local caches in the architecture, the partitioning algorithm takes into consideration the amount of dependencies among statements when placing them in different threads

3 Thread Partition Cost Model

We assume that a program is written in a sequential language augmented with high level-parallel constructs, and that the data has been partitioned among the memory modules in the processing nodes of the machine. Therefore, given a program statement we can determine whether the statement is *local* or *remote*. We also assume that the program has been translated into a Data Dependence Graph (DDG).

Thus the program is represented by a graph $G(V, E)$ where each node in V represents a collection of program statements. A node can be a *simple node* such as an assignment statement or a *compound node* such as a loop. If the execution of the program statements requires accesses to a remote memory module, the node is a *remote node* otherwise the node is a *local node*. Each edge $(v_i,\ v_j)$ in E represents a data dependency from v_i to v_j. An edge departing from a remote node is a *remote edge*, and an edge departing from a local node is a *local edge*. Like in [7] we represent E by an adjacency matrix C defined as:

$$C_{i,j} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ 0 \text{ otherwise} \end{cases}$$

The partitioning algorithm is based on a cost model that associates a *local cost* c_i^L and a *remote cost* c_i^R to each node v_i that represent the number of cycles that the nodes spends in the EU and the number of cycles that elapses between the request and the completion of a split-phase transaction started by the node. One of the main goals of the partitioning algorithm is to decide when it is advantageous to dispatch the request, terminate the current thread and to start the execution of a new thread when the remote operation is completed. For some interconnection networks a model to predict the network performance will be necessary because the remote cost is affected by the load in the network. For the experiments with our partitioning algorithm, we assume that this cost

[2] If the underline machine does not allow direct accesses to remote memory, the EARTH systems emulates a global address space.

is bounded by a constant. The cost model presented in this paper can be constructed for any machine through profiling experiments and examination of the machine specification for the number of cycles required to execute each class of instruction. The network latency and bandwidth can also be readily measured. For our experiments we use the values obtained by Theobald for the EARTH-MANNA implementation [8].

4 Problem Formulation

Assume that at runtime threads are selected for execution from a single queue of ready threads using an efficient scheduler. Given a DDG G with each node $v_i \in G$ annotated with its local cost c_i^L and its remote cost c_i^R, and a constant thread switching cost δ, the **thread partitioning problem** is the problem of finding a thread partition P that meets two goals: (1) minimizes the total execution time, (2) maximizes the affinity between nodes assigned to the same thread. The **affinity** of a node v_i of G to a thread T_k of P, $A(v_i, T_k)$, is given by the ratio between the number of dependences between nodes of T_k and v_i, and the total number of incoming edges of v_i.

$$A(v_i, T_k) = \frac{\sum_{v_j \in T_k} C_{ji}}{\sum_j C_{ji}}$$

Observe that if all the incoming edges of v_i are from nodes in T_k, then $A(v_i, T_k) = 1$. On the other hand, if none of the incoming edges of v_i are from nodes in T_k, then $A(v_i, T_k) = 0$.

Goals (1) and (2) should be pursued under the constraint that nodes connected by a remote edge are assigned to different threads. Goal (1) is the principal goal of the partition algorithm, while goal (2) is necessary to favor locality of access because the abstract model assumes that all processors have equal probability of fetching a thread from the ready queue.

The thread partitioning algorithm uses the affinity function to decide in which thread to place a node of the DDG. The algorithm keeps a partial schedule of the threads already formed and searches into this schedule for the best place to insert a node from the DDG. To minimize the searching time, an event list is used to store the starting and finishing time of each thread. A detailed description of the thread partitioning algorithm including an example is presented in [1].

5 Experimental Results

We use the Thread Partition Test Bed presented in [7] to generate random DDGs to test the partition algorithm. We vary several properties of the DDGs generated, including the number of nodes, the average number of outgoing edges from a node, the percentage of remote nodes in the graph and the distribution of local and remote costs in the nodes.

The distribution of execution costs for the nodes is as follows: A local node can be of three types: 1) Local I/O (20%), 2) Local function call (10%), 3) Other(math,etc.) (70%). A local I/O is assigned a cost of 10 cycles. A local function call is assigned 10 cycles and a node of other type has 3 cycles. Remote nodes can be of type: 1) Remote I/O (80%), 2) Remote function call (20%). A remote I/O is assumed to take 300 cycles, and the costs of remote function calls are uniformly distributed between 400 cycles and 4000 cycles. This distribution of node types and the degree of the DDG generated are based on static profiling of EARTH-C benchmarks [7, 3]

5.1 Summary of Main Experimental Results

The main results of our experiments can be summarized as follow.

Absolute Efficiency Our new partition algorithm is very efficient for a wide range of DDGs on all the machine models except the EARTH-Dual model where only a single EU is available per node and there is a high cost associated with thread switching. The algorithm performs remarkably well when the machine model has multiple execution units – e.g. for SMP and SCMP models – the average absolute efficiency is above 99%.

Effectiveness of Search Heuristics The use of an event list and of a time line schedule results in an effective search for the placement of a new node (see [1] for details).

Latency Tolerance Capacity The algorithm is robust to variations in latency. In our experiments a remote operation latency varied between 400 and 4000 cycles. As shown in Table 1 the partition algorithm produces thread partitions that are able to tolerate these varied latencies.

5.2 Machine Architectures

We define four different machine architectures for our experiments. MANNA is a multiprocessor machine with 40 processors distributed in 20 processing nodes interconnected by a crossbar switch. MANNA is the first platform in which the EARTH model was implemented [5].

EARTH-MANNA-DUAL: An implementation of the EARTH architecture on the MANNA machine. The second processor is used for the SU function. Therefore, according to our model, this is a machine with a single EU per node. The thread switching cost in this machine is $\delta = 36$ cycles (see measurements reported in [8]).

EARTH-MANNA-SPN: The two processors in the machine are used to implement the SU functions. Therefore, this machine has two EUs per processing node. The thread switching cost is $\delta = 16$ (see measurements reported in [8]).

EARTH-SU: This is the EARTH architecture with a custom hardware SU. We consider a machine with a single execution unit per processing node and with a thread switching cost $\delta = 2$ (see measurements reported in [8]).

SMP: This is an Symmetric Multi-Processor machine with 4 processors per node. In such a machine we expect the thread switching cost to be similar to the one for the MANNA-SPN, therefore we will use $\delta = 16$.

Single-Chip Multi-threaded Processor (SCMP) In this case we consider a hypothetical machine that has multiple functional units with multi-threading support. We assume a machine with 8 EUs and with a thread switching cost of $\delta = 10$.

5.3 Measuring Absolute Efficiency

An optimal partition cannot result in an execution time that is shorter than the critical path of the program or shorter than the total amount of work to be performed divided by the number of EUs available. Thus, a lower bound for the execution time is given by:

$$T_{lowest} = \max(T_{crit}, \frac{\sum_{i \in V} c_i^L}{N})$$

where T_{crit}, is the length of the critical path, the sum of the local cost c_i^L of all nodes is the total amount of work to be performed by the program, and N is the number of EUs in the machine. We define the *absolute efficiency* as the ratio:

$$E = T_{lowest}/T_{end}$$

where T_{end} is the execution time for the program with the thread partition produced by our algorithm running under an efficient FIFO scheduler. $E = 100\%$ means that the partition algorithm found a partition that can result in the optimal execution time.

To measure the efficiency of the algorithm, we varied the percentage of remote nodes in the randomly generated DDG from 25% to 75%, and the size of the graph from 10 nodes to 1000 nodes. Each graph has three times as many edges as the number of nodes. Then we generated twenty distinct random DDGs and applied the partition algorithm to each one of them. We computed the average execution time for each run, compared it with T_lowest, and present the average efficiency in Table 1. The algorithm did remarkably well for machines with four or eight execution units per processing node (SMP and SCMP). The algorithm also did quite well both for the EARTH-SU that has a single EU and a very low thread switching cost and for the EARTH-MANNA-SPN that has two EUs per processing node. The results for graphs with a large number of nodes for the EARTH-MANNA-DUAL are not as good. This should not come as a surprise because this architecture has a single EU and very large thread switching costs.

6 Related Work

The thread partition problem for multi-threaded architectures is similar to the task partitioning and scheduling problem [6, 9]. In both problems a program has

Machine	% Remote Edges	Nodes						
		10	20	50	100	200	500	1000
EARTH-MANNA-DUAL	25%	87.6	91.8	90.0	83.6	82.3	77.0	78.1
	50%	96.3	95.8	96.8	93.7	83.8	71.6	71.2
	75%	96.5	98.7	98.9	98.2	95.6	69.5	66.0
EARTH-MANNA-SPN	25%	95.2	98.2	95.9	94.2	91.5	87.1	88.6
	50%	95.7	97.6	98.7	97.7	97.8	88.5	83.7
	75%	99.3	99.8	99.9	99.8	99.8	99.3	80.3
EARTH-SU	25%	95.3	98.3	92.4	93.2	96.7	99.9	100.0
	50%	95.9	95.4	99.2	97.0	92.0	92.8	98.1
	75%	98.4	99.3	98.4	97.8	98.5	90.3	94.1
SMP	25%	99.2	99.7	99.8	99.9	99.3	99.9	99.9
	50%	99.7	99.8	99.9	99.9	99.9	99.9	100.0
	75%	99.8	99.9	99.9	99.9	99.9	100.0	100.0
SCMP	25%	99.2	99.8	99.8	99.9	99.3	99.9	99.9
	50%	99.7	99.8	99.9	99.9	99.9	99.9	100.0
	75%	99.7	99.9	99.9	99.9	99.9	100.0	100.0

Table 1. Average partition algorithm efficiency.

to be divided into smaller pieces with respect to some constraints (dependencies). The focus of task partition is to allocate N tasks onto M processors in order to reduce the total execution time. This problem is often represented as a graph partition problem in which nodes denote tasks and edges represent two types of constraints: *precedent constraint*, i.e., one task must complete before another task can start; and *communication constraint*, i.e., data must be exchanged between two tasks. When the communication constraint is taken into consideration the task partitioning problem is an NP-complete problem [2]. In this case, the optimization goal is often reduced to minimize the total communication [6]. For further discussion of related work we refer to [7] and [1].

7 Conclusion

We designed, implemented, and evaluated an efficient, effective, and robust algorithm to partition a program into threads for the case in which multiple execution units are available in each processing node of a parallel architecture. The algorithm is efficient because it generates a partition that results in an execution time that is very close to the best possible execution time determined by the length of the critical path and the total amount of computation existing in the program. The algorithm is robust because it worked efficiently for a varied set of architectures and a wide range of latencies between processing node. The algorithm is effective because it employs a data structure associated with a searching algorithm that reduce the time complexity of the algorithms. On our experimental framework we tested the algorithm with several thousand data dependency graphs with up to a thousand nodes and several thousand connections.

Acknowledgements

We would like to thank Andres Marquez, Arthour Stoutchinin, Gagan Agarwal, Kevin Theobald, and Mark Butala for productive discussions and valuable help. We acknowledge the support of DARPA, NSA and NASA through a subcontract with JPL/Caltech. The current EARTH research is partly funded by the NSF.

References

1. J. N. Amaral, G. R. Gao, E. D. Kocalar, P. O'Neill, and X. Tang. Design and implementation of an efficient thread partitioning algorithm. Technical report, University of Delaware, Newark, DE, July 1999. CAPSL Technical Memo 30.
2. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freemann and Co., New York, New York, 1979.
3. Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
4. Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.
5. Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.
6. Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. Revised version of the author's Ph.D. dissertation (Stanford University, April 1987).
7. Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.
8. Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.
9. T. Yang and A. Gerasoulis. List scheduling with and without communication delay. *Parallel Computing*, 19:1321–1344, 1993.