

# Automatic Parallelization of Divide-and-Conquer Algorithms

Bernd Freisleben and Thilo Kielmann

Department of Computer Science (FB 20), University of Darmstadt, Alexanderstr. 10,  
D-6100 Darmstadt, Germany

**Abstract.** In this paper we present a system that automatically partitions sequential divide-and-conquer algorithms programmed in *C* into independent tasks, maps these to a MEIKO transputer system and executes them in parallel. The feasibility of our approach is illustrated by parallelizing several example algorithms and measuring the resulting performance speedups.

## 1 General Approach and System Description

Our approach for mapping divide-and-conquer algorithms to a multiprocessor architecture is to use the partitioning information inherently included in the algorithm and let one processor divide the initial problem into two subproblems, pass one of these to a further processor and keep the other one to itself [2]. Every processor repeats this step recursively until the problem size is sufficiently small and performs the computation assigned to it, which logically constitutes a mapping to a binomial tree topology. The results are propagated up the tree and combined in the reverse order in which the subproblems were passed down.

The fundamental communication pattern used in our system is based on a master/slave organization in which a unique master task distributes the work to a set of slave tasks via an asynchronous remote procedure call mechanism. In addition to the master task, which is responsible for the particular problem to be solved, there is a unique *scheduler* task which controls the pool of available processors and is thus the only entity that knows the physical topology of the network. The master and the scheduler are assigned to the same dedicated processor, while each slave runs on a unique processor of the pool.

A few language extensions are required for generating code which is able to operate in parallel. Besides a keyword indicating parallel compilation, additional declarations are required for specifying input and output parameters of the (possibly remotely) called function and a synchronization point after which the results of the subproblems are available.

The source code is restructured into three parts: the master part with the original `main()` function, the function part which replaces the recursive function calls by directing them to a stub, and the slave part which contains code for receiving the parameters, calling the function and returning the results. The function part is concatenated with both the master and slave part. These parts are separately compiled with the *C* compiler of the target system, resulting in two object files.

In a last step, the object files produced so far are linked together with appropriate main programs and a communications library which essentially contains the

implementation of an asynchronous RPC mechanism and provides the interface to the runtime system of the target architecture.

## 2 Implementation and Performance

Our system was implemented in *C* on a SUN Sparcstation, using the MEIKO cross compiler. The computation time required for transforming a sequential divide-and-conquer algorithm to concurrently executable code is about 1–2 seconds.

The algorithms selected as test cases are the well known quicksort, an algorithm for matrix multiplication, an algorithm for adaptive numerical integration [1] and an algorithm for the knapsack problem. The four programs have been automatically parallelized with our system and executed on the MEIKO transputer system consisting of 72 transputers T800. The physical topologies used were hypercubes for up to 8 processors and cube-connected cycles (with 2 processors in each corner) for 16 and 32 processors. The programs have also been compiled to run sequentially on one processor. The runtimes measured for the sequential programs ( $T_s$ ) were used to compute the speedup  $S = T_s/T_n$ , where  $T_n$  is the runtime of the parallel execution with  $n$  processors. The speedup factors achieved are shown in Figure 1.

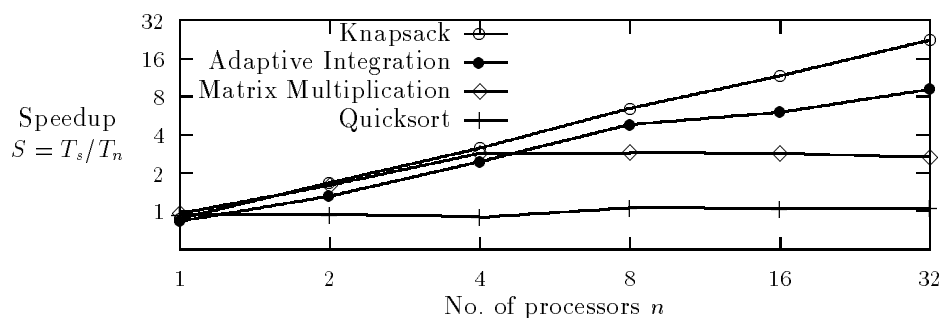


Fig. 1. Speedups relative to sequential programs

The speedups achieved for the individual problems differ significantly and thus indicate the suitability of the different algorithms for parallel execution in a message-passing multicomputer environment. For the quicksort algorithm with a relatively low sequential time complexity of  $O(n \log n)$  the communication overhead clearly dominates the total cost. With increasing dominance of computation over communication, the results become better. A peak value of 23 for 32 processors is achieved by the parallelized knapsack algorithm.

## References

1. G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, 1991.
2. V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed and J. Telle. Mapping Divide-and-Conquer Algorithms to Parallel Architectures. In *International Conference on Parallel Processing*, Vol. III, pages 128–135, CRC Press, 1990.