Performance Evaluation of Parallel Divide–and–Conquer Algorithms

Technical Report THD-BS-1993-01

Bernd Freisleben and Thilo Kielmann

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

Tel.: (49)–6151–165306
Fax: (49)–6151–165410
Email: freisleb@isa.informatik.th-darmstadt.de
kielmann@isa.informatik.th-darmstadt.de

1 Introduction

Divide–and–conquer algorithms obtain the solution to a problem by recursively dividing it into subproblems of the same kind until the problem size has been sufficiently reduced, then solving these subproblems independently and finally combining their solutions [5]. Due to the importance of the divide–and–conquer paradigm in a wide range of problem domains, several parallel divide–and–conquer algorithms for message–passing multicomputer systems have been proposed [3, 4, 6] in order to achieve faster computation times. The speedups reported in these proposals indicate that the particular algorithms selected benefit from a parallel implementation, but it is difficult to compare the results with each other, because the approaches taken to implement the algorithms on a parallel architecture differ significantly.

In this paper we evaluate the performance of several parallel divide–and–conquer algorithms which are created in a uniform manner by a software system developed for automatically transforming a sequential divide–and–conquer algorithm into parallel code, instead of individually designing a parallel algorithm by hand. The system partitions a sequential algorithm programmed in *C* into independent tasks, maps these to a MEIKO transputer system and executes them in parallel. From the speedups obtained it is evident that not all algorithms considered are worth to be parallelized, and we will present additional measurements to explain why.

2 General Approach

Our approach for mapping divide–and–conquer algorithms to a message–passing multicomputer is to use the partitioning information included in the algorithm itself 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 [4]. 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 tree.

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.

Initially, both the master and all slaves identify themselves to the scheduler. The problem to be solved is then given to the master who in turn divides it and requests the unique identification of an idle slave from the scheduler. Upon receipt, the master sends the subproblem to this slave. The slave repeats the process and thus acts as the master of the subproblem which it intends to propagate. Once a slave has finished its computation and has returned the results to its master, the master informs the scheduler to release the slave and make it again available to the pool of idle processors. The whole computation is finished when the master terminates and all slaves have been returned to the pool.

3 System Description

Our system for automatic parallelization of sequential divide–and–conquer algorithms processes the *C* language conforming to the ANSI standard, but a few language extensions are required for generating code which is able to operate in parallel. These extensions are only of declarative nature, primarily included to overcome *C*'s deficiencies in the description of formal parameters. Thus, the language extensions are not necessary as far as partitioning and mapping for parallel execution is concerned.

The original *C* source code is precompiled with the C–preprocessor and then 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, these two object files are linked together with the 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.

4 Implementation and Performance

Our system was implemented in *C* on a SUN Sparcstation, employing standard UNIX tools such as LEX/YACC, the MEIKO cross compiler and the communication features offered by the *MEIKO CS Tools* programming environment. The computation time required for transforming a sequential divide–and–conquer algorithm to concurrently executable binaries is about 1–2 seconds.

The algorithms selected as test cases are the well known quicksort algorithm, an algorithm for matrix multiplication, an algorithm for adaptive numerical integration [1], an algorithm for the knapsack problem, an algorithm for the decomposition of positive integer numbers into prime factors, and an exact algorithm for the set–covering problem [2]. The sizes of the test problems were chosen as large as possible with respect to the limited main memory capacities of the processors used.

The six programs have been automatically parallelized with our system and executed on the MEIKO transputer system. The physical topologies used were hypercubes for up to $n = 8$ processors and cube–connected cycles (with 2 processors in each corner) for $n = 16$

and $n = 32$. The latter topology has been selected to minimize the path length under the constraint that each transputer is equipped with four physical links only.

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.

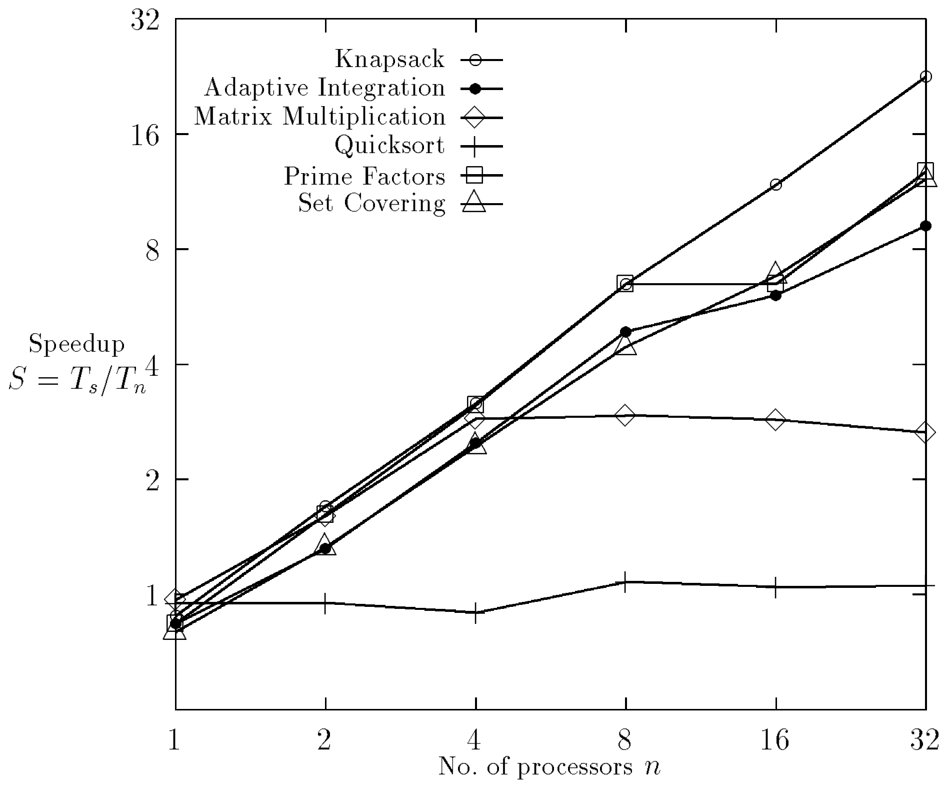

Figure 1: Speedups relative to sequential programs

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. In order to explain the different speedup factors, we need a closer look at the communication behaviour of the parallelized algorithms.

Our system transforms sequential divide-and-conquer algorithms into concurrently executable counterparts. To do so, new code is inserted into the programs for communication between the processors, for parameter and result transfers and for synchronization purposes. Thus, the resulting programs consist of user-written code for solving a given problem P and of generated code for communication purposes C.

The parallelization tool optionally includes profiling code for monitoring the runtimes of both parts P and C. Using this profiling, we are able to obtain the runtimes T_P and T_C spent in P and C for each processor.

These measured runtimes were used to compute the ratio $R = \frac{T_P}{T_P+T_C}$ for each algorithm and for each number of processors investigated. Looking at R as a function of the number of processors n allows deeper insights into the algorithms' behaviours. The speedup factors obtained suggest a classification of the algorithms into the three categories *almost linear speedup*, *slowly increasing speedup* and *poor or no speedup*. For the first category (knapsack problem), a factor of $R \geq 0.85$ is required for all numbers of processors. The category constituting slowly increasing speedup factors (set covering, adaptive integration) has values of $R \geq 0.7$. This means that a problem is only worth parallelizing if less than about 30% of the total runtime is spent for communication, and satisfactory speedups can only be obtained if the communication time consumes less than 15% of the total time required. A factor $R < 0.7$ suddenly outweighs all additional parallel computing power. This explains the poor performance speedups of the quicksort algorithm, which crosses the 0.7 border for already 2 processors. The matrix-multiplication algorithm reaches the border for 8 processors. As a result, the speedup factors obtained stagnate for more than 4 processors.

The numbers of processors for which the critical values of R are reached clearly depend on our implementation. However, since we have tried to eliminate the effects of possibly interfering parameters by providing equal prerequisites for all algorithms investigated, the results are likely to remain valid on a qualitative basis, independent of the particular implementation considered.

References

[1] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[2] Thomas H. Cormen, Charles E. Leisureson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] Renate Knecht. Implementation of divide-and-conquer algorithms on multiprocessors. In *Parallelism, Learning, Evolution – Workshop on Evolutionary Models and Strategies*, pages 121–136, Neubiberg, Germany. Springer–Verlag, 1989.

[4] 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.

[5] Robert Sedgewick. *Algorithms*. Addison Wesley, 2nd edition, 1988.

[6] I-Chen Wu. Efficient parallel divide-and-conquer for a class of interconnection topologies. In *2nd International Symposium on Algorithms (ISA '91)*, pages 229–240, Taipei, Republic of China. Springer–Verlag, 1991.