OPTIMIZING SKELETAL STREAM PROCESSING FOR DIVIDE AND CONQUER

Michael Poldner
University of Münster, Department of Information Systems
Leonardo Campus 3, D-48149 Münster, Germany
Email: poldner@wi.uni-muenster.de

Herbert Kuchen
University of Münster, Department of Information Systems
Leonardo Campus 3, D-48149 Münster, Germany
Email: kuchen@uni-muenster.de

Keywords: Parallel Computing, Algorithmic Skeletons, Divide & Conquer.

Abstract: Algorithmic skeletons intend to simplify parallel programming by providing recurring forms of program structure as predefined components. We present a new distributed task parallel skeleton for a very general class of divide and conquer algorithms for MIMD machines with distributed memory. Our approach combines skeletal internal task parallelism with stream parallelism. This approach is compared to alternative topologies for a task parallel divide and conquer skeleton with respect to their aptitude of solving streams of divide and conquer problems. Based on experimental results for matrix chain multiplication problems, we show that our new approach enables a better processor load and memory utilization of the engaged solvers, and reduces communication costs.

1 INTRODUCTION

Parallel programming of MIMD machines with distributed memory is typically based on standard message passing libraries such as MPI (MPI, 2008), which leads to platform independent and efficient software. However, the programming level is still rather low and thus error-prone and time consuming. Programmers have to fight against low-level communication problems such as deadlocks, starvation, and termination detection. Moreover, the program is split into a set of processes which are assigned to the different processors, whereas each process only has a local view of the overall activity. A global view of the overall computation only exists in the programmer's mind, and there is no way to express it more directly on this level.

For this reasons many approaches have been suggested, which provide a higher level of abstraction and an easier program development to overcome the mentioned disadvantages. The skeletal approach to parallel programming proposes that typical communication and computation patterns for parallel programming should be offered to the user as predefined and application independent components, which can be combined and nested by the user to form the skeletal structure of the entire parallel application. These components are referred to as algorithmic skeletons (E. Alba, 2002; Cole, 1989; J. Darlington, 1995; H. Kuchen, 2002; Kuchen, 2002; K. Matsuzaki, 2006; Pelagatti, 2003). Typically, algorithmic skeletons are offered to the user as higher-order functions, which get the details of the specific application problem as argument functions. In this way the user can adapt the skeletons to the considered parallel application without bothering about low-level implementation details such as synchronization, interprocessor communication, load balancing, and data distribution. Efficient implementations of many skeletons exist, such that the resulting parallel application can be almost as efficient as one based on low-level message passing.

Depending on the kind of parallelism used, skeletons can roughly be classified into data parallel and task parallel ones. A data parallel skeleton (G. H. Botorog, 1996; Kuchen, 2002; Kuchen, 2004) works on a distributed data structure such as a distributed array or matrix as a whole, performing the same operations on some or all elements of this data structure. Task parallel skeletons (A. Benoit, 2005; Cole, 2004; H. Kuchen, 2002; Kuchen, 2002; Poldner and Kuchen, 2005; Poldner and Kuchen, 2006) create a system of processes communicating via streams of data by nesting and combining predefined process topologies such as pipeline, farm, parallel composition, divide and conquer, and branch and bound. In the present paper we will consider task parallel divide and conquer skeletons with respect to their aptitude of solving streams of divide and conquer problems.

Divide and conquer is a well known computation paradigm, in which the solution to a problem is obtained by dividing the original problem into smaller subproblems, solving the subproblems recursively, and combining the partial solutions to the final solution. A simple problem is solved directly without dividing it further. Examples of divide and conquer computations include various sorting methods such as mergesort and quicksort, computational geometry algorithms such as the construction of the convex hull or the delaunay triangulation, combinatorial search such as constraint satisfaction techniques, graph algorithmic problems such as graph coloring, numerical methods such as the Karatsuba multiplication algorithm, and linear algebra such as Strassen's algorithm for matrix multiplication. In many cases there is the need of solving multiple divide and conquer problems in sequence. Examples here are the triangulation of several geometric figures, matrix chain multiplication, and factoring of large numbers.

In the present paper we will consider different task parallel divide and conquer skeletons in the context of the skeleton library Muesli (Kuchen, 2002; Poldner and Kuchen, 2005; Poldner and Kuchen, 2006; Poldner and Kuchen, 2008b; Poldner and Kuchen, 2008a), which are used to solve streams of divide and conquer problems. Muesli is based on MPI internally in order to inherit its platform independence. We have implemented a new fully distributed divide and conquer skeleton and compare it to a farm of sequentially working divide and conquer skeletons, and to a fully distributed divide and conquer skeleton used in a previous version of Muesli. We will show that our new approach enables a better processor load and memory utilization of the engaged solvers, and reduces communication costs.

The rest of this paper is structured as follows. In section 2, we briefly introduce divide and conquer skeletons and general terms. In Section 3, we present different parallel implementation schemes of the considered skeletons in the framework of the skeleton library Muesli, and discuss their application in the context of streams. Section 4 contains experimental results demonstrating the strength of our new distributed design. In Section 5 we compare our approach to related work, and finally, we conclude and point out future work in section 6.

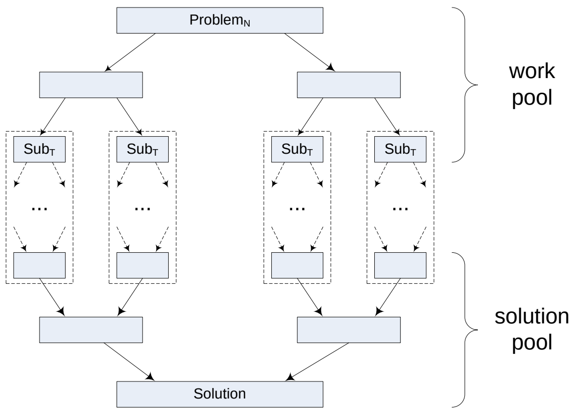

Figure 1: A divide and conquer tree

2 DIVIDE AND CONQUER SKELETONS

A divide and conquer algorithm solve an initial problem by dividing it into smaller subproblems, solving the subproblems recursively, and combining the partial solutions to the final solution, whereas simple problems are solved directly without dividing them further. The computation can be viewed as a process of expanding and shrinking a tree, in which the nodes represent problem instances and partial solutions, respectively (Fig.1).

A divide and conquer skeleton offers this basic strategy to the user as predefined parallel component. Typically, the user has to provide a divide and conquer skeleton with four application specific basic operators, with which the user can exactly adapt the skeleton to the considered problem: a `divide` operator which describes how the considered (sub)problem can be divided into subproblems, a `combine` operator which specifies how partial solutions can be combined to the solution of the considered parent problem, an `isSimple` operator which indicates if a subproblem is simple enough so that it can be solved directly, and last but not least a `solve` operator, which explains how to solve simple problems. During the solution of a concrete divide and conquer problem, the skeleton generates a multitude of subproblems and partial solutions (i.e. the nodes in the divide and conquer tree), which are stored in a work pool, and a solution pool respectively. In the beginning the work pool only contains the initial problem, and the solution pool is empty. In each iteration one such problem is selected from the work pool corresponding to a particular traversal strategy such as depth first or breadth first. The problem is either divided into d subproblems, which are stored again in the work pool, or it is