

Task Parallel Skeletons for Irregularly Structured Problems

Petra Hofstedt*

Department of Computer Science, Dresden University of Technology,
hofstedt@inf.tu-dresden.de

Abstract. The integration of a task parallel skeleton into a functional programming language is presented. Task parallel skeletons, as other algorithmic skeletons, represent general parallelization patterns. They are introduced into otherwise sequential languages to enable the development of parallel applications. Into functional programming languages, they naturally are integrated as higher-order functional forms. We show by means of the example branch-and-bound that the introduction of task parallel skeletons into a functional programming language is advantageous with regard to the comfort of programming, achieving good computation performance at the same time.

1 Introduction

Most parallel programs were and are written in imperative languages. In many of these languages, the programmer has to use low-level constructs to express parallelism, synchronization and communication. To support platform-independent development of parallel programs standards and systems have been invented, e.g. MPI and PVM. In functional languages, such supporting libraries have been added in rudimentary form only recently. Hence, the advantages of functional programs, such as their ability to state powerful algorithms in a short, abstract and precise way, cannot be combined with the ability to control the parallel execution of processes on parallel architectures.

Our aim is to remedy that situation. A functional language has been extended by constructs for data and task parallel programming. We want to provide comfortable tools to exploit parallelism for the user, so that she is burdened as few as possible with communication, synchronization, load balancing, data and task distribution, reaching at the same time good performance by exploitation of parallelism. The extension of functional languages by algorithmic skeletons is a promising approach to introduce data parallelism as well as task parallelism into these languages.

As demonstrated for imperative languages, e.g. by Cole [3], there are several approaches how to introduce skeletons into functional languages as higher-order

* The work of this author was supported by the ‘Graduiertenkolleg Werkzeuge zum effektiven Einsatz paralleler und verteilter Rechnersysteme’ of the German Research Foundation (DFG) at the Dresden University of Technology.

parallel forms. However, most authors concentrated on data parallel skeletons, e.g. [1], [4], [5]. Hence, our aim has been to explore the promising concept of task parallel skeletons for functional languages by integrating them into a functional language. A major focus of our work is on reusability of methods implemented as skeletons.

Algorithmic skeletons are integrated into otherwise sequential languages to express parallelization patterns. In our approach, currently skeletons are implemented in a lower-level imperative programming language, but presented as higher-order functions in the functional language. Implementation-details are hidden within the skeletons. In this way, it is possible to combine expressiveness and flexibility of the sequential functional language with the efficiency of parallel special purpose algorithms. Depending on the type of parallelism exploited, skeletons are distinguished in data and task parallel ones. Data parallel skeletons apply functions on multiple data at the same time. Task parallel skeletons express which elements of a computation may be executed in parallel. The implementation in the underlying system determines the number and the location of parallel processes that are generated to execute the task parallel skeleton.

2 The Branch-and-Bound Skeleton in a Functional Language

Branch-and-bound methods are systematic search techniques for solving discrete optimization problems. Starting with a set of variables with a finite set of discrete values (a domain) assigned to each of the variables, the aim is to assign a value of the corresponding domain to each variable in such a way that a given objective function reaches a minimum or a maximum value and several constraints are satisfied. First, mutually disjunct subproblems are generated from a given initial problem by using an appropriate branching rule (**branch**). For each of the generated subproblems an estimation (**bound**) is computed. By means of this estimation, the subproblem to be branched next is chosen (**select**) and decomposed (branched). If the chosen problem cannot be branched into further subproblems, its solution (if existing) is an optimal solution. Subproblems with non-optimal or inadmissible variable assignments can be eliminated during the computation (**elimination**). The four rules **branch**, **bound**, **select** and **elimination** are called basic rules.

The principal difference between parallel and sequential branch-and-bound algorithms lies in the way of handling the generated knowledge. Subproblems generated from problems by decomposition and knowledge about local and global optima belong to this knowledge. While with sequential branch-and-bound one processor generates and uses the complete knowledge, the distribution of work causes a distribution of knowledge, and the interaction of the processors working together to solve the problem becomes necessary.

Starting point for our implementation was the functional language DFS ('Datenparallele funktionale Sprache' [6]), which already contained data parallel skeletons for distributed arrays. DFS is an experimental programming lan-

guage to be used on parallel computers. The language is strict and evaluates DFS-programs in a call-by-value strategy accordingly.

To give the user the possibility to exploit parallelism in a very comfortable way, we have extended the functional language DFS by task parallel skeletons. One of them was a branch-and-bound skeleton. The user provides the basic rules **branch**, **bound**, **select** and **elimination** using the functional language. Then she can make a function call to the skeleton as follows:

`branch&bound branch bound select elimination problem.`

A parallel abstract machine (PAM) represents the runtime environment for DFS. The PAM consists of a number of interconnected nodes communicating by messages. Each node consists of three units: the message administration unit, which handles the incoming and outgoing messages, the skeleton unit, which is responsible for skeleton processing, and the reduction unit, which performs the actual computation. Skeletons are the only source of parallelism in the programs.

To implement the parallel branch-and-bound skeleton, several design decisions had to be made with the objective of good computation performance and high comfort. In the following, the implementation is characterized according to Trienekens' classification ([7]) of parallel branch-and-bound algorithms.

Table 1. Classification by Trienekens

knowledge sharing	global/local knowledge base complete/partial knowledge base update strategy
knowledge use	access strategy reaction strategy
dividing the work	units of work load balancing strategy
synchronicity	synchronicity of each process
basic rules	branch, bound, select, elimination

Each process uses a *local partial knowledge base* containing only a part of the complete generated knowledge. In this way, the bottleneck arising from the access of all processes to a shared knowledge base is avoided, but at the expense of the actuality of the knowledge base. A process stores newly generated knowledge at its local knowledge base only; if a local optimum has been computed, the value is broadcasted to all other processes (*update strategy*).

When a process has finished a subtask, it accesses its local knowledge base, to store the results at the knowledge base and to get a new subtask to solve. If a process receives a message containing a local optimum, the process compares this optimum with its actual local optimum and the bounds of the subtasks still to be solved (*access strategy*). A process receiving a local optimum from another process first finishes its actual task and then reacts according to the received

message (*reaction strategy*). This may result in the execution of unnecessary work. But the extent of this work is small because of the high granularity of the distributed work.

A *unit of work* consists of branching a problem into subproblems and computing the bounds of the newly generated subproblems. The *load balancing strategy* is simple and suited to the structure of the computation, because new subproblems are generated during computation. If a processor has no more work, it asks its neighbours one after the other for work. If a processor receives a request for work, it returns a unit of work – if one is available – to the asking processor. The processor sends that unit of work which is nearest to the root of the problem tree and has not been solved yet.

The implemented distributed algorithm works *asynchronously*.

The basic rules are provided by the user using the functional language.

3 Performance Evaluation

To evaluate the performance of task parallel skeletons, we implemented branch-and-bound for the language DFS as a task parallel skeleton in C for a GigaCluster GCell1024 with 1024 transputers T805(30 MHz) (each with 4 MByte local memory) running the operating system Parix.

Performance measurements for several machine scheduling problems – typical applications of the branch-and-bound method – were made to demonstrate the advantageous application of skeletons. In the following, three cases of a machine scheduling problem for 2 machines and 5 products have been considered. The number of possible orders of machine allocation is $5! = 120$. This very small problem size is sufficient to demonstrate the consequences for the distribution of work and the computation performance, if the part of the problem tree, which must be computed, has a different extent. In case (a) the complete problem tree had to be generated. In case (b) only one branch of the tree had to be computed. Case (c) is a case where a larger part of the problem tree than in case (b) had to be computed. Each of the machine scheduling problems has been defined first in the standard functional way and second by use of the branch-and-bound skeleton. The measurements were made using different numbers of processors.

First we counted branching steps, i.e. we measured the average overall number of decompositions of subproblems of all processors working together in the computation of the problem. These measurements showed the extend of the computed problem tree working sequentially and parallelly. It became obvious that the overall number of branching steps is increasing in the case of a small number of to be branched subproblems. The local partial knowledge bases and the asynchronous behaviour of the algorithm cause the execution of unnecessary work. If the whole problem tree had to be generated, we observed a decrease of the average overall number of branching steps with increasing number of processors. This behaviour is called *acceleration anomaly* ([2]). Acceleration anomalies occur if the search tree generated in the parallel case is smaller than the one generated in the sequential case. This can happen in the parallel case because of

branching several subproblems at the same time. Therefore it is possible to find an optimum earlier than in the sequential case. Acceleration anomalies cause a disproportional decrease of the average maximum number of branching steps per processor with increasing number of processors, a super speedup.

Table 2. Average overall number of reduction steps

	1 proc. functional	1 proc. skeleton	4 proc. skeleton	6 proc. skeleton	8 proc. skeleton	16 proc. skeleton
(a)	17197	20727	1848,2	1792,9	1074,9	709,3
(b)	54	47	138,0	218,6	299,0	451,6
(c)	138	118	179,8	261,8	258,6	484,8

Table 3. Average maximum number of reduction steps per processor

	1 proc. functional	1 proc. skeleton	4 proc. skeleton	6 proc. skeleton	8 proc. skeleton	16 proc. skeleton
(a)	17197	20727	896,0	710,2	390,5	113,1
(b)	54	47	43,2	44,7	47,0	46,4
(c)	138	118	65,5	66,1	60,9	51,7

To compare sequential functional programs with programs defined by means of skeletons, we counted reduction steps. The reduction steps include besides branching a problem, the computation of a bound of the optimal solution of a subproblem, the comparison of these bounds for selection and elimination of a subproblem from a set of to be solved subproblems, and the comparison of bounds to determine an optimum. Table 2 shows the *average overall number of reduction steps* of all processors participating in the computation. In Table 3 the *average maximum numbers of reduction steps per processor* are given. Table 2 and Table 3 clearly show the described effect of an acceleration anomaly. Because the set of reduction steps contains comparison steps of the operation ‘selection of subproblems from a set of to be solved subproblems’, the distribution of work causes a decrease of the number of comparison steps for this operation at each processor.

Looking at both Table 2 and Table 3 it becomes apparent that the numbers of reduction steps in the sequential cases of (a), (b), and (c) of the computation of the problem first defined in the standard functional way and second using the skeleton differ. That is caused by different styles of programming in functional and imperative languages. In case (a) an obvious decrease of the average maximum number of reduction steps per processor (Table 3) caused by the distribution of the subproblems onto several processors is observable. At the same time the average overall number of reduction steps (Table 2) is also decreasing

as explained before. The distribution of work onto several processors yields a large increase of efficiency in cases when a large part of the problem tree must be computed. In case (b) the average maximum number of reduction steps per processor nearly does not change while the overall number of reduction steps is increasing, because, firstly, subproblems, which are to be branched, are distributed, and secondly, a larger part of the problem tree is computed. Because in case (b) the solution can be found in a short time, working parallelly as well as sequentially, the use of several processors produces overhead only. In case (c) the same phenomena as in case (b) are observable. Moreover, the average maximum number of reduction steps decreases to nearly 50% in case of parallel computation in comparison to the sequential computation.

4 Conclusion

The concept, implementation, and application of task parallel skeletons in a functional language were presented. Task parallel skeletons appear to be a natural and elegant extension to functional programming languages. This has been shown using the language DFS and a parallel branch-and-bound skeleton as an example. Performance evaluations showed that using the implemented skeleton for finding solutions for a machine scheduling problem is performance better, especially if a large part of the problem tree has to be generated. Also in the case of the necessity to compute a smaller part of the problem tree only, a distribution of work is advantageous.

Acknowledgements The author would like to thank Herbert Kuchen and Hermann Härtig for discussions, helpful suggestions and comments.

References

1. Botorog, G.H., Kuchen, H.: Efficient Parallel Programming with Algorithmic Skeletons. In: Boug, L. (Ed.): Proceedings of Euro-Par'96, Vol.1. LNCS 1123. 1996.
2. de Bruin, A., Kindvater, G.A.P., Trienekens, H.W.J.M.: Asynchronous Parallel Branch and Bound and Anomalies. In: Ferreira, A.: Parallel algorithms for irregularly structured problems. Irregular '95. LNCS 980. 1995.
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press. 1989.
4. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel Programming Using Skeleton Functions. In: Bode, A. (Ed.): Parallel Architectures and Languages Europe : 5th International PARLE Conference. LNCS 694. 1993.
5. Darlington, J., Guo, Y., To, H.W., Yang, J.: Functional Skeletons for Parallel Coordination. In: Haridi, S. (Ed.): Proceedings of Euro-Par'95. LNCS 966. 1995.
6. Park, S.-B.: Implementierung einer datenparallelen funktionalen Programmiersprache auf einem Transputersystem. Diplomarbeit. RWTH Aachen 1995.
7. Trienekens, H.W.J.M.: Parallel Branch and Bound Algorithms. Dissertation. Universität Rotterdam 1990.