Draft Proposal of an EC Funded Project
A Standard Skeleton Library

1 Situation

Today, parallel programming of MIMD machines with distributed memory is typically based on low-level message passing. Owing to the availability of standard message passing libraries such as MPI [GLS99] and PVM [Ge94], the resulting software is no longer platform dependent but can be exchanged easily between parallel machines.

Many approaches to providing a higher programming level have been proposed. Few of them could gain significant acceptance by parallel programmers. It is impossible to mention all high-level approaches to parallel programming here. Thus, we will focus just on a few interesting ones.

Bulk synchronous parallel processing (BSP) [Va90, SHM97, MC96] is a restrictive model where a computation consists of a sequence of *supersteps*, i.e. independent parallel computations followed by a global communication and a barrier synchronization. The advantage of this restrictive approach is that a cost model for BSP is available which allows us to estimate the costs of some parallel computation on some specific machine based on certain parameters of that machine. BSP has been successfully applied to a few data-parallel application problems [BSP], but owing to its restrictive model it cannot easily be used for irregularly structured problems such as adaptive multigrid methods for the numerical solution of differential equations. Moreover, it does not provide parallel composition as an independent concept and hence it cannot easily express divide-and-conquer algorithms. A parallel computation is still seen as a collection of local computations at each processor rather than from a global perspective.

An even higher programming level than BSP is provided by *algorithmic skeletons*, i.e. typical parallel programming patterns which are efficiently implemented on the available parallel machine and usually offered to the user as higher-order functions, which get the details of the specific application problem as argument functions. Thus, a parallel computation consists of a sequence of calls to such skeletons, possibly interleaved by some local computations. The computation is now seen from a global perspective. Several implementations of algorithmic skeletons are available. They differ in the kind of host language used and in the particular set of skeletons offered. Since higher-order functions are taken from functional languages, many approaches use such a language as host language [Da93, KPS94, 2, LHF00]. In order to increase the efficiency, imperative languages such as C and C++ have been used as host languages, too [BK96a, BK96b, BK96c, BK98, 1, B+93].

Depending on the kind of parallelism used, skeletons can be classified into *task parallel* and *data parallel* ones. In the first case, a skeleton (dynamically) creates processes, which run concurrently. Some examples are `pipe`, `farm` and `divide&conquer` [B+93, Co89, Da93]. In the second case, a skeleton works on a distributed data structure, performing the same operations on some or all elements of this structure. Data parallel skeletons, such as `map`, `fold` or `rotate` are used in [B+93, BK98, Da93, Da95, KPS94, 2]. They can be implemented very efficiently.

Although skeletons have many advantages, they are rarely used to solve practical application problems. One of the reasons is that there is not a common system of skeletons — each research group has its own approach. By agreeing on some common set of skeletons, the acceptance of skeletons could be increased. Moreover, it would be easy to exchange tools such as cost analyzers, optimizers, debuggers and so on.

2 Goals

Thus, the idea is to agree on some set of skeletons. The implementation of these skeletons should be platform independent. This could e.g. be achieved by implementing them on top of MPI or possibly BSP. In order to facilitate the use of skeletons, they should ideally be available as a library which can easily be linked to an application program. In order to enable as many users as possible to access this library there should be interfaces to the main programming languages used by parallel programmers, i.e. to C, C++, and Fortran. Moreover, it would be interesting to have an interface to a functional language such as Haskell or ML.

2.1 Implementing the Skeleton Library

There are two aspects that complicate the development of such a library, namely the fact that skeletons are higher-order functions and that they are typically polymorphic. Neither concept is available in C or Fortran. Thus, convincing solutions must be found for connecting a first-order language with a monomorphic type system to polymorphic higher-order skeletons. Skil [BK98] achieves this by using a precompiler that removes higher-order functions by some kind of lambda lifting [Jo85] and instantiates polymorphic functions to monomorphic ones. If the skeletons are to be provided as a library, such a precompilation is no longer possible. A solution to the typing problem could be for the library to support types much as MPI does, namely to have library functions that inform the system about the types to be used. A similar approach could be used for telling the skeleton runtime system which argument functions will be provided by the user.

The Skil implementation could be used as a starting point for the implementation of the skeleton library.

Another consequence of the fact that the skeletons will be provided as a library is that it will not be possible to nest them. This means that nested skeletons must be decomposed into a sequence of calls to unnested skeletons. This is a restriction, although for typical applications not a very serious one.

2.2 Kernel Skeletons

In order to gain acceptance within the parallel computing community, it is important that the skeletons are implemented as efficiently as possible. Since most parallel applications are data parallel, it is particularly important that the data parallel skeletons work efficiently. In order to simplify the implementation and limit the implementation effort, a small set of kernel skeletons could be implemented directly, whilst other skeletons could be implemented on top of them. The kernel should include data parallel computation skeletons such as `map` and `fold` as well as data parallel communication skeletons such as `permute`. The latter cause some exchange of elements of a distributed data structure among the processors. Other communication skeletons such as `broadcast`, `scatter`, `gather`, `all-to-all`, and so on can be implemented on top of more basic skeletons such as `permute`. Moreover, task parallel skeletons such as `pipe`, `farm`, and `divide-and-conquer` can be implemented on top of the data parallel ones.

2.3 Cost Model

Just as for BSP, there are also cost models for skeletons (see e.g. [B+93, LHF00]), which allow us to estimate the costs of a parallel application using the skeletons. These cost models must be adapted to the skeleton library. The cost model will be used by the tools that will be developed for the skeleton library, in particular for the cost analyzer and the skeleton optimizer.

2.4 Tools

The agreement on a standard skeleton library will stimulate the development of various corresponding tools. First of all, tools for testing and debugging applications using skeletons will have to be developed. Moreover, there will need to be tools for estimating the running time of an application on a particular platform. These tools will use the cost model as well as machine specific parameters in the spirit of BSP. Another interesting tool could be constructed for clustering computations on certain processors. Thus, the application program could be written for some virtual machine with an optimal number of logical processors. Then, the clustering tool must map these virtual processors to physical ones so as to allow as much communication as possible among virtual processors mapped to the same physical processor. Furthermore, optimizers can be developed that take a skeleton-based application written in some host language such as C and transform it to a program that runs according to the cost model more efficiently on a given parallel machine. Such a tool will need to parse the source program in order to find sequences of calls to skeletons that can be replaced by more efficient sequences. Thus, they will not be language independent, at least not as far as parsing is concerned.

3 Administrative Issues

We intend to apply for an EC-funded project in order to carry out the activities sketched above. Before doing that a few points must be clarified. Who is willing to coordinate this project? What kind of funding should be requested? Should we apply for a network of excellence which pays for traveling, workshops, and maybe equipment for video conferences? Or should we try to get funding for research positions? What deadline should we fix?

References

[BK96a] G.H. Botorog, H. Kuchen: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, in *Fifth International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, 1996.

[BK96b] G.H. Botorog, H. Kuchen: Using Algorithmic Skeletons with Dynamic Data Structures, in Proceedings of Irregular '96, LNCS 1117, Springer, 1996.

[BK96c] G.H. Botorog, H. Kuchen: Efficient Parallel Programming with Algorithmic Skeletons, Proceedings of EuroPar '96, LNCS 1123, Springer, 1996.

[BK98] G. H. Botorog, H. Kuchen: Efficient High-Level Parallel Programming, Theoretical Computer Science 196, pp. 71-107, 1998.

[BSP] BSP Worldwide web page, http://olwww.comlab.ox.ac.uk/oucl/groups/bsp/

[B+93] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi: P^3L : a Structured High-level Parallel Language and its Structured Support, Technical Report HPL-PSC-93-55, Hewlett-Packard Laboratories, Pisa Science Center, 1993.

[Co89] M.I. Cole: Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, 1989.

[Da93] J. Darlington, A.J. Field, P.G. Harrison, et al.: Parallel Programming Using Skeleton Functions, in Proceedings of PARLE '93, LNCS 694, Springer, 1993.

[Da95] J. Darlington, Y. Guo, H.W. To, J. Yang: Functional Skeletons for Parallel Coordination, in Proceedings of EuroPar '95, LNCS 966, Springer, 1995.

[1] I. Foster, R. Olson, S. Tuecke: Productive Parallel Programming: The PCN Approach, in Scientific Programming 1(1), 1992.

[Ge94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

[GLS99] W. Gropp, E. Lusk, A. Skjellum: Using MPI, MIT Press, 1999.

[Jo85] T. Johnsson: Lambda Lifting: Transforming Programs to Recursive Equations, in Proceedings of FPCA '85, LNCS 201, Springer, 1985.

[KPS94] H. Kuchen, R. Plasmeijer, H. Stoltze: Distributed Implementation of a Data Parallel Functional Language, 6th International Conference on Parallel Architectures and Languages Europe (PARLE'94), LNCS 817, pp. 464-477, Springer Verlag, 1994.

[LHF00] F. Loulergue, G. Hains, C. Foisy: A Calculus of Functional BSP Programs, Science of Computer Programming 37, pp. 253-277, 2000.

[MC96] W. F. McColl: Scalable Computing, Computer Science Today: Recent Trends and Developments, LNCS 1000, pp. 41-61. Springer-Verlag, 1996.

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, W. F. McColl: Questions and answers about BSP, Scientific Programming 6(3): 249-274, 1997.

[2] D. Skillicorn: Foundations of Parallel Programming, Cambridge University Press, 1994.

[Va90] L.G. Valiant: A bridging model for parallel computation, Communications of the ACM 33(8), p. 103, 1990. Bratvold, Lengauer, Gorlatch, Herrmann, ...?