Programming Frames for the Efficient Use of Parallel Systems*

Thomas Römke† Jordi Petit i Silvestre‡

Abstract

Frames will provide support for the programming of distributed memory machines via a library of basic algorithms, data structures and so-called programming frames (or frameworks). The latter are skeletons with problem dependent parameters to be provided by the users. Frames focuses on re-usability and portability as well as on small and easy-to-learn interfaces. Thus, expert and non-expert users will be provided with tools to program and exploit parallel machines efficiently.

Frames will be constructed for different target machines and common programming environments (like PVM or MPI). The focus, however, is on distributed-memory machines. Frames will be adapted optimally to the target systems, contain efficient state-of-the-art programming techniques, and therefore increase the usability and therefore the acceptance of parallel computing.

Key words: Efficiency, Re-usability, Portability, state-of-the-art algorithms, Templates, Skeletons, Frames

1 Motivation

Parallel computing systems offer a cost efficient way to provide the large computing power that is necessary to handle the problems of increasing complexity in science and engineering. However, there are only a few people trained to program these machines efficiently. Because of this, parallel

*This work was partly supported by the EU ESPRIT Long Term Research. Project 20244 (ALCOM-IT).

†Paderborn Center for Parallel Computing (PC^2), University of Paderborn, Germany.

‡Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain.

systems are not as widely accepted and usable as possible. Our aim is to improve this situation by providing a way to encapsulate expert knowledge in so-called *programming frames*.

Programming frames facilitate the development of efficient parallel code for distributed memory systems. They are intended to be used by experts on the one hand, and by non-experts who are either unfamiliar with parallel systems or unwilling to cope with new machines, environments and languages, on the other.

Several other projects (eg templates [7], skeletons [8, 9, 10, 11, 12], clips [13, 14], or the BACS approach [15, 16]) have been initiated to develop new and more sophisticated ways for supporting the programming of distributed memory systems via libraries of basic algorithms, data structures and programming templates. Like LEDA [17] for the sequential case, each of these approaches provides tools to program and exploit parallel machines efficiently.

Our frames are like black boxes with problem dependent holes. The basic idea is to comprise expert knowledge about the problem and its parallelization into the black box and to let the user specify the holes only, ie the parts that are different at each instantiation. The black boxes are either constructed using efficient basic primitives, standard parallel data types, communication schemes, load balancing and mapping facilities, or they are derived from well optimized, complete applications. In any case, frames contain efficient state-of-the-art techniques focusing on re-usability and portability. This saves software development costs and improves the reliability of the target code.

Consider the Finite Element frame that is depicted in Figure 1 and constructed by two different experts: an expert of the algorithm and an expert of parallelization. The boxes shown in this example are the interchangeable parts of the frame. The experts can, for example, exchange the Communication Library or the Load Balancer (usually done by the parallelization expert) or the Solver (usually done by the algorithm expert). The advantage for the parallelization expert is that he can easily provide his knowledge and test more sophisticated versions of his parallel codes. The advantage for the algorithm expert is that he does not have to become an expert in parallelization and is getting an efficient parallel program. The end user, finally, will only have to specify his problem dependent holes, and the Frames system will select the appropriate parts for him (according to some rules given by the experts). For the Finite Element frame, for example, the end user would give some values for the *dimension of the problem*, the *dimension of the space*, some *boundary conditions*, the derivation *d_t*, etc.

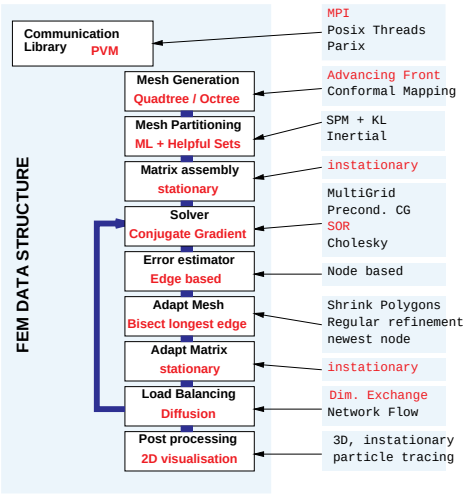

Figure 1: A Finite Element Method example

A brief abstract of the work described in this paper has been given in [1] as a demonstrator of our work aiming at the efficient use of parallel and distributed systems. [2] and [3] describe the way our frame approach can be used for Metacomputing and Finite Element programs. [4, 5, 6] finally give an overview on the current state of our system. This is the first paper describing the concept behind our system and its features in detail.

In Section 2 we will shed some light on the features of our system that are re-usability, portability, efficiency, state-of-the-art algorithms, and ease of use. Then we will compare it briefly to other approaches. Section 3 will explain the frame model behind our approach and how we have implemented it. It will describe the three major levels that are involved with our system and the tools that are used to process them. We will use the Finite Element Method frame as an example for all of these levels, and explain the interaction with the graphical user-interface that we provide. We finally will conclude this paper with an overview of our current and future work.

2 Features and other approaches

The Frames approach is basically aimed at the encapsulation of problem independent functionality and expert knowledge, letting the end user only specify those parts that are really needed for the corresponding kind of problem. For example, in a Divide-and-Conquer frame, we would have two

parameters corresponding to the `Problem` and `Solution` types and four parameters for the `Divide`, `Combine`, `EasySolution` and `Indivisible?` functions.

Frames should enhance the knowledge on parallel computing that is already available in the community but until now only used in dedicated applications. This knowledge can be extracted from the experts' applications, encapsulated, and then be re-used by many others. If, for example, there is already an efficient parallel Divide-and-Conquer, there is no reason to program it again. Others can take advantage of the existing one. In our opinion, Frames will enable parallelization experts to apply the same kind of modularity and re-usability to their programs as they are used to in sequential programming.

Frames give support for the software engineers and can —from the users' point of view— provide some kind of "automatic parallelization". There are two reasons for this. First, the user has to provide the parameters only, which are sequential. All implementation details, and therefore the parallelization as well, are hidden from him. Secondly, the user does not have to change his specification when switching to a different target system or environment. As long as corresponding frames are provided, the same instance can be used without any changes. This could be an argument for a manufacturer as well.

Finally, we will combine the knowledge of at least two experts, the expert of the algorithm in mind and a parallelization expert. The main observation here was that there is a large number of algorithm experts that want to take advantage of parallelism, but do not want to become experts in parallelization first. These experts will be aided by the frames. In the following, we develop the main features of our system, which are *re-usability*, *portability*, *efficiency*, *state-of-the-art algorithms*, and an *easy graphical user interface*.

Re-usability —as stated above— means that the frame parameters can be re-used when switching to other target machines. Furthermore, since frames can be built from others, certain "basic frames" can be provided to the expert. These basic frames contain parts that are frequently used in parallel programming, eg load balancers, parallel data types, or communication schemes.

We can grant portability at least for the parameter values because we require them to be given in the C syntax. For the implementation sources that are also written in C but belong to a certain target environment (like Parix, MPI, or PVM) portability is given as long as no system specific items are used. Of course, this doesn't mean that an efficient MPI implementation for a certain target machine is also efficiently running on another one.

Programming frames contain state-of-the-art programming techniques provided by experts of both the algorithm and the parallel system in mind. We therefore suppose the algorithms to be efficient as well. Whether or not it can be assured that composed frames are efficient is still an open question.

Graphical user interfaces are vital for the acceptance of frames by the users. For the expert, the GUI will offer graphical support for the composition (construction, modification) of frames and therefore ease the handling of the Frame description language. For the end user, it will give detailed information on existing frames and on the parameters to specify. The parameter values can be input, and the target frames be selected by several criteria. The corresponding sources are then adapted optimally according to the parameter values to the target system.

Moreover, the system itself is portable and easy to use. It is made of three different compilers written in Ansi C and generated by ELI [18] and by graphical tools written in Java.

Other approaches. The most common names for frameworks are frames, skeletons, and templates. The major difference between them is the type of the target implementation language, the type of the frame description language, as well as the level of expertise the end user must have.

The BACS approach [15, 16] was built for small example applications. The process structure and data placement, as well as the interaction can be described in a PASCAL like language. It is supposed to be the optimal tool for experts of parallel machines for testing new algorithms. We couldn't use this approach, since we don't want the user to be an expert, nor do we want him to learn a new complex language (see above).

The group of K. M. Chandy [7] develops a multi-media system that provides users of parallel systems with a library of programming templates. The focus of this project is on software engineering technology to (i) get a parallel program up and running with minimal programming effort (ii) reasoning about correctness and debugging, and (iii) stepwise refinement to obtain greater efficiency. The particular emphasis is on object-oriented technology and re-use. The approach consists of the following steps. (i) A transportable parallel extension of C++ which is now being used to build libraries of parallel templates in mesh computations and linear algebra has been developed. (ii) Two "chapters" of a multi-media "textbook" that are a prototype of all the remaining chapters of the book have been developed. As of now, this approach has only be used for some few applications like solving linear systems.

Murray Cole made his PhD thesis on "Algorithmic Skeletons: Structured Management of Parallel Computation" [8]. In his approach, skeletons are supposed to be high order functions that are describing the structure of a particular algorithm. This approach is restricted to the functional programming world and therefore not appropriate for the general user. Furthermore, Murray wasn't able to present any real-world application running with his system.

P. H. J. Kelly's work is also a demonstrator for skeletons. He discusses skeletons for pipes, farms, ramps, and dynamic programming. A couple of example applications are given, but none that require multiple skeletons. Same restrictions apply as with Cole's approach.

The P3L system of the University of Pisa [9] is based on another skeleton approach. P3L is built from sequential parts (C++) and a set of constructors for the exploitation of parallelism including farms, pipes, data parallelism, loops, trees and geometric. The code is compiled into an intermediate language which provides explicit cost functions. Using this information a mapper selects the most appropriate scheme. Therefore, the user is supposed to be an expert in parallel programming, since he has to select the constructor. Furthermore, there aren't any reports on the efficiency.

In contrast to these approaches, our main observations are the following:

- *Users are only willing to learn as few new things as possible.* Therefore, our approach is based on the C programming language: the elements in the description language, the problem dependent values given by the user and the implementation sources are written in Ansi C.
- *Users and experts want to develop in a nice environment.* For that reason, we would like to make the details of our internal languages transparent by using graphical tools (visual programming).
- In order to become widely available and to *spread our system quickly*, we decided to maintain Frame Repositories in the Internet.

3 The frame model

Our frame model is depicted in Figure 2. In this model, each generated target executable is built from three specifications, an *abstract level specification*, an *instance level specification*, and an *implementation level specification*. The specifications are stored in the **.abs**, **.ins**, and **.imp** files, respectively. The *abstract level specification* is usually given by an expert of the problem or algorithm that should be fit into a frame. It defines the

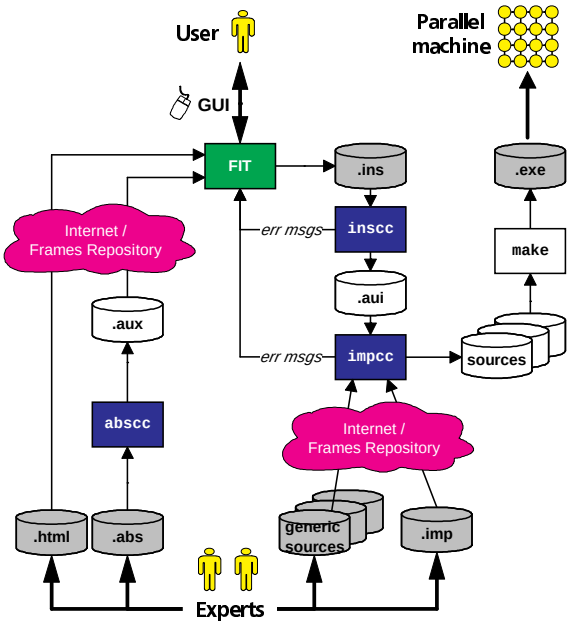

Figure 2: The Frames System

```
ABSTRACT FRAME FEM;
  DECLARATION
    TYPE SolverT = enum { MultiGrid, ConjugateGradient, GaussSeidel,
                          SOR, ConjugateResiduals};
    TYPE STRING = char *;
    TYPE bool = int;
    TYPE BcT = enum {Dirichlet,Neumann,ThirdKind,Userdefined};

    CONSTANT SolverT Solver;
    CONSTANT int pdim = "3";
    CONSTANT int sdim = "2";
    CONSTANT int xdim = "sdim + 1";
    CONSTANT double d_t = "1";

    [pdim] OPTIONAL EXPRESSION double beta([sdim] double u);
    CONSTANT bool beta_takeall = "1";

    ... some more specifications here

  DOCUMENTATION
    DOC(FEM) = URL "http://www.uni-paderborn.de/ alcom-it/frames/dp/fem.html";

END FEM.
```

Figure 3: A part of the abstract level specification for the FEM frame

properties of the parameters of the problem and documents them (**.html** files). These parameters can be viewed as the holes that later on will be filled by the user. In the *instance level*, values are bound to the parameters specified in the corresponding abstract level specification. In this way, the user "tailors" the general problem to his needs, or in other words, he is getting a new instance of the given frame. Finally, for the *implementation level*, we need both, an expert of the problem and an expert of the target machine. Together they will provide an *implementation level specification*. This specification consists of a set of generic source files together with rules on how to link these sources with the values provided by the user in the instance level. Furthermore, this level gives opportunity to re-use or compose existing frames for this purpose.

In the following, we discuss in more detail the characteristics of these three levels and the languages and tools we need to achieve our objectives. We will also give example descriptions for a Finite Element Method frame.

The abstract specification. For a given problem, the abstract specification defines the properties of its parameters. These parameters are strongly sequential and independent of the possible target architectures. In

our approach, the following objects can be parameters: types, constants, expressions, statements, functions and procedures. The definition of these parameters is given according to the C syntax inside a specification language. Frame parameters can be declared in-coming (default) or out-going, where *in* parameters are supposed to be provided by the user and *out* parameters are provided by the frame. Most parameters can be declared as optional or having a default value. Furthermore, replicators can be used as macros to replicate some parameters a determined or undetermined number of times. Finally, the documentation of the parameters in textual and hypertext (HTML) versions is handled in this level. We found that it is not only necessary to provide the frame itself, but to provide a good documentation as well.

Figure 3 depicts a part of the abstract level specification for the FEM frame. First, four internal types are defined. Then *Solver, pdim, sdim, xdim, and d_t* are specified as constant parameters. A simple value or an enumeration constant has to be given for each of them in the instance level. *beta* is a coefficient. The user has to specify *pdim* expressions each having *sdim* double parameters. If the corresponding *take_all* variable is true, only the first component must be specified. Therefore, the expression is marked as optional. Finally, a documentation is given for the overall frame.

Two of our tools will process this specification: a compiler (the **abscc**) that checks the syntactic and semantic correctness and provides an auxiliary file used in the following levels, and a graphical user interface FIT (**F**rames **I**nstantiator **T**ool).

Frames Repositories. The auxiliary file, the documentation as well as the files needed for the implementation level are stored in so-called Frames Repositories [6]. A Frame Repository is a directory hierarchy kept locally to each site using Frames. Additionally, and to be up-to-date, there will be some Internet mirrors, where a local site can get the latest revision of a certain frame or where you can put your newest sources (if you are the provider of the corresponding frame). The Internet mirrors are supposed to synchronize their content frequently.

Instance specification. The task of this level is to bind values to the parameters and to check their consistency against the types defined in the abstract specification. This is done by the **inscc** tool. The instance level specification itself consists of a list of assignments to the parameters. Since the user is not required to have any knowledge on parallel programming, all

```
FRAME My_FEM IS FEM(MPI)
  CONSTANT Solver = ConjugateGradient;
  CONSTANT pdim = 1;
  CONSTANT sdim = 2;
  CONSTANT xdim = 3;
  CONSTANT d_t = 0.0;
  EXPRESSION beta[1] = 0.0;
  CONSTANT beta_takeall = 1;
  .... some more assignments
END My_FEM.
```

Figure 4: A part of an instance level specification for the FEM frame

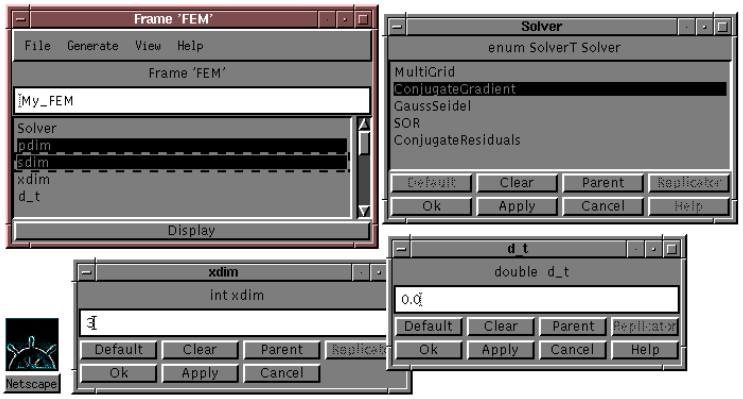

Figure 5: FIT with FEM frame

the parameters he gives are sequential and thus can be run in any architecture. Moreover, the user must not learn any new language for specifying his instance because the instance specification is supposed to be generated by the easy-to-use FIT. Figure 4 depicts a part of an instance level specification for the FEM frame. For all the parameters given in Figure 3, a value is provided here. The name of the instance is *My_FEM*, and the target environment is MPI.

The frames instantiator tool. A GUI for easily acquiring the values of the parameters of the frames is generated automatically by the Frames Instantiator Tool. With this graphical tool, the user can easily access the Frames Repositories, input the values for the abstract parameters, display their corresponding documentation, select its target architecture, and launch the compilation processes. The code of Figure 4 could then be obtained using

```
FRAME FEM(MPI) IMPLEMENTATION
  RESOURCES "fem.h", "fem.c", "Makefile"

  "_incl.h" {
    APPEND
      "/* This file has been processed by the Framework System. */\n"
      "/* (c) ALCOM IT */\n\n"
      "#ifndef _incl_INCLH_\n"
      "#define _incl_INCLH_\n"
      "#define _FRAME_pdim (" FEM.pdim ")\n"
      "#define _FRAME_sdim (" FEM.sdim ")\n"
      "#define _FRAME_xdim (" FEM.xdim ")\n"
      "#define _FRAME_d_t (" FEM.d_t ")\n"
      "#define _FRAME_Solver_" FEM.Solver " 1\n"
      ;
    CALL PERL
      -{- # copy in to out
         while(<>)  print STDOUT ;
         .... some more modifications here
      -}-
    }
    .... some more rules here
END FEM.
```

Figure 6: A part of an implementation level specification for the FEM frame

the FIT, which is depicted in Figure 5.

On the other hand, the expert could also provide a dedicated GUI for his frame [2]. However, this involves a considerable extra effort that the FIT aims to save providing generality, portability and uniformity.

Implementation specification. This level contains a number of (usually parallel) implementation sources, one for each architecture we support (eg MPI or PVM). The **impcc** links the selected implementation with the parameters provided by the user according to a set of rules that specify transformations on the source code and the order in which they must be applied. These transformations are mainly *replacements* and *generations*.

Simple replacements and generations, their relative order and the resources they modify or create can be written in our specification language. Nevertheless, we found that static replacements and generations will generally not suffice. Therefore, we provided an interface for calling widely available and portable pre-processors or script languages. We decided to keep our specification languages as small as possible, not to force an implementor to learn new complex tools. Currently, we support the TCL and Perl languages for this purpose.

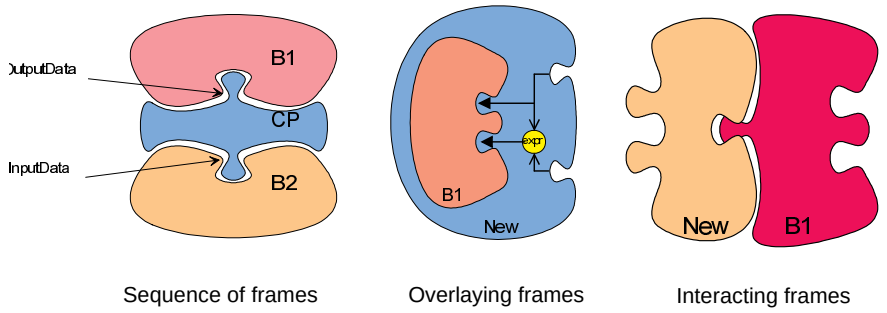

Figure 7: Composing frames

Figure 6 depicts a small part of an implementation level specification for our example. It takes the resource files *fem.h*, *fem.c*, and *Makefile*, copies them to a local area and then starts processing. In this figure only the generation of an additional header file is visible.

Composing Frames. Once a frame exists, one of our objectives is to re-use it when implementing new frames. This is done by composing frames. We have found three useful ways of composing frames: sequencing, overlaying and interacting. They are depicted in Figure 7.

A *sequence of frames* $F_1, ..., F_n$ would execute F_1 first, then F_2, and further on until F_n. These frames can be connected by other frames that can be used to redistribute data between processors. In our example, the *B1* frame is executed first, then the data is redistributed by the *CP* frame, and finally *B2* is executed.

Overlaying frames means re-using existing frames by defining the relationship between its parameters and the parameters of the used frame, eg supplying fixed values or taking an expression of given parameters values. In our example, both the *New* and the *B1* frame have two parameters. *New* is using *B1* by passing the first parameter value without modification, and calculates the second one as an expression of its own two parameters.

Interacting frames can provide access to other frames as if they were abstract data types or parallel threads. In this way, one frame can use properties provided by other frames. In our example, *B1* could for example be a load balancer that is running in parallel. Like with sequences of frames, there could be some more frames in between that are used to redistribute the data.

The composition of existing frames to design a new one will be done by graphical tools (visual programming) and is right now a part of our current work.

4 Conclusion

We presented our approach for solving the lack of software solutions in parallel programming. Our approach is useful for both, experts and non-experts. Experts can easily exchange important parts of their parallel applications and provide their knowledge to others. Non-experts get an efficient parallel version tailored to their needs.

Our frames model mainly consists of three parts. The first one is used to describe properties of the problem in mind. It remains the same for all instances that are taken for this particular frame and for all implementation specifications that are written for it. The second part is handling the instances. It must comply with the abstract specifications and contains a list of assignments to the frame parameters. An instance specification can be re-used when switching from one target environment to another. Finally, the implementation specifications complete our system. They are used to specify the modifications that have to be applied to the state-of-the-art source codes according to the parameter values given in the instance level.

In the near future we will provide the composing features with our tools and also the first basic frames (like load balancing, mapping, or communication patterns) for MPI and PVM. Furthermore, the first real world applications using our system will be available. These are the complex frameworks for our Finite Element code and for Genetic Algorithms.

Acknowledgments. We would like to thank Ralf Diekmann and Uwe Dralle who provided us with the information on the Finite Element Methods, and Josep Díaz for stimulating discussions and their help in preparing this paper. We also acknowledge the help of Jordi Giribet who helped implementing the graphical tools.

References

[1] Burkhard Monien, Ralf Diekmann, Rainer Feldmann, Ralf Klasing, Reinhard Lüling, Knut Menzel, Thomas Römke, Ulf-Peter Schroeder, *Efficient Use of Parallel & Distributed Systems: From Theory to Prac-*

tice, Lecture Notes in Computer Science No. 1000, pp. 62-77, Springer-Verlag, 1995

[2] Ralf Diekmann, Uwe Dralle, Friedhelm Neugebauer, Thomas Römke, *PadFEM: A Portable Parallel FEM-Tool*, HPCN Europe'96, Belgium, Brussels, April 15-19, 1996

[3] Alexander Reinefeld, Ranieri Baraglia, Thomas Decker, Jörn Gehring, Domenico Laforenza, Friedhelm Ramme, Thomas Römke, Jens Simon, *The MOL project: An Open, Extensible Metacomputer*, Heterogeneous Computing Workshop HCW'97, April 1, 1997, Geneva, Switzerland.

[4] Jordi Petit i Silvestre, Jordi Giribet Perich, Thomas Römke, Uwe Dralle, *FIT: A Frame Instantiator Tool*, Technical Report LSI-96-14-T, Universitat Politècnica de Catalunya, Barcelona, Spain

[5] Jordi Petit i Silvestre, Thomas Römke, *The Frames Poster*, Technical Report LSI-97-1-T, Universitat Politècnica de Catalunya, Barcelona, Spain

[6] Thomas Römke, Jordi Petit i Silvestre, *The Frames Web Pages*, `http://www.uni-paderborn.de/~alcom-it/frames/`

[7] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Tech. Rep., CS-Dept., Univ. of Tennessee, 1993. `http://www.netlib.org/templates/templates.ps`

[8] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, PhD, Research Monographs in Par. and Distr. Computing, MIT Press.

[9] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, M. Vanneschi: A Methodology for the Development and the Support of Massively Parallel Programs, *J. on Future Generation Computer Systems (FCGS)*, Vol. 8, 1992.

[10] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu: *Parallel Programming Using Skeleton Functions*, Proc. of Par. Arch. and Lang. Europe (PARLE '93), Lecture Notes in Computer Science No. 694, Springer-Verlag, 1993.

[11] J. Darlington, M.J. Reeve, S. Wright, *Programming Parallel Computer Systems using Functional Languages and Program Transformations*, Parallel Processing 89, Leiden

[12] J. Darlington, Y. Guo, H. Wing To, J. Yang, *Parallel Skeletons for Structured Composition*, PPOPP 95, Santa Clara

[13] J. Dvorak, *Two Perspectives on the EOOPS Theme*, Programming Environments for Parallel Computing, IFIP WG 10.3 Workshop, IFIP transactions, 1992

[14] J. Dvorak, *Using Clips in the domain of knowledge-based massively parallel programming*, Programming Environments for Parallel Computing, IFIP WG 10.3 Workshop, IFIP transactions, 1992

[15] H. Burkhart et al, *BACS: Basel Algorithm Classification Scheme*, and *Werkzeuge und Methoden des Skelettorientierten Programmierens*, Technical reports, University of Basel, 1992-1993

[16] H. Burkhart, S. Gutzwiller, *Steps towards re-usability and portability in parallel programming*, in K.M. Decker and R.M. Rehmann (Eds.), Programming Environments for Massively Parallel Distributed Systems, 147-157, Birkhäuser Verlag, Basel 1994

[17] K. Mehlhorn, S. Näher, *LEDA, a Library of Efficient Data Types and Algorithms*, Proc. 14th Int. Symp. on Mathem. Foundations of Computer Science (MFCS '89), Lecture Notes in Computer Science No. 379, Springer-Verlag, pp. 88-106, 1989.

[18] University of Boulder, University of Paderborn, *The Eli System*, `http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_explainE.html`