Skeletons and Transformations in an Integrated Parallel Programming Environment*

Bruno Bacci[1], Sergei Gorlatch[2], Christian Lengauer[2], and Susanna Pelagatti[3]

[1] Quadrics Supercomputers World Ltd., Via S. Maria 83, I-56125 Pisa, Italy
[2] Universität Passau, D-94030 Passau, Germany
[3] Universitá di Pisa, Corso Italia 40, I-56125 Pisa, Italy

Abstract. We sketch an integrated environment for the systematic development of parallel and distributed programs. Our approach allows the user to construct complex applications by composing and transforming *skeletons*, i.e., recurring patterns of task and data parallelism. First academic and commercial experience with skeleton-based systems has demonstrated both the benefits of the approach but also the lack of a dedicated set of methods for algorithm design and performance prediction. We take a first step towards such a set of methods by proposing an environment which integrates a transformational framework, called FAN, with two existing skeleton-based programming systems: P3L and SkIE.

1 Introduction

Current difficulties in low-level parallel and distributed programming using, e.g., the MPI (Message Passing Interface) standard [14] can be addressed by high-level programming models together with convenient programming environments.

A number of parallel programming environments are already available. For instance, in HeNCE (Heterogeneous Network Computing Environment) [4, 5], applications are written in C or Fortran77 and run on top of PVM. The HeNCE programmer writes parallel applications by graphically drawing the interrelationships between the different (sequential) process components of the parallel application. The ANNAI project [8] led to the development of a set of tools including PST (a parallelization support tool), PMA (a performance monitor and analyzer) and PDT (a parallel debugging tool). The kinds of code restructuring and optimization that can be carried out by these environments are rather limited. Decisions concerning difficult problems such as scheduling, mapping, load balancing and data distribution are made on the basis of a few weak heuristics, since there is little knowledge on the parallel structure being defined. This forces the user to restructure the code by hand both when tuning performance on a particular machine and when porting an application to a different machine.

An alternative, higher-level approach is based on so-called *skeletons* [9], which can be viewed as recurring algorithmic and communication patterns, expressed

* *Contact author:* Sergei Gorlatch, University of Passau, D-94030 Passau, Germany. Tel: +49 851 509-3074, Fax: +49 851 509-3092, Email: gorlatch@fmi.uni-passau.de

in a rigorous way [15]. Representatives of skeleton-based systems are the P3L system at the University of Pisa [3], its commercial analogue SkIE at QSW Ltd. [2], SKIL at RWTH Aachen [6], and SCL at Imperial College [10]. These systems provide the user with a fixed number of higher-order skeletons, which can be customized for a particular application. A skeletal program is then translated (semi)automatically to some target language, e.g., C plus MPI, using prepackaged parallel implementations of skeletons. The abstraction from communication and other details gives skeletal programs considerably better structure and makes them less error-prone than their low-level counterparts.

In the long run, the *approach of skeleton-based programming* should include methods and tools for choosing suitable skeletons, composing them to a program, estimating its expected performance, and making changes for better efficiency. Our present proposal has grown out of experience in transformational programming [1, 13], compiler optimization [7] and efficiency analysis [11]. In particular, we have proposed a framework, called FAN (for Formal Abstract Notation), for transforming parallel algorithms at a high level of abstraction [12].

Here, we sketch an integrated environment which provides the user with specific methods and tools for skeleton-based program development. The environment extends the existing versions of the systems P3L and SkIE by transformation methods for application algorithms which are composed of skeletons. We describe the main parts of the environment and the way in which the user interacts with it. In the full paper, we will add an assessment of the environment on a case study.

2 System Structure Overview

In this section, we take the P3L system [3] as a representative of skeleton-based systems, and outline how it is augmented by the FAN transformational framework. The overall structure of the resulting environment is presented in Figure 1.

The figure shows how the user communicates via the visual support system (described in Section 4) with the programming environment. The latter is partitioned by horizontal fat, solid lines into three parts – from top to bottom: the transformational framework, the P3L system, and the target machines. Solid arrows show the connections between the parts of the system, dashed arrows depict the user's interaction with the system, with bold, dashed arrows for the new interactions added by the transformational framework.

In the P3L system, the user starts the development by writing a complete skeletal P3L program (in the middle of Figure 1). The user must provide the complete skeleton-based algorithm and also supply all necessary sequential modules, input and output files. The program is optimized and translated by the P3L compiler, which provides the user with preliminary cost estimates for the program. If the user is satisfied with the cost, the C plus MPI code produced by the compiler can be run on an available target machine (some current platforms targeted by the P3L compiler are shown in the figure).

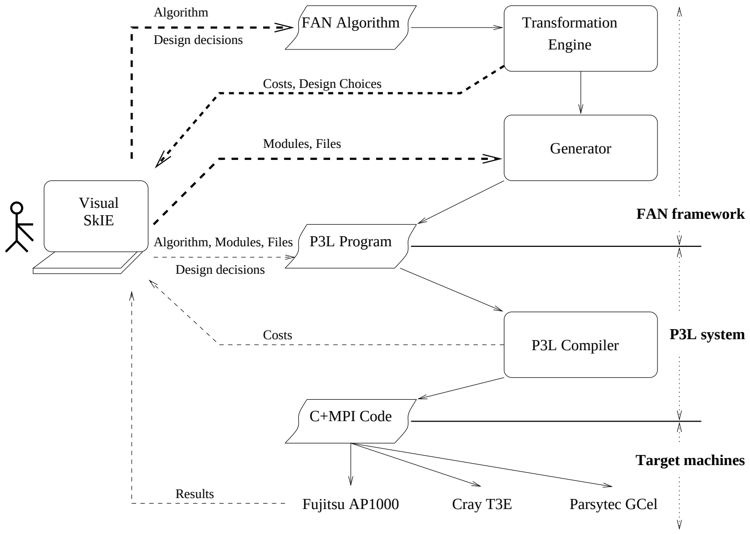

Fig. 1. FAN on top of P3L

The transformational FAN framework, shown in the upper part of Figure 1, offers the user additional support in designing a skeletal program. Using FAN, the design process starts by writing a functional version of the algorithm, without providing concrete modules and files. The algorithm is analyzed by the transformation engine, which attempts to apply transformations from its depository of rules, thereby suggesting a choice of design alternatives to the user, with a cost estimate for each alternative. After, possibly, several iterations of design choices, the user may decide to generate a P3L program, which is then compiled and executed as described above.

3 Skeletons and Transformations

The skeletons available in the integrated environment can be divided into three classes: *control* skeletons, used to encapsulate sequential or unstructured parallel code; *stream-parallel* skeletons, modeling parallel structures with task parallelism; and *data-parallel* skeletons.

Control skeletons:

`seq`: encapsulates code written in a sequential language (the *host language*) in a module with well defined in-out interfaces. Sequential languages currently supported include C, as well as C and Fortran plus MPI.

`loop`: iterates skeleton composition finitely or infinitely.

Stream-parallel skeletons:

`pipe`: models pipelined execution of a sequence of SkIECL modules.

`farm`: models a *task farm* computation in which a stream of independent tasks is executed by a pool of equivalent executors (the *workers*).

Data-parallel skeletons:

`map`: applies the same computation to all elements of a data structure.

`reduce, scan`: model the parallel reduction and scan (parallel prefix) on the elements of an array when given an associative binary operator.

`comp`: combines several data-parallel stages.

Figures 2 and 3 depict some of the supported control-parallel, stream-parallel and data-parallel skeletons, with their P3L syntax:

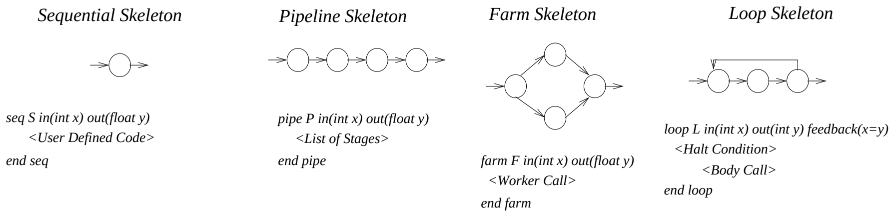

Fig. 2. Control-parallel and stream-parallel skeletons

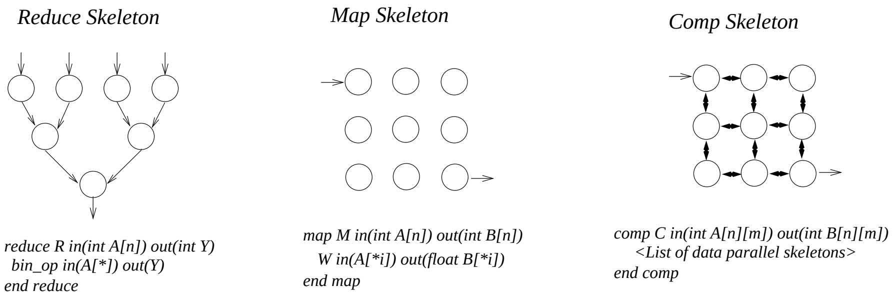

Fig. 3. Data parallel skeletons

The design of a skeleton program consists of transformation and cost estimation steps. The goal of the transformations is to try to reduce the number of communications. This can improve performance substantially. As a non-trivial example, consider the scan-reduce fusion:

Rule SR-ARA

b = $\mathsf{scanL}\ Op1\ a$
c = $\mathsf{reduce}\ Op2\ b$
b = $\mathsf{reduce}\ (New\ (Op1,\ Op2))\,(\mathsf{arrange}\ \ (a,a))$
c = $\mathsf{arrange}\ (\mathsf{proj}\ [1])\ b$
If $Op1$ distributes forward over $Op2$
$(a_1, b_1)\ New(Op1, Op2)\ (a_2, b_2)\ =\ (a_1\ Op2\,(b_1\ Op1\ a_2),\ b_1\ Op2\ b_2)$

The name of the rule, SR-ARA, hints on the transformation it performs: "**S**can;**R**educe $\rightarrow$ **A**rrange;**R**educe;**A**rrange", where $\mathsf{arrange}$ stands for an auxiliary skeleton manipulating data structures. We present transformation rules in a format that consists of four boxes; from top to bottom: (1) the FAN program fragment before the transformation (the "left-hand side" of the rule), (2) the fragment after the transformation (the "right-hand side"), (3) optional: a precondition, stating when the rule is applicable, (4) optional: the definition(s) of new function(s) used by the rule. Rule SR-ARA expects two operators, $Op1$ and $Op2$, as parameters.

A rich set of transformation rules for various skeletons has been developed recently [1, 11, 13, 16].

4 Visual Support

The development of parallel applications is carried out using VisualSkIE (VSkIE), the SkIE graphical working window. Figure 4 shows the VSkIE main window. The horizontal toolbar provides easy access to all main functions and tools.

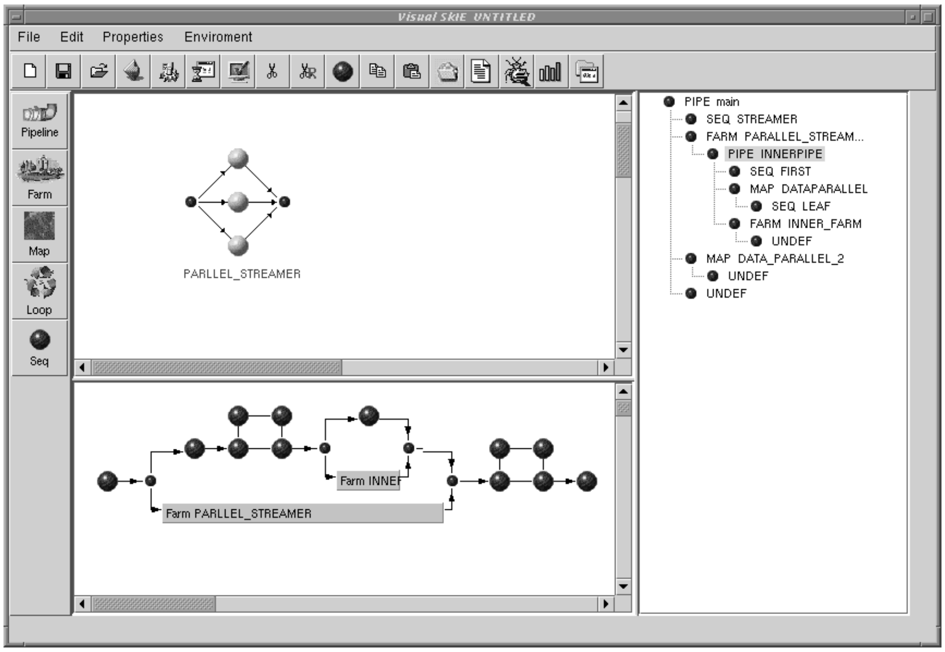

Fig. 4. Visual SkIE, the SkIE working graphical environment

The user can define the global structure of his/her application interactively by editing new sequential parts of the application, using an integrated editor, and by encapsulating already developed sequential/parallel software. The parallel structure of the application can be defined either *explicitly*, in C and Fortran plus MPI, or by using the built-in skeletons. The available skeletons are shown in vertical toolbar on the left.

The three subwindows provide three different views of the application being developed. The upper window shows the logical structure – in this case, a farm skeleton. The lower window shows the global process network being built. The tall window on the right describes how the skeletons are nested to build the global application structure (the *construct* or *skeleton tree*).

In the development process, a new instance of a predefined skeleton can be created interactively. In the dialog box, the user can choose or change the skeleton being defined, specify its input and output parameters, and decide on the skeleton-dependent parameters such as the number of stages in a pipeline or the number of workers in a task farm or in a map skeleton.

After having defined the structure of a parallel application, the VSkIE upper toolbar provides access to the integrated environment functions and tools. In particular, it facilitates the following activities: *transformation and cost estimation* of a skeleton program, *code generation and global optimization* of the application structure, *application debugging*, and *performance analysis*. We will describe the main features of each activity in the full paper, and demonstrate it on our case study.

5 Conclusion

We argue that the implementations of high-level languages should be extended by special programming environments to support the development of efficient, high-level parallel programs. We have sketched an integrated environment which combines the transformational framework FAN with the programming systems P3L and SkIE, and demonstrated the use of the environment. The environment will provide the user with a rapid prototyping tool by automatically producing executable code in C plus MPI, together with expected performance estimates. The current implementation of the environment includes the visual support system, the compiler and the performance estimation tools. We are presently working on the implementation of the transformation engine and on the finalization of the FAN syntax and semantics.

The main novelty of our work is the intensive use of program transformations in the early stages of the programming process, supported by corresponding cost models and programming tools. The framework is language-independent and can be integrated with the existing high-level parallel programming environments, as our experience with P3L and SkIE demonstrates.

Acknowledgements

This work has been supported by a travel grant from the German-Italian academic exchange programme VIGONI.

References

1. M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In *Proc. 1st Int. Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, pages 48–58. Fakultt fr Mathematik und Informatik, Universitt Passau, May 1998. Technical Report MIP-9805.
2. B. Bacci, B. Cantalupo, P. Pesciullesi, R. Ravazzolo, A. Riaudo, and M. Vanneschi. Skie user guide (version 2.0). Technical report, QSW Ltd., Dec. 1998.
3. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.
4. A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. HeNCE: A users' guide. Available at http://www.netlib.org/hence/.
5. A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proc. Supercomputing '91*, pages 435–444. IEEE Computer Society Press, 1991.
6. G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. Fifth Int. Symp. on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.
7. S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. Anacleto: A template-based P^3L compiler. In *Proc. 7th Parallel Computing Workshop (PCW'97)*. Australian National University, 1997.
8. C. Clèmençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie. Annai: An integrated parallel programming environment for multicomputers. In A. Zaky and T. Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, chapter 2, pages 33–59. Kluwer, 1996.
9. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.
10. J. Darlington, Y. ke Guo, H. W. To, and J. Yang. Skeletons for structured parallel composition. In *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 19–28. ACM Press, 1995.
11. S. Gorlatch and C. Lengauer. (De)Compositions for parallel scan and reduction. In *Proc. 3rd Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 23–32. IEEE Computer Society Press, 1998.
12. S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim, editor, *Workshops at IPPS'99*, Lecture Notes in Computer Science, 1999. To appear.
13. S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In M. Atallah, editor, *Proc. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*. IEEE Computer Society Press, 1999. To appear.
14. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation Series. MIT Press, 1994.
15. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.
16. C. Wedler and C. Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, 1998.