A Fusion-Embedded Skeleton Library

Kiminori Matsuzaki[1], Kazuhiko Kakehi[1], Hideya Iwasaki[2],
Zhenjiang Hu[1,3], and Yoshiki Akashi[2]

[1] Graduate School of Information Science and Technology,
University of Tokyo
{Kiminori_Matsuzaki,kaz,hu}@mist.i.u-tokyo.ac.jp
[2] Department of Computer Science,
The University of Electro-Communications
iwasaki@cs.uec.ac.jp, yoshiki@ipl.cs.uec.ac.jp
[3] PRESTO21, Japan Science and Technology Agency.

Abstract. This paper addresses a new framework for designing and implementing skeleton libraries, in which each skeleton should not only be efficiently implemented as is usually done, but also be equipped with a structured interface to combine it efficiently with other skeletons. We illustrate our idea with a new skeleton library for parallel programming in C++. It is simple and efficient to use just like other C++ libraries. A distinctive feature of the library is its modularity: Our optimization framework treats newly defined skeletons equally to existing ones if the interface is given. Our current experiments are encouraging, indicating that this approach is promising both theoretically and in practice.

Keywords: Skeletal Parallel Programming, Optimization, Program Transformation, Fusion Transformation, Bird-Meertens Formalism.

1 Introduction

The increasing popularity of parallel programming environments, such as PC clusters, calls for a simple model and methodology that can assist programmers, including those who have little knowledge of parallel architecture and parallel algorithms, to develop efficient and correct parallel programs to solve various kinds of problems. Skeletal parallel programming, first proposed by Cole [1] and well-documented in [2], is such a methodology for building parallel programs in terms of a set of useful ready-made components (parallel primitives) called *skeletons*. Skeletons are generic and recurring patterns of parallel processing, whose parallel implementations are hidden from the programmers. These skeletons cannot only be efficiently implemented on various parallel architectures, but also be suitable as the target for systematic development by human [3–6].

The importance of equipping existing popular languages (like C or C++) with a well-structured skeleton library has been recognized [7–11], with which one can write parallel programs as sequential ones that call the library functions. However, the skeleton programs are slow in comparison with those directly coded in MPI, and this is an issue that prevents the skeletal parallel programming approach from being widely used. In fact, the simplicity of each skeleton often gives rise to a complicated combination of skeletons in a program, introducing a lot of data communication between them. Although individual skeletons can be efficiently implemented, their combination is inefficient unless the data communication between them can be eliminated.

There have been many attempts to apply transformation to optimizing combinations of skeletons [12, 13], where a set of transformation rules is defined, and an automatic or semi-automatic rewriting system is designed to apply these rules. There have been two problems with this approach. First, one would need a large set of rules to account for all possibilities of their combinations. Second, introducing a new skeleton would lead to large changes or extensions to the existing rule set. Lack of modularity causes these problems.

This paper proposes a new framework for designing of skeleton libraries that guarantees efficient combinations of skeletons, extending the theory developed in [14] in practice. Our idea was to associate each skeleton not only with an efficient parallel implementation but also with an interface for *efficient combination* with other skeletons. This interface contains information on how the skeleton consumes and produces its data. This idea is not new in the functional community, where we have seen the success of *shortcut deforestation (fusion)* [15] in optimizing sequential programs in Haskell compilers. However, as far as we know, we are the first to introduce this idea to the design of parallel skeleton libraries.

We designed and implemented a new skeleton library for skeletal parallel programming in C++. Our skeleton library has the following new features.

– *Single Optimization Rule*: Basically, we need just a single rule (Section 3) to optimize combinations of skeletons in the library, thanks to their structured interface. This is in sharp contrast to other transformation approaches [12, 13], where a large set of rules needs to be prepared. Furthermore, our rule can be applied to skeletal parallel programs in any way required, guaranteeing the same result and termination.
– *Modularity*: Our library allows new skeletons to be introduced without any change to the existing optimization framework, and ensures their efficient combination with existing skeletons in the library. This remedies the situation where transformation rules must take combinations of the skeletons with existing ones into account.
– *Simplicity*: From the programmers' point of view, as our library does not introduce any new syntax, a programmer who knows C++ should have no trouble in using it. We are able to construct a structured interface for the skeletons as well as apply a general optimization rule concisely and quickly (Section 4) with the help of the reflection mechanism provided with OpenC++ [16]. We found it very useful to use meta programming in implementing the transformation, which, we believe, is worth greater recognition in the skeleton community.

Our experiments in Section 4 demonstrate how promising our approach is.

In the rest of this paper, we will explain our idea based on the BMF data parallel programming model [17, 3], which provides us with a concise way of describing and manipulating parallel programs. After briefly reviewing the notations and basic concepts of BMF and skeletal parallel programming in Section 2, we show how to structure skeletons by standardizing their consumption and production of data. We then give our general rule for optimizing the combinations of skeletons in Section 3. We highlight our implementation of the library together with experiments in Section 4, and finally conclude the paper in Section 5.

2 BMF and Parallel Computation

We will now address our idea on the BMF data parallel programming model [17, 3]. Those familiar with the functional language Haskell [18] should have few problem in understanding the programs in this paper. From the notational viewpoint, the main difference is that we have used more symbols or special parentheses to shorten the expressions so that expressions can be manipulated more concisely.

2.1 Functions

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. A function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, not $f\,(a \oplus b)$. *Function composition* is denoted by a period. By definition, we have $(f\,.\,g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by id.

Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary or binary functions by $a \oplus b = (a\,\oplus)\,b = (\oplus\,b)\,a = (\oplus)\,a\,b$.

2.2 Parallel Data Structure: Join Lists

Join lists (or *append lists*) are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (and $[\cdot]$ for the function taking a to $[a]$), and $x +\!\!+\, y$ for the concatenation (join) of two lists x and y. Concatenation is associative, and $[\,]$ is its unit. For example, $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$.

We also write $a : x$ for $[a] +\!\!+\, x$. If a list is constructed with constructor $[\,]$ and $:$, we call it a *cons list*.

2.3 Parallel Skeletons: map, reduce, and scan

It has been shown [3] that BMF [17] is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher-order functions that can be regarded as parallel skeletons suitable for parallel implementation. Important higher-order functions are *map*, *reduce*, and *scan*.

Map is a skeleton that applies a function to every element in a list. It is written as infix $*$. Informally, we have

$$k * [x_1, x_2, \ldots, x_n] = [k\,x_1, k\,x_2, \ldots, k\,x_n].$$

Reduce is a skeleton that collapses a list into a single value by repeatedly applying a certain associative binary operator. It is written as infix $/$. Informally, for associative binary operator $\oplus$ and initial value e, we have

$$\oplus/_e\ [x_1, x_2, \ldots, x_n] = e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Scan is a skeleton that accumulates all intermediate results for computation of reduce. Informally, for associative binary operator $\oplus$ and initial value e, we have

$$\oplus\!\!\not\!\!\!/\!\!\!/_e\ [x_1, x_2, \ldots, x_n]\ =\ [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n].$$

Note that this definition is slightly different from that in [17]; the e there is assumed to be the unit of $\oplus$. In fact, efficient implementation of the scan skeleton does not need this restriction.

2.4 An Example: Computing Variance

Consider, given the sequence $as = [a_1, a_2, \ldots, a_n]$, computing its variance var by $var = \Sigma_{i=1}^{n}(a_i - ave)^2/n$, where $ave = \Sigma_{i=1}^{n} a_i/n$, as a simple running example. This computation can be described using skeletons in BMF as follows.

$$\begin{array}{lll} var\ as\ n = sqSum/n & & \\ \quad \textbf{where}\ ave & = (+/_0\ as)/n \\ \quad\quad sqSum & = +/_0\ (square * ((-ave) * as)) \\ \quad\quad square\ x & = x \times x \end{array}$$

This BMF description is the best way of explaining our idea. In fact, to be processed by our system, it should be written in C++ using our skeleton library as follows. Note that the functions `add`, `sub`, and `sq` are defined by the user, and `map1` is a variation of the map skeleton designed to treat sectioning notation; `map1( sub, ave )` corresponds to $(-ave)*$.

```
double variance( vector< double > *as, int size ) {
  double sum, ave, sq_sum;
  vector< double > *subs, *sqs;
  sum = as->reduce( add, 0.0 ); ave = sum / size;
  subs = as->map1( sub, ave ); sqs = subs->map( sq );
  sq_sum = sqs->reduce( add, 0.0 );
  return = sq_sum / size;
}
```

3 Shortcut Fusion on Skeleton Programs

To fuse the composition of skeletons into one to eliminate unnecessary intermediate data structures passed between skeletons, one may develop rules to do algebraic transformations on skeletal parallel programs like the authors in [12, 13]. Unfortunately, this would require a huge set of rules to take all possible combinations of skeletal functions into account. In this paper, we borrow the idea of shortcut deforestation [15], which optimizes sequential programs, and simplifi es the entire set into just a single rule. The idea is to structure each skeleton with an interface that characterizes how it consumes and produces the parallel data structure, namely join lists.

3.1 Structuring Skeletons

To manipulate skeletal parallel programs, we structured skeletons in terms of the following three functions: acc, cataJ and buildJ.

Definition 1 (acc). Let g, p, q be functions, and $\oplus$ and $\otimes$ be associative operators. The skeleton acc, for which we write $[\![g, (p, \oplus), (q, \otimes)]\!]$, is defi ned by

$$\begin{aligned} [\![g, (p, \oplus), (q, \otimes)]\!]\ [\,]\ e &= g\ e \\ [\![g, (p, \oplus), (q, \otimes)]\!]\ (a : x)\ e &= p\ (a, e)\ \oplus\ [\![g, (p, \oplus), (q, \otimes)]\!]\ x\ (e \otimes q\ a)\ . \end{aligned}$$

The function acc has an accumulation parameter, e. We defi ne cataJ as a special case of acc, where the accumulation parameter is not used.

Definition 2 (cataJ). Given are function p and associative operator $\oplus$ with identity e. The skeleton cataJ, for which we write $(\![\oplus, p, e]\!)$, is defi ned by

$$\begin{aligned} (\![\oplus, p, e]\!)\ [\,] &= e \\ (\![\oplus, p, e]\!)\ (a : x) &= p\ a\ \oplus\ (\![\oplus, p, e]\!)\ x\ . \end{aligned}$$

The last function, buildJ, is to standardize the production of join-lists with implicit parallelism.

$$\mathsf{buildJ}\ gen = gen\ (+\!\!+)\ [\cdot]\ [\,]$$

We can now express our skeletons in terms of the above three functions.

Definition 3 (Skeletons in Structured Form).

$$\begin{aligned} f* &= \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\![c, f.s, e]\!)) \\ \oplus/_e &= (\![\oplus, id, e]\!) \\ \oplus\!\!\not\!/\!\!\!\!/_e x &= \mathsf{buildJ}\ (\lambda c\ s\ e.\ [\![s, (\lambda(a, e).\ s\ e, c), (id, \oplus)]\!])\ x\ e \end{aligned}$$

3.2 Shortcut Fusion Rule

Following the thought in [15], we may defi ne our shortcut fusion for join lists as follows.

Definition 4 (CataJ-BuildJ Rule).

$$(\![c, s, e]\!)\ .\ \mathsf{buildJ}\ \mathit{gen} = \mathit{gen}\ c\ s\ e$$

An example of applying this rule shows that reduce after map can be fused into a single cataJ.

$$
\begin{aligned}
\oplus/ \,.\, f* = \quad & \{ \text{map, reduce} \} \\
& (\!|\oplus, \mathit{id}, \iota_\oplus|\!) \,.\, \mathsf{buildJ}\ (\lambda\ c\ s\ e\ .\ (\!|c, s.f, e|\!)) \\
= \quad & \{ \text{CataJ-BuildJ} \} \\
& ((\lambda\ c\ s\ e\ .\ (\!|c, s.f, e|\!))\ \oplus\ \mathit{id}\ \iota_\oplus) \\
= \quad & \{ \text{lambda application} \} \\
& (\!|\oplus, f, \iota_\oplus|\!)\ .
\end{aligned}
$$

This CataJ-BuildJ rule, however, is not suffi cient for some cases. Consider where we want to fuse $f*\ .\ g*$.

$$f*\ .\ g* = \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s.f, e|\!))\ .\ \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s.g, e|\!))$$

We are blocked here, since the left cataJ is enclosed in buildJ. To proceed with the transformation, we may have to unfold the left buildJ, fi nally to have $(\!|{+\!\!+}, [.].f.g, [\,]|\!)$.

This result is unsatisfactory; this is not a form of the producer! Once transformed like this, further fusion by other consuming functions cannot take place. The trouble occurs due to uninvertible unfolding of buildJ, which we should avoid. The following rule does the trick in eliminating unnecessary unfolding of buildJ.

Definition 5 (BuildJ(CataJ-BuildJ) Rule).

$$\mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|\phi_1, \phi_2, \phi_3|\!))\ .\ \mathsf{buildJ}\ \mathit{gen} = \mathsf{buildJ}\ (\lambda c\ s\ e.\ \mathit{gen}\ \phi_1\ \phi_2\ \phi_3)$$

This rule enables us to have

$$
\begin{aligned}
& f*\ .\ g* \\
= \quad & \{ \text{Map, BuildJ(CataJ-BuildJ), lambda application} \} \\
& \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s.f.g, e|\!)),
\end{aligned}
$$

which is exactly the structured form of $(f.g)*$.

Finally, we generalize the above rule to the following most generic fusion rule (for acc).

Definition 6 (BuildJ(Acc-BuildJ) Rule).

$$
\begin{aligned}
& \mathsf{buildJ}(\lambda c\ s\ e.\ [\![g, (\oplus, p), (q, \otimes)]\!])\ (\mathsf{buildJ}\ \mathit{gen}\ x)\ e \\
& \quad = \mathit{fst}\ (\mathsf{buildJ}\ (\lambda c\ s\ e.\ \mathit{gen}\ (\odot)\ f\ d)\ x\ e) \\
& \text{where} \\
& \quad (u \odot v)\ e = \mathbf{let}\ (r_1, s_1, t_1) = u\ e \\
& \qquad\qquad\qquad\quad\ (r_2, s_2, t_2) = v\ (e \otimes t_1) \\
& \qquad\qquad\qquad \mathbf{in}\ (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2) \\
& \quad f\ a\ e = (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a)) \\
& \quad d\ e = (g\ e, \_, \_)
\end{aligned}
$$

Here, *fst* returns the fi rst element of a pair, and _ denotes a “don’t care” value.

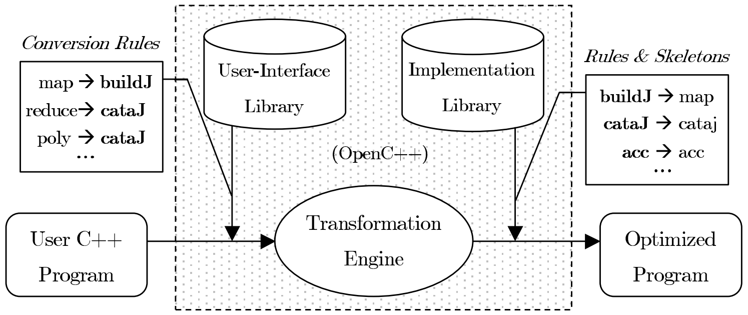

Fig. 1. Overview of our system

3.3 Modularity of Structured Forms

The shortcut fusion rule guarantees the fusibility of the composition of functions defined using acc (or cataJ in special) and buildJ. The implication of this is: User-defined skeletons are also the target of optimization once their structured form is known.

For example, we may want to introduce a new skeleton to capture a general polynomial computation pattern, which has many interesting applications including solving the maximum sum problems [17]. This new skeleton is parameterized by two associative operators $\oplus$ and $\otimes$ whose units are $\iota_\oplus$ and $\iota_\otimes$.

$$\textsf{poly}\ (\oplus)\ (\otimes)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_1 \otimes x_2) \oplus (x_1 \otimes x_2 \otimes \cdots \otimes x_n)$$

This new skeleton can be structured as follows, which can consume a join list produced by any other skeletons.

$$\begin{aligned}\textsf{poly}\ (\oplus)\ (\otimes) &= \textit{fst}\ .\ (\![\odot,\ (\lambda\ a\ .\ (a,a)),\ (\iota_\oplus, \iota_\otimes)]\!) \\ &\quad \text{where } (a_l, c_l) \odot (a_r, c_r) = (a_l \oplus c_l \otimes a_r,\ c_l \otimes c_r)\end{aligned}$$

Note that the poly skeleton produces a single value, which could not be consumed by other skeletons. Therefore, we do not need to structure the output using buildJ.

4 Implementation of Transformation System

We implemented a prototype system with OpenC++, which transforms the skeletal parallel programs written in C++ with our skeleton system [10]. Fig. 1 overviews our system, which consists of three parts: (1) the user-interface library, (2) the generic transformation engine, and (3) the implementation library. Taking a C++ program, our transformation system first converts the skeletons in the program into structured form by applying rules given as meta-programs. The generic transformation engine manipulates and fuses the converted program with the shortcut fusion rules in Section 3. Finally, our system links the optimized program with efficiently implemented skeletons in our library.

4.1 Skeleton Interfaces

In OpenC++, the program text is accessible at the meta level in the form of a parse tree, represented as a nested list of logical tokens. A part of the C++ program to compute the variance in Section 2 is converted into the following parse tree.

```
[[sum = [[as -> reduce] ( [add , 0.0] )]] ;]
[[ave = [sum / size]] ;]
[[subs = [[as -> map1] ( [sub , ave] )]] ;]
[[sqs = [[subs -> map] ( [sq] )]] ;]
[[sq_sum = [[sqs -> reduce] ( [add , 0.0] )]] ;]
```

We defi ne the rules to convert user skeletons to structured form and vice versa in OpenC++. For example, a meta program that converts a map skeleton into buildJ may be implemented as follows.

```
Ptree* map_to_buildJ( Ptree *sentence )
{
  Ptree *dst, *src, *function;
  if (Ptree::Match( sentence, "[[%? = [[%? -> map] ( %? )]] ;]",
                    &dst, &src, &function) ) {
    return make_buildJ( dst, src, Ptree::List( var_c ),
                        Ptree::List( var_s, function ), Ptree::List( var_e ));
  }
  ...
```

The reflection mechanism in OpenC++ enables pattern matching and function composition to be easily implemented. Thus, we can easily convert skeletons to their structured forms, e.g. for the poly skeleton, we can obtain the arguments $\oplus$, $\otimes$, $\iota_\oplus$, and $\iota_\otimes$ by pattern matching and derive structured forms after generating the new functions $\odot$ and $\lambda\, a\, .\, (a, a)$.

Using conversion with our user-interface library, the last three lines in the parse tree above are converted into the following structured forms.

```
['buildJ' subs as [[var_c] [var_s [sub ave]] [var_e]] ;]
['buildJ' sqs subs [[var_c] [var_s [sq]] [var_e]] ;]
['cataJ' sq_sum sqs [[add] [func_id] [0.0]] ;]
```

4.2 Generic Transformation Engine

Our system implements the fusion rule in Section 3, and it repeatedly applies the rule on structured forms. We restricted the elements in structured forms so that they were represented as a composition of functions. This simplifi ed the application of the fusion rule so that just the occurrences of a bound variable to the corresponding argument had to be replaced. Note that such a restriction is insignifi cant since reflection can take care of it.

Our generic transformation engine applies the **CataJ-BuildJ** rule twice on the structured forms above in our running example, and optimizes it into a single cataJ form as follows.

```
['cataJ' sq_sum as [[add] [func_id [sq] [sub ave]] [0.0]] ;]
```

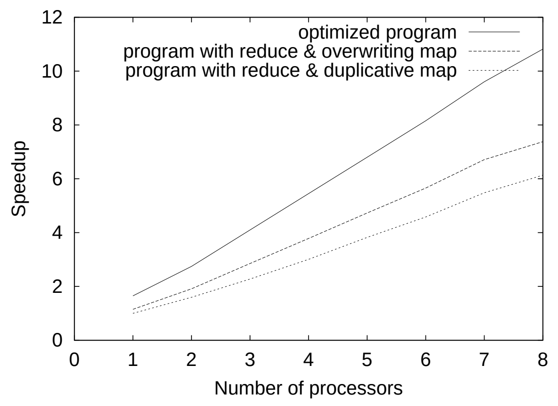

Fig. 2. Experimental results

4.3 Experimental Results

The first program in Section 2 is inefficient because small functions are called one by one and unnecessary intermediate data are passed between skeletons. Our system automatically transforms the program so that it is efficient by composing skeletons.

To see how efficient the generated optimized program is, we compared it with the following two programs: (1) the original program using a map skeleton that produces new data, (2) another program using a map skeleton that overwrites the input data. In the second program, the individual skeletons are optimized so that they do not generate unnecessary data. We implemented these programs with our skeleton library in C++ and MPI, and did our experiments on a cluster of four Pentium 4 Xeon 2.0-GHz dual-processor PCs with 1 GB of memory, connected through a Gigabit Ethernet. The OS was FreeBSD 4.8 and we used gcc 2.95 for the compiler.

Fig. 2 plots the results of speedups to the original program with one processor for an array of 1,000,000 elements. The computation time for the original program with one processor is 1.67 (sec) and the computation times for (1), (2), and the optimized one with eight processors are 0.243, 0.197, and 0.138 (sec), respectively. As a natural consequence of using the skeletons, all programs demonstrated outstanding scalability. Comparison with (2) proves the success of our framework: The effect of fusion far exceeds individual refinements on each skeleton.

5 Conclusions

We proposed a new approach to the design and implementation of skeleton libraries. We implemented a parallel skeleton library in C++, which not only guaranteed each skeleton was efficiently implemented, but also efficiently combined the skeletons such that data communication between them could be eliminated. In contrast to popular approaches where the design of skeleton libraries has mainly focused on efficiently implementing single skeletons with little consideration to how combinations of skeletons are optimized, our approach unifi es the two phases by structuring skeletons with an additional interface. This new approach is not only theoretically interesting, but also quite promising in practice. As we demonstrated, our library is easy to use, simple to implement, and suitable for extension.

We are still in the early stages of producing a really useful library supporting parallel programming in C++. In terms of theory, we have not yet taken functions like zip into account that traverse multiple data structures simultaneously; we are also interested in generalizing the approach from join lists to other parallel data structures such as matrices or trees. In terms of practice, our current implementation, whose main purpose is to test our idea, is expected to be improved through further analysis so that the generic optimization rule can be applied to more applications of skeletal parallel programming.

References

1. Cole, M.: Algorithmic skeletons : A structured approach to the management of parallel computation. Research Monographs in Parallel and Distributed Computing, Pitman, London (1989)
2. Rabhi, F., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer Verlag (2002)
3. Skillicorn, D.B.: The Bird-Meertens Formalism as a Parallel Model. In: NATO ARW “Software for Parallel Computation”. (1992)
4. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Annual European Conference on Parallel Processing, LNCS 1124, LIP, ENS Lyon, France, Springer-Verlag (1996) 401–408
5. Hu, Z., Iwasaki, H., Takeichi, M.: Formal derivation of efficient parallel programs by construction of list homomorphisms. ACM Transactions on Programming Languages and Systems **19** (1997) 444–461
6. Hu, Z., Takeichi, M., Chin, W.: Parallelization in calculational forms. In: 25th ACM Symposium on Principles of Programming Languages, San Diego, California, USA (1998) 316–328
7. Kuchen, H.: A skeleton library. In: EuroPar’02, LNCS, Springer-Verlag (2002)
8. Cole, M.: eSkel home page. `http://homepages.inf.ed.ac.uk/mic/eSkel/` (2002)
9. Danelutto, M., Stigliani, M.: SKElib: parallel programming with skeletons in c. In: EuroPar’00, LNCS 1900, Springer-Verlag (2000) 1175–1184
10. Adachi, S., Iwasaki, H., Hu, Z.: Diff: A powerful parallel skeleton. In: The 2000 International Conference on Parallel and Distributed Processing Techniques and Application. Volume 4., Las Vegas, CSREA Press (2000) 525–527
11. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A structured high level programming language and its structured support. Concurrency Practice and Experience **7** (1995) 225–255
12. Gorlatch, S., Pelagatti, S.: A transformational framework for skeletal programs: Overview and case study. In Rohlim, J., et al., eds.: Parallel and Distributed Processing. IPPS/SPDP’99 Workshops Proceedings. Lecture Notes in Computer Science 1586 (1999) 123–137
13. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms and Applications **16** (2001) 87–122
14. Hu, Z., Iwasaki, H., Takeichi, M.: An accumulative parallel skeleton for all. In: 11st European Symposium on Programming (ESOP 2002), Grenoble, France, Springer Verlag, LNCS 2305 (2002) 83–97

15. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Proc. Conference on Functional Programming Languages and Computer Architecture, Copenhagen (1993) 223–232
16. Chiba, S.: A metaobject protocol for C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95). SIGPLAN Notices 30(10), Austin, Texas, USA (1995) 285–299
17. Bird, R.S.: An introduction to the theory of lists. In Broy, M., ed.: Logic of Programming and Calculi of Discrete Design. Volume 36 of NATO ASI Series F., Springer-Verlag (1987) 5–42
18. Bird, R.: Introduction to Functional Programming using Haskell. Prentice Hall (1998)