A Transformational Framework for Skeletal Programs: Overview and Case Study

Sergei Gorlatch[1] and Susanna Pelagatti[2]

[1] Universität Passau, D-94030 Passau, Germany
[2] Universitá di Pisa, Corso Italia 40, I-56125 Pisa, Italy

Abstract. A structured approach to parallel programming allows to construct applications by composing *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 and the lack of a special methodology for algorithm design and performance prediction. In the paper, we take a first step toward such a methodology, by developing a general transformational framework named FAN, and integrating it with an existing skeleton-based programming system, P3L. The framework includes a new functional abstract notation for expressing parallel algorithms, a set of semantics-preserving transformation rules, and analytical estimates of the rules' impact on the program performance. The use of FAN is demonstrated on a case study: we design a parallel algorithm for the maximum segment sum problem, translate the algorithm in P3L, and experiment with the target C+MPI code on a Fujitsu AP1000 parallel machine.

1 Introduction

Current difficulties in the low-level parallel programming, using e.g., MPI [17], can be addressed by developing higher-level programming models together with convenient programming environments.

One of popular higher-level approaches is based on so-called *skeletons*, which can be viewed as recurring algorithmic and communication patterns, expressed in a rigorous way [19]. Representatives of the skeleton-based systems are the P3L system at the University of Pisa [3], its commercial analog SKIECL at QSW Ltd., 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 and used to construct a parallel program. A skeletal program is then (semi)automatically translated into some target language, e.g., C+MPI, using prepackaged parallel implementations of particular skeletons. The abstraction from communication and other details makes skeletal programs considerably better structured and less error-prone than their low-level counterparts.

The experience with skeletal programming, both in academia and industry, has brought also a new challenge. Since abstract constructs are usually very "coarse-grained", a slight change in one of skeletons, and especially in the way they are composed with each other, can lead to dramatic changes in the target program performance. Analyze the possible changes, predicting their influence on performance and optimizing the high-level program appropriately, has proven to be a non-trivial task for the user. In the long term, the challenge of designing efficient high-level programs should be addressed by providing a *methodology of skeleton-based programming* which would include methods and tools for choosing suitable skeletons, composing them in a program, estimating its expected performance, and making changes in design if necessary.

In this paper, we take a first step towards such a methodology, by presenting a general transformational framework for designing parallel algorithms on a high level of abstraction. The aim of the framework is to support the user in producing a high-quality skeletal program. We integrate this framework with the current P3L implementation, and assess the whole system in a case study.

The contributions and structure of the paper are as follows:

- Section 2 gives an overview of the framework, and puts it into the context of the P3L programming system, which we use as a testbed in our work.
- In Section 3, we present a new notation for parallel algorithms, called FAN (Functional Abstract Notation), which facilitates semantics-preserving transformations of algorithms in the design process.
- Section 4 describes a case study: designing a parallel algorithm for the maximum segment sum problem (a programming pearl [4]). We start with an intuitively clear FAN-formulation of the algorithm, and proceed, by applying FAN-transformations, towards an efficient but intricate parallel version, which hardly could be written by the user from scratch.
- In Section 5, we describe our cost model, which helps to decide under which conditions a particular transformation rule improves target performance.
- Section 6 revisits our case study: from the FAN-algorithm for the maximum segment sum, a P3L skeletal program is generated. This program is then automatically translated by the P3L compiler into a C+MPI code. We report some experimental result with thus obtained target program on a Fujitsu AP1000 parallel machine.

In conclusion, we compare to related work and outline our future efforts.

2 System Structure Overview

This section takes the P3L system [3] as a representative of skeleton-based systems, and outlines how it is augmented by the FAN transformational framework. The overall structure of the resulting FAN-P3L system is presented in Figure 1.

The figure shows how the user, depicted on the left side, communicates with the high-level programming system, which is partitioned using horizontal bold, solid lines into three parts, top-down: the transformational framework (presented in this paper), the (current) P3L system, and the target machines. Solid arrows show the connections between the parts of the system, dashed arrows depict the user interaction with the system, with bold, dashed arrows for the new interactions added by the transformational framework.

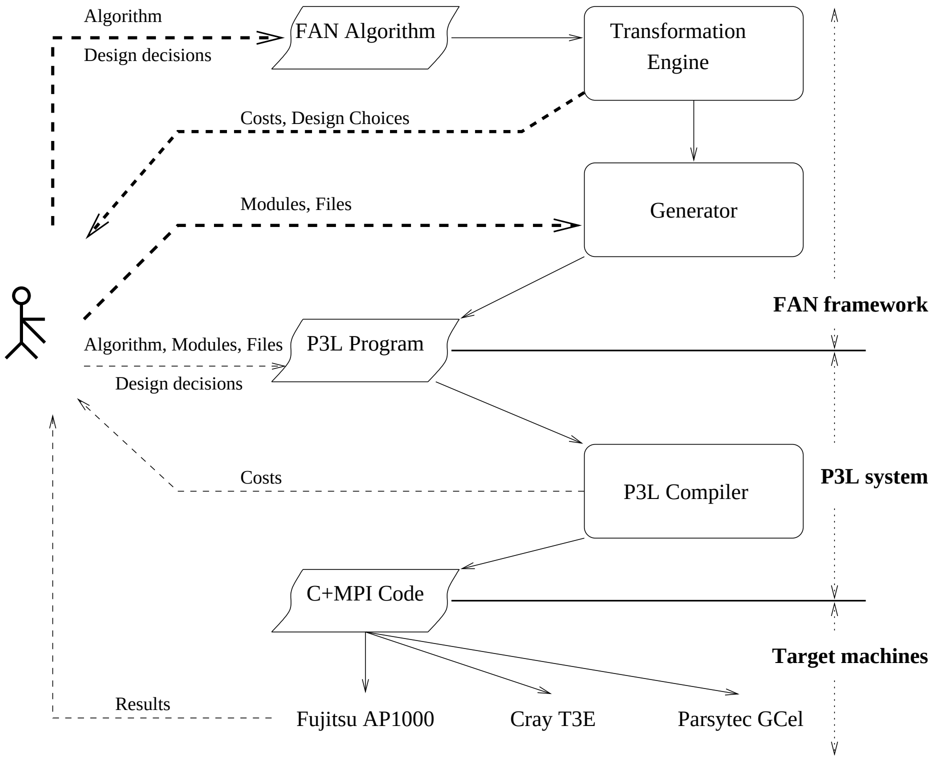


Fig. 1. Skeleton-based system P3L, augmented by the FAN framework

In the current 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 skeleton-based algorithm itself and also supply all necessary sequential modules, input and output files. This makes the corresponding arrow in the figure look quite overloaded. The program is optimized and translated by the P3L compiler, which provides the user with preliminary cost estimates for the program. On the base of these estimates, the user has to make design decisions which may require changing the algorithm, rewriting modules and reorganizing the files. If the user is satisfied with costs, the C+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). The translation/estimation/rewriting process, which may have to be repeated several times, represents the challenge mentioned in the introduction and addressed in this paper.

The transformational FAN framework, shown in the upper part of Figure 1, offers the user an additional support in designing a skeletal program. The design process now 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 for the user, with cost estimates for the alternatives. After possibly several iterations with different versions of the algorithm, the user may decide to generate a P3L program, for which the descriptors of files and modules becomes necessary. The generated P3L program is given to the current P3L system, which proceeds as described in the previous paragraphs.

3 FAN: Functional Abstract Notation

In this section, we briefly introduce an abstract notation, FAN, for specifying and transforming parallel programs.

The data space in a FAN program is viewed globally and there is no notion of processor at this abstract level. FAN allows the user to concentrate on the parallel structure of the program, abstracting from the syntactic details of a particular skeleton environment and from the application-specific sequential parts of the program.

3.1 Support for Transformations

The current P3L system implements several basic transformations and optimizations of skeleton programs:

- Data redistributions: if collecting data in one place becomes redundant due to a subsequent data scattering, the compiler eliminates the unnecessary data movements. Similarly, no rearrangement is done when data are produced and consumed by two consecutive skeletons requiring the same data distribution.
- Skeleton compositions: several important sequences of skeletons are optimized in order to redistribute the minimal amount of data among two consecutive local computations.
- Computation fusion: local pieces of computations, corresponding to different skeletons in a program, are composed within the sequential processor programs, which are subsequently optimized by native compilers, e.g., C.

The motivation of FAN is to allow much more powerful transformations, and to make them controllable by the program designer. Programming with skeletons deserves scrutiny and requires a method. The FAN framework supports such a method, based on a set of transformation rules, augmented with a cost calculus.

The FAN framework is based on the paradigm of functional programming. In particular, all algorithmic skeletons are captured as higher-order functions on data structures. Hence, program transformations can be proved as semantic equalities in the corresponding functional formalism. Functional framework offers effective support in prototyping and analysis, and does not limit the choice of the final implementation.

3.2 Notation Overview and Example

A FAN program contains a *header*, specifying the program name, a list of variables (arrays, tuples and scalars) which are taken as input and produced as output, and a *body*, defining a sequence of skeleton applications.

The declaration of a program header reads as follows:

progname (in *inlist* , out *outlist*)

where *inlist* describes the input variables on which the program works and *outlist* describes the variables computed by the program. Both input and output lists contain variables along with their types such as

$$v_1 : T_1, \ldots v_n : T_n$$

where $v_i : T_i$ means that variable v_i has type T_i. Data types allowed in variable declarations are scalars (scalar), vectors (arr[n]) and multidimensional arrays (arr[n][m]). There are also tuples which are arrays of a fixed dimension. A program body is a list of skeleton calls expressed in functional notation using single assignment variables. FAN skeletons are second order functions defined on arrays and scalars.

As an example, let us consider the following algorithm which computes the inner product of two vectors a and b of length n:

```
alg-inprod (in a,b : arr[n], out      c : scalar)
  t = map (*) (arrange (a,b));
  c = reduce (+) t;
```

The input of program alg-inprod are two arrays of length n, and the output is a scalar, c. The body describes how the parallel computation takes place. First, the elements of a and b are paired using the arrange skeleton. In the example, (a,b) arranges a and b in a vector s of length n, in which each element $s[i]$ is a pair $(a[i], b[i])$. Then, the map skeleton multiplies all the pairs in s and stores the results in array t, so that $t[i] = a[i] * b[i]$. Finally, elements of t are summed up using the reduce skeleton.

3.3 Data- and Task-parallel Skeletons

There are two kinds of skeletons in FAN: data- and task-parallel skeletons. Task-parallel skeletons, including farm and pipeline, exploit parallelism among independent parts of an application. Independent parts typically exploit data parallelism internally and are more coarse grain [10, 19]. Data-parallel skeletons express parallel structures in which the same operation has to be applied to a number of related data. Such skeletons in FAN include data arrangements expressed by arrange, and computations expressed by map, reduce, scanL, and scanR. In the rest of the paper we will concentrate on data parallel skeletons.

Data-parallel skeleton arrange is used to express data manipulations (replication, alignment etc.) which are needed to put data in place for subsequent data parallel computations. Different kinds of data arrangement are specified by providing particular predefined functions as parameters of arrange skeleton.

Some legal arrangements which will be used in the rest of the paper are shown in Table 3.3, where a and b are vectors of length n, c is a vector of pairs of length n and s is a scalar. (a,b) builds a vector of pairs as described before and proj [1] returns the first element of a particular pair:

arrangement	*description*	*possible result*
c = (a,b);	creates tuples (pairs)	$c[i] = (a[i], b[i])$, c :arr[n][2]
s = proj [1] c;	projects tuples/arrays	$s = (a[1], b[1])$, s :arr[2]

Table 1. Some legal arrangement patterns. a and b are vectors of length n.

The simplest computational skeleton, map, applies a function, f, to all the elements of an array, arranging results in an array with the same shape, for instance, if $a = [a[1], a[2], \ldots, a[n]]$ is a vector of length n then

$$\mathsf{map}\ f\ [\,a[1], a[2], \ldots, a[n]] \;=\; [f\,a[1], f\,a[2],\,\ldots\,,\ f\,a[n]\,]$$

Finally, skeletons reduce, scanL and scanR provide the usual data-parallel reduce and scan operations on vectors:

$$\begin{aligned}
\mathsf{reduce}\ (\oplus)\ a &= a[0] \oplus \ldots \oplus a[n-1] \\
\mathsf{scanL}\ (\oplus)\ a &= [\,a[0],\ a[0] \oplus a[1],\ \ldots,\ a[0] \oplus \ldots \oplus a[n-1]\,] \\
\mathsf{scanR}\ (\oplus)\ a &= [\,a[0] \oplus \ldots \oplus a[n-1],\ \ldots,\ a[n-2] \oplus a[n-1],\ a[n-1]\,]
\end{aligned}$$

The base operator, $\oplus$, plays an important role in the reduction and scan skeletons: if the operator is associative, the corresponding skeleton can be parallelized, with parallel time logarithmic in the size of the argument array.

4 Case Study: Maximum Segment Sum

In this section, we demonstrate the use of the FAN notation and the transformation rules in the program design process on a particular case study. We present only three transformations which are used in the example; a rich set of transformations, with details about their performance, is developed in a more general context in paper [16] being presented at IPPS/SPDP'99.

We consider the famous *Maximum Segment Sum* (MSS) problem – a *programming pearl* [4], studied by many authors [5, 8, 20, 22, 23]. Given a one-dimensional array of integers, function mss finds a contiguous array segment, whose members have the largest sum among all such segments, and returns this sum. For example:

$$mss\,[\,2, -4, 2, -1, 6, -3\,] \;=\; 7$$

where the result is contributed by the segment $[2, -1, 6]$.

4.1 Intuitive algorithm.

The first version of the program for computing mss could be quite simple:

alg-mss1 (in x : arr[n], out r : scalar);
  $s =$ scanL $(op1)$ x;
  $r =$ reduce (max) s;

where operator op1 is defined as follows:

$$a_1 \ \mathsf{op1} \ a_2 \ = \ max\,((a_1 + a_2), a_2) \qquad (1)$$

Algorithm alg-mss1 takes array x as input, and proceeds intuitively as follows:

- The first stage, scanL $(op1)$, produces array s of type arr[n], whose i-th element is the maximum sum of segments of x ending in position i.
- The second stage, reduce (max), computes the maximum element of array s, thus yielding the resulting value, r, so that $r \ = \ mss\,x$.

Our first algorithm expressed by alg-mss1 has several good features. First, it is intuitively clear, and its correctness with respect to the specification of the mss problem can be proved. Second, it consists of only two skeletons, scan and reduction, both being well parallelizable. However, a closer look reveals also a serious drawback: the scan skeleton in alg-mss1 cannot be parallelized since operator $op1$ defined by (1) is not associative.

4.2 Parallelizable algorithm.

In order to enable parallelization, we must construct an associative operator for the scan skeleton. An ubiquitous technique used for that is the introduction of auxiliary variables. In our case, we define the new, associative operator on pairs, so that the original operator is modeled by the first element of a pair:

$$(a_1, b_1)\, op2\, (a_2, b_2) \ = \Big(\ max\,((a_1 + b_2), a_2)\, , \ b_1 + b_2 \ \Big) \qquad (2)$$

Using this operator, we can rewrite the initial program as follows:

alg-mss2 (in x : arr[n], out r : scalar)
  $y =$ scanL $(op2)$ (arrange (x,x));
  $s =$ arrange (proj [1]) y;
  $r =$ reduce (max) s;

Note that the first two statements of alg-mss2 are semantically equivalent to the first statement of alg-mss1.

The asymptotic parallel complexity of algorithm alg-mss2 is logarithmic in the size of the input array. This follows from the logarithmic complexity of both the scan and the reduction stage with associative operators, and from the constant parallel complexity of map. Thus, theoretically the algorithm is optimal.

Algorithm alg-mss2 can be programmed directly in any parallel language which provides primitives for scans and reduction, i.e. P3L, MPI, etc. However, its performance in practice may suffer from the fact that it exploits two collective operations, each of which involves a considerable amount of interprocessor communication, with start-up time etc. Note that the algorithm presented in textbook [1] may have an even higher cost than alg-mss2, since it exploits three collective operations, two scans and one reduce.

Thus, the user's problem is not only to produce an algorithm but also to understand whether the obtained algorithm is really usable in practice, and, even more important, how it can be transformed and how the possibly different versions can be compared.

4.3 Program transformations.

The goal of our transformations is to try and reduce the number of collective operations in alg-mss2. A rich set of transformation rules for collective operations has been developed recently [15, 16]. The rule we could try for alg-mss2 is the scan-reduce fusion:

Rule SR-ARA

b = scanL $Op1$ a c = reduce $Op2$ b
b = reduce (New ($Op1$, $Op2$)) (arrange (a,a)) c = arrange (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 rule is named SR-ARA, which reflects the transformation expressed by it: "**S**can;**R**educe → **A**rrange;**R**educe;**A**rrange". We present this and other rules in a format that consists of four boxes, top-down: (1) a program fragment before transformation (the "left-hand side" of the rule), (2) the fragment after transformation (the "right-hand side"), (3) optional: the If-condition expressing when the rule is applicable, (4) optional: the definition(s) of new function(s) used by the rule. The transformation rules use parameter operators $Op1$, $Op2$, etc. Note their difference to the concrete operators $op1$, $op2$, which we have used in the programs alg-mss1, etc.: those are written with a small "o". In the SR-ARA rule, the base operation of the reduction on the right-hand side is constructed by applying (higher-order) function New to the parameter operators of the left-hand side; New is defined in the bottom box of the rule.

Trying to apply the SR-ARA rule to our algorithm alg-mss2, the transformation engine (see Figure 1) immediately finds a mismatch: there is an extra arrange statement between scan and reduction in the algorithm.

To make the SR-ARA rule applicable, we can use the following transformation:

Rule AR-RA

$c = \mathsf{reduce}\ Op\ (\mathsf{arrange}\ (\mathsf{proj}\ [1])\ b)$
$c = \mathsf{arrange}(\mathsf{proj}\ [1])\ (\mathsf{reduce}\ Op.pair\ \ b)$
$(a_1, b_1)\, Op.pair\, (a_2, b_2) \ = \ (a_1\, Op\, a_2,\ b_1\, Op\, b_2)$

In this rule, it is assumed that the input, b, is an array of tuples, and that the projection arrangement refers to a correct position in the tuple. We provide here the instance of the rule when the tuples are pairs, and the projection yields the first position. The rule is universally applicable since it has no If-part.

Note that the AR-RA transformation is applicable in both directions, with completely different consequences: from right to left, the transformation eliminates redundant computations, thereby reducing the cost of the overall composition; from left to right, it restructures the program by pushing the projection after the reduction, at the price of redundant computations. In our case study, we have the latter situation for alg-mss2: we need to push the projection, which would enable us to apply the scan-reduce fusion afterwards.

Since the left-to-right application of AR-RA makes sense only when enabling the SR-ARA transformation afterwards, it is useful to combine both rules – the one pushing the projection and the scan-reduce fusion – into one bigger rule:

Rule SAR-ARA

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

4.4 Target algorithm.

Now we are ready to apply the SAR-ARA rule to alg-mss2, with the following instantiations of the parameter operators: $Op1 \ = \ op2$ and $Op2 \ = \ max$. The applicability condition is the forward distributivity of $Op1$ over $Op2.pair$, which in case of alg-mss2 are the following operations:

$$(a_1, b_1)\, Op1\, (a_2, b_2) \ = \ (\, max\, ((a_1 + b_2), a_2)\, ,\ b_1 + b_2\,) \tag{3}$$
$$(a_1, b_1)\, Op2.pair\, (a_2, b_2) \ = \ (max(a_1, a_2), max(b_1, b_2)) \tag{4}$$

The distributivity required by the rule can be checked straightforwardly, under the obviously correct assumption that operator max is commutative.

After applying the SAR-ARA rule to program alg-mss2, we obtain the following *target program for the maximum segment sum problem*:

alg-mss3 (in x : arr[n], out r : scalar);
  s = reduce $(New(op2, max.pair))$ (arrange (arrange (x,x), arrange (x,x))) ;
  r = arrange (proj [1]) (arrange proj [1] x);

Target program alg-mss3 contains only one computational skeleton, reduction, with an associative base operator. Such a reduction is easily parallelizable. Thus, we arrive at a better solution than both the intuitive and the first parallelizable version. Note that data arrangements and the reduction are quite intricate: $max.pair$ is constructed according to (4) for $Op2 = max$, and $op2$ is defined by (2). It is rather unlikely that the programmer could write such a program without support from the system.

5 Execution Model and Cost Estimates

In this section, we sketch the execution model and execution costs of FAN programs in the current implementation. For the sake of simplicity, we consider only data-parallel skeletons on one-dimensional arrays. We assume all input arrays to be distributed block-wise on processors in advance, while input scalars are replicated. Skeletons in the body are executed one after the other on all available processors, and the total cost of a program is the sum of costs of all skeletons in the body. The costs of skeletons are given in Table 2 and explained below:

FAN Operation	Time estimate
map f	mt_f
arrange proj [i]	0
arrange (x,x)	$2mt_{copy}$
reduce ($\oplus$)	$mt_\oplus + \log p(t_s + t_w + t_\oplus)$
scanR ($\oplus$)	$2mt_\oplus + \log p(t_s + t_w + 2t_\oplus)$
scanL ($\oplus$)	$2mt_\oplus + \log p(t_s + t_w + 2t_\oplus)$

Table 2. Costs for basic operations and arrangement functions.

We assume to work on vectors of length n, on p processors so that each processor has $m = n/p$ contiguous elements to compute. map f applies function f in parallel to all the array elements residing on each processor. If t_f is the cost of applying f to an array element than the cost of map f is mt_f.

The execution and cost of arrange depend on its particular parameters. For instance, arrange proj [i] does not imply costs as we simply select the needed value, ignoring the rest. However, projection may sometimes incur costs, e.g., if the user wants to save memory on the program. On the other hand, arrange (a,b) requires copying, which can be done in parallel on all processors, leading to a cost of $2mt_{copy}$.

Skeleton reduce $\oplus$ is executed in two steps. The first step reduces data locally to each processor according to the operator $\oplus$; this costs $mt_{\oplus}$. Then a global reduce is computed in $\log p$ phases using a butterfly-like communication pattern [14]. In each phase, two elements of the vector are exchanged pairwise between processes. Assuming that $T_{sr} = t_s + mt_w$ is the cost for exchanging a two m sized message between two processes (t_s is the start-up time and t_w is the per-element transfer time), the cost estimate if the global phase is $\log p(t_s + t_w + t_{\oplus})$. Adding up the two phases, we obtain the cost of reduce ($\oplus$), as $mt_{\oplus} + \log p(t_s + t_w + t_{\oplus})$.

The execution model for scanR differs from reduction in that we need three steps. The first step computes scanR locally in parallel on the blocks of data resident on processors (cost $mt_{\oplus}$). The second step executes a global scanR using the last element of a local vector, on a butterfly-like schema similar to the one used for reduce, except that reduce requires one operation per element received whereas scanR requires two. This phase costs $\log p(t_s + t_w + 2t_{\oplus})$. Finally, each processor combines the received value to local values which costs $mt_{\oplus}$. Thus, the total cost is: $2mt_{\oplus} + \log p(t_s + t_w + 2t_{\oplus})$. scanL works in a similar way and has the same cost.

Costs for transformation rules are given in Tab. 3. To make things simpler, costs for copying and making local operations are assumed to be all equal, and t_s and t_w are normalized to this cost.

Rule	Time before	Time after rule application
SR-ARA	$3m + \log p(2(t_s + t_w) + 3))$	$2m + \log p(t_s + 2t_w + 2)$
AR-RA	$m + \log p(t_s + t_w + 1)$	$2m + \log p(t_s + 2(t_w + 1))$
SAR-ARA	$3m + \log p(2(t_s + t_w) + 3)$	$5m + \log p(t_s + t_w + 1)$

Table 3. Performance estimates for optimization rules

The cost of the left-hand side of the SR-ARA rule is obtained by adding the costs of reduce and scanL (vectors a, b have length n). On the right-hand side, initial pairing doubles the size of elements in vector b with respect to vector a. Thus, reduce works on elements of length 2 (and thus the cost of sending/computing is $2(t_w+1)$). The cost of rule SAR-ARA are derived assuming that before the application a and b are vectors of pairs of length n, and after the application a is a vector of pairs of length n and b is a vector of quadruples of length n. Thus scanL and reduce before application work on elements on length 2, while after application, the composed reduce works on elements of size 4.

We can use the costs derived for rules to estimate the conditions under which a particular rule is worth applying. Rule SR-ARA always improves costs, whereas rule AR-RA, applied alone, worsens costs (as expected). The combined rule SAR-ARA improves costs if

$$(t_s + t_w + 2)\ \log p \ > \ 2m$$

For our case study, this result indicates that program alg-mss3 is better than alg-mss2 for larger numbers of processors, and for systems with comparatively slow communication, i.e. big values for t_w and t_s.

6 Experimental Results

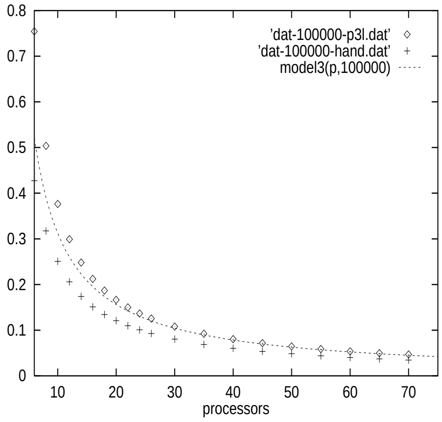
Our target program alg-mss3 has been manually rewritten in the P3L language as shown in Fig 2:

```
#define N 100000
seq op4 in(int x[4], int y[4]) out(int z[4])
 ${     z[0]=max(max(x[0],y[0]),(x[2]+y[1]));
        z[1]=max(x[1],(x[3]+y[1]));
        z[2]=max(y[2],(x[2]+y[3]));
        z[3]=x[3]+y[3];  }$
end seq

seq  w_make_quad in(int x) out(int y[4])
 ${ int i;
    for(i=0;i<4;i++)
      y[i]=x;  }$
end seq

reduce red_four in(int x[N][4]) out(int y[4])
    op4 in(x[*][]) out(y)
end reduce

map make_quadruple in(int a[N]) out(int a4[N][4])
     w_make_quad in(a[*i]) out(a4[*i][])
end map

comp mss in(int a[N]) out(int b[4])
  make_quadruple in(a) out(int a4[N][4])
  red_four in(a4) out(b)
end comp
```

Fig. 2. The testbed P3L version of the alg-mss3 program.

Here the sequential skeleton (introduced by `seq`) encapsulated a fragment of C code defining an operator or a function which is to be applied in a skeleton instance. The sequential `op4` defines the $New(\ldots)$ operation in terms of C assignments and the sequential `w_make_quad` creates a 4-tuple replicating an integer value. The `map` skeleton in general implements a FAN arrange followed by a map. In this case, the map `make_quad` then implements the arrange$(\ldots)$ and creates in parallel the array of tuples. The P3L `reduce` skeleton implements the FAN reduce using the binary operator `op4`.

The P3L program was automatically translated into a C+MPI code, and run on a Fujitsu AP1000 located at the Imperial College London. The results are summarized by the plots in Fig. 3. They show both the predicted and the measured performance, for two problem sizes: 10^5 and 10^6.



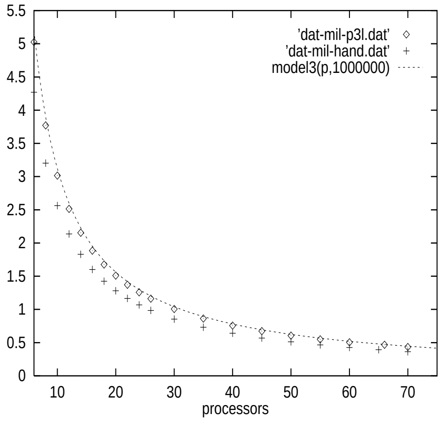


Fig. 3. Run times (sec) of alg-mss3 on the problem size 10^5 (left) and 10^6 (right).

In our experiments, the data are already distributed on processors. We compare the C+MPI code, xxxp3l.dat, generated by the p3l compiler [7], and a corresponding hand-coded MPI version, xxxhand.dat. Curve model3 shows the costs predicted using our performance estimate of Section 5. The plots demonstrate that the p3l code is a bit slower than the hand-coded one, but still reaches good efficiency. The code analysis indicates that the main sources of inefficiency are function calls inserted to compute quadruples and copying in the communication buffers. The predictability of performance is strikingly good, which we explain by a very simple structure of the target program, which consists of just one reduction. In addition, computations are very regular: the cost of the reduction operator New (op2, max.pair) does not depend on the input data.

7 Conclusion

In this paper, we argue that the implementations of high-level languages should be extended by special frameworks to support the process of designing efficient high-level programs. We presented the FAN transformational framework, showed how it can be integrated with the P3L programming system, and demonstrated its use on a case study with machine experiments. The integrated system consisting of FAN and P3L provides the user with a rapid prototyping tool by automatically producing an executable C+MPI code for the algorithm under design, together with expected performance estimates.

Because of the lack of space we could only sketch some aspects of the framework which deserve special consideration: the rich collection of transformations available (more details in [16]), details of cost estimates and benchmarking on different platforms, complete syntax and exact semantics of FAN notation, task-parallel skeletons like pipe, farm, etc.

Related work which inspired our research includes both higher-order functional programming [20] and high-level imperative programming environments. Data parallel languages such as NESL [6] and HPF [18] supports implicit parallelization over bulk data structures. Coordination languages like PCN [12] take a much more explicit approach than ours: tasks are composed by connecting pairs of communication ports, using primitive composition operators. Further examples are the use of programmer-oriented abstractions on top of MPI in the PEMPI system at Basel University [11], and a high-level extension of C++, PROMOTER, developed at GMD FIRST Berlin [13]. For an excellent survey of a variety of high-level parallel models, see [21]. Some ongoing projects were presented at a recent Dagstuhl seminar [9].

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

Our current work and future plans address the following aspects:

- to develop design strategies for applying sequences of transformations, and to incorporate these strategies into the FAN transformation engine;
- providing that there is a choice of parallel implementations for particular FAN operations, to exploit this in the transformation process, and to pass the resulting algorithms together with implementation hints to the P3L compiler;
- to lift some of the arrangement optimizations, currently done by the P3L compiler, to the FAN level;
- to incorporate transformations for task-parallel skeletons of P3L [2] into our framework.

Thanks to Silvia Ciarpaglini for helping with the computer experiments. We also thank IFPC London for the access to the Fujitsu AP1000 used in the experiments. The anonymous referees helped to improve the presentation.

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

References

1. S. G. Akl. *Parallel Computation: models and methods.* Prentice-Hall, 1997.
2. M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *Proceedings of First Int. Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, pages 44–58. Techreport 9805, University of Passau, May 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. J. Bentley. Programming pearls. *Comm. ACM*, 27:865–871, 1984.

5. R. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.
6. G. Blelloch. NESL: a nested data-parallel language (Version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993.
7. S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. Anacleto: a template-based P^3L compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, Australian National University, Canberra, 1997.
8. M. Cole. Parallel programming with list homomorphisms. *Par. Proc. Letters*, 5(2):191–204, 1994.
9. M. Cole, S. Gorlatch, C. Lengauer, and D. Skillicorn, editors. *Theory and Practice of Higher-Order Parallel Programming*. Dagstuhl-Seminar Report 169, Schlo Dagstuhl. 1997.
10. J. Darlington, Y. Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.
11. N. Fang and H. Burkhart. Structured parallel programming using mpi. In *Proceedings of HPCN'96*, pages 840–847. Springer-Verlag, 1996.
12. I. Foster and S. Taylor. A compiler approach to scalable concurrent-program design. *ACM TOPLAS*, 16(3):577–604, 1994.
13. W. K. Giloi, M. Kessler, and A. Schramm. A high-level, massively parallel programming environment and its realization. In *Proceedings of the Real World Computing Symposium (RWC'97). Tokyo*, 1997.
14. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Parallel Processing. Euro-Par'96, Vol. II*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
15. S. Gorlatch and C. Lengauer. (De)Composition rules for parallel scan and reduction. In *Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 23–32. IEEE Computer Society Press, 1998. Available at http://brahms.fmi.uni-passau.de/cl/papers/GorLe97c.html.
16. S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. Report MIP-9813, Universität Passau, October 1998. http://brahms.fmi.uni-passau.de/cl/papers/Gor.Wed.Len:report.oct98.html.
17. W. Gropp, E. Lusk, and A. Skijellum. *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, 1994.
18. High Performance Fortran Forum. High Performance Fortran language specification. Version 2.0, Department of Computer Science, Rice University, Jan. 1997.
19. S. Pelagatti. *Structured development of parallel programs*. Taylor&Francis, 1998.
20. D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
21. D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 1998.
22. D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.
23. D. Swierstra and O. de Moor. Virtual data structures. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 355–371. Springer-Verlag.