# AUTOMATIC PROGRAM TRANSFORMATION: THE META TOOL FOR SKELETON-BASED LANGUAGES*

MARCO ALDINUCCI†

**Abstract.** Academic and commercial experience with skeleton-based systems has demonstrated the benefits of the approach but also the lack of methods and tools for algorithm design and performance prediction. We propose a (graphical) transformation tool based on a novel internal representation of programs that enables the user to effectively deal with program transformation. Given a skeleton-based language and a set of semantic-preserving transformation rules, the tool locates applicable transformations and provides performance estimates, thereby helping the programmer in navigating through the program refinement space.

**Key words.** Algorithmic skeletons, program transformation, parallel programming, performance models, Bulk-Synchronous Parallelism.

**1. Introduction.** Structured parallel programming systems allow a parallel application to be constructed by composing a set of basic parallel patterns called *skeletons*. A skeleton is formally an higher order function taking one or more other skeletons or portions of sequential code as parameters, and modeling a parallel computation out of them.

Cole introduces the skeleton concept in the late 80's [10]. Cole's skeletons represent parallelism exploitation patterns that can be used (instantiated) to model common parallel applications. Later, different authors acknowledge that skeletons can be used as constructs of an explicitly parallel programming language, actually as the only way to express parallel computations in these languages [11, 6]. Recently, the skeleton concept evolved, and became the coordination layer of structured parallel programming environments [5, 7, 20].

Usually, the set of skeletons includes both data parallel and task parallel patterns. Data parallel skeletons model computations in which different processes cooperate to compute a single data item, whereas task parallel skeletons model computations whose parallel activities come from the computation of different and independent data items.

In most cases, in order to implement skeletons on parallel architectures efficiently, compiling tools based on the concept of *implementation template* (actually a parametric processes network) have been developed [10, 6].

Furthermore, due to the fact that the skeletons have a clear functional and parallel semantics, different rewriting techniques have been developed that allow skeleton programs to be transformed/rewritten into equivalent ones achieving different performances when implemented on the target architecture [8, 13]. These transformations can also be driven by some kind of analytical performance models, associated with the implementation templates of the skeletons, in such a way that only those rewritings leading to efficient implementations of the skeleton code are considered.

The research community has been proposing several development frameworks based on the refinement of skeletons [11, 12, 21]. In such frameworks, the user starts by writing an initial skeletal program/specification. Afterwards, the initial specification may be subjected to a cost-driven transformation process with the aim of improving the performance of the parallel program. Such transformation is done by means of semantic-preserving rewriting

* The preliminary version of this paper appeared in [1]. This work has been partially supported by the VIGONI German-Italian project and by the MOSAICO Italian project.

†Computer Science Department, University of Pisa, Corso Italia 40, I-56125 Pisa, Italy. (aldinuc@di.unipi.it).

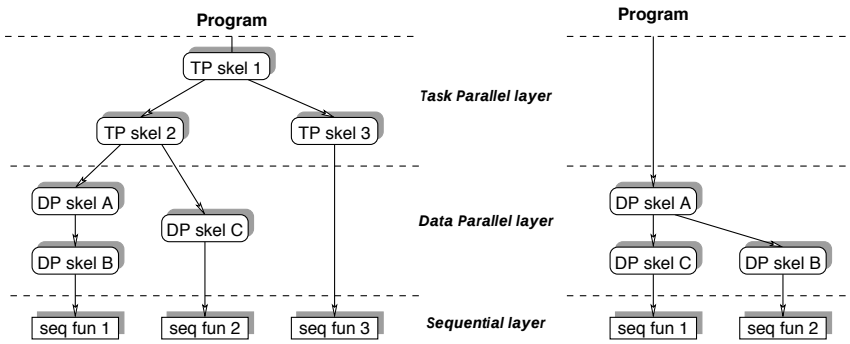



FIG. 2.1. *Three-tier applications: two correct skeleton calling schemes.*

rules. A rich set of rewriting rules [2, 3, 4, 13] and cost models [21, 23] for various skeletons have been developed recently.

In this paper we present Meta, an interactive transformation tool for skeleton-based programs. The tool basically implements a term rewriting system that may be instantiated with a broad class of skeleton-based languages and skeleton rewriting rules. Basic features of the tool include the identification of applicable rules and the transformation of a subject program by the application of a rule chosen either by the user or accordingly with some performance-driven heuristics. Meta is based on a novel program representation (called *dependence tree*) that allows to effectively implement a rewriting system via pattern-matching.

The Meta tool can be used as a building block in general transformational refinement environments for skeleton languages. Meta has already been used as transformation engine of the FAN skeleton framework [4, 12], that is a pure data parallel skeleton framework. Actually, Meta is more general and may be also used in a broad class of mixed task/data parallel skeleton languages [7, 20, 23].

The paper is organized as follows. Section 2 frames the kind of languages and transformations Meta can deal with. The Skel-BSP language, used as a test-bed for Meta, is presented. Section 3 describes the Meta transformation tool and its architecture. Then, Section 4 discusses a case study and the cost models for Skel-BSP, presenting some experimental results. Section 5 assesses some related work and concludes.

**2. Skeletons and transformations.** We consider a generic structured coordination language TL (for *target language*) where parallel programs are constructed by composing procedures in a conventional base language using a set of high-level pre-defined skeletons. We also assume that the skeletons set has three kinds of skeletons: *data parallel*, *task parallel* and *sequential* skeletons. Sequential skeletons encapsulate functions written in any sequential base language and are not considered for parallel execution. The others provide typical task and data parallel patterns. Finally, we constrain data parallel skeletons to call only sequential skeletons. This is usually the case in real applications and it is satisfied by the existing skeleton languages [11, 5, 7, 4, 23, 20]. Applications written in this way have the (up to) three-tier structure sketched out in Fig. 2.1.

In order to preserve generality, Meta can be specialized with the TL syntax and its three skeleton sets. The only requirement we ask is that the above constraint on skeleton calls holds. This makes our work applicable to a variety of existing languages.

Besides a skeleton-based TL, the other ingredient of program refinement by transforma-

tion is a set of semantic-preserving rewriting rules. A rule for TL is a pair $L \rightarrow R$, where $L$ and $R$ are fragments of TL programs with variables $\nu_0, \nu_1 \ldots$ ranging over TL types, acting as placeholder for any piece of program. We require that every variable occurring in $R$ must occur also in $L$ and that $L$ is not a variable. Moreover, a variable may be constrained to assume a specified type or satisfy a specific property (e.g., we may require an operator to distribute over another operator). The left-hand side $L$ of a rule is called a *pattern*.

In the rest of the paper, we consider a simple concrete target language as a test-bed for the Meta transformation tool: Skel-BSP[23]. Skel-BSP has been defined as a subset of P3L [6] on top of BSP (Bulk-Synchronous Parallel model [22]) and it can express both data and task parallelism. The following defines a simplified Skel-BSP syntax which is particularly suitable for expressing rules and programs in a compact way:

*TL_prog* ::= *TP* | *DP*
*TP* ::= farm "(" *TP* ")" | pipe "{" *TPlist* "}" | *DP*
*TPlist* ::= *TP* | *TP*, *TPlist*
*DP* ::= map *Seq* | map2 *Seq* | scanL *Seq* | reduce *Seq* | *Seq* |
comp "(" out *Var*, in *Varlist* ")" "{" *DPlist* "}"
*DPlist* ::= *Var* "=" *DP Varlist* | *Var* "=" *DP Varlist*, *DPlist*
*Var* ::= ⟨ *a string* ⟩
*Varlist* ::= *Var* | *Var*, *Varlist*
*Seq* ::= ⟨ *a sequential C function* ⟩

*TL_prog* can be formed with skeleton applications, constants, variables or function applications. Each skeleton instance may be further specified by its name just adding a dotted string after the keywords (e.g. comp.*mss*). Variables are specified by a name and by a type ranging over (all or some of) the base language types (e.g. all C types except pointers). The type of variables may be suppressed where no confusion can arise.

The pipe skeleton denotes functional composition where each function (stage) is executed in pipeline on a stream of data. Each stage of the pipe runs on different (sets of) processors. The farm skeleton denotes "stateless" functional replication on a stream of data. The map, scanL and reduce skeletons denote the namesake data parallel functions [8] and do not need any further comment. map2 is an extended version of map, which works on two arrays (of the same lengths) as follows: map2 $f\ [x_0, \ldots, x_n]\ [y_0, \ldots, y_n] = [f\ x_0\ y_0, \ldots, f\ x_n\ y_n]$. The comp skeleton expresses the sequential composition of data parallel skeletons. The body of the comp skeleton is a sequence of equations defining variables via expressions. Such definitions follows the *single-assignment* rule: there is at most one equation defining each variable.

comp.*name* (out *outvar*, in *invars*){
$outvar_1$ = dp.*1* $Op_1$ $invars_1$
⋮
$outvar_n$ = dp.*n* $Op_n$ $invars_n$}
where: $\forall k = 1..n, invars_k \subseteq \left(\bigcup_{i<k} outvar_i \cup invars\right), outvar \in \bigcup_{i \leq n} outvar_i$

The skeletons into the comp are executed in sequence on a single set of processors in a lock-step fashion, possibly with a (all-to-all) data re-distribution among steps. The cost estimate of Skel-BSP is based on the Valiant's Bulk-Synchronous Parallel model [22, 23]. The cost model for Skel-BSP is discussed in Section 4 along with some results on its accuracy.

Results show that close estimate are possible on a fairly common parallel platform like a cluster of Pentium PCs.

**2.1. Examples.** In this section, we consider a couple of simple Skel-BSP programs: the maximum segment sum and the polynomial evaluation. Both programs are the Skel-BSP presentation of parallel algorithms appeared in [4, 12].

**Maximum segment sum.** Given a one-dimensional array of integers $v$, the maximum segment sum (MSS) is a contiguous array segment whose members have the largest sum among all segments in $v$. Suppose we would like to compute the MSS of a stream of arrays. The following code is a first parallel program for computing MSS following a simple strategy [4, 12]:

```
pipe.mss {
  map pair,                 /* : int [n]    → int [n][2]  */
  scanL Op+,                /* : int [n][2] → int [n][2]  */
  map P1,                   /* : int [n][2] → int [n]     */
  reduce max}               /* : int [n]    → int         */
```

The comments on the right hand side state the type of each skeleton instance; types are expressed using a C-like notation. The operator $Op_+$ is defined as follows:

$$[x_{i,1}, x_{i,2}]Op_+[x_{j,1}, x_{j,2}] = [max\{x_{i,1} + x_{j,2}, x_{j,1}\}, x_{i,2} + x_{j,2}]$$

while $pair\ x = [x, x]$ and $P_1\,[x_1, x_2] = x_1$. Intuitively, the purpose of scanL is to produce an array $s$ whose $i$th element is the maximum sum of the segments of $x$ ending at position $i$. Using a sequential program, this task can be accomplished simply by using scanL with operator $Op_1(a, b) = max(a + b, b)$. Unfortunately, such operator is not associative, thus this simple scanL cannot be parallelized. $Op_+$ uses an auxiliary variable to preserve the associativity. This variable is thrown away at the end of the scanL computation by the $P_1$ operator. Finally, reduce sorts out the maximum element of array $s$ yielding to the desired maximum segment sum $r$.

**Polynomial evaluation.** Let us consider the problem of evaluating in parallel a polynomial $a_1x + a_2x^2 + \ldots a_nx^n$ at $m$ points $y_1, \ldots y_m$. The most intuitive solution consists in parallelizing each basic step of the straightforward evaluation algorithm, i.e. first compute the vector of powers $ys^i = [y_1^i, \ldots, y_m^i], i = 1 \ldots n$, then multiply by the coefficients, and, finally, sum up the intermediate results. The algorithm can be coded in Skel-BSP as follows.

```
comp.pol_eval (out zs, in ys, as) {
  ts = scanL * ys,                    /* ts[i] = ys^i : float [n][m] */
  ds = map2 (*sa) as, ts,             /* ds[i] = [a_i * y_1^i, ..., a_i * y_m^i] : float [n][m] */
  zs = reduce + ds}                   /* zs[i] = [sum_{i=1}^n a_i * y_1^i, ..., sum_{i=1}^n a_i * y_m^i] : float [m] */
```

where $*_{sa}$ multiplies each element of a vector by a scalar value, $*$ and $+$ are overloaded to work both on scalars and (element-wise) on vectors. On the right side (in comments) we describe the variable values and types.

**2.2. Transformation rules.** When we design a transformation system a foremost step is the choice of the rewriting rules to be included and the definition of their costs. The goal of the system is to derive a skeletal program with the best performance estimate by successive (semantic-preserving) transformations (rewrites). Each transformation/rewrite

correspond to the application of a rewriting rule. Here, we only collect the transformations needed to demonstrate the use of Meta on an example. We refer back to the literature for the proofs of the soundness of the rules [2, 3, 4, 8, 13]. For the sake of brevity, we use $L \rightleftarrows R$ to denote the couple of rules $L \rightarrow R$ and $R \rightarrow L$.

In the following, $\mathsf{TSk}_i$ can be any skeleton (task or data parallel, sequential), $\mathsf{DSk}_i$ can be any data parallel or sequential skeleton. $Op_1, Op_2, \ldots$ denote variables ranging over sequential functions. *pair* and $P_1$ are sequential auxiliary functions defined in the previous section. The labelled elision $< \cdots >_n$ represents an unspecified chunk of code that appears (unchanged) in both sides of the rules.

farm **insertion/elimination.** These rules state that farms can be removed or introduced on top of a TSk skeleton [3]. The rule preserves the constraint on layers since TSk cannot appear into a data parallel skeleton. A farm replicates TSk without changing the function it computes. Thus, it just increases task parallelism among different copies during execution.

$$\mathsf{TSk} \quad \underset{\leftarrow}{\overset{\rightarrow}{}} \quad \mathsf{farm}\ (\mathsf{TSk})$$

pipe $\rightarrow$ comp**.** The pipe skeleton represents the functional composition for both task and data parallel skeletons. The comp models a (possibly) more complex interaction among data parallel skeletons. If all the stages $\mathsf{DSk}_1$, $\mathsf{DSk}_2 \ldots$ of the pipe are data parallel (or sequential) skeletons, then the pipe can be rewritten as a comp in which each $\mathsf{DSk}_i$ gets its input from $\mathsf{DSk}_{i-1}$ and outputs towards $\mathsf{DSk}_{i+1}$ only. Also in this case the two formulations differ primarily in the parallel execution model. When arranged in a pipe, the $\mathsf{DSk}_1$, $\mathsf{DSk}_2 \ldots$ are supposed to run on different sets of processors, while arranged in a comp, they are supposed to run (in sequence) on a single set of processors.

$$\begin{array}{l} \mathsf{pipe}\ \{ \\ \quad \mathsf{DSk}_1\ Op_1, \\ \quad \mathsf{DSk}_2\ Op_2, \\ \quad < \cdots >_1 \\ \quad \mathsf{DSk}_n\ Op_n \} \end{array} \quad \rightarrow \quad \begin{array}{l} \mathsf{comp}\ (\mathsf{out}\ z, \mathsf{in}\ a)\ \{ \\ \quad b = \mathsf{DSk}_1\ Op_1\ a, \\ \quad c = \mathsf{DSk}_2\ Op_2\ b, \\ \quad < \cdots >_1 \\ \quad z = \mathsf{DSk}_n\ Op_n\ y \} \end{array}$$

map **fusion/fission.** This rule denotes the map (backwards) distribution through functional composition [8]. Notice that when we apply form left-to-right we do not require the two maps in the left hand side to be adjacent in the program code. We just require that the input to the second one ($q$) is the output from the first one.

$$\begin{array}{l} \mathsf{comp}\ (\mathsf{out}\ outvar, \mathsf{in}\ invars)\ \{ \\ \quad < \cdots >_1 \\ \quad q = \mathsf{map}\ Op_1\ p, \\ \quad < \cdots >_2 \\ \quad r = \mathsf{map}\ Op_2\ q, \\ \quad < \cdots >_3\ \} \end{array} \quad \underset{\leftarrow}{\overset{\rightarrow}{}} \quad \begin{array}{l} \mathsf{comp}\ (\mathsf{out}\ outvar, \mathsf{in}\ invars)\ \{ \\ \quad < \cdots >_1 \\ \quad q = \mathsf{map}\ Op_1\ p, \\ \quad r = \mathsf{map}\ (Op_2 \circ Op_1)\ p, \\ \quad < \cdots >_2 \\ \quad < \cdots >_3\ \} \end{array}$$

It is important to notice that, while rules are required to be *locally* correct, Meta ensures the *global* correctness of programs. For instance, using the rule from left-to-right (map fusion) the assignment in the grey box is not required to appear. Meta provides the program with the additional assignment (in the grey box) only if the intermediate result $q$ is referenced in some expressions into $< \cdots >_2$ or $< \cdots >_3$.

**SAR-ARA.** This rule (applied from left-to-right) aims to reduce the number of communications using the very complex operator $Op_3$. In general, the left-hand side is more

communication intensive and less computation intensive than the right-hand side. The exact tradeoff for an advantageous application heavily depends on the cost calculus chosen (see [4, 12]).

comp (out *outvar*, in *invars*){
$< \cdots >_1$
$q$ = scanL $Op_1$ $p$,
$r$ = map $P_1$ $q$,
$s$ = reduce $Op_2$ $r$,
$< \cdots >_2$ }

$\rightarrow$
$\leftarrow$

comp (out *outvar*, in *invars*){
$< \cdots >_1$
$t$ = map *pair* $p$,
$u$ = reduce $Op_3$ $t$,
$v$ = map $P_1$ $u$,
$x$ = map $P_1$ $v$,
$< \cdots >_2$ }

$Op_1$ must distribute forward over $Op_{\text{aux}}$. $Op_3$ is defined as follows:

$$[x_{i,1}, x_{i,2}]Op_3[x_{j,1}, x_{j,2}] = [x_{i,1}Op_{\text{aux}}(x_{i,2}Op_1x_{j,1}), x_{i,2}Op_1x_{j,2}]$$

$$[x_{i,1}, x_{i,2}]Op_{\text{aux}}[x_{j,1}, x_{j,2}] = [x_{i,1}Op_2x_{j,1}, x_{i,2}Op_2x_{j,2}]$$

Notice that, whereas operator $Op_2$ works on single elements, operators $Op_1$ and $Op_{\text{aux}}$ are defined for pairs (arrays of length 2), and $Op_3$ works on pairs of pairs.

**3. The transformation tool.** In this section, we describe a transformation tool which allows the user to write, evaluate and transform TL programs, preserving their functional semantics, and possibly improving their performance. The tool is interactive. Given an initial TL algorithm, it proposes a set of transformation rules along with their expected performance impact. The programmer chooses a rule to be applied and successively (after the application) the tool looks for new matches. This process is iterated until the programmer deems the resulting program satisfactory, or there are no more applicable rules.

The strategy of program transformation is in charge of the programmer since, in general, the rewriting calculus of TL is not confluent: applying the same rules in a different order may lead to programs with different performance. The best transformation sequence may require a (potentially exponential) exhaustive search.

In the following, we define an abstract representation of TL programs and transformation rules, we describe the algorithm used for rule matching, and finally we sketch the structure of the tool.

**3.1. Representing programs and rules.** The Meta transformation system is basically a term-rewriting system. Both TL programs and transformation rules are represented by means of a novel data structure, so-called *dependence tree*. Dependence trees are basically labelled trees, thus the search for applicable rules reduces to the well established theory of subtree matching [16]. The tool attempts to annotate as many nodes of the tree representation as possible with a *matching rule instance*, i.e., a structure describing which rule can be used to transform the subtree rooted at the node, together with the information describing how the rule has been instantiated, the performance improvement expected and the applicability conditions to be checked (e.g., the distributivity of one operator over another).

The dependence tree is essentially an abstract syntax tree in which each non-leaf node represents a skeleton, with sons representing the skeleton parameters that may in turn be skeletons or sequential functions. The leaves must be sequential functions, constants or the special node *Arg( )*. Unlike a parse tree, a dependence tree directly represents the data

TABLE 3.1
*Building up the dependence tree.*

*Input:* PT and DFG for a correct TL program. The starting node $x$ is the root of PT.
No nested *DPblock* are allowed (which can be easily flattened).
*Output:* The dependence tree DT.

*Method:*

1. Let $x$ denote the current node, starting from the root of PT;
2. Copy $x$ from PT on DT along with the arc joining it with its parent (if any), the arc is undirected as it comes from PT;
3. **if** not($x$ = *DPblock*)
4. **then** Recursively apply the algorithm to all sons of $x$ in PT (in any order);
5. **else** Apply Procedure dpb(*DPblock*).

**Procedure** dpb($Node$):
a. From $Node$ follow backward the incoming edges in DFG;
b. **for each** node $C_i$ reached in this way, **do**
c. Copy $C_i$ from DFG to DT along with its out-coming edges;
d. Recursively apply dpb($C_i$) until the starting node *DPblock* or a sink is reached; In the former case add a node *Arg* to represent the formal parameter name.

dependence among skeletons: if the skeleton $\mathsf{Sk}_1$ directly uses data produced by another skeleton $\mathsf{Sk}_2$, then they will appear as adjacent nodes in the dependence tree, irrespectively of their position in the parse tree. Each edge in the dependence tree represents the dependence of the head node from the data produced by the tail node. The dependence tree of a program is defined constructively, combining information held in the parse tree (PT) and in the data flow graph (DFG) of the program. The algorithm to build dependence trees is shown in Table 3.1. The algorithm is illustrated in Fig. 3.1, which shows the parse tree, the data flow graph and the correspondent dependence tree of the polynomial evaluation example (see § 2.1). The nodes labelled with *DPblock* mark the minimum subtrees containing at least one data parallel skeleton, nodes *Arg(as)* and *Arg(ys)* represent the input data of a *DPblock*. In other words, *DPblock* nodes delimit the border between the task parallel and the data parallel layers.

It is important to understand why we need to introduce a new data structure instead of using the parse tree directly. The main reason lies in the nature of the class of languages we aim to deal with, i.e. mixed task/data parallel languages. Nested skeleton calls find a very natural representation as trees. On the contrary, data parallel blocks based on the single-assignment rule (e.g. Skel-BSP comp) need a richer representation in order to catch the dependences among the skeletons (for example a data flow graph). The dependence tree enables us to compact all the information we need in a single tree, i.e. in a data structure on which we can do pattern-matching very efficiently.

There is one more point to address. The dependences shown in Fig. 3.1 are rather simple. In general, as shown in Fig. 3.2, a data structure produced by a single TL statement may be used by more than one statement in the rest of the program. We have two choices: (1) to keep a shared reference to the expression (tree), or (2) to replicate it. In option (1), the data flow can no longer be fully described by a tree. In addition, sharing the subtrees rules out the possibility of applying different transformations at the shared expression (tree) for different contexts. The Meta transformational engine adopts the second option, allowing us to map the data flow graph into a tree-shaped dependence structure. The drawback of replicating expressions is a possible explosion of the code size when we rebuild a TL program from the internal representation. To avoid this, the engine keeps track of all the replications made. This ensures a single copy of all replicated subtrees that have not been subject to an

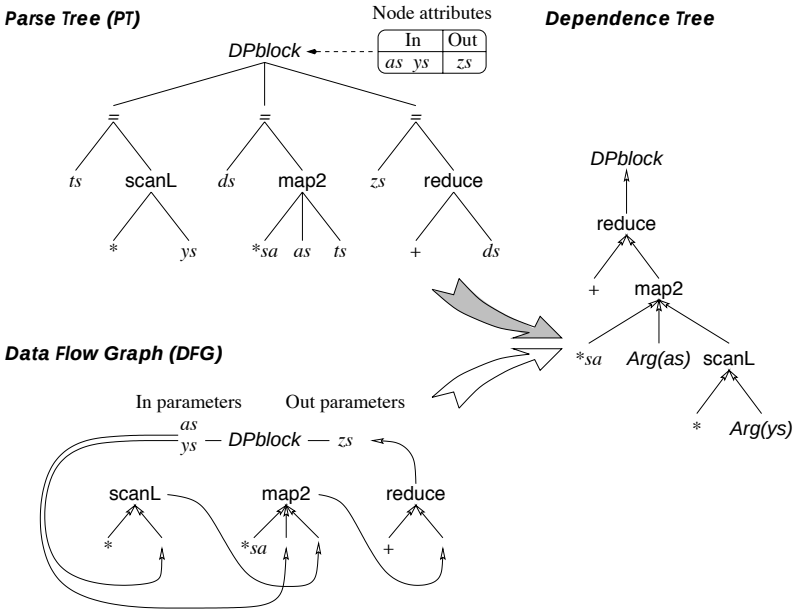



FIG. 3.1. *The parse tree, the data flow graph and the dependence tree of polynomial evaluation.* Skel-BSP *skeletons are in* serif *font. Special nodes are in* slanted serif *font. Sequential functions are in italic font.*

independent transformation.

Figure 3.3 depicts the internal representation of rule map fusion from § 2.2. We represent the two sides of the rule as dependence trees, some leaves of which are variables represented by circled numbers. During the rule application, the instantiations of the left-hand side variables are substituted against their counterparts on the right-hand side. Figure 3.3 demonstrates how the conditions of applicability and the performance of the two sides of a rule are reported to the programmer. Notice in Fig. 3.3 the "functional" fcomp, i.e. a special node used to specify rules in which two (or more) variables of the pattern are rewritten in the functional composition of them. Since variables have no sons, Meta first rewrites variables as sons of fcomp, then it makes the contractum $\{\nu_0 = f_0, \ldots \nu_n = f_n\}$ and, afterwards the result is equated using $\mathsf{fcomp}(f_0, \ldots f_n) = f_n \circ \cdots \circ f_0$.

**3.2. Rule matching.** Since programs and rules are represented by trees, we can state the problem of finding a candidate rule for transforming an expression as the well-known *subtree matching problem* [17, 19, 16]. In the most general case, given a pattern tree $P$ and a subject tree $T$, all occurrences of $P$ as a subtree of $T$ can be determined in time $\mathcal{O}(|P|+|T|)$ by applying a fast string matching algorithm to a proper string representation [19]. Our problem is a bit more specific: the same patterns are matched against many subjects and the subject may be modified incrementally by the sequence of rule applications. Therefore, we distinguish a *preprocessing phase*, involving operations on patterns independent of any subject tree, and a *matching phase*, involving all operations dependent on some subject tree. Minimizing the matching time is our first priority.

The Hoffmann-O'Donnell bottom-up algorithm [16] fits our problem better than the string matching algorithm. With it, we can find all occurrences of a forest of patterns $F$ as

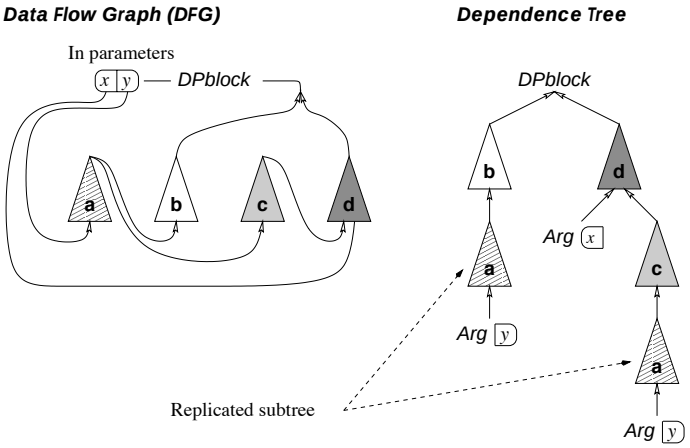



FIG. 3.2. *Replicating shared trees. Each triangle stands for a tree representing a* TL *expression.*

subtrees of $T$ in time $\mathcal{O}(|T|)$, after suitable preprocessing of the pattern set. Moreover, the algorithm is efficient in practice: after the preprocessing, all the occurrences of elements in $F$ can be found with a single traversal of $T$. The algorithm works in two steps: it constructs a *driving table*, which contains the patterns and their interrelations; then, the table is used to drive the matching algorithm.

**The bottom-up matching algorithm.** We textually represent labelled trees as $\Sigma$-terms over a given alphabet $\Sigma$. Formally, all symbols in $\Sigma$ are $\Sigma$-terms and if $a$ is a q-ary symbol in $\Sigma$ then $a(t_1, \ldots, t_q)$ is a $\Sigma$-term provided each of $t_i$ is. Nothing else is a $\Sigma$-term. Let $S_\nu$ denote the set of $(\Sigma \cup \{\nu\})$-terms.

In addition, let $F = \{P_1, P_2, \ldots\}$ be a pattern set, where each pattern $P_i$ is a tree. The set of all subtrees of the $P_i$ is called a *pattern forest (PF)*. A subset $M$ of $PF$ is a match set for $F$ if there exists a tree $t \in S_\nu$ such that every pattern in $M$ matches $t$ at the root and every pattern in $PF \setminus M$ does not match $t$ at the root.

The key idea of the Hoffmann-O'Donnell bottom-up matching algorithm is to find, at each point (node) $n$ in the subject tree, the set of all patterns and all parts of patterns which match at this point. Suppose $n$ is a node labelled with the symbol $b$, and suppose also we have already computed such sets for each of the sons of $n$. Call these sets, from left to right $M_1, \ldots, M_q$. Then the set $M$ of all pattern subtrees that match at $n$ contains $\nu$ (that match anywhere), plus those patterns subtrees $b(t_1, \ldots, t_q)$ such that $t_i$ is in $M_i$, $1 \leq i \leq q$. Therefore, we could compute $M$ by forming $b(t_1, \ldots, t_q)$ for all combinations $(t_1, \ldots, t_q)$, $t_i \in M_i$. Once we have assigned these sets to each node, we have essentially solved the matching problem, since each match is triggered by the presence of a complete pattern in some set.

Notice that there can be only finitely many such sets $M$, because both $\Sigma$ and the set of sub-patterns are finite. Thus we could precompute these sets, and code them by some enumeration to build driving tables. Given such tables, the matching algorithm becomes straightforward: traverse the subject tree in postorder and assign to each node $n$ the code of the set of partial matches as $n$. However, for certain pattern forest the number of such sets $M$ (thus the complexity of the generation of driving tables) grows exponentially with the cardinality of the pattern set.

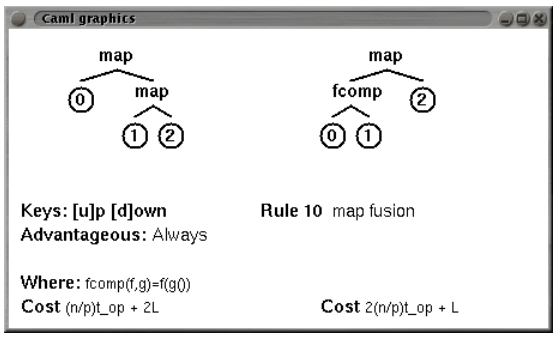



FIG. 3.3. *Internal representation of rule* map *fusion, conditions of its applicability and performance of the two sides of the rule.*

Nevertheless, there is a broad class of pattern sets which can be preprocessed in polynomial time/space in the size of the set: all sets yielding *simple pattern forests*. For an extensive treatment we refer back to Hoffmann-O'Donnell paper [16] and provide only a brief explanation here.

Let $a, b, c \ldots \in \Sigma$. Now, let $P$ and $P'$ be pattern trees. $P$ *subsumes* $P'$ if, for all subject trees $T$, $P$ has a match in $T$ implies that $P'$ has a match in $T$. Then $P$ is *inconsistent* with $P'$ if there is no subject tree $T$ matched by both $P$ and $P'$. $P$ and $P'$ are *independent* if there exist $T_1$, $T_2$, and $T_3$ such that $T_1$ is matched by $P$ but not by $P'$, $T_2$ is matched by $P'$ but not by $P$, and $T_3$ is matched by both $P$ and $P'$. Given distinct patterns $P$ and $P'$, exactly one of the three previous relations must hold between them. A pattern forest is called *simple* if it contains no independent subtrees. For instance, the pattern forest including the pattern trees $P = a(b, \nu)$ and $P' = a(\nu, c)$ is not simple, since $P$ and $P'$ are independent with respect to $T_1 = a(b, b), T_2 = a(c, c), T_3 = a(b, c)$. On the contrary, the pattern set $F_s = \{a(a(\nu, \nu), b),\ a(b, \nu)\}$ lead to a simple pattern forest. The current set of Skel-BSP rules [2, 3, 23] and FAN rules [4] can be fully described by simple pattern forests. Simple pattern forests suffice even for producing interpreters of much more complex languages like LISP and the combinator calculus [15]. In addition, since the driving table depends only on the language and on the list of rules, it can be generated once and for all for a given set of rules and permanently stored for several subsequent match searches.

**3.3. Tool architecture and implementation.** The transformation engine applies the matching algorithm in an interactive cycle as follows:

1. Use the matching algorithm to annotate the dependence tree with the matching rules.
2. Check whether the rules found satisfy the type constraints and whether the side conditions hold (possibly interacting with the user).
3. Apply the performance estimates to establish the effect of each rule.
4. Ask the programmer to select one rule for application. In case no rule is applied, terminate; otherwise start again from Step 1.

We envision the Meta tool as a part of a general tool implementing a transformational refinement framework for a given target language TL. The global tool structure is depicted in Fig. 3.4 (the part already implemented is highlighted with a dotted box). The whole system has two main capabilities: the conversion between TL programs and their internal

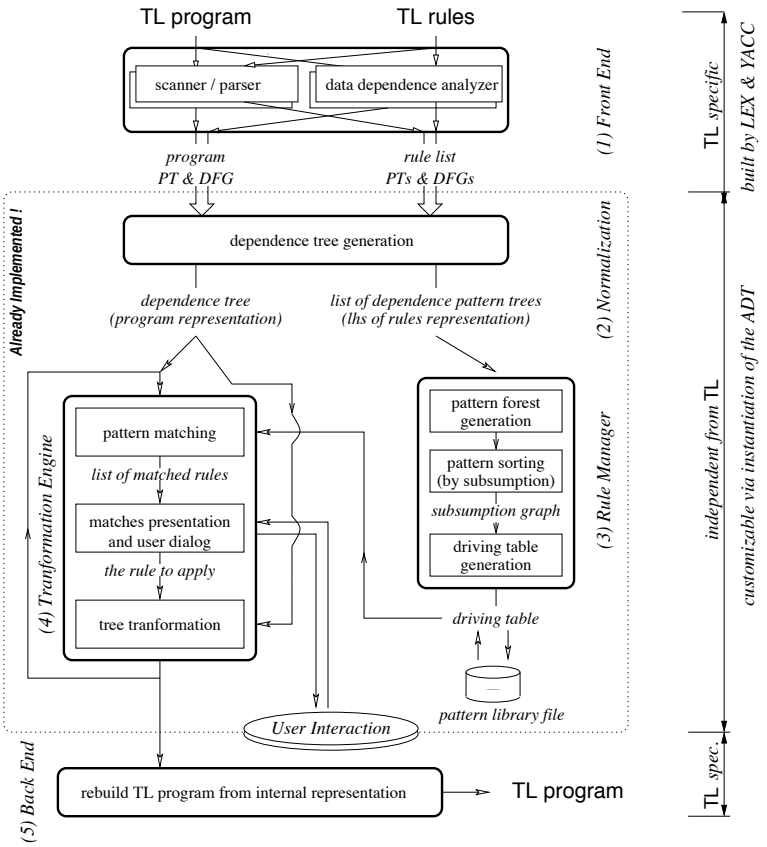



FIG. 3.4. *Global structure of the* Meta *transformation system.*

representation (dependence tree) and the transformation engine working on dependence trees.

The system architecture is divided into five basic blocks:

1. The *Front End* converts a TL program into a parse tree and a data flow graph.
2. The *Normalization* uses the PT and DFG to build the dependence tree both for the TL program and for the set of transformation rules.
3. The *Rule Manager* implements the preprocessing of rules (preprocessing phase, see §3.2); it delivers a matching table to drive the transformation engine. The driving table may be stored in a file.
4. The *Transformation Engine* interacts with the user and governs the transformation cycle.
5. The *Back End* generates a new TL program from the internal representation.

A prototype of the system kernel (highlighted in Fig. 3.4 with a dotted box) has been implemented in Objective Caml 2.02. Our implementation is based on an abstract data type (ADT) which describes the internal representation (dependence tree) and the functions working on it. The implementation is very general and can handle, via instantiation of the ADT, different languages with the requirement that rules and programs are written

in the same language. Moreover, since several execution models and many cost calculi may be associated with the same language, any compositional way of describing program performance may be embedded in the tool by just instantiating the performance formulae of every construct. We call a cost calculus *compositional* if the performance of a language expression is either described by a function of its components or by a constant.

The Meta transformation tool prototype is currently working under both Linux and Microsoft Windows. A graphical interface is implemented using the embedded OCaml graphics library.

**4. A case study: design by transformation.** We discuss how Meta can be used in the program design process for the MSS algorithm, introduced in §2.1 and reported in the top-left corner of Table 4.1.

First, the tool displays the internal representation of the program (Fig. 4.1 (a)) and proposes 5 rules (Fig. 4.1 (b)). The first one is pipe→comp rule, the others are instances of the farm introduction rule. The four stages of the pipe use exactly the same data distribution, but since each stage use a different set of processors each stage has to scatter and gather each data item. Transforming the pipe in a comp (that uses just one set of processors) would get rid of many unnecessary data re-distributions. Let us suppose the user chooses to apply the pipe→comp rule achieving the program version shown in Fig. 4.1 (c). Next, Meta proposes a couple of rules (Fig. 4.1 (d)): SAR-ARA to further reduce the number of communications into the comp, thus to optimize the program behavior on a single data item, and farm introduction to enhance the parallelism among different data items of the stream. Both rules may improve the performance of the program, let us suppose to choose the SAR-ARA (Fig. 4.1 (e)).

Then, the transformation process continues choosing (in sequence) map fusion rule (2 times) and farm introduction rule. The resulting program is only one of the more than twenty different formulations Meta is able to find applying the transformation rules to the initial program. Table 4.1 shows some of the semantic-equivalent formulations derivable.

In the rest of this section, we discuss a cost prediction model for Skel-BSP and we give some results of its accuracy on a concrete parallel architecture.

It is worth reminding that choosing in every step the transformation with the best performance gain does not guarantee to find the fastest program (optimum). Nevertheless, the knowledge of the performance gain/loss of each transformation is quite important to the programmer, since he can make decisions or build transformation strategies (e.g. greedy, tabu search, etc.) using such kind of informations. An accurate prediction of transformations cost is quite important to this end.

In the case of Skel-BSP equipped with BSP costs, such prediction is pretty accurate. In the following section, we give evidence of this accuracy through the following steps. We first describe how Skel-BSP is implemented on a BSP abstract machine running on our concrete parallel architecture. Then we describe how programs and rules can be costed in this implementation. Finally, we compare the predicted and measured performance figures of two versions of our MSS example and compare the performance gain predicted by one transformation rule used by Meta with the real measured figures.

**4.1. Prototyping Skel-BSP.** Our Skel-BSP prototype is implemented using the C language and the PUB library (Padeborn University BSP-library [9]). The PUB library is a C-library of communication routines. These routines allow straightforward implementation of BSP algorithms.

The PUB library offers the implementation of a superset of the *BSP Worldwide Standard*

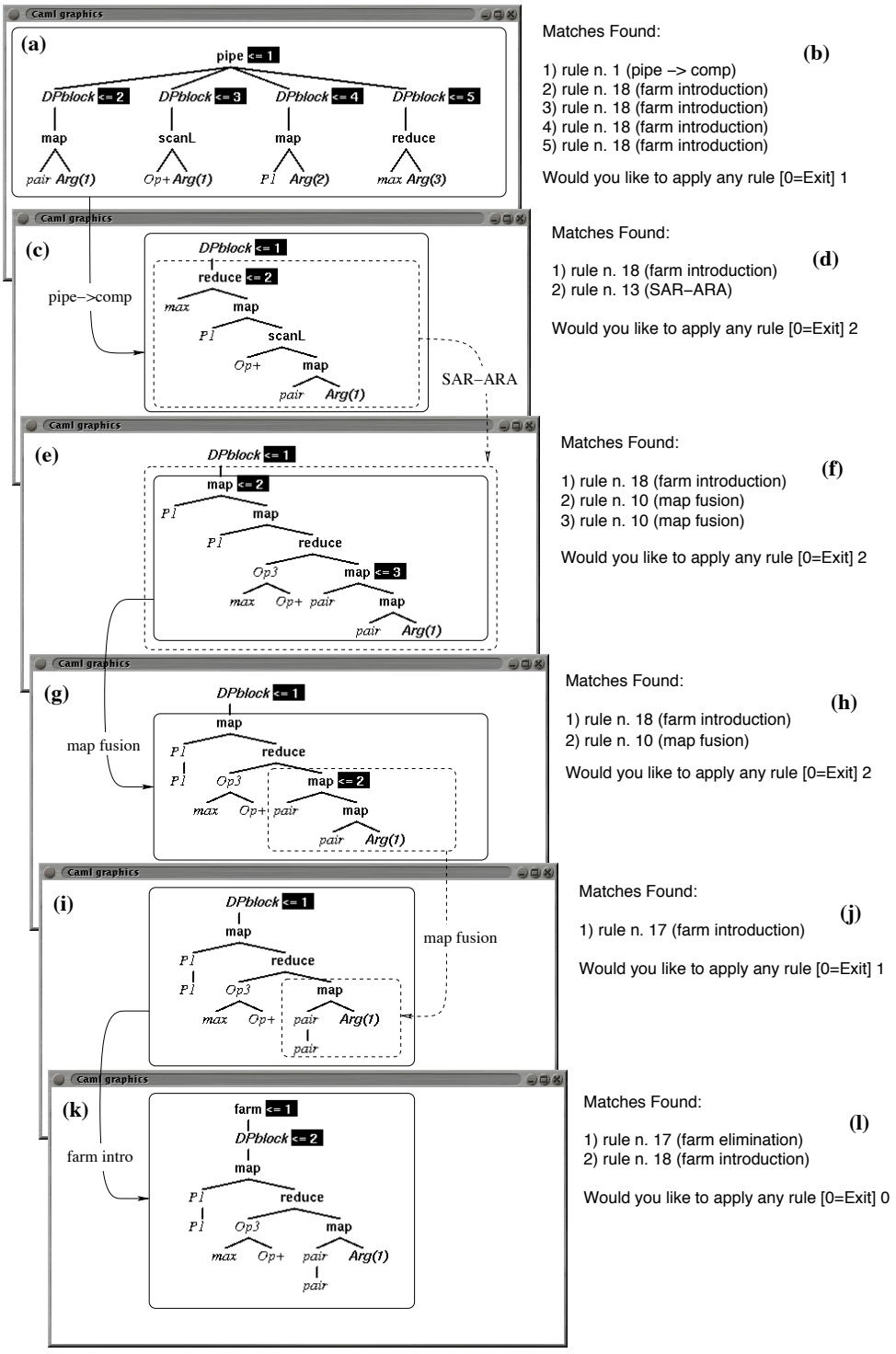
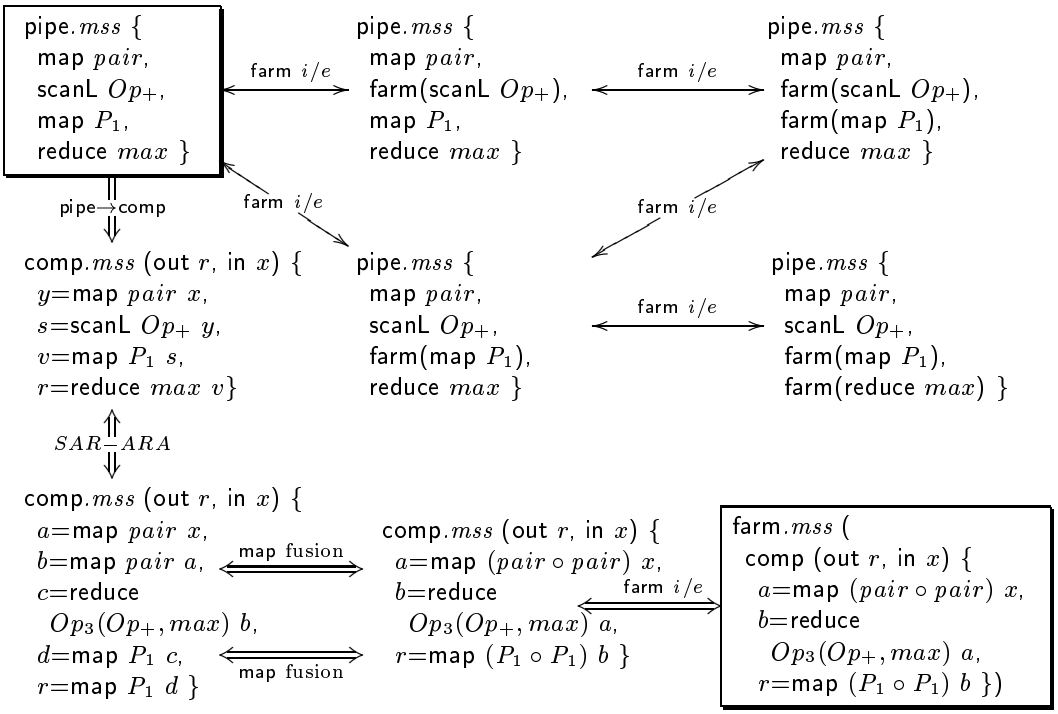



FIG. 4.1. *Transformation of the MSS program using the* Meta *tool.* Skel-BSP *skeletons are in* serif *font. Special nodes are in* slanted serif *font. Sequential functions are in italic font.*

TABLE 4.1
*Some of the transformations proposed by Meta for the MSS example. The double-arrow path denotes the derivation path followed in Fig. 4.1.*




*Interface* [14]. In addition, PUB offers some collective operations (`scan` and `reduce`), and it allows creating independent BSP objects each representing a virtual BSP computer. The last two features make PUB particularly suitable for prototyping Skel-BSP programs:

(i) PUB collective operations may be used to implement Skel-BSP collective operations in a straightforward way. Unfortunately, PUB requires all operations used in `scan` and `reduce` to be commutative. Thus, the direct mapping from PUB to Skel-BSP collective operations may be done only if operations involved are commutative.

(ii) Independent (virtual) BSP computer may be used to implement effectively task parallel skeletons in Skel-BSP. Task parallel activities are often asynchronous on different pool of processors, and do not require all processing elements to synchronize at each superstep. PUB offers the possibility to divide a BSP computer in several subgroups each representing a virtual BSP computer. In this way, computations among processors belonging to different subgroups may proceed asynchronously since superstep barriers involve only processors belonging to the same subgroup.

Since global operations $Op_+$ and $Op_3$ we used in MSS programs are associative but not commutative, we extended PUB with a new parallel prefix operation (`TPscanL`) that requires global operations only to be associative. `TPscanL` is implemented using message passing primitives of PUB (send, receive, broadcast) following the two-phase BSP algorithm:

1. Each processor performs a (local) reduce on the local portion of the structure and broadcasts the result to all the processors with greater index.

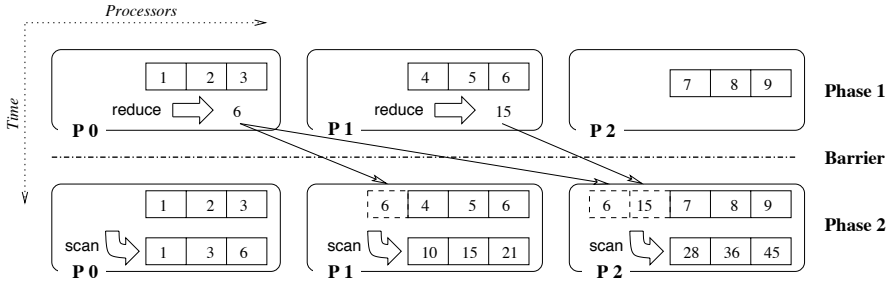



FIG. 4.2. *Two-phase BSP parallel prefix* (`TPscanL`) *using* + *as global operation.*

2. Processor $i > 0$ computes the $i^{th}$ segment of the prefix performing a local scan of the prefix array extended (on the left) with the $i$ results received from all processors with index lower than $i$.

The two-phase parallel prefix algorithm is sketched in Fig. 4.2 using + as global operation. Let $p$ be the number of processors, $n$ the length of prefix array (assumed multiple of $p$), $t_{Op}$ the cost of the global operation, $msg$ the size of a prefix array element and $\{g, l\}$ the usual BSP cost parameters. The BSP cost of `TPscanL` is:

$$T(\texttt{TPscanL}\ \ Op) = \underbrace{\left(\frac{n}{p} - 1\right)\ t_{Op}}_{\text{Phase1}} + \underbrace{g\ (p-1)\ msg + l}_{\text{barrier}} + \underbrace{\left(\frac{n}{p} + p - 2\right)\ t_{Op}}_{\text{Phase2}}$$

The two-phase algorithm is just one of the possible choices for the parallel prefix problem. We use the two-phase parallel prefix `TPscanL` to implement both Skel-BSP scanL and reduce. Notice that, to check the effectiveness of the transformation process, we only need to have an implementation with known cost, we do not need a particularly good implementation. For a comparison between two-phase algorithm and others BSP parallel prefix algorithms we refer back to [23].

**4.2. Running and costing MSS programs.** We focus on two different MSS Skel-BSP programs found using Meta and the proposed set of rules. Let us call *mss_c* and *mss_e* the programs in Fig. 4.1 (c) and (e), respectively. *mss_e* is obtained from *mss_c* using the SAR-ARA rule, as follows:

```
comp.mss_c (out r, in x) {
 y=map pair x,
 s=scanL Op+ y,
 v=map P1 s,
 r=reduce max v}
```

$\Longrightarrow^{SAR-ARA}$

```
comp.mss_e (out r, in x) {
 a=map pair x,
 b=map pair a,
 c=reduce
  Op3(Op+, max) b,
 d=map P1 c,
 r=map P1 d }
```

We describe the expected BSP cost of the two programs. Afterwards, we run a prototype of the two programs on a concrete parallel architecture, consisting in a cluster of Pentium II PCs (@266MHz) interconnected by a 100Mbit switched Ethernet. We instantiate cost formulae with BSP parameters collected during the experiments, miming the behavior of

Meta. Finally, we discuss the accurateness of expected performance predicted by Meta using cost formulae with respect to experimental performance.

Let us assume each processor holds $n/p$ elements of the input array. Since the comp skeleton executes its components in sequence, the cost of the *mss_c* program is figured out summing up the costs of each skeleton appearing into the comp. Notice we use the same primitive (TPscanL) to implement both scanL and reduce, thus the reduce will cost as much as scanL. All operations work on integers (4 bytes long). The *pair* operation consists in copying an integer, thus costs one BSP basic operation ($1 \cdot s$); the cost of the projection $P_1$ is zero. Messages sizes are two integers for the first TPscanL and one integer for the second one. The cost of $Op_+$ operation is assessed in $3 \cdot s$, while $max$ costs just $1 \cdot s$. In total, we assess for the *mss_c* program:

$$\begin{aligned} T(mss\_c) &= T(\mathsf{map}\ pair\ \mathsf{int}) + T(\texttt{TPscanL}\ \ Op_+) + T(\texttt{TPscanL}\ \ max) \\ &= s \cdot (9\ n/p + 4p - 12) + 12g(p-1) + 2l \end{aligned}$$

In the same way we evaluate the cost of the *mss_e* program. The first *pair* operation consists in copying one integer while the second *pair* in copying two integers, the total cost is $3 \cdot s$. The message size for TPscanL is four integers. The cost of $Op_3(Op_+, max)$ is $7 \cdot s$. In total we assess for the *mss_e* program:

$$\begin{aligned} T(mss\_e) &= T(\mathsf{map}\ pair\ \mathsf{int}) + T(\mathsf{map}\ pair\ \mathsf{int}[2]) + T(\texttt{TPscanL}\ \ Op_3(Op_+, max)) \\ &= s \cdot (17\ n/p + 7p - 21) + 16g(p-1) + l \end{aligned}$$

Each run consists on evaluating the MSS for 300 input arrays on several cluster configurations (2, 4, 8, 16 PCs). The length of input arrays ranges from $2^{13}$ to $2^{18}$ integers. All standard BSP parameters are profiled directly by the PUB library:

$$\begin{aligned} &s = 5.7 \cdot 10^{-8} \quad (17.54\ \text{M}\ BSP_{Ops}/\text{sec}) \\ &g = 0.2 \cdot 10^{-6} \quad (500\ \text{K Bytes/sec}) \\ &l = 3.2 \cdot 10^{-4} \cdot p \quad (640 - 5120\ \mu\text{secs, with } p = 2 - 16) \end{aligned} \tag{4.1}$$

Predicted performance and experimental performance of *mss_c* and *mss_e* programs are compared in Fig. 4.3 a) and b), respectively. Considering all experiments, the average relative error of predicted performance with respect to experimental performance is 13% with 7% of standard deviation. Notice that the error in predicted performance grows with the number of processors, thus it grows with the number of communications. This suggests us that communications patterns used by TPscanL and by PUB profiling algorithms are not totally aligned in performance.

**4.3. Performance driven transformations.** Given a language and a set of semantic-preserving transformations, the Meta tool assists the user in the transformation process. The transformation process may be also *performance driven*, provided each rewriting rule is equipped with a performance formula, which depends on the particular skeleton implementations for the target language. In such case, proposing a transformation to the user, Meta suggests in which cases the transformation is advantageous, and what is the predicted performance for the transformed program. The prediction is figured out instantiating performance formulae with architecture parameters (e.g. BSP parameters) and basic operations cost (e.g. $t_{Op_+}$, $t_{Op_3}$).

Let us consider the application of SAR-ARA rule. Prototyping Skel-BSP as described in §4.1, thus supposing both scanL and reduce Skel-BSP skeletons are implemented using

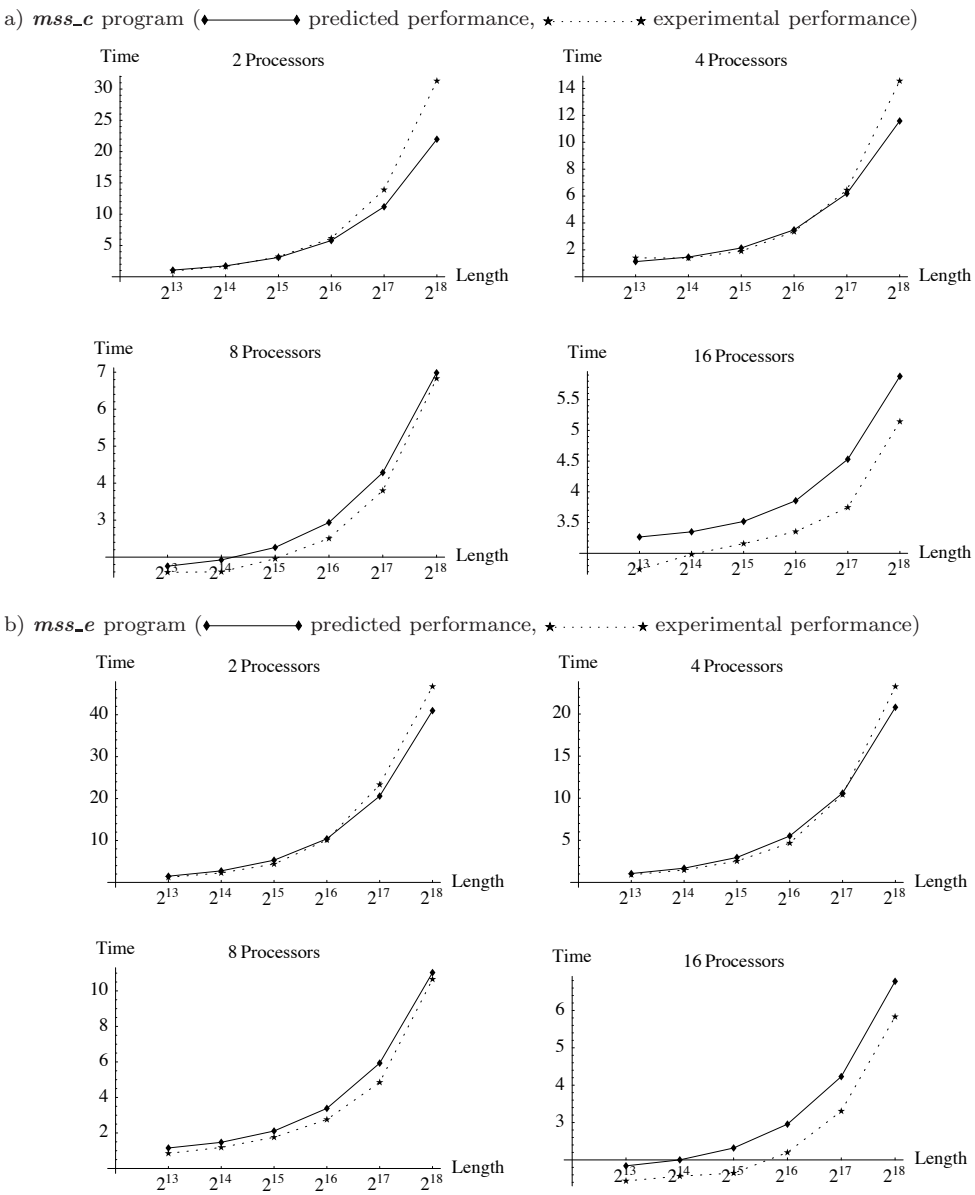



FIG. 4.3. *a)* ***mss_c*** *and b)* ***mss_e****: Comparing predicted performance (solid lines) with experimental performance (dotted lines). Each experiment is performed on several array lengths (x-axis). Four different cluster configurations are experimental (2,4,8,16 processors).*

`TPscanL`, and BSP parameters are assigned as (4.1), the SAR-ARA rule is advantageous when:

$$n < \frac{p^2}{s}\left(-\frac{3\,s}{8}-\frac{g}{2}\right)+\frac{p}{s}\left(\frac{9\,s}{8}+\frac{g}{2}+l\right)=1.6p+654.9p^2 \tag{4.2}$$

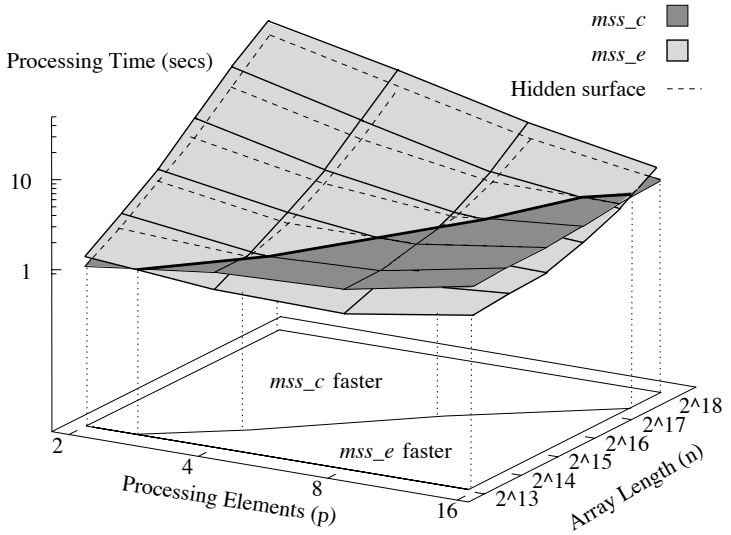



FIG. 4.4. *Experimental performance of* ***mss_c*** *and* ***mss_e*** *programs on several cluster configurations and several array lengths.*

Instantiating this formula with $n$ and $p$, the Meta user may decide for each instance of the problem if the SAR-ARA application is advantageous, i.e. if *mss_e* perform better than *mss_c*. The same decision may be made on real data using Fig. 4.4, which offers another view of data collected running *mss_c* and *mss_e* programs on several cluster configurations and array lengths. In the picture, given a point $(p, n)$ in the $(x, y)$-grid, the best MSS program for that point is the one that belongs to the lower surface in the point. The (interpolated) intersection of the two surfaces is projected on the $(x, y)$-plane.

Finally, to give the flavor of accurateness in performance gain/loss prediction for rules, we compare the predicted and experimental behavior of Skel-BSP SAR-ARA rule. In Fig. 4.5 the $(p, n)$-plane is partitioned by equation (4.2) and by the experimental performance of both *mss_c* and *mss_e*. The picture shows that (4.2) strikingly models the real behavior of the two programs in this case.

Notice that a more effective implementation of reduce would move the border in Fig 4.5 making greater the area where *mss_e* is faster.

**5. Related work and conclusions.** In this paper, we have discussed the design and the implementation of an interactive, graphical transformation tool for skeleton-based languages. The Meta tool is (indeed) language-independent and is easily customizable with a broad class of languages, rewriting rules and cost calculi.

The design of our transformation engine Meta was influenced by the PARAMAT system [18]. However, our approach differs in many aspects. First, our goal is the optimization of high-level parallelism, rather than the parallelization of low-level sequential codes. Second, we do not define (as PARAMAT does) any a priori "good" parallel structure, we rather try to facilitate the exploration of the solution space toward the best parallel structure.

In addition to the described features, Meta may be instantiated with a set pre-defined heuristics to work as semi-automatic optimization tool. As an example Meta recognizes

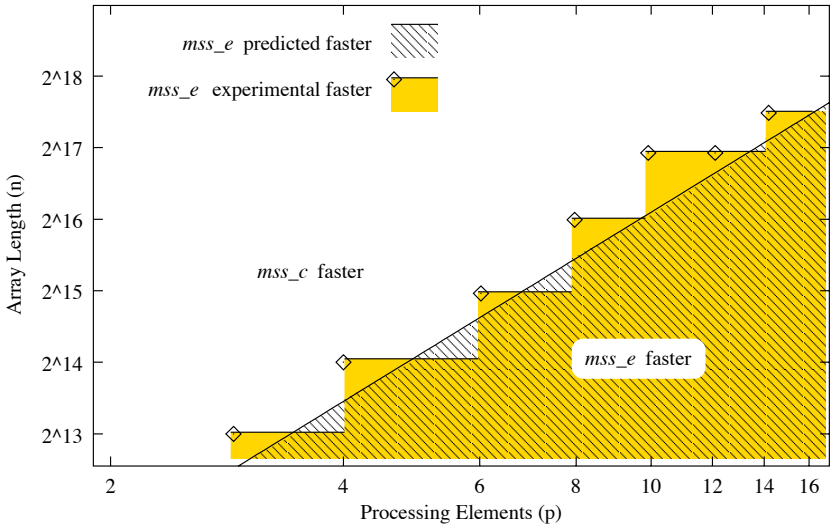



FIG. 4.5. *SAR-ARA rule: Predicted and experimental behavior (**mss_e** is faster than **mss_c** when SAR-ARA is advantageous).*

Skel-BSP data-parallel-free programs and optimizes them with a standard sequence of rewriting rules. Such program formulation (called *normal form*) is proved to be, under mild requirements, the fastest among the semantic-equivalent formulations that can be obtained using the rewriting rules [3].

Meta assists the user in the transformation process also driving it with performance predictions, even if, it is clear that the accurateness of prediction made by Meta primarily depends on the accurateness of the target language cost calculus. The use of Meta with FAN has proved that in many cases good parallel programs can be obtained via transformations [4]. Described experiments (§4.3) on Skel-BSP enforce the accurateness in the prediction of performance gain/loss due to a rule.

We are currently completing the integration of Meta with FAN and we plan to experiment it in the transformation of large real world application structures.

**Acknowledgements.** I am very grateful to Sergei Gorlatch, Christian Lengauer and Susanna Pelagatti for many fruitful discussions.

REFERENCES

[1] M. ALDINUCCI, *The Meta Transformation Tool for Skeleton-Based Languages*, in CMPP2000: Second International Workshop on Constructive Methods for Parallel Programming, S. Gorlatch and C. Lengauer, eds., no. MIP-0007 in University of Passau technical report, July 2000.

[2] M. ALDINUCCI, M. COPPOLA, AND M. DANELUTTO, *Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff*, in CMPP'98: First International Workshop on Constructive Methods for Parallel Programming., S. Gorlatch, ed., no. MIP-9805 in University of Passau technical report, May 1998.

[3] M. ALDINUCCI AND M. DANELUTTO, *Stream parallel skeleton optimization*, in proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT, Boston, USA, Nov. 1999, IASTED/ACTA press.

[4] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti, *Towards parallel programming by transformation: The FAN skeleton framework*, Parallel Algorithms & Applications, Gordon & Breach (Taylor & Francys group), 16(2–3):87–122, 2001.

[5] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, and J. Yang, *Co-ordinating heterogeneous parallel computation*, in Europar '96, L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, eds., Springer-Verlag, 1996, pp. 601–614.

[6] 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 (1995), pp. 225–255.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *SkIE: an heterogeneous HPC environment*, Parallel Computing, 25 (1999), pp. 1827–1852.

[8] R. S. Bird, *Lectures on constructive functional programming*, in Constructive Methods in Computing Science, M. Broy, ed., NATO ASI Series, 1988. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

[9] O. Bonorden, N. Hüppelshäuser, B. Juurlink, and I. Rieping, *PUB Library. User Guide and Function Reference (release 7.0)*, University of Paderborn, Fürstenallee 11, 33102 Padeborn, Germany, Dec. 1999. `http://www.uni-paderborn.de/~pub`.

[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.

[11] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, *Parallel Programming Using Skeleton Functions*, in PARLE'93 Parallel Architectures and Langauges Europe, A. Bode, M. Reeve, and G. Wolf, eds., vol. 694 of LNCS, Springer-Verlag, June 1993.

[12] S. Gorlatch and S. Pelagatti, *A transformational framework for skeletal programs: Overview and case study*, in proceedings of Parallel and Distributed Processing. Workshop held in conjunction with IPPS/SPDP'99, J. Rohlim, ed., vol. 1586 of LNCS, Berlin, 1999, Springer-Verlag, pp. 123–137.

[13] S. Gorlatch, C. Wedler, and C. Lengauer, *Optimization rules for programming with collective operations*, in proceedings of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99), IEEE Computer Society Press, 1999, pp. 492–499.

[14] M. W. Goudreau, J. M. D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas, *A proposal for the BSP worldwide standard library*, tech. report, Oxford University Computing Laboratory, Apr. 1996.

[15] C. M. Hoffmann and M. J. O'Donnell, *Interpreter generation using tree pattern matching*, in Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL'79), New York, USA, Jan. 1979, ACM Press, pp. 169–179.

[16] C. M. Hoffmann and M. J. O'Donnell, *Pattern matching in trees*, Journal of the ACM, 29 (1982), pp. 68–95.

[17] S. R. Kasaraju, *Efficient tree pattern matching*, in Proceedings of the 30th IEEE Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, 1989, IEEE Computer Society Press, pp. 178–183.

[18] C. W. Kessler, *Pattern-driven automatic program transformation and parallelization*, in proceedings of 3rd EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, January 1995.

[19] E. Mäkinen, *On the subtree isomorphism problem for ordered trees*, Information Processing Letters, 32 (1989), pp. 271–273.

[20] T. Rauber and G. Rünger, *A coordination language for mixed task and data parallel programs.*, in proceedings of 3rd Annual ACM Symposium on Applied Computing (SAC'99), ACM Press, 1999, pp. 146–155.

[21] D. B. Skillicorn and W. Cai, *A cost calculus for parallel functional programming*, Journal of Parallel and Distributed Computing, 28 (1995), pp. 65–83.

[22] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–11.

[23] A. Zavanella, *Skeletons and BSP: Performance portability for parallel programming*, PhD thesis, Computer Science Department, University of Pisa, Italy, Mar. 2000.