Compilation of a Skeleton-Based Parallel Language Through Symbolic Cost Analysis and Automatic Data Distribution

Julien Mallet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compilation of a Skeleton-Based Parallel Language Through Symbolic Cost Analysis and Automatic Data Distribution

Julien MALLET

N ° 3436

Mai 1998

THÈME 2

ISSN 0249-6399

Compilation of a Skeleton-Based Parallel Language Through Symbolic Cost Analysis and Automatic Data Distribution

Julien MALLET *

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n3436 — Mai 1998 — 20 pages

Abstract: We present a skeleton-based language which leads to portable and cost-predictable implementations on MIMD computers. The compilation process is described as a series of program transformations. We focus in this paper on the step concerning the distribution choice. The problem of automatic mapping of input vectors onto processors is addressed using symbolic cost evaluation. Source language restrictions are crucial since they permit to use powerful techniques on polytope volume computations to evaluate costs precisely. The approach can be seen as a cross-fertilization between techniques developed within the FORTRAN parallelization and skeleton communities.

Key-words: skeleton-based language, parallelism, cost analysis, automatic data distribution

(Résumé : tsvp)

* mallet@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Compilation d'un langage à patrons parallèles avec analyse de coût symbolique et distribution automatique des données

Résumé : Nous présentons un langage à patrons dont l'implémentation sur les machines MIMD est portable et a un coût prédictible. Le processus de compilation est décrit par une suite de transformations de programmes. Nous nous concentrons, dans ce rapport, sur l'étape du choix de la meilleure distribution des données. Le choix automatique de l'allocation des vecteurs d'entrée sur les processeurs s'appuye sur une évaluation du coût symbolique. Les restrictions du langage source permettent d'utiliser des techniques classiques et puissantes, basées sur les polytopes pour évaluer précisément le coût. Notre approche peut être vue comme un exemple de fertilisation croisée entre les techniques développées dans le domaine de la parallélisation FORTRAN et celles développées dans le domaine des langages à patrons.

Mots-clé : langage à patrons, parallélisme, analyse de coût, distribution de donnée automatique

1 Introduction

A good parallel programming model must be portable and cost predictable. General sequential languages such as FORTRAN achieve portability but cost estimations are often very approximate. The approach described in this paper is based on a restricted language which is portable and allows an accurate cost analysis. The language enforces a programming discipline which ensures a predictable performance on the target parallel computer (there will be no "performance bugs").

Language restrictions are introduced through skeletons which encapsulate control and data flow in the sense of [Col88], [DFH+93]. The skeletons act on vectors which can be nested. There are three classes defined as sets of restricted skeletons: *computation skeletons* (classical data parallel skeletons), *communication skeletons* (data motion over vectors), and *mask skeletons* (conditional data parallel computation). The target parallel computers are MIMD computers with shared or distributed memory. Our compilation process can be described as a series of program transformations starting from a source skeleton-based language to an SPMD-like skeleton-based language. There are three main transformations: in-place updating, making all communications explicit and distribution. Due to space concerns, we focus in this paper only on the step concerning the choice of the distribution.

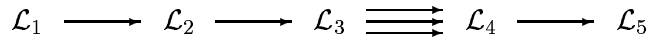
We tackle the problem of automatic mapping of input vectors onto processors using a fixed set of classical distributions (e.g. row block, block cyclic,...). The goal is to determine mechanically the best distribution for each input vector through cost evaluation. The restrictions of the source language and the fixed set of standard data distributions guarantee that the parallel cost (computation + communication) can be computed accurately for all source programs. Moreover, these restrictions allow us to evaluate and compare accurate execution times in a symbolic form. So, the results do not depend on specific vector sizes nor on a specific number of processors.

Even if our approach is rooted in the field of data-parallel and skeleton-based languages, one of its specificities is to reuse techniques developed for FORTRAN parallelization (polytope volume computation) and functional languages (single-threading analysis) in a unified framework.

The article is structured as follows. Section 2 is an overview of the whole compilation process. Section 3 presents the source language and the target parallel language. In Section 4, we describe the symbolic cost analysis through an example: LU decomposition. We report some experiments done on a MIMD distributed memory computer in Section 5. We conclude by a review of related work.

2 Compilation Overview

The compilation process consists of a series of program transformations:



Each transformation compiles a particular task by mapping skeleton programs from one intermediate language into another. The source language ($\mathcal{L}_1$) is composed of a collection of higher-order functions (skeletons) acting on vectors (see Section 3). It is primarily designed for a particular domain where high performance is a crucial issue: numerical algorithms. $\mathcal{L}_1$ is best viewed as a parallel kernel language which is supposed to be embedded in a general sequential language (e.g. C). Only, parts of programs written in $\mathcal{L}_1$ will be executed in parallel whereas others parts will be executed in sequential, for example, on the host computer of the parallel machine.

The first transformation ($\mathcal{L}_1 \rightarrow \mathcal{L}_2$) deals with in-place updating, a standard problem in functional programs with arrays. We rely on a type system in the spirit of [Wad90] to ensure that vectors are manipulated in a single-threaded fashion. The user may have to insert explicit copies in order for his program to be well-typed. As a result, any vector in a $\mathcal{L}_2$ program can be implemented by a global variable.

The second transformation ($\mathcal{L}_2 \rightarrow \mathcal{L}_3$) makes all communications explicit. Intuitively, in order to execute an expression such as `map (λx.x+y)` in parallel, `y` must be broadcast to every processor before applying the function. The transformation makes this kind of communication explicit. In $\mathcal{L}_3$, all communications are expressed through skeletons.

The transformation $\mathcal{L}_3 \rightarrow \mathcal{L}_4$ concerns automatic data distribution. We restrict ourselves to a small set of standard distributions. A vector can be distributed cyclicly, by contiguous blocks or allocated to a single processor. For a matrix (vector of vectors), this gives 9 possible distributions (cyclic cyclic, block cyclic, line cyclic, etc.). The transformation constructs a single vector from the input vectors according to a given data distribution. This means, in particular, that all vector accesses have to be changed according to the distributions. Once the distribution is fixed, some optimizations (such as copy elimination) become possible and are performed on the program. The transformation yields an SPMD-like skeleton program acting on a vector of processors.

In order to choose the best distribution, we transform the $\mathcal{L}_3$ program according to all the possible distributions of its input parameters. The symbolic cost of each transformed program can then be evaluated and the smallest one chosen (see Section 4). For most numerical algorithms, the number of input vectors is small and this

```
LU(M)
     where
     M ::  Vect n (Vect n Float)
     LU(a)                 = iterfor (n-1) (calc.fac.apivot.colrow) a
     calc(i,a,row,piv)     = (i,(map (map first)
                                     .rect i (n-1) (rect i (n-1) fcalc)
                                     .map zip3.zip3)(a,row,piv))
     fcalc(a,row,piv)      = (a-row*piv,row,piv)
     fac(i,a,row,col,piv)  = (i,a,row,(map(map /).map zip.zip)(col,piv))
     apivot(i,a,row,col)   = (i,a,row,col,map (brdcast (i-1)) row)
     colrow(i,a)           = (i,a,brdcast (i-1) a,map (brdcast (i-1)) a)
     zip3(x,y,z)           = map p2t (zip(x,zip(y,z)))
     p2t(x,(y,z))          = (x,y,z)
```

Figure 1: LU decomposition in $\mathcal{L}_1$

approach is practical. In other cases, we would have to rely on the programmer to prune the search space.

The $\mathcal{L}_4 \rightarrow \mathcal{L}_5$ step is a straightforward translation of the SPMD skeleton program to an imperative program with calls to a standard communication library. We currently use C with the MPI library.

Note that all the transformations are automatic. The user may have to interact only to insert copies ($\mathcal{L}_1 \rightarrow \mathcal{L}_2$ step) or, in $\mathcal{L}_4$, when the symbolic formulation does not designate a single best distribution (see 4.4).

3 The Source and Target Languages

The source language $\mathcal{L}_1$ is basically a first-order functional language without general recursion, extended with a collection of higher-order functions (the skeletons). The main data structure is the vector which can be nested to model multidimensional arrays. The $\mathcal{L}_1$ program implementing LU decomposition is given in Figure 1.

The syntax of $\mathcal{L}_1$ is defined in Figure 2. A program is a main expression followed by definitions. A definition is either a function definition or the declaration of the (symbolic) size of an input vector. Combination of computations is done through function composition (.) and the predefined iterator `iterfor`. `iterfor n f a` acts like a loop applying `n` times its function argument `f` on `a`. Further, it makes the current loop index accessible to its function argument.

Prog_1	::=	Exp_1 `where` Decl_1
Decl_1	::=	Decl_1 Decl_1 | $f\,(x_1, ..., x_n) = \text{Exp}_1$ | $a :: \text{Shape}_1$
Shape_1	::=	$(\text{Shape}_1, \ldots, \text{Shape}_1)$ | `Vect` LinF_1 Shape_1 | `Int` | `Float` | `Bool`
Exp_1	::=	Fun_1 $(\text{Exp}_1, \ldots, \text{Exp}_1)$ | $(\text{Exp}_1, \ldots, \text{Exp}_1)$ | x | k
Fun_1	::=	Fun_1 . Fun_1 | `iterfor` LinF_1 Fun_1 | Op_1 | f
	|	CompSkel_1 | CommSkel_1 | MaskSkel_1
Op_1	::=	`+` | ... | `/` | ... | `==` | ... | `zip` | `unzip` | `makearray` LinF_1
LinF_1	::=	$\text{LinF}_1 + \text{LinF}_1$ | $\text{LinF}_1 - \text{LinF}_1$ | $k * x$ | x | k
CompSkel_1	::=	`map` Fun_1 | `fold` Exp_1 Fun_1 | `scan` Exp_1 Fun_1
CommSkel_1	::=	`transfer` LinF_1 LinF_1 | `rotate` LinF_1 | `brdcast` LinF_1
	|	`scatter` LinF_1 | `gather` LinF_1 | `allgather` | `allbrdcast`
MaskSkel_1	::=	`rect` LinF_1 LinF_1 Fun_1 | `trv` LinF_1 LinF_1 Fun_1

$x,\ x_1,\ \ldots\ ,\ x_n \in$ IdentVariable. $k \in$ Constant. $a \in$ IdentVarIn. $f \in$ IdentFunction. IdentVariable $\cap$ IdentFunction $= \emptyset$. IdentVarIn $\subset$ IdentVariable.

Figure 2: Skeleton language $\mathcal{L}_1$.

For handling vectors, there are three classes of skeletons: computation, communication, and mask skeletons. The *computation skeletons* (CompSkel_1) are the classical higher order functions `map`, `fold` and `scan`. The *communication skeletons* (CommSkel_1) describe restricted data motion in vectors. In the definition of `colrow` in LU (Figure 1), `brdcast (i-1)` copies the (i-1)th vector element to all the other elements. Another communication skeleton is `gather i M` which copies the ith column of the matrix M to the ith row of M. All these *communication skeletons* have been chosen because of their availability on parallel computers as hard-wired or optimized communication routines. The *mask skeletons* (MaskSkel_1) are data-parallel skeletons which apply their function argument only to a selected set of vector elements. In the definition of `calc` in LU (Figure 1), `rect i (n-1) fcalc` applies `fcalc` on vector elements whose index ranges from i to (n-1). Note that our approach is not restricted to these sole skeletons: even if it requires some work and caution, more iterators, computation or mask skeletons could be added.

In order to enable a precise symbolic cost analysis, additional syntactic restrictions are necessary. The scalar arguments of communication skeletons, the iterfor operator and mask skeletons must be linear expressions of iterfor indexes and variables of vector size. This restriction is formalized by the nonterminal LinF_1 in $\mathcal{L}_1$ which denotes linear expressions of such variables. We rely on a type system with

simple subtyping (not described here) to ensure that variables x, in LinF_1 expressions, are only iterfor and vector size variables.

It is easy to check that these restrictions are verified in LU. The integer argument of `iterfor` is a linear expression of an input vector size. The arguments of `rect` and `brdcast` skeletons are made of linear expressions of iterfor variable `i` or vector size variable `n`.

The target language $\mathcal{L}_4$ expresses SPMD (Single Program Multiple Data) computations. A program acts on a single vector whose elements represent the data spaces of the processors. Communications between processors are made explicit by data motion within the vector of processors in the spirit of the source parallel language of [DGTJ95]. $\mathcal{L}_4$ introduces new versions of skeletons acting on the vector of processors.

An $\mathcal{L}_4$ program is a composition of computations, with possibly a communication inserted between each computation. The syntax of $\mathcal{L}_4$ is defined Figure 3. Only, the nonterminals which differ from $\mathcal{L}_1$ are described. A program is a main function f followed by definitions. The body of f is the application of a parallel function (FunP_4) to the vector of processors. A parallel function is either the composition of a parallel function, a communication and another parallel function, either a parallel computation `pimap` Fun which applies Fun on each processor, or an iteration `piterfor` LinF FunP which applies LinF times the parallel computation FunP to the vector of processors. The grammar defining Fun is similar to Fun_1 in $\mathcal{L}_1$. The only differences are the mask skeleton arguments which may include modulo, integer division, minimum and maximum functions (as expressed by the nonterminals ExpR_4 and ExpRi_4 in $\mathcal{L}_4$). The nonterminal Comm_4 denotes the communications between processors. It describes data motion in the same way as the communication skeletons of $\mathcal{L}_1$ (CommSkel_1) but on the vector of processors.

Distributing the input vector row-block wise, the implementation of LU becomes a loop whose body is of the form `pimap` F . Com . `pimap` F (Figure 4). The variable `M` in `LUbloc` definition is the distributed matrix. For example, the expression `pimap colrow2` denotes the parallel application of the sequential function `colrow2` to each processor; `a` is a block of rows, `n` denotes the size of the original matrix and `b` the block size (i.e. `n` divided by the number of processors). Before the call of the communication routine `pbrdcast`, each local processor memory is a 3-tuple of the form (number of the broadcasting processor, value to be broadcast, rest of the local memory). The resulting vector of processors is a vector of pair of the form (broadcast value, rest of the local memory). The version of LU with the input matrix distributed

$$
\begin{array}{lcl}
\mathrm{Prog}_4 & ::= & f(x_1,...,x_n) = \mathrm{FunP}_4\ x\ \texttt{where}\ \mathrm{Decl}_4 \\
\mathrm{FunP}_4 & ::= & \mathrm{FunP}_4 \circ \mathrm{Comm}_4 \circ \mathrm{FunP}_4 \\
 & | & \texttt{pimap}\ \ \mathrm{Fun}_4 \mid \texttt{piterfor}\ \ \mathrm{LinF}_4\ \mathrm{FunP}_4 \\
\mathrm{Comm}_4 & ::= & \texttt{ptransfer} \mid \texttt{protate} \mid \texttt{pbrdcast} \\
 & | & \texttt{pscatter} \mid \texttt{pgather} \mid \texttt{pallgather} \mid \texttt{pallbrdcast} \\
\mathrm{MasqSkel}_4 & ::= & \texttt{rect}\ \mathrm{ExpR}_4\ \mathrm{ExpR}_4\ \mathrm{Fun}_4 \mid \texttt{trv}\ \mathrm{ExpR}_4\ \mathrm{ExpR}_4\ \mathrm{Fun}_4 \\
\mathrm{ExpR}_4 & ::= & \texttt{min}(\mathrm{LinF}_1,\ \mathrm{LinF}_4 - \mathrm{ExpRi}_4 * n) \mid \texttt{min}\left(\mathrm{LinF}_4, \left\lfloor \frac{\mathrm{LinF}_4 - \mathrm{ExpRi}_4}{p} \right\rfloor\right) \\
 & | & \texttt{max}(\mathrm{LinF}_4,\ \mathrm{LinF}_4 - \mathrm{ExpRi}_4 * n) \mid \texttt{max}\left(\mathrm{LinF}_4, \left\lfloor \frac{\mathrm{LinF}_4 - \mathrm{ExpRi}_4}{p} \right\rfloor\right) \\
\mathrm{ExpRi}_4 & ::= & \texttt{div}(\mathrm{Expi}_4,\ p) \mid \texttt{mod}(\mathrm{Expi}_4,\ p) \mid ip
\end{array}
$$

$n \in$ SizeIdent. $p \in$ SizeProcIdent. $ip \in$ IdentProc. $k \in$ Constant. These sets are mutually disjoint. $x, x_1, ..., x_n \in$ IdentVariable.

Figure 3: Skeleton language $\mathcal{L}_4$

row-cyclic wise is given in Figure 5. The only differences with the row-block version are the bounds of outer mask skeleton `rect` in function `calc` and the indexes involved in the broadcast communication in the functions `colrow2` and `colrow1`.

4 Accurate Symbolic Cost Analysis

After transformation of the program according to different distributions, this step aims at automatically evaluating the complexity of each $\mathcal{L}_4$ program obtained in order to choose the most efficient. Our approach to get an accurate symbolic cost is to reuse results on polytope volume computations ([Taw94], [Cla96]). It is possible because the restrictions of the source language and the fixed set of data distributions guarantee that the abstracted cost of all transformed source programs can be translated into a polytope volume description.

First, an abstraction function $\mathcal{CA}$ takes the program and yields its symbolic parallel cost. The cost expression is transformed so that it can be seen as the definition of a polytope volume. Then, classic methods to compute the volume of a polytope are applied to get a symbolic cost in polynomial form. Finally, a symbolic math package (such as Maple) can be used to compare symbolic costs and find the smallest cost among the $\mathcal{L}_4$ programs corresponding to the different distribution choices.

```
LUbloc(M,n,b)
   = piterfor (n-1)
              (pimap (calc.fac.apivot.colrow1).pbrdcast.pimap colrow2) M
    where
    calc(ip,i,a,row,piv)
     = (i,(map (map first)
          .rect max(0,i-ip*b) min(b-1,n-1-ip*b+b) (rect i (n-1) fcalc)
          .map zip3.zip3) (a,row,piv))
    fcalc(a,row,piv)       = a-row*piv
    fac(ip,i,a,row,col,piv)
     = (ip,i,a,row,(map(map /).map zip.zip) (col,piv))
    apivot(ip,i,a,row,col) = (ip,i,a,row,map (brdcast (i-1)).copy a)
    colrow1(ip,(buf,i,a,row,col))
     = (ip,i,a,brdcast mod(i-1,b).update mod(i-1,b) buf row,
        map (brdcast (i-1)).copy a,col)
    colrow2(ip,(i,a))
     = (div(i-1,b),a!mod(i-1,b),(i,a,copy a,map (brdcast (i-1)).copy a))
    zip3(x,y,z)
     = map p2t (zip(x,zip(y,z)))
    p2t(x,(y,z))
     = (x,y,z)
```

Figure 4: Row block distribution version of LU expressed in $\mathcal{L}_4$

```
LUcyc(M,n,b)
   = piterfor (n-1)
              (pimap (calc.fac.apivot.colrow1).pbrdcast.pimap colrow2) M
    where
    calc(ip,i,a,row,piv)
     = (i,(map (map first)
          .rect max(0,div(i-ip,p)) min(b-1,div(n-1-ip,p))
               (rect i (n-1) fcalc)
          .map zip3.zip3) (a,row,piv))
    fcalc(a,row,piv)       = a-row*piv
    fac(ip,i,a,row,col,piv)
     = (ip,i,a,row,(map(map /).map zip.zip) (col,piv))
    apivot(ip,i,a,row,col) = (ip,i,a,row,map (brdcast (i-1)).copy a)
    colrow1(ip,(buf,i,a,row,col))
     = (ip,i,a,brdcast div(i-1,p).update div(i-1,p) buf row,
        map (brdcast (i-1)).copy a,col)
    colrow2(ip,(i,a))
     = (mod(i-1,p),a!div(i-1,p),(i,a,copy a,map (brdcast (i-1)).copy a))
    zip3(x,y,z)
     = map p2t (zip(x,zip(y,z)))
    p2t(x,(y,z))
     = (x,y,z)
```

Figure 5: Row cyclic distribution version of LU expressed in $\mathcal{L}_4$

4.1 Cost Abstraction and Cost Language $\mathcal{LC}_1$

The symbolic cost abstraction $\mathcal{CA}$ is a function which extracts cost information from parallel programs. The main rules are shown in Figure 6. We use a non-standard notation for indexed sums: instead of $\sum_{i=1}^{n}$, we note $\sum_i \left\{ {1 \le i \atop i \le n} \right\}$. This notation is needed because the transformation to polytopes will introduce new inequalities over sum variables which cannot be expressed in the standard notation. Communication costs are expressed as polynomials whose constants depend on the target computer. For example, the cost of `pbrdcast` involves the parameters $\alpha_{\mathtt{br}}$ and $\beta_{\mathtt{br}}$ which denote respectively the time of one-word transfer between two processors and the message startup time on the parallel computer considered. One just has to set those constants to adapt the analysis for a specific parallel machine.

$\mathcal{CA}$ [[f1 . f2]]	=	$\mathcal{CA}$ [[f1]] + $\mathcal{CA}$ [[f2]]	
$\mathcal{CA}$ [[`piterfor` e (λi.f)]]	=	$\sum_i \left\{ {1 \le i \atop i \le \mathtt{e}} \right\}$ $\mathcal{CA}$ [[f]]	
$\mathcal{CA}$ [[`pimap` (λi.f)]]	=	$\max_{i=0}^{p-1}$ $\mathcal{CA}$ [[f]]	where p is the number of processors
$\mathcal{CA}$ [[`pbrdcast`]]	=	$(\alpha_{\mathtt{br}} * b + \beta_{\mathtt{br}}) * p$	where $\begin{cases} p \text{ is the number of processors} \\ b \text{ is the size of broadcast data} \end{cases}$
$\mathcal{CA}$ [[`rect` e1 e2 f]]	=	$\sum_i \left\{ {\mathtt{e1} \le i \atop i \le \mathtt{e2}} \right\}$ $\mathcal{CA}$ [[f]]	

Figure 6: Symbolic cost abstraction (extract)

From the grammar of $\mathcal{L}_4$ (Figure 3) and the definition of $\mathcal{CA}$, it is easy to show that the cost expressions belong to the language $\mathcal{LC}_1$ shown in Figure 7. $\mathcal{LC}_1$ is made of generalized sums (G-sum) and generalized maxima (G-max). A G-max denotes the maximum expression one can get when the max variable ranges over the integer interval. Inequalities in G-sum may involve maximum and minimum functions. The variables of polynomials are only denoting vector sizes (n) or the number of processors (p). In order to illustrate the analysis, we consider here only the two best distribution choices for LU: row block and row cyclic. The abstracted costs are given below; note that only the part of the costs which differs is detailed:

$$\mathcal{C}^1_{bloc} \equiv \sum_i \left\{ {1 \le i \atop i \le n-1} \right\} \left(\max_{i_p=0}^{p-1} \sum_j \left\{ {\mathtt{max}(0, i - i_p * b) \le j \atop j \le \mathtt{min}(b-1, n-1-i_p*b+b)} \right\} \left(\sum_k \left\{ {i \le k \atop k \le n-1} \right\} 1 \right) \right) + C$$

$$\mathcal{C}^1_{cyc} \equiv \sum_i \left\{ {1 \le i \atop i \le n-1} \right\} \left(\max_{i_p=0}^{p-1} \sum_j \left\{ {\mathtt{max}(0, \left\lfloor \frac{i-i_p}{p} \right\rfloor) \le j \atop j \le \mathtt{min}(b-1, \left\lceil \frac{n-1-i_p}{p} \right\rceil)} \right\} \left(\sum_k \left\{ {i \le k \atop k \le n-1} \right\} 1 \right) \right) + C$$

$$
\begin{array}{lcl}
\mathrm{CPrg}_1 & ::= & \mathrm{CPrg}_1 + \mathrm{CPrg}_1 \mid \mathrm{CPrgS}_1 \\
\mathrm{CPrgS}_1 & ::= & \sum_{i\ \{\mathrm{Ineq}_1 \wedge \mathrm{Ineq}_1\}} \mathrm{CPrgS}_1 \mid \max_{ip=0}^{p} \mathrm{CPrgS}_1 \mid \mathrm{Poly}_1 \\
\mathrm{Ineq}_1 & ::= & i \leq \mathrm{MinExp}_1 \mid \mathrm{MaxExp}_1 \leq i \mid \mathrm{LinF}_1 \leq \mathrm{LinF}_1 \\
\mathrm{MinExp}_1 & ::= & \mathtt{min}(\mathrm{LinF}_1,\ \mathrm{LinF}_1 - \mathrm{ExpRi}_1 * n) \mid \mathtt{min}\left(\mathrm{LinF}_1, \left\lceil \frac{\mathrm{LinF}_1 - \mathrm{ExpRi}_1}{p} \right\rceil\right) \\
\mathrm{MaxExp}_1 & ::= & \mathtt{max}(\mathrm{LinF}_1,\ \mathrm{LinF}_1 - \mathrm{ExpRi}_1 * n) \mid \mathtt{max}\left(\mathrm{LinF}_1, \left\lfloor \frac{\mathrm{LinF}_1 - \mathrm{ExpRi}_1}{p} \right\rfloor\right) \\
\mathrm{ExpRi}_1 & ::= & \mathtt{div}(\mathrm{ExpRi}_1,\ p) \mid \mathtt{mod}(\mathrm{ExpRi}_1,\ p) \mid ip \\
\mathrm{LinF}_1 & ::= & \mathrm{LinF}_1 + \mathrm{LinF}_1 \mid \mathrm{LinF}_1 - \mathrm{LinF}_1 \mid k * x \mid x \mid k \\
\mathrm{Poly}_1 & ::= & \mathrm{Poly}_1 + \mathrm{Poly}_1 \mid \mathrm{Poly}_1 - \mathrm{Poly}_1 \mid k * \mathrm{Pvar}_1 \\
\mathrm{Pvar}_1 & ::= & \mathrm{Pvar}_1 * \mathrm{Pvar}_1 \mid p \mid n
\end{array}
$$

$n \in$ SizeIdent. $p \in$ SizeProcIdent. $i \in$ IndexIdent. $ip \in$ ProcIdent. $k \in$ Constant.
These sets are mutually disjoint. $x \in$ SizeIdent $\cup$ SizeProcIdent $\cup$ IndexIdent.

Figure 7: Cost language $\mathcal{LC}_1$.

We emphasize that writing the source program in $\mathcal{L}_1$ is crucial to get an accurate symbolic cost. First of all, without a severe limitation of the use of recursion no precise cost could be evaluated in general. Further, the restrictions imposed by $\mathcal{L}_1$ ensure that every abstract parallel cost belongs to $\mathcal{LC}_1$. The mask skeletons (which limit conditional application to index intervals), the communication and the computation skeletons all have a complexity which depends polynomially on the vector size. Their costs can be described by nested sums. Another important restriction is that expressions involving iteration indexes (mask skeletons and iterfor bounds) are linear. This restriction, expressed by the nonterminal LinF_1 in the definition of $\mathcal{L}_1$, in addition to the form of the standard distributions which keep the linearity of the vector accesses ensure that the inequalities occurring in $\mathcal{LC}_1$ are linear.

4.2 Transformation to Descriptions of Polytope Volume

Polytope volumes are only defined in terms of nested G-sums with linear inequalities containing neither max nor min as described in [Cla96]. In order to apply methods to compute polytope volumes, we must remove the G-max, min, max occurring in the cost expressions.

The transformation takes the form of a structural recursive scan of the cost expression. The min and max appearing in inequalities are transformed into conjunctions of inequalities. For example, the expression $\mathtt{max}(0, i - i_p * b) \leq j$, in $\mathcal{C}^1_{bloc}$, becomes $0 \leq j \wedge i - i_p * b \leq j$. G-max expressions are propagated through their su-

bexpressions. For example, in $\mathcal{C}^1_{bloc}$, $\max_{i_p=0}^{p-1} \sum_j \left\{ {}^{0\le j\wedge i-i_p*b\le j}_{j\le b-1\wedge j\le n-1-i_p*b+b} \right\}$ (...) is temporarily transformed into $\sum_j \left\{ {}^{0\le j\wedge \min_{i_p=0}^{p-1}(i-i_p*b)\le j}_{j\le b-1\wedge j\le \max_{i_p=0}^{p-1}(n-1-i_p*b+b)} \right\} \max_{i_p=0}^{p-1}$ (...). The max value of a G-sum is a sum where the upper bounds are maximized and the lower ones minimized. For a sum inequality $i \le e$, we propagate the G-max inside e until it reaches a variable x or a constant. We can finally use the facts that $\max_{i_p=0}^{p-1} i_p = p-1$ and $\max_{i_p=0}^{p-1} k = k$ if $k \not\equiv i_p$. For an inequality $e \le i$, the transformation propagates a G-min analogously. Since no polynomial (Poly_1 in $\mathcal{LC}_1$) may contain a G-max index, the G-max of a polynomial is just this polynomial.

The grammar of the resulting cost language $\mathcal{LC}_2$ appears in Figure 8. Only, the nonterminals which differ from $\mathcal{LC}_1$ are described.

CPrgS_2	::=	$\sum_{(i_1,\ldots,i_m)}{}^{\{\text{Ineq}_2\}}\ \text{Poly}_2$
Ineq_2	::=	$\text{Ineq}_2 \wedge \text{Ineq}_2 \mid i \le \text{ExpR}_2 \mid \text{ExpR}_2 \le i \mid \text{LinF}_2 \le \text{LinF}_2$
ExpR_2	::=	$\left\lceil \frac{\text{LinF}_2}{p} \right\rceil \mid \left\lfloor \frac{\text{LinF}_2}{p} \right\rfloor$

Figure 8: Cost Language $\mathcal{LC}_2$.

In the case of distributed LU, the cost expressions are transformed into

$$\begin{aligned} \mathcal{C}^1_{bloc} &= \sum_{(i,j,k)} \left\{ {}^{1\le i\le n-1}_{0\le j\wedge i-(p-1)*b\le j\wedge i\le k\le n-1}_{j\le b-1\wedge j\le n-1+b} \right\} 1+C \equiv \mathcal{C}^2_{bloc} \\ \mathcal{C}^1_{cyc} &= \sum_{(i,j,k)} \left\{ {}^{1\le i\le n-1\wedge 0\le j\wedge \left\lfloor \frac{i-p+1}{p} \right\rfloor \le j}_{j\le b-1\wedge j\le \left\lceil \frac{n}{p} \right\rceil -1\wedge i\le k\le n-1} \right\} 1+C \equiv \mathcal{C}^2_{cyc} \end{aligned}$$

The technique described to remove G-max expressions may maximize the real cost. This is due to the duplication of G-max in G-sums. A more complicated but accurate symbolic solution also exists. The idea is to evaluate (as described in the next section) the nested G-sums first. When the G-sum is reduced to a polynomial the problem amounts to computing symbolic maximum of polynomials over symbolic intervals. This can be done using symbolic differentiation, solution and comparison of polynomials. At this point, we have only used the simpler approximated solution because the approximation is minor for our examples (e.g. none for LU) and the accurate symbolic solution is not implemented as such in existing symbolic math packages.

4.3 Parametrized Polytope Volume Computation

A parametrized polytope is a set of points whose coordinates satisfy a system of linear inequalities with possibly unknown parameters. After the previous transformation, the cost expression denotes polytope volume, that is the number of points in the associated polytope. For example, $\mathcal{C}^2_{bloc}$ is the number of points (i, j, k) which satisfy the system shown in Figure 9(a).

(a) $\begin{cases} 1 \leq i \leq n-1 \\ 0 \leq j,\ i-(p-1)*b \leq j \\ j \leq b-1,\ j \leq n-1+b \\ i \leq k \leq n-1 \end{cases}$ (b) $\begin{cases} 1 \leq i \leq n-1 \\ 0 \leq j,\ x \leq j,\ j \leq b-1 \\ i \leq k \leq n-1 \\ p*(x-1) < i-p+1 \leq p*x \end{cases}$

Figure 9: Inequality systems associated to $\mathcal{C}_{bloc}$ (a), $\mathcal{C}_{cyc}$ (b).

[Taw94], [Cla96] describe algorithms for computing symbolically the volume of parametrized polytopes. The result is a polynomial whose variables are the system parameters. Further, Clauss [Cla96] presents an extension for systems of non-linear inequalities with floor ($\lfloor\ \rfloor$) and ceiling ($\lceil\ \rceil$) functions. The technique consists in first transforming the non-linear inequalities to conjunction of linear inequalities and then applying the volume computation. If an expression of the form $\left\lfloor \frac{e}{p} \right\rfloor$ appears in an inequality, a new free variable x and a new inequality $p*x \leq e < p*(x+1)$ are introduced, and every occurrence of $\left\lfloor \frac{e}{p} \right\rfloor$ is replaced by the variable x. For ceiling functions, the transformation is similar.

Thus, the algorithms of [Taw94] and [Cla96] allow to transform each parallel cost expression into a polynomial. This method applied to LU yields:

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{C}^2_{bloc} & = & n^3\left(\frac{3p^2-1}{6p^3}\right) - \frac{n^2}{2p} + \frac{n}{6p} + C & \equiv & \mathcal{C}^3_{bloc} \\ \mathcal{C}^2_{cyc} & = & \frac{n^3}{3p} + n^2\left(\frac{p-2}{2p}\right) + n\left(\frac{-3p^2+6p-2}{6p}\right) + \frac{p^2-3p+2}{6} + C & \equiv & \mathcal{C}^3_{cyc} \end{array}$$

4.4 Symbolic Cost Comparison

The last step is to compare the symbolic costs of different distribution choices. It amounts to computing the symbolic intervals where the difference of cost (polynomial) is positive or negative. Symbolic math packages such as Maple can be used for solving this problem. In the case of LU, Maple produces the following condition:

$$\mathcal{C}^3_{bloc} - \mathcal{C}^3_{cyc} \geq 0 \Leftrightarrow n \geq p$$

The programmer may have to indicate if the relations given by Maple are satisfied or not. In our example, he must indicate if $n \geq p$. Another (automatic) solution is to use the relations as run-time tests which choose between several versions of the program.

In our example, the difference in the two costs can be explained by the fact that the cyclic distribution provides a much better load balancing than the block distribution whereas communications are identical.

5 Experiments

We have performed experiments on an Intel Paragon XP/S with a handful of standard linear algebra programs (LU, Cholesky factorization, Householder, Jacobi elimination, ...). Our implementation is not completed and some compilation steps, such as the destructive update step and part of the symbolic cost computation, were done manually. Figure 10 gathers the execution times obtained for LU decomposition, Cholesky factorization and Householder. They are representative of the results we got for the other few programs. For all programs, the distribution chosen by the cost analysis proved to be the best one in practice.

We compared the sequential execution of skeleton programs with standard (and portable) C versions taken from [PTVF86] and our parallel implementation with High Performance Fortran (a manual distribution approach). No significant sequential or parallel runtime penalty seems to result from programming using skeletons, at least for such regular algorithms.

We compared our code with the parallel implementation of NESL, a skeleton-based language [BCH+93]. The work on the implementation of NESL has mostly been directed towards SIMD machines. On the Paragon, the NESL compiler distributes vectors uniformly on processors and communications are not optimized. Not surprisingly, the parallel code is very inefficient (at least fifty times slower than our code).

We also compared our implementation with ScaLAPACK, an optimized library of linear algebra programs designed for distributed memory MIMD parallel computers [CD95]. In ScaLAPACK, the user may explicitly indicate the data distribution. So, we indicated the best distribution found by the cost analysis in each ScaLAPACK program considered. If our code on 1 processor is much slower than its ScaLAPACK equivalent (between 3 to 6 times slower), the difference decreases as the number of processors increases (typically, 1.8 times slower on 32 processors). Much of this difference comes from the machine specific routines used by ScaLAPACK for performing

Processors	Skel. cyclic	Skel. bloc	C Seq.	HPF cyclic	ScaLAPACK cyclic
1	14.77	15.07	13.61	15.36	3.78
2	8.43	10.71	×	8.67	2.4
4	5.25	6.75	×	5.41	1.84
8	3.23	5.50	×	3.38	1.66
16	2.97	5.33	×	3.06	1.50
32	2.57	5.58	×	2.67	1.41

(a) LU decomposition for a 512x512 matrix

Processors	Skel. cyclic	C Seq.	ScaLAPACK cyclic	ScaLAPACK bloc cyclic
1	67.45	53.17	55.80	11.41
2	35.54	×	34.95	5.93
4	21.35	×	21.80	3.60
8	14.81	×	15.56	2.11
16	11.55	×	12.56	1.39
32	9.91	×	11.32	0.92

(b) Cholesky factorization for a 1024x1024 matrix

Processors	Skel.	C Seq.	ScaLAPACK
1	318.16	308.17	56.35
2	159.04	×	35.23
4	82.12	×	22.57
8	44.59	×	16.27
16	26.68	×	12.82
32	17.98	×	10.83

(c) Householder for a 1024x1024 matrix

Figure 10: Execution Time on Paragon XP/S

matrix operations (the BLAS library). This suggests a possible interesting extension of our source language. The idea would be to introduce new skeletons corresponding to the BLAS operations in order to benefit from these machine specific routines.

Note that ScaLAPACK allows block cyclic distributions with a variable size of blocks which are a more general form of distribution than ours. This enables the programmer to sometimes find a better compromise between communication costs and load balancing by guessing the right block size. This is the case for the Cholesky factorization (Figure 10(c)). The ScaLAPACK program with a block cyclic distribution is always faster than the cyclic skeleton one (between 6 to 10 times faster). Indeed, the optimal distribution for Cholesky is not cyclic but block cyclic with the block size greater than one and smaller than the vector size divided by the number of processors. We believe that block cyclic distributions (with a variable size of blocks) could fit within our framework. It is clear that an exhaustive analysis of all possible distributions would become unrealistic in this case. However, a symbolic cost analysis would remain of interest for a programmer who hesitates between two specific block cyclic distributions.

These preliminary results are promising but more experiments are necessary to assess both the expressiveness of the language and the efficiency of the compilation. We believe that these experiments may also indicate useful linguistic extensions (e.g. new skeletons) and new optimizations.

6 Related Work and Conclusion

The community interested in the automatic parallelization of FORTRAN has studied automatic data distribution through parallel cost estimation ([GB92], [CGST93]). If the complete FORTRAN language (unrestricted conditional, indexing with runtime value, ...) is to be taken into account, communication and computation costs cannot be accurately estimated. In practice, the approximated cost may be far from the real execution time leading to a bad distribution choice. [Taw94], [Cla96] focus on a subset of FORTRAN: loop bound and array indexes are linear expressions of the loop variables. This restriction allows them to compute a precise symbolic computation cost through their computations of polytope volume. Unfortunately, using this approach to estimate communication costs is not realistic. Indeed, the cost would be expressed in terms of point-to-point communications without taking into account hard-wired communication primitives. Working on the same FORTRAN subset, Feautrier [Fea94] points out this fact and considers only a rough estimation

of communication costs. All these works estimate real costs too roughly to ensure that a good distribution is chosen.

The skeleton community has studied the transformation of restricted computation patterns into lower-level parallel primitives. [DFH+93] defines a restricted set of skeletons which are transformed using cost estimation. Only cost-reducing transformations are considered. It is, however, well known that often intermediary cost-increasing transformations are necessary to derive a globally cost optimal algorithm. [SC93], [JCSS97] and [Ran96] define cost analysis for skeleton-based languages. Their skeletons are less restricted than ours leading to approximate parallel cost (communication or/and computation). Furthermore, the costs are not symbolic (the size of input matrices and the number of processors are supposed to be known). [GL98] define precise communication cost for scan and fold skeleton on several parallel topologies (hypercube, mesh, ...) which allows them to apply optimization of communications through cost-reducing transformations. There are also a few real parallel implementations of skeleton-based languages. NESL [BCH+93] is a nested vector language. Its compilation is specialized to SIMD computer and, so, the execution on distributed memory machine is not efficient. [Bra93] uses cost estimations based on profiling which does not ensure a good parallel performance for different sizes of inputs. [Pel93] uses a finer cost estimation but implementation decisions are taken locally and no arbitration of tradeoffs is possible.

We have presented in this paper the compilation of a skeleton-based language for MIMD computers. Working by program transformations in a unified framework simplifies the correctness proof of the implementation. One can show independently for each step that the transformation preserves the semantics and that the transformed program respects the restrictions enforced by the target language. The overall approach can be seen as promoting a programming discipline whose benefit is to allow precise analyses and a predictable parallel implementation. The source language restrictions are central to the approach as well as the techniques to evaluate the volume of polytopes. We regard this work as a rare instance of cross-fertilization between techniques developed within the FORTRAN parallelization and skeleton communities. A possible research direction is to study dynamic redistributions chosen at compile-time. Some parallel algorithms (e.g. Alternative-Direction-Implicite Integration) are much more efficient in the context of dynamic data redistribution. A completely automatic and precise approach to this problem would be possible in our framework. However, this would lead to a search space of exponential size. A possible solution to this problem is to consider (high-level) interactions with the user.

Acknowledgements : Thanks to Pascal Fradet, Daniel Le Métayer, Mario Südholt for commenting on an earlier version of this paper.

References

[BCH+93] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *4th ACM Symp. on Princ. and Practice of Parallel Prog.*, pages 102–112. 1993.

[Bra93] T. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *5th Int. Workshop on the Imp. of Fun. Lang.*, pages 23–33, 1993.

[CD95] J. Choi and J. J. Dongarra. Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 170–177, 1995.

[CGST93] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel program. In *20th ACM Symp. on Princ. of Prog. Lang.*, pages 16–28, 1993.

[Cla96] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, 1996.

[Col88] M. Cole. A skeletal approach to the exploitation of parallelism. In *CONPAR'88*, pages 667–675. Cambridge University Press, 1988.

[DFH+93] 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*, pages 146–160. LNCS 694, 1993.

[DGTJ95] J. Darlington, Y. K Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *5th ACM Symp. on Princ. and Practice of Parallel Prog.*, pages 19–28, 1995.

[Fea94] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[GL98] Sergei Gorlatch and Christian Lengauer. (De)Composition rules for parallel scan and reduction. In *Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97)*. IEEE Computer Press, 1998.

[JCSS97] C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97 Parallel Processing*, pages 650–661. LNCS 1300, 1997.

[Pel93] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Pise University, 1993.

[PTVF86] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1986.

[Ran96] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model.* PhD thesis, Edinburgh University, 1996.

[SC93] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. Technical report, Queen's University, 1993.

[Taw94] N. Tawbi. Estimation of nested loops execution time by integer arithmetic in convex polyhedra. In *Int. Symp. on Par. Proc.*, pages 217–223, 1994.

[Wad90] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, pages 561–581. North Holland, 1990.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399