Journal of Parallel and Distributed Computing **61**, 501–519 (2001)
doi:10.1006/jpdc.2000.1683, available online at http://www.idealibrary.com on IDEAL®

Alignment and Distribution Is Not (Always) NP-Hard[1]

Vincent Boudet, Fabrice Rastello, and Yves Robert

LIP, URA CNRS 1938, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
E-mail: Vincent.Boudet@ens-lyon.fr, Fabrice.Rastello@ens-lyon.fr, Yves.Robert@ens-lyon.fr

Received August 11, 1998; accepted September 1, 2000

In this paper, an efficient algorithm to simultaneously implement *array alignment* and *data/computation distribution* is introduced and evaluated. We revisit previous work of J. Li and M. Chen (*in* "Frontiers 90: The Third Symposium on the Frontiers of Massively Parallel Computation," pp. 424–433, College Park MD, Oct. 1990; and *J. Parallel Distrib. Comput.* **13** (1991), 213–221), and we show that their alignment step should not be conducted without preserving the potential parallelism. In other words, the optimal alignment may well sequentialize computations, whatever the distribution afterward. We provide an efficient algorithm that handles alignment and data/computation distribution simultaneously. The good news is that several important instances of the whole alignment /distribution problem have polynomial complexity, while alignment itself is NP-complete (Li and Chen, 1990).

Key Words: compilation techniques; parallel loops; alignment; distribution; "the owner computes" rule.

1. INTRODUCTION

Compile-time techniques for mapping arrays and computations onto distributed memory machines have focused a large research effort recently, as illustrated by the survey paper of Ayguadé *et al.* [5]. Several methods and tools have been presented since the reference papers of Li and Chen [13, 14], who studied the problem of aligning arrays so as to minimize communications. Because Li and Chen have shown the alignment problem to be NP-complete (in the number of data arrays and statements within the loop nest), heuristics or costly (exponential) algorithms, such as Integer Linear Programming, have been introduced. We briefly survey the related literature in Section 2.

[1] This work was supported by the CNRS ENS Lyon-INRIA project *ReMaP* and by the Eureka Project *EuroTOPS*.

0743-7315/01 $35.00

In this paper we revisit previous work of Li and Chen [13, 14], and we show that their alignment step should not be conducted without preserving the potential parallelism. In other words, the optimal alignment may well sequentialize computations, whatever the distribution afterward. We provide an efficient algorithm that handles alignment and data/computation distribution simultaneously. The good news is that several important instances of the whole alignment–distribution problem have polynomial complexity, while alignment itself is NP-complete [13].

We take as input a loop nest, possibly nonperfect, where parallelism has been made explicit, e.g., after applying the Allen and Kennedy parallelization algorithm [2]. We construct a new graph, the *alignment–distribution* graph, which replaces Li and Chen's component affinity graph. Using this graph, we are able to determine which parallel loop(s) and which array dimension(s) should be distributed to the processors so as to preserve parallelism while minimizing communications. Our alignment–distribution graph is weighted, and the weights represent estimates of the communication costs: it is a very flexible approach, and we are able to take advantage of recent results on modeling such communication costs accurately [3, 4, 7, 11]. Because the choice of the distributed loops provides kind of a "reference" pattern, the alignment step is conducted according to this choice, and the complexity to finding the optimal solution reduces to a fast (polynomial) path algorithm on the alignment–distribution graph. This is a very nice result for the practical applicability of our approach (again, previous techniques aimed at solving a NP-complete problem).

The paper is organized as follows: we start with a motivating example in Section 2. We use the example to summarize the approach of Li and Chen [13, 14] and to point out its limitations. We briefly review the existing literature in Section 3. We describe our new algorithm, and we state complexity results, in Section 4. We give some final remarks in Section 5.

2. MOTIVATION

We use a simple example to explain why aligning arrays and distributing parallel loops should be dealt with simultaneously.

EXAMPLE 1

```
for i = 2 to n do
    for // j = i + 1 to n do
            S1: a(i, j) = b(i, j) + a(i − 1, j)
            S2: b(j, i) = a(j, j) + 1
    end for //
end for
```

To check that the second loop on j is indeed parallel, we can use a dependence analysis tool like Tiny [18]. Using such a tool, we check that there is only one flow dependence of level 1 from S_1 to itself, which is due to a. The reduced dependence graph for Example 1 is depicted in Fig. 1.

FIG. 1. The reduced dependence graph (using dependence levels) for Example 1.

First we review Li and Chen's approach [13, 14] through Example 1. Then we explain why their technique may kill the potential parallelism.

2.1. Li and Chen's Component Affinity Graph

We represent in Fig. 2 the *component affinity graph* (CAG) that Li and Chen [13, 14] would derive for Example 1. We informally explain how the CAG is built using the example. The CAG contains two columns of two nodes, because they are two arrays a and b (hence two columns) of dimension 2 each (hence two nodes in each column). Node $a1$ represents the first dimension of array a, and so on. There is an edge between two nodes, i.e., between two dimensions of different arrays, if, roughly speaking, the subscripts of these dimensions are the same up to a translation by a constant, and if these arrays appear on both sides of the same assignment. The CAG is undirected. Self-references are not taken into account. In our example, there is an edge between nodes $a1$ and $b1$ because of statement S_1: the same subscript i appears in the first dimension of a and b. In general, when the same subscript, up to a translation by a constant, appears in dimension i_x of array x and in dimension i_y of array y, these two dimensions are said to have an affinity relationship, and we draw an edge between the corresponding nodes. Similarly, due to S_1 again, there is an edge between $b2$ and $a2$. Because self-references are not taken into account, the occurrence of $a(i-1, j)$ on the right-hand side has no impact on the graph. The intuitive idea is that edges imply an alignment preference between the corresponding arrays. The term *alignment* may well be understood here as an HPF ALIGN directive [10] onto a virtual template. Aligning arrays according to the edges will reduce, or even suppress (as in statement S_1), the possible communications induced by the distribution of the arrays onto parallel processors.

Statement S_2 introduces some complication, because the same index j appears in the first dimension of a on the left-hand side, and in both dimensions of b on the right-hand side. The two edges $(a1, b1)$ and $(a1, b2)$ are said to be competing.

The CAG is weighted: edges are valued according to the strength of preference. A competing edge has weight ε, a value much smaller than 1. The weight of an edge between nodes indexed by a spatial variable (a subscript of a parallel loop, like j in Example 1) is 1. Finally, the weight of an edge between nodes indexed by a temporal variable (a subscript of a sequential loop, like i in Example 1) is ∞. We are led to the graph of Fig. 2. If there are several edges between two nodes, we only keep one, whose weight is the sum of all edge weights between the two nodes.

Li and Chen [13, 14] state the alignment problem as follows: partition the nodes of all columns into disjoint subsets that represent aligned dimensions. The rule of

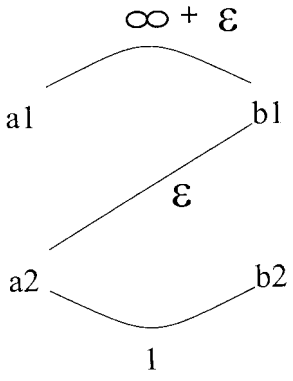

FIG. 2. The component affinity graph for Example 1.

the game is that no two nodes of the same column are in the same subset. The objective is to minimize the sum of the edge weights between subsets. Unfortunately, the problem is NP-complete in the size of the CAG (Li and Chen use a reduction from MAX-CUT [6]). To compute a satisfactory alignment, Li and Chen use a greedy heuristic based upon bipartite matching [14]. For Example 1, their heuristic leads to the optimal (minimal-weight) solution, namely aligning $a1$ with $b1$ and $a2$ with $b2$. In other words arrays a and b are directly superimposed onto the same template.

2.2. Distributing Parallel Loops

The previous alignment, however, causes all the potential parallelism to be lost when it comes to distributing array elements onto processors! To see why, consider the following two possible data distributions onto a unidimensional processor grid:

Distributing the first dimension. This means that rows of arrays a and b are distributed to processors: elements $a(i, j)$ and $b(i, j)$, for $1 \leqslant j \leqslant n$, are stored in (virtual) processor P_i. If we obey "the owner-computes" rule, this causes statement S_1 to be executed sequentially: given a value of the first loop index i, all iterations of the second loop index j are computed by the same processor P_i.

Distributing the second dimension. Quite similarly, distributing columns of a and b to processors will lead statement S_2 to be executed sequentially.

To summarize, the best alignment, as computed by Li and Chen, turns out to kill the parallelism. We claim that the alignment step should be conducted while having parallelism in mind: distributing parallel loops to processors is the true priority. A good alignment can reduce or suppress communications, but what if it leads to gather all parallel computations onto the same processor, as in our example?

We informally explain our approach using Example 1. See Section 4 for a complete description of our algorithm. Assume we target a one-dimensional processor grid. The highest priority is to distribute parallel computations, i.e., instances of the parallel loop j, on processors. In the example there is not much freedom: we distribute columns of a and rows of b to processors: processor P_j receives $a(i, j)$ and

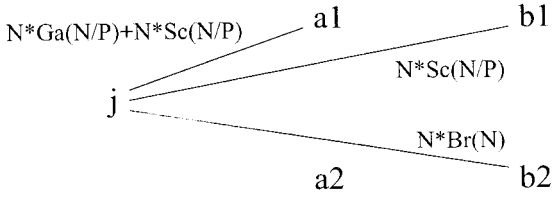

FIG. 3. The alignment–distribution graph for Example 1.

$b(j, i)$ for all $1 \leqslant i \leqslant n$. Owing to this distribution, for each instance of the external loop i, we distribute the parallel computations of loop j to processors. There remains some communications: for each instance i of the external loop, because of statement S_1, the ith row of b must be scattered from processor P_i to all processors, but parallelism has been preserved. Our approach does lead to this solution, based upon an *alignment–distribution graph* that privileges parallel loops. The alignment-distribution graph for Example 1 is represented in Fig. 3. It is built as follows: there are 4 *array dimension nodes*, one per array and per dimension, as in Li and Chen's CAG, plus an additional *loop node* for the parallel j loop. There is an edge between the loop node and an array dimension node if distributing both of them onto the processors induces communications. Edge weight corresponds to (estimated) communication costs. In Fig.3, "Ga" stands for "gather," and "Sc" for "Scatter."

The detailed construction of the graph as well as our solution to the problem are described in Section 4. We conclude our study of Example 1 with a few important remarks:

Remark 1: "*The owner-computes*" *rule.* There is no major reason to obey "the owner-computes" rule. The true objective is to distribute the *parallel computations* $S_1(i, j)$ and $S_2(i, j)$ to processor P_j, for $1 \leqslant i \leqslant n$. To this purpose, we might distribute columns of a and b to processors, which corresponds to Li and Chen's alignment, but we would insist that $S_2(i, j)$ is executed by processor P_j, at the price of a communication after the computation, to store the written value $b(j, i)$ into the memory of processor P_i. For each value of i, statement S_2 would then induce a gather operation (P_i owns $a(j, j)$, writes into $b(j, i)$, and sends it to P_i).

Remark 2: *Computations versus communications.* Example 1 is a toy example and should be considered as such. In this example, our solution may not be significantly better than a solution that sequentializes the parallel loop, because of the cost of the communications. Still, we can easily modify the example! Also, we can take benefit of the many papers in the literature to derive the best physical distribution, i.e., deciding whether rows of a and columns of b will be distributed in a pure cyclic, pure block, or block–cyclic fashion over p physical processors, where p is likely to be much smaller than n, the array size. In fact, our approach is quite flexible and can benefit from any precise modeling of the computation and communication costs: our alignment–distribution graph is vertex-weighted and edge-weighted, and the more precise the weights, the more accurate the solution. See the literature survey in Section 3.

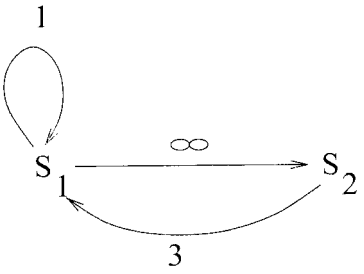

FIG. 4. The reduced dependence graphs (using dependence levels) for Example 2.

Remark 3: *Loop parallelization algorithms and redistribution.* An experienced programmer may have decided to apply loop distribution [19, p. 323][2] on Example 1 before considering alignment and distribution. Such a transformation is perfectly legal and leads to the following loop nest:

```
Distributing loops
for i = 2 to n do
    for // j = i + 1 to n do
            S1: a(i, j) = b(i, j) + a(i − 1, j)
    end for //
end for
for // i = 2 to n do
    for // j = i + 1 to n do
            S2: b(j, i) = a(j, j) + 1
    end for //
end for
```

We could then perform the alignment step separately on the two nests, and eventually redistribute some data array (say b in between. If the modified loop nest (having distributed the loop) is given as input to our alignment–distribution graph, and if the redistribution of one array (say b) is optimal, our algorithm will find it. However, given the original loop nest of Example 1, we do not deal with *any* loop transformation.

Consider the following modification of Example 1:

EXAMPLE 2

```
for i = 2 to n do
    for // j = i + 1 to n do
        for k = 2 to n do
            S1: a(i, j, k) = b(i, j, k) + b(i + 1, i, k − 1) + a(i − 1, j, k)
            S2: b(j, i, k) = a(i, j, k) + a(i, i + 1, k)
        end for
    end for //
end for
```

[2] A confusing terminology! Loop distribution here amounts to distributing statements inside the same loop so that they appear in separate loops. It is not related to distributing loop instances to processors.

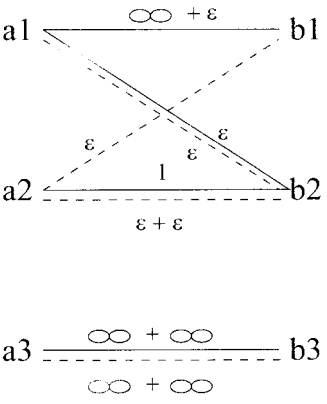

FIG. 5. The component affinity graph for Example 2.

The reduced dependence graph is shown in Fig. 4: loop distribution is no longer valid. We represent Li and Chen's CAG in Fig. 5: solid arrows correspond to statement S_1, and dashed arrows to S_2. Again, the optimal solution for the CAG is to superimpose arrays a and b, i.e., align each dimension of a with the same dimension of b. Again, this would lead to a sequential execution, whatever the distribution chosen. However, as before, our alignment–distribution graph, represented in Fig. 6, gives priority to the parallel loop j and distributes the first dimension of a and the second dimension of b to processors.

To summarize, our approach starts from a "*parallelized*" *loop nest*, i.e., a loop nest for which dependence analysis and loop parallelization have already been carried out. The most popular tools for these two steps are dependence levels [1, 2] and the Allen–Kennedy algorithm [2]. Given a parallelized loop nest, we determine which parallel loops should be distributed to processors, and the best alignment and distribution of arrays to minimize communications. This is done through the alignment–distribution graph.

Our main contribution is for a single-loop nest, possibly nonperfectly nested. When there are several consecutive loop nests, or an iterative loop surrounding

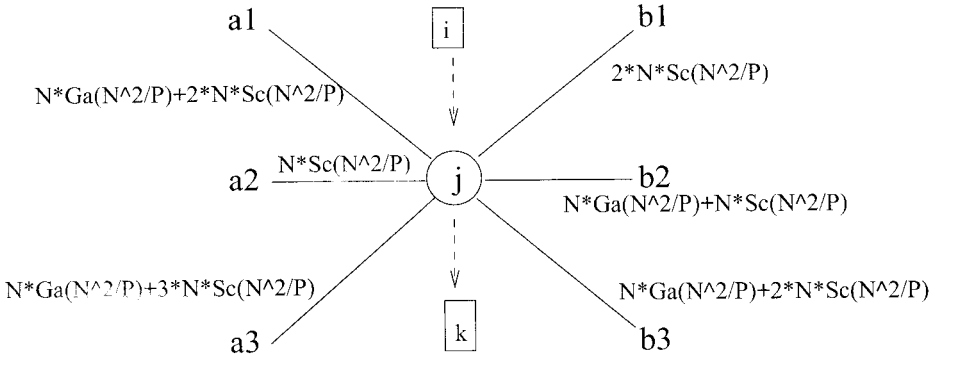

FIG. 6. The alignment–distribution graph for Example 2.

several loop nests, we use the approach of Lee [11], which we briefly summarize in Section 4.3 when dealing with multiple nests.

3. RELATED WORK

There are numerous papers on the alignment and distribution problem. We refer the reader to the survey [5] and the references therein. In this section, we summarize a few selected papers. In addition to Li and Chen's alignment method [13, 14] (already described in Section 2.1), we describe four papers by Tandri and Abdelrahman [17], Kelly and Pugh [9], and Ayguadé *et al.* [3, 4] whose goal is similar to ours. Next we present results by Gupta and Banerjee [7] and Li and Chen [12] on identifying structured communications and estimating their weight.

Our algorithm also uses the dynamic programming algorithm of Lee [11] when dealing with several loop nests. Indeed, redistributing some arrays between two consecutive nests may well prove more efficient. We describe Lee's technique in Section 4.3.

3.1. Tandri and Abdelrahman

Given a loop nest, Tandri and Abdelrahman [17] construct an undirected graph where each node represents either a parallel loop or an array dimension. There is an edge between a loop node and an array node if the dimension considered is indexed by the loop variable.

Attributes are assigned to the nodes: *, Cyclic or CyclicRCyclic for loop nodes, to favor load balancing, and *, Block or BlockCyclic for array node, to favor local access. For example, if X is referred to as $X(a * i + b * j)$ where j (outer) is parallel and i (inner) is sequential, then the attribute will be BlockCyclic.

There is a conflict when an edge connects two nodes whose attributes are different. To solve such a conflict, we replace the attributes by an intermediary. Thus, Cyclic and Block resolve to BlockCyclic.

Once all conflicts are solved, we must assign dimensions of the processor geometry to the nodes. The algorithm is a greedy one. We consider first the outer loop. We assign to them and to the array nodes connected to them a dimension of processors. We pursue then with the other nodes. A distribution scheme is then found.

Tandri and Abdelrahman's method is somewhat crude, in that communication costs are not taken into account precisely. Also, their selection of the best array dimension to be distributed is not clear. Still, they give priority to distributing parallel loops, and next they align the array dimensions onto those loops: we believe this is the right way to go, and we use a similar (but refined) scheme in our algorithm.

3.2. Ayguadé *et al.*

Ayguadé *et al.* [3, 4] consider programs composed of several consecutive perfect loop nests $L_1 L_2 \cdots L_n$. All arrays are assumed to have the same dimension d. They

describe their method for 1D and 2D grids, but we only deal with 1D grids in this short survey. We start with the construction of a graph called the *communication–parallelism graph*. Nodes are organized in columns. Each column represents an array in a nest an it contains d nodes.

There are two types of edges. Data movement edges show possible alignment alternatives between the dimensions of two arrays in a nest L_i. The assigned weight reflects the data movement cost to be paid if these two dimensions are aligned and distributed. We add other data movement edges to show possible realignment in a sequence of nests. If the array A in L_i is used in L_j, then $d \times d$ edges connect each node of array A in L_i to each node of A in L_j. If the edge connects the same dimension, its weight is null; otherwise its weight is the cost of a realignment.

Parallelism hyperedges show possible parallelization strategies for the loops in L_i. An hyperedge connects the nodes corresponding to the array dimensions that must be distributed to parallelize the loop according to the owner-computes rule. Its weight is the time that is saved when the loop is parallelized.

We must find a path in the CPG that includes exactly one node of each column so that the sum of weights of the edges minus the sum of weights of the hyperedges that connect nodes in the chosen path is minimized. This problem is formulated as a linear 0-1 programming problem. The variables are $Y_{PQ}(i, j)$, which corresponds to the edge between the ith dimension of P and the jth dimension of Q, and Z_k which corresponds to the kth hyperedge.

The constraints are the following:

- $\sum_j Y_{PQ}(i, j) = \sum_j Y_{QR}(i, j)$ for all i, P, Q, R
- $\sum_i \sum_j Y_{PQ}(i, j) = 1$ for all P, Q
- If Z_k connects the nodes $X_{P^1}(i_1), ..., X_{P^h}(i_h)$ which are connected by the edges $Y_{P^1Q^1}, ..., Y_{P^hQ^h}$, we need $\sum_j Y_{P^lQ^l}(i_l, j) \geqslant Z_k$ for all $l \in [1..h]$.

The approach of Ayguadé *et al.* [3, 4] is interesting because of their precise estimation of edge weights. Also they can handle redistribution between consecutive nests. However, the requirement that all nests are perfect and that all arrays have the same dimension is very restrictive. In addition, the integer linear programming solution may prove too expensive in practice.

3.3. Kelly and Pugh

The title of Kelly and Pugh's paper [9] is "*Minimizing Communication While Preserving Parallelism.*" This title exactly corresponds to our goal! However Kelly and Pugh consider a framework quite different from ours: they study all the possible transformations (loop permutations) of the program to determine which one induces the maximum of parallelism and the best mapping of the computations.

To determine valid loop permutations, Kelly and Pugh use a dependence analysis more sophisticated than the dependence levels. The direct dependences are computed by the Omega software and the indirect dependences are computed by transitive closure.

For each legal permutation, they determine the parallelism level that is allowed and they estimate the number of required synchronizations (they use a sophisticated model that allows pipelining to be taken into account). Finally, for each statement pair, they compute the number of data written in the first statement and read in the second one, using value-based flow dependence analysis.

To summarize, in the case where a precise dependence analysis is possible (e.g., when all dependences are affine), Kelly and Pugh's method is quite powerful. However, it cannot be applied to general loop nests where only limited information (such as dependence levels) is available.

3.4. Communication Patterns

Li and Chen [12] present interesting results on communication routines. They consider already parallelized programs with sequential and parallel loops. They assume that each array element can be assigned only once, that left-hand side subscripts are index variables, and that arrays are aligned to have a common index domain within each loop nest. We have a distribution scheme over a template and we want to recognize communication routines.

Each assignment $a(\sigma_1, ..., \sigma_n) = \cdots b(\delta_1, ..., \delta_n) \cdots$ may generate communications. If the tuples differ in only one corresponding pair of elements, the communication is either spread or a reduce or a copy or a shift or a multispread. The routine can be found with a pattern matching on these elements.

If the tuples are strongly different, we try by pattern matching on the tuples to recognize one of these routines: one-all-broadcast, all-one-reduce, single-send-receive, uniform-shift, or affine-transform. When a pattern cannot be matched with a routine, we decompose it into subpatterns. Indeed, a pattern over an n-dimensional index domain can be thought of as a composition of n simple patterns. For example, send $a(c(i, j), j-3)$ to (i, j) can be decomposed into two simple communications: send $a(c(i, j), j-3)$ to $(i, j-3)$, which is a multispread, and then send (the data) from $(i, j-3)$ to (i, j), which is a shift.

Gupta and Banerjee [7] improve Li and Chen's alignment method to estimate communication costs. Their method is based on pattern matching, applied upon the different assignments that could generate communications in the program. Their communication primitives are transfer, onetomanymulticast, manytomanymulticast, scatter, gather, shift, and reduction.

They allow operations on the structure of the program to decrease the communications costs by founding a better placement of communication. For instance they use loop distribution over two components to enable any communication placed between those components to be aggregated with respect to that loop. They try to permute loops when there is a parallel loop outside a loop in which communication takes place. To control the size of communication buffers required, they propose to strip-mine the loops.

Sometimes, the compiler may generate more communication than necessary, for example, when there are conditionals. Information about the frequency of execution of statements can help the compiler decide between carrying out potentially extra

communication and using a large number of messages. Since the primitives corresponding to different terms implement the data movement in distinct grid dimensions, they can legally be composed in any order. So another optimization is to permute the communications in favor of reducing the message sizes handled by processors.

4. SOLVING THE ALIGNMENT-DISTRIBUTION PROBLEM

As already stated, we start from a *parallelized* program, i.e., a program for which dependence analysis and loop parallelization have already been carried out: we are using the same hypotheses as Li and Chen [14]. Our goal is to preserve the potential parallelism while conducting the alignment step. We first describe our algorithm for a unidimensional processor grid. Next we move to a bidimensional grid. In both cases, we target a single (possibly nonperfectly nested) loop nest. For several consecutive loop nests, we simply use the approach of Lee [11], who uses a dynamic programming algorithm to determine whether some data redistribution is needed between two successive loop nests.

4.1. Unidimensional Grids

4.1.1. Construction of the Alignment-Communication Graph

We have two kinds of nodes in the graph, *array dimension* nodes and *loop* nodes:

- For each array, each dimension of this array is represented by a node (like for the Li and Chen graph). The weight of such a node is zero.
- Each loop is also represented by a node. We give a weight to this node, which represents the (approximated) execution time of the loop. For parallel loops, we divide the sequential execution time by the number of processors, as in Ayguadé *et al.* [3, 4].

Edges link array dimension nodes to loop nodes. There is an edge between two such vertices if there is a reference to the corresponding array dimension in the corresponding loop; the edge weight represents the (estimated) communication costs induced by the distribution of both the array dimension and the loop instances to the processors.

Finally, we add dashed arrows to illustrate the loop nesting. This is only for convenience. We refer to loop nodes and dashed arrows as the *loop subgraph* of the alignment–communication graph.

Consider the Cholesky factorization algorithm showed in Example 3. We use this example to describe our algorithm because it is a classical in compilation literature. Data dependence analysis can be conducted exactly on this example because all references are affine, but this is by no means a requirement for our algorithm.

EXAMPLE 3

```
for k = 1, to n do
    S1: a(k, k) = sqrt(a(k, k))
    for // j = k + 1 to n do
        S2: a(k, j) = a(k, j) / a(k, k)
    end for //
    for //i = k + 1 to n − 1
        for // j = i to n − 1
            S3 = a(i, j) = a(i, j) − a(k, i) * a(k, j)
        end for //
    end for //
end for
```

Note that Li and Chen's CAG for Cholesky has no edge, because there is a single array in the nest, and they do not take self-references into account. We represent the alignment–distribution graph in Fig. 7. Boxed nodes are the loop nodes: we use a circle for a parallel loop and a square for a sequential loop. The other nodes are the array dimension nodes. We use the following routines for the edge weights:

$\mathrm{Br}(N) = \mathrm{Broadcast}(N)$	a given process or sends the same message of size N to all processors.
$\mathrm{Sc}(N) = \mathrm{Scatter}(N)$	a given processor sends a (different) message of size N to each processor.
$\mathrm{Ga}(N) = \mathrm{Gather}(N)$	a given processor receives a (different) message of size N from each processor.
$\mathrm{Aa}(N) = \mathrm{All\_to\_all}(N)$	e ach processor sends the same message of size N to all processors.
$\mathrm{Aap}(N) = \mathrm{All\_to\_all\ person.}(N)$	each processor sends a (different) message of size N to each processor.

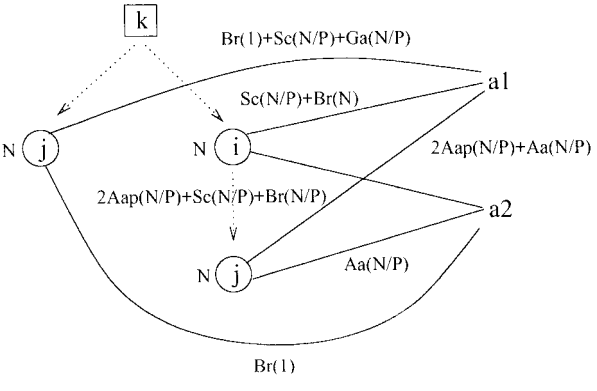

FIG. 7. The assignment–distribution graph for Example 3.

For example, the edge between $a2$ and the left parallel node j comes from statement S_2. It means that if we distribute this j loop and the second dimension of a, each processor j that computes $a(k, j)$ must receive from the same processor k the value of $a(k, k)$, hence the label $Br(1)$.

4.1.2. The Algorithm

The goal is to find exactly one parallel loop node to distribute along each path of the loop subgraph. We also need to distribute a dimension of each array. The optimization criteria is to minimize residual communications costs.

The optimal solution is to consider all different possibilities to distribute the parallel loops. Once a given distribution is chosen, we compare for each array the communication costs generated by this distribution, and we select the dimension that minimizes the communications. We sum the costs over all arrays and we obtain the total cost of the selected loop distribution. We keep the loop distribution scheme of minimal cost.

Coming back to Example 3, there are two different paths. We must choose j in the left path, and either i or j in the right path. In the case of the distribution scheme (j, i), we have for $a1$ the weight $N * Br(1) + 2N * Sc(N/P) + N * Ga(N/P) + N * Br(N)$ and $N * Br(1) + 2N * Aap(N/P) + N * Sc(N/P) + N * Br(N/P)$ for $a2$. The weight of $a1$ is lower; hence we distribute $a1$. For the other distribution scheme (j, j), the weight is $N * Br(1) + N * Sc(N/P) + N * Ga(N/P) + 2N * Aap(N/P) + N * Aa(N/P)$ for $a1$ and $N * Br(1) + N * aa(N/P)$ for $a2$. In this case, we choose $a2$. Then we must compare the two solutions. The cost of the first solution is $N * Br(1) + 2N * Sc(N/P) + N * Ga(N/P) + N * Br(N)$, and the cost of the second solution is $N * Br(1) + N * aa(N/P)$. Since a personalized all-to-all is expensive, we would most certainly select the first solution.

4.1.3. Complexity

Consider first the case of a perfect loop nest. Let s be the number of parallel loops, T be the number of arrays, and d_i the dimension of the ith array T_i. The complexity of our algorithm is $\mathcal{O}(s \times \sum_{i=1}^{T} d_i)$ because for each parallel loop and for each array, we search for the best dimension to distribute. Letting $d = \max_i(d_i)$ be the largest array dimension, the complexity of our algorithm is $\mathcal{O}(d \times T \times s)$.

It is important to understand why this result does not contradict the NP-completeness result of Li and Chen, who show that the alignment problem is NP-complete in the size of the CAG, i.e., the number of arrays T multiplied by the largest array dimension d. The intuitive explanation is the following: Li and Chen have no template reference for the alignment problem, so they must explore the possibility of aligning each dimension of each array with every dimension of every other array, hence the combinatorial swell. In contrast in our approach, because we aim at preserving the potential parallelism, each loop distribution scheme constitutes a reference pattern for which we search the best distribution for each array. Because we have few possible loop distribution schemes, the overall complexity is kept small.

THEOREM 1. *The alignment–distribution problem can be solved in time* $\mathcal{O}(d \times T \times s)$ *for a perfect loop nest with* s *parallel loops and* T *arrays with largest dimension* d.

In the case of a nonperfect nest, on a given path labeled i in the loop nodes of the alignment–distribution graph, there are s_i parallel loops. For instance in Example 3, we have two paths in the loop subgraph, $s_1 = 1$ and $s_2 = 2$. The complexity of the algorithm is $\mathcal{O}(d \times T \times \prod_{i=1}^{p} s_i)$ because $\prod_{i=1}^{p} s_i$ represents the number of distribution scheme. In the worst case, the complexity is $\mathcal{O}(d \times T \times e^s)$.

The exponential term is not important. Indeed, the number of parallel loops in a nest is not higher than 3 in practice.

4.1.4. Remarks

Remark 1. In the above version of the algorithm, we always distribute exactly one parallel loop along each path of the loop subgraph. In certain cases, it may well be more efficient to execute a parallel loop in sequential mode on a single processor. We can implement this modification, which amounts to select *at most one* (instead of *exactly one*) parallel loop along each path of the loop subgraph: we make a copy of each parallel node. One copy indicates a sequential execution and the other a parallel execution. So, there are twice as many loop nodes, hence more loop distribution schemes to evaluate.

Similarly, we always distribute one dimension of each array. Sometimes, it will be better to allocate a whole array to an unique processor. To that purpose, we can add a node for each array, which indicates that we do not want to distribute this array.

Remark 2. The problem (and of course the alignment–communication graph) is "symmetric" between loop nodes and array dimension nodes. Sometimes, it will be better to iterate on all possible distribution schemes for the arrays, and to deduce the best distribution scheme for the loops. For Example 3, there is a single array of dimension 2 and several loop nodes, so we should indeed consider the different choices for distributing a, and for each of them to determine the best distribution scheme for the loops.

Remark 3. For the (mostly theoretical) situation where our algorithm would be too costly, we can introduce the following greedy heuristic: along each path of the loop subgraph, give priority to distributing the most external parallel loop. This will lead to the largest granularity of the tasks that are distributed to processors.

4.2. Bidimensional Grids

If the dimension of the processor grid of processors is larger than 1, we propose the following two strategies.

4.2.1. Recursive Algorithm

We build the alignment–distribution graph just as in Section 4.1, and we use the previous unidimensional algorithm. At this stage we have chosen to distribute one

parallel loop and one dimension of each array. We distribute them along the first dimension of the grid.

We construct a new graph by deleting already chosen nodes. We update edge weights by taking the distribution scheme for the first grid dimension into account. Then we use a second time the unidimensional algorithm to determine which loops and which array dimensions will be distributed along the second grid dimension.

We iterate the process as many times as there are dimensions in the processor grid.

In all

EXAMPLE 4. Assume that we target a 2D-processor grid for the nest

```
for // i = 1 to n do
    for // j = 1 to n do
        for // k = 1 to n do
            a(i, j, k) = b(j, i, k) * b(i, j, k)
        end for //
    end for //
end for //
```

Using this recursive algorithm, we first distribute the k loop and the last dimension of a and b. Indeed, such a choice preserves the parallelism and is communication-free. After deleting the corresponding nodes and updating the weights, we obtain the graph of Fig. 8. Next the recursive algorithm decides to distribute i and the first dimension of a and b along the second grid dimension.

4.2.2. *Optimal Algorithm*

The main principle of the optimal algorithm is the same as in the unidimensional case. Instead of considering one node by path of the loop subgraph, we consider g nodes by path, where g is the dimension of the target processor grid. When g loop nodes are chosen along each path, we determine for each dimension of each array the cost of the communications induced by the distribution of this dimension and these loops. We keep the loop distribution scheme, which minimizes the communications.

Coming back to Example 4, we construct the graph depicted in Fig. 9. In this graph, we must compare the three following cases: distribute (i, j), distribute (i, k), or distribute (j, k).

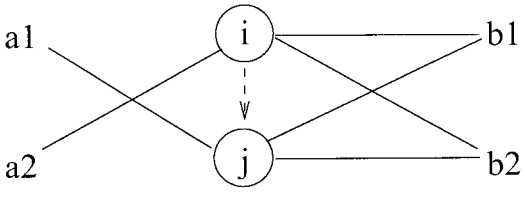

FIG. 8. Recursive algorithm: after the first step.

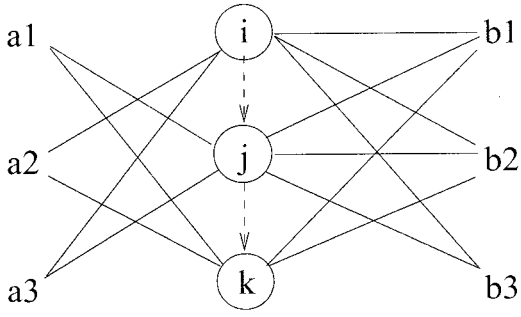

FIG. 9. The alignment–distribution graph for Example 4.

Distribute (i, j): We distribute $(a1, a2)$ and $(b1, b2)$.
Distribute (i, k): We distribute $(a1, a3)$ and $(b1, b3)$.
Distribute (j, k): We distribute $(a2, a3)$ and $(b2, b3)$.

In all three cases communications come from accessing $b(j, i, k)$. The first case is very expensive. We must choose between the second and the third. Since the communications are the same for both, we distribute (i, k) the solution with largest task granularity.

4.2.3. Comparison

Let g be the number of dimensions of the processor grid. For the recursive algorithm, the complexity for a perfect loop nest is $\mathcal{O}(g \times d \times T \times s)$. For a nonperfect nest, we get $\mathcal{O}(g \times d \times T \times e^s)$. This is because we use the unidimensional algorithm g times. Of course g can be viewed as a small constant in practice ($g = 2$ or 3 for current machines).

For the optimal algorithm, the complexity for a perfect nest is $\mathcal{O}(\sharp\, schemes \times T \times d)$. The number of loop distribution schemes is $s(s-1)\cdots(s-g+1)$. Hence the complexity is $\mathcal{O}(d \times T \times s^g)$. For a nonperfect nest, the complexity is $\mathcal{O}(d \times T \times \prod_{i=1}^{d} s_i^g)$. So in the worst case, it's $\mathcal{O}(d \times T \times e^{g \times s})$.

Of course the optimal algorithm has higher complexity. However, it relies on a more accurate estimation of the communication costs, because when we search for a loop distribution scheme we look for g dimensions of arrays to distribute together with the selected loops.

4.3. Several Nests

In the case of several loop nests, we use the method proposed Lee [11]. Given a program constituted by a sequence of n nests, we want to determine the best distribution scheme (for parallel loops and arrays) for the whole program. In a word, Lee [11] uses Li and Chen's CAG as a basic block for a single-loop nest, together with a dynamic programming algorithm to determine whether to redistribute some array in between two consecutive blocks. We simply suggest to use our alignment–distribution graph as a new basic block, and to keep the dynamic approach

unchanged. This will preserve parallelism over the whole program in addition to determining the best distribution and redistribution of arrays.

When we consider two consecutive nests, we have two main choices:

- Either we keep the same alignment–distribution for the two nests, and we look for the scheme that minimizes the sum of the communications for both nests,
- or we determine the best alignment–distribution for each nest, and we use a redistribution in between.

Consider a sequence of n loop nests $L_1 L_2 \cdots L_n$. For each subsequence $L_i L_{i+1} \cdots L_{i+j-1}$, where $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n-i+1$. Let $T_{i,j}$ be the minimal time to compute $L_1 L_2 \cdots L_{i+j-1}$ with the restriction that it uses the distribution scheme $P_{i,j}$ for the sequence $L_i L_{i+1} \cdots L_{i+j-1}$. Thus the final distribution scheme after computing $T_{i,j}$ is $P_{i,j}$. At the beginning, $T_{1,j}$ is equal to $M_{1,j}$. Let $cost(P_{i-k,k}, P_{i,j})$ be the communication cost of changing data layouts from $P_{i-k,k}$ to $P_{i,j}$. Lee [11] uses the dynamic programming algorithm

```
for i = 2 to s do
    for j = 1 to s − i + 1 do
        T_{i,j} = min_{1 ⩽ k < i} (T_{i−k,k} + M_{i,j} + cost(P_{i−k,k}, P_{i,j}))
    end for
end for
Minimum = min_{1 ⩽ k ⩽ s} (T_{s−k+1,k})
```

If the sequence of nests is enclosed by an iterative loop, the last line of the algorithm is modified as

$$\text{Minimum} = \min_{1 \leqslant k \leqslant s} (T_{s-k+1,k} + MAX\_ITER \times dependence(T_{s-k+1,k})),$$

where $dependence(T_{s-k+1})$ returns the cost of changing data layouts from the distribution scheme of the last nest to the first one.

Consider the following simple example:

EXAMPLE 5

```
for // i = 1 to n do
    for j = 1 to n do
        a(i, j) = a(i, j + 1) + a(i, j)
    end for
end for //
for i = 1 to n do
    for // j = 1 to n do
        a(i, j) = a(i − 1, j) * a(i, j)
    end for //
end for
```

Lee's algorithm consists in considering the program either as a unique nest or as two nests for which we may need to determine a redistribution scheme.

A unique nest. Our alignment–distribution algorithm decides to distribute the two parallel loops and the first dimension of a. The second nest induces many communications.

Two different nests. For the first nest, we distribute the i loop and the first dimension of a. For the second nest we distribute the j loop and the second dimension of a. There is no communication inside the two nests, but we need communications to redistribute a between them.

We must compare both solutions. In the first case, processor P_j receives $a(i, j)$ from P_i and $a(i-1, j)$ from P_{i-1}, and then sends the result to P_i. Each processor must communicate with all the others several times. However, if we use a block distribution, these communications are often transformed into local memory accesses. So the final solution is to distribute i, j, and $a1$ (the unique nest strategy).

5. CONCLUSION

We have introduced the alignment–distribution graph to replace Li and Chen's component affinity graph. The major two advantages of our approach are the following:

- Parallelism is preserved: we derive the best loop distribution together with the best array alignment.
- Complexity is polynomial for perfect loop nests. Complexity is always polynomial in the number of arrays addressed inside the nest.

In addition, we retain all the flexibility of Li and Chen's approach: new results from the literature and from experiments can be easily incorporated, for instance, to refine the estimation of the communication and computation weights. Indeed, our weight model for communications is much more refined than the original CAG of Li and Chen; as for computation costs, we can also benefit from the literature, e.g., [8, 15, 16]. Finally, our graph can be used as a building block for techniques that manipulate larger programs.

The current largest limitation is that our alignment–distribution graph is built for a fixed, already parallelized loop nest. It would be nice to incorporate loop transformations in the framework: how to determine the best way of writing the loop nest in order to derive the best way to distribute and computations to processors?

REFERENCES

1. J. R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Technical Report, MASC-TR82-6, Rice University, Houston, TX, 1982.
2. J. R. Allen and K. Kennedy, Automatic translation of Fortran programs to vector form, *ACM Trans. Programm. Lang. Syst.* **9** (October 1987), 491–542.

3. E. Ayguadé, J. Garcia, M. Gironès, M. L. Grande, and J. Labarta, DDT: A research tool for automatic data distribtution in HPF, *Sci. Programm.* **6** (1997), 73–94.
4. E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres, and M. Valero, Detecting and using affinity in an automatic data distribution tool, *in* "Languages and Compilers for Parallel Computing," pp. 61–75, Springer-Verlag, Berlin, 1995.
5. E. Ayguadé, J. Garcia, and U. Kremer, Tools and techniques for automatic data layout: A case study, *Parallel Comput.* **24** (1998), 557–578.
6. M. R. Garey and D. S. Johnson, "Computers and Intractability, a Guide to the Theory of NP-completeness," W. H. Freeman, New York, 1991.
7. M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers, *IEEE Trans. Parallel Distrib. Syst.* **3** (1992), 179–193.
8. M. R. Haghighat, "Symbolic Analysis for Parallelizing Compilers," Kluwer Academic, Dordrecht, 1995.
9. W. Kelly and W. Pugh, Minimizing communication while preserving parallelism, *in* "Proceedings of the 10th ACM International Conference on Supercomputing," Assoc. Comput. Mach., New York, 1996.
10. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel, "The High Performance Fortran Handbook," MIT Press, Cambridge, MA, 1994.
11. P. Lee, Efficient algorithms for data distribution on distributed memory parallel computers, *IEEE Trans. Parallel Distrib. Syst.* **8** (1997), 825–839.
12. J. Li and M. Chen, Compiling communication-efficient programs for massively parallel machines, *IEEE Trans. Parallel Distrib. Syst.* **2** (1991), 361–375.
13. J. Li and M. Chen, Index domain alignment: Minimizing cost of cross-referencing between distributed arrays, *in* "Frontiers 90: The 3rd Symposium on the Frontiers of Massively Parallel Computation," pp. 424–433, College Park, MD, IEEE Computer Society Press, Los Alamitos, CA, October 1990.
14. J. Li and M. Chen, The data alignment phase in compiling programs for distributed-memory machines, *J. Parallel Distrib. Comput.* **13** (1991), 213–221.
15. K. S. McKinley, "Automatic and Interactive Parallelization," Ph.D. thesis, Department of Computer Science, Rice University, 1992.
16. W. Pugh, Counting solutions to Pressburger formulas: How and why, *in* "ACM SIGPLAN Conference on Programming Language, Design and Implementation," Assoc. Comput. Mach., New York, 1994.
17. S. Tandri and T. S. Abdelrahman, Automatic data and computation partitioning on scalable shared memory multiprocessors, *in* "3rd Workshop on Automatic Data Layout and Performance Prediction, 1997."
18. M. Wolfe, The tiny loop restructuring research tool, *in* "International Conference on Parallel Processing" (H. D. Schwetman, Ed.), Vol. II, pp. 46–53, CRC Press, Boca Raton, FL, 1991.
19. M. Wolfe, "High Performance Compilers For Parallel Computing," Addison–Wesley, Reading, MA, 1996.