Electronic version of an article published as: "Parallel processing letters", vol. 6, num. 1, 1996, p. 159-171.
DOI 10.1142/S0129626496000169. ©World Scientific Publishing Company
https://www.worldscientific.com/doi/10.1142/S0129626496000169

USING A 0-1 INTEGER PROGRAMMING MODEL FOR AUTOMATIC STATIC DATA DISTRIBUTION

JORDI GARCIA, EDUARD AYGUADÉ and JESÚS LABARTA
Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
Gran Capità s/núm, Mòdul D6, 08034 - Barcelona, SPAIN
e-mail: {jordig | eduard | jesus}@ac.upc.es

Received 13 March 1995
Revised 15 June 1995
Accepted by Prof. H. Zima

ABSTRACT

This paper describes an automatic data distribution method which deal with both the alignment and the distribution problems in a single optimization phase, as opposed to sequentially solving these two inter-dependent approaches as done by previous work. The core of this work is called the Communication-Parallelism Graph, which describes the relationships among array dimensions of the same and different array references regarding communication and parallelism. The overall data distribution problem is then formulated as a linear 0-1 integer programming problem, where the objective function to be minimized is the total execution time. The solution is static in the sense that the layout of the arrays does not change during the execution of the program. We also show the feasibility of using this approach to solve the problem in terms of compilation time and quality of the solutions generated.

Keywords: Automatic Data Distribution, Automatic Parallelization, Distributed-memory Multiprocessors, Computation and Data Movement Costs, 0-1 Linear Programming.

1 Introduction

Data distribution is one of the key aspects that have to be considered in a parallelizing environment for Massive Parallel Machines, in which each processor has direct access to a local (or close) memory and indirect access to the remote memories of other processors. The cost of accessing a local memory location can be more than one order of magnitude faster than the cost of accessing a remote memory location. In these systems, the choice of a good data distribution can dramatically affect performance because of the non-uniformity of the memory system. However, the data distribution problem has been proved to be NP-complete in [16, 18].

Several researchers have targeted their research efforts to this topic. For instance, the Crystal compiler and language project [18], the implementation of PARADIGM [13] on top of Parafrase-2 and its continuation on the PTRAN II compiler [12] at IBM, the framework for the automatic determination of array alignment and distribution presented in [6, 7], or the automatic data layout strategy [4, 17] for use in the D programming environment currently under development at Rice University are examples of projects in

this area. Other groups have targetted their effort to the effective compilation of programs containing the specification of the data distribution, such as the VFCS [5] for the Vienna Fortran language [22], the Fortran-D compiler [10, 21].

All the automatic approaches to automatically distribute data perform the job in two main independent steps: alignment and distribution. The alignment step tries to find appropriate alignments between all arrays in a block of code, that is, to decide for each array the dimensions that will be aligned into the dimensions of another array called the template (interdimensional alignment), and for each aligned dimension, to decide whether it is better to shift the array with respect to the template or not (intradimensional alignment). A good alignment will minimize the overhead of interprocessor communication.

The distribution step decides which dimension or dimensions of the template are distributed, and the number of processors assigned to each of them. The mapping of the arrays is determined by their alignment with respect to the template and its distribution. A good distribution maximizes the potential parallelism of the code, and offers the possibility of further reducing communication by serializing. This goal could be trivially satisfied assigning a datum to each processor, which maximizes parallelism.

This paper describes an automatic data distribution method which deal with both the alignment and the distribution problems in a single optimization phase, as opposed to sequentially solving these two inter-dependent approaches as done by previous work.

Our approach builds the Communication-Parallelism Graph (CPG), a directed weighted graph that holds information about both the data movement and parallelism inherent in a block of code. This information is used to formulate a minimal path problem with a set of additional constraints, which is solved using a general purpose linear 0-1 integer programming solver. This solver finds the optimal solution, in a small amount of time, to the two problems in a single step. This allows us to minimize communication while maximizing parallelism, avoiding the use of heuristics and possibly wrong assumptions about distribution during the alignment phase. In the framework of automatic data distribution, the use of linear 0-1 integer programming solvers was proposed by [4] when looking for a dynamic solution for the data layout problem.

```
CHPF$ DECOMPOSITION TEMPLATE(N, N, N)
CHPF$ ALIGN A(I,J,K) WITH TEMPLATE(K,J,I)
CHPF$ ALIGN B(I,J,K) WITH TEMPLATE(I,J,K)
CHPF$ ALIGN C(I,J) WITH TEMPLATE(1,I,J)
CHPF$ ALIGN D(I,J,K) WITH TEMPLATE(I,J,K)
CHPF$ DISTRIBUTE TEMPLATE(:,BLOCK,:)

      do i = 1, N
CAPR$ DO PAR ON B<:,1~1,:>
            do j = 1, N
                  do k = 1, N
                       B (i, j, k) = C (j, k) + i
                       A (k, j, i) = B (i, j, k) + 1
                  enddo
                  do k = 2, N
                       D (i, j, k) = D (i, j, k - 1)
                  enddo
            enddo
      enddo
```

Figure 1: Working example with data mapping and loop parallelization directives.

This paper presents the method that finds the optimal solution for one-dimensional array distributions. Figure 1 shows the code that will be used as working example along the paper, and the corresponding HPF [14] data mapping directives that are selected by our approach, after performing the corresponding analysis and finding the minimal cost solution. The parallelization directive in loop j has been specified using the syntax defined in [3].

The rest of this paper is organized as follows: in Section 2 we describe the Communication-Parallelism Graph (CPG). Section 3 shows how the constraints in the CPG can be formulated as a linear 0-1 integer programming problem, when a single array dimension is distributed. Section 4 summarizes our implementation and shows our first experimental results. Finally, a few concluding remarks are given in Section 5.

2 The Communication-Parallelism Graph

The main structure of our method is the Communication-Parallelism Graph (CPG). It is a directed graph that contains all the information related to communication and parallelism in a block of code. The CPG is created from the analysis of all assignment statements within loops that contain one array in the left hand side of the assignment.

2.1 CPG Nodes

The nodes in the CPG are organized in columns. Each column represents an array, and it contains as many nodes as the maximum dimensionality d of all arrays in the block of code. Each node in a column, thus, represents one dimension of the array that will be mapped into one of the dimensions of a common array called template with dimensionality d. If the array has dimensionality d' < d, then the column is padded with (d - d') additional nodes. These nodes are included to allow an embedding of the array on the template (sequentialization). As seen in the example in Figure 1, array C is embedded into the first dimension of the template. Since four arrays are used in our working example (three of them with dimensionality three, and one with dimensionality two) the CPG consists of 12 nodes (four columns with three nodes each). This is the basis of the CPG, over which the communication and parallelism information will be added in terms of data movement edges and parallelism hyperedges

2.2 CPG Data Movement Edges

Data movement information is obtained from the analysis of reference patterns. The meaning of a reference patterns is defined in [18], and represents a collection of dependences between the array in the left-hand side and each array in the right-hand side of an assignment statement. When the same array is used in both sides, the reference pattern is called self-reference pattern. For instance, consider our working example in Figure 1. From the assignment statements inside the loop nest, two reference patterns can be extracted:

`B (i, j, k)` ← `C (j, k)` and `A (k, j, i)` ← `B (i, j, k)`

and one self-reference pattern:

`D (i, j, k)` ← `D (i, j, k - 1)`

For each reference pattern between two different arrays, (d x d) directed edges are added, where d is the maximum dimensionality of all arrays. The edges connect all nodes of the array to be read (right hand side) to all nodes of the array to be updated (left hand side), and represent all alignment possibilities between these two arrays. The weight that is assigned to the edge is a symbolic expression that represents the cost of the data movement that has to be performed when these two dimensions are aligned and distributed over a generic number p of processors. This cost reflects the number of remote memory accesses that have to be performed, and it is a function of p and the data movement timings in the target machine. When a self reference pattern is found, d self edges are added in the column of the referenced array, one in each node. As in the previous case, the weight is a symbolic expression that represents the cost of the data movement that has to be performed when this dimension is distributed. Several edges between a pair of nodes are replaced by a single edge with a weight equal to the sum of the original ones. Details about the matching of reference patterns to data movement routines and the estimation of their cost can be found elsewhere [11, 18].

After adding the data movement information, the CPG (without weights) that is obtained is shown in Figure 2.a. In this graph, the edges due to the assignment of array B to array A, the assignment of array C to array B, and the self assignment of array D are shown. The cost functions have been omitted. But for instance, the costs associated to the self-edges for array D would correspond to the costs of performing local memory accesses for edges D[1]-D[1] and D[2]-D[2] and the cost of performing shift-like data movements for edge D[3]-D[3].

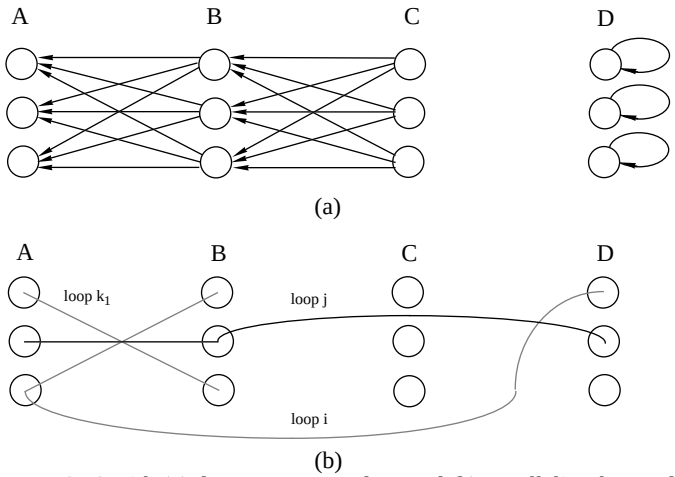


Figure 2: CPG with (a) data movement edges and (b) parallelism hyperedges.

Note that the CPG at this point is not the Component Affinity Graph (CAG) used by other authors [18, 13]. The meaning of the edges in the CAG is a preference for alignment, that is, how good is to align two dimensions. The weight should reflect the extra communication cost incurred if those dimensions are not aligned. The problem is that one can not identify two unique alignment configurations under which the communication costs may be estimated and compared to determine the penalty. In our approach, the

meaning of one edge in the CPG is just the opposite, that is, how costly is (in terms of data movement) to align and distribute the two dimensions. This cost is independent of any other alignment strategy, and it will be useful to determine the distribution strategy.

2.3 CPG Parallelism Hyperedges

Parallelism information is obtained from data dependence analysis. First, all loops that can be executed in parallel are detected. In distributed memory machines you can fully parallelize one loop when the loop does not carry any data flow dependence. When all possible parallel loops have been marked, the analyzer must only look at the left hand side of the assignments that are inside any parallel loop. According to the owner computes rule, the processor that owns a data element is the one who performs all the computations to update it. So if one loop is going to be parallelized, it must be ensured for the arrays that appear in the left hand side of each assignment statement inside the loop, that the dimension subscripted by the loop control variable of the parallel loop is the one that will be distributed.

For each parallel loop, the analyzer searches for all the assignment statements inside it with an array in the left hand side. If the array uses the loop control variable in any dimension of its subscript, then the analyzer links the corresponding node to a hyperedge. The hyperedge is a generalization of an edge, as it can connect more than two nodes, and in this case, it is undirected. Each parallel loop has a hyperedge in the CPG. If all the nodes connected by a hyperedge are aligned and distributed, then the corresponding loop can be parallelized. But if one of the nodes connected by a hyperedge is not aligned, then the loop can not be parallelized, and guard expressions must be inserted before each assignment in that loop [21].

The weight of a hyperedge represents the execution time that is saved when the loop is parallelized. This time is a function of the sequential execution time of the loop, and the number of processors that will be assigned to that dimension.

In the example of Figure 1, after performing the data dependence analysis, one can see that there is a single flow dependence carried by the second k-loop. This means that the i, j, and the first k loops can be executed in parallel. So the parallelism information that the analyzer will add to the CPG is shown in Figure 2.b. Three hyperedges have been added: one linking B[1], A[3], and D[1]; this one is associated to loop i. Another hyperedge links A[2], B[2] and D[2], and it is associated to loop j. The last one links A[1] and B[3], and it is associated to the first loop k.

2.4 Problem Formulation

Once the CPG is built, it contains all the necessary information regarding data movement and parallelism in the block of code analyzed. The weights in the CPG are expressed in a symbolic form, function of the number of processors that will be assigned to that dimension. If all symbolic expressions in the CPG are replaced for its constant value assuming P processors, then the weights in the edges will represent the cost of aligning and distributing the corresponding dimensions. In this section, the CPG can be considered as an undirected graph, and all pairs of edges connecting two nodes in different direction, can be replaced by a single undirected edge with weight sum of the original edges. So if one

node (dimension) of each column (array) is selected, the sum of the weights of all data movement edges that connect any two nodes inside the selection is the total data movement cost of the block of code when distributing the selected dimensions. Similarly, the sum of weights of all parallelism hyperedges whose nodes remain completely inside the selected set is the total execution time saved when parallelizing and assigning P processors.

It can be assumed that there exists a path between any pair of columns in the CPG. If any set of arrays is not connected, then this set will be analyzed independently, and assigned a different template. The reason is that a relation (alignment) between two unrelated sets of arrays should not be imposed. If an array is connected to the CPG but only through a hyperedge, then all nodes of this array will be connected to all nodes of any other array with a data movement edge, and assigned a null weight.

Now, it can be ensured that the CPG is an undirected connected graph, where every pair of columns are connected by a path, and all edges and hyperedges have constant weights. The problem to solve is to find the set of nodes (one for each column), that minimizes the communication time, and maximizes the time saved due to parallelization. The total execution time can be computed as the sequential execution time plus the overhead due to data distribution and parallelization; according to the weights in the CPG, this overhead is the data movement overhead minus the time saved due to the parallel execution of the loops. The problem is formulated as a linear 0-1 integer programming problem, where the objective function to minimize is the total execution time. The optimal solution can be found fastly as explained in the next section.

3 One-dimensional Distribution

Linear integer programming is a tool for solving optimization problems. As stated by [4] data layout problems can be very efficiently solved using linear integer programming. In this case, the problem to solve is to find a path in the CPG that includes exactly one node of each column, so that the sum of weights of the edges (data movement edges) minus the sum of weights of the hyperedges (parallelism hyperedges) that connect nodes inside the selected path, is minimized. This problem can be formulated as a linear 0-1 integer programming problem, that is, a linear integer programming problem where each variable has two possible values: 0 or 1.

The model that we have selected is a shortest path problem with a set of additional constraints. When formulating it, two dummy nodes are considered: a source S and a sink T. All edges going from the dummy source S to each node in the first column, and all edges going from each node in the last column to the dummy sink T, must be defined. In addition, all columns C with self-edges will be replaced by two columns named C and C', and each node in column C will be linked by an edge to each node in column C'. The weights of the new edges will be the same of the self-edge if it links two nodes of the same level, or infinite otherwise. This infinite weight prevents from selecting two nodes of different level from the same column. Figure 3 shows the graph that finally will be considered to solve the problem for the working example of Figure 1. Neither infinite weighted edges are present, nor hyperedges.

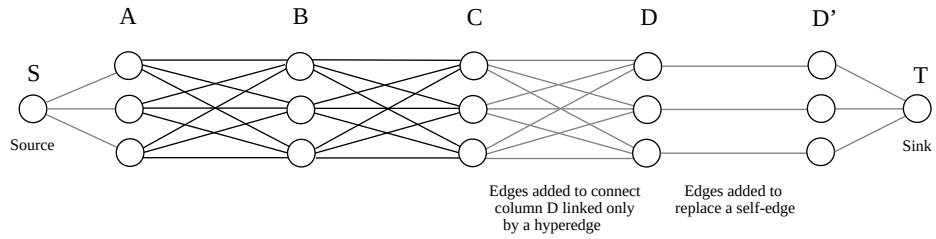


Figure 3: CPG with all the data movement edges that fulfils the properties mentioned in Section 3.

As usual in this kind of problems, one 0-1 integer variable associated to each edge, plus one 0-1 integer variable associated to each hyperedge are defined. Let Y_{PQ} denote the set of edges connecting nodes in column P to nodes in column Q. When Y_{PQ} is not empty, it contains (d x d) elements. If S denotes the column associated to the source, the sets Y_{SP} are all empty, except when P denotes the first of the columns, and it has d elements. Similarly, if T denotes the column associated to the sink, the sets Y_{QT} are all empty except when Q denotes the last of the columns, and it has d elements. Let $Y_{PQ}[i, j]$ be the variable associated to the edge connecting node i in column P to node j in column Q. Its value will be one if the corresponding edge belongs to the path, and zero otherwise. Note that, as the graph is undirected, $Y_{PQ}[i, j]$ is the same than $Y_{QP}[j, i]$. Finally, if an index is assigned to each hyperedge, Z_m will denote the 0-1 integer variable associated to the m-th hyperedge. Similarly, its value will be one if the nodes it links belong to the path, and zero otherwise.

To ensure the correctness of the solution, some constraints must be defined. There are four types of constraints:

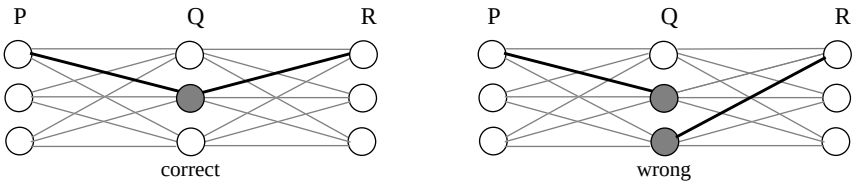
C1: The solution is a path.

C2: The path goes through one node in each column.

C3: All edges connecting selected nodes, must be included as well.

C4: If a node of a hyperedge is not selected, neither it is.

Constraints C1 ensure that the solution is connected. That is, for each column Q conected to more than one column P and R, if one edge leading to a node in Q is selected in the set Y_{PQ}, one edge leaving this same node must be selected in the set Y_{QR}. This can be graphically seen in the following two examples:



In terms of the variables and their values, it can be stated that at each node of each column Q (except for S and T) connected to more than one column P and R, the sum of the values of the edges that connect this node to column P, must be equal to the sum of the values of the edges that connect this node to column R. This must be accomplished for each pair of sets Y_{PQ}-Y_{QR} with a column in common.

$$\sum_{j=1}^{ncols} Y_{PQ}[j,i] = \sum_{j=1}^{ncols} Y_{QR}[i,j] \qquad ; i = 1 \ldots ncols$$

Constraints C2 and C3 can be specified together. They ensure that only one node can be selected in each column, and that all edges connecting two selected nodes, must also be included. These can be accomplished forcing that, for each non empty set of edges Y_{PQ}, exactly one edge must be selected. Two examples are shown below:

correct wrong

This can be stated, in terms of variables and their values, that the summatory of a non empty set of edges Y_{PQ} must equal one. This set of constraints, must be accomplished for each non empty set of edges.

$$\sum_{i=1}^{ncols} \sum_{j=1}^{ncols} Y_{PQ}[i,j] = 1 \qquad ; PQ = 1 \ldots nsets$$

Applying these constraints may result in a path that is not simple; it may contain cycles, or that the same node may be visited more than once in the path. Note that this is not contradictory with the 2nd constraint, since exactly one different node in each column belongs to the path.

Finally, constraints C4 ensure the correct behavior of the hyperedges. A hyperedge m belongs to the selection if all nodes linked by it have been selected. One node n of column P belongs to the selection, if the value of one of the edges that connects it to any other column Q equals one. Or in other words, if the summatory of the values of the edges that connect it to column Q equals one. This can be stated, in terms of variables and their values, that the sum of the values of the edges that connect the node n of column P to any other column Q, is greater or equal than Z_m, where Z_m is the 0-1 integer variable associated to the m-th hyperedge.

$$\sum_{j=1}^{ncols} Y_{PQ}(n,j) \geq Z_m$$

This must be accomplished for each node linked by the hyperedge. For instance, in the CPG corresponding to the example code of Figure 1, the hyperedge labeled i links nodes B[1], A[3] and D[1]. Column A is connected to column B, column B is connected to column C, and column C is connected to column D. This set of constraints, can be stated as:

$$\sum_{j=1}^{ncols} Y_{BC}(1,j) \geq Z_i \qquad ; \sum_{j=1}^{ncols} Y_{AB}(3,j) \geq Z_i \qquad ; \sum_{i=1}^{ncols} Y_{CD}(i,1) \geq Z_i$$

The 0-1 integer variable associated to hyperedge labeled i is Z_i. In this case, if any of the three summatories is zero, then Z_i will also be zero. This constraint must be formulated for each hyperedge m in the graph.

Once all constraints have been formulated, the objective function to minimize must be specified: the sum of the weights of all selected edges, minus the sum of the weights of all selected hyperedges. Let r be the total number edges in the CPG. The sum of the weights of all selected edges can be expressed as the scalar product in the space R^r of Y and C:

$$CostOfEdges = Y \cdot C$$

where Y represents the vector of all 0-1 integer variables associated to edges, and C represents their respective weights.

Similarly, let m be the number of hyperedges in the CPG. The sum of the weights of all selected hyperedges can be expressed as the scalar product in the space R^m of Z and T:

$$CostOfHyper = Z \cdot T$$

where Z represents the vector of all 0-1 integer variables associated to hyperedges, and T represents their respective weights.

Finally, the objective function can be expressed as:

$$Cost = CostOfEdges - CostOfHyper$$

The objective function must be minimized. When all constraints and the objective function have been specified, the integer linear programming solver finds the optimal solution (minimum) subject to the specified constraints.

4 Some Experimental Results

The main elements of our tool are described next. The parsing of the code is performed using the parser module of DDT [2], a tool that analyzes reference patterns in Fortran programs; DDT has been developed on top of ParaScope [15]. It obtains all reference patterns after performing some well known optimizations, like expression substitution, subscript substitution, and induction variable detection [1] that improves the quality and quantity of reference patterns analyzed. A profile of the sequential execution of the program is used to estimate the costs of the parallel hyperedges in the CPG. Machine specific information is used to estimate the costs of the data movement edges. The analyzer creates a data file with the set of constraints and the objective function to minimize. This file is the input of a general-purpose linear programming solver, which minimizes the objective function, and generates an output file with the final value of each 0-1 integer variable. The general-purpose solver used is LINGO, developed by LINDO Systems Inc [20].

The solution file is used to annotate the original sequential Fortran file with the data mapping and loop parallelization directives. This file is compiled using the xHPF [3] compiler form APR Inc, which generates a message-passing Fortran parallelized program. Finally, the Forge Performance Simulator is used to predict performance, and to validate the solutions generated.

The set of programs and routines selected[1] to evaluate this proposal is listed in Table 1. Routine *eflux* has been extracted from the FLO52 program in the Perfect Club, and *tred2* is a routine taken from the Eispack library. These have been analyzed in [13]. *rhs* is the more time consuming routine in the APPSP NAS benchmark, included in the xHPF benchmark set. From program BARO, also in the xHPF benchmark set, routines *intba1* and *comp* have been extracted. And finally, the *erlebacher* benchmark program is a three-dimensional tridiagonal solver using the Alternating Direction Implicit integration (also used in [4] to illustrate their dynamic layout techniques and selected here because of its complexity).

The static characteristics of the set of programs analyzed are summarized in Table 1. This includes the number of valid lines of code in each program (this is, the number of lines once all comment and null lines have been removed), the number of loops that can be executed in parallel in our model of architecture (this is, loops without any loop carried flow dependence), the number of reference patterns between different pairs of arrays and in brackets the total number of reference patterns, the number of arrays used in the programs, and their maximal dimensionality.

The complexity of the method is a function of the 0-1 integer variables required in the model, and this is a function of the number of different patterns, the maximal dimensionality of the arrays, and the number of possible parallel loops in the program. The last three columns in Table 1 summarize all these characterictics, and the time spent in finding the optimal solution for each analyzed program in a Sun SuperSparc 20. As a summary, one can see that the analysis of program *erlebacher* is the one that requires more time, as it is also the one that defines more number of 0-1 integer variables. Routine *comp* spends 1.3 seconds, and all other analyses spend less than a second.

Name	lines	parallel loops	patterns	arrays	Dim	0-1 variables	constraints	solver time (sec.)
eflux	58	11	4 (161)	5	3	47	24	0.2
tred2	96	11	8 (38)	4	2	43	31	0.3
rhs	353	37	5 (346)	4	4	117	63	0.5
intba1	71	10	15 (30)	10	2	70	73	0.5
comp	174	18	43 (162)	24	2	190	180	1.3
erlebacher	288	72	56 (153)	25	3	576	318	3.4

Table 1: Characteristics of the selected programs.

Finally, Table 2 summarizes the run-time behavior of the selected programs. In the second column the sequential execution time is listed. The third column reflects the simulated execution time, when parallelizing the programs and assuming a default mapping. The default mapping for each program is the distribution of the first dimension of all arrays, except for routine *rhs*, where the first dimension only contains five elements, and the second one has been selected as default. The speed-up achieved with this

1. If the programs selected contain routine calls, these have been inlined. And if the code selected is a routine, it has to be transformed into a program, replacing parameters and arguments by local variables.

distribution is listed in the fourth column. The fifth column shows the simulated execution time when evaluating the solution suggested by our tool, and its respective speed-up can be seen in the last column. All times are expressed in miliseconds and all parallel simulations have been done assuming 8 processors.

From this table one can see that the selected mapping by the tool for the first routine is the default one. The same was selected in [13]. For routine *tred2*, the selected mapping achieves a poor speed-up of 1.3, although the simulated execution time for the default one is more than 3 times the sequential execution time. In [13] the selected mapping was the default one, but in a cyclic manner. Our tool does not yet generate cyclic distributions. However, the simulated execution time for the cyclic distribution is very close to the default one, as all loops that can be parallelized (without performing any kind of loop transformation) just include one single statement, whereas the loops that can be parallelized with our mapping strategy are outer, and the benefits of parallelizing are higher. Routine *rhs* results in a speed-up of 4.2 with the suggested mapping, this is, distributing the third dimension of all arrays; while a speed-up of 3.6 is achieved with the default one. The parallel version of this routine given in the xHPF benchmark set[2] suggests a distribution of the fourth dimension of all arrays; the run-time behavior of these two alternatives is very similar. Routines *intba1* and *comp* also increase the simulated performance with respect to the default mapping, doubling its speed-up. The selected parallelization strategy is the same than the one suggested in the hand coded parallel version of these routines, also included in the xHPF benchmark set. Finally, program *erlebacher* reduces 30% the sequential execution time when distributing arrays as suggested by our tool, while the simulated execution time of the default distribution is almost twice the sequential one. The selected mapping strategy parallelizes almost all outer loops in the program. The reason for the poor speed-up is that there is a lot of communication involved, in particular some reductions not detected by the parser module of our tool.

Name	Sequential time	Parallel time for Default	Speed-up	Parallel time for Suggested	Speed-up
eflux	1727.6	282.6	6.1	282.6	6.1
tred2	707.3	2473.0	0.3	546.8	1.3
rhs	4371.6	1218.0	3.6	1053.0	4.2
intba1	400.9	138.7	2.9	81.3	4.9
comp	3618.6	1624.7	2.2	653.5	5.5
erlebacher	2571.1	4798.1	0.5	1795.5	1.4

Table 2: Run-time behavior of the selected programs.

5 Conclusions

This paper describes a novel strategy for finding a good data layout by applying an integer programming method, which is used by related research as well. The work follows the main stream in terms of automatic data distribution. The most interesting aspect of this solution

2. Source code for these benchmarks can be found via WWW to the URL http://www.infomall.org/apri/.

is that both alignment and distribution problems are combined in a single optimization phase. To do this, the Communication-Parallelism Graph (CPG) is defined, a data structure able to hold all the information required to solve the two problems altogether. The CPG contains edges that show communication constraints and edges that show parallelization constraints in the program. Edges are weighted according to their cost, in terms of data movement and computation time. Constraints in the graph are formulated using a linear 0-1 integer programming model, where the problem to solve is to find a path in the CPG that minimizes the overall execution time of the application. A general purpose linear programming solver has been used in our experiments.

The preliminary experimental results show the quality of the data distributions generated by the tool, and the feasibility of using linear 0-1 integer programming models to solve these kind of problems. In addition, it allows us to find an optimal solution in a small amount of time. We have compared the solution generated by the solver with a default solution that would be generated by a naive data distribution tool, and with the results of some other authors or hand coded parallel versions.

This approach is restricted to one-dimensional array distributions which is a severe drawback in view of real applications. The extension to multi-dimensional array distributions can be found elsewhere [11]. A lot of additional aspects should be considered in the problem formulation in order to improve the quality of the solutions generated, such as integrating communication optimizations (detection and elimination of redundant communication, overlapping of computation and communication, or combination of communication messages) or control flow analysis. A problem of this approach is the absence of varying problem sizes in the optimization model, which would requiere a symbolic solver.

Most of the ideas presented in this paper, can be also used to extend this approach to dynamic data mappings, where a set of computational intensive phases are detected, the mapping for each of them is obtained, and remapping actions (realignment and redistribution) between phases are introduced. This topic is also investigated by other research groups [4, 8, 9, 17].

6 Acknowledgements

We would like to thank the rest of the members of the DDTeam - Mercé Gironès, and M. Luz Grande - for their support for the implementation of this work, and to Uli Kremer for the fruitful discussions in the topic. The authors especially thank Elena Fernández for her insight comments and suggestions in the specification of the linear programming model. This work has been partially supported by the Ministry of Education of Spain under contract TIC880/92, the CEPBA (European Center for Parallelism of Barcelona) and by CONVEX Computer Corporation, that supports the development of the DDT tool.

7 References

1. E. Ayguadé, J. Garcia, M. Gironés, J. Labarta, J. Torres and M. Valero, “Detecting and Using Affinity in an Automatic Data Distribution Tool”. Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, August 1994.
2. E. Ayguadé, J. Labarta, J. Garcia and M. Gironés, “Data Partitioning Methods: Implementation

and Evaluation Reports". Technical Report UPC-DAC-94/28 and UPC-CEPBA-94/18, Computer Science Department, Universitat Politècnica de Catalunya, January 1994.

3. Applied Parallel Research Inc., "xHPF Version 1.2, User's Guide". May 1994 Release.
4. R. Bixby, K. Kennedy and U. Kremer, "Automatic Data Layout Using 0-1 Integer Programming". Proc. of the International Conference on Parallel Architectures and Compilation Techniques, August 1994.
5. B. Chapman, T. Fahringer and H. Zima, "Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems", 6th Annual Workshop on Languages and Compilers for Parallel Computing, Portland-Oregon, August 1993.
6. S. Chaterjee, J. R. Gilbert, R. Schreiber and S.-H. Teng, "Automatic Array Alignment in Data-Parallel Programs". Proc. of the 20th Annual ACM Symposium on Principles of Programming Languages, January 1993.
7. S. Chaterjee, J. R. Gilbert, R. Schreiber and T. J. Sheffler, "Array Distribution in Data-Parallel Programs". Proc. of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, August 1994.
8. S. Chaterjee, J. R. Gilbert and R. Schreiber, "Mobile and Replicated Alignment of Arrays in Data-Parallel Programs". Proc. of Supercomputing'93, November 1993.
9. P. Crooks and R. Perrott, "An Automatic Data Distribution Generator for Distributed Memory MIMD Machines". 4th Int. Workshop on Compilers for Parallel Computers, December 1993.
10. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng and M. Wu, "Fortran D Language Specification". Technical Report CRPC TR 90-141, Department of Computer Science, Rice University, December 1990.
11. J. Garcia, E. Ayguadé, and J. Labarta, "A Novel Approach Towards Automatic Data Distribution", 2nd Workshop on Automatic Data Layout and Performance Prediction, April 1995 (also available as Research Report CEPBA/UPC RR95-04).
12. M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang and K. Burke, "PTRAN II - A Compiler for High Performance Fortran". 4th International Workshop on Compilers for Parallel Computers, December 1993.
13. M. Gupta, "Automatic Data Partitioning on Distributed Memory Multicomputers". PhD thesis, University of Illinois at Urbana-Champaign, September 1992. Also available as technical report UILU-ENG-92-2237 and CRHC-92-19.
14. High Performance Fortran Forum, "High Performance Fortran Language Specification. Version 1.0". Scientific Programming, May 1993.
15. K. Kennedy, K. McKinley and C.-W. Tseng, "Interactive Parallel Programming Using the ParaScope Editor". Technical Report CRPC-TR90096, Center for Research on Parallel Computation, Rice University, October 1990.
16. U. Kremer, "NP-completeness of Dynamic Remapping". 4th International Workshop on Compilers for Parallel Computers, December 1993.
17. U. Kremer, J. Mellor-Crummey, K. Kennedy and A. Carle, "Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment". 1st International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction, January 1993.
18. J. Li and M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays". Proc. of the 3rd Symposium on the Frontiers of Massively Parallel Computation, October 1990.
19. J. Li and M. Chen, "Compiling Communication-efficient Programs for Massively Parallel Machines", IEEE Trans. on Parallel and Distributed Systems, vol. 2, no. 3, July 1991.
20. LINDO Systems Inc., "LINGO Optimization Modeling Language". © April 1994 by LINDO Systems Inc.
21. C. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines". PhD thesis, Rice University, January 1993. Rice COMP TR93-199.
22. H. Zima, P. Brezany, B. Chapman, P. Mehrotra and Schwald, "Vienna Fortran - A Language Specification". Technical Report, Austrian Center for Parallel Computation, University of Vienna, 1991.