INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Automatic Alignment of Array Data and Processes To Reduce Communication Time on DMPPs

Michael Philippsen*
phlipp @icsi.berkeley.edu

TR-94-070

July 1995

Abstract

This paper investigates the problem of aligning data and processes in a distributed-memory implementation. We present complete algorithms for compile-time analysis, the necessary program restructuring, and subsequent code-generation, and discuss their complexity. We finally evaluate the practical usefulness by quantitative experimentation.

The technique presented analyzes complete programs, including branches, loops, and nested parallelism. Alignment is determined with respect to offset, stride, and general axis relations. Both placement of data and processes are computed in a unifying framework based on an extended preference graph and its analysis. Furthermore, dynamic redistribution and replication are considered in the same technique.

The experimental results are very encouraging. The optimization algorithms implemented in the Modula-2* compiler improved the execution times of the programs by over 40% on a MasPar MP-1 with 16384 processors.

This paper appeared in: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 156–165, Santa Barbara, CA, July 19–21, 1995.

*On leave from Department of Computer Science, University of Karlsruhe, Germany

Contents

This paper appeared in: Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 156–165, Santa Barbara, CA, July 19–21, 1995.

Automatic Alignment of Array Data and Processes To Reduce Communication Time on DMPPs

Michael Philippsen*
ICSI, International Computer Science Institute, Berkeley, CA, phlipp@icsi.berkeley.edu

Abstract

This paper investigates the problem of aligning array data and processes in a distributed-memory implementation. We present complete algorithms for compile-time analysis, the necessary program restructuring, and subsequent code-generation, and discuss their complexity. We finally evaluate the practical usefulness by quantitative experiments.

The technique presented analyzes complete programs, including branches, loops, and nested parallelism. Alignment is determined with respect to offset, stride, and general axis relations. Pplacement of both data and processes are computed in a unifying framework based on an extended preference graph and its analysis. Dynamic redistributions are derived.

The experimental results are very encouraging. The optimization algorithms implemented in our Modula-2 compiler improved the execution times of the programs by an average over 40% on a MasPar MP-1 with 16384 processors.*

1 Introduction

Straightforward compilation of **forall** statements or array expressions and naive mapping of array elements onto distributed memory parallel processors (DMPP) usually result in a significant amount of interprocessor data motion. Therefore, data and process placement is an essential problem of several compiler projects targeting DMPPs.

There is agreement about the two goals of data and process placement: (1) Data locality. To reduce the amount of communication and achieve minimal runtime, all data elements which are used by a process should be stored locally on the same PE. (2) Parallelism. Using just one processor results in perfect data locality and minimal communication cost. In general, however, the runtime can be improved by exploiting the parallelism provided by the hardware. A trade-off between the conflicting goals must be found.

Whereas the goals are agreed upon, different approaches to reach them have been developed. Some of those require that data is mapped onto the topology by hand [24, 32], others are user guided and offer sets of directives for the compiler, abstract topologies (so-called templates), or interactive or knowledge-based environments that help determine the alignment of array dimensions and mapping functions [3, 15, 16].

Much recent work focuses on static compile-time analysis to automatically find good data decompositions for vector and data-parallel operations. We describe this work in more detail in section 2.

Placement optimization is often done in two steps. First, the *alignment* phase examines the relationship between arrays and determines in which way different array elements are used together and hence should be co-located. In the subsequent *distribution* phase, co-located array elements will be mapped to the same processor's local memory. While the first phase is machine-independent since only relative positions of array elements are considered, the second phase deals with absolute positions on the DMPP.

This paper is based on the following approach: We automatically determine an alignment of arrays *and* processes. This alignment is used in a source-to-source code transformation where user defined arrays are replaced by (possibly several) substitute arrays. These substitutes get distributed in the second phase with a fixed distribution scheme described elsewhere [25, 26].

The transformation is presented using Modula-2* [33] – a high-level, problem-oriented, and machine-independent parallel language – but is directly applicable to other languages, like HPF [16].

The remainder of the paper is organized as follows. After discussing related work in section 2, we briefly introduce our notation of a **forall**. Section 4 formulates the alignment optimization and discusses its properties. Finally, section 5 describes the setup and results of the experiments evaluating the effectiveness of our techniques.

2 Related Work

The two phase approach to placement (alignment + distribution) is used by Fortran D [10], HPF [16], CM-Fortran [31], and Vienna Fortran [3]. Although our work as well is based on the two phase approach, it is different, since it performs both phases automatically.

*On leave from Dept. of Informatics, University of Karlsruhe.

We consider it to be premature to have manual placement in languages; although the optimal placement is NP-complete [19, 21, 22, 23], an automatic solution of the placement problem is necessary for three reasons: First, Wholey has shown in [35] that the best placement depends on three factors: the topology of the network, the number of available processors, and the size of the problem. Hence, approaches that require the user to explicitly provide static or dynamic mappings, result in programs that may be source code portable but will show different runtime performance. Second, in different sections of a program different placements may be optimal. The programmer not only has to find an optimal placement for code segments, which is itself difficult as explained above, but s/he must find a global optimum that includes the cost of potential redistributions at certain points of the code. The necessary cost considerations are complex [1, 8] and in general undecidable since they require knowledge about compiler strategies, about network characteristics etc. Finally, even if the programmer provides explicit placement information for the declared data structures, it remains the task of the compiler to place temporaries.

Furthermore, we feel that both process and data alignment must be considered. Simply deriving process placements from data placements is suboptimal as has been shown for the owner-computes rule in [5, 17].

There are already approaches to automatic alignment. In comparison to our work – see [26, 28] for previous contributions – these have some restrictions.

Knobe et al. [17, 18] first introduced alignment analysis and the notion of preference graphs. Although they considered dimension, stride, and offset alignment, some problems remained. Their greedy algorithm for solving alignment conflicts often returns suboptimal solutions since cost estimates are basic and decisions are based on local information. Dynamic redistribution is considered only in special cases.

Neither dynamic redistribution nor loops or branches are considered by Wholey [35], Li and Chen [21, 22], and Gupta [13, 14]. Li and Chen only align dimensions of given arrays to each other. Gupta ignores offset alignment, e.g., communication resulting from an access to not co-located `A[i]` and `B[i+1]`. He does consider both alignment and distribution and hence uses a more accurate cost estimate than Li and Chen do.

The approach by Ramanujam [29] delivers good data placement unless conflicting alignment preferences exist. However, such conflicts occur commonly.

Kremer [2, 19, 20] takes a different approach. Instead of building placement optimization into a compiler, he develops tools to support the programmer in finding good placements. Although the approach is different, Kremer applies placement optimization techniques based on Li and Chen's Component-Affinity-Graph. He introduced the idea of using 0–1 integer programming for solving placement problems.

Most of the groups have two basic approaches in common, both of which result in suboptimal solutions: process placement is derived from data placement and placement of temporaries is not considered.

Gilbert, Schreiber, Chatterjee [4, 5, 12, 30] base their work on the Alignment-Distribution-Graph. In contrast to the approaches mentioned before, the authors tackle the problem of placing intermediate results. To our knowledge, their model for representing the cost of communication on the underlying topologies is the most advanced. It clearly is to be preferred over the simplistic cost model used here.

The main contributions of this paper are (1) the automatic computation of both data and process alignment in one framework, (2) an extended preference graph and a novel technique for its analysis, and (3) the performance results given in section 5.

3 Modula-2*

For the purpose of this article it is sufficient to understand the key features of Modula-2*. The only way to introduce parallelism into Modula-2* programs is by means of the **forall** *statement*, which has a synchronous and an asynchronous variant. The syntax of the **forall** statement is:

```
FORALL <ident> ":" <SimpleType> IN (PARALLEL | SYNC)
  <StatementSequence>
END.
```

SimpleType is an enumeration or a possibly *non-static* subrange, i.e. the boundary expressions may contain variables. The **forall** creates as many (conceptual) processes as there are elements in *SimpleType*. The identifier introduced by the **forall** statement is local to it and serves as a runtime constant *ident* for every process. The runtime constant of each process is initialized to a unique value of *SimpleType*.

Each process executes the statements in *StatementSequence*. The `END` of a **forall** statement imposes a *synchronization barrier* on the participating processes: the termination of the **forall** statement is delayed until all created processes have finished their execution of *StatementSequence*. In a synchronous **forall**, the created processes execute *StatementSequence* in lockstep, while in the asynchronous case, they can work concurrently.

The behavior of branches and loops inside synchronous **forall**s has MSIMD (multiple SIMD) semantics. This means that Modula-2* does not require any synchronization between different branches of synchronous **case** or **if** statements. The exact synchronous semantics of all Modula-2* statements, including nested **forall**s, are defined in [33].

4 Alignment Optimization

4.1 Example

```
VAR G: ARRAY [0..2*N-1]         OF INTEGER;
    H: ARRAY [0..N-1],[0..N-1] OF INTEGER;
G[.]   := ...
H[.,.] := ...
FORALL i:[0..N-2] IN PARALLEL
  G[2*i] := H[i,3] + H[i+1,4];
END
```

The optimal placement can be achieved in this simple example if the elements of `H` are stored with a stride of 2 in the first dimension so that `G[2*i]` is stored either where `H[i,3]` or where `H[i+1,4]` is. The array `G` is enlarged to be two-dimensional. There is a choice of storing `G[2*i]` together with `H[i,3]` or `H[i+1,4]`. Independent of the choice, process `i` executes where two of the three array elements are stored. Access to the third element needs communication. When `H[i,3]` is chosen, the placement algorithm transforms the above code into:

```
VAR G': ARRAY [0..2*N-1],[0..N-1] OF INTEGER;
    H': ARRAY [0..2*N-1],[0..N-1] OF INTEGER;
G'[.,.] := ...
H'[.,.] := ...
FORALL i:[0..N-2] IN PARALLEL (* ALIGN G'[2*i,3] *)
  G'[2*i,3] := H'[2*i,3] + H'[2*i+2,4];
END
```

4.2 Graph Representation

The basic data structure is a preference graph P. Nodes of P are array accesses and **forall** arrays (see 4.2.4) of a given program. The basic idea of a preference graph as introduced by Knobe [18] is that edges express alignment preferences. Co-location of arrays that are connected by these edges will result in locality of access. If edge constraints are not obeyed, arrays will be stored differently and communication is necessary at runtime. The type of edges we consider and the information attached to edges differentiate our work from earlier work based on preference graphs.

Whereas identity and conformance preferences (see 4.2.3) have been used by several groups, the activity and virtualization preferences (see 4.2.4) and the allocation information (see 4.2.1) used to decorate edges of P are unique aspects of the preference graph presented here. Moreover, **forall** arrays (see 4.2.4) are not considered elsewhere.

4.2.1 Allocation Information

For two nodes representing array accesses to `A` and `B`, a connecting edge is labeled with the *allocation information* $(\mathtt{A}, d_\mathtt{A}, s_\mathtt{A}, o_\mathtt{A}) \bowtie (\mathtt{B}, d_\mathtt{B}, s_\mathtt{B}, o_\mathtt{B})$. The first node is an access to `A` in dimension $d_\mathtt{A}$ as `A[...,`$s_\mathtt{A}$`i+`$o_\mathtt{A}$`,...]` with `i` being a **forall** variable. The second node is a similar access to array `B` using the same **forall** variable. Positions of the allocation information are marked with * if all dimensions are affected or if the index is not an affine expression.

4.2.2 Cost Model

In addition to the alignment information, edges are labeled with *costs*. The cost must be paid if the placement of the arrays cannot implement the co-location preference expressed by the edge. For this paper we use a simple cost model and do not try to represent the exact cost induced by not obeying an edge. In particular, we ignore the number of bytes to be communicated, the size of the packets, the communication pattern, and the distance. Our cost model differentiates:

c_{nw}	cost of a parallel network write of data elements
c_{nr}	cost of a parallel network read of data elements
c_f	cost of an asynchronous **forall** with empty body
c_s	cost of a synchronization barrier

As suggested by performance results [27] this simplistic cost model is not too far off for the MasPar when our layout algorithm [25] is applied. Better cost estimates are known and could be used instead.

In this representation, the problem is to find placements for all nodes minimizing the total cost of all edges that cannot be obeyed.

In 4.2.3 we discuss in detail the edges of P that are caused by data arrays. 4.2.4 extends P to **forall** statements and thus process scheduling.

4.2.3 Data Array Accesses

Identity Preferences. Nodes representing accesses to the same array are connected by identity edges if the following conditions hold: (1) At least one access is a write, (2) if one access is a read, it will be executed after the other, and (3) if both are writes, then the second one is partial, i.e., not all array elements get written.

An identity edge indicates that the placement of the array should not change between the execution of the connected nodes. Two reads do not induce identity edges since the second could use a different placement without problems. Two full writes do not require identity edges, since the old values and their placement vanish after the second write.

The allocation information of identity edges looks like: $(\mathtt{A}, *, *, *) \bowtie (\mathtt{A}, *, *, *)$. Hence, the placement of `A` is supposed not to change in any form.

If an identity edge cannot be obeyed, the whole array, say `A`, must be copied into another placement, say `A'`. For this purpose the following code is generated[1]:

```
FORALL i:[...] IN PARALLEL (* ALIGN A[i] *)
  A'[i] := A[i]
END;
```

The total cost of identity edges thus is $c_f + c_{nw} + c_s$.

Conformance Preferences. Nodes are connected by conformance edges if the arrays are accessed in the same statement and if the indexes use a **forall** variable. If the arrays are multidimensional and the **forall** variable occurs in more than one dimension, then there is one edge per dimension. For example, the statement `A[i]:=B[i,i]` will induce two conformance edges. If two nodes are both reads of the same array no conformance edge is necessary, since both nodes are already transitively connected by identity edges expressing the desired locality.

The allocation information of conformance edges looks like: $(\mathtt{A}, d_\mathtt{A}, s_\mathtt{A}, o_\mathtt{A}) \bowtie (\mathtt{B}, d_\mathtt{B}, s_\mathtt{B}, o_\mathtt{B})$. If the index expressions that use the **forall** variable are not affine, stride and offset use * instead of an explicit value.

Disobeying conformance edges results in network access at runtime. The cost of a conformance edge is estimated to be c_{nr}.

[1] To be more exact: Instead of `[i]` the allocation function computed in 4.4 must be used to access the arrays.

4.2.4 Integrating FORALL variables

To integrate data and process alignment a **forall** statement which declares `i:[1..n]` processes is *considered* to create a *conceptual array* of **forall** variables. Nested **forall** statements result in multidimensional arrays of **forall** variables. The scheduling strategy then is as follows: process i is scheduled where element `i` of that array would be stored. Hence, by determining a placement for array `[1..n]` a process scheduling is found.

Since it might be advantageous to schedule different statements of a **forall** differently, conceptually a node representing a **forall** array is introduced in P *per* statement of the body.

Activity Preference. The **forall** array introduced for a statement is considered to be written when the processes are scheduled. Inside the statement the elements of the **forall** array are read when user arrays are accessed. Similar to identity preferences there is an edge in P between the node representing the write and every occurrence of the **forall** variable in an index expression in the statement, i.e., the read node.

In the example below, two **forall** arrays (`FA1` and `FA2`) are introduced, one per statement. For the first statement two activity edges are in P: one between `A[i+1]` and `FA1[i]` and the other between `B[i+1]` and `FA1[i]`. For the second statement only one activity edge is in P.

```
FORALL i:[...] DO
  A[i+1] := B[i+1];     (*  FA1[i] *)
  C[i]   := 0;          (*  FA2[i] *)
END
```

The motivation for these edges is obvious. Even if `A` and `B` are perfectly aligned, the access is slow if the accessing processes (represented by `FA1`) have a different placement.

The allocation information of activity edges looks like $(\mathtt{A}, d_\mathtt{A}, s_\mathtt{A}, o_\mathtt{A}) \bowtie (\mathtt{FA1}, v, s_\mathtt{B}, o_\mathtt{B})$ with v being the nesting depth with respect to **forall** statements.

Since disobeying activity edges results in network access at runtime, the cost of an activity edge is c_{nr}.

Virtualization Preferences. Having one **forall** array per statement might result in different schedulings per statement which is in general too costly because of two reasons: Instead of one large virtualization loop comprising several statements, code for several small loops must be generated. Larger loops have more potential for optimization and use of registers. The second reason is the necessity to store and communicate control flow information. Consider a re-scheduling of processes inside an **if** statement in the **forall**. After re-scheduling, the processes must know about the result of the condition evaluation, therefore it must be communicated to the new placement.

The desire for large virtualization loops is reflected in the cost estimate of virtualization edges as follows. Disobeying a virtualization edge means re-scheduling of the processes which is implemented in Modula-2* by breaking the **forall** in two parts and scheduling each of the two **forall**s individually. Hence, the cost is $c_s + c_f + \delta \cdot c_{nw}$. The summand $\delta \cdot c_{nw}$ represents the cost of transmitting state information from one group of processes to the other. Iff this is necessary then $\delta = 1$, otherwise $\delta = 0$.[2]

Virtualization edges connect **forall** arrays of statements in the same **forall** that might follow on each other at runtime. The allocation information of virtualization edges looks like $(\mathtt{FA1}, *, *, *) \bowtie (\mathtt{FA2}, *, *, *)$. In the example four virtualization edges are in P.

```
FORALL i:[...] DO
  IF B[i] THEN                  (* FA1[i] *)
    U[i] := 0       (* FA2[i] *)
  ELSE                                      (* FA3[i] *)
    U[i] := 1
  END
  B[i] := TRUE;                 (* FA4[i] *)
END
```

4.2.5 Normalized Allocation Information

Completely specified allocation information can be normalized to a form where on one side of the $\bowtie$ sign, stride is 1 and offset is 0. The equivalence is: $(\mathtt{A}, d_\mathtt{A}, s_\mathtt{A}, o_\mathtt{A}) \bowtie (\mathtt{B}, d_\mathtt{B}, s_\mathtt{B}, o_\mathtt{B}) \Leftrightarrow (\mathtt{A}, d_\mathtt{A}, 1, 0) \bowtie (\mathtt{B}, d_\mathtt{B}, \frac{s_\mathtt{B}}{s_\mathtt{A}}, (o_\mathtt{B} - \frac{s_\mathtt{B}}{s_\mathtt{A}} o_\mathtt{A}))$

4.3 Central Idea

Finding placements for arrays represented by nodes in P requires us to map m dimensional arrays into n dimensional substitute arrays by computing *allocation functions*. An allocation function will be applied to the index expressions given in the program. An index expression $(i_1, i_2, \ldots, i_{m_\mathtt{A}})$ which is used to access an element of array `A` will be transformed into an index expression $(f_{\mathtt{A}1}(i_1, i_2, \ldots, i_{m_\mathtt{A}}), \ldots, f_{\mathtt{A}n}(i_1, i_2, \ldots, i_{m_\mathtt{A}}))$ to access the substitute array. We restrict our considerations to affine allocation functions of the form $f_{\mathtt{A}k}(i_1, i_2, \ldots, i_{m_\mathtt{A}}) = s_{\mathtt{A}k} \cdot i_\kappa + o_{\mathtt{A}k}$ with $1 \leq \kappa \leq m_\mathtt{A}$ and $s_{\mathtt{A}k}, o_{\mathtt{A}k} \in \mathbb{Z}$. We call $s_{\mathtt{A}k}$ *stride* and $o_{\mathtt{A}k}$ *offset* of the mapping to the substitute array.

In this notation, the substitute array of every array `A` occurring in a given program is specified by

- a *dimension mapping* $\triangleright_\mathtt{A} = \{1, 2, \ldots, m_\mathtt{A}\} \longrightarrow \{1, 2, \ldots, n\}$ which injectively assigns to each dimension of `A` a dimension of the substitute array,
- strides $s_{\mathtt{A}k}$ and offsets $o_{\mathtt{A}k}$ for each allocation function $f_{\mathtt{A}k} (1 \leq k \leq n)$.

If $n > m_\mathtt{A}$ there are $f_{\mathtt{A}k}$ that do not depend on an index of the original array ($s_{\mathtt{A}k} = 0$) but might have an offset $o_{\mathtt{A}k} \neq 0$. These degrees of freedom are used for runtime dependent allocation.

It is in general not possible to find dimension mappings and allocation functions for all data and **forall**

[2] This estimate can be refined by exploiting the fact that the synchronization barrier is not always necessary. But since code generation then becomes more intricate this must be left out of this paper due to space limitations.

arrays occurring in a program so that the locality preferences of *all* edges in P are obeyed. Usually there are conflicting preferences. We first show in 4.4 how these conflicts can be detected by processing the graph. Then section 4.5 presents heuristics to find a cheap solution with respect to edge costs, i.e., to find dimension mappings and allocation functions that will result in a high degree of locality and little remaining communication.

4.4 Conflict Detection

There are two classes of conflicts. Conflicts can occur in the allocation function and in the dimension mapping.

4.4.1 Conflicting Allocation Functions

Let the allocation function of an array A be given. The allocation function of a neighboring array in P can be computed by means of the allocation information as follows. An index $S_{\mathtt{A}}$ is mapped to the substitute array by $f_k(S_{\mathtt{A}}) = s_k S_{\mathtt{A}} + o_k$. If stride and offset of the allocation information are unspecified, s_k and o_k are copied to the neighboring node. If a conformance edge is attributed with the normalized allocation information $(\mathtt{A}, d_{\mathtt{A}}, 1, 0) \bowtie (\mathtt{B}, d_{\mathtt{B}}, \frac{s_{\mathtt{B}}}{s_{\mathtt{A}}}, (o_{\mathtt{B}} - \frac{s_{\mathtt{B}}}{s_{\mathtt{A}}} o_{\mathtt{A}}))$ the index expressions fulfill: $S_{\mathtt{A}} = \frac{s_{\mathtt{A}}}{s_{\mathtt{B}}} S_{\mathtt{B}} + (o_{\mathtt{A}} - \frac{s_{\mathtt{A}}}{s_{\mathtt{B}}} o_{\mathtt{B}})$. By using this equation in $f(S_{\mathtt{A}})$ the allocation function of $f_\kappa(S_{\mathtt{B}}) = s_\kappa S_{\mathtt{B}} + o_\kappa$ can be read as $s_\kappa = s_k \frac{s_{\mathtt{A}}}{s_{\mathtt{B}}}$ and $o_\kappa = s_k(o_{\mathtt{A}} - \frac{s_{\mathtt{A}}}{s_{\mathtt{B}}} o_{\mathtt{B}}) + o_k$. Based on this computation of allocation functions, conflicts can be detected:

> Consider a cycle of edges in P.
> Start with an arbitrary allocation function at an arbitrary node of the cycle. Compute the allocation function of neighboring nodes along the cycle. If the allocation information of an edge is incomplete, copy the allocation function to the neighboring node, otherwise use s_κ and o_κ as derived above.
> If after returning to the starting node an allocation function is computed, which is different from the initial one, then P has a conflict.

Note that useful allocation functions are computed if no conflict exists in the cycle. Possible non-integer values are removed by multiplying by the least common multiplier of the occurring denominators.

4.4.2 Conflicting Dimension Mappings

Since each dimension of a given array has to be mapped to exactly one dimension of the substitute array, an injective mapping $\triangleright_{\mathtt{A}} : \{1, 2, \ldots, m_{\mathtt{A}}\} \longrightarrow \{1, 2, \ldots, n\}$ is needed, where $m_{\mathtt{A}}$ and n are the number of dimensions of the given array and its substitute, respectively. The value of n is determined by the largest dimensionality in a given connected component of P. Allocation information of the form $(\mathtt{A}, d_{\mathtt{A}}, ., .) \bowtie (\mathtt{B}, d_{\mathtt{B}}, ., .)$ mean that dimension $d_{\mathtt{A}}$ of A and dimension $d_{\mathtt{B}}$ of B should both be mapped to the same dimension of the substitute arrays. Hence, the following implication is derived from the allocation information: $(d_{\mathtt{A}}, d_{targ}) \in \triangleright_{\mathtt{A}} \Longrightarrow (d_{\mathtt{B}}, d_{targ}) \in \triangleright_{\mathtt{B}}$. When the dimension mapping is computed elementwise the injectivity is destroyed if

$$\begin{array}{l} \exists d \in \{1, 2, \ldots, m_{\mathtt{B}}\} \backslash \{d_{\mathtt{B}}\} : (d, d_{targ}) \in \triangleright_{\mathtt{B}} \vee \\ \exists d \in \{1, 2, \ldots, n\} \backslash \{d_{targ}\} : \quad (d_{\mathtt{B}}, d) \in \triangleright_{\mathtt{B}} \end{array} \qquad (1)$$

Based on the above computation of dimension mappings, conflicts can be detected as follows.

> Consider a cycle of edges in P.
> Start with an arbitrary dimension mapping $\triangleright$ at an arbitrary node of the cycle. Compute the dimension mapping of neighboring nodes along the cycle using the above implication.
> Then P has a conflict, if one of the conditions given in (1) occurs before the starting node is reached.

Note that useful dimension mappings are computed if no conflict exists on the cycle.

4.5 Search Space and Complexity

In general, an optimal solution of the placement problem can be found in two steps. First, all cycles bearing a conflict must be detected in P by the above methods. *All* these cycles must then be cut to derive a placement. The difficulty is to find a set of edges that cuts all cycles and has the minimal total cost. Since this problem is NP-complete, heuristics must be used to prune the search space.

4.5.1 Fundamental Cycles

Instead of finding the set of all cycles, we restrict our considerations to the set of fundamental cycles[3] in P that bear a conflict. Nothing is lost by this restriction since all cycles in P can be constructed by combinations of fundamental cycles. Iff there is a conflict in a cycle in P there is a conflict in a fundamental cycle as well.

For the general solution *all* sets of fundamental cycles must be studied. For each graph P with n nodes, e edges, k components, each set has $\mu = e - n + k$ cycles, and there is one set for each spanning tree. We restrict our analysis:

> Find the *minimal* set of fundamental cycles with respect to the sum of the costs of the edges.

The underlying idea is as follows: It is more likely (but cannot be guaranteed) that the set of edges to cut with minimal cost is found in the minimal set of fundamental cycles than in any other set of fundamental cycles because of the minimality of total edge cost. Although the sub-problem of finding the minimal set of fundamental cycles is itself NP-complete, good polynomial time approximations are known [7].

4.5.2 Minimal Covering

Even if the minimal set of fundamental cycles is known, the remaining sub-problem still is NP-complete. The problem is to determine which of the edges to cut to

[3] If unfamiliar with these terms, see for example [6].

achieve minimal cost by cutting all cycles. This problem is another representation of the weighted set covering problem [11] since some edges belong to several cycles. It can be written as a linear programming problem:

For $i \in I$ numbering the edges in the set of fundamental cycles, let c_i be the cost of edge i and $x_i \in \{0,1\}$ be integer variables. When $x_i = 0$ the edge i remains uncut. For index sets $J_1, ..., J_\mu \subseteq I$ representing the edges of individual fundamental cycles use the simplex algorithm to solve:

$$\textbf{min} \sum_{i \in I} x_i \cdot c_i$$
$$\wedge \forall j \in \{1, ..., \mu\} : \sum_{i_j \in J_j} x_{i_j} \geq 1$$

Although the simplex algorithm cannot be guaranteed to terminate fast – its worst case behavior is in $O(\binom{l}{\mu})$ with l being the length of the longest fundamental cycle – it usually terminates in time proportional to the number of equations μ and variables l [34]. A simple basic solution is $x_i = 1$ for all $i \in I$.

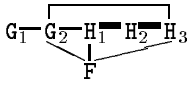
Results by Kremer [2, 20] also indicate the usefulness of integer programming for placement problems. Their use of 0–1 integer programming is restricted to subproblems, e.g., data remapping and axis alignment, and is intended for a tool supporting the programmer.

4.5.3 Replication

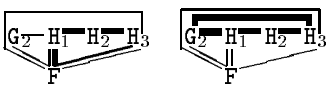
Replication is easily introduced in this scheme by consideration of node splittings. A cycle can not only be cut by splitting up one of its edges but additionally by replacing one node with two substitute nodes. This adds up to n new variables to the above system and enlarges each of the inequalities by the number of nodes in the corresponding fundamental cycle. Since node splitting and disobeyed identity edges both require that an array using one placement must be transformed into a second one, both have the same cost. It is still an unsolved question for us how to decide between edge splitting and node splitting, i.e., when to replicate. Nodes representing **forall** arrays or writes may not be split.

4.6 Example – Continued

For the example of section 4.1 eight edges are in P. Nodes are numbered in order of their appearance in the program, e.g., $\mathtt{H}_2$ represents `H[i,3]`. `F` is the **forall** array. Fat lines cost $c_n + c_f + c_s$, thin lines are c_n.



There are 24 spanning trees for P each of which has a set of 3 fundamental cycles. When considering only those cycles that bear a conflict, two minimal sets of fundamental cycles can be found. In the schematic representation only fat cycles have a conflict.



Although it is sufficient to consider one minimal set, we present both for explanatory reasons. The simplex algorithm finds, that in the first case either the edge $\mathtt{F} \bowtie \mathtt{H}_1$ or the edge $\mathtt{F} \bowtie \mathtt{H}_3$ must be cut to achieve minimal cost of c_n. For the second case either edge $\mathtt{G}_2 \bowtie \mathtt{H}_1$ or $\mathtt{G}_2 \bowtie \mathtt{H}_3$ is chosen with cost c_n. By removing any of these edges, dimension mappings and allocation functions are computed as shown in section 4.4. If in any case the first mentioned edge is cut, `G` will be placed according to `H[2*i+2,4]`. Otherwise, `H[2*i,3]` is selected which results in the transformation given in 4.1. Note that replication is not an issue here since the cost of node splitting always surpasses c_n.

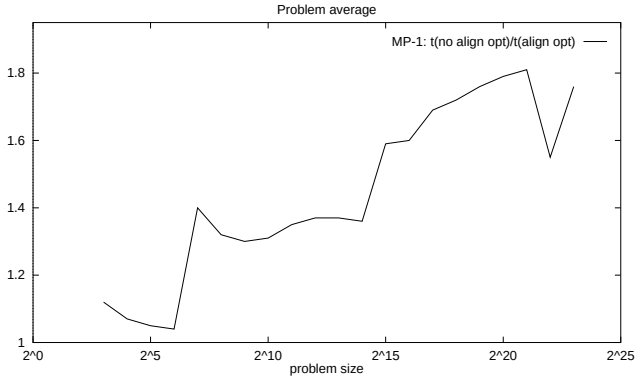
5 Performance Results

At the moment, our benchmark suite consists of 17 problems collected from literature, see [27] for details. Here, we only consider those 15 problems whose Modula-2* solutions are not totally aligned right from the beginning.

The programs were compiled for a 16K processor MasPar MP-1 (SIMD) by our Modula-2* compiler. Application of the automatic alignment optimization improved the execution times of the programs by over 40% on average. Because our work on Modula-2* compilers for MIMD machines, namely LANs of workstations and (virtual) shared memory multiprocessors, is still in progress, we cannot present any measurements for them. But we expect even better results since remote communication is more costly.

For time measurements we used the high resolution DPU timer on the MasPar. Below, $t_{align-opt}$ and $t_{no-align-opt}$ represent program execution times with the optimization techniques presented in the paper applied and not applied, respectively.

We define performance as work or problem size per time and focus on the following relative performances:[4] $\frac{size}{t_{align-opt}} / \frac{size}{t_{no-align-opt}} = t_{no-align-opt}/t_{align-opt}$. Thus, the diagrams show a ratio scale as the vertical axis. Good performance of the alignment optimization is indicated by curves above unity, e.g. a curve around 2 shows that the alignment optimization halved the execution time.



[4] Comparisons with hand-coded programs are given in [27].

For problem sizes ranging from 2^3 to 2^{23} we derived the relative performances from our execution time measurements. The resulting general, relative performances are shown above, averaged arithmetically over all test programs per problem size. (Only results with at least three measurement points per problem size are included in this average graph.)

Alignment optimization improves performance in two ways. Obvious improvement is due to achieving locality where without optimization remote access would occur.

A secondary improvement results from knowledge about existing locality, which the Modula-2* compiler exploits in the following way. For problem sizes above the number of processors, the compiler generates virtualization loops on each processor. The iteration variable can reflect the true value of the **forall** variable with respect to the section of the **forall** range that is assigned to a particular PE. Or it can just count iterations, starting from 0 on all PEs.

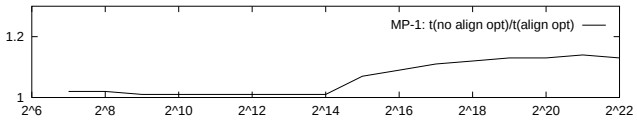
If locality of an array access is uncertain, first the corresponding processor number and the local address must be computed from the true **forall** variable, then a subsequent **if** statement must decide locality at runtime. In contrast, for known locality the iteration count can often be used for direct indexing into the local segment of an array, thus removing the runtime decision and often alleviating the cost of address calculations.

This often explains the increase of relative performance for problem sizes above machine size (2^{14}).

5.1 Problems

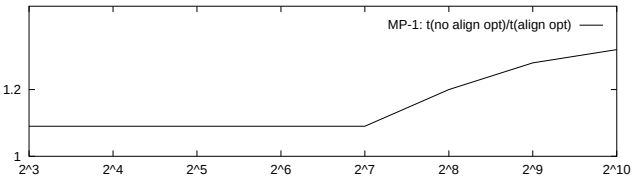
5.1.1 Root Search

Problem: Determine the value of $x \in [a, b]$ such that $f(x) = 0$, given that f is monotone and continuously differentiable. **Approach:** The problem is solved with multisection. The interval $[a, b]$ is evenly divided over n processes. If f has a root in $[a, b]$ then there is exactly one process p with $f(x_{p-1}) \cdot f(x_p) \leq 0$. Update the interval $[a', b'] := [x_{p-1}, x_p]$. Iterate until the error $b' - a' < \epsilon$. **Discussion:** Remote access cannot be reduced here. The improvement is due to knowledge about locality (see above).



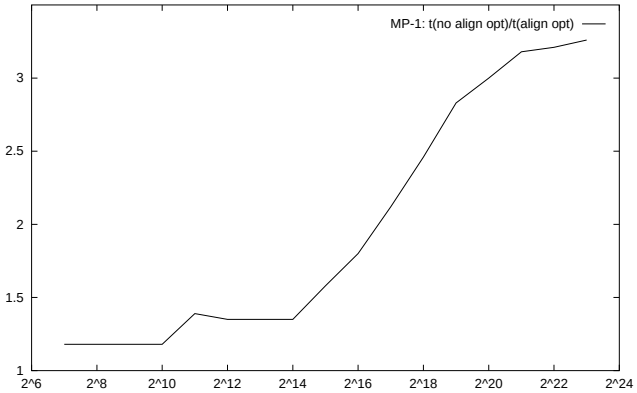
5.1.2 Heat Diffusion Kernel

Problem: The temperature on the edges of a square surface are given as constants, while those on the inside are to be calculated with a diffusion equation. **Approach:** The value of a grid point is iteratively computed based on the values of its neighbors. **Discussion:** See 5.1.1. Since the problem size is the length of one axis of the square surface, virtualization loops start at 2^7.

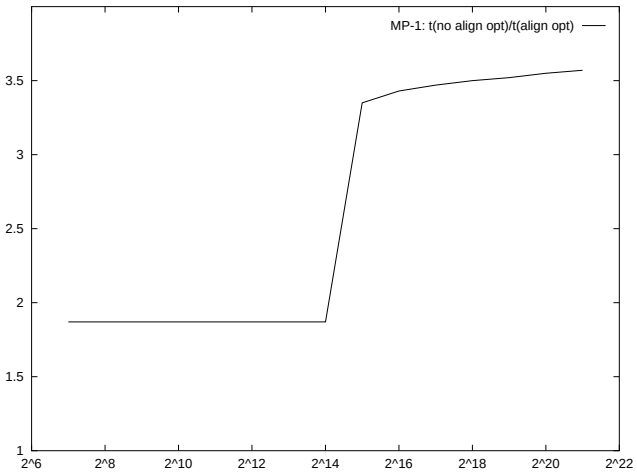


5.1.3 Doctor's Office

Problem: Given a set of n patients, a set of doctors, and a receptionist, model the following: Initially, all patients are well and all doctors are in a queue awaiting sick patients. Then patients become sick at random and enter a queue for treatment by one of the doctors. The receptionist handles the two queues, assigning patients to doctors. As soon as a doctor and a patient are paired, the doctor diagnoses the illness and treats the patient in a random amount of time. After curing a patient, the doctor rejoins the doctor's queue to await another patient (from [9]). **Approach:** The random amounts of time that patients are well and that doctors need to treat illnesses are counted down in parallel. The assignments of doctors to patients is done in parallel. The output is a list of timestamps, indicating when patients became ill, and list of triples (doctor, patient, treatment time). **Note:** The vertical axis is scaled differently. **Discussion:** See 5.1.1.



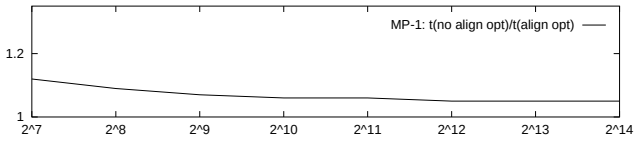
5.1.4 Synchronous Example



Problem: This is an example of a Modula-2* program with one synchronous **forall** that does a lot of (unmotivated) array operations. **Note:** The vertical axis is scaled differently. **Discussion:** See 5.1.1.

5.1.5 Longest Common Subsequence

Problem: Two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ are given. Find a string $C = c_1 c_2 \cdots c_p$ such that C is a longest common subsequence of A and B. (C is a subsequence of A if it can be constructed by removing elements from A without changing their order.) **Approach:** The solution uses a wave-front implementation of dynamic programming. It causes intensive access to neighboring data elements. **Discussion:** Currently, access to neighboring data elements is implemented with global communication primitives. Since relative overhead of the work incurred by unnecessary virtualization loops will increase when faster grid communication can be used instead, we expect better results on SIMD machines in future.



5.1.6 Red/Black Iteration

Problem: Implement a red/black iteration, i.e., the kernel of a solver for partial differential equations. **Approach:** The implementation intensively references neighboring data elements in a $n \cdot n$-matrix. **Discussion:** Since the diagram is similar to the one of 5.1.5 it is omitted.

5.1.7 List Rank

Problem: A linked list of n elements is given in an array $A[1..n]$. Compute for each element its rank in the list. **Approach:** This problem is solved by pointer jumping. **Discussion:** Since the diagram is similar to the one of 5.1.5 it is omitted.

5.1.8 Transitive Closure

Problem: The adjacency matrix of a directed graph with n nodes is given. Find its transitive closure. **Approach:** Process the adjacency matrix according to the property that if nodes x and m as well as nodes m and y are (transitively) adjacent, then x and y are (transitively) adjacent. **Discussion:** Since the diagram is similar to the one of 5.1.5 it is omitted.

5.1.9 Prime Sieve

Problem: Compute all prime numbers in $[2..n]$. **Approach:** Rather than using a virtual process per candidate, our implementation of the classical prime sieve assigns a segment of candidates to each processor. This adaptive version works much faster since division can be replaced by indexing within each segment. **Discussion:** Since the diagram is similar to the one of 5.1.5 it is omitted.

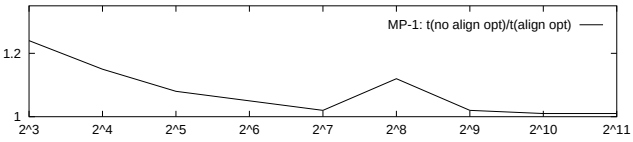
5.1.10 Game of Life

Problem: Apply Conway's rules of life to a given matrix. **Approach:** The value of a grid point depends on the sum of the values of its neighbors. **Discussion:** Since the diagram is similar to the one of 5.1.5 it is omitted.

5.1.11 Pairs of Relative Primes

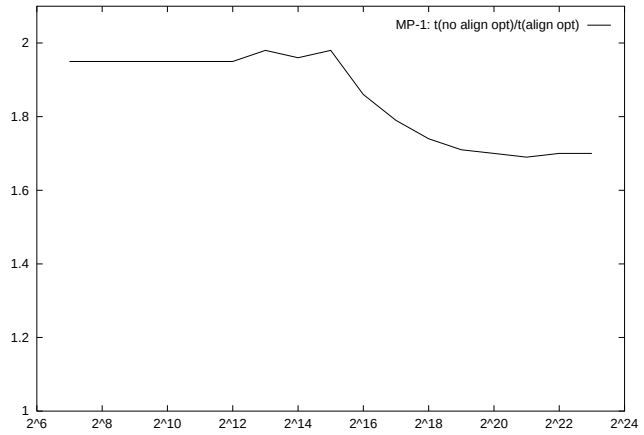
Problem: Count the number of pairs (i, j) with $2 \leq i < j \leq n$ that are relatively prime, i.e. the greatest common divisor of i and j is 1. **Approach:** The solution is based on a data-parallel implementation of the GCD algorithm followed by an add-scan. **Discussion:** The relative effectiveness of the optimization depends on the relation between data access time and computation time in the program. If the computation time is predominant improvement of data access shows only little effect. This benchmark is a good example.

The parallel invocation of the GCD function and the while loop inside are the dominant cost producers. Due to the SIMD model, the overall runtime is determined by the pair of numbers that requires the most iterations. Up to a problem size of 2^7 the virtualization ratio is 1, hence smaller problems mean fewer and in general less complicated pairs to consider and thus less computation and a better effect of the optimization. Starting with 2^8 the virtualization loops are iterated more than once. At first, newly added GCD invocations terminate fast. As more pairs are considered, more computation becomes predominant again.



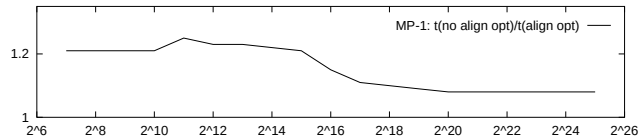
5.1.12 Point in Polygon

Problem: A simple polygon P with n edges and a point q are given. Determine whether the point lies inside the polygon. (A polygon is simple if pairs of line segments do not intersect except at their common vertex.) **Approach:** Draw a line from q that is parallel to the vertical axis. Count the number of intersections with P. The point q lies inside P if and only if this number is odd. **Discussion:** Up to the machine size, access to a locally stored array element is implemented as access to a local variable. For larger problem sizes, local access needs arrays and the computation of index expressions, which slightly increases computation time and hence reduces the effect of the optimization.



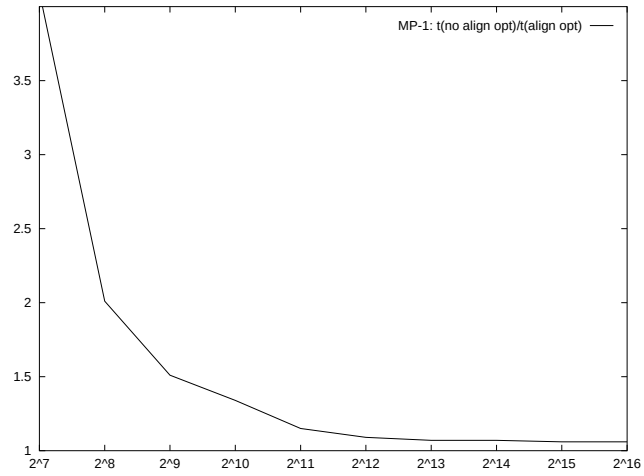
5.1.13 Estimation of Pi

Problem: Compute π using the equation $\pi = \int_0^1 \frac{4}{1+x^2}$. **Approach:** Approximate the solution by computing $\frac{1}{n}\sum_{i=0}^{n-1} \frac{4}{1+x_i^2}$ (rectangular rule), where n is the problem size parameter and $x_i = (i + \frac{1}{2})/n$ is the midpoint of the ith interval. **Discussion:** See 5.1.12.



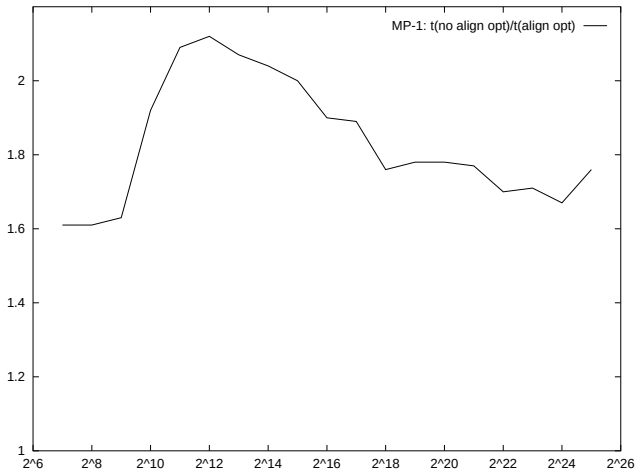
5.1.14 Paraffins Problem

Problem: Given an integer n, output the chemical structure of all paraffin molecules for $i \leq n$, without repetition and in order of increasing size. Include all isomers, but no duplicates (from [9]). **Note:** The vertical axis is scaled differently. **Discussion:** The unavoidable amount of remote communication becomes the predominant cost factor with growing problems. The optimization is most effective for smaller problem sizes.



5.1.15 Hamming's Problem

Problem: A set of primes $\{a, b, c, \ldots\}$ of arbitrary size and an integer n are given. Find all integers of the form $a^i \cdot b^j \cdot c^k \cdot \ldots \leq n$ in increasing order and without duplicates. **Approach:** For each prime p compute $\{p^i | p^i \leq n\}$. Combine any two power sets to a new one, while enforcing that the products remain $\leq n$. Repeat the combination for all power sets. **Discussion:** The curve is due to a combination of the effects described in 5.1.1 and in 5.1.12.



6 Conclusion

In this paper we presented evidence that in many cases the problem of determining an efficient alignment of data and processes can be solved automatically.

The technique presented analyzes complete programs, including branches, loops, and nested parallelism. Alignment is determined with respect to offset, stride, and general axis relations. Both placement of data and processes are computed in a unified framework based on an extended preference graph and its analysis. Dynamic redistributions are derived.

The main contributions of this paper are (1) the automatic computation of both data and process alignment in one framework, (2) an extended preference graph and a novel technique for its analysis, and (3) the performance results which are very encouraging.

On average, the optimization algorithms implemented in our Modula-2* compiler improved the execution times of the programs by on average over 40% on a MasPar MP-1 with 16384 processors.

The IPD Modula-2* system is available by anonymous ftp from ftp.ira.uka.de in pub/programming/modula2star.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. *Sigplan Notices* 26(7):213–223, 1991.

[2] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic data layout using 0-1 integer programming. Tech. Report CRPC-TR93349-S, Rice University, 1993.

[3] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *6th Distrib. Memory Computing Conf.*, pp. 51–58, Portland, OR, 1991.

[4] S. Chatterjee, J. Gilbert, and R. Schreiber. The alignment-distribution graph. In *6th Workshop on Languages and Compilers for Parallelism*, pp. 234–252, Portland, OR, 1993.

[5] S. Chatterjee, J. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *20th ACM Symp. on Principles of Programming Languages*, pp. 16–28, 1993.

[6] N. Deo. *Graph Theory with Appl. to Engineering and Computer Science.* Prentice Hall, 1974.

[7] N. Deo, M. Prabhu, and M.S. Krishnamoothy. Algorithms for generating fundamental cycles in a graph. *ACM Trans. on Mathem. Software*, 8(1):26–42, 1982.

[8] T. Fahringer, R. Blasko, and H. Zima. Static performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Int. Conf. on Supercomputing*, pp. 347–356, Washington, 1992.

[9] J.T. Feo, editor. *A Comparative Study of Parallel Programming Languages.* Elsevier , Holland, 1992.

[10] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Tech. Report CRPC-TR90079, Rice University, 1990.

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, 1991.

[12] J. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *J. Parallel and Distributed Computing*, 13(1):58–64, 1991.

[13] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *6th Distrib. Memory Computing Conf.*, pp. 43–50, Portland, OR, 1991.

[14] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *CACM*, 35(8):66–80, 1992.

[16] High Performance Fortran: Language specification. Tech. Report, CRPC, Rice University, 1992.

[17] K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *3rd Workshop on Compilers for Parallel Computers*, pp. 394–404, Vienna, Austria, 1992.

[18] K. Knobe, J.D. Lukas, and G.L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. Parallel and Distributed Computing*, 8(2):102–118, 1990.

[19] Ulrich Kremer. NP-completeness of dynamic remapping. Tech. Report CRPC-TR93330-S, Rice University, 1993.

[20] Ken Kennedy and Ulrich Kremer. Automatic data layout for high performance computing. Tech. Report CRPC-TR94498-S, Rice University, 1994.

[21] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *3rd Frontiers of Massively Parallel Computation*, pp. 424–433, 1990.

[22] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel and Distributed Computing*, 13(4):213–221, 1991.

[23] M.E. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, 1987.

[24] MasPar Computer Corporation. *MasPar Parallel Application Language Reference Manual*, 1990.

[25] M. Philippsen. Automatic data distribution for nearest neighbor networks. In *4th Frontiers of Massively Parallel Computation*, pp. 178–185, 1992.

[26] M. Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* PhD thesis, University of Karlsruhe, Informatics, 1993.

[27] M. Philippsen, E.A. Heinz, and P. Lukowicz. Compiling machine-independent parallel programs. *Sigplan Notices*, 28(8):99–108, 1993.

[28] M. Philippsen and M.U. Mock. Data and process alignment in Modula-2*. In *Automatic Parallelization: New Approaches*, pp. 177–191. Verlag Vieweg, 1994.

[29] J. Ramanujam and P. Sadayappan. Access based data decomposition for distributed memory machines. In *6th Distributed Memory Computing Conf.*, pp. 196–199, Portland, OR, 1991.

[30] T. Sheffler, R. Schreiber, J. Gilbert, and S. Chatterjee. Aligning parallel arrays to reduce communication. In *5th Frontiers of Massively Parallel Computation*, pp. 324–331, 1995.

[31] Thinking Machines Corporation, Cambridge, Massachusetts. *CM-Fortran Reference Manual*, 1989.

[32] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, 1991.

[33] W.F. Tichy and C.G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Tech. Report 4/90, University of Karlsruhe, 1990.

[34] S. Walukiewicz. *Integer Programming.* Kluwer Academic Publishers, 1991.

[35] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, CMU, 1991.