This paper appeared in: In AP'93, Int. Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction, Saarbr"ucken, Germany, March 1-3, 1993. Verlag Viewe(g)
Wiesbaden, Germany, Advanced Studies in Computer Sciene Series, 1994.

# 11 Data and Process Alignment in Modula-2*

Michael Philippsen and Markus U. Mock
University of Karlsruhe
Department of Informatics
D-76128 Karlsruhe, F.R.G.
email: (phlipp|mock)@ira.uka.de

**Abstract:** Exploiting locality is a central goal of translating problem-oriented parallel programming languages for distributed memory parallel machines. Modula-2* places the burden of automatically deriving good data *and* process distribution on the compiler.

In this paper we present a technique implemented in our optimizing compiler that enhances locality in a source-to-source transformation. Analysis of data access patterns and parallel operations leads to an arrangement graph. Processing of this graph reveals conflicting arrangements. Some assumptions and a heuristic based on dynamic programming enables the compiler to find the best alignment in logarithmic time. The technique has improved runtime performance on benchmarks by over 60%.

## 11.1 Introduction

Straightforward compilation of `FORALL` statements and allocation of array elements onto massively parallel machines results in a significant amount of interprocessor data motion. Therefore, data and process distribution is an essential problem of numerous compiler projects targeting distributed memory machines.

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

Whereas the goals are agreed upon, totally different approaches to reach them have been developed. In many programming languages the user must explicitly provide the data layout. Some languages require an explicit mapping of the data onto the topology [1, 14, 11], others are more abstract and offer either sets of directives for the compiler or interactive or knowledge-based environments that help determine the alignment of array dimensions and mapping functions [4, 10, 8, 3, 2]. Recent work [6, 7, 5, 16, 9] focuses on static compile-time analysis to automatically find a data decomposition that achieves both goals for vector and data-parallel operations.

Modula-2* [17] is designed for high-level, problem-oriented, and machine-independent parallel programming. The programmer can focus on the problem he has to solve, abstracting from the available number of processors and the interconnection network. Therefore, the compiler has to determine an appropriate data and process distribution.

Known approaches to automatically derive good data allocations have been targeting pure data-parallel programming languages, i.e. the parallelism has come from vector manipulations. In these approaches it is sufficient to find good data allocations. Locality is achieved by applying the owner-computes rule to distribute the statement execution onto the processors accordingly.

Modula-2*, however, is not a purely data-parallel programming language. When designing Modula-2*, we wanted to preserve the main advantages of data-parallel languages while avoiding the drawbacks [13]. Although data-parallel programming is possible, the notion of *process* is present. Therefore, both data *and* process distribution must be found by the compiler.

In this paper we present our approach to derive both data and process distribution for Modula-2* programs. Our technique is based on the work of Knobe [7] but extends her ideas with the consideration of process distribution and the clear separation of high-level data arrangement and physical data layout.

In section 11.2 we present the basic characteristics of Modula-2*. Section 11.3 explains the general approach of the Modula-2* compiler. In sections 11.4 and 11.5 we give some more details on the alignment graphs, the conflict detection, and the heuristic search mechanism.

## 11.2 Modula-2*

The programming language Modula-2* was developed to allow for high-level, problem-oriented and machine-independent parallel programming. As described in [17], it provides the following features:

- An arbitrary number of processes operate on data in the same *single address space*. Note that shared memory is not required; a single address space merely permits all memory to be addressed, but not necessarily at uniform speed.

- Synchronous and asynchronous parallel computations as well as arbitrary nestings thereof can be formulated in a totally machine-independent way.

- Procedures may be called in any context (sequential, synchronous, or asynchronous) at any nesting depth. Furthermore, additional parallel processes can be created inside procedures (recursive parallelism).

- All abstraction mechanisms of Modula-2 are available for parallel programming.

Modula-2* extends Modula-2 with just two language constructs:

1. The only way to introduce parallelism into Modula-2* programs is by means of the `FORALL` *statement*, which has a synchronous and an asynchronous version.
2. The distribution of array data is optionally specified by so-called *allocators*. These machine-independent allocators do not have any semantic meaning. They are just hints about data layout for the compiler.

Because of the compactness and simplicity of these extensions, they could easily be incorporated into other imperative programming languages, such as Fortran, C, or Ada.

### 11.2.1 FORALL statement

In Modula-2*, the syntax of the `FORALL` statement is:

```
ForallStatement =
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 for every process created by the `FORALL`. The runtime constant of each process is initialized to a unique value of *SimpleType*.

Each process created by a `FORALL` executes the statements in *StatementSequence*. The `END` of a `FORALL` statement imposes a *synchronization barrier* on the participating processes: the termination of the whole `FORALL` statement is delayed until *all* created processes have finished their execution of *StatementSequence*.

In a synchronous `FORALL`, the created processes execute *StatementSequence* in lock-step, while in the asynchronous case, they work concurrently.

The behavior of branches and loops inside synchronous `FORALL`s is defined with a MSIMD (multiple SIMD) machine in mind. 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 [17].

### 11.2.2 Allocation of array data

Modula-2* provides a simple, machine-independent construct for controlling the allocation of array data. This construct is optional and does not change the meaning of a program. The modified declaration syntax for arrays is:

```
ArrayType =
ARRAY SimpleType [allocator]
      {"," SimpleType [allocator]} OF type.
allocator =
LOCAL | SPREAD | CYCLE | RANDOM | SBLOCK | CBLOCK.
```

Array elements whose indices differ only in dimensions that are marked `LOCAL` are associated with the same processor. This facility is used to avoid distribution of data in a given dimension.

Dimensions with allocator `SPREAD` are divided into segments, one for each of the available processors. A vector with $n$ elements is assigned to $P$ processors by allocating a segment of length $\lceil n/P \rceil$ to each processor. While utilizing all available processors, it minimizes the cost of nearest-neighbor communication.

Dimensions with allocator `CYCLE` are distributed in a round-robin fashion over the available processors. Given $P$ processors, the elements of a vector whose indices are identical modulo $P$ are associated with the same processor. In contrast to `SPREAD`, `CYCLE` maximizes the cost of nearest-neighbor communication: neighboring array elements are always on different processors, leading to better processor utilization if a parallel algorithm operates on subsegments of a vector.

Dimensions with allocator `RANDOM` are distributed randomly over the available processors. In contrast to `CYCLE`, `RANDOM` leads to a better processor utilization if a parallel algorithms accesses the dimension in a random pattern.

If either `SPREAD`, `CYCLE`, or `RANDOM` apply to several successive dimensions, then these dimensions are "unrolled" into one pseudo-vector with a length that is the product of the lengths of the individual dimensions. This scheme idles fewer processors than applying `SPREAD`, `CYCLE`, or `RANDOM` to individual dimensions.

Allocators `SBLOCK` and `CBLOCK` apply `SPREAD` and `CYCLE` resp. to each dimension individually. For two successive dimensions, `SBLOCK` has the effect of creating rectangular subarrays and assigning those to the processors. With this arrangement, nearest-neighbor communication in all dimensions is best supported when the interconnection network can be configured into the same number of dimensions as the arrays.

`CBLOCK` for two dimensions also creates two-dimensional subarrays, but the rows and columns of these subarrays are then distributed in a round-robin fashion over the processor grid. Again, `SBLOCK` minimizes nearest-neighbor communication, while `CBLOCK` allows high processor utilization if smaller subarrays are processed in parallel.

## 11.3 Alignment in Modula-2*

In this section we present the general ideas of our data and process alignment strategies.

Data *layout* is the decision which element of an array is physically stored on which processor. *Arrangement* is the process of arranging array elements so that the elements of different arrays which are used together will end up in the same processor.

Although arrangement and layout are seen as one step in the literature, we propose to separate these issues into two phases:

Alignment = Arrangement + Layout

### 11.3.1 Data Alignment

In terms of Modula-2* we use a source-to-source transformation in the first phase to achieve the arrangement. For the second phase we have developed an adequate layout algorithm [12] that maps arrays onto the machine depending on their declarations. Consider the following example:

```
VAR A: ARRAY [1..90]  SPREAD OF INTEGER;
    B: ARRAY [0..100] SPREAD OF INTEGER;

BEGIN
  FORALL i:[1..90] IN SYNC
    A[i] := B[i-1];
    B[i] := 0
  END
END
```

To arrange arrays `A` and `B`, array `A` is enlarged and shifted to the left. All index expressions involved are transformed accordingly. After this, elements which are used together have the same index. Note that the new array `A` is larger than the old one. Since the primary goal of our optimization is runtime performance we allow for moderate waste in storage consumption.

```
VAR (* A: ARRAY [1..90] SPREAD OF INTEGER; *)
    A,B : ARRAY [0..100] SPREAD OF INTEGER;

BEGIN
  FORALL i:[1..90] IN SYNC
    A[i-1] := B[i-1];
    B[i]   := 0
  END
END
```

The analysis would not arrange arrays `A` and `B` if the programmer had used different allocators. In this case, the compiler issues a performance warning, which suggests to reconsider the used allocators. If the programmer does not use any allocator, the compiler selects an appropriate one.

In the second phase, the layout algorithm maps both arrays to the available processors in the same way. Since both arrays have the same declaration, elements with the same index end up in the same processor. Our layout algorithm, which is described in [12], reaches the following goals: (a) Exploit fast communication patterns if there is special hardware support, e.g. nearest-neighbor networks. (b) Perform simple address calculations. The computation of processor numbers and addresses of data elements are fast shift and mask operations.

### 11.3.2 Process Alignment

Up to now we have only dealt with the data alignment and its realization. Process alignment is also achieved by means of a source-to-source transformation. For this purpose, we have augmented the `FORALL` statement as follows:

```
ForallStatement =
FORALL ident ":" SimpleType IN (PARALLEL | SYNC)
[ALIGNED WITH Designator]
  StatementSequence
END.
```

The `ALIGNED WITH` term is not present in the original Modula-2* program. It is derived by compile-time analysis. In the example the transformation results in:

```
VAR (* A: ARRAY [1..90] SPREAD OF INTEGER; *)
    A,B : ARRAY [0..100] SPREAD OF INTEGER;

BEGIN
  FORALL i:[0..89] IN SYNC ALIGNED WITH B[i]
    A[i] := B[i]
  END;
  FORALL i:[1..90] IN SYNC ALIGNED WITH B[i]
    B[i] := 0
  END
END
```

The code generator then simply considers the range of the `FORALL` as an array and invokes the layout algorithm to determine which processor has to simulate which of the conceptual processes in a virtualization loop. In the above example the original `FORALL` has been split into two parts. In both `FORALL`s the process with index `i` will be executed where data element `B[i]` resides, resulting in purely local accesses. This could not be achieved with a single `FORALL`. Furthermore, providing the code generator with exact alignment information facilitates easy exploitation of nearest-neighbor communication networks.

The arrangement does not always work that smoothly. In general, there are lots of alignment preferences both for data usage and process alignment. Additionally, suitable cost estimation is required. Depending of the overhead cost of splitting up a `FORALL`, it may be advantageous on particular parallel hardware to accept some non-locality instead.

The following two sections are more specific and show our arrangement algorithm in some detail.

## 11.4 Arrangement Graphs and Conflicts

During static compile-time analysis we create an arrangement graph. Nodes of this graph are array references of arbitrary type and `FORALL`-variables. Edges express arrangement preferences and are attributed with the *type* and the *structure* of the detected preference.

### 11.4.1 Type and Structure

We found four types of arrangement preferences to be necessary. The first two types were introduced by Knobe and provide data arrangement information.

- An *identity preference* is an arrangement request that relates a defining occurrence of an array to a using occurrence of the same array. It indicates a preference to align identical elements of the array on the same processor for the two occurrences. The idea is to avoid redistribution cost.
- A *conformance preference* relates two array occurrences that are operated on together in a parallel expression. The goal is to group elements of different arrays so that all data accesses can be done locally.

Knobe has introduced a third preference for expressing data arrangement information. An *independence anti-preference* is a property of specific array dimensions if these dimensions contain a potentially parallel subscript. For analysis of Modula-2*, this type of preference is not necessary, because of (a) the allocators already indicate distributed storage and (b) the explicitness of parallelism in array subscripts inside of `FORALL` statements.

The next two types of arrangement preferences are used to gather information for process alignment.

- A *process preference* relates the `FORALL`-variable to the leftmost occurrence (LMO) of an array reference if the following conditions are fulfilled: (a) The LMO is in a statement inside of the body of that `FORALL` and (b) the `FORALL`-variable appears in the subscript expression of the LMO. Any other array occurrence fulfilling (a) and (b) could be chosen as well.

Arranging the processes with all LMOs in the body of the FORALL will achieve perfect locality of processes and data that is accessed in parallel. The process will run where the data is located. Since conformance preferences already ensure that all data which is operated on together will be arranged, only LMOs are considered.

- An *LMO preference* relates two successive LMOs of the same array in the body of a FORALL if these are subscripted in the same dimension with an expression using the same FORALL-variable. LMO preferences represent the cost of splitting up FORALLs, i.e. the increased virtualization overhead. If all LMO preferences are honored the FORALL will not be split up.

The arrangement graph contains all four types of edges. If only the first or the last two types are considered the graph is called either *data* arrangement graph or *process* arrangement graph.

For afine index expressions, the edges are labeled with the preferred arrangement structure. For two arrays A and B, this becomes ALIGN (A,$d_A$,$s_A$,$o_A$) WITH (B,$d_B$,$s_B$,$o_B$) for all types of preferences except process preferences. $d_A$ and $d_B$ are the numbers of the dimensions that impose the preference, and the subscript expressions $s_A\cdot$ i+$o_A$ and $s_B\cdot$i+$o_B$ denote elements that should be arranged in the specified array dimensions. Normalization results in structure information of the form ALIGN (.,.,1,0) WITH (.,.,$c$,$d$). For LMO preferences we have $s_A = 1$ and $o_A = 0$. Analogously, process preferences have the structure ALIGN i WITH (A,$d_A$,$s_A$,$o_A$) since only one node is an array occurrence.

### 11.4.2 Conflicts

The arrangement graph usually is not free of conflicts. In general, it is impossible to arrange data elements and processes in a way that all accesses are local without any redistribution of data or processes. We distinguish between data arrangement conflicts and process arrangement conflicts.

*Data Arrangement Conflicts*

In the following example the data arrangement graph (see Figure 11.1) is cyclic.

```
A[.] := ...
B[.] := ...
FORALL i : [1..N] IN PARALLEL
  s[i] := A[2*i] + B[i];
  t[i] := A[ i ] + B[i]
END
```

We do not consider the edges to or from occurrences of `s` and `t`, since these do not contribute to the cycle. There are two conformance preferences inside of the `FORALL`. The first one is caused by the first assignment in the `FORALL`. It relates `A[2*i]` to $B_1$`[i]`. The second one relates the array occurrences `A[i]` and $B_2$`[i]` of the second assignment.

All array occurrences inside of the `FORALL` are related to their defining occurrences in front of the `FORALL` with identity preferences. Thus, there are four identity preferences between (`A[.]`, `A[2*i]`), (`A[.]`, `A[i]`), (`B[.]`, $B_1$`[i]`), and (`B[.]`, $B_2$`[i]`).

It is impossible to achieve locality between the elements `A[2*i]` and $B_1$`[i]`, demanded by the conformance preference of the first assignment, and at the same time honor the second conformance preference between `A[i]` and $B_2$`[i]`.

In our approach, we avoid data redistribution at run-time inside of `FORALL`s. Therefore, there are two possible data arrangements. In both cases, one conformance preference is honored, the other one is broken.

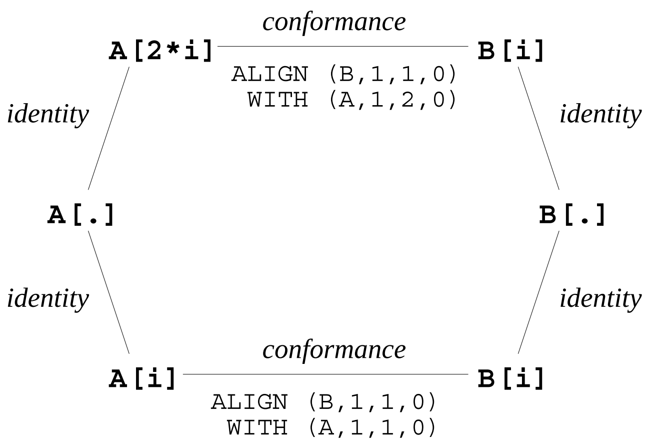

**Figure 11.1** Data Arrangement Graph

To determine all possible arrangements, we apply the following algorithm to each cycle in the data arrangement graph:

1. Start with $a$:=1, $b$:= 0 at an arbitrary node $N$ of the cycle. $D$ denotes the dimension of the array that is in the cycle.
2. Proceed to the next node in the cycle and change $a$ and $b$ as follows:
    - If the edge is a normalized conformance preference that relates different array occurrences and is attributed with the information `ALIGN (.,.,1,0) WITH (.,.,`$c$`,`$d$`)` then replace $a$ with $a \cdot c$ and $b$ with $b \cdot c + d$
    - Otherwise, leave $a$ and $b$ unchanged.

3.Repeat step 2, as long as $N$ is not reached again.

4.$N$ is reached at dimension $D'$. There is

- an *offset conflict* if $b \neq 0$ and $D = D'$,
- a *stride conflict* if $a \neq 1$ and $D = D'$, and
- a *dimension conflict* if $D \neq D'$.

Otherwise, there is no data arrangement conflict.

The compiler preserves all conflict free data arrangements and all conflicts, i.e. all possible data arrangements that require to break at least one data arrangement preference. The way this information is used is presented in section 11.5.

*Process Arrangement Conflicts*

In the following example the process arrangement graph (see Figure 11.2) is cyclic:

```
FORALL i : [1..N] IN SYNC
  A[ i ] := t[i];
  A[i+1] := A[i+1] + 1
END
```

Only process and LMO preferences are taken into account. In the example there are two process preferences (i, A[i]) and (i, A[i+1]). Additionally, there is an LMO preference between A[i] and A[i+1].

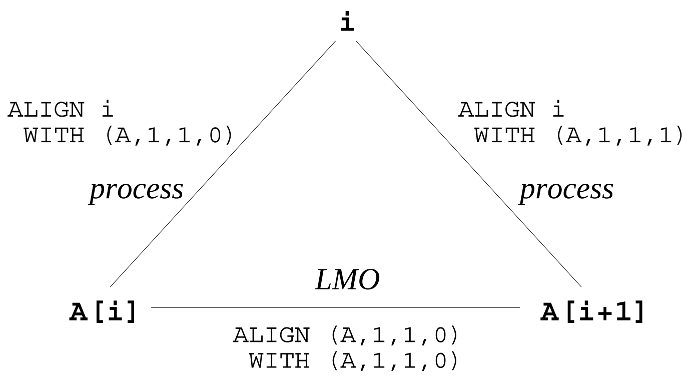

**Figure 11.2** Process Arrangement Graph

Although there are no data arrangement conflicts, there are process arrangement conflicts: process preferences to A[i] and A[i+1] contradict.

To determine all possible process arrangements, the process arrangement graph is processed as follows:

1. The process alignment graph is divided into subgraphs that are processed in turn. A subgraph consists of a `FORALL`-variable, and all LMOs that are related to that `FORALL`-variable, either directly via process preference edges or indirectly via a chain of LMO preferences.
2. For each cycle in each subgraph that contains the `FORALL`-variable exactly once, execute steps 3–6:
3. Start with $a$:=1, $b$:= 0, and $flag$:=`FALSE` at the node of the `FORALL`-variable.
4. Proceed to the next node in the cycle. The edge is attributed with the normalized structure `ALIGN . WITH (.,.,`$c$`,`$d$`)`.
   - If the edge is an LMO preference or $flag$ =`FALSE` replace $a$ with $a \cdot c$ and $b$ with $b \cdot c + d$. In case of a process preference, set $flag$:=`TRUE`.
   - The last edge in the cycle is a process preference with $flag$ =`TRUE`. Replace $a$ with $a/c$ and $b$ with $(b - d)/c$.
5. Repeat step 4, as long as the node of the `FORALL`-variable is not reached again.
6. There is
   - an *offset conflict* if $b \neq 0$ and
   - a *stride conflict* if $a \neq 1$.
7. Consider all edges of a subgraph. There is a *dimension conflict* if among those there is pair of process preference edges with differing dimensions in a single array.

The compiler keeps all conflict free process arrangements and all conflicts, i.e. all possible process arrangements that require to break at least one LMO preference. The way this information is used is presented in the following section.

## 11.5 Cost Considerations

In the previous section the processing of the data arrangement graph has resulted in a collection of several possible data arrangements for the whole program. For each `FORALL` statement in this program the compiler has derived a collection of possible process distributions.

Finding an optimal process distribution with a brute force algorithm would involve an exponential search space. A `FORALL` with $n$ statements and $p$ possible distributions requires the cost estimation for $p^n$ different combinations.

Unfortunately, the combination of two optimal process distributions for the statement sequences $1 \ldots \lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1 \ldots n$ does not necessarily result in a global optimum, since redistribution of processes imposes additional costs. With the assumption that the process redistribution cost, i.e., the cost of splitting up a `FORALL` into several `FORALL`s, are small compared to the communication cost due to data access, the probable loss of optimality can be tolerated. Therefore, a dynamic programming approach with a time complexity of $O(n \log n)$ is feasible:

1. For each data arrangement perform steps 2–5.
2. For each process arrangement in each `FORALL` statement perform steps 3–4.
3. Derive the optimal process distribution and thus the appropriate splitting of the `FORALL` by dynamic programming.
4. Select the best alternative for the given data arrangement.
5. Sum up the cost of all `FORALL` statements in the program for the given data arrangement.
6. Select the data arrangement that results in the global optimum.

The above is a high-level description of our technique. In reality the situation is more complex: Loops and nested `FORALL`s require multidimensional cost vectors instead of simple communication costs. The runtime of `IF`- and `CASE`-statements can be improved if different data arrangements are chosen for different branches. To exploit this possibility, the algorithm considers dynamic array redistribution that ensures the unification of different data arrangements after the branching statements. Details can be found in [15].

## 11.6 Example

Consider the following code fragment:

```
FORALL i : [1..N] IN SYNC
  A[i+1] := T[i] + C[i];
  A[i]   := A[i+1] + T[i];
  A[i+1] := T[i+1] + D[i];
  A[i]   := T[i] + A[i+1];
END
```

The data alignment analysis (see section 11.4.2) returns two possible patterns:

```
ALIGN (C,1,1,0) WITH (T,1,1,0)          ALIGN (C,1,1,0) WITH (T,1,1,0)
ALIGN (C,1,1,0) WITH (A,1,1,1)    or    ALIGN (C,1,1,0) WITH (A,1,1,0)
ALIGN (C,1,1,0) WITH (D,1,1,-1)         ALIGN (C,1,1,0) WITH (D,1,1,-1)
```

The process alignment analysis (see section 11.4.2) returns two possible patterns:

```
ALIGN    i    WITH (A,1,1,1)    or    ALIGN    i    WITH (A,1,1,0)
```

Although the compiler considers both possible data arrangements, in this example we will only consider the second arrangement. Therefore, we will only present steps 2–5 of the search algorithm from section 11.5.

| line | (A,1,1,1) | (A,1,1,0) | align | cost |
|---|---|---|---|---|
| 1 | 2g | 1s | 0 | 1s |
| 2 | 1g+1s | 1g | 0 | 1g |
| 3 | 1g | 2g+1s | 1 | 1g |
| 4 | 1g+1s | 1g | 0 | 1g |
| 1-2 | 3g+1s |  | 0-0 | 1g+1s |
| 3-4 | 2g+1s | 3g+1s | 1-0 | 2g+1f |
| 1-4 | 5g+2s | 6g+2s+1f | 0-0-1-0 | 3g+1s+2f |

In the above table $s$,$g$, and $f$ denote the cost of a send operation, a get operation, and the cost of splitting up a `FORALL` [1]. In the first step, the costs of executing individual lines are computed for all process distributions. Merging lines 1 and 2 is obvious, since in both lines `(A,1,1,0)` is superior. This is shown by 0—0 in the table. For merging lines 3 and 4 there are three possibilities. All must be considered, since f is not zero. (1) use `(A,1,1,1)` for both lines at a cost of $2g + 1s$, (2) use `(A,1,1,0)` for both lines at a cost of $3g + 1s$, or (3) redistribute $1 \rightarrow 0$ at a cost of $2g + 1f$, which is the cheapest. When considering the whole `FORALL` statement in the last step, there are again three options: (1) select data distribution `(A,1,1,1)` for all lines at a cost of $3g + 2s$, (2) select `(A,1,1,0)` for the first two lines and `(A,1,1,1)` for the last two lines at a cost of $6g + 2s + 1f$, or (3) redistribute again resulting in a cost of $3g + 1s + 2f$. Given the values for $g$,$s$, and $f$, the best process distribution will split up the given `FORALL` twice, after the second and after the third line.

Assuming the second data arrangement, the code fragment is transformed as follows.

```
FORALL i : [1..N] ALIGNED WITH A[i] IN SYNC
  A[i+1] := T[i] + C[i];
  A[i]   := A[i+1] + T[i];
END;
FORALL i : [1..N] ALIGNED WITH A[i+1] IN SYNC
  A[i+1] := T[i+1] + D[i];
END;
```

[1]In the example we set $s = 128$, $g = 256$, and $f = 20$ time units.

```
    FORALL i : [1..N] ALIGNED WITH A[i] IN SYNC
      A[i]   := T[i] + A[i+1];
    END
```

Note that for sake of clarity the transformations related to data arrangement are left out in this example, i.e. all arrays are still presented in their original declaration with the original subscripts.

## 11.7 Conclusion

In this paper we have presented a technique that enhances locality using a source-to-source transformation. The result of this program transformation is a data and process alignment that results in better performance: first benchmarking yields an improvement of performance by at least 60% on the MasPar MP-1.

We consider this result to be initial evidence that automatic data and process distribution by the compiler is possible and can achieve attractive performance improvements.

## References

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

[2]Barbara M. Chapman, Heinz Herbeck, and Hans P. Zima. Automatic support for data distribution. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 51–58, Portland, Oregon, April 28 – May 1, 1991.

[3]American National Standards Institute, Inc., Washington, D.C. *ANSI, Programming Language Fortran Extended (Fortran 90). ANSI X3.198-1992*, 1992.

[4]Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[5]Manish Gupta and Prithviraj Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 43–50, Portland, Oregon, April 28 – May 1, 1991.

[6]Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[7]Kathleen Knobe and Venkataraman Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90:The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, University of Maryland, October 8–10, 1990.

[8]Charles Koelbel and Piyush Mehrotra. Supporting shared data structures and distributed memory architectures. In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, March 1990.

[9]Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–433, College Park, University of Maryland, October 8–10, 1990.

[10]Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[11]Prentice Hall, Englewood Cliffs, New Jersey. *INMOS Limited: Occam Programming Manual*, 1984.

[12]Michael Philippsen. Automatic data distribution for nearest neighbor networks. In *Frontiers '92:The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 178–185, Mc Lean, Virginia, October 19–21, 1992.

[13]Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[14]MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, September 1990.

[15]Markus U. Mock. Alignment in Modula-2*. Master's thesis, University of Karlsruhe, Department of Informatics, December 1992.

[16]J. Ramanujam and P. Sadayappan. Access based data decomposition for distributed memory machines. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 196–199, Portland, Oregon, April 28 – May 1, 1991.

[17]Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, University of Karlsruhe, Department of Informatics, January 1990.