Reducing Communication by Honoring Multiple Alignments *

David A. Garza-Salazar and Wim Böhm
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
tel: (303) 491-7595
fax: (303) 491-2466
email: *bohm@cs.colostate.edu*

Abstract

Data Decomposition involves the mapping of array elements to processors of a Distributed Memory Machine with the goal to obtain the best possible performance of a program by keeping communication costs low while exploiting parallelism. Data decomposition is typically divided into two subproblems: alignment and partitioning. Alignment deals with the relative allocation of different arrays. Partitioning is concerned with the actual distribution of the array elements among processors. Conflicting alignments may cause communication. This paper presents a technique for reducing communication by honoring multiple alignments and applies this approach in a distributed memory implementation of the strict functional language Sisal. Multiple alignment leads to recomputation and replication of array elements, which is safe in a functional, and hence side effect free, setting. We present performance improvements of up to 80% for one dimensional arrays, and up to 50% for two dimensional arrays, compared to single alignment implementations on a cluster of workstations.

1 Introduction

Data decomposition involves the mapping of array elements to processors of a Distributed Memory Machine. Some research groups have concentrated on language extensions to guide the data decomposition [4, 9, 11]. A more transparent approach is sought by researchers that are trying to derive the data decomposition automatically during a specialized analysis phase of the compiler based on information extracted from the source code [5, 6, 13, 23, 27]. Regardless of the method used to derive the decomposition, the problem is typically approached in two phases: *alignment and partitioning.* The goal of alignment is to reduce communication due to *inter-array* references, references among different arrays. Alignment involves the relative allocation of arrays by mapping elements of different arrays to a common virtual array known as a *template,* which is a Cartesian grid of sufficiently high dimensionality and size.

* This work is supported in part by LLNL Grant B282313

When a computation requires elements mapped to the same template point, no communication will occur. Alignment is machine independent. For simplicity we will assume that one of the arrays of the program represents the template. We will identify this array as the *target array.* We will call the array that is being aligned with the target array *the alignee.*

Data partitioning is concerned with the distribution of the target array over processors. The issue is how to distribute the data while minimizing communication, maximizing parallelism, and balancing workload. Data partitioning is a more machine dependent problem since it is related to the machine architecture [1].

Since alignment is a machine independent problem and partitioning a more machine dependent problem, most of the data decomposition approaches perform alignment first followed by partitioning.

There are situations where *conflicting alignments* exist, e.g., in `A[i]= B[i] + B[i+5]` the reference `B[i]` requires `B[i]` and `A[i]` to be aligned, whereas `B[i+5]` requires `B[i+5]` and `A[i]` to be aligned. Most approaches to cope with conflicting alignments select one of the multiple alignments to be honored while leaving the remaining references unhonored with the consequence that, depending on the partitioning scheme, unaligned references can lead to communication. The goal of this paper is to present an alternative approach that reduces communication by multiple alignment.

Functional languages provide and implicitly parallel, deterministic, and machine-independent programming paradigm. We are interested in data decomposition techniques that can be applied to monolithic strict single assignment arrays found in languages such as SISAL [24]. *Monolithic arrays* are created in one syntactic construct. The operation `A = mk_array(n,f)` creates an array A of size n where each of it's elements i, is defined as $A[i] = f(i)$ where f is any function defined in the program [16, 31]. In Sisal this is written as:

```
for i in 1,n
   returns array of f(i)
end for
```

Strict arrays must be completely defined before any of the array elements can be used. On the other hand, languages that use *non-strict* arrays allow that not all of the array needs to be defined before elements can be used. Monolithic, strict, single assignment arrays have two important characteristics:

- *No Intra-array references.* Elements of a strict array cannot be defined in terms of other elements of the same array. Since the array definition is free of intra-array references, data decomposition is concerned with reducing communication due to inter-array references. Therefore unaligned references become the only source of communication.

- *Safe Replication via Recomputation.* Since every array element is defined just once during the lifetime of the program and the functional language guarantees side-effect freeness, it is safe to compute the same array element in more than one processor. This is what we call *recomputation.* Because of the single assignment rule, coherence among distributed copies of array elements is guaranteed. Also, it is very easy to identify the process in the program that defines a particular element of a monolithic array. *Replication* of data, having an array element available in more than one processor, can thus be realized by either communication or recomputation. This allows us to trade off communication and recomputation.

The rest of the paper is structured as follows. Section two discusses the cluster of workstations we use in our experiments, and the timing behavior of passing messages of certain sizes between processors in such a cluster. Section three introduces multiple alignment in the static case and in loops. Examples of both a one-dimensional problem and a two-dimensional problem are treated in detail. Section four discusses related work, and section five provides conclusions and future work.

2 Message Passing in a Cluster of Workstations

We will measure the behavior of the C plus message passing target code of the Distributed Memory Sisal compiler that we are implementing. We will compare programs using single alignment to those employing multiple alignment. Multiple array elements to be sent from one processor to another are always grouped together in one message. Our experiments were done using up to 16 HP/9000 series 400 workstations interconnected via Ethernet. These workstations were not isolated during our tests. However, we ran the experiments during low traffic hours. The running times are all reported in seconds and represent the average CPU time of 5 runs.

In any Distributed Memory Machine, there is a message startup time and a message transfer time. The message transfer time depends on the length of the message being transferred. The startup time is several orders of magnitude higher than the transfer time for a small message. It is this phenomenon that we are trying to exploit when decreasing the number of messages and increasing their size.

The message passing library used is MPI [10]. We are experimenting with two implementations of MPI: MPICH [12] and MPI-LAM [3]. Fig. 1 shows the timing behavior of the two MPI implementations on the cluster of HP400 workstations. It gives the time it takes to exchange 100 messages of a certain message size, measured in floats, between two workstations. We can see that the message startup time for MPICH is about four milliseconds and for MPI-LAM about twelve milliseconds. To exchange a message of 120 floats takes about half a millisecond in transfer time. In general, MPICH is much faster than MPI-LAM. MPICH behaves non-linearly: messages of size less than 256 floats are exchanged faster than larger messages. This is because MPICH employs different algorithms for small and large

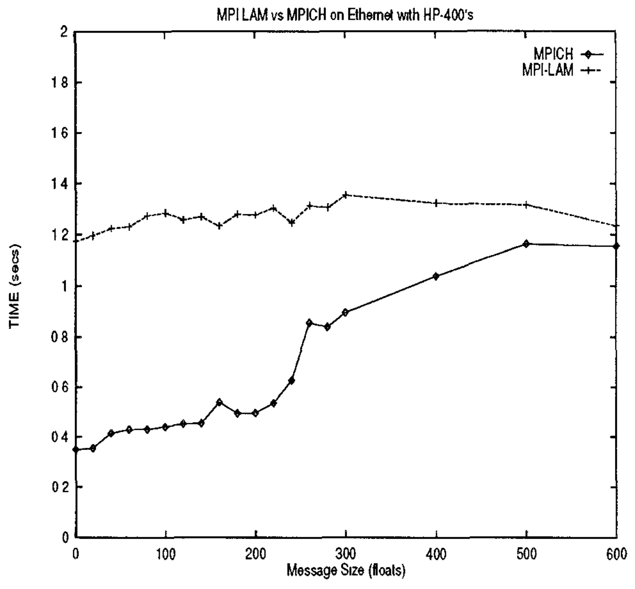

Figure 1: Message Exchange Times for 100 Messages in MPICH and MPI-LAM

messages [26]. When using MPICH it is therefore not profitable to exchange messages that are larger, but not much larger, than 256 floats. The consequences of this will become clear when we analyse the results of our experiments.

3 Honoring Multiple Conflicting Alignments

In the sequel, when we talk about arrays we will be referring to monolithic, strict, functional arrays. Decomposition concentrates on reducing costs due to unaligned references. We will investigate the effect of honoring multiple conflicting alignments by replication via recomputation. We call this approach *Multiple Alignment.* Two conditions determine the profitability of multiple alignment:

- *Recomputation vs Communication costs.* The cost of recomputation of the replicated data elements in multiple alignment implementations should be less that the cost of communicating the same data elements in the single alignment implementation. This is a machine and program dependent issue.
- *Memory availability.* There should be enough memory available on each processor to allow for the replication of data elements. The memory requirements of the multiple alignment implementation is program dependent. Often only a small fraction of the memory is needed for replication.

Parallelizing compilers for Distributed Memory Machines typically have a code generation phase that uses data decomposition information to generate the node program for the processors of a distributed memory machine [15]. We assume that the code is generated for SPMD (Single Program Multiple Data) execution model[17] and the code follows the owner computes rule for the target array. However, the rule is relaxed since several processors can be the owners of the same data element. Code generation to support multiple alignment will be different from single alignment: arrays are divided over processors in overlapping chunks or parts, and loop bounds in various processors overlap.

Our methods for obtaining the local index sets and generation of local computations are based on techniques described in [14]. The local index set is defined as the set of indices of all those elements of an array that reside in processor p. Data distribution defines a distribution function that maps array elements to processors. The distribution function is used to obtain the local index set. The local computation is obtained by applying the inverse array subscript function to the local index set, giving rise to the local loop bounds. Given the appropriate decomposition information, such as the target array, the references that require multiple alignment and the partitioning scheme of the target array, the following summarizes the code generation strategy.

- Compute the local index set of the target array.
- Compute the local index set of the alignee based on the references that require multiple alignment.
- Generate the local loop bounds from the local index set of the target array.
- Generate the local access patterns into alignee array based on its local index set.

The simplest case of alignment occurs for statically named arrays, i.e. arrays that are not redefined in loops. An example is Livermore Loop #1, which performs the following computation:

$$X_i = Q + (Y_i * (R * Z_{i+10} + T * Z_{i+11}))$$

Although this is a very simple example, reference patterns similar to this occur frequently in the Livermore Loops Benchmark Suite [8]. Single alignment will align X_i with Y_i and with one of the two references of Z, Z_{i+10} or Z_{i+11}. When the arrays are equally partitioned over p processors, either of the two alignments will give rise to one exchange per processor. In this simple case, by multiple alignment all communication can be avoided.

In the more general case, arrays are redefined in loops, and not all communication can be avoided. In a functional program, multiple versions of an array or scalar can be defined in a loop construct, as in the following Sisal loop sketch:

```
for initial
   V := A; count := 0;
repeat
   count := old count + 1;
   V:= f( old V )
until count = k
returns value of V
end for
```

For each loop body a new version of V and *count* is created, and the value of the previous body (or the initial value) can be referred to using the keyword *old*. The function f produces an array. In most cases this is done using a loop construct. In order to apply multiple alignment in a loop as the one above, the compiler unrolls the *outer* loop, thereby creating a number of statically named arrays in the loop body. In the following code we have, for the sake of simplicity, ignored complications dealing with the unrolled loop overrunning the count.

```
for initial
   V := A; count := 0;
repeat
   count := old count + 3;
   V'' := f( old V )
   V' := f( V'' )
   V := f( V' )
until count >= k
returns value of V
end for
```

Given the array reference patterns in f, V'', V' and V can now be multiple aligned. Partitioning the multiple aligned loops will give rise to overlapping parts of the array being allocated to processors and the recomputation in the various processors takes the form of overlapping loop bounds. Communication is only necessary at the top of the unrolled loop bodies.

As an example, Successive Over Relaxation (SOR) repeatedly performs the following operations on an array:

$$A'_i = \begin{cases} A_i & \text{if } i = 1 \text{ or } i = n \\ (A_{i-1} + A_i + A_{i+1})/3.0 & \text{otherwise} \end{cases}$$

The SISAL code for SOR can be found in appendix A.

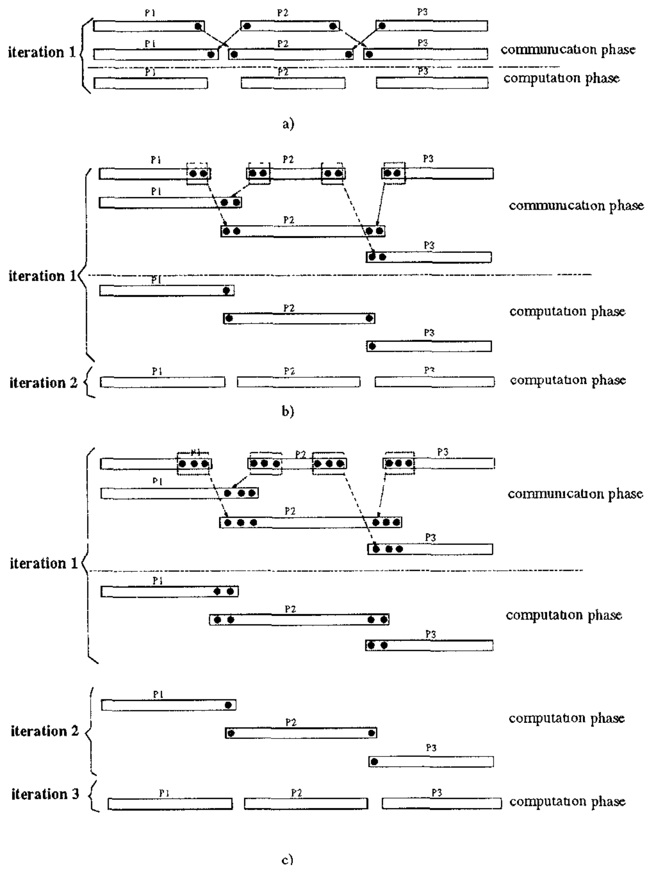

Figure 2: SOR: a) no unrolling, b) unrolling depth of 1, c) unrolling depth of 2.

Fig. 2 shows the effect of various levels of loop unrolling on the computation of SOR on 3 processors. We will analyse the number and size of messages sent and received by the middle processor P_2, as the behavior of this processor is typical for cases of larger numbers of processors. Fig. 2 a) shows the case of no loop unrolling and single alignment. In

PE	*Single Alignment*		*Multiple Alignment*										Imp
			depth = 1		*depth = 5*		*depth = 10*		*depth = 15*		*depth = 20*		
	Time	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Imp*
1	24.18	100	24 18	100	24.18	100	24.18	100	24 18	100	24.18	100	0.00
2	21 20	57	14 63	83	14.40	84	14.33	84	14 30	85	14.23	85	48 98
4	10.96	55	7.46	81	7.32	83	7 28	83	7.26	83	7 25	83	51 59
8	5.88	51	4.16	73	3 87	78	3 80	79	3.84	79	3 75	81	56 80
16	3 84	39	2.37	64	2 24	67	2 06	73	2.12	71	2 06	73	86 40

Table 1: Comparative performance results for SOR with N=32768 and K=100.

each iteration P_2 exchanges four messages (two sends and two receives) of the size of one data element per message. Fig. 2 b) shows the case of a loop unrolling of one combined with multiple alignment. Communication occurs at the top of the unrolled loop, i.e. every other iteration. Four messages of size two each are exchanged in P_2. Therefore, the total number of messages exchanged in the whole computation is reduced by 50% when compared to the single alignment implementation. However, the messages exchanged in the multiple alignment implementation are longer. The recomputation performed among processors, shown as the overlapping regions in Fig. 2, causes the first unrolled iteration to run two steps more: starting one step earlier and ending one step later. Fig. 2 c) shows the implementation of multiple alignment with unrolling factor two. Communication now occurs once per three iterations. The number of messages exchanged is reduced to 33% compared to the single alignment case. The size of each message is tripled and there is more recomputation: the first iteration runs four steps more, the second iteration runs two steps more. When further increasing the depth of unrolling, the size of the messages and the recomputation requirements increase linearly. With an unrolling factor of k, the amount of recomputation is $2 * (\sum_{i=1}^{k} i)$, which is quadratic in k.

Table. 1 shows the performance results that were obtained with the single alignment and multiple alignment implementation of SOR with depths of replication 1, 5, 10, 15, and 20. All times are in seconds. The timing for one processor is that of a message passing free sequential code. The efficiencies (Eff) are all relative to this. The best improvement: (*Single Alignment Time / Multiple Alignment Time* - 1) percentage is given in column Imp. Fig. 3 summarizes the efficiencies of the various versions of SOR. A depth of replication 0 in the graph of Fig. 3 represents single alignment. We can draw two conclusions from the results. Firstly, it is clear that multiple alignment pays off handsomely for this one dimensional problem. From efficiencies of 39% to 57% for the single alignment code, the multiple alignment code jumps to efficiencies between 73% and 85%. Secondly, most of the gain is achieved at unrolling factor depth one. This is because for higher unrolling factors, the quadratic amount of recomputation becomes costly.

3.1 Multiple Alignment in Higher Dimensional Arrays

For higher dimensional arrays the same multiple alignment techniques apply. Again, by unrolling the outer loop, several versions of an array are created. In many cases the dependence between one array and the next takes the form of a stencil, i.e. the dependence distance vectors are constant. Row partitioning versus block partitioning of e.g. a two dimensional array gives rise to interesting trade off problems, as the amount of communication in stencil computations is a linear function of the size of the boundary between partitions. For example, an $n \times n$ array block partitioned over $p \times p$ processors gives rise to a boundary of size $4n/p$, whereas row partitioning would give rise to a boundary of size n.

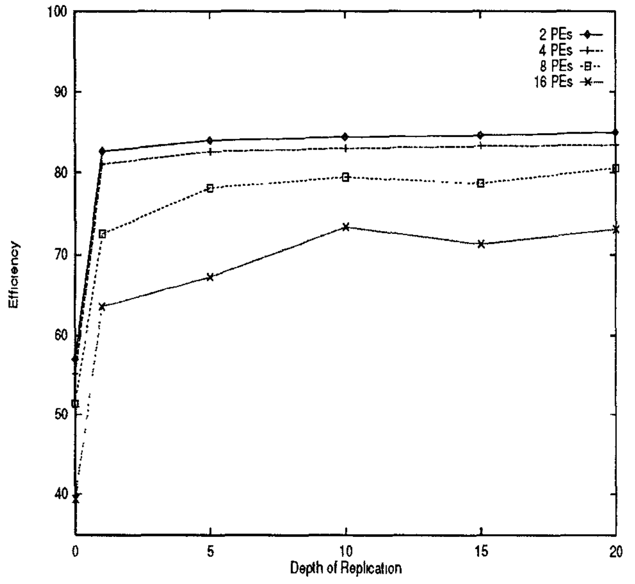

Figure 3: Performance results of SOR for different Depths of Replication

Laplace performs repeated smoothing in a five-point stencil over a two-dimensional array:

$$A'_{i,j} = \begin{cases} A_{i,j} & \text{if } i = 1 \text{ or } i = n \text{ or } j = 1 \text{ or } j = n\text{, otherwise} \\ A_{i,j}/2\ 0 + (A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})/8\ 0 \end{cases}$$

The SISAL code for Laplace can be found in appendix B. Fig. 4 shows the inter-array dependences for the computation of Laplace. In the case of row partioning and single alignment each iteration requires a processor P_M, which deals with a set of rows in the middle of the array, to exchange four rows, two to send and two to receive, in four messages. In multiple alignment with a loop unrolling factor of k, $k+1$ iterations in P_M require four messages, two of size $(k+1) * n$ to send and two of size $(k+1) * n$ to receive. The amount of recomputation required equals $(\sum_{i=1}^{k} i) * n$, which is $O(k^2 * n)$.

Multiple alignment combined with block partitioning introduces an increase in the number of processors a processor needs to communicate with. Figure 5 shows the effect of loop unrolling and multiple alignment on the communication requirements when the array is block partitioned. In single alignment a processor P_M, dealing with a block in the middle of the matrix, needs to communicate with four neighbours and each message is of size n/p. In multiple alignment with

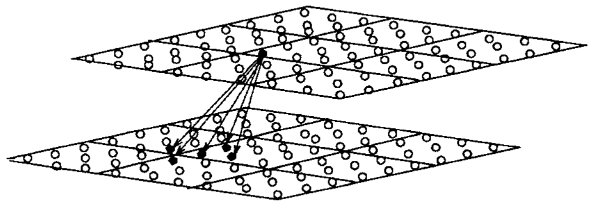

Figure 4: Inter-Array dependences for Laplace.

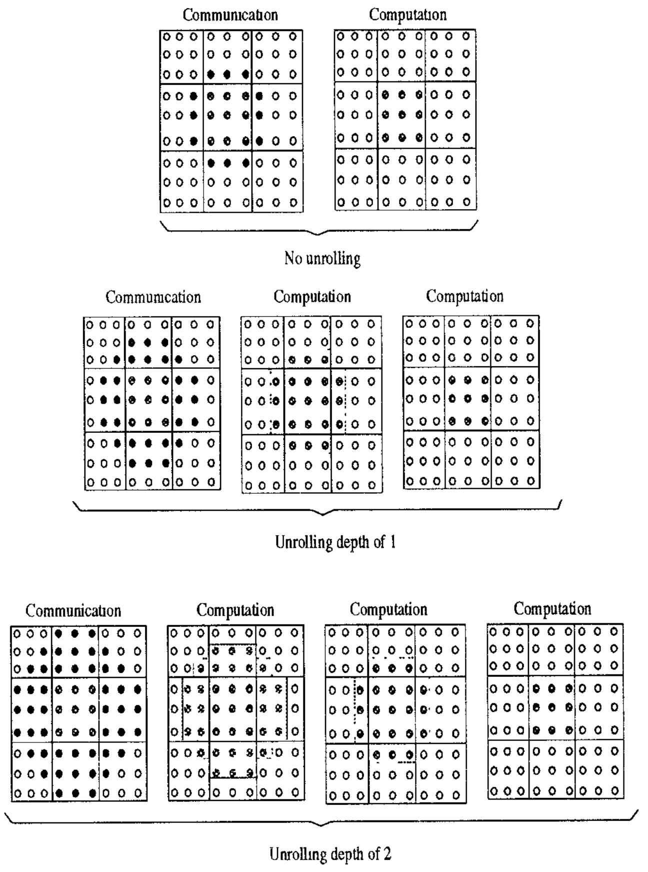

Figure 5: Multiple Alignment in Laplace.

a loop unrolling factor k, P_M needs to communicate with eight processors. For each $k+1$ iterations P_M requires eight messages of size $(k+1)*(n/p)$ plus eight messages of size $\sum_{i=1}^{k} i$. Thus, block partitioning combined with multiple alignment requires four times more messages than row partitioning combined with multiple alignment. The amount of recomputation is

$$4*(\sum_{i=1}^{k} i*(n/p)+\sum_{j=1}^{k}\sum_{i=1}^{j} i)$$

which is again $O(k^2*n)$. Tables 2, 3, 4 and 5 at he end of the paper give the timing results for Laplace comparing block and row partitioning, single and multiple alignment, and MPICH and MPI-LAM. All efficiencies are relative to a non message passing sequential code run on one processor. Figures 6, 7, 8, 9, 10 and 11, also at the end of the paper, summarize the efficiencies. The results lead to the following observations.

Communication Speed The faster message passing in MPICH versus MPI-LAM has a significant impact on the efficiency of the programs run. This is the case for single as well as multiple alignment. In MPICH we have a slow-down in message passing for messages larger than 256 floats. This has the strongest effect on row partitioning of problem size 256*256. Here multiple aligmment does not pay off as sending one message of size 256 takes more than sending two messages of size 128.

MPI-LAM exemplifies the effects of low communication speed. Here multiple alignment is much more effective. We observe improvements of over 50% for both block and row distribution. Thus, multiple alignment is more relevant in cases of a high compute/communicate ratio.

Problem size $64*64$ on 16 PEs is a case of too much communication overhead per computation sweep. However, especially there, multiple alignment improves performance.

Row versus Block Partitioning For all machine configurations and problem sizes tested, row partitioning outperformed block partitioning. This is due to the smaller number of messages required for row partitioning. As data and processor sizes increase, the gap between the two decreases.

Recomputation In both row and block partitioning, recomputation grows quadratically with the unrolling factor. In most cases, therefore, only small unrolling factors pay off. The biggest improvement for row partitioning (18.47 %) is achieved by an unrolling factor of one on a problem size of $64*64$ using MPICH. In this case we have a low number of messages of small enough size, and a low amount of recomputation.

For block partitioning, improvements of about 18.5% occur for unrolling factors of three and four on problem sizes $64*64$ and $128*128$ for 16 processors using MPICH. Here the message sizes are all under 256, which allows for efficient message passing. We expect that the same improvements can be achieved for larger problem sizes and larger number of processors, as this would give rise to the same local matrix size, message size and amount of recompution.

4 Related Work

Data Decomposition techniques have been studied by several research groups. In the context of functional languages, Li and Chen [22, 23] propose automatic array alignment and partitioning techniques for the functional language Crystal [30]. Rogers and Pingali [29] present some work where the user specifies data decomposition for the functional language Id Noveau [25]. In [27] M. O'Boyle describes program and data transformations for executing a restricted version of SISAL on distributed memory machines. The techniques proposed are intended to solve problems of load imbalance, data alignment, data distribution, and partitioning of loop iterations.

In the context of imperative languages, Knobe Lukas, Dally and Steel [18, 19], propose an approach for automatic alignment of arrays, They consider the fact that an array can be required to be aligned in different ways in different parts of the program. When conflicting alignments are present some of the alignments have to stay unhonored causing communication.

In Fortran D [15, 11], the user selects the data alignment and distribution by means of `DECOMPOSITION`, `ALIGN` and `DISTRIBUTE` statements or directives. The `DECOMPOSITION` statement specifies a virtual template. The ALIGN directive maps arrays onto this virtual template. Although the language specification [11] states that replication can be done via the ALIGN directive, we have not seen any publications on the potential benefits of this, or how it is used in the presence of conflicting alignments.

Recent work by Kremer [21, 20], considers a tool to derive automatic data alignment and distribution. The goal is to translate Fortran 77 into Fortran D with appropriate data decomposition statements to execute on a distributed memory machine. This work considers the detection of when is profitable to re-distribute arrays at run time by inserting redistribution statements in the code. Our approach is a static approach that saves the cost of redistributing array values by recomputing them locally.

Chatterjee, Gilbert, Schreiber and Teng [6] propose an approach to automatically determine alignment of variables and intermediate values. The language model used is Fortran 90. The alignment problem is stated as a minimization problem trying to find an alignment function that minimizes the communication cost. This work considers just the selection of one of the possible alignments. In [7] Chatterjee, Gilbert and Schreiber present how to identify alignments that vary in loops and arrays that require replicated alignments. Several differences with our approach are the following: First they just consider replication for situations where the dimensionality of the target array is higher than the dimensionality of the array to be aligned. Replication occurs along the extra dimensions of the target array. Another major difference is that replication is done after the array element has been computed in one processor. This again makes our approach more static since the replication is encoded in the program generated by means of the recomputation of data. No broadcast of data is required in order to achieve replication.

In [2], Chapman, Mehrotra, Moritsch and Zima propose dynamic data distribution in Vienna Fortran [4], where the DISTRIBUTE statement is used to dynamically distribute and realign arrays. The language specification for HPF [9] includes realignment and redistribution directives. This is different to what we propose since we want to statically honor several alignments via replication and recomputation.

Ramanujam and Sadayappan [28] describe a method for finding a partitioning of arrays where there is no communication among processors even in the presence of multiple alignments. The approach works for some type of access pattern where a communication free partition can be found. Our approach can be seen as an extension of this work where we achieve communication free computation for those accesses where communication-free partitioning does not exist. Our approach works for conflicting alignments where as theirs just consider communication free computation for multiple non-conflicting alignments. They propose a cost function to select one alignment in the presence of conflicting alignments.

5 Conclusions and Future Work

In this paper we have shown how communication can be reduced and overall performance increased by allowing multiple alignment of conflicting array references. The examples presented here suggest that recomputation of data can be considered as a viable option to improve the performance of regular problems implemented on distributed memory machines. We have shown improvements of up to 85% for one dimensional arrays, and up to 50% for two dimensional arrays.

Key issues that make multiple alignment a profitable approach are recomputation versus communication costs, and depth of replication.

Multiple alignment is most effective when the communication speed is relatively low. Since the current trend in hardware technology is that processor speed increases faster than network speed, we hypothesize that multiple alignment will become more effective in improving the performance of clusters of workstations with more powerful processors, used as parallel machines. Also, with faster processors the extra cost of recomputation will be less important.

The partitioning scheme has an impact on the message size and on the amount of recomputation required in multiple alignment based on loop unrolling. It turns out that most of the gains are made with low depth of unrolling, as the message sizes and recomputation requirements are kept low.

We are working on data dependence analysis and integer linear programming techniques to implement multiple alignment. Data dependence and ILP are also used to determine the elements that must be replicated on each processor. Input dependence analysis can help to find the local index set of the recomputed arrays. Data dependence can also contribute to give an estimate of the memory requirements of the multiple alignment version.

Training sets [1] can be used to determinate the profitability of multiple alignment. Training sets provide information about the execution times of several arithmetic and communication operations typically performed by a program. With this information it should become possible to statically estimate the relative effect of depth of unrolling in multiple alignment, and of certain partitioning methods.

We expect to derive formal algorithms for this and integrate it into a prototype SISAL compiler that we are currently extending to support generation of code for Distributed Memory Machine.

Due to the high network latencies, scientific applications show poor performance on clusters of workstations. Multiple alignment is a simple optimization that can improve performance in these environments.

		Single		*Multiple Alignment*								
		Alignment		*depth = 1*		*depth = 2*		*depth = 3*		*depth = 4*		
Size	*PE*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Imp*
64x64	1	6 33	100	6 33	100	6 33	100	6 33	100	6 33	100	0.00
	4	3 11	51	3 59	44	3.20	50 0	3 43	46	3 34	47	-2 81
	16	2.81	14	2 70	15	2 72	15	2 39	17	2 37	17	18.57
128x128	1	25 10	100	25.10	100	25.10	100	25 10	100	25 10	100	0 00
	4	9.27	68	9.80	64	9 58	65	9 68	65	9 47	66	-2.11
	16	4.74	33	4.48	35	4.70	33	4 11	38	4 22	37	15.33
256x256	1	99.95	100	99 95	100	99 95	100	99.95	100	99 95	100	0 00
	4	30 35	82	32 64	77	33.30	75	33 07	76	33 12	75	-7 02
	16	12.56	50	13 17	47	12 70	49	13 04	48	12 69	49	-1 02

Table 2: Performance results for Laplace, Block Distribution, K=100 (MPICH)

		Single		*Multiple Alignment*								
		Alignment		*depth = 1*		*depth = 2*		*depth = 3*		*depth = 4*		
Size	*PE*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Imp*
64x64	1	6 33	100	6 33	100	6 33	100	6 33	100	6 33	100	0.00
	4	2 63	60	2 22	71	2.25	70	2 29	69	2 34	68	18 47
	16	1 05	38	1 04	38	0 98	40	0 99	40	1 07	37	7 14
128x128	1	25 10	100	25 10	100	25 10	100	25 10	100	25 10	100	0 00
	4	8 12	77	7.84	80	7 84	80	7 94	79	7 98	79	3 57
	16	2 91	54	2 91	54	3 10	51	3 13	50	2 99	52	0 00
256x256	1	99 95	100	99 95	100	99 95	100	99 95	100	99.95	100	0 00
	4	29 62	84	30 28	83	30 48	82	30 77	81	31 04	81	-2 81
	16	8 70	72	9 06	69	9 28	67	9 44	66	9 63	65	-3 97

Table 3: Performance results for Laplace K=100, Row Distribution, (MPICH)

		Single		*Multiple Alignment*								
		Alignment		*depth = 1*		*depth = 2*		*depth = 3*		*depth = 4*		
Size	*PE*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Imp*
64x64	1	6 33	100	6 33	100	6.33	100	6 33	100	6 33	100	0 00
	4	6.26	25	5 95	27	4 92	32	4 82	33	4 60	34	34 91
	16	5 69	7	5 54	7	4 61	9	4.13	10	4.01	10	52 55
128x128	1	25 10	100	25.10	100	25 10	100	25 10	100	25 10	100	0 00
	4	13 18	48	13.14	48	12.33	51	11 89	53	12.17	52	10 85
	16	9 76	16	9.22	17	8 00	20	8 78	18	7 95	20	25.13
256x256	1	99 95	100	99 95	100	99 95	100	99 95	100	99 95	100	0 00
	4	36 74	68	39 15	64	38 39	65	37.89	66	38 01	66	-3 04
	16	19 44	32	19.29	32	19 78	32	18 83	33	18 29	34	6 29

Table 4: Performance results for Laplace K=100, Block Distribution, (MPI-LAM)

		Single		*Multiple Alignment*								
		Alignment		*depth = 1*		*depth = 2*		*depth = 3*		*depth = 4*		
Size	*PE*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Time*	*Eff*	*Imp*
64x64	1	6 33	100	6 33	100	6 33	100	6 33	100	6 33	100	0 00
	4	4 03	39	3 38	47	3 15	50	3 12	51	3 04	52	32 57
	16	2 53	16	2.01	20	1.71	23	1 72	23	1.68	24	50 59
128x128	1	25 10	100	25 10	100	25.10	100	25 10	100	25 10	100	0 00
	4	9 79	64	9 43	67	9 15	69	9 12	69	9 39	67	7 35
	16	4 4	36	3 83	41	3.86	41	3 82	41	3 84	41	15 18
256x256	1	99 95	100	99.95	100	99 95	100	99 95	100	99 95	100	0 00
	4	32 40	77	32 94	76	32 86	76	33 20	75	33 48	75	-1 40
	16	11.37	55	10 72	58	10 86	58	10 80	58	11 46	54	6 06

Table 5: Performance results for Laplace K=100, Row Distribution, (MPI-LAM)

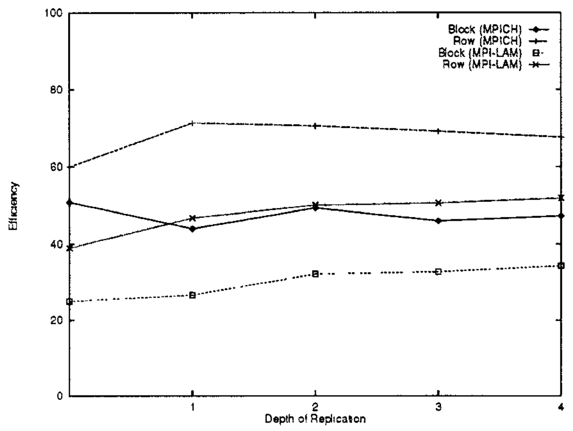

Figure 6: Laplace 64x64, 4 PEs

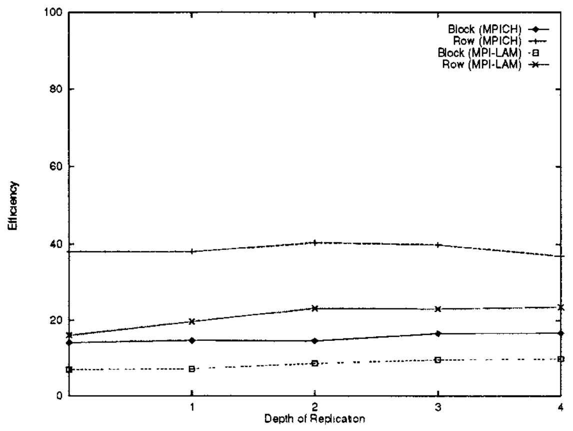

Figure 7: Laplace 64x64, 16 PEs

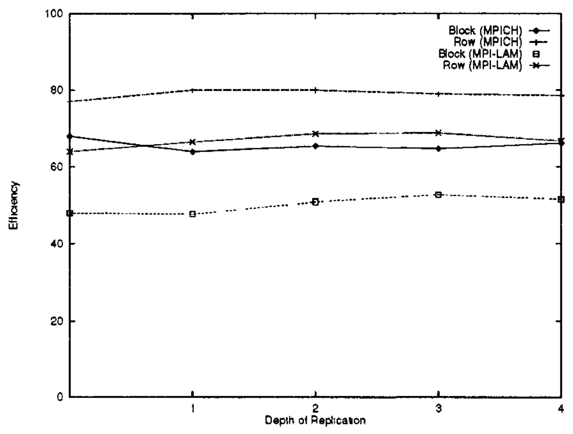

Figure 8: Laplace 128x128, 4 PEs

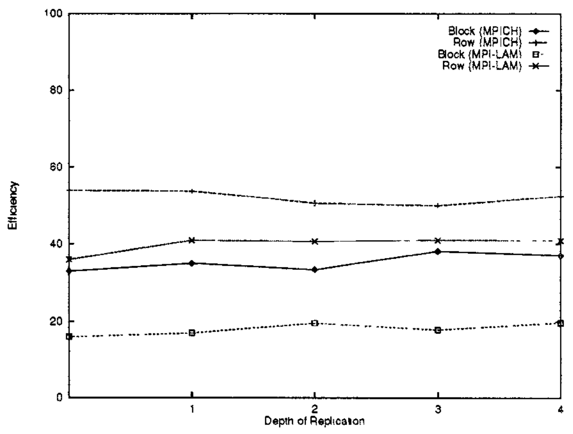

Figure 9: Laplace 128x128, 16 PEs

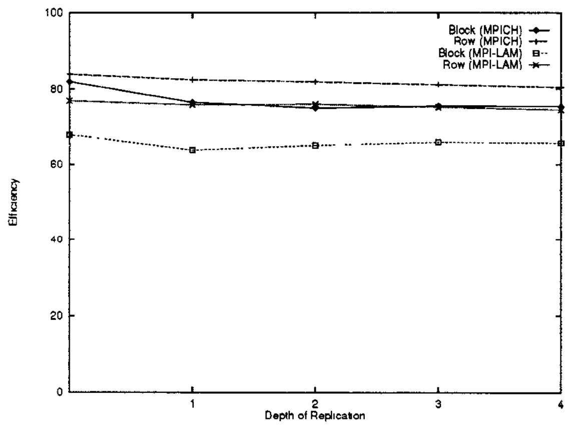

Figure 10: Laplace 256x256, 4 PEs

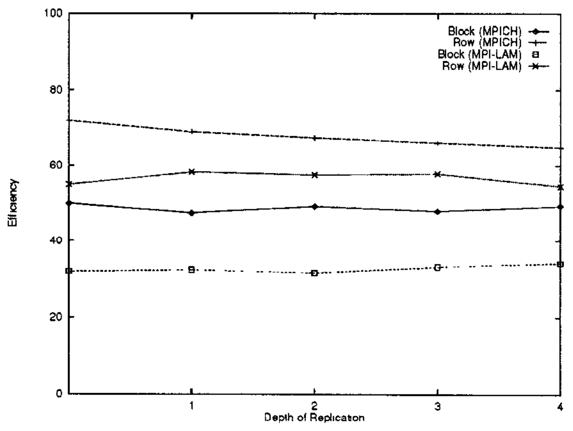

Figure 11: Laplace 256x256, 16 PEs

A Sisal code for Successive Over Relaxation

```
type OneD = array[real]

function sor(n,k:integer;A:OneD returns OneD)
for initial
  V:=A; loop:=0;
repeat
  loop:= old loop + 1;
  V:=array[1:1.0]
  ||
  for i in 2, n-1
   el:=(old V[i-1] + old V[i] + old V[i+1]) / 3.0
  returns array of el
  end for
  ||
  array[1:real(n)]
until loop=k
returns value of V
end for
end function % sor
```

B Sisal code for Laplace

```
type Vector = array[real];
type Matrix = array[Vector]

function lapl( Init_M: matrix; N,KMax: integer
               returns matrix)
for initial
 K:=1;
 M:=Init_M
 repeat
   K := OLD K +1;
   M := for I in 1,N cross J in 1,N
         nM := if I=1|I=N|J=1|J=N  then old M[I,J]
               else old M[I,J] / 2.0 +
                   (old M[I-1,J] + old M[I+1,J] +
                    old M[I,J-1] + old M[I,J+1])/8.0
               end if
        returns array of nM
        end for
 until K = KMax
returns value of M
end for
end function % laplace
```

References

[1] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, 1991.

[2] Hans Moritsch Barbara Chapman, Piyush Mehrotra and Hans Zima. Dynamic data distributions in Vienna Fortran. In *Supercomputing '93*, pages 284–293, Portalnd, OR, 1993.

[3] G. Burns, R. Daoud, and J. Vaigl. *LAM: An Open Cluster Environment for MPI.* Ohio Supercomputer Center, 1994.

[4] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, fall 1992.

[5] Barbara Chapman, Piyush Mehrotra, and Hans P. Zima. High Performance Fortran without templates: An alternative model for distribution and alignment. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 92–101, San Diego, CA, 1993. ACM press.

[6] Siddhartha Chattergee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, 1993.

[7] Siddhartha Chatterjee, John R. Gilbert, and Robert Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Supercomputing '93*, pages 420–429, Portland, OR, 1993.

[8] J. T. Feo. The Livermore Loops in Sisal. Technical Report UCID-21159, Computing Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550, August 1987.

[9] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, 1993.

[10] M.P.I. Forum. *MPI: A Message-Passing Interface Standard*, 1994.

[11] G.C. Fox, S. Hiranandani, Ken Kennedy, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251-1892, 1990.

[12] B. Gropp, R. Lusk, T. Skjellum, and N. Doss. *Portable MPI Model Implementation.* Argonne National Laboratory, 1994.

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

[14] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report CRPC-TR91132, Center for Research on Parallel Computation, Rice University, Houston, TX, 1991.

[15] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[16] P. Hudak. *Graph Reduction*, chapter Array, Non-Determinism, Side-Effects and Parallelism: A Functional Perspective, pages 312–327. Springer-Verlag, 1986.

[17] Alan H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, 1987.

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

[19] Katheleen Knobe, Joan D. Lukas, and William J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 394–404, 1992.

[20] U. Kremer. Automatic data layout for distributed-memory machines. Technical Report CRPC-TR93-299, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

[21] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Vieweg Verlag, editor, *Proceedings of the First International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Wiesbaden, Germany, 1993. Also available as CRPC-TR93298-S, Center for Research on Parallel Computation, Rice University, February 1993.

[22] J Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computations*, College Park, MD, 1990.

[23] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, 1991.

[24] James McGraw, Stephen Skedsielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. Sisal: Streams and iteration in a single assignment language, language reference manual version 1.2. Report M-146, Lawrence Livermore National Laboratory, Livermore, California, 1985.

[25] R.S. Nikhil. Id (version 90.0) reference manual. Technical Report CSG Memo 284-1, Massachusetts Institute of Technology, Laboratory for Computer Science, 1990.

[26] Natawut Nupairoj and Lionel M. Ni. Performance evaluation of some MPI implementations on workstation clusters. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 98–105, October 1994.

[27] Micheal F.P. O'Boyle. *Program and Data Transformations for Efficient Execution on Distributed Memory Architectures.* PhD thesis, University of Manchester, Department of Computer Science, Oxford Rd. Manchester, U.K., 1992.

[28] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, 1991.

[29] A. Rogers and K. Pingali. Process decomposition through locality of reference. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–80, 1989.

[30] Boleslaw K. Szymanski. *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[31] Philip Wadler. *Graph Reduction*, chapter A New Array Operation, pages 327–335. Lecture Notes in Computer Science 279. Springer-Verlag, 1986.