Automatic Data Layout With Read–Only Replication and Memory Constraints

Ulrich Kremer
Department of Computer Science
Rutgers University

TECHICAL REPORT

LCSR–TR283

December 1996

Automatic Data Layout With Read–Only Replication and Memory Constraints*

Ulrich Kremer†

Department of Computer Science
Rutgers University

Abstract

Besides the algorithm selection, the data layout choice is the key intellectual step in writing an efficient HPF program. Although finding an efficient data layout fully automatically may not be possible in all cases, HPF users will need support during the difficult data layout selection process. In particular, this support is necessary if the user is not familiar with the characteristics of the target HPF compiler and target architecture, or even with HPF itself. In addition to the target compiler and architecture, the quality of a data layout depends on the problem size, the number of processors, and the available memory on each processor. Therefore, tools and techniques for automatic data layout will be crucial if the HPF is to find general acceptance in the scientific community.

The memory requirement characteristics of a data layout are particularly important for applications that are executed on a parallel machine mainly because of the amount of main memory that the machine provides, rather than its computation power. It may not be possible to execute such a memory intensive program on a conventional uniprocessor due to the lack of the necessary memory resources.

This paper discusses a new framework for automatic data layout that considers read–only data replication and minimizes the overall execution time under given memory constraints. The framework can be used to generate data layout specifications with additional read–only array copies. Many appli cations use arrays that are assigned a value and keep this value over large portions of the program. In such read–only regions, multiple read–only copies of the array — each copy with a different data layout — may avoid otherwise necessary communication, resulting in a reduction of the overall execution time. Read–only replication does not come for free, since it increases the memory requirements of the program.

*Parts of this research were conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation (NSF) and New York State, with additional support from the Advanced Research Projects Agency (ARPA), the National Center for Research Resources at the National Institutes of Health (NIH), IBM Corporation, and other members of the center's Corporate Partnership Program.

†e-mail: uli@cs.rutgers.edu; phone: (908) 445-4974; address: Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855

The approach presented in this paper addresses the necessary tradeoff decisions between read–only replication and memory requirements in a new, unified framework that extends our previous framework for automatic data layout with remapping. As in our previous work, the data layout selection problem is formulated as an efficient 0–1 integer programming problem. Preliminary experiments show the performance tradeoffs between the new and old formulations.

1 Introduction

Multiprocessor architectures provide not only computing cycles, but also large amounts of main memory. Using a parallel machine instead of a uniprocessor is often the only choice for running memory intensive applications. In fact, some researchers argue that machines with a large main memory should always have multiple processors in order to make cost–effective use of the memory's capacity and bandwidth [WH95].

To run an application on a parallel architecture, the program's data and computation has to be mapped onto the different processors. In the context of this paper, a data layout not only provides a mapping of the program's data objects onto the parallel machine, but in addition may specify read–only copies of the data for some program regions. Read–only copies with appropriate life times and data mappings avoid otherwise necessary communication. However, read–only replication may not always be possible due to the increase in the program's memory requirements.

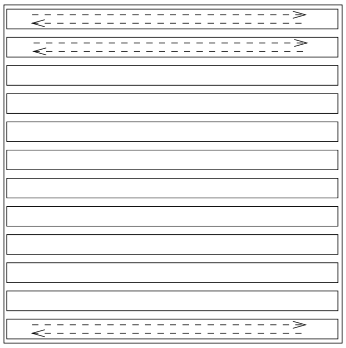
Many scientific applications contain array variables that are much less frequently modified than they are referenced (*glacial variables* [AW96]). A *read–only region* of an array variable is a program region where the array is not modified and all references to the array within the region refer to the same array value. Read–only copies of a glacial array within its read–only regions can reduce program execution times significantly if the array is needed with different mappings within the read–only regions.

We will discuss a new framework for automatic data layout for regular problems that performs read–only replication if possible and profitable. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. For a regular problem, the computation and communication requirements of a data layout can be determined statically, i.e., before program execution. Based on a specified problem size, machine size, and memory size on each processor, a data layout has to be selected that results in the fastest possible code that fits into the available memory. The resulting data layout may contain dynamic remapping.

The new framework can also be used to address other problems such as finding a data layout with the smallest number of processors on which a specified problem size can be executed with a parallel efficiency of at least x%. However, the discussion of such modified data layout optimization problems is beyond the scope of this paper.

The following example illustrates the issues an automatic data layout tool, a compiler, or a user is faced with when choosing an efficient data layout.

```
REAL c(N, N), a(N, N), b(N, N)

// READ (c, a, b)

DO   iter = 1, max
    // Forward and backward sweeps along rows

    DO j = 2, N
        DO i = 1, N
            c(i, j) = c(i, j) - c(i, j - 1) * a(i, j) / b(i, j - 1)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
        ENDDO
    ENDDO

    DO i = 1, N
        c(i, N) = c(i, N) / b(i, N)
    ENDDO

    DO j = N - 1, 1, -1
        DO i = 2, N
            c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
        ENDDO
    ENDDO
```



```
    // Downward and upward sweeps along columns

    DO j = 1, N
        DO i = 2, N
            c(i, j) = c(i, j) - c(i - 1, j) * a(i, j) / b(i - 1, j)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
        ENDDO
    ENDDO

    DO j = 1, N
        c(N, j) = c(N, j) / b(N, j)
    ENDDO

    DO j = 1, N
        DO i = N - 1, 1, -1
            c(i, j) = ( c(i, j) - a(i + 1, j) * c(i + 1, j) ) / b(i, j)
        ENDDO
    ENDDO

ENDDO

// WRITE (c, b)
```

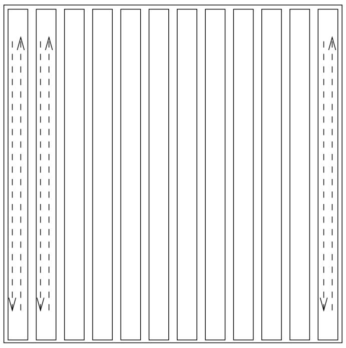


Figure 1: ADI integration kernel with computation illustration

Example

Figure 1 shows an Alternating Direction Implicit (ADI) integration kernel. ADI integration is a technique frequently used to solve partial differential equations (PDEs).

The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance. The same holds for a column layout for the column sweeps. Transposing the arrays between all row and all column sweeps eliminates communication within the sweeps. In the remainder of this section, such a data layout is referred to as *dynamic*. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between the sweeps but will make communication necessary either in the row or column sweeps (*static data layout*).

In the example kernel, array a is read in at the beginning of program execution and is never modified. Such a read–only array can be replicated with different data layouts in order to avoid otherwise necessary communication for the dynamic data layout. By keeping copies of a with a row–wise and column–wise distribution, transposing a between sweeps along different dimensions can be avoided. However, arrays c and b still have to be transposed since they are assigned new values during each dimensional sweep.

The price for read–only replication is an increase in memory–requirement. Therefore, replication has to be used selectively in order to prevent the memory requirement from exceeding the available memory on the target machine. For the dynamic data layout, replicating read–only array a will led to a 33% increase in memory usage over the versions of the program that keep only a single copy of each array at any given point during program execution.

In order to assess the benefits of read–only replication, we ran the ADI kernel with different data layouts on a IBM SP–2 multiprocessor. Figure 2 shows the execution times of a static, column–wise data layout, and two dynamic data layouts, with and without read–only replication. The data layouts were specified as HPF directives. For the dynamic layouts, explicit transpose operations were inserted in the code. The programs were compiled at the highest level of optimization using IBM's HPF compiler (xlhpf -O3) and executed on four thin nodes of a IBM SP–2.

In general, the best data layout choice will depend on the speed of the communication hardware and software of the target distributed-memory machine, and the ability of the compiler to exploit pipelined parallelism efficiently. In addition, the actual size N of the arrays and the number of available processors will influence the data layout choice.

In our example, the dynamic data layout with replication is the best for all problem sizes. In the cases where read–only replication is not possible due to the additional memory usage, the static data layout should be selected. For our example, all measured problem sizes fit into memory.

2 Previous Work

The *phase control flow graph* (PCFG) and the *data layout graph* (DLG) are the main program representation and data structure of our previous framework. A phase identifies operations

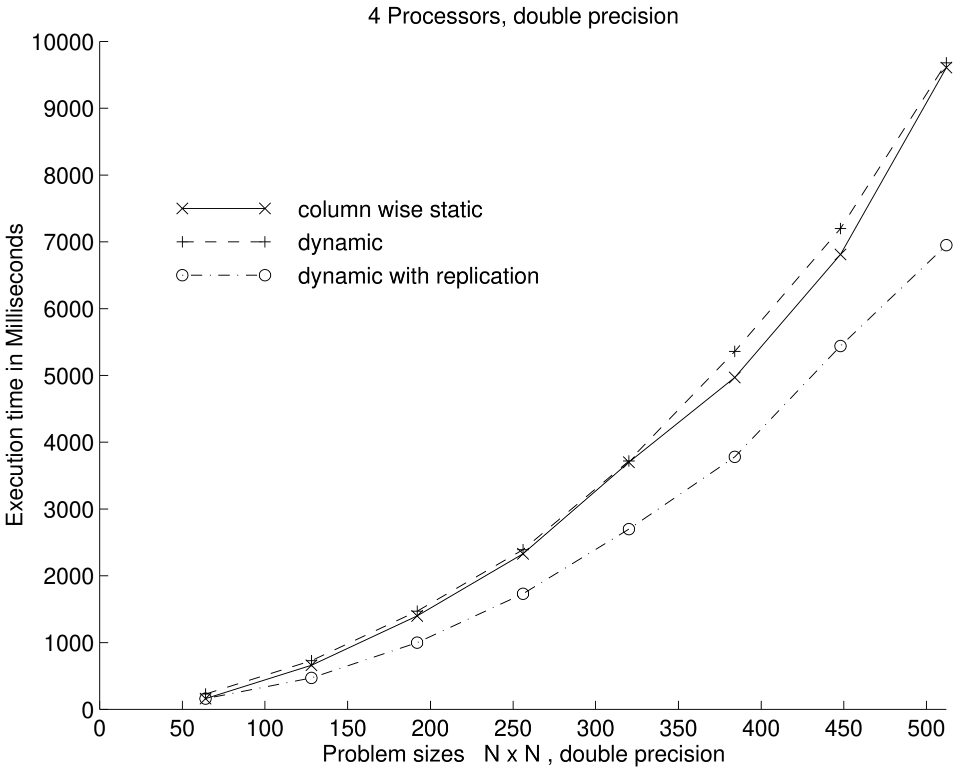


Figure 2: Execution times for different problem sizes on four IBM SP–2 nodes

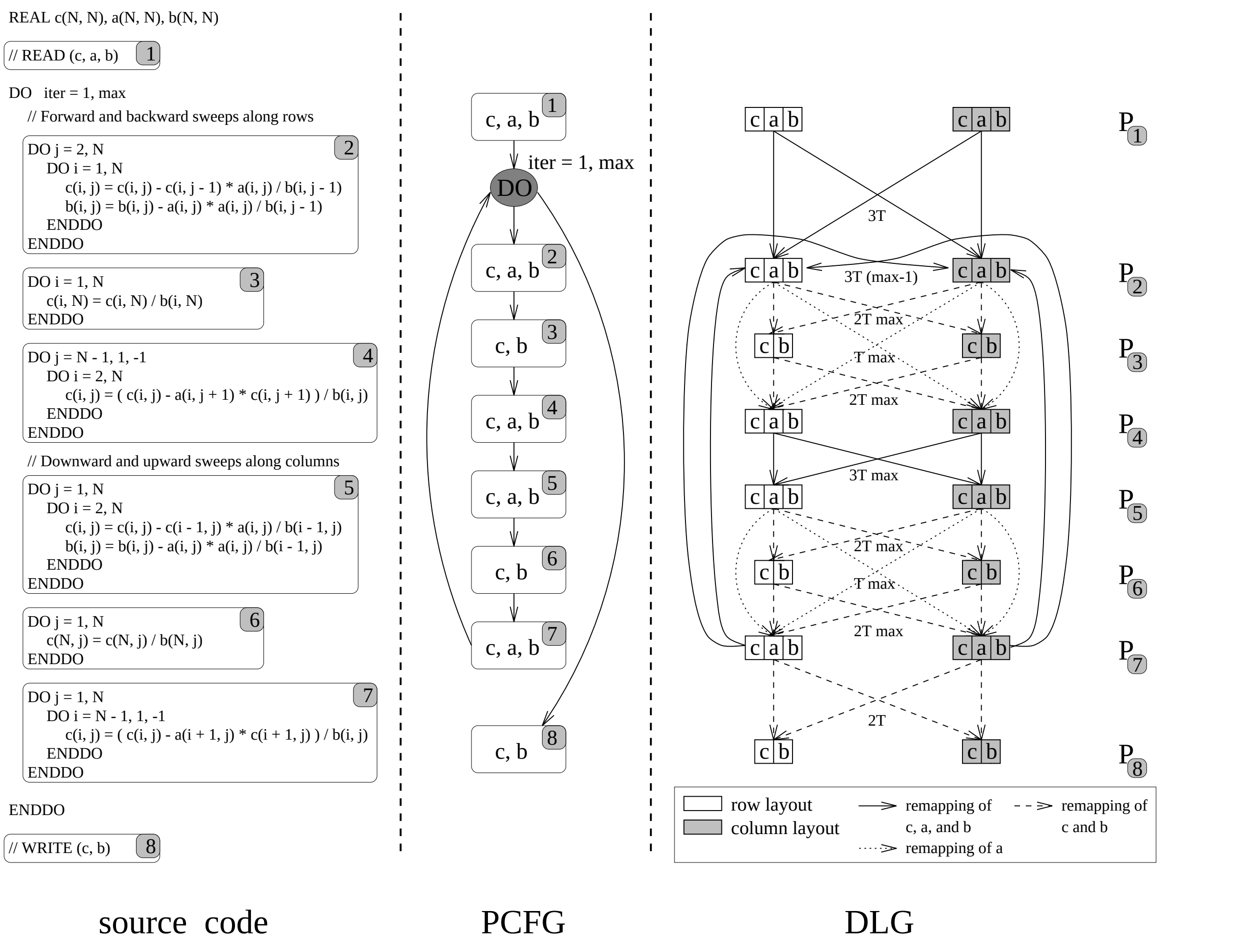


Figure 3: ADI integration kernel representations

on arrays between which remappings may be profitable. Figure 3 shows the partitioning of our ADI kernel example into phases, resulting in the PCFG shown in the middle of the figure. A PCFG is a compacted version of a control flow graph where all CFG nodes associated with a phase are represented by a single node in the PCFG, and edges are annotated with control flow information such as branch probabilities or frequency of execution [BKK94b, Kre95].

A node in the DLG represents a candidate data layout in the data layout search space of a phase. The edges represent possible remappings between candidate layouts. Nodes and edges are weighted with their estimated execution times. A solution to the data layout selection problem picks exactly one candidate data layout for each phase such that the overall cost of the resulting path is minimal. To simplify the DLG for our example in Figure 3, we assume that there are only two candidate data layouts in each search space. Node weights are not shown. Unlabeled edges have zero weight. "T" is the cost of performing a single array transpose, and "max" is the number of iterations of the outermost loop of the ADI integration kernel.

The DLG construction as described in [Kre95] uses the assumption that only a single copy of an array can exist at any time during program execution. Phases from which data layouts can be inherited are determined by solving the most recently referenced data flow problem (MR-Refs):

$$\text{MR-Refs}(b) = \bigcup_{x \in pred(b)} (\text{Refs}(x) \cup (\text{MR-Refs}(x) - \text{Kill-Refs}(x)))$$

where Refs(x) contains an entry for each array referenced at PCFG node x. Kill–Refs contains all references to arrays represented in Refs(x) outside of phase x. Each element in MR-Refs(b) is associated with a probability that the reference will actually reach phase b. If a reference to array a in phase p_1 is the LR–reference of a for phase p_2, and phase p_2 references a, then the DLG will contain remapping edges from all candidate layouts in phase p_1 to all candidate layouts in phase p_2. Note that for each array and control flow path that reaches a phase there is a unique MR–reference of the array. This is not true any more in the presence of read–only array copies since a path may contain multiple references to an array, resulting in multiple phases from which a data layout can be inherited.

3 New Framework

The new framework has to insert additional edges into the DLG to model the fact that a data layout for a read–only array can be inherited from the phase that defines the value of the read–only array, and from all phases that subsequently use the read–only array. In addition, the 0–1 formulation of the data layout selection problem over the DLG has to be modified in order to generate correct, minimal solutions in the presence of read–only copies. In the degenerate case where only single copies of read–only arrays are possible due to memory constraints, the new framework will produce the same global data layout as the framework discussed in our previous work. In other words, the estimated execution time of the data layout generated by the new framework can never be worse than the one for the data layout generated by the old framework.

In the following, we will discuss automatic data layout that considers read–only replications for arrays with single definition points and for single, perfectly nested loops without internal branches.

3.1 Single Perfectly Nested Loop

The single definition of a read–only variable can either occur within the loop or outside of the loop. The read–only region associated with the definition of a read–only array is the part of the program for which the read–only value is life, i.e., is its life range [Bri92].

The definition of a read–only array dominates all uses of the array in the PCFG. Read–only regions of different arrays may overlap, resulting in potential conflicts where not all desired copies may fit into the available memory.

All arrays referenced in the loop body are classified as either read–only or not read–only. The following discussion concentrates on the impact of read–only arrays on the DLG construction and 0–1 integer programming formulation of the data layout selection problem. For arrays that are not read–only, the standard DLG representation and 0–1 formulation is used as discussed in [BKK94b, Kre95].

3.1.1 DLG Edges

New edges for read–only arrays are introduced into the DLG as follows:

1. If the read–only region is restricted to a single iteration of the loop, edges are introduced between candidate layouts of a phase that references a read–only array a and all candidate layouts of phases that precede the phase and reference array a.

2. If the read–only region includes the entire loop nest, i.e., its definition occurs outside the loop, two sets of edges are introduced. The first set represents the case where control enters the loop the first time. The second set represents the situation in subsequent loop iterations. The potential sources from which data layouts can be inherited are different for the two cases.

   outside&first — During the first iteration of the loop, data layouts can be inherited from outside the loop and all phases that reference the read–only array and precede a phase in the loop body.

   outside&follow — During subsequent iterations of the loop, data layouts can be inherited from outside the loop and any phase that references the read–only array inside the loop.

   Based on these two sets, remapping edges between candidate data layouts are introduced and labeled with remapping costs that reflect the expected execution frequencies of the remapping if the edge is selected.

3.1.2 0–1 Formulation

The standard, single copy 0–1 integer programming formulation for the data layout selection problem over the DLG consists of a set of layout constraints and remapping constraints. In the new framework, additional constraints have to be introduced that describe the memory restrictions. In addition, the remapping constraints for read–only arrays have to be redefined.

<u>Remapping constraints</u> — The modifications to the remapping constraints for read–only arrays are as follows:

- If the read–only region is restricted to a single loop body iteration,
  - the IN–constraints for a read–only array A and a candidate layout have the form $\sum \ldots = x$, where $\sum \ldots$ is the summation over all 0–1 variables of incoming edges for A and x is the 0–1 variable for the candidate layout. In other words, the layout for A has to come from exactly one source data layout if the candidate layout is selected. If the candidate layout is not selected, no incoming edge can be selected.
  - the OUT–constraints for a read–only array A and a candidate layout have the form $\sum \ldots \leq x$, where $\sum \ldots$ is the summation over all 0–1 variables of outgoing edges to candidate layouts of the *same* phase, and x is the 0–1 variable for the

candidate layout. In other words, if the candidate layout is selected, one edge in any group of outgoing edges to candidate layouts of the same phase may be selected. However, selecting no such edge is allowed.

In contrast to the standard, single copy formulation the new formulation has typically more OUT–constraints since a given candidate layout may be inherited by more than one phase. In addtion, a selected candidate data layout in the new formulation may not be inherited by any phase it reaches. The latter property requires the selection of the $\leq$–relation in the constraint formulation instead of equality as used by the single copy formulation.

- If the read–only region includes the entire loop body, separate IN–constraints are formed for the *outside&first* and *outside&follow* cases. The form of the constraints is analogue to above.

  The same holds for the OUT–constraints.

Resource constraints — The formulation of memory constraints is based on the assumption that the sizes of all data objects in the program are known at compile time or tool invocation time. Define *avail_memory* as the amount of memory that can be used to store additional copies of read–only arrays. The resource constraints have to ensure that the sum of the sizes of read–only copies will not exceed *avail_memory* at any point in the program.

Resource constraints are constructed as follows:

1. For each phase that references a read–only array, compute the sets of edges associated with read–only arrays that are crossing the phase (1) during the first iteration of the loop body, and (2) during all subsequent iterations. Crossing a phase P means that there is an execution path such that the phase associated with the source of the edge is executed before phase P, and the phase associated with the sink of the edge is executed after P.

2. For each set E of crossing edges, compute constraints of the form

$$\sum_{e \in E} x_e \; size(e) \leq avail\_memory$$

   where $size(e)$ is the size of the array associated with the remapping edge represented by 0–1 variable x_e. The resource constraint is conservative in the sense that two selected edges may represent two read–only copies of a data object with the same layout, although the formulation assumes that each copy has a distinct layout. Crossing edges have only to be considered for phases that reference a read–only array.

   For efficiency reasons, the above constraints should be scaled down in order to generate small integral values. Note that the absolute values of $size(e)$ and *avail_memory* are not relevant for the correctness of the constraints. Scaling can be performed on a constraint by constraint basis.

3.2 Branches And Not Perfectly Nested Loops

In the single nested loop case, constraints were constructed for each execution path within the read–only region that reached a phase (*outside&first* and *outside&follow*). Each such summary execution path is associated with an execution probability. Any reference to the read–only variable on the path can reach the phase with this probability.

In the case of nested control flow with branches, it may not be feasible to distinguish all execution paths. In addition, the problem can be simplified further by not considering all references to the read–only variable on a given path as a source for a read–only copy. Heuristics can be used to select the classes of execution paths that should be considered together with the references on these paths that can generate read–only copies.

4 Experiments

To compare the efficiency of the old and new framework, 10 instances of the DLG for our ADI integration example were generated, where each phase was represented by five possible candidate data layouts. Each problem instance had different node and edge weights. Only array a was considered as a read–only variable.

In the following table, the column standard refers to the formulation of the data layout selection problem according to our previous framework. Column standard+rep is the formulation including replication edges and constraints. The rightmost column, standard+rep+constr includes in addition the resource constraints specifying enough (non-restr.) and insufficient memory (restr.) to allow the replication of array a. For the example program, seven resource constraints over the crossing edges are needed to express the memory restrictions, three for *outside&first* and four for *outside&follow*.

The problem instances were solved using *CPLEX*[1]V4.0, a linear integer programming tool and library, partly developed by Robert Bixby at Rice University [Bix92], running on a SUN UltraSparc1 (143MHz/64Mbyte). The reported numbers are for a solution strategy that uses a dual simplex instead of the default primal simplex for the initial relaxation[2]. The timing routine had a granularity of 10 milliseconds. In addition to the average solution times for all 10 DLG problems instances, the best and worst solution times are also reported.

[1] *CPLEX* is a trademark of CPLEX Optimization, Inc.

[2] Ed Rothberg from SGI suggested the use of dual simplex.

	standard	**standard+rep**	**standard+rep+constr**	
			non-restr.	restr.
sizes of 0–1 problem instances				
#layout variables	40	40	40	
#remapping variables	250	870	870	
#constraints	108	278	285	
solution times in milliseconds				
best	10	50	50	40
worst	20	110	100	60
average	13	68	75	50

The standard+rep+constr formulation that specified enough memory to allow the replication of array a (non-restr.) took approx. six times longer than the standard version, but generated a better data layout. In contrast, the version that did not provide enough memory to allow replication (restr.) took on average only 50 milliseconds to solve, i.e., approx. four times longer than the standard version. The comparison between standard+rep and standard+rep+constr shows the overhead due to the additional memory constraints.

5 Related Work

The problem of automatic data layout has been addressed by many researchers [AL93, CGS93, CGST93, Gup92, HA90, KLS90, KLD92, LT93, LC90, RS89, TA96, BKK+94a, Who91]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target machine architecture.

Our work is similar in nature to the recent work done by Anderson and Lam at Stanford University [AL93], Chatterjee, Gilbert, Schreiber, Sheffler, and Pugh at RIACS, Xerox Parc, and the University of Maryland [CGSS94, SSP+95], Ayguadé, Garcia, Girones, Labarta, Torres and Valero at the Polytechnic University of Catalunya in Barcelona, [AGG+94, GAL95], and Ning, Van Dongen, and Gao at CRIM and McGill University [NDG95]. The notion of glacial variables has been introduced by Autrey and Wolfe at the Oregon Graduate Institute in the context of program specialization and run–time code generation [AW96]. They showed that a significant number of glacial array variables can be found in ten of the PERFECT benchmark programs [Clu89].

In contrast to previous work, we are the first to consider read–only replication and memory constraints in a unified framework. More recently, other researchers have started to investigate the feasibility of 0–1 integer programming techniques in the context of automatic data layout [GAL95, Phi95] Using integer programming for instruction scheduling under resource constraints for super-scalar machines has been discussed by Feautrier [Fea94] and Ning, Govindarajan, Altman and Gao [NG93, AG94, AGG95].

6 Conclusion and Future Work

Read–only replication is an important technique to reduce communication costs by avoiding otherwise necessary global communication such as an array transpose. However, if not applied selectively, read–only replication may result in an executable that no longer fits onto the parallel machine. Since many application programs are memory intensive, uncontrolled read–only replication is not desirable.

We have developed a new framework for automatic data layout that allows read–only replication under given memory constraints. This new framework is an extension of our previous framework for automatic data layout. Preliminary experiments on a single program kernel showed that the overhead is significant — within a factor of six — as compared to our previous framework which allowed no read–only replication. However, the expected benefits of read–only replication are significant as well. More work will be needed to evaluate the new framework on larger problems and to develop an efficient treatment of complex loops and branch structures.

References

[AG94] R. Govindarajan E. R. Altman and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, December 1994.

[AGG+94] E. Ayguadé, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[AGG95] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[AL93] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[AW96] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *Proceedings of the Nineth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[Bix92] R. Bixby. Implementing the Simplex method: The initial basis. *ORSA Journal on Computing*, 4(3), 1992.

[BKK+94a] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[BKK94b] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0–1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[CGS93] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[CGSS94] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[CGST93] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

[Clu89] The Perfect Club. The Perfect Club benchmarks: efficient performance evaluation of supercomputers. *Int. J. Supercomp. Appl.*, 3(3):5–40, 1989.

[Fea94] P. Feautrier. Fine-grain scheduling under resource constraints. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[GAL95] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95)*, Houston, TX, April 1995.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, September 1992.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[KLD92] K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[Kre95] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, October 1995. Available as CRPC-TR95-559-S.

[LC90] 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 Computation*, College Park, MD, October 1990.

[LT93] P. Lee and T-B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[NDG95] Q. Ning, V. V. Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1995.

[NG93] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

[Phi95] M. Philippsen. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[SSP+95] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95)*, Houston, TX, April 1995.

[TA96] S. Tandri and T.S. Abdelrahman. Automatic data and computation partitioning on scalable shared–memory multiprocessors. In *Proceedings of the Nineth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[WH95] D. Wood and M. Hill. Cost–effective parallel computing. *IEEE Computer*, Feb 1995.

[Who91] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.