Optimizing Array Distributions in Data-Parallel Programs

Krishna Kunchithapadam krishna@cs.wisc.edu
Barton P. Miller bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

Data parallel programs are sensitive to the distribution of data across processor nodes. We formulate the reduction of inter-node communication as an optimization on a colored graph. We present a technique that records the run time inter-node communication caused by the movement of array data between nodes during execution and builds the colored graph, and provide a simple algorithm that optimizes the coloring of this graph to describe new data distributions that would result in less inter-node communication. From the distribution information, we write compiler pragmas to be used in the application program.

Using these techniques, we traced the execution of a real data-parallel application (written in CM Fortran) and collected the array access information. We computed new distributions that should provide an overall reduction in program execution time. However, compiler optimizations and poor interfaces between the compiler and runtime systems counteracted any potential benefit from the new data layouts. In this context, we provide a set of recommendations for compiler writers that we think are needed to both write efficient programs and to build the next generation of tools for parallel systems.

The techniques that we have developed form the basis for future work in monitoring array access patterns and generate on-the-fly redistributions of arrays.

1. INTRODUCTION

Parallel programming languages provide a programmer with abstractions that ease the development of code. Data parallel programming languages [1] allow the programmer to reason about a single thread of control that executes in parallel on a large collection of data. The mechanisms of parallel execution, communication and synchronization are all handled automatically by the compiler; the details are hidden from the programmer. While the abstraction of these details can simplify the programmer’s life, they can also obscure the cause of performance problems. Careless or naive array declarations or loop structures can cause poor performance, with no obvious explanation.

The performance of parallel programs is usually sensitive to the distribution of data amongst the processing nodes. For good performance, the programmer must choose an appropriate algorithm and a matching data distribution that minimizes communication. Figuring out optimal (or even acceptable)

This work is supported in part by NSF Grant CCR-9100968, Office of Naval Research Grant N00014-89-J-1222, and Department of Energy Grant DE-FG02-93ER25176.

data distributions can be a non-trivial problem−except in the simplest cases.

Current data parallel languages [2-4] require the programmer to specify data distributions. Our goal is to help the programmer improve their data distributions so that their programs run faster. In this work, we monitor the execution of a parallel program and use this information to improve the array distributions. We present an approach based on the execution-time tracing and post-mortem analysis of a data-parallel CM Fortran program. We formulate the optimality criterion in terms of a novel graph-coloring problem and provide some simple techniques for approximating an optimal solution.

We experiment with these techniques on a real mechanical engineering application program running on our TMC CM-5. Starting with an unfamiliar application program and using only the information provided by our analyses, we were provided clear information for modifying the array distribution directives in the program. These modifications resulted in a 30% reduction in predicted communications (based on our weighted graph representation).

Actual performance improvements were prevented in part because the CM Fortran compiler generated code that counter-acted our distribution directives (we describe this in more detail in Section 5). We provide a set of recommendations for compiler writers that we think are important in allowing both programmers and tool builders to make the best use of parallel programming environments.

Our current analysis of array accesses is post-morten, but our goal is to be able to monitor and evaluate access patterns on-the-fly (while the program is running). In systems that support execution time data redistributions (such as described in the HPF specification [5]), programmers would be able to improve their array distributions automatically.

1.1. Data Distribution in Parallel programs

Almost all parallel programming environments require programmers to specify the distribution of their data structures across the target hardware. In message passing programs, the data distribution and access is explicit and under the control of the programmer. In a data parallel language, the compiler and runtime system provide a set of data distribution pragmas that describe the layout of parallel data structures. We develop our initial work in the context of parallel arrays.

Data distribution pragmas specify the details of the layout of a parallel array onto the nodes of a parallel machine. For example, CM Fortran (which we used in our study) provides block-structured layout pragmas. A language like Fortran-D [4] supports both block and cyclic layouts while the HPF language definition mentions arbitrary and dynamically changeable permutations of arrays.

Usually, a programmer chooses a simple data layout scheme or lets the compiler choose a default layout. However, it is easy to come up with examples where a default layout can result in poor performance.

Much of the current work on this problem has focused on static compiler analysis [6]. The compiler either tries to figure out a good distribution from the control and data flow graphs of a program or explores a small number of canonical distributions for suitable ones. Static analysis is not always able to determine good data distributions, especially for access patterns that are dependent on the input data. An execution-time approach like the one we suggest would be useful in such cases.

Ultimately, we envision a combination of both static and runtime analysis; with compiler analysis determining static access patterns and runtime analysis tracking those parts of the program that are not amenable to static analysis. In addition, static analysis may also provide a good starting point for the runtime analysis tools.

2. OPTIMIZING DATA DISTRIBUTION AS A GRAPH COLORING PROBLEM

We model the access patterns of a program execution as a colored graph. Each vertex of the graph represents a part of an array and the color of a vertex represents the current processing node to which this array part is assigned. Edges of the graph represent assignments of values arising from part of one or more arrays to part of another array (assuming an owner-computes [7] model). This notion is made more precise below. Our approach to finding good data distributions is based on graph coloring with a specific conservation property on the graph. Our approach also weights communication costs over computation costs.

Proximity Graph Definition:

We define a *proximity graph* $G = (V, E, W)$ as follows:

V is the set of vertices of the graph. Each vertex $v \in V$ has a name $n_v \in N$ and a color c_v. Each element of each unique parallel array in a program forms a vertex in the proximity graph. The name n_v of a vertex is the name of the array containing the data element v. N is the set of all variable names used by the application. The initial color of a vertex is the node to which the data element was assigned by the compiler. The number of colors used in the graph G corresponds to the number of nodes of the parallel machine executing the application.

E is the set of directed edges of the graph. Each edge $e = (u, v)$ has a source vertex u, a destination vertex v, and a weight w_e. An edge e between vertices u and v in the graph corresponds to the fact that the data elements corresponding to u and v were used together in a computation in the parallel program.

More precisely, all assignment statements in a program can be written as *lvalue=expression*(*rvaluelist*). If we eliminate all temporary variables in the *rvalue* list, then the proximity graph for this statement will have an edge from each *rvalue* to the *lvalue*. An edge from u to v therefore indicates that the element corresponding to u was read and the element corresponding to v was written. The weight w_e of an edge from u to v represents the number of times u and v were used together (in the manner defined above) in the program.

W is the weight of the graph G and is defined as the sum of the weights of all edges e from u to v, such that $c_u \neq c_v$; only edges (or computation) between vertices of different colors (data elements located on different nodes of the parallel machine) contribute to the weight (communication cost). Computation involving data elements located on the same node does not contribute to the weight of the graph. ■

Balance Conservation Property:

The colors of the vertices V satisfy the conservation property that, for each name $n \in N$, with $n_i = n_j$, the number of vertices with colors c_i and c_j are equal for all vertex pairs (i, j), i.e. there is a balanced distribution of array elements across the nodes of a parallel machine. This property is also important in stating the minimization criterion on the proximity graph. ■

To perform optimizations on the data distributions, we define *balance preserving functions* that operate on G.

Balance Preserving Function:

We define a function F on graph G as follows:

$F(G) = (G')$, such that

1. $G = (V, E, W)$ and $G' = (V', E, W')$,
2. For all $u, v \in V$, $u', v' \in V'$, $c_u = c_{v'}$ and $c_v = c_{u'}$,
3. $n_u = n_{u'} = n_v = n_{v'}$. ■

In other words, the function F exchanges the colors of two vertices u and v (condition 2) iff they correspond to data elements of the same array (condition 3). Note that F preserves the balance conservation property of the graph. G' corresponds to a data distribution that is exactly like that of the original graph G, except that data elements corresponding to u and v have exchanged home nodes.

Given the above, the graph optimization problem would reach an ideal result when a sequence of applications of the function F to G produces $W(F^*(G)) = 0$; i.e., a sequence of color exchanges (data redistributions) that reduces the weight of the graph (communication costs) to zero. If such a sequence can be found, then the data distribution resulting from this sequence of applications of F to G will not lead to any communication during program execution and will therefore be optimal (assuming that computation costs are always negligible). In practice, we can hope only to find a sequence of color exchanges that minimizes the weight of the graph (since some communication may always be required).

We can now also motivate the need for a balance conservation property. If there were no restrictions on the number of vertices of a given color, then a trivial solution would assign the same color to all vertices; i.e. all parallel arrays are stored on one node, effectively reducing the application to a sequential one). Such a trivial coloring certainly has a zero communication cost, but is not interesting to parallel programmers. The balance conservation property is one way of ensuring a balanced data distribution onto the nodes of a parallel machine.

The proximity graph does not model any spatial or temporal locality in communication. If the compiler optimizes communication via block-transfers or caching, the optimization must be factored in when constructing the proximity graph.

3. THE STEPS IN OPTIMIZING ARRAY DISTRIBUTIONS

The problem of optimizing array distributions can be divided into three steps: (1) collecting dynamic array reference data to build the proximity graph, (2) computing a new (and better) array distribution by re-coloring the graph, and (3) converting the new distribution back to pragmas specifiable in the source language. We first describe some of the variations in each step, and then describe the variations that we have explored in our experiments.

Data Collection: Trace Granularity and Storage

Our model in Section 2 is based on counting the data movements between each pair of array elements referenced in a program statement. The basic trade-off is between post-mortem versus on-the-fly evaluation. The reference data could be collected by generating a trace record for each execution of a program statement; this collection of traces would be processed to construct the proximity graph. Alternatively, the data could be summarized as the program was executing; this would require a state space proportional to the number of pairs of array locations referenced during execution (potentially as large as the square of the program's total array space). The amount of primary memory needed for post-mortem tracing is simply the size of memory used to buffer the traces before they are written to secondary storage (or sent to another process for on-line reduction). If the traces are written directly to secondary storage, then secondary storage space requirements are proportional to the length of the program execution.

One technique for reducing the amount of data space needed by on-the-fly data collection is *data blocking*. Each array is divided into blocks and reference data is recorded in terms of the block in which an array location resides. Blocks could be formed by dividing an array into equal size rectangles. Data blocking also can reduce the size of secondary storage space needed for the post-mortem tracing. Regular patterns of access to an array will benefit from this technique. Irregular patterns, such as occur using index arrays (`A(B(i))`), could perform poorly using such a technique.

The granularity of trace data also has an impact on the size of the proximity graph. As the blocking factor is increased, the maximum number of possible edges (and vertices) in the graph decreases (as does the complexity of computing colorings). Data can be blocked at any time up to the moment of constructing a proximity graph, though if data is combined during instrumentation, the lost details cannot be recovered in the later stages.

Computing an Improved Distribution: Graph Coloring Algorithms

Once the array access data is collected, we use it to build a proximity graph and then use this graph to optimize array distributions. The initial coloring of vertices is based on the current assignment of array locations to processor nodes. Given an initial coloring, we evaluate the amount of inter-node communication caused by the coloring (array distribution) and compute a new coloring that attempts to reduce the amount of inter-node communication. The assumption is that reduced communication should result in improved program performance.

While computing the optimal coloring of the graph is NP-complete [8] there are many possible heuristics that we can use. Simple greedy algorithms are preferable because they are fast. Other possibilities include genetic algorithms [9], simulated annealing [10] and algorithms based on optimization problems [11]. The basic tradeoff is between the complexity of the algorithm and the quality of the solution. A simple, fast algorithm is more appropriate in the context of application steering.

Realizing a Distribution: Mapping a Coloring to Language Pragmas

The colorings produced by an optimization algorithm need to be expressed in terms of the data-distribution pragmas provided by the source language. It is possible that the pragmas of the source language are not powerful enough to express the data distribution patterns produced by the coloring.

Extracting a data distribution from a proximity graph is a form of pattern-matching; a formal description of the array distribution directives in the language would be matched against the proximity distribution.

A simpler approach would be to investigate a small number of array distribution specifications that we know are expressible in the source language. It is easy to compute the weight of a proximity graph based on a given distribution and compare it with the weight of the computed coloring. The enumerations that result in weights closest to the computed coloring are candidates for further investigation.

A third approach is to simply animate or visualize the data distributions produced by the graph coloring algorithm [12]. The programmer may be able to identify global patterns in a picture and map them to a array distribution pragma; block and cyclic distribution patterns may be especially easy to recognize in this manner. We expect that a practical tool would use some combination of the above techniques.

If the data access patterns of an application do not depend either on the problem size or on the degree of parallelism, we can construct a detailed proximity graph for a small problem running on a small number of nodes and use powerful optimization techniques to come up with good solutions to both the graph coloring and the distribution-to-pragma mapping problems. The cost of a detailed analysis can then be amortized over a large number of runs of the application on large data sets.

Carefully chosen training data-sets will also help in finding distributions that work well for most large-scale executions of the application.

4. RESULTS

4.1. The Experimental Testbed

To experiment with some of the ideas that we described in the previous section, we chose a mechanical engineering application written in CM Fortran. The application, DRBEM (Dual Reciprocity Boundary Element Method), was written by members of our Mechanical Engineering Department; therefore, we did not use any knowledge of the semantics of the program in any of our studies. The application was 2,200 lines long, distributed among 18 source files.

DRBEM is a non-linear solution technique used for heat-transfer and vibration analysis applications. The technique allows non-linearities to be solved as boundary integral problems. Like other boundary element methods, it relies on Green’s theorem to reduce a two-dimensional area problem into a one-dimensional line integral. The line-integral is then solved by discretizing and solvings sets of linear equations.

The application program reads in initial conditions from a file, sets up a system of linear equations, solves the equations for a series of time steps and finally writes the results to a file.

The sequence of viscous dissipation and convection computation, transient effect *curing* and linear-systems solution is repeated a large number of times (200 in our case) to obtain convergence. Therefore, even small array sizes have a significant impact on the execution time of the program. For example, array sizes of 100×100 result in an execution time of over a minute, while a full-scale run with arrays of sizes 1000×1000 take over an hour to complete.

To trace access to individual array elements, we hand-instrumented the program to write out array geometry and data access information to files. Hand-instrumentation is tedious, but was used only for this initial study. With the availability of def/use information from the compiler, the instrumentation step is straight-forward to automate. Ultimately, we expect to use binary rewriting [13] or dynamic instrumentation [14].

All program versions were run on a 32-node partition of a CM-5 computer (without using the vector units). The applications were compiled with CM Fortran (version 2.1.1-2) with optimization enabled. The trace data collected from the instrumented runs was used to build our proximity graph.

4.2. Our Investigations

In this section, we describe our algorithms and experiments. These algorithms and experiments cover a small subset of the different possibilities enumerated in Section 3.

Data Collection: Trace Granularity and Storage

In our experiments, we studied a single problem size–one involving arrays of size 32×32. This was done to keep the volume of trace data small. The application program was instrumented to trace all data accesses and a post-morten filter used to vary trace granularity. The advantage of this approach was that we could use data collected from a single run of the application many times.

For this problem size, we constructed proximity graphs based on the following trace granularities: (1) full tracing (2) tracing assuming a blocking factor of 2×2, 1×4, and 4×1 (3) tracing assuming a blocking factor of 4×4, 1×16, and 16×1.

Computing an Improved Distribution: Graph Coloring Algorithms

To compute a better coloring of the proximity graphs constructed in the previous step, we used a simple greedy algorithm. Our coloring algorithm was written to be as fast as possible. Its heuristic is very similar to a restricted version of the Kernighan-Lin [15] optimization algorithm; we examine only 2-neighbors while the Kernighan-Lin algorithm examines arbitrary *n*-neighbors.

Realizing a Distribution: Mapping a Coloring to Language Pragmas

To map the distribution given by the optimized proximity graphs, we used a simple visualization approach to assist the programmer in identifying regular patterns in the distributions.

In our experiments, a set of scripts extract the coloring information for arrays that were redistributed by the coloring algorithm and plot a picture of the original and modified distributions. We were able to immediately identify both the nature of the redistributions and the pragmas that could be used to specify the new distributions from looking at these pictures.

4.3. Evaluating the Tracing, Coloring and Remapping Steps

Our experiments report the difference in the weights of the proximity graphs for the default distribution and the distribution produced by the coloring algorithm. The reduction in the weights of the graphs is an upper bound on the reduction in execution times of the application–for a given coloring algorithm.

The first set of results shows the behavior of our graph coloring algorithm with variation in the tracing granularity. The metric used for evaluating the coloring algorithm is the reduction in the weight of the graph. In an ideal scenario, a reduction in graph weight should manifest as a reduction in execution time for the new distribution. The second set of results present the nature of the new distributions that were obtained from the coloring algorithm. For this step, we used a collection of scripts that would draw pictures of an array before and after the coloring (using different screen colors for different graph colors); here we were looking for simple and easy-to-identify patterns.

4.4. Evaluating the Coloring Algorithm

Table 1 summarizes the performance of our greedy graph coloring algorithm. The first column represents the trace granularity; a trace block size of $m \times n$ means that all array elements within rectangular blocks of dimensions $m \times n$ are represented by a single vertex in the proximity graph. A larger value for m and n results in the corresponding proximity graph having a smaller number of vertices and edges. The second column indicates the size of the proximity graph (as $|V|$, and $|E|$, where V and E are the vertex and edge sets of the proximity graph. The third, fourth and last columns show the initial weight of the graph, the weight after using our greedy coloring algorithm, and the percentage change in graph weights.

Block Size	**Graph Size** $\vert V \vert$	$\vert E \vert$	**Initial Graph Weight**	**Final Graph Weight**	**Change**
1×1	32,032	293,930	502,941	354,773	-29.5%
2×2	14,496	43,031	485,850	342,820	-29.4%
1×4	23,392	63,770	496,442	351,292	-29.2%
4×1	10,208	54,468	490,219	456,897	-6.8%
4×4	7,904	8,105	454,257	320,611	-29.4%
1×16	20,512	22,421	454,458	319,896	-29.6%
16×1	4,256	13,617	424,195	419,428	-1.1%

Table 1: Proximity Graph Data for DRBEM.

The results in Table 1 are grouped into three sets of rows. Each set represents a different number of array locations represented by a vertex in the proximity graph; the first row (1×1) has one element per vertex, the second set of rows has four elements per vertex, and the last set has 16 elements per vertex.

The execution time of the algorithm is (obviously) sensitive to the number of edges in the proximity graph. There is about an order of magnitude reduction in the running times from one granularity group to the next, as the blocking factor is increased.

It is interesting to note that except for the case of the $m \times 1$ blocking, all the other granularities for blocking trace data do not seem to perturb the coloring algorithm; the reduction in the weight of the proximity graph is the same. The differences between the original and new distributions also did not show any qualitative variation with the different blocking factors.

The default distribution that was chosen by the compiler was a column-major ordering of the arrays. Since the $m \times 1$ blocking treats elements of different columns (and hence different initial colors) as being identical, it loses some essential information and produces a poor proximity graph. The $m \times n$ and $1 \times n$ blocking heuristics both reduce trace sizes and preserve the initial distribution patterns and thus lead to more accurate proximity graphs.

Based on mapping information collected during the tracing of the application, we were able to tell which of the arrays of the application were redistributed†, namely: `drm_temp:s`, `drm_temp:scopy`, `drm_temp:dfx`, and `drm_temp:dfy`. We discuss only changes made to two-dimensional arrays; the redistributions of vectors were not significant.

† The names are of the form `function_name:array_name`.

4.5. Visualizing the Array Redistributions

To get an intuitive feel for how the coloring algorithm redistributed the above mentioned arrays, we extracted the geometry information of the arrays from the original traces and from the new distributions. We then displayed this information as a picture of the two distributions using different screen colors for different graph colors.

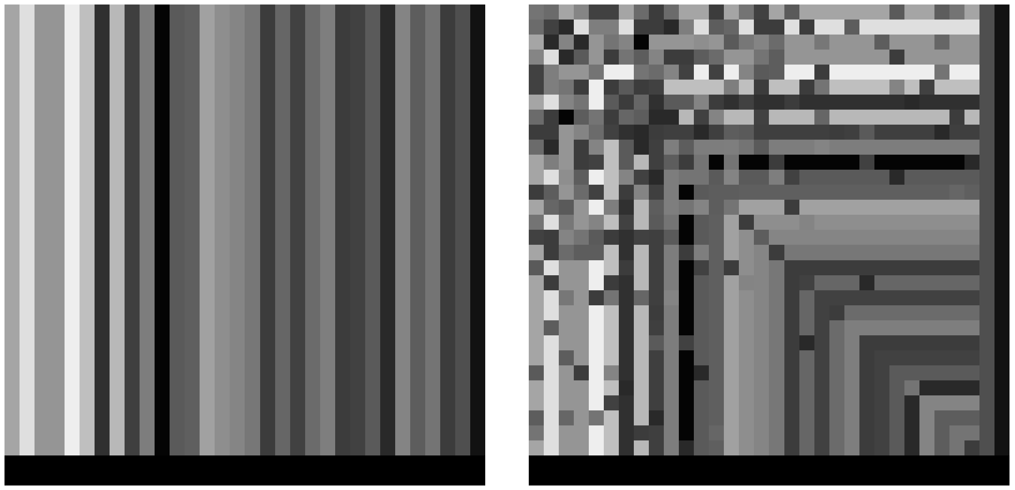
For the 1×1 blocking, the redistribution of the arrays `drm_temp:s` and `drm_temp:scopy` was immediately apparent. The CM Fortran compiler had allocated these two arrays by co-locating elements along the column while our coloring algorithm had co-located elements along the rows. The *rotation* of the original picture into the new one was unmistakable.

For granularities of 2×2 and 4×4, this *rotation* of the axis was equally apparent. However, for the $1 \times n$ and $m \times 1$ blocks, the pictures did not provide any insight into the new distribution or the way in which it differed from the original. It is possible that the arrays at which we were looking were too small for such blocking geometries to be useful. It is also interesting that the $1 \times n$ blockings not only provided no visual insight into the redistribution of the arrays but performed no better than the $n \times n$ blockings with respect to the graph colorings.

The pictures for the remaining two arrays `drm_temp:dfx` and `drm_temp:dfy` were similar to each other but not as striking as those for the previous two arrays. Once again, the compiler had originally allocated these arrays by co-locating elements along the column. In the new distribution, each of these arrays showed a wavefront pattern of distribution in the bottom right of the arrays (higher values of the row and column indices). The original and optimized distributions are shown in Figure 1.

4.6. Evaluating the effect of Redistributions

Based on the observations we made in the previous section, We modified the distribution of arrays `drm_temp:s` and `drm_temp:scopy` using the pragmas provided by CM Fortran. It was impossible to redistribute the other two arrays as indicated by the coloring without a language that supported arbitrary distribution primitives (like HPF). We rebuilt the application program with the geometries of the two arrays modified to favor co-location of elements along rows. Even though the geometry modifications we specified were close to optimal (within 1%), they had no discernible effect on the execution times of the application. The next section discusses reasons for the absence of performance improvements.



Original distribution of array drm_temp:dfx *Optimized distribution of array drm_temp:dfx*

Figure 1. Redistribution of array drm_temp:dfx

5. THE USER VS. THE COMPILER

5.1. Initial Results

We were initially encouraged to see a significant reduction in the weight of the proximity graph for the application. We expected to see the same (or comparable) reduction in execution time. However, the redistributions we specified did not change the runtime at all.

The CM Fortran compiler provides an interface to a few inquiry functions in the CM runtime system †. We used an array geometry description function to see if the arrays were being redistributed according to our new specifications. It turned out that the compiler was ignoring our distributions and using the default layout.

We tried a number of different distribution directives to get the compiler to generate the exact distribution we wanted but the CM Fortran compiler seemed to show a propensity for column-major distributions (while we wanted row-major distributions for our arrays). In the end we were only able to get the compiler to layout the arrays in N-row x M-column blocks, where N < M (but N is never equal to 1, which is what we wanted).

When we ran the applications with these somewhat sub-optimal distributions, we still expected a decrease in runtime. However, there was either no change or a very slight increase in runtimes.

† For example, a CM Fortran application can use these functions to determine/print the size and geometry of an array.

Using a compiler option, we then examined the names of communication functions and points in the code where they were called. It turned out that in each case where the specified distribution was in any way different from the default distribution, the compiler had inserted runtime system functions to redistribute these arrays into the canonical form before performing a parallel operation (and functions to redistribute any results back to the distribution specified by us). In essence, the compiler was undoing any possible optimizations that we specified.

5.2. Testing Simple Kernels: Matrix Multiplication

Since our application program was fairly complex with many interactions between different arrays, we started to work with small kernels to better understand the compiler.

Matrix multiplication is a simple kernel for which we knew the optimal data distribution. We wrote parallel versions of the kernel (using the intrinsic `matmul` and `dotproduct` functions). In each case, we varied the distribution of the source and destination arrays involved in the computation.

Once again, we noted that in each case where our distributions were different from the default ones, the compiler had generated code to redistribute the arrays into a canonical form.

CM Fortran does not generate parallel code unless the programmer uses parallel constructs or intrinsic functions. However, using these constructs or intrinsic functions constrains the programmer to use the canonical distribution, even though this distribution may be sub-optimal for the operation involved.

Moreover, for operations like matrix multiplication even the canonical distribution requires individual node-to-node communication. However, this communication is hidden within the runtime system. Even when the programmer specifies an optimal distribution, the compiler undoes the optimization to conform to the runtime system interface, and runtime system then re-redistributes the operands back into a more optimal configuration. Needless to say, it is better not to have redundant data moves in the first place.

5.3. Recommendations for Compiler support

Our experience is based on one compiler and runtime system on a small collection of programs. However, we would like to make the following general recommendations to parallel-compiler writers.

- The compiler should trust the programmer's data distributions, even if they are different from canonical ones. We found it very difficult to get the compiler to accept our data layout, let alone generate efficient code for it.

- A corollary to the previous recommendation is that in every case of automated analysis that a compiler makes on an application, it must provide manual interfaces of comparable quality so that knowledgeable programmers can perform the optimizations most appropriate to their code.
- The interface between the compiler and runtime system should be made either broader (e.g., the runtime system should provide different matrix multiplication clones, each best suited for certain combinations of operand layout) or narrower (e.g., the runtime system provides a single matrix multiplication function that is guaranteed to work with operands of all distributions, without the need for the compiler to specify data movements). With the current interface, neither the compiler nor the runtime system can make optimizations with the guarantee that they will not be undone by the other.

The above recommendations are of paramount importance if programmers are to get any benefit from using different kinds of data distribution pragmas. Strategies such as the one used in CM Fortran make data distribution pragmas redundant.

In addition, we would like compiler writers to provide tool builders with detailed information on compile-time analysis. Many of the problems we faced were specific to CM Fortran, but unless all compilers provide a reasonable amount of information, tool builders will not be able to write their tools.

- The compiler must use a consistent internal namespace to identify parallel data objects. For example, the CM Fortran compiler uses descriptors to identify arrays, but the meaning of a descriptor is different depending on whether the associated array is a parameter to a subroutine or is a local. There should be a one-to-one mapping between descriptors and array data.
- The compiler should also provide a mapping of descriptor names to source level names. Currently tools use ad hoc techniques to present results to the programmer in a usable form.
- We used source-level instrumentation in our study since no other way to get information on array references. If compilers provided a summary of the dependence analysis (in the form of def-use points in the code), tool builders would be able to use either binary rewriting or runtime instrumentation on the programs. Not only is this approach simpler than source instrumentation (or compiler modification), it allows tool writers to experiment with more programs–it is easier to obtain binaries than it is to get source code.
- The compiler should also provide information on the kind of optimizations it performs. Otherwise, there may be little correlation between the specifications made by the user, the optimizations done by the compiler and the analysis done by external tools.

6. CONCLUSION

In this work, we presented a graph-coloring based model for analyzing the data access patterns of an application and a simple algorithm that produces reorderings of the distributions that would result in reduced communication.

Our experiments with a real-world application and with test kernels show that compiler and runtime system interfaces need to be improved so that programmers and tools can make use of optimized data layout pragmas. We have provided a set of features that we, as tool builders, would like to see in compilers for parallel systems.

This work complements other approaches. We can use distributions from static analyses as an initial guess and tune it with data from a program execution. From typical data parallel compilers that treat array distributions as a given and then optimize code organization and data movement, we provide the additional dimension of choosing more effective data distributions. In this context, our work also forms the basis for generating on-the-fly data redistributions of arrays.

7. REFERENCES

1. W. D. Hillis and G. L. Steele, Data Parallel Algorithms, *Communications of the ACM*, December 1986, 1170-1183.
2. CMFortran Reference Manual (Online document), *Thinking Machines Corp. Version 2.2.1-2* .
3. C*: C-star Reference Manual (Online document), *Thinking Machines Corp. Version 7.1* .
4. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kramer and C. Tseng, Fortran-D Language Specification, *Technical Report, Computer TR90-141, Rice University*, 1990.
5. High Performance Fortran Language Specification, *High Performance Fortran Forum Version 1.0* (May 1993).
6. U. Kremer, J. Mellor-Crummey, K. Kennedy and A. Carle, Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment, *Technical Report CRPC-TR93-298-S, Rice University*, .
7. A. Rogers and K. Pingali, Process Decomposition Through Locality of Reference, *Proc. of the 1989 Conf. on Programming Language Design and Implementation*, Portland, Oregon, June 1989, 69-80.
8. U. Kremer, NP-Completeness of Dynamic Remapping, *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, December 1993, 135-141.
9. L. D. Whitley, Foundations of Genetic Algorithms, *M. Kaufmann Publishers*, San Mateo, California, 1993.
10. D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, Optimization by Simulated Annealing: An Experimental Evaluation, *Operations Research 39*, 3 (May-June 1991), 378-406.
11. J. R. Evans and E. Minieka, Optimization Algorithms for Networks and Graphs, *M. Dekker*, New York, 1992.
12. B. H. McCormick, T. A. DeFanti and M. D. Brown, Visualization in Scientific Computing, *Computer Graphics 21*, 6 (November 1987).
13. J. R. Larus and T. Ball, Rewriting Executable Files to Measure Program Behavior, *Software—Practice & Experience 24*, 2 (Feb, 1994), 197-218.
14. J. K. Hollingsworth, B. P. Miller and J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., 1994.
15. B. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell Systems Technical Journal 49* (1970), 291-307.