Hyperplane Partitioning : An Approach to Global Data Partitioning for Distributed Memory Machines

S. R. Prakash[*]and Y. N. Srikant
Department of CSA, Indian Institute of Science
Bangalore, India, 560 012

Abstract

Automatic Global Data Partitioning for Distributed Memory Machines (DMMs) is a difficult problem. In this work, we present a partitioning strategy called 'Hyperplane Partitioning' which works well loops with non-uniform dependences also. Several optimizations and an implementation on IBM-SP2 are described.

1. Introduction

Data Partitioning (or Distribution) for Distributed Memory Machines (DMMs) has been a difficult problem. Significant amount of work has been done in getting more and more parallelism from the programs [2]. However, without reducing communication overhead in the programs, they cannot be expected to run efficiently. There has been efforts in which the programmers are asked to give data allocation themselves and compiler will generate code automatically [3]. This might prove to be very difficult for programmers especially when the loops contain numerous array references. Reduction in communication overhead has been studied by many researchers such as Ramanujam and Sadayappan [6] where they tried to transform the programs to get parallelism as well reduce communication overhead. It has been seen, as in [7], that for a loop whose access patterns cannot be statically analyzed to get the best partitioning, compilers have traditionally generated sequential code. Although this pessimistic strategy is safe and simple, it essentially precludes the automatic parallelization of entire class of programs with irregular domains and/or dynamically changing interactions. For such loops, the general strategy adopted is to use *inspector*, *scheduler* and *executer* codes [7]. However, such techniques are for shared-memory machines, where there is no problem of data partitioning.

[*]with Hewlett-Packard India Software Operation, Bangalore

In this paper, we propose a method by which we can find a partition of a nested do-loops which reduces communication when executed on distributed memory machines. Our method differs from the other works in the following ways: *firstly*, no assumptions are made regarding whether loops are doall or not, as assumed in [1]. *Secondly*, the array references that are in the loops can have any linear functions of induction variables, not just $\vec{i}+\vec{c}$ type of functions of loop indices as assumed in [2] and others, where $\vec{i}$ is index vector and $\vec{c}$ is a constant vector. Such functions of loop indices can lead to to non-uniform dependences. According to an empirical study 44.34% of two dimensional array references contain coupled-subscripts [8], and most compiler run them sequentially due to difficulty in analyzing such loops.

2. Hyperplane Partitioning

We first see the program model assumed in this work and later we will see how to partition the iteration space using *Hyperplane Partitioning.*

2.1. The Program Model

In general, scientific programs contain a large number of array references in nested loops. In such programs, nested loops are main source of parallelism and are most time-consuming parts. A normalized n-nested loop [2] is considered in this work.

The body of the loop, $H[i_1, i_2, \ldots, i_n]$, contains a set of assignment statements possibly containing array references. The array references considered in this work are of the form, $X(a_{11}i_1 + a_{12}i_2 + \ldots + a_{1n}i_n + a_{01}, \ldots\ldots, a_{m1}i_1 + \ldots + a_{mn}i_n + a_{0n})$, which can be compactly written as $X(A\vec{i} + \vec{a_0})$ where $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$, $\vec{a_0}^T =$

$\begin{pmatrix} a_{01} & a_{02} & \cdots & a_{0m} \end{pmatrix}$

In this work, without any loss of generality, we assume that $m = n$. This assumption will not reduce the generality as we can add dummy dimensions, if $m < n$, or we can add dummy loops which runs for 0 times if $n < m$.

2.2. Iteration Space Partitioning

Consider the following example.

Example 1:

```
for i = 0 to N/2 do
 for j = 0 to N/2 do
  begin
   A[i+2j,i+j] = A[i,j] + B[i,j]
   B[i,j] = A[i,j]*2
  end;
```

The dependence graph for the Example 1 is given in Figure 1(a). The dependence graph shows how the index points are dependent on each other.

The problem here is to find the iteration partition such that the communication that is incurred in executing these partitions will be as less as possible. Finding the best possible partition, which is zero-communication partition, will require, finding the *vertical partition* of the dependence graph [2] which, in general, requires spanning the entire iteration space. This will take enormous amount of time especially when the number of loops are many and each running over a large number of iterations.

2.3. Computing the Hyperplane of Partition

The method to compute the Hyperplane of Partition, in brief, is as follows.

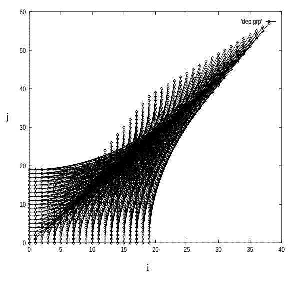
- First, for every pair of references, the dependence equation is computed.
- For each pair, the direction of dependence is computed.
- From these directions of dependences, we compute a hyperplane which is used to partition the iteration space into as many number of partitions as there are logical processors.
- These dependence directions induce data space partitions in every array used in the loop.
- Each logical processor executes different partition in parallel keeping corresponding data partitions locally, and synchronization and non-local accesses are handled at runtime.

Consider a loop given in Example 1. There are two arrays that are accessed in the loop in Example 1.We see that the need for communication arise only when there is dependence between iterations. On such situation different processors are trying to access same data which is owned by only one of them. If we localize such dependences (i.e, run both the source iteration and the target iteration on the same processor), and keep the data accessed by the iteration locally then we have removed the necessity to communicate for both synchronizations between processors (to satisfy inter-iteration dependences) non-local data, thus reducing the overall communication. For Example 1, the dependence graph for N=16 is shown in Figure 1(a). As can be seen, the dependence direction tend to align themselves along a particular direction (in this case the direction of (-0.67,1)), provided the conditions (given later) hold. By partitioning the iteration space along that direction and by placing the data accessed by these iterations locally, we can expect the communication to be reduced significantly. Further, since the dependence direction tend to align along a particular direction eventually, we expect the communication cost to reduce as the size iteration space increases. In the Figure 1(b) we see the effect for the Example 1 (The curve termed 'bm4.c'). Note also the same phenomenon does not happen with some standard HPF partitions like block distributions. Again referring to the Figure 1(b), the curve termed 'bm4.sc' shows this effect. The direction of convergence can be computed analytically for every pair of references in a loop and the hyperplane which "best fits" these set of directions will be taken as the hyperplane of partition for the iteration space. This hyperplane is induced into the different data spaces that are referenced in the loop to get the data partitions.

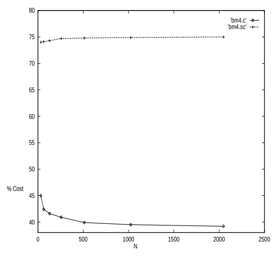
Theorem 2.1 (Dependence equation) *: The general dependence equation for the array references, $X[A\vec{i} + a_0]$ and $X[B\vec{i} + b_0]$ is given by : $\vec{d} = C\vec{i} + c_0$, where $C = B^{-1}(A - B)$, $c_0 = B^{-1}(a_0 - b_0)$ and any $\vec{j} \succ \vec{i}$ depends on $\vec{i}$ if $\vec{j} = \vec{i} + \vec{d}$. (See [4] for proof).*

In the above theorem, it is assumed that the coefficient matrices are non-singular, i.e, inverses exist. The case of singular matrices are dealt in [4].

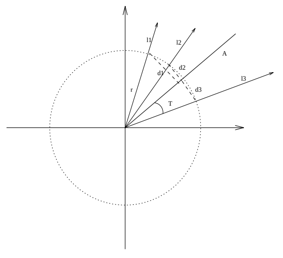
Definition 2.1 (Trajectory of index points) *For a given pair of references, for a given loop with lower bounds $\vec{lb}$ and upper bounds $\vec{ub}$, we can build a trajectory of index points by applying repeatedly the dependence equation from an initial index point $\vec{s}$ which takes us to the final index point $\vec{f}$, where both $\vec{s}$ and $\vec{f}$ lie between $\vec{lb}$ and $\vec{ub}$.*



(a) Dependence Graph



(b) The Nature of Dependences



(c) Best fit line for given lines

Figure 1. (a) Dependence Graph (b) Nature of Dependences (c) Best-Fit line

Definition 2.2 (Direction of Dependence) *The direction of dependence for a pair of reference, having subscript functions, $A\vec{i} + a_0$ and $B\vec{i} + b_0$, is defined as the direction, $\vec{d_k}$ such that $\vec{d_{k+1}} = \eta\vec{d_k}$, where $\vec{d_{k+1}}$ and $\vec{d_k}$ are dependence directions at two adjacent points on the trajectory of index points for the given pair of references and η is constant, provided such a constant exists. Otherwise direction of dependence is said to be oscillatory.*

Theorem 2.2 (Direction of Dependence) *Suppose $D = C + I$ (where C is the matrix of the dependence equation). The direction of dependence for a pair of references, having subscript functions, $A\vec{i} + a_0$ and $B\vec{i} + b_0$, is the eigen vector of the matrix, D corresponding to the dominant eigen value of that matrix, if D has n linearly independent eigen vectors and D has a dominant eigen value. (See See [4] for the proof.)*

The theorem 2.2 also says that this need not happen always the other case being when such eigen value does not exist. Then, the trajectory either will revolve round the origin spirally or diverge depending on whether eigen values are not real or real respectively. For such cases refer [4]. In this section, we will see how to get the hyperplane which partitions the iteration space, into as many tiles as the number of processors. These tiles can be used to induce data partition in the data space, using a particular reference in the loop for every array in the loop. These tiles and the corresponding data partitions can be placed in the local memory of the respective processor, and we can run those partitions in parallel by introducing the synchronizing messages to handle dependences. The hyperplane that minimizes the deviations from the given directions will be the one which minimizes the sum-squared of the sine of the angle between the hyperplane and the directions. Refer Figure 1(c). See [4] for details.

Theorem 2.3 (The best fit hyperplane) *Given p lines in n dimensions, with direction cosines, $\Delta x_{ij}, 1 \leq i \leq p, 1 \leq j \leq n$, passing through the origin, the hyperplane, $\sum_{j=1}^{n} a_j x_j = 0$ which also passes through the origin and is the best fit for the points at unit distance from the origin has the coefficients $a_j, 1 \leq j \leq n$, such that $Xa = b$, where the matrix $X_{kj} = \sum_{i=1}^{p} \Delta x_{ik} \Delta x_{ij}$, for $n - 1 \geq j, k \geq 1$ and $b_k = -\sum_{i=1}^{p} \Delta x_{ik} \Delta x_{in}$, for $n - 1 \geq k \geq 1$ and $a_n = 1$ (See [4] for proof).*

Finally, we need to partition the array data space from the iteration partition. The hyperplane that we got from the above analysis can be induced into the array data spaces also.

3. Compiler Optimizations

3.1. Space Optimization

We should use memory more efficiently by keeping just that amount of memory as is required by the partition to reside in the processor. Consider an array A in a loop with distribution as shown in the Figure 2. Every processor will use just the partition which it owns in the memory. This will result in what are known as *holes* in the memory. The locations which a processor doesn't own is called a *hole*. In general for processor p, the locations in sections P_k are holes for all $k \neq p$. The holes are not used, i.e, they are neither read nor written. The hyperplane which partitions the iteration

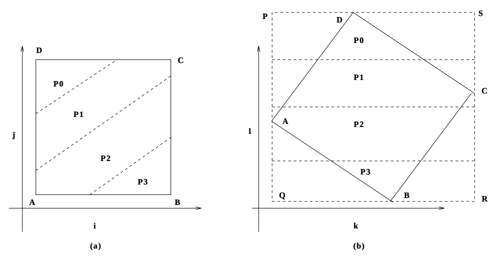


Figure 2. Illustrating holes in the memory

space and in turn which induces partition in the data space, can be thought of as the plane with *standard basis vectors*. Now, if we change the basis so that we make the hyperplane parallel to one of the axes-planes, then partitions would look very simple to compute.

Example 2: Consider Figure 2. For the sections in the figure (a) the corresponding sections after the change of basis is shown in the figure (b). The indexes have been changed from (i, j) to (k, l) and the section would be as shown in Figure (b). So, every processor will have one-fourth of PQRS instead of complete array ABCD. Still there are holes in the new sections as well, but fewer than what were there before.

Similarly we can construct the basis which has $(n - 1)$ vectors lying on the plane and one normal to the plane, so that we get the new partition plane which is parallel to this hyperplane which would be one of axes-planes with the new basis (as shown in the Figure 2). The Algorithm which constructs such basis is given in [4]. We can also prove the linear independence of such computed basis (see [4] for details).

3.2. Time Optimization or Uniform Scheduling

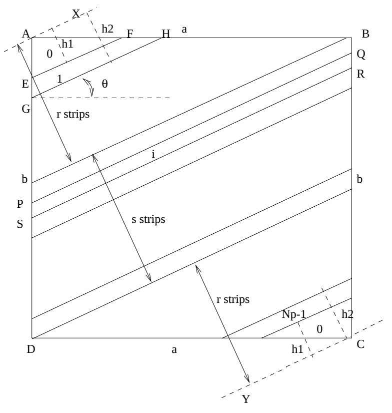


Figure 3. The Rectangular Iteration Space

Consider the rectangular iteration space ABCD as shown in Figure 3. We have to partition the ABCD with a hyperplane which makes an angle of θ with the X-Axis. The idea is to partition the iteration space into as many equal parts (by area) as there are logical processors (N_p). If these parts have (roughly) the same area then they will have same number of iteration points, thus scheduling will be uniform across processors [1]. We partition the iteration space $N_p + 1$ parts with the first and the last part owned by processor 0. With that we see that the middle strip covers the origin and localizes many dependences (since the dependences align themselves along the Eigen Vector passing through the origin). We partition the strips into three sets. The first and the last set with r strips and the second with s strips, so that $2r + s = N_p + 1$. The strip i are placed at a distance of h_i from A (or C). The algorithm to compute the strip sizes, h_i, is given in [4]. Essentially, we keep the strips at such distances to keep the area of each strip to be (roughly) the same. Thus we achieve uniform scheduling. The scheduling for non-rectangular iteration spaces and iteration spaces for higher dimensions with examples are given in [4].

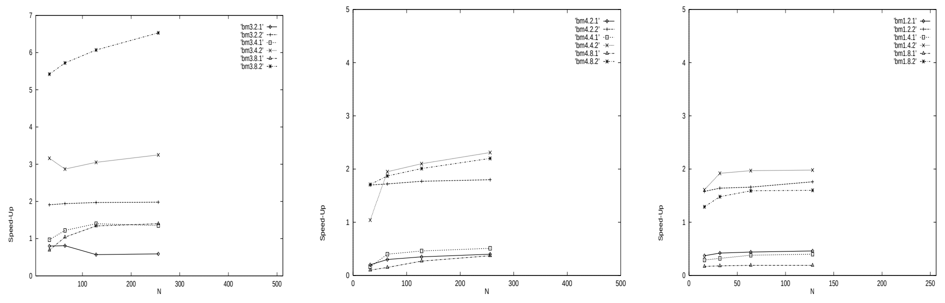
3.3. Message Optimizations

Since, sending a message is far more costlier than a computation, efforts have to be made to reduce the number of messages, even if that amounts to delaying few messages. We have implemented *Message Vectorization* and *Message Aggregation* by a single strategy in our tool [4]. Whenever a processor p has to send a message to another processor q, instead of sending the message immediately, it just waits to see whether there any more messages for the same destination, q. But amount of time it can wait is critical, since indefinite waiting can cause deadlocks. So, the processor p waits until *either* the buffer where the processor p has stored the pending messages, is full *or*, the processor p has to wait for any processor s for some other message which contains either the data or synchronization signal. We have seen the message optimization has reduced upto 60% of the messages and thus has improved performance to a significant amount. See [4] for details.

4. Performance

The Hyperplane Partitioning technique explained above, is for local optimization, i.e., for a loop. The same technique can be applied to a sequence of loops which ensures minimal communication for all the loops taken together when run on a DMM. This needs a good communication cost estimator, which estimates the communication that would be incurred if the given loop is run with given iteration and data partitions [5]. The tool, *Hyperplane Partitioner*, which we have developed will do the *Global Data Partitioning.* The Hyperplane Partitioner was tested for performance on some benchmark programs from NAS and some programs designed by us to show the merits of the tool. The Benchmarks selected were ADI (Alternating Direction Implicit) and SYR2K. We give the results for ADI here. (Refer [4] for SYR2K results).

ADI program has 6 loops with both single-nested and two-nested loops. Since the ADI loops are very short, we unroll the loops a few times, and then run them in parallel. The Global Data Partitioner had found that the best way to run the program is by partitioning the loop sequence into two regions, first with first three loops, and the second with last three loops,



(a) The Speed-ups for BM-1 **(b) The Speed-Ups for BM-2** **(c) The Speed-Ups for ADI**

Figure 4. The Speed-Ups for different benchmarks

notationally it is {{123},{456}}. The partition plane for the first set had the coefficients $(1, 0)$ and for the second set $(0, 1)$, which is (BLOCK,*) and (*,BLOCK) partitions respectively. Figure 4(c) shows the speed-ups for this benchmark for different sizes of arrays and different number of processors. In the figure, 'bmx.n.y' gives the performance for benchmark BM-x for 'n' processors. The number of times the loops were unrolled is given by 'y', which varies from benchmark to benchmark and which is found experimentally. In this case, if 'y' is 1 means we have to unroll 100 times and if it is 2 means unrolling has to be done 1000 times.

The first of our programs (BM-1) has five loops with three two-dimensional arrays. All the loops access the arrays in a similar manner (has the same dependence direction). This program is to show that the tool finds the static distribution if those are the best. Figure 4(a) shows the performance on IBM-SP2 for different sizes of arrays and different number of processors. For IBM-SP2, it has chosen to run all the loops with the same partition of the data spaces, i.e, {{0-4}} with the same hyperplane with coefficients $(1, -1)$. The second program (BM-2) has six loops with no static partition. There are three arrays accessed in different ways in different loops. For IBM-SP2, it decides to partition the loops as {{01}{2}{345}}. Figure 4(b) shows the performance on IBM-SP2 for {{01}{2}{345}}. We also saw by experiments that speed-ups for these programs with both BLOCK and CYCLIC distributions were inferior when compared to the partitions which our tool has suggested. See [4] for more details.

5. Conclusions

We have seen that there are many cases where we encounter loops which have coupled subscripts and we want to effectively run them on DMMs. The implementation results shows good performance for our tool with such non-uniform dependences. The tool also finds HPF-like distributions whenever such distributions are good. Inter procedural data partitioning analysis and a good scheme for global data partitioning for a general program are lacking in the current implementation.

References

[1] Ananth Agarwal, David Kranz, and Venkat Natarajan. Automatic partitioning of parallel loops and data distribution for distributed shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(9):943–962, September 1995.

[2] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Norwell, Mass.: Kluwer Academic Publishers, 1993.

[3] Charles H. Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[4] Prakash S R. Hyperplane partitioning : An approach to global data partitioning for distributed memory machines. Ph.d. dissertation, Submitted to Dept. Computer Science and Automation, Indian Institute of Science, Bangalore, July 1998.

[5] Prakash S R and Y N Srikant. Communication cost estimation and global data distribution for distributed memory machines. In *International Conference on High Performance Computing*, Bangalore, India, December 1997.

[6] 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, October 1991.

[7] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 23(5):537–576, May 1995.

[8] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of array subscripts and data dependencies. In *1989 International Conference on Parallel Processing*, volume II, pages 145–152, St. Charles, Ill., August 1989.