A Static Performance Estimator to Guide Data Partitioning Decisions

Vasanth Balasundaram* Geoffrey Fox† Ken Kennedy‡ Ulrich Kremer‡

Abstract

The choice of the data domain partitioning scheme is an important factor in determining the available parallelism and hence the performance of an application on a distributed memory multiprocessor. In this paper, we present a performance estimator for statically evaluating the relative efficiency of different data partitioning schemes for any given program on any given distributed memory multiprocessor. Our method is not based on a theoretical machine model, but instead uses a set of kernel routines to "train" the estimator for each target machine. We also describe a prototype implementation of this technique and discuss an experimental evaluation of its accuracy.

1 Introduction

Perhaps the most important intellectual step in preparing a program for execution on a distributed-memory parallel computer is to choose a data partitioning scheme for the fundamental data structures used in the program. Once decided, this partitioning often completely determines the parallelism in the resulting program. Unfortunately, there are no existing tools to help the programmer make this important step correctly. As of today, the programmer must guess a partitioning, implement it, and run the resulting program to determine its effectiveness. Comparing two data partitioning schemes requires implementing and running both versions of the program, a tedious task at best.

Several research groups have proposed "distributed memory compilers" that can automatically determine the parallelism in a sequential program, based on a specified data partitioning scheme [5, 21, 17, 18, 15, 12, 6, 14, 11, 19]. At Rice, the Fortran D implementation group is building a similar system to compile a Fortran 77 program, augmented with specifications of data decompositions, to generate efficient code for a distributed-memory parallel computer [8, 10]. In such a system, the choice of the decomposition is critical in achieving high efficiency. The project is faced with two important problems: (1) how can the compiler be designed to work well for a variety of target machines, when different target machines are differently balanced with respect to computation and communication, and (2) how can we provide the user with a way to predict the performance implications of a particular data partitioning scheme? Clearly, it is impractical to use dynamic performance information to solve these problems. We need an accurate *static* performance estimation scheme. If efficient enough, such a scheme could be used by the compiler to make decisions between code alternatives and by a programming environment to predict the implications of data partitioning decisions [3]. To be effective, the performance estimator needs to accurately predict the tradeoff points where the performance of one partitioning strategy crosses the performance of an alternative.

In this paper, we describe an experimental performance estimator that is aimed at the first of the problems above — prediction of the performance of a program with given communication calls under a given data partitioning scheme. This system is not based on a general theoretical model of distributed memory computers. Instead, it employs the notion of a "training set" of kernel routines that test various primitive computational operations and communication patterns on the target machine, and uses the results to "train" the performance estimator for that machine. The experience with our prototype, described later in the paper, indicates that this approach

*IBM T.J. Watson Research Center, vasb@ibm.com [Affiliated with Caltech during this work].

†Syracuse University, gcf@nova.npac.syr.edu [Affiliated with Caltech during this work].

‡Rice University, ken@rice.edu and kremer@rice.edu

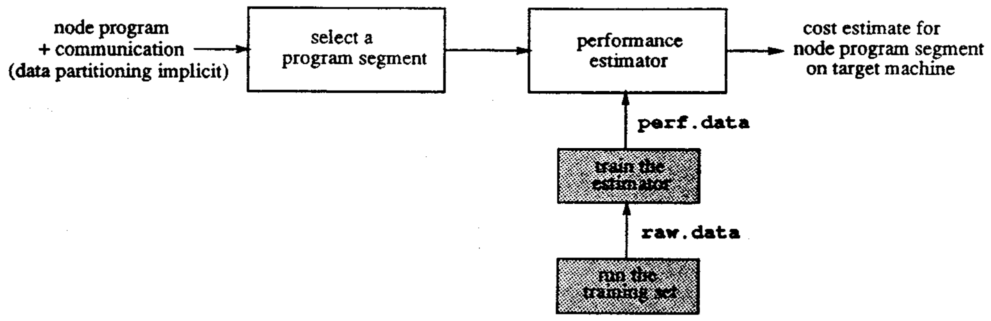


Figure 1: The performance estimation process.

is able to deliver surprising accuracy.

We believe that the training set approach can also be used to address the second problem—that of predicting the performance of code generated by a distributed memory compiler (such as the Fortran D compiler) on a particular target machine—once the compiler has been developed. A second "training set", consisting of a collection of kernel computations can be used to gather information about the compiler itself, such as transformations performed and communication primitives selected for each kernel computation.

Used in this context, the training set method provides a natural way to respond to changes in the compiler as well as the machine — simply rerun the training sets with the new compiler to initialize a new performance estimator.

2 The distributed memory programming model

We assume that the distributed memory program is written using the *loosely synchronous* model as defined by Fox, et.al. [9]. The salient features of this programming model are outlined below.

All processor nodes execute the *same* node program. The programmer therefore only writes a single generic node program (and perhaps a host program that executes on some front-end machine).

All computation within a processor's node program can only involve data items contained in the processor's local memory. Non-local data items must be accessed through inter-processor communication, which is assumed to be implemented via message-passing. Thus, *the writing of the node program implicitly defines the partitioning of the data domain.*

Communication between a group of processors imposes a synchronizing condition among all the proessors. This implies that *all the processors operate in a loose lockstep, consisting of alternating phases of parallel asynchronous computation and synchronous communication.*

The previous feature is an important property of the loosely synchronous distributed memory programming model: any data transfer among processors occurs simultaneously in concert, with all processors participating within a well-defined phase. This allows any inherent regularity in the communication pattern to be exploited for maximum efficiency on the target machine. Experiments conducted on a wide variety of real applications has shown that a large class of regular problems and some spatially irregular problems are very well suited for the loosely synchronous model [9].

One disadvantage of this model is that it may not be well suited for temporally irregular problems. However, in this paper we restrict ourselves to problems with regular geometries whose data dependency graphs can be computed statically (i.e., spatially and temporally regular problems), so that the loosely synchronous model is well suited to our needs.

3 Choosing the data partitioning scheme

Let us consider the problem of deriving a distributed memory node program (specified according to the loosely synchronous model) from a sequential program. This requires three steps: (1) identify opportunities for data parallelism in the sequential program, (2) partition the data domain across the processors, and (3) generate the node program (with loosely synchronous communication), which, when executed in parallel by all the processors, exploits the available data parallelism.

Step (3) involves the specification of the program that operates on a generic data partition, and also the

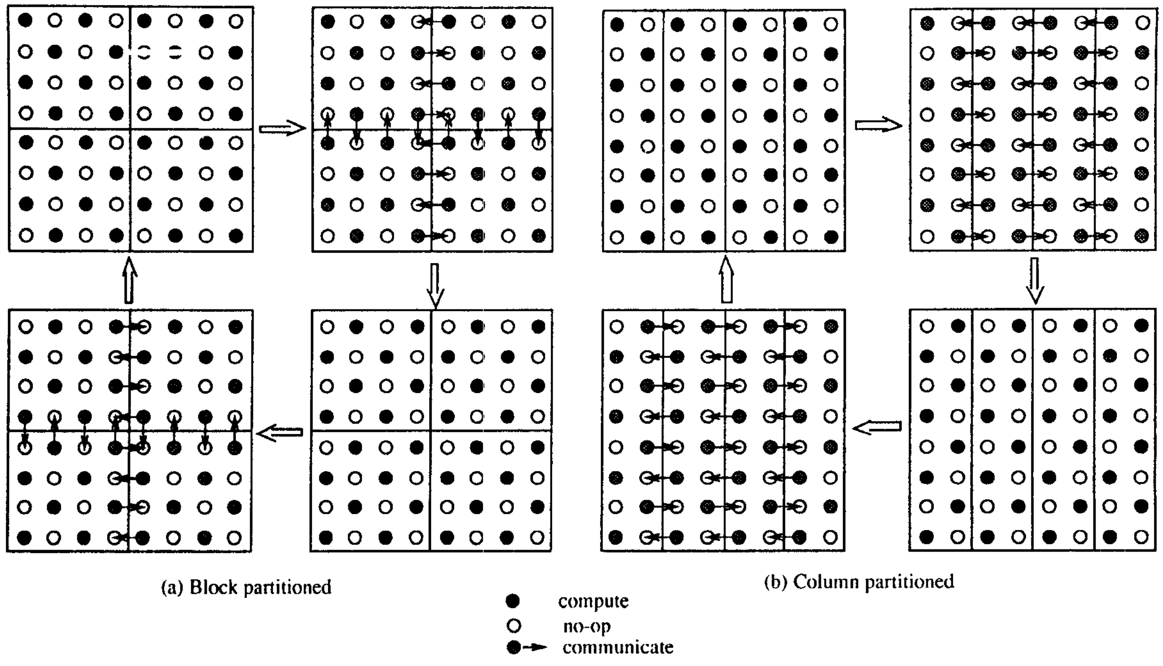


Figure 2: Two possible data partitioning schemes for program REDBLACK.

specification of the loosely synchronous inter-processor communication. This can be done automatically by a "distributed memory compiler" as mentioned earlier. Such a compiler would typically use the *data dependence graph* of the sequential program, along with the data partitioning information, to derive the node program and the communication. For the purposes of this paper, we will assume that steps (1) and (2) are the programmer's responsibility, and step (3) is performed by a distributed memory compiler.

Experiments with preliminary implementations of such compilers have indicated that the choice of the data partitioning strategy strongly influences the performance of the parallel program on the distributed memory machine. This is not surprising, since it is the data partitioning scheme that ultimately determines how much of the available data parallelism in the application can be effectively exploited on the target machine.

In an attempt to better understand how the data partitioning strategy affects the performance of the generated node program, we decided to build an interactive data partitioning tool. Our idea was to use this data partitioning tool to statically explore different data partitioning schemes without having to run the node program each time on the machine. Details of this tool are described in an earlier work [3].

The techniques discussed in that earlier work focussed on the derivation of the node program and the inter-processor communication in response to the user's choice of a particular data partitioning scheme. We wanted to add to the tool the ability to statically determine which choice of data partitioning was best for a given target machine. In order to do this, we needed a scheme for comparing several different data partitioning schemes statically. Providing a measure for comparing different data partitioning schemes statically would be very useful in a data partitioning tool, as well as in sophisticated distributed memory compilers that attempt to derive the data partitioning scheme automatically. Clearly, any relative evaluation of data partitioning schemes will depend not only on the nature of data dependences in the program, but also on several target machine specific parameters. To make our job easier, we will restrict ourselves to data partitions that are rectangular, and also assume that the data domain is partitioned uniformly (i.e., all partitions are of the same shape). Extending our techniques for more general cases is a topic of future research.

4 An example

To better motivate the relationship between the data partitioning scheme and factors such as communication overhead, data domain size, execution time and number of processors used, let us consider an example. Program REDBLACK, listed below, is a segment of the node program for a pointwise relaxation using the "red-black" checkerboard algorithm.

The original data domain for this problem is a 2 dimensional grid, which must be appropriately partitioned across the processors. In the following node program segment, (idim × jdim) is the size of each processor's data partition. This local data is stored within each processor's local memory as an array val(0:idim+1, 0:jdim+1). The interior of the array val(1:idim, 1:jdim) contains the actual elements of the processor's local partition of the data domain, while the borders val(0, 1:jdim), val(idim+1, 1:jdim), val(1:idim, 0) and val(1:idim, jdim+1) are used to store boundary data elements that are received from the neighboring processors.

```
program REDBLACK

do k = 1, ncycles
   //Compute the RED points in my partition.
   do j = 1, jdim, 2
      do i = 1, idim, 2
         val(i,j) = a*(val(i,j-1) + val(i-1,j)
                    + val(i,j+1) + val(i+1,j)) + b*val(i,j)
      enddo
   enddo
   do j = 2, jdim, 2
      do i = 2, idim, 2
         val(i,j) = a*(val(i,j-1) + val(i-1,j)
                    + val(i,j+1) + val(i+1,j)) + b*val(i,j)
      enddo
   enddo
   //Communicate RED points on my partition
   //boundary to the neighboring procs.
   call communicate (val, RED, neighbors)
   //Compute the BLACK points in my partition.
   do j = 1, jdim, 2
      do i = 2, idim, 2
         val(i,j) = a*(val(i,j-1) + val(i-1,j)
                    + val(i,j+1) + val(i+1,j)) + b*val(i,j)
      enddo
   enddo
   do j = 2, jdim, 2
      do i = 1, idim, 2
         val(i,j) = a*(val(i,j-1) + val(i-1,j)
                    + val(i,j+1) + val(i+1,j)) + b*val(i,j)
      enddo
   enddo
   //Communicate BLACK points on my partition
   //boundary to the neighboring procs.
   call communicate (val, BLACK, neighbors)
enddo
```

The node program given above is the code that is executed by each processor, on its local partition of the data domain. Depending on the values of idim and jdim, the size of the partition can be varied. This lets us change the data partitioning scheme by merely changing the loop bounds, and without having to make any extensive modifications to the node program.

For example, Figure 2 shows two possible partitioning schemes of a 2 dimensional data domain of size 8×8. The two data partitioning schemes shown in the figure are block-partitioning and column-partitioning. When idim = 4, jdim = 4 each partition is a 4×4 block, and we get the block-partitioned case. When idim = 8, jdim = 2, each partition is an 8×2 strip, and we get the column-partitioned case. In both cases, four partitions of equal size are created, which are assigned to four processors of the target distributed memory machine.

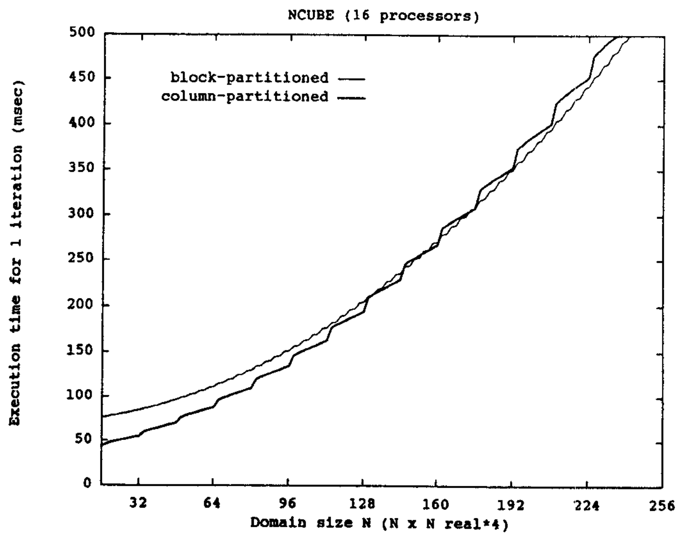
Figure 2 also illustrates the compute-communicate sequence for the block-partitionined and column-partitioned schemes. There are several methods for performing the communication between neighboring processors (indicated in the figure by arrows that cross partition boundaries). In most existing distributed memory machines, message startup cost is usually an important factor in determining the particular communication strategy to employ. A useful rule of thumb is: communicating a small number of large messages is more efficient than communicating a large number of small messages. Program REDBLACK achieves larger size messages by performing the communication outside the do j loop, so that all the boundary elements can be communicated together as a vector, instead of individually.

We wrote the REDBLACK node program in Fortran, using the EXPRESS[1] programming environment. EXPRESS is a commercially available package, that extends Fortran77 with a portable communication library, and is based on the loosely synchronous programming model [7]. The program was executed on the NCUBE, and execution times were determined for increasing data domain sizes and number of processors. Figure 3 shows the graph of data domain size (i.e., size of the original 2 dimensional problem grid) versus average execution time for a single iteration of the do k loop. For a data domain of size $N \times N$, and P processors, the logical processor interconnection topology for the block-partitioned case is assumed to be a $\sqrt{P} \times \sqrt{P}$ 2 dimensional grid, with each processor getting a block partition of size $N/\sqrt{P} \times N/\sqrt{P}$ data elements, and for the column-partitioned case it is assumed to be a linear array of size P with each processor getting a column partition of size $N \times N/P$ data elements.

For example, when $P = 16$ and $N = 32$, block-partitioning assigns to each processor a partition of size 8×8, while column-partitioning assigns to each processor a partition of size 32×2.

It is interesting to note the crossover point of the curves for the block-partitioned and column-partitioned cases. It indicates that the efficiency of a data partitioning scheme is dependent on the data domain size as well as number of processors used. For 16 processors, column partitioning is more efficient than block partitioning for data domain sizes less than 180×180, whereas for 64 processors, the critical do-

[1]EXPRESS is a copyright of ParaSoft Corporation.



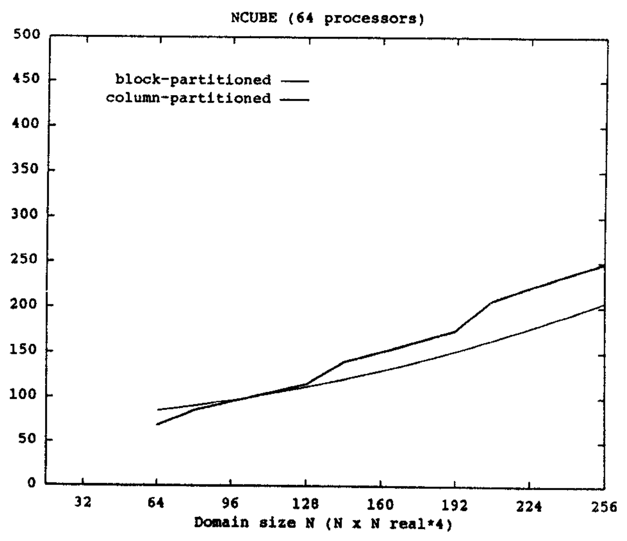


Figure 3: Graphs of data domain size vs. average execution time of the main loop for program REDBLACK on the NCUBE.

main size is 128 × 128.

The steps in the curves are caused by load imbalance effects. When the data domain size N is not exactly divisible by the number of processors P, some processors are assigned an extra data partition to work on. The load imbalance created due to this is greater for the column-partitioned scheme compared to the block-partitioned scheme, resulting in larger step sizes for the column-partitioned case.

5 The Training Set Method of performance estimation

The REDBLACK example illustrates some of the subtleties involved in trying to compare different data partitioning schemes. We experimented with different partitioning strategies on several distributed memory machines, including the NCUBE, Symult S2010 and the Meiko transputer array[2], and attempted to derive a theoretical performance model based on our observations. Our attempts were based on published techniques for developing performance evaluation models, such as [20, 1] for example. We found that it was difficult to derive a single theoretical model that could predict the experimentally observed behavior of the node program on different data partitioning schemes.

Although the theoretical model could predict overall program execution time with a fair degree of accuracy, it failed to predict the *crossover point* (see Figure 3) of the curves with consistent accuracy. Several other effects such as the undulations (or change in slope) of the curve were also difficult to predict. The accuracy of estimating the overall execution time did not matter so much to us as the accuracy of estimating the crossover point at which one data partitioning scheme is preferable over another.

Note that it in general it may be impossible to specify the "crossover point" in terms of data domain size alone, as suggested by the graphs in Figure 3. The crossover point may shift depending on the particular communication routines used for message-passing, hardware and operating system peculiarities of the target machine and on the implementation of the low-level software layer. It could also be affected by minor algorithmic changes to the program.

We were eventually convinced that a purely theoretical model was too general for our purposes, and attempting to refine the theoretical prediction techniques would only result in increasingly complex models with little improvement in accuracy. We began experimenting with alternative methods of performance estimation that are more directed to our specific needs. The resulting technique, called the *Training Set Method*, will now be described.

The Training Set Method consists of two initialization steps that must be performed once for each target machine (see Figure 1), and an estimation algorithm that can be invoked by an interactive tool or a distributed memory compiler on any node program to be run on the specified target machine.

[2]All programs were written in EXPRESS.

In the first initialization step, a set of routines called the *training set* is run on the target machine. The training set is a node program that performs a sequence of basic arithmetic and control flow operations, followed by a sequence of communication calls. The average execution time of the arithmetic and control flow operations are measured on the target machine, and the data is written out to a **raw.data** file. The communication calls are designed to test several data movement patterns using all possible combinations of the various message-passing utilities supported on the target machine. The average times taken for each communication is measured for increasing data sizes and processor numbers, as well as for unit and non-unit data strides (e.g., communicating a column of data vs. communicating a row of data in Fortran). This data is also written out to the **raw.data** file as a set of (x, y) points, where x is the message size in bytes and y is the average time that *one processor* spends in participating in the loosely synchronous communication. In the present version of this system, the training set is written in EXPRESS, and the communication calls use the loosely synchronous message-passing utilities provided by EXPRESS.

In the second initialization step, the performance data in the **raw.data** file is analyzed and the raw performance data is converted into a more campact and usable form. Functions are fitted to the communication performance data and average values are computed for the execution times of the arithmetic and control operations. The functions and the average values are then written out to a **perf.data** file. The **perf.data** file is a compressed form of the **raw.data** file. On the NCUBE for example, our preliminary implementation resulted in a **raw.data** file that was several tens of Mbytes in size, while the size of the **perf.data** file was around a Kbyte. The functions in **perf.data** will be used by the performance estimation algorithm to reconstruct the communication performance data in the **raw.data** file.

A variant of the chi-squared fit method [16] is used to fit functions to the communication data in the **raw.data** file. The communication data however, are piecewise linear functions, and the chi-squared fit can only be applied to continuous linear functions (see Figure 4). The discontinuities arise due to packetization costs incurred by the need to pad the message so that it can be sent as a whole number of packets. To get around this caveat, we use the chi-squared fit to fit a function of the form y = a + bx only to the continuous linear segments of the piecewise linear curve. In addition, the step size between the adjacent linear line segments is also determined. Knowing the function y = a + bx for a continuous line segment, the step size between adjacent line segments and the packet size on the machine, we can reconstruct a close approximation to the original piecewise linear curve of the communication performance data as specified in the **raw.data** file. Further details on the construction of the training set are given in [2].

The graph in Figure 4 shows the actual communication performance data for some data movement patterns on a 64 processor NCUBE. Although this is only a small sample of the data obtained using the training set, it is illustrative of the unusual behavior of different communication patterns. The curves in the graph correspond to the following communication patterns, all written using EXPRESS:

- **iSR:** circular shift using individual element send/receive. All processors send data to their right neighbor and then receive data from their left neighbor.
- **vSR:** same as above, except that the data is communicated as a vector instead of element by element.
- **EXCH:** synchronous circular shift (also called "exchange"). The communication is done in two stages. During the first stage, all even numbered processors send data to the right, while all odd numbered processors receive data from the left. In the second stage, all odd processors send to the right while the even processors receive from the left. Contrary to **vSR**, the **EXCH** mechanism of pairing of sends and receives between neighboring processors ensures that the receiving processor is always prepared to receive the message, regardless of its size.
- **BCAST:** one to all broadcast. Note that it is incorrect to conclude from the graph that "broadcast is faster than circular shift". This is because the graph shows the time that each processor contributes (measured as an average across all the processors) towards the overall data movement, and *not* the time for the completion of the communication call itself.
- **COMBN:** the "combine" operation, where a global reduction is performed using some associative and commutative operator, after which the results of the reduction are communicated to all the processors. This is equivalent to performing a tree reduction over the processors while applying the reduction operator at each node, and then broadcasting the result computed at the root of the tree to all the processors.

6 The performance estimation algorithm

Procedure ESTIMATE shown in Figure 5 implements the performance estimation algorithm. The algorithm

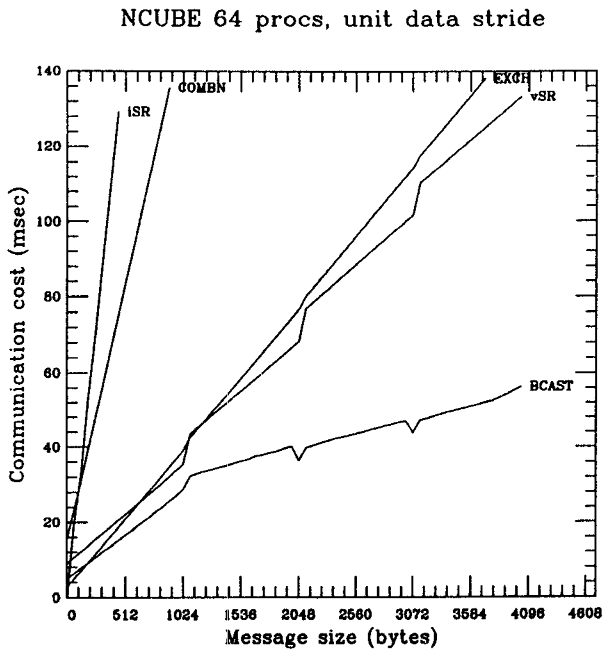


Figure 4: A sample of the communication performance data for the NCUBE.

uses the arithmetic/control and communication performance data stored in the `perf.data` file to determine an execution and communication time estimate. The procedure takes as input a node program statement, which can be either a *simple* statement or a *compound* statement. A compound statement is one which consists of other compound and simple statements, while a simple statement does not. An assignment is an example of a simple statement; do loops, subroutine calls, if-then, etc., are examples of compound statements. The **program** statement in Fortran is a special case of a compound statement: it consists of the entire body of the program. The execution time estimate etime and the communication time estimate ctime are assumed to be initialized to zero.

Procedure ESTIMATE allows us to estimate the execution time and communication time of any selected node program segment, regardless of its size. The branch probability is assumed to be 0.5 by default, but the user is allowed to override it. Note that this algorithm is best used for *comparing* two different data partitioning schemes (i.e., two different node programs for the same application). The etime and ctime values for the different partitioning schemes can then be compared to get a fairly good estimate of what their relative performance on the target machine would be.

7 A prototype implementation

We implemented a prototype version of the estimator in the ParaScope interactive parallel programming environment [4]. Figure 7 shows a screen snapshot during a typical performance estimation session.

The user can edit the distributed memory node program within ParaScope, and make appropriate changes to it to reflect a particular data partitioning scheme. A statement such as a do loop can then be selected, and the performance estimator invoked by clicking on the `estimate` button. ParaScope responds with an execution time estimate of the selected segment on the target machine, and also the communication time estimate given as a percentage of the execution time. In this way, the effect of different data partitioning strategies can be evaluated on any part of the node program.

If the selected program segment contains symbolic variables, for instance in the upper and lower bounds of a do loop, the user is prompted for the values or value ranges of these variables, since they are necessary for applying procedure ESTIMATE. Once the value of a symbolic variable is supplied, some limited constant propagation is automatically done so that all other occurances of that variable within the selected program segment also get the value. In addition, the user is also prompted for a branch probability when an if-then statement is encountered by procedure ESTIMATE. The user can optionally accept the default banch probability which is assumed to be between 0.25 and 0.75. The current implementation only handles structured control flow in the node program, and arbitary branching using goto is prohibited.

We used this prototype implementation to test the accuracy of the estimation technique for several problems. Figure 6 shows the results of the estimator when

```
procedure ESTIMATE (S, etime, ctime)

//Input: a node program statement S.
//Output: the execution time estimate (etime) and
//communication time estimate (ctime) for S on the
//target machine.

if (S is a simple statement) {
    for (each arithmetic operation $\alpha$ in S) {
        let $t(\alpha)$ be its avg. execution time
        specified in the perf.data file;
        etime += $t(\alpha)$;
    }
    for (each load/store operation $\lambda$ in S) {
        let $t(\lambda)$ be its avg. execution time
        specified in the perf.data file;
        etime += $t(\lambda)$;
    }
}
else if (S is a compound statement) {
    let B be the body of S;
    etemp = 0.0; ctemp = 0.0;
    for (each statement b $\in$ B) {
        ESTIMATE (b, etemp, ctemp);
    }
    if (S is a do loop) {
        etime += etemp * # of iterations of loop;
        ctime += ctemp * # of iterations of loop;
    }
    if (S is an if-then branch) {
        etime += etemp * probability of branch;
        ctime += ctemp * probability of branch;
    }
    if (S is a subroutine call) {
        let $\sigma$ be the cost of a subroutine call
        specified in the perf.data file;
        etime += etemp + $\sigma$;
        ctime += ctemp;
    }
    if (S is a communication call) {
        let $\delta$ be the message size in bytes;
        compute the time estimate $t(\delta)$ for this
        message using the method outlined in Sec. 5;
        ctime += $t(\delta)$;
        etime += $t(\delta)$;
    }
}
```

Figure 5: The performance estimation algorithm.

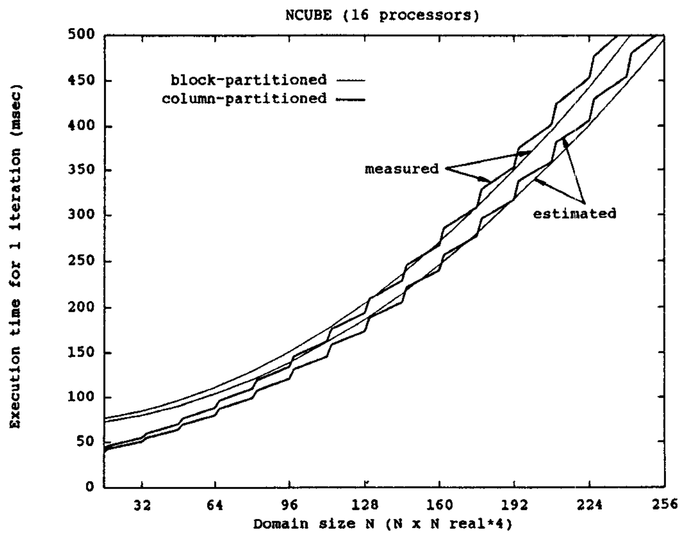
applied to the do k loop of program REDBLACK, compared to actual measurements on the target machine. The error in the estimation of execution and communication time is approximately 5-10%, but the "crossover point" of the block and column partition curves is predicted quite accurately. Note that besides indicating the crossover point at which one partitioning is preferable over another, the estimator also gives us a measure of the difference in performance between the two strategies. This may be very useful to the user in making several tradeoff decisions.

8 Conclusion and future work

We have proposed a method for discriminating between different data partitioning choices in a distributed memory parallel program. Our technique, the Training Set Method, is inspired by the observation that it is very hard to build a parameterized model that can help make a comparison of different data distribution strategies across a wide range of communication patterns, problem sizes and target machines. The Training Set Method bases its estimation not on a "hard-wired" theoretical model, but rather on a "training" mechanism that uses a carefully written program (called the training set) to train the model for a particular target machine. This training procedure needs to be done only once for each target machine, during the environment or compiler installation time.

Although the use of a training set simplifies the task of performance estimation significantly, its complexity now lies in the design of the training set program, which must be able to generate a variety of computation and data movement patterns to extract the effect on performance of the hardware/software characteristics of the target machine. Fortunately, real applications (especially the regular and spatially irregular ones) rarely show random data movment patterns; there is often an inherent regularity in their behavior. We would therefore conjecture that the training set program that we designed will probably give fairly accurate estimates for a large number of real applications, even though it tests only a small (regular) subset of all the possible communication patterns. We are planning to investigate how neural networks in conjunction with a test suite of programs and the training set can be used to identify computation or communication patterns where the performance estimation is imprecise and help adapt the performance estimator accordingly.

The estimator we have implemented works for node programs in which calls to the communication library have already been inserted. Hence, is not suitable for estimating the performance of a complete sequential program annotated with data partitioning information as described in the introduction [8, 10]. However, we



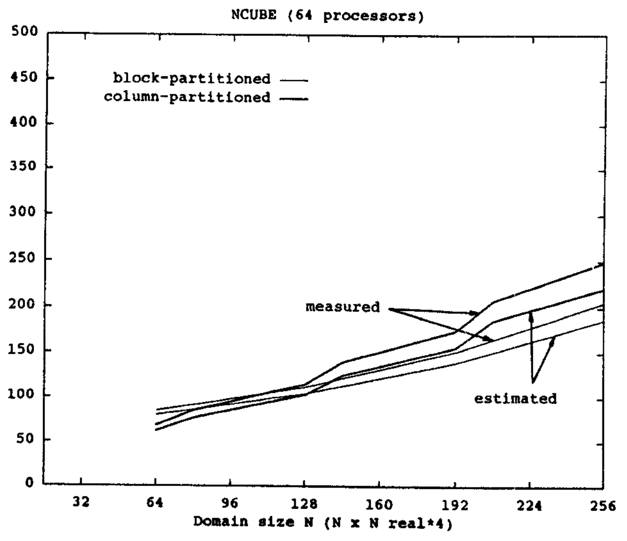


Figure 6: Measured and predicted execution times for column and block partitioned REDBLACK for the NCUBE.

believe the same basic strategy can be used for such an estimator if knowledge of how the compiler works is used. For this case, the algorithm of Figure 5 will need to be modified so it can automatically discern what communication is needed, instead of reading it from the program. A communication analysis algorithm (as described in our earlier work [3]) together with the results gained by applying the compiler to a set of kernel computations can accomplish this task. An appealing feature of this approach is that changes in the compiler will require only rerunning the training sets to initialize the tables for the new performance estimator, in the same way that changing the target machine requires rerunning the training set described in this paper.

For the longer term, we are exploring the possibility of automating the selection of good data partitioning schemes using the performance estimator to evaluate alternative strategies in a search through the space of reasonable partitionings [13]. Using such a scheme, we may be able to significantly reduce the effort required to implement scientific programs on distributed memory multiprocessors.

References

[1] M. Annaratone, C. Pommerell, and R. Rühl. Interprocessor communication and performance in distributed memory parallel processors. *Proceedings of the 16th Symposium on Computer Architecture, Jerusalem*, pages 315–324, May 1989.

[2] V. Balasundaram. A static performance estimator. Technical Report C3P-941, Caltech Concurrent Computation Program, Caltech, Pasadena, CA, August 1990.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. *Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, S. Carolina*, April 1990.

[4] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope editor: an interactive parallel programming tool. *Supercomputing 89, Reno, Nevada*, November 1989.

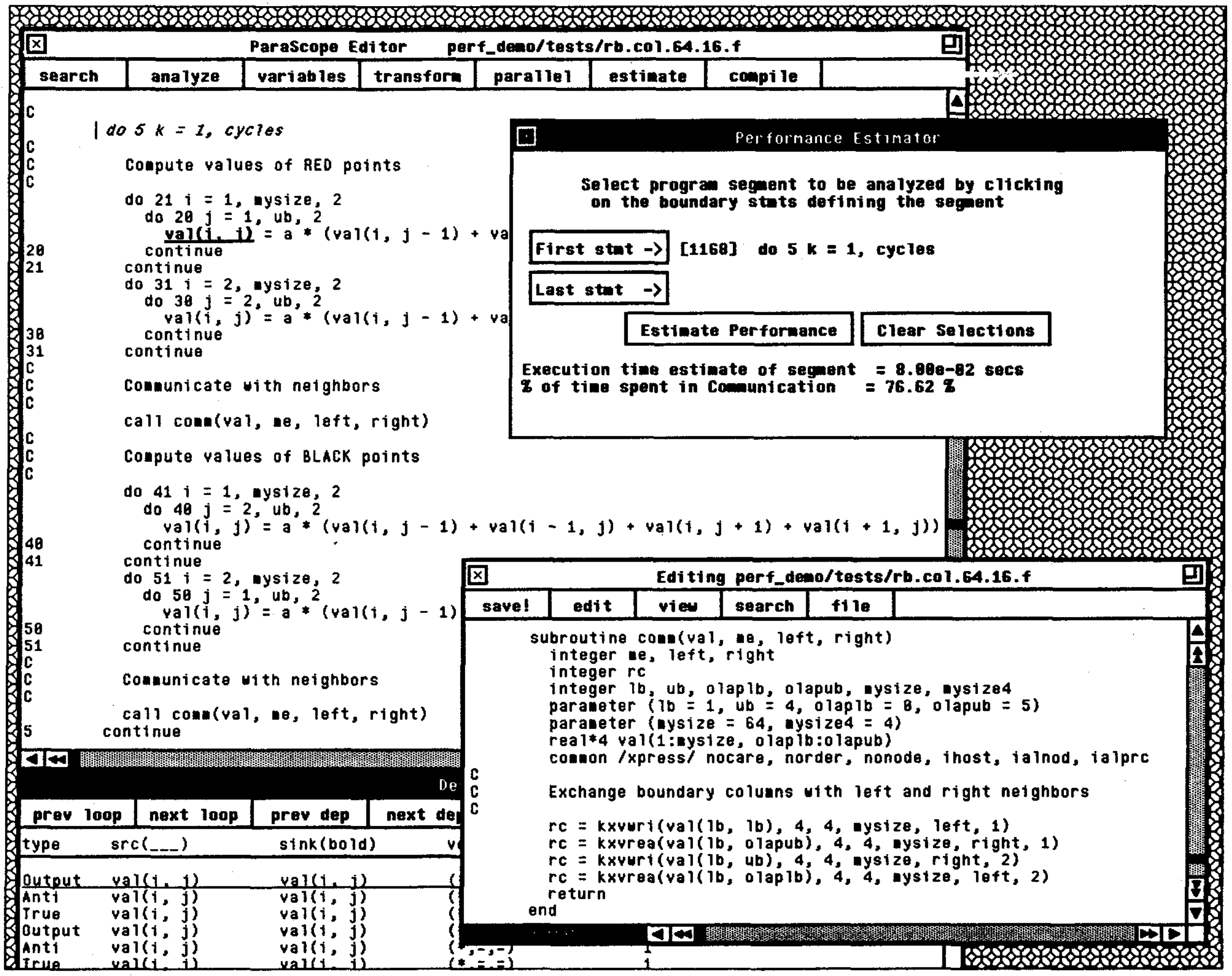
[5] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[6] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 2:171–207, 1988.

[7] ParaSoft Corporation. EXPRESS user manual. 1989.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report 90-141, Rice University, Houston, TX, December 1990.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on*



```
C
      | do 5 k = 1, cycles
C
C        Compute values of RED points
C
         do 21 i = 1, mysize, 2
           do 20 j = 1, ub, 2
             val(i, j) = a * (val(i, j - 1) + va
20         continue
21       continue
         do 31 i = 2, mysize, 2
           do 30 j = 2, ub, 2
             val(i, j) = a * (val(i, j - 1) + va
30         continue
31       continue
C
C        Communicate with neighbors
C
         call comm(val, me, left, right)
C
C        Compute values of BLACK points
C
         do 41 i = 1, mysize, 2
           do 40 j = 2, ub, 2
             val(i, j) = a * (val(i, j - 1) + val(i - 1, j) + val(i, j + 1) + val(i + 1, j))
40         continue
41       continue
         do 51 i = 2, mysize, 2
           do 50 j = 1, ub, 2
             val(i, j) = a * (val(i, j - 1)
50         continue
51       continue
C
C        Communicate with neighbors
C
         call comm(val, me, left, right)
5      continue
```

```
Select program segment to be analyzed by clicking
 on the boundary stmts defining the segment

[1168]  do 5 k = 1, cycles

Execution time estimate of segment  = 8.00e-02 secs
% of time spent in Communication   = 76.62 %
```

```
type     src(___)        sink(bold)
Output   val(i, j)       val(i, j)       (
Anti     val(i, j)       val(i, j)       (
True     val(i, j)       val(i, j)       (
Output   val(i, j)       val(i, j)       (
Anti     val(i, j)       val(i, j)       (*,-,-)
True     val(i, j)       val(i, j)       (*,=,=)
```

```
      subroutine comm(val, me, left, right)
        integer me, left, right
        integer rc
        integer lb, ub, olaplb, olapub, mysize, mysize4
        parameter (lb = 1, ub = 4, olaplb = 0, olapub = 5)
        parameter (mysize = 64, mysize4 = 4)
        real*4 val(1:mysize, olaplb:olapub)
        common /xpress/ nocare, norder, nonode, ihost, ialnod, ialprc
C
C       Exchange boundary columns with left and right neighbors
C
        rc = kxvwri(val(lb, lb), 4, 4, mysize, left, 1)
        rc = kxvrea(val(lb, olapub), 4, 4, mysize, right, 1)
        rc = kxvwri(val(lb, ub), 4, 4, mysize, right, 2)
        rc = kxvrea(val(lb, olaplb), 4, 4, mysize, left, 2)
        return
      end
```

Figure 7: Screen snapshot of the performance estimator in ParaScope.

Concurrent Processors, Vols. 1 and 2. Prentice Hall, Englewood Cliffs, NJ, 1988.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. *In Compilers and Runtime Software for Scalable Multiprocessors (J. Saltz and P. Mehrotra, eds.)*, to appear 1991.

[11] K. Ikudome, G. Fox, A. Kolawa, and J.W. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. *Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, S. Carolina*, April 1990.

[12] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, March 1990.

[13] U. Kremer. Automatic data partitioning and distribution for loosely synchronous problems in an interactive programming environment. Technical Report In preparation, Rice University, Houston, TX.

[14] J. Li and M. Chen. Generating explicit communication from shared memory program references. *Supercomputing 90, New York*, pages 865–877, Nov 1990.

[15] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[16] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical recipes in C: the art of scientific computing.* Cambridge University Press, 1988.

[17] J. Ramanujan and P. Sadayappan. A methodology for parallelizing programs for complex memory multiprocessors. *Supercomputing 89, Reno, Nevada*, November 1989.

[18] A. Rogers and K. Pingali. Process decomposition through locality of reference. *ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[19] R. Rühl and M. Annaratone. Parallelization of Fortran code on distributed memory parallel processors. *Proceedings of the ACM International Conference on Supercomputing*, 1990.

[20] V. Sarkar. Determining average program execution times and their variance. *Proceedings of the SIGPLAN 89 conference on programming language design and implementation*, pages 298–312, July 1989.

[21] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.