A Static Performance Estimator to Guide Data Partitioning Decisions

Vansanth Balasundaram
Geoffrey Fox
Ken Kennedy
Uli Kremer

CRPC-TR90093
October, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A Static Performance Estimator to Guide Data Partitioning Decisions

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

Abstract

A common approach to distributed memory parallel programming allows the programmer to explicitly specify the data decomposition via program annotations in his/her sequential or shared memory program. Given such an annotated program as input, the compiler generates all the necessary communication. While this frees the programmer from the tedium of thinking about message-passing, no assistance is provided in determining the data decomposition scheme that gives the best performance on the target machine. In this paper, we discuss performance estimation as part of an interactive software tool that provides assistance for this very task. We describe a new approach to performance estimation that is based on experiments rather than on a fixed machine model. Preliminary experiments indicate that the proposed techniques will work well for a large class of scientific programs.

Efficient partitioning of the data domain is crucial for good performance on distributed memory MIMD computers. Much work is being done in the area of compile-time analysis and deduction of communication from user provided data partitioning specifications [4, 5, 8, 11, 12, 15]. Given a sequential or parallel program and a data partitioning specifications as input, the communication routines are selected and inserted into a single node program that will be executed on each processor. This strategy is illustrated in step II and III in Figure 1.

While these techniques free the user from the tedious and error prone job of inserting communication, they do not support the user's decision on a good data layout. Our approach, described in [2], is to use an interactive data partitioning tool that allows the user to explore different partitioning strategies, as well as apply several performance improving program transformations where appropriate (step I in Figure 1). This allows the user to evaluate several options for any chosen program segment, eventually converging on the choice that is most suitable for the target machine. Thus the tool lets the user derive the data partitioning scheme for the whole program in relatively easier incremental steps, by analyzing a small program segment at a time.

The focus of our earlier work was the overall design of the data partitioning tool, the internal representation of data partition information, and algorithms for deducing communication once the partitioning was specified [2]. Although we alluded to the use of target machine specific parameters in helping the user "evaluate" the data partitioning choices at each step, we did not elaborate on the exact details of this procedure.

This paper discusses new techniques for static performance estimation in the context of an interactive tool that supports the user's choice of a good data partitioning strategy. It is important to note that our goal is *not* to provide the user with exact, absolute performance estimates, but with relative performance figures that support his/her tradeoff decisions between different partitioning schemes. We restrict our proposed tool to problems with arrays as their data structures. In particular, we do not handle representations of data objects as they occur, for instance, in sparse matrix and unstructured mesh problems ([14, 8]).

We will investigate how performance estimation can guide the user's decision on a data partitioning

*IBM T.J. Watson Research Ctr. e-mail: vasb@ibm.com

†Syracuse University. e-mail: gcf@npac.syr.edu.

‡Dept. of Computer Science, Rice University, Houston, TX 77251. e-mail: ken@rice.edu. Please address all correspondence regarding this paper to Ulrich Kremer. e-mail: kremer@rice.edu

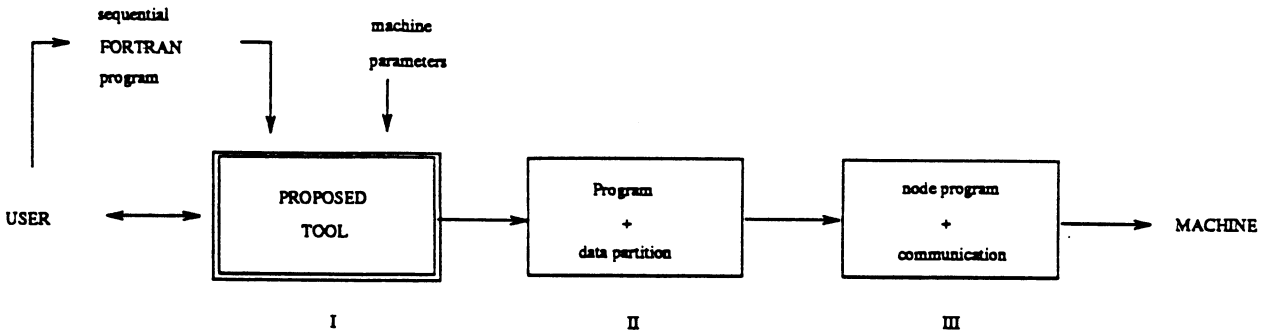

Figure 1: The program development process.

scheme for a given program segment. The discussed performance estimator is an integral part of the interactive tool as described in our earlier work. In such a system the estimator will be invoked *after* communication analysis has been performed by the tool. In order to be able to validate our techniques, we implemented a version of the performance estimator that takes a node program with explicit communication as its input (step III in Figure 1). The main goal of our preliminary prototype is not to use it as an independent tool, but as a testbed for our newly developed estimation techniques.

The prototype has been implemented as part of the ParaScope parallel programming environment [3]. Preliminary experiments indicate that the estimator will work well for a large class of scientific computations, called *loosely synchronous* problems [6]. Loosely synchronous problems can be characterized by computation intensive regions that have substantial parallelism, with communication required between the regions.

We believe that understanding how to treat this "well-structured" and common class of problems will give us insights into the general complexity of static performance estimation.

1 Choosing a data partitioning

The choice of a good data decomposition and distribution scheme that will be passed as annotations of a sequential program to a compiler of step II in Figure 1 is influenced by many factors. All these factors make it extremely difficult for a human to predict the behavior of a given data layout scheme without having to compile and run the program on the specific target system. These factors include:

- *compiler characteristics*, such as the communication analysis algorithm and the transformation system used, the set of communication primitives and routines that can be generated, and the strategy for performing static load balancing.
- *machine characteristics*, such as communication and computation costs, and machine topology.
- *problem characteristics*, such as actual problem size and number of processors to be used.

Using pointwise red-black relaxation as an example (see Figure 2), we will briefly examine the relationship between the listed factors, in particular the relationship between the program's data size and the number of processors actually used.

The graphs of Figure 3 show the execution times of a single iteration of the main loop in a program that performs pointwise red-black relaxation on 16 and 64 processors for different problem sizes. In the case of 16 processors the column partitioning scheme is more profitable than block partitioning for array sizes up to approximately 180. With 64 processors, block partitioning is the better choice for nearly all array sizes greater than approximately 72. The steps in the graphs are due to effects of load imbalance and the number of message packets needed by the communication utility used on the Ncube. Note that the program uses

```
      do k=1, cycles

c         Compute values of RED points

          do j=lb, ub, 2
               do i=lb, ub, 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=lb + 1, ub, 2
               do i=lb + 1, ub, 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

c         Communicate with neighbors

          call comm (val, me, ...)

c         Compute values of BLACK points

          do j=lb, ub, 2
               do i=lb + 1, ub, 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=lb + 1, ub, 2
               do i=lb, ub, 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

c         Communicate with neighbors

          call comm (val, me, ...)

      enddo
```

Figure 2: Pointwise red-black relaxation (node program).

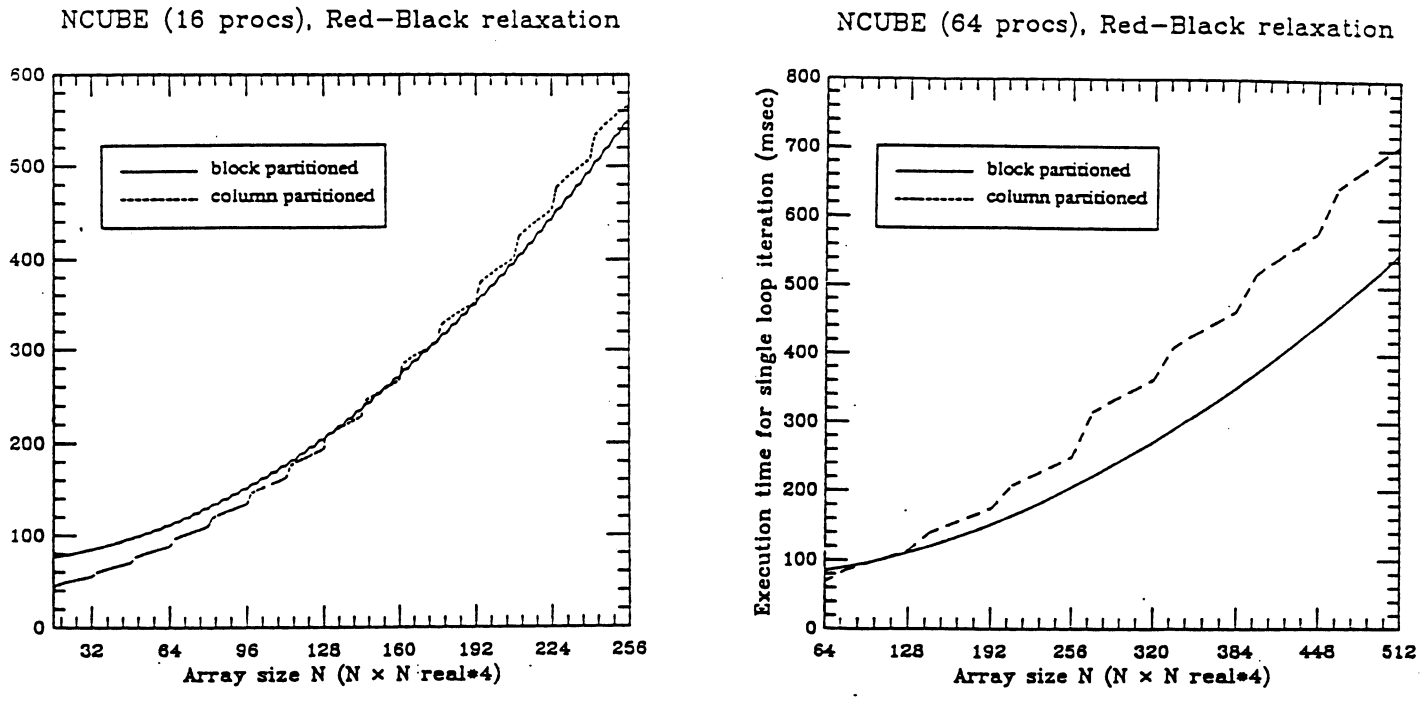

Figure 3: Pointwise red-black relaxation.

asynchronous, vector-send/vector-receive operations for communication, which will be described in more detail below.

This example illustrates the subtle complexities that must be taken into account in order to find the best data partitioning and distribution. In the remainder of this paper we will not be concerned with compiler characteristics since the performance estimator prototype takes a node program generated by the compilation process as input.

2 The training set method

Initially, we decided to use theoretical methods of estimation. The theoretical method consists of deriving a set of equations for computation and communication costs based on some abstract model of the target machine. We found that using such a "hard-wired" machine model had two major drawbacks:

- The difficulty of modeling the combined effects of hardware, operating system, and software layer on the performance of a program results in an extremely complex machine model. In order to reduce the complexity, simplifying assumptions about the interaction of these three layers have to be made leading to a significant loss of accuracy of the prediction.
- Any change in the hardware, operating system, or software layer forces a reevaluation of the used machine model, in the worst case leading to its total redesign. This inflexibility of the machine model places a huge burden on the system programmer responsible for maintaining the estimation program. Substantial intellectual effort is necessary to adapt the machine machine model in response to every change in the software or hardware layer.

We began experimenting with an alternative method of performance prediction, in an attempt to overcome some of the above problems. We wrote a set of routines that tested several communication utilities as

well as many different arithmetic operations and control flow structures. This set of routines, called the "training set" is currently written as a Cubix program, using Express communication utilities. Cubix is an environment that lets the programmer write the program for a single processor node, without specifying the host program. Express is a portable communications package for distributed memory parallel computers. It provides high-level extensions to C and Fortran to specify message passing between processor nodes. For a detailed description of these concepts, see [6, 9].

The training set routines were designed to test several possible combinations of communication calls, and a wide range of message sizes. The training set is part of the *Train Module*, that is used to "train" our performance estimation system. The Train Module is run *once* on the target machine, to generate a machine data file which contains actual computation and communication time measurements on the target machine. The file can be quite large (up to several Mbytes, depending on the accuracy desired in the measurement), especially since the communication calls are tested for a wide range of message sizes. The *Performance Initializer Module* provides routines to convert the machine data file into a compact representation. The resulting representation allows fast access of the timing data by the performance estimation system. The Performance Initializer Module is discussed in the next section.

Figure 4 shows an example of the kind of data collected by the Train Module. It is a graphical depiction of some of the communication performance data that is generated by the Train Module for a 64 processor NCUBE. The curves in the graph represent the communication time versus message size for the following Express communication utilities:

- `iSR`: nearest neighbor individual element send and receive, using the Express calls kxwrit and kxread.

```
kxwrit(buf, len, dest_pid, type)
kxread(buf, len, src_pid, type)
```

  `kxwrit` sends a message of `len` bytes from buffer `buf` to processor `dest_pid`, and `kxread` receives a message of `len` bytes from processor `src_id` into buffer `buf`.

- `vSR`: nearest neighbor vector send and receive along one direction, using the Express calls kxvwri and kxvrea.

```
kxvwri(buf, size, offset, nitems, dest, type)
kxvrea(buf, size, offset, nitems, src, type)
```

  `kxvwri` sends `nitems`, each of size `size` bytes from buffer `buf` to processor `dest`. Successive items of `buf` are taken to be `offset` bytes apart. `kxvrea` receives `nitems`, each of size `size` bytes from processor `src` into buffer `buf`. Successive items are entered into locations of `buf` that are `offset` bytes apart. The vector send/receive functions allow the user to transfer both contiguous and non-contiguous chunks of data between processors.

- `EXCH`: nearest neighbor vector exchange along one direction, using the Express call kxvcha.

```
kxvcha(ibuf,isize,ioff,iitems,isrc,itype, obuf,osize,ooff,oitems,odest,otyp e)
```

  `kxvcha` does a simultaneous send and receive of data between two processors. The handshaking between the processors is transparent to the user. The receiving of `iitems` of data each of size `isize` from processor `isrc` into `ibuf` is done simultaneously with the sending of `oitems` of data each of size `osize` from buffer `obuf` to processor `odest`. The vector exchange operation generally exhibits superior performance when compared to a combination of separate vector sends and receives.

- `BCAST`: one to all broadcast, using the Express call kxbrod.

```
kxbrod(buf, origin, nbytes, nnodes, nodel, type)
```

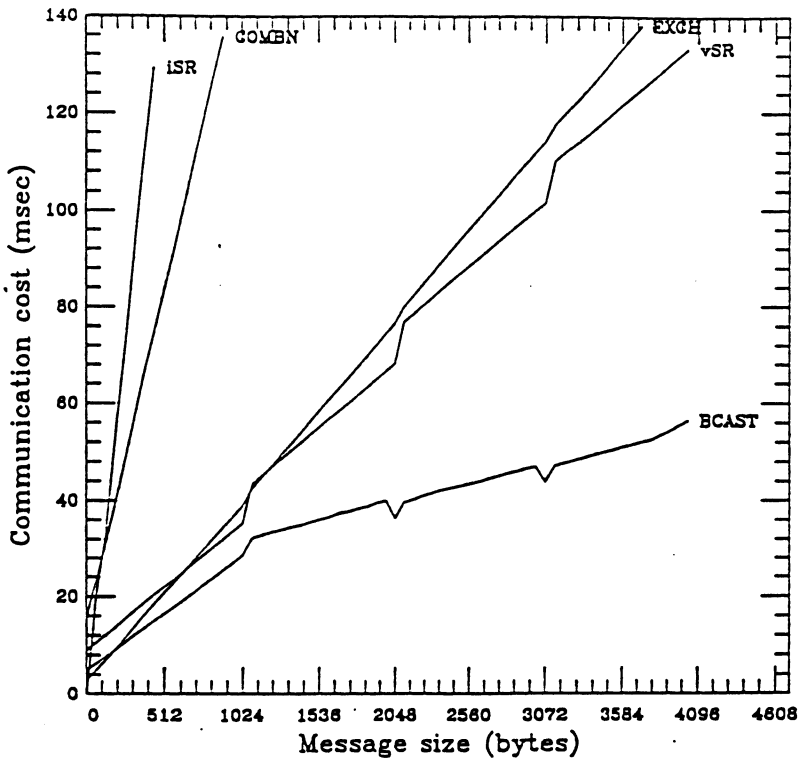

Figure 4: Graph of communication performance data for the Ncube.

kxbrod broadcasts **nbytes** of data from processor **origin** to all processors specified in the **nodel** array. **nnodes** is the number of processor ids specified in the **nodel** array. If **nnodes** is the special variable "ialnod", the broadcast is done to all other processors, and the value of **nodel** is ignored.

- **COMBN**: combine operation over all processors, using the Express call kxcomb.

  kxcomb(buf, func, size, nitems, nnodes, nodel, type)

  kxcomb determines the result of applying an associative and commutative operator **func** on data elements **buf** that are distributed over the set of processors specified in the **nodel** array. The result of this "combine" operation is stored locally within each processor in **buf**. An example of such an operation is global sum-reduction. This function implements the operation in time logarithmic in the number of participating processors.

The above communication utilities are just some of the many provided by Express. The training set routines are set up so that the above routines are tested in several combinations on the target machine for varying number of processors and message sizes. The performance data is collected by the Train Module as a set of points (x, y) where x is the message size in bytes and y is the average communication time in msecs.

The arithmetic/control and communication performance data is written into a **machine.data** file. There is also a routine provided in the Train Module, that creates a graph of this performance data and displays it graphically. The raw data file generated in this manner is then processed by a Performance Initializer Module.

2.1 The Performance Initializer Module

The raw performance data file **machine.data** is typically very large (a few to several Mbytes, depending on the accuracy of measurement desired by the user). Some processing of this data is needed before it can be

used within the interactive data partitioning tool. The performance initializer is only called once, after the `machine.data` file has been created by the Train Module.

The routines comprising the Performance Initializer Module read the `machine.data` file and try to represent the data internally as compact as possible. For instance, linear or piecewise linear functions can be represented very efficiently using a variant of the chi-squared fit algorithm [10]. Nonlinear functions can be approximated using methods based on neural networks [1, 13]. If no approximation technique matches the actual measured data, the function is represented internally as a table. The latter technique is the least accurate since not all data points can be stored in the table due to memory restrictions. While the approximation techniques based on neural networks are currently being investigated, the methods based on the chi-squared fit algorithm have been implemented in a performance estimator prototype. They are briefly described below.

A variant of the chi-squared fit algorithm is applied to fit a linear function of the form $y = a + bx$ onto the data. The values of a and b are then stored for each of the communication utilities tested by the Train Module. However, as Figure 4 shows, the graphs of the communication utilities are piecewise linear functions, with discontinuities occurring at the packet boundaries. The packet size on the NCUBE is 1024 bytes. These discontinuities are due to the packetization cost involved in gathering data to form the next message packet. Unfortunately, the chi-squared fit only works for continuous linear functions. To get around this caveat, the data for a given communication utility is separated into groups, where each group corresponds to the data for a single packet (i.e., one linear section of the graph). The chi-squared fit is then used to fit a function to the data corresponding to each packet. Thus, we will end up with a different function for each packet, and this is stored in the form of a linked list.

The processed data is written out into a `machine.data.fastinit` file, that is used by the Performance Estimator Module within the data partitioning tool. The size of the fastinit file is typically of the order of a few Kbytes. Compared to the raw data file, this is a significant compression, of both information and physical storage. The fastinit data is much easier to use and is also a lot faster to process.

2.2 The performance estimation algorithm

The performance estimation for a given program segment is done in a recursive manner. The statements of the program segment are visited in order, and their individual contributions to the overall execution and communication time are computed. The computation and communication cost of a program segment is determined as the sum of the communication and computation costs of its components.

If the visited statement is itself a compound statement, this procedure is repeated recursively on its component statements. If the visited statement is an assignment, the expression on the RHS is visited, and the arithmetic operations data from the fastinit file is used to compute the execution time of the RHS expression. If the visited statement is an Express communication call, the message size is determined and used to locate its corresponding data in fastinit file. The function parameters a and b are then used to compute the estimated communication time estimate. The execution time of the Express call is taken to be equal to its communication time estimate. If the visited statement is a non-Express function call or subroutine call, a performance estimate for the body of the function or subroutine is computed.

3 The performance estimator prototype

A prototype of the performance estimator has been implemented in the ParaScope parallel programming environment [3]. The estimator takes Cubix/Express Fortran [6, 9] with explicit calls to communication routines as its input. The initial implementation estimates the performance of the input node program based on the knowledge of the local data sizes, and the frequency and type of computations and communication operations. Since all processors execute the same node program, the estimated overall execution time of the whole program is assumed to be equal to the execution time estimate for a single node program. This assumption is true for the large class of loosely synchronous problems where computation and communication

phases alternate, i.e. do not overlap. However, synchronization effects that lead, for instance, to a wavefront behavior of the computation cannot be estimated by the current implementation. Since we assume that load imbalancing is handled statically by the compiler, the maximal data sizes in all node programs are known at the time the performance estimator is called. The execution time of the "maximal" node program will determine the execution time of the whole program.

Figure 5 shows a screen snapshot during a typical performance estimation session. The selection of the *<estimate>* button in the ParaScope main window invokes the performance estimation module which responds by opening a secondary window, labeled *Performance Estimator*. The user defines his/her program segment of interest by specifying the first and last statement of the segment in the secondary window. In the example session, the user wants to know the performance of the main loop in a program that performs pointwise red-black relaxation. After selecting the *<Estimate Performance>* button in the secondary window, the system replies with an execution time estimate for the chosen program segment together with the percentage of the overall execution time that was spent performing communication. The program segment contains two calls to the subroutine `comm`. The definition of the subroutine is shown in the secondary window below the *Performance Estimator* window.

If the selected program segment contains symbolic variables, for instance symbolic loop upper and lower bounds, the values or value ranges of these variables have to be made known to the tool, either by compile time analysis or by user input. In the current implementation, the performance estimator initiates a user dialog in order to get the necessary information for those variables whose values are unknown after performing constant propagation. In addition, a branch prediction dialog is activated if the program segment contains if-then-else branches. The default banch probability is assumed to be between 0.25 and 0.75. The current implementation does not handle arbitrary control flow.

4 Experimental evaluation

As mentioned above, the purpose of the discussed performance estimator is *not* to provide an exact, absolute performance estimate for a given program segment, but to allow the user to distinguish between different partitioning schemes for the segment. The *relative* performance estimates should reflect the quantitative performance differences of different partitioning strategies. Quantitative information is needed to allow the user to make tradeoff decisions. For example, the choice of whether to redistribute an array in a program segment or not depends on the costs for the redistribution relative to the execution time penalty incurred by using a "suboptimal" partitioning scheme instead of the "optimal" scheme.

Figure 6 shows the results of preliminary experiments based on the algorithm for pointwise red-black relaxation, using the asynchronous `kxvrea` and `kxvwri` communication utilities. The performance estimate is based on the actual local data sizes of the input node program. Therefore, in contrast to Figure 3, the estimated execution times of Figure 6 do not reflect the effects of static load balancing by the compiler. The measured execution time graphs of Figure 6 are derived from the ones of Figure 3 by considering only the load balanced data points.

We see that for both, block and column partitioning, the estimator underestimates the required time by approximately 5-10%. However, it predicts the crossover point at which the user should switch from one partitioning scheme to another very accurately, within approximately 10 elements of the array. The experiments show that the relative performance of the block and column partitioning strategies can be estimated with a high accuracy. Depending on the actual problem size and the number of processors used, the performance estimator not only tells the user the crossover point, but also by how much the performance of the two partitioning schemes will differ. This information is vital for tradeoff decisions between different partitiong strategies. For example, in the 16 processor case, choosing a block instead of a column partitioning for array sizes in the range of 16 to 80 will lead to a significant degradation in performance. For array sizes greater than 200, however, choosing the "inferior" column partitioning has only a small impact on the overall execution time.

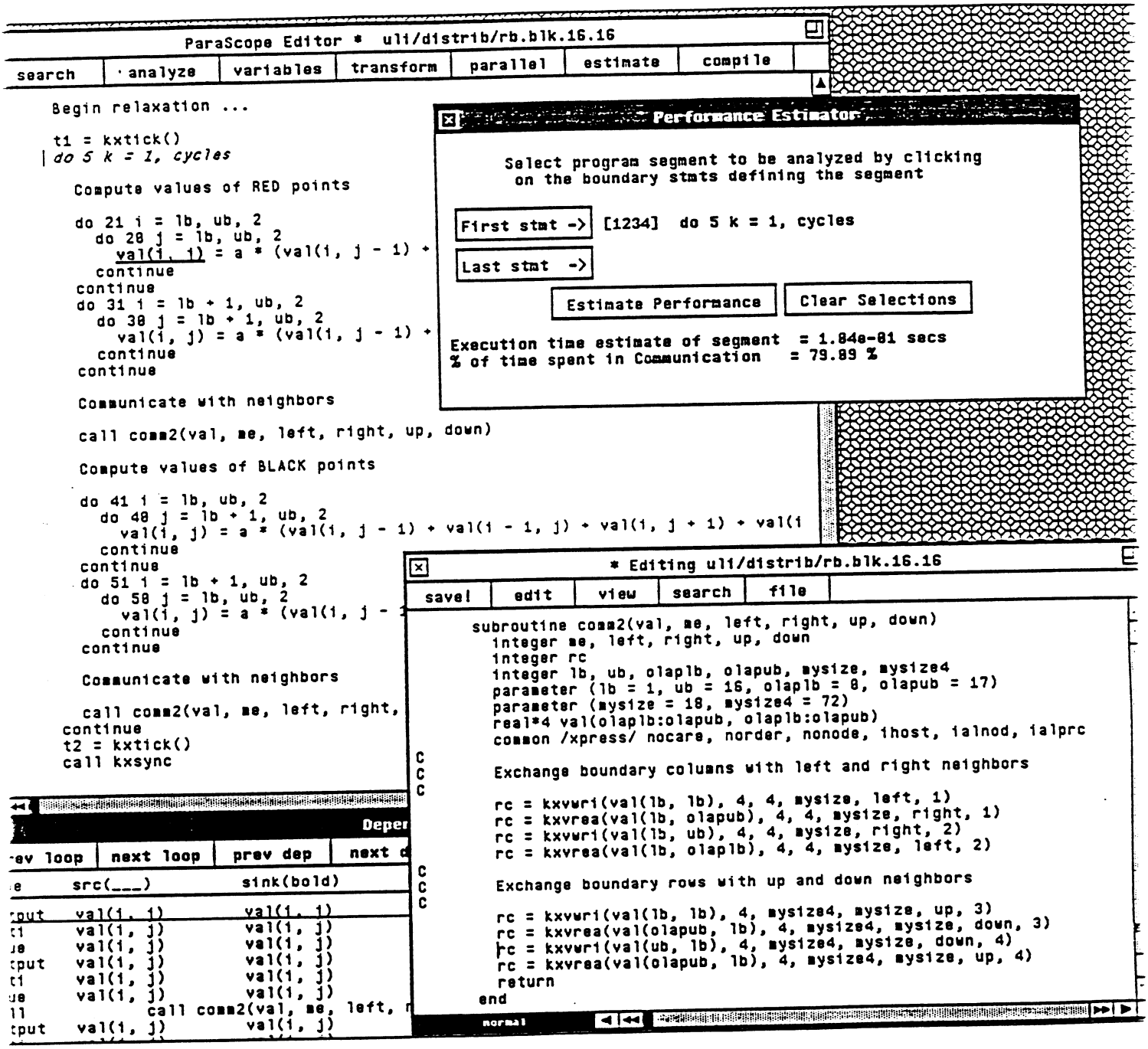

Figure 5: Screen snapshot.

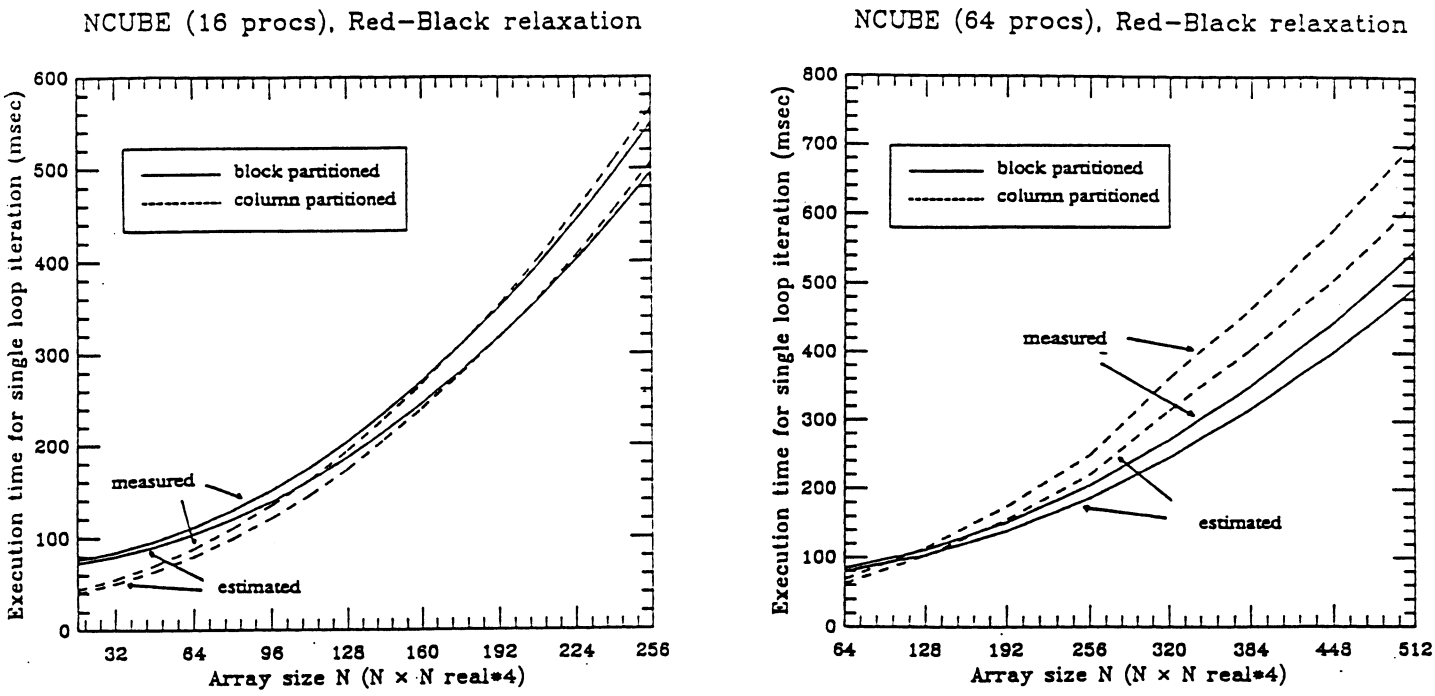

Figure 6: Measured and predicted execution times for pointwise red-black relaxation.

5 Conclusions and future work

Static performance estimation is able to provide information that can help the user to distinguish between different data partitioning schemes for a selected program segment. The proposed techniques for performance estimation are inspired by the observation that it is very hard to build a parameterized model that is able to predict the performance of a program across different machines and problem sizes. Instead of using such a "hard-wired" model, we choose to base our performance prediction on timing experiments for different communication patterns and computations. The timing experiments define a *training set* which needs to be executed only once at environment or compiler installation time. We also provide routines to represent the results in a compact fashion that allows fast access of the timing data by the performance estimator. Compaction methods that are based on neural networks are currently being investigated [1, 13].

Although the use of a training set reduces the complexity of performance estimation significantly, its complexity now lies in the design of the experiments which have to anticipate the effects of specific hardware/software characteristics on the execution time. We are planning to investigate how neural networks in conjunction with a carefully designed test suite of programs and the training set can be used to identify computation or communication patterns where the performance estimation is imprecise and to adapt the performance estimator accordingly.

We intend to integrate the performance estimator with the distributed memory compiler in an interactive advising tool for data partitioning in the near future. We believe that static performance estimates can be used to support automatic data partitioning and distribution. Research to determine the feasibility of automatic data partitioning and distribution is currently underway [7].

References

[1] Battiti. Multiscale methods, parallel computation and neural networks for computer vision. Ph.D. dissertation,

[2] 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, South Carolina*, April 1990.

[3] 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.

[4] D. Callahan and K. Kennedy. Compiling parallel programs for distributed-memory multiprocessors. *Journal of Supercomputing*, pp.151-169, October 1988.

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

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Vol.1.* Prentice Hall, Englewood Cliffs, NJ 07632, 1988.

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

[8] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. *Principles and Practice of Parallel Programming (PPOPP), Seattle, Washington*, March 1990.

[9] Parasoft Corporation. EXPRESS user's manual, 1989.

[10] W.H. Press. *Numerical Recipes in C: The Art of Scientific Computation*, Cambridge University Press.

[11] A. Rogers and K. Pingali. Process decomposition through locality of reference. *SIGPLAN 89 Conference on Programming Language Design and Implementation*, June 1989.

[12] M. Rosing, R.B. Schnabel, and R.P. Weaver. DINO: Summary and examples. *Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications*, 1988.

[13] D. Rumelhart and J. McClelland. *Parallel distributed processing: Explorations in the microstructure of cognition.* MIT Press, 1986.

[14] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8, pp303-312, 1990.

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