INDIANA UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 402

Analyzing Data-structure Movements in Message-Passing Programs

Sekhar R. Sarukkai
Recom Technologies,
MS 269-3, NASA Ames Research Center,
Moffett Field, CA 94035-1000

email: sekhar@kronos.arc.nasa.gov
phone: (415) - 604 - 4242

Jacob K. Gotwals
Dept. of Computer Science,
Indiana University,
Bloomington, IN - 47405

email: jgotwals@moose.cs.indiana.edu
phone: (812) - 855 - 9761

March 1994

Abstract

In this paper we show that the analysis of interprocessor data movement in terms of source-level data structures can be effective in performance debugging. We present a method for the low overhead run time monitoring of interprocessor communication in terms of data structures. We show how performance indices based on postmortem analysis of the collected trace data can guide the user directly to the causes of poor performance.

One of the most important decisions a programmer has to make in writing parallel programs is with regard to data structure distributions and alignments. Even so, there are very few performance tools which attempt to provide statistics or views of programs in terms of the data structure interactions resulting from those alignments. Current tools for message passing programs provide mechanisms for studying performance from the processor and function perspectives only. We demonstrate that our approach, based on postmortem analysis of trace files augmented with data structure information, offers a rich set of performance indices and views that can be used to debug the performance of parallel programs.

1 Introduction

Theoretically, current and future generations of distributed memory, massively parallel multicomputers can provide the high level of performance required to solve the grand challenge computational science problems. However, harnessing the power of these machines has proven elusive and successes have been laborious. This can be attributed to the programming paradigm involved (i.e. explicit message passing) compounded by the lack of useful performance tools to aid in the process of developing efficient parallel programs efficiently.

Recently there have been significant efforts toward developing parallel programming paradigms supporting higher level abstractions of parallelism, such as in HPF [8] and pC++ [9]. These languages provide means for explicit specification of data alignments and distributions. Though this approach largely hides explicit message communication from the programmer, the programmer still makes the important decisions on how to distribute and align the various data structures to be operated on. These decisions play a critical role in determining the nature and amount of communication performed during program execution. To study the impact of these decisions on the program executions, performance tools that highlight the cost of data structure interactions are essential.

Thus, if any sort of performance tool is to be useful in the context of either explicit message passing programs or for higher-level languages such as HPF, it has to address the issue of isolating the performance of individual data structures and contextual data structure interactions. In this paper we address this issue for explicit message passing programs, while the approach can be extended for monitoring the movement of data in languages such as HPF as well.

In recent years there have been a number of efforts in providing software tools for debugging the performance of programs. These tools can broadly be classified into:

- Performance visualization tools such as [15, 13, 16, 7].
- Performance tools centered on metrics, such as [11, 6, 1].
- Expert systems for performance debugging [17].

In the next section we discuss some tools which are centered on metrics. Then with the help of a number of examples, we show how we can complement traditional performance indices centered on functions and processes, with a set of data structure indices that can be used to systematically guide the programmer towards performance bottlenecks.

1.1 Related Work

Quartz [1] is a tool for tuning parallel program performance on shared memory multiprocessors. The principle metric of Quartz is the *normalized processor time*. This is the total processor time spent in each section of the code divided by the number of other processors which are concurrently busy when that section of the code is being executed.

IPS-2 [11] is a parallel performance tool which provides a set of system and application based metrics, by which program performance is debugged. Profile tables similar to the type of information presented by the standard UNIX profiling tool Gprof[6] are provided. Critical path analysis is based on identifying the path through the program's execution that consumed the most time. The profile table lists actual amounts of time spent in selected phases of the computation and communication as well. The above metrics support isolation of a problem with respect to procedures, phases and processors.

The most significant difference between the above tools and the one described in this paper has to do with the tracking of interprocessor data movement. Previous tools present performance information mainly in terms of processors and functions. But we have found that some of the most useful and intuitive information for tuning parallel program performance is information presented in terms of source level data structures. To our knowledge, our tool and methodology is the first to allow the efficient tracking of interprocessor data movement in terms of source level data structures in distributed memory parallel programs. This information on data movement is used to help the user determine the locations and causes of performance bottlenecks.

Further, once poorly performing sections of code have been identified, most of the above tools do not provide any feedback to the user about the possible *causes* of poor performance within those sections. This is a hard problem for distributed memory machines with message passing, since there are a number of possible causes of poor performance. We provide a comprehensive set of performance indices that highlight the significance of many of the possible causes of poor performance.

In this paper, we introduce a performance debugging methodology from a data structure perspective. This methodology helps in isolating data structures which have the most significant interprocessor communication times. Our methodology identifies a data structure *pair* for each communication: the data structure being communicated, and the data structure using the communicated data. We highlight performance problems in terms of these data structure pairs, with the help of a comprehensive set of performance indices.

The concept of using data structure oriented views was incorporated into the MemSpy system [10]. This tool provided a means of studying performance with respect to individual data structures, based on tracking individual data structure references for sequential and shared memory systems. The statistics were primarily geared to determine the hit ratio of data structure references in cache and to provide a possible explanation for the same. Since this tool needs information regarding cache hits and misses, it was built on top of the Tango-lite simulator [4]. Our work differs from this work in a number of significant ways.

Firstly, we do not track every data use. Instead we track those which are potentially the most expensive: interprocessor data references, involving movement of data between processors. Further, with each such interprocessor data movement, we associate a sending (u) and using (v) data structure. This corresponds to communication of parts of data structure u, required to update the values of parts of data structure v not local to the sending processor. For message passing programs, determining these data structures involves the use of static flow analysis.[1]

Secondly, for a tool to be applicable to real problems (and for users to want to use it), it must run without significant time and/or space overheads. To meet that requirement, our approach is based on trace collection, rather than simulation. The traces are obtained by executions of the program directly on the target multiprocessor.

In keeping with the need for low penalty for tracking data movements, we have designed techniques whereby the overheads associated with tracking data structures are not significantly larger than overheads for generating traces without data structure information, as shown in section 3. In the programs we tested, the entire monitoring process (involving the monitoring of data structure information along with the monitoring of more standard information) had a *maximum* overhead of about 30% of the parallel execution time. In contrast, simulations of parallel programs are much more expensive, even without monitoring, and take orders of magnitude more time than actual parallel execution times.

Finally, data should be presented in a manner that enables programmers to readily detect the *cause* of poor performance. We present performance data in terms of indices which highlight common performance problems encountered in message passing programs, associated with interprocessor communication. This allows the user to easily determine the cause of performance problems associated with the distribution and alignment of data, and to develop more scalable parallel programs.

[1]Performance optimizations on parts of the code that do not involve interprocessor communication are also possible. These optimizations may also be assisted by the use of performance tools. However, we do not address that issue here. Instead, we concentrate on optimizations involving interprocessor communication. Once interprocessor communication has been optimized, sequential performance tools are available to optimize local computations.

2 A Methodology for Automatically Tracking Data Movement

Our approach to tracking data structure movements is built on top of methodologies and tools already available for tracking message movements, in distributed memory programs.

Typically there are 3 phases to be followed in trace based analysis of programs. First, the program is instrumented by inserting calls to appropriate monitoring routines at important locations in the program. Second, during the execution of the instrumented program, a monitor enables the collection of records which signify the type and time of the occurrence of events, into a trace file. Finally, postmortem tools to display visualizations and statistics are used to comprehend and debug the program's performance.

Figure 1 summarizes the steps involved in our method. Before compilation, a program restructuring tool (the instrumenter) is used to transform communication system calls into calls to the monitor library. Input from a flow analysis tool helps determine the data structures that are "important" in terms of interprocessor communication. Those data structures appear as parameters to the monitor library calls. At execution time, the monitor makes the intended communication calls, and generates a trace file, augmented with codes for the important data structures. The monitor also outputs a lookup table relating the data structure codes to the actual data structure names in the source code. Finally, postmortem analysis tools use the trace file and the lookup table to present statistics and views of the execution. The data structure information allows the tools to display information in terms of data structure interactions, as well as in more traditional formats.

In our implementation, we restrict our focus to arrays in SPMD message passing FORTRAN programs, on distributed memory multiprocessors. We consider only simple, point to point transfers of data. All monitored global operations (such as broadcast) are treated as a sequence of point to point communications.

Our model of interprocessor communication is as follows. Communication is required when a portion of an array (the *source array*) on one node is needed for a computation on another node. The portion of the source array to be sent may have to be copied into a temporary array (a "send buffer") to put it into contiguous memory before sending. The final destination for the data is a region of some array, the *destination array*, on the receiving node. If the receiving region of the destination array is a contiguous block of memory, then the data can be received directly into the destination array; otherwise, the data is received into a temporary "receive buffer", then copied into the *destination array*. The destination array may then be used to compute values for another array, the *using array*, on the receiving node.

In this section we will present the approach we follow at compile and execution time in tracking interprocessor data structure interactions. Our goal is to automatically determine the identity of a pair of arrays for each communication, the sending and using arrays, and to make that array pair information available to postmortem performance analysis tools.

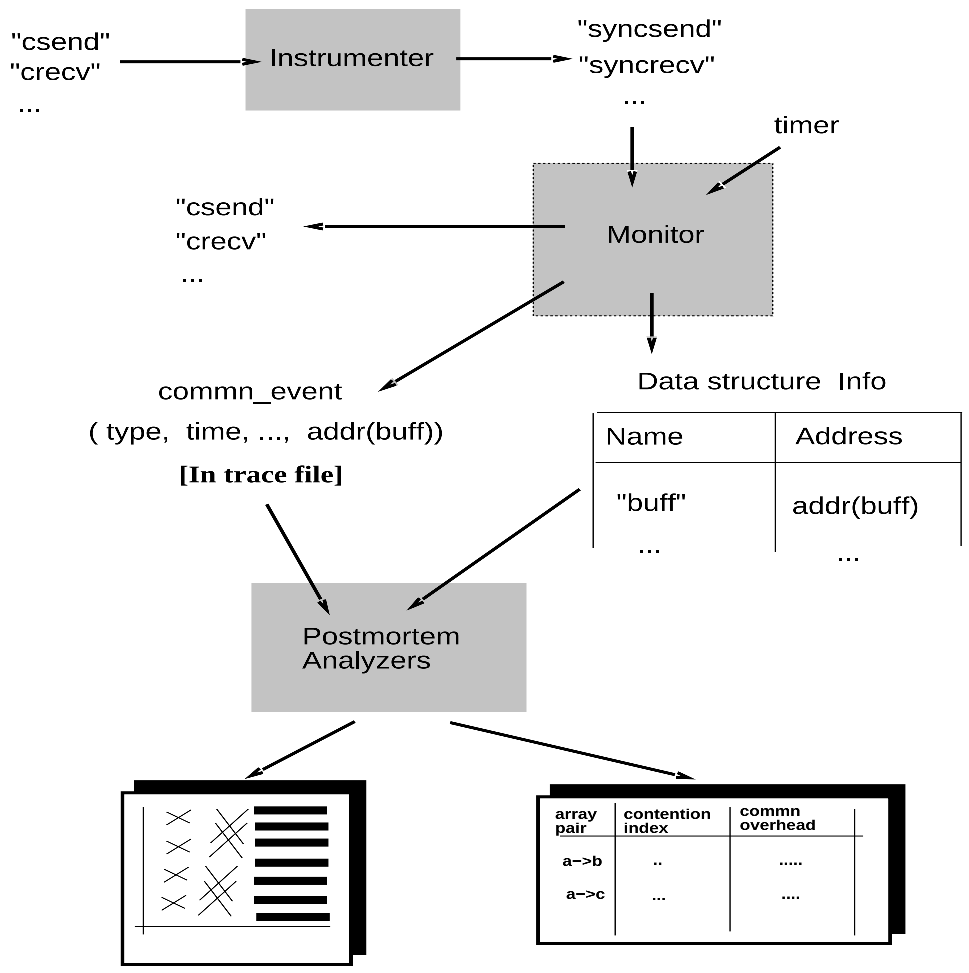


Figure 1: A schematic of the data structure analysis system, showing interactions between the various components.

2.1 Version 1: No Static Analysis

Our instrumenter implementation is built using the Sigma system [5]. Sigma parses the user's source code, building an internal representation and performing data flow analysis. We make calls to the Sigma library routines, to analyze and instrument the user's code, and to output the instrumented version of the program.

Consider a segment of a simple *S*ingle *P*rogram *M*ultiple *D*ata (*SPMD*) program whose data structure movement has to be tracked. For simplicity, assume that the only important data structures are arrays (as in F77). Figure 2 shows a segment of code where a part of array A is copied to a buffer $buff$ which is then sent to another processor (using a call to *csend*), where it is received into a different part of buffer $buff$, which eventually gets copied into array B.

The first version of the algorithm that we follow during instrumentation, to track the data structure associated with each communication call, is as follows (see Figure 1):

- Replace the original communication call with a call to a monitor library routine.

- Detect the data-structure being communicated (which is a specific parameter to the original communication call) and pass its name (as a string) and base address as additional parameters to the library call.
- At run time, the monitor performs three major functions:
    1. Obtains timestamps and stores communication event in local memory, which is periodically flushed to the trace file. (Communication events have fields which signify the data-structure being communicated),
    2. perform the actual communication call, and
    3. output a data-structure look-up table, containing data-structure names and starting addresses.
- At analysis time, interpret the trace file, using the data structure lookup table to relate the data structure addresses in the trace file to data structure names from the source code.

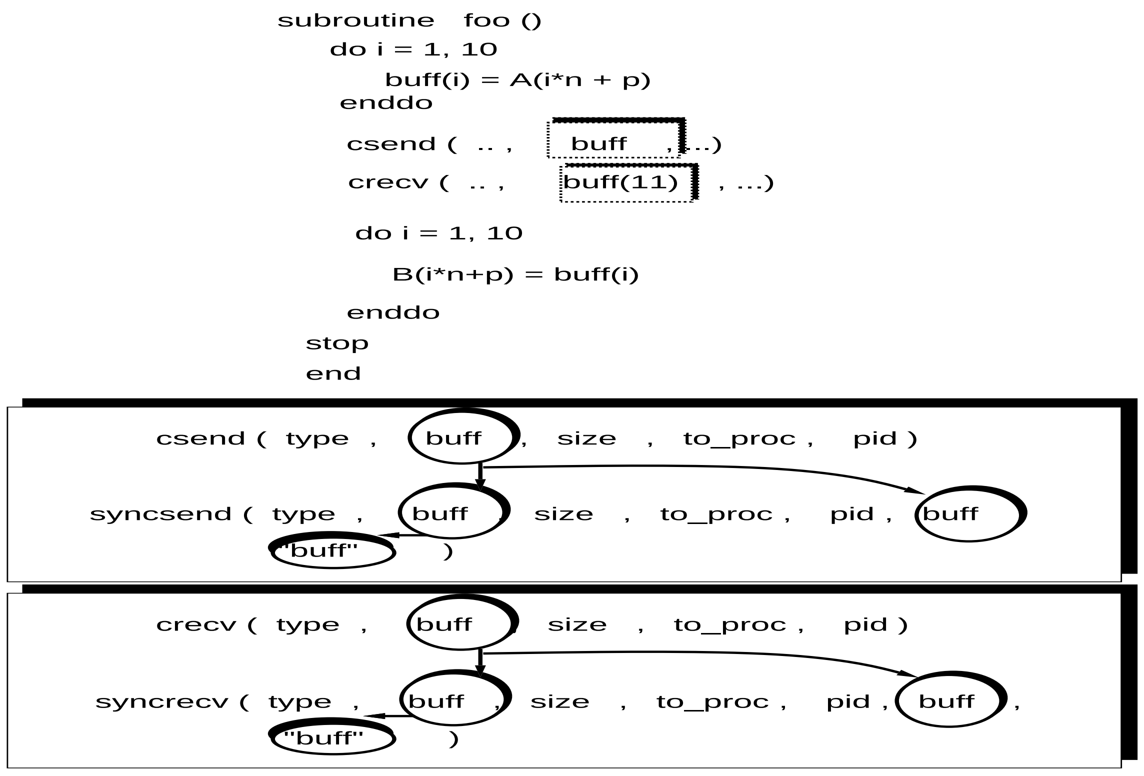
In Figure 2 we show the result of the transformation to the communication calls after instrumentation. The instrumenter replaces the call to the communication routine by a call to a library routine with two additional parameters: the name of the data structure and the starting address of the data structure. For example, the *csend* is replaced by *syncsend* with two additional fields.

2.2 Version 2: Avoiding Problems with Buffers

The simple implementation discussed in the previous section, does not resolve the issue of the use of communication buffer arrays. If temporary buffers are used in the communication process, it is likely that the names of the real data structures being communicated will not show up in the final statistics; the names of the temporary buffers will show up, instead. We would much prefer to have data structure interactions labeled with the names of the actual data structures that are communicated. Furthermore, if buffers are reused, then separate data structures may be labeled with the same buffer name; then when performance information is tabulated by data structure name in the postmortem analysis, this may lead to the presentation of misleading information. For these reasons, the previous version will not suffice.

So for example, in the example considered in Figure 2, the statistics displayed by the analysis tools will only show communications from $buff$ to $buff$, when in reality, it was array A which was communicated to array B. To detect such situations automatically, we need to make use of static program flow analysis.

To detect the case where the actual data structure is copied into a buffer, we use the following heuristic: if the closest reaching definition of the buffer is a simple copy within a loop (as in the example in figure 3), then the array being communicated is considered to be a buffer, and the array



```
subroutine   foo ()
    do i = 1, 10
         buff(i) = A(i*n + p)
     enddo

     csend ( .. ,     buff     ,...)
     crecv ( .. ,     buff(11)   , ...)

      do i = 1, 10

         B(i*n+p) = buff(i)

     enddo
  stop
  end
```

```
      csend ( type ,  buff ,  size  ,  to_proc ,   pid )

  syncsend ( type ,  buff    size   ,  to_proc ,   pid ,  buff
          "buff"          )

      crecv ( type ,  buff    size   ,  to_proc ,   pid )

  syncrecv ( type ,  buff ,  size   ,  to_proc ,   pid ,  buff  ,
          "buff"          )
```

Figure 2: Transformation of communication calls during instrumentation. Top: the original program; Bottom: the communication call is replaced with a call to a monitor library routine with two additional fields.

from which the data is copied is the one we associate with the communication. A similar process is used on the receiving end; if there is a reaching use of the received array which is a simple copy within a loop, then the receive is associated with the array to which the data is copied, rather than the array in which the message is received. Thus, for the case considered in Figure 3, the send is associated with the array A and the receive is associated with array B.

With the above technique, postmortem analyzers can more accurately segregate data structure interactions involving reused buffers, than in the previous case. In our implementation, the flow analysis required is carried out by the Sigma library.

2.3 Version 3: Tracking *Use* of Data Structures

Typically, data structure values are communicated to enable new values of a (possibly different) data structure to be updated, using the communicated values. Consider figure 4. The program on the left is the original, and the one on the right is the instrumented version. In this case, array A is communicated so that parts of array C can be updated, using the communicated values. In order for the information about the use of the communicated data to be captured in the trace file for postmortem analysis, we need to perform static flow analysis to determine where the communicated data structure is being used. So,

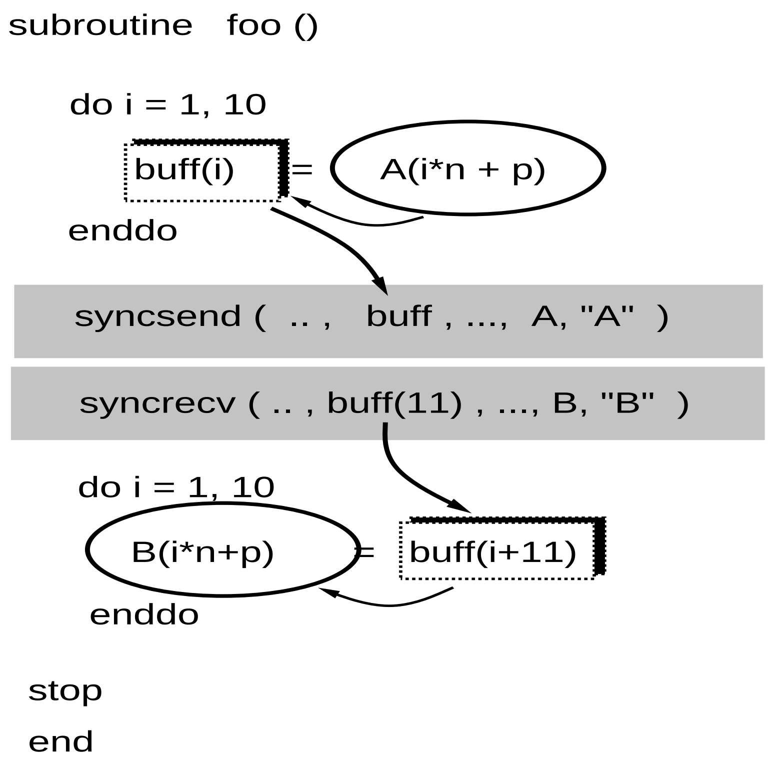


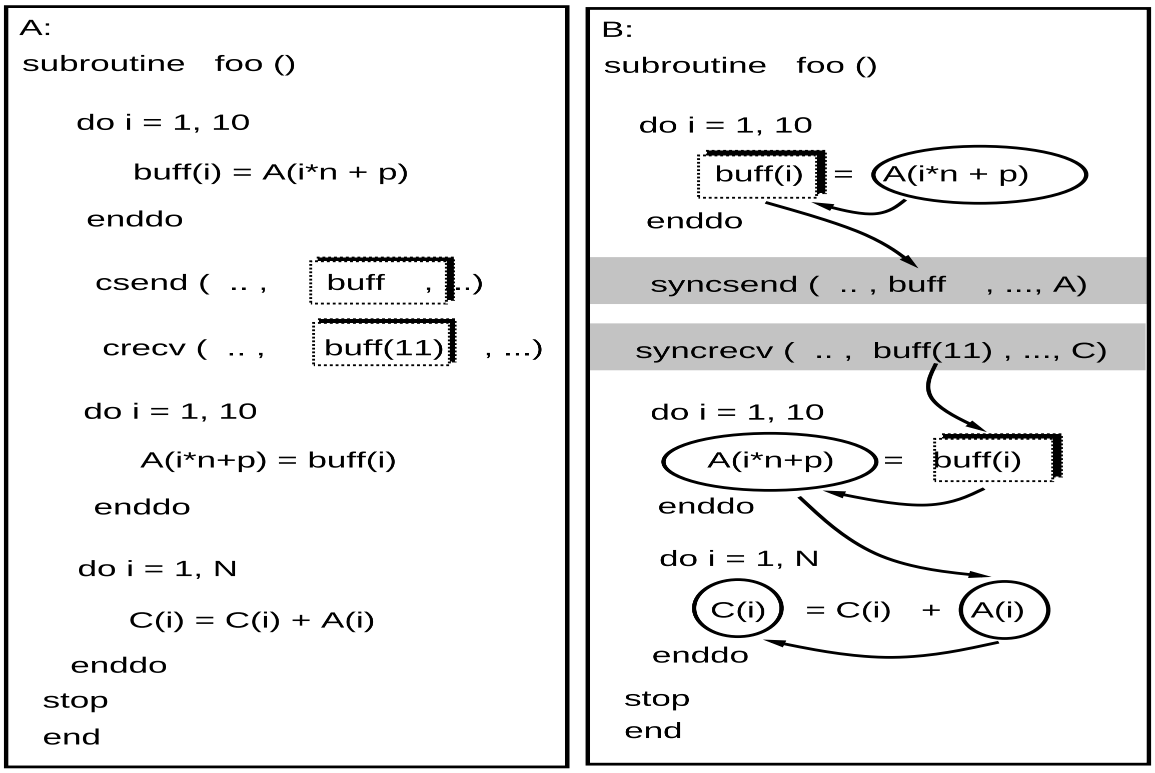
Figure 3: Communication calls with buffer copies, after instrumentation. *Actual* data communicated from A to B.

we extend the analysis discussed so far to determine the next use of the actual receiving data structure. Figure 4 illustrates the program dependence information as arcs. Arcs are drawn between the locations of definitions of data structures and corresponding statements which they reach. These dependence arcs are traversed by the instrumenter in determining the actual data structure being communicated.

2.4 Version 4: Parameters to Functions

We would like to have a single name for each region of array storage in memory. A difficulty is that arrays can be (and generally are) renamed when they are passed into subroutines as parameters. So we have adopted the following array naming convention: arrays are named by the identifier given them at the original point of declaration. If the array is not originally declared in the main program, then its name is prefixed by the name of the subroutine where it is originally declared, followed by a period. So an array x originally declared in the main program would be named x. An array x originally declared in a subroutine f would be named $f.x$. If that array is passed into a subroutine g (from subroutine f) as formal parameter y, then we will still call that array $f.x$, because it was originally declared in subroutine f.

The implementation of this approach, for static arrays in FORTRAN, is actually a fairly straightforward extension of the implementation discussed so far. Instead of passing the array names with every



```
A:
subroutine   foo ()

     do i = 1, 10

          buff(i) = A(i*n + p)

      enddo

       csend (  .. ,      buff    , ..)

        crecv (  .. ,      buff(11)    , ...)

      do i = 1, 10

           A(i*n+p) = buff(i)

       enddo

     do i = 1, N

          C(i) = C(i) + A(i)

     enddo
  stop
  end
```

```
B:
subroutine   foo ()

   do i = 1, 10

       buff(i) = A(i*n + p)

    enddo

     syncsend (  .. , buff    , ..., A)

    syncrecv (  .. ,  buff(11) , ..., C)

     do i = 1, 10

       A(i*n+p) = buff(i)

     enddo

     do i = 1, N

       C(i)  = C(i)   +  A(i)

     enddo

  stop
  end
```

Figure 4: "Use" of communicated data. Left: Original program; Right: Instrumented program. Arcs show that the communicated data comes from A, and is used by C.

communication call, pass only the base addresses, and introduce a new monitor call to define the array names in terms of the base addresses. The parameters to that call are the name and address of the array to be defined. The instrumenter should insert a call to this routine for each array name encountered in the program, immediately following the declaration of that name. At run time, that monitor routine simply accumulates the name/address pairs into a list, appending a pair to the end of the list with each call. That list then becomes the array name/address table output by the monitor. In the trace file, arrays are coded by their address. At analysis time, when a array address is encountered in the trace file, it is translated to a name by simply finding the first matching address in the name/address table. The name corresponding to that first address is guaranteed to be the original name of the array, before being passed to any subroutine, since arrays must be declared before being passed into subroutines (hence the name as originally declared would appear earlier in the name/address list).

2.5 Additional Features

In some cases, the receive is performed directly into the receiving data structure with no intervening buffer. Handling this situation is not difficult, but cannot be discussed in this paper due to space limitations. Our implementation is robust enough to detect these cases.

Situations can arise that cannot currently be handled by the heuristic we use for identifying the important data structures. The analysis performed by our implementation currently does not cross function boundaries, so we can identify buffer copies only when they are performed in the same function as the corresponding communication calls. [2]

3 Performance of Our Implementation

Using our approach, existing postmortem performance analyzers can be extended to track interprocessor data structure references and uses, with a very low additional overhead in storage requirements and execution time.

Existing postmortem performance analyzers such as AIMS [16] instrument the source code of the program being analyzed, replacing communication system calls with calls to a monitor library. The monitor library subroutines perform the intended communication calls, appending timing and other performance information onto a trace file. Other calls to the monitor library record information such as times of subroutine entrances and exits.

Using our approach, data structure information can be added to existing postmortem performance analyzers, with a storage overhead of just two integer fields per communication-related trace record. At monitor time, in addition to the typical overheads due to monitoring, we incur overhead in maintaining the data structure table. Table 1 shows execution times for several applications, instrumented several different ways. The applications are a number of different versions of a block tridiagonal solver, and one of the NAS parallel benchmarks. Execution times are given for each application (in msec) without instrumentation and monitoring, with normal AIMS instrumentation and monitoring, and with the additional instrumentation and monitoring that implement our data structure tracking method. Each of the executions is for a single time step or iteration of the program. Procedure begins and ends and all communication events are monitored in the instrumented runs. The table shows that the time overhead for adding data structure information to existing monitoring systems is less than 4%, and the time overhead for monitoring itself is less than 28%, for all the cases presented.

Table 2 shows the difference in ASCII trace file size for the instrumented applications discussed above, in terms of number of fields (counted by the standard *unix* utility '*wc*'). This table illustrates that the percentage increase in size for data structure tracing is not a function of trace file size. Rather, it is a function of the percentage of communication events in the trace file. In all cases in the table, the trace file size overhead for data structure tracing is less than 12%.

The low overhead of our approach is made possible by the fact that we have singled out interprocessor data structure references for attention. This approach is reasonable, since interprocessor data references are several orders of magnitude slower than local references, and thus account for significant

[2] Note however that our method is not inherently limited to single procedure analysis; interprocedural analysis is just more complex to handle.

	XT1	XT2	XT3	XT4	XT5	XT 7
Un-instrumented	149	55	143	54	154	58
Instrumented w/o D-str	180	64	182	61	158	60
% Instrumenting overhead	21	16	27	13	2.5	3.4
Instrumented w D-str	185	65	188	62	159	62
% D-str overhead	2.7	1.6	3.3	1.6	.6	3.3

Table 1: Execution times in msec for various versions of a tridiagonal solver (run on 16 nodes for a problem size of 256×256 for one iteration).

	XT1	XT2	XT3	XT4	XT5	XT7
Instrumented w/o D-str	263624	35144	246298	33050	32456	13498
Instrumented w D-str	294380	39020	275014	36678	35052	13796
% difference	11.6	11.0	11.6	10.9	8.0	2.2

Table 2: Trace file size overhead for tracing data structure information, for various versions of a tridiagonal solver (run on 16 nodes for a problem size of 256×256 for one iteration).

performance degradation. Tracking local data structure references (such as in Mtool [10]) is another possible approach, which can yield information on local data structure performance (e.g. cache interactions between data structures), at the expense of significant increases in execution time (for simulation based analyzers), or trace file size (for postmortem analyzers).

All the above results are for relatively short executions, with the effect that there is no significant perturbation of the program execution. As the number of events collected increases with increasing run time, run-time perturbation may become more significant. However, recently techniques have been established to eliminate or reduce the effect of flushes and monitor overheads (and hence perturbations) for *SPMD* programs, as discussed in [12, 14]. We have incorporated this methodology for perturbation compensation into AIMS, but since it is beyond the scope of this paper we do not discuss it any further.

4 Data structure Oriented Statistics and Views

The trace data collected on typical executions of a program is very large, and hence cannot be meaningfully understood by manually looking through the mire of numbers. Instead, tools are often used to present statistics of the program as well as to graphically depict program executions, based on the trace data. In this section we will describe some useful statistics that can be generated using the trace data. In the next section we will consider some graphical representations.

Statistics such as the communication and computation times can be tabulated by functions and processors. However, these *raw* numbers do not provide a uniform platform for comparing the performance of different programs, or of different versions of the same program. To circumvent this problem, a number of indices have been proposed, such as the communication/computation ratio, the normalized CPU-time index and critical path analysis. All the above indices can be useful in identifying a bottleneck's location, in terms of functions and processors. We will show that performance indices phrased in terms of *data-structures* can be of further use in identifying the actual *causes* of the bottleneck, in terms of interprocessor data structure interactions, within the identified function or processor. Furthermore, some bottlenecks related directly to data structures themselves can be most efficiently tracked down by first considering statistics in terms of data structures, then in terms of functions and processors. For example, when poor performance is caused by poor combinations of data structure distributions, the poor performance will be manifest across all functions and processors using those data structures. In those cases, statistics tabulated by processor and function may fail to clearly identify the location of, or even the existence of the performance bottleneck, since the performance degradation will be spread across the functions and processors involved.

The aim of the performance tuning process is to minimize the total execution time (or lifetime) of the program. Hence, each of our indices highlights the contribution of a particular performance problem to the lifetime of the program; each index represents the percentage reduction in lifetime that could be achieved, if the problem were completely eliminated. By comparing the indices one can determine the most significant performance problems and determine ways of eliminating them from the program. With the methodology we describe, users can trace such problems directly to the source code data structures being communicated.

We take the following three step approach to the systematic detection of the cause of performance problems:

- Select Data structure: Identify the data structure communications that cause the degradation in performance.
- Select code regions: Identify the function(s) in which those data structure communications have a significant performance cost.
- Determine cause of poor performance: Identify possible reasons for poor performance for the data structure communications, in the regions of code identified.

Each step requires the use of various indices in order to focus on the cause of the problem.

In this section we consider an example: a tridiagonal solver, which implements several methods for solving systems of tridiagonal equations. This kernel operation is used commonly in a number of fluid dynamics computations such as a block tridiagonal solver (one of the NAS parallel benchmarks [2]) which solves multiple, independent systems of block tridiagonal equations. Our example program is executed on an Intel iPSC/860 hypercube with 16 nodes. All the statistics shown in this section are derived by the data movement analyzer, from trace files generated during execution of the program on the hypercube.

This code was written by S. K. Weeratunga at NASA Ames Research Center. A complete description of the application is beyond the scope of this paper, but we will briefly discuss the nature of the communication and computation complexities involved in the program.

In this program a number of systems of the form $A \times b = c$ are solved, where b and c are vectors and A is a tridiagonal matrix. This data is organized efficiently into four distributed arrays, which represent the entire set of systems to be solved. The data is organized so that each processor does not have the complete information regarding the N/P systems it is supposed to solve. For the computation to be performed, pieces of data from all the processors are required to be communicated (achieved by using a transpose). A transpose of the four arrays are performed in $log(p)$ stages using bidirectional nearest neighbor communications, across successive hypercube dimensions, in sequence. The transpose is performed in the routine *xtrans*. This data is then used to solve the N/P systems locally, and the final solution vectors are transposed to correspond to the original distribution. This reverse transpose is performed in the routine *xrtrans*.

4.1 Which Data Structure Interactions cause Poor Performance?

As mentioned earlier, data structure interactions to enforce dependencies are the main reason for interprocessor communications. Typically, relative data distributions and alignments of data-structures significantly impact the performance of the program and dictate the communication-computation ratio in different functions or subroutines in the program. Thus, identifying costly data-structure interactions is a necessary first step in identifying the cause of poor performance.

The relative importance of a data-structure can be gauged by considering the significance of the total communication time involving the data-structure with respect to the entire program execution time. That is, a data structure's significance for performance debugging can be gauged by the fraction of total program execution time spent in communication involving that data structure, either as sender or receiver. We call this fraction the *communication index.*

$T_{send}(d,*)$ corresponds to the sum over all processors of the times during which the processor is blocked while sending the data structure d to any receiving data structure $(*)$. Similarly, $T_{recv}(*,d)$ is the total time blocked in receives into the data structure d from any data structure $(*)$. T_{total} is the sum over all processors of the program execution time.

$$CI(d) = \frac{T_{send}(d,*) + T_{recv}(*,d)}{T_{total}}$$

Just as we define the communication index CI, we can define send index and receive index to be

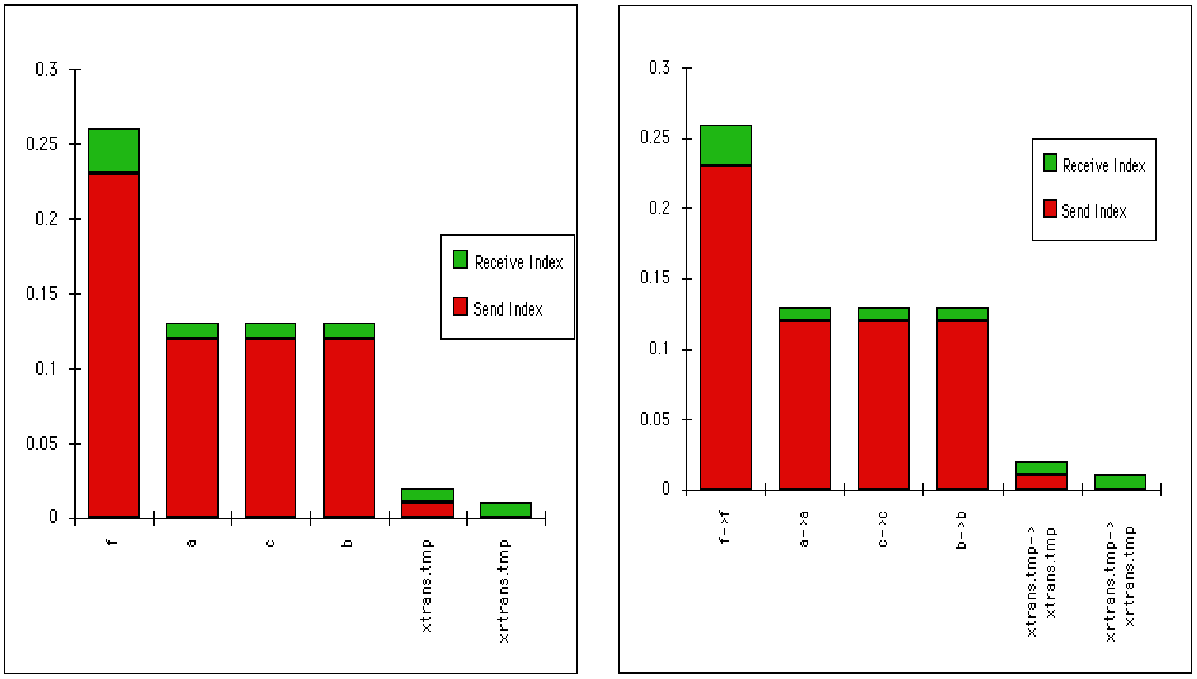


Figure 5: A: Communication index for individual data-structures that are communicated, in a version of a tridiagonal solver. B: Communication index for data structure pairs: shows that data structures are communicated to themselves (X-axis label).

$SI(d) = \frac{T_{send}(d,*)}{T_{total}}$ and $RI(d) = \frac{T_{recv}(*,d)}{T_{total}}$ respectively. Then, $CI(d) = SI(d) + RI(d)$.

Sending a segment of a data-structure involves buffer copy and communication initiation times. This time is incorporated into the T_{send} time of the data-structure being sent. Receiving into a data-structure could have idle time (enforced by synchronization at the receiving end), the network latency, and buffer copy time. This time is incorporated into the T_{recv} time of the data-structure which uses the data being sent.

The data structure which generally should be tuned first is the one with the largest communication index, since the larger the communication index value, the larger the potential savings in execution time achievable by the reduction of communications involving that data structure.

Figure 5 shows the communication index of all the arrays, during the execution in our example program. The figure shows that array f is the array with the largest communication index, roughly accounting for about 25% of the execution time of the program. Arrays a, b and c also have significant communication indices accounting for a total of about 35% of the lifetime of the program. The two temporary arrays $xtrans.tmp$ and $xrtrans.tmp$ have very low communication-indices and hence optimizing their communications will not have a significant impact on the program execution. For performance to be improved, we need to study arrays f, a, b and c in that order.

The indices defined so far have been in terms of single data-structures. But to detect problems that occur in communication between particular pairs of data structures, we require indices in terms of data structure pairs. Hence we can define a communication index for data structure pairs, that encapsulates the performance cost of the communication of values from data-structure (d_s), received by data-structure (d_r) as:

$$CI(d_s, d_r) = \frac{T_{send}(d_s, d_r) + T_{recv}(d_s, d_r)}{T_{total}}$$

Using the communication index of data structure pairs, we can focus in on the data structure interactions which have the most significant impact on the lifetime of the program.

Figure 5 (B) shows the communication index for array pairs. The X-axis corresponds to the array interactions, showing sending and receiving arrays, and the Y-axis corresponds to the communication index. In this case, arrays are communicated to themselves (to perform a transpose); hence the sending and receiving arrays match in all the array pairs, and the communication index is the same for individual arrays and for array pairs.

4.2 Where does Costly Data-structure Interaction Manifest?

Once we have determined the actual data-structures causing poor performance, we need to determine the functions or code segments in which these data-structure communication times are most significant. Instrumenting function entries and exits generally results in trace files of acceptably small size (unlike instrumenting loops and basic-blocks), so instrumentation of function entries provides a convenient means of demarking important code segments. [3].

As before with data structures, we use the communication index as a means of prioritizing the order in which functions need to be studied. That is, the communication index for the data structure pair (with sending data-structure d_s, and receiving data-structure d_r) in function f is given by:

$$CI(d_s, d_r, f) = \frac{T^{(f)}_{send}(d_s, d_r) + T^{(f)}_{recv}(d_s, d_r)}{T_{total}}$$

Figure 6 shows the communication index of various array pairs across two functions. The first set of four indices correspond to array communications in function *xtrans*, while the last two represent array communications in *xrtrans*. (None of the other functions have any communication in them and are hence not displayed in this figure.) It can be observed from this figure that all the array pair communications indeed take roughly the same amount of time. To determine how we can improve the performance of these array interactions, we need to study various performance indices described in the next subsection.

[3]Finer level instrumentation can be automatically enabled by using suitable selections in the AIMS instrumenter [16]. User defined blocks can be introduced and statistics of data-structure interaction in these blocks will also be reported. Further, linking the code segment to the data-structure interactions is possible, using the standard clickback facility in AIMS [16]

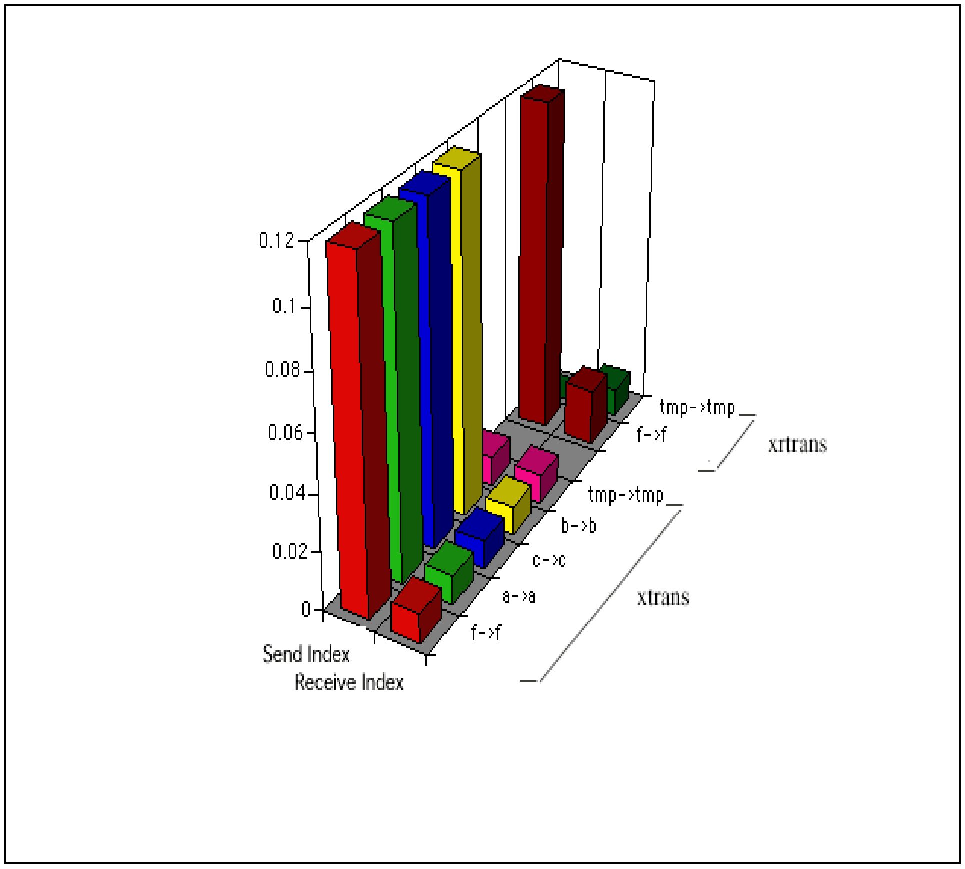


Figure 6: Communication-index for data-structure pairs: in terms of individual functions

Given the communication index values for array-pairs, tabulated over the whole program and over each function/subroutine, we can quickly determine the communications which should be studied, and the significance of these communications with respect to the overall execution of the program. The next step is to determine if the performance of these data structure interactions can be improved.

4.3 What is the Cause of Poor Data Structure Interaction?

Some of the common causes for poor performance of parallel programs are:

- Load imbalance
- Poor Link utilization
- Communication contention
- Communication overhead
- Bandwidth Utilization
- Poor communication computation overlap.

We define indices for each of these factors, with a goal of identifying the cause of poor performance in a particular data-structure interaction. All the indices are presented with respect to individual data-structures, data-structure pairs and data-structure pairs in individual functions or user-defined blocks of code. These indices are automatically generated and are all determined postmortem from trace data. The range of values for all the above indices is between zero and one: zero being good and one being bad. Having a normalized range for the performance indices helps in quickly tracking down the significant factors contributing to poor performance. In this paper, we will not discuss how we extract these index values from the trace file, due to space limitations. However, we will show how these statistics can help in comprehending program performance.

Figure 7 shows a summary view of all of our index values for each array pair in the tridiagonal solver program. Each axis corresponds to a performance index. There are six different performance indices displayed in this figure. A line is drawn connecting the corresponding values in each axis, for each array pair interaction. If performance indices for some data structure interactions are the same, then the index values are marked on top of each other.

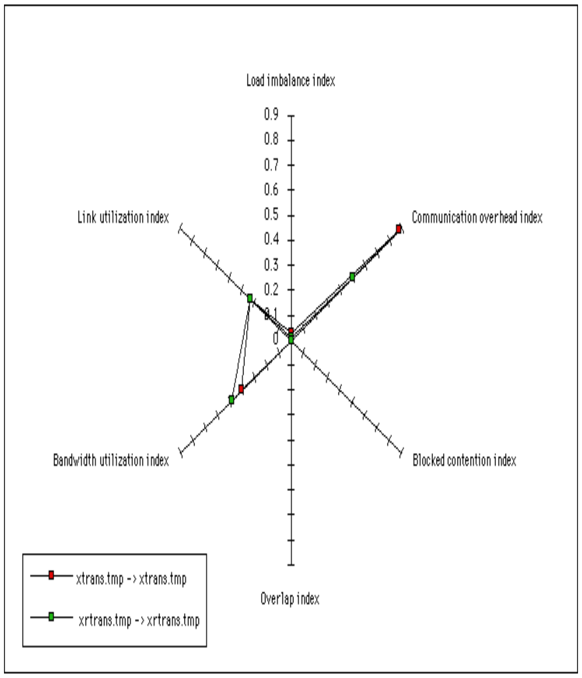
Figure 7 (A), shows the performance indices for two buffers (both called *tmp* but local to two different functions). From the figure we can gather that the most significant performance problem with these buffer movements is associated with communication overhead (the value of communication overhead in this plot is expressed as a fraction of the communication time between the two data-structures). This is due to the fact that these are temporary buffers used to transmit zero byte messages to set up a *forced* message communication of the actual arrays to be transposed. [4] Zero byte messages are used to set up links and buffers for the forced message containing the actual data to be communicated. The startup time is significant with respect to the communication time of these messages. However, since the communication index involving these data-structures is very small, we know that reducing the communication time of the two buffers will have little or no significant impact on the lifetime of the program.

Figure 7 (B) on the other hand shows that the other arrays (a, b, c and f) do not have a communication overhead problem, because their message sizes are large (in this case each message is approximately 16 Kbytes). The figure also indicates that the only performance related issue which could potentially be improved is the communication link utilization. As shown in the figure, the link utilization index is around 0.67. This implies that on the average, only about a third of the links are used during communication phases.

Hence, if there is any scope for improving the performance of this program, we need to reorganize the code in order to more effectively utilize the links (if possible).

Consider the algorithm for transpose, to understand why only a third of the links are used on the average during communication. There are $log(p)$ stages ($p = 8$) for performing a transpose. The processors communicate by exchanging data along each dimension of the cube, one dimension at a time.

[4] Forced messages are messages which exhibit better communication characteristics on the hypercube, by eliminating the need for lower level handshaking.



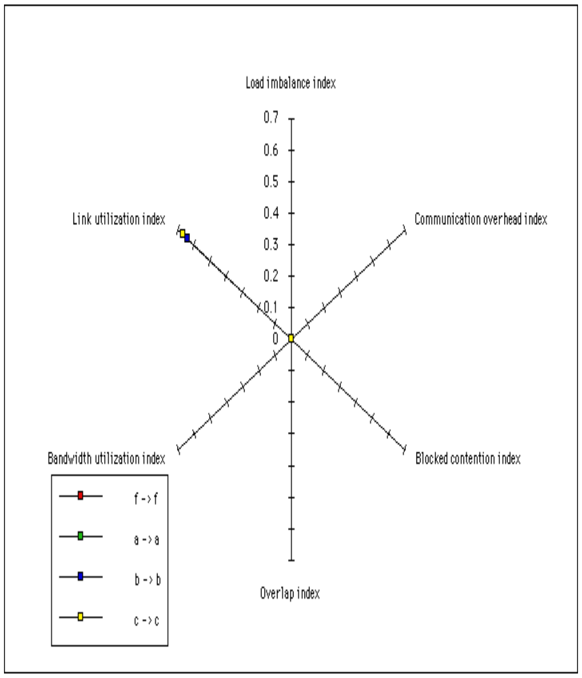


Figure 7: Comparison of various performance indices for array interactions. A. Performance indices for arrays *xtrans.tmp* and *xrtrans.tmp* B. Performance indices for arrays a, b, c and f.

Thus, each processor uses only one of its bidirectional links at any given time. This implies that for a p processor hypercube, there are only $p/2$ bi-directional links which will be used for each stage of the transpose. That is, $\frac{p \times log(p) - p}{2}$ links are not in use. For a cube with 8 processors, the number of links not in use for each stage is therefore twice that in use. This agrees with the link utilization index, for each data-structure, presented in the data-structure statistics.

Another aspect of the performance of this code is highlighted by a large value for send-index (not shown in the graph). The large value for the send-index indicates that the copy time for the source-array from the user space to I/O space is significant and significant reductions in communication times could be obtained by overlapping this copy time with useful computation. However, hardware support is needed to hide this copy time.

Unfortunately we cannot do better than the current performance of this program on the hypercubes (at NASA), since they do not have a communication co-processor to handle multiple communications on different links or overlap buffer copy time with useful computation. This implies that this program performs very well on the existing machine and its communication performance cannot be improved significantly without hardware support. Any improvements in the solution time for the problem will have to be made by modifying the algorithm.

In addition to the statistics discussed above, graphical representations showing interprocessor data movement can be useful for performance debugging. Figure 8 shows one such graphical view of the program with data structure information. This screen dump is a view of a run of the tridiagonal solver

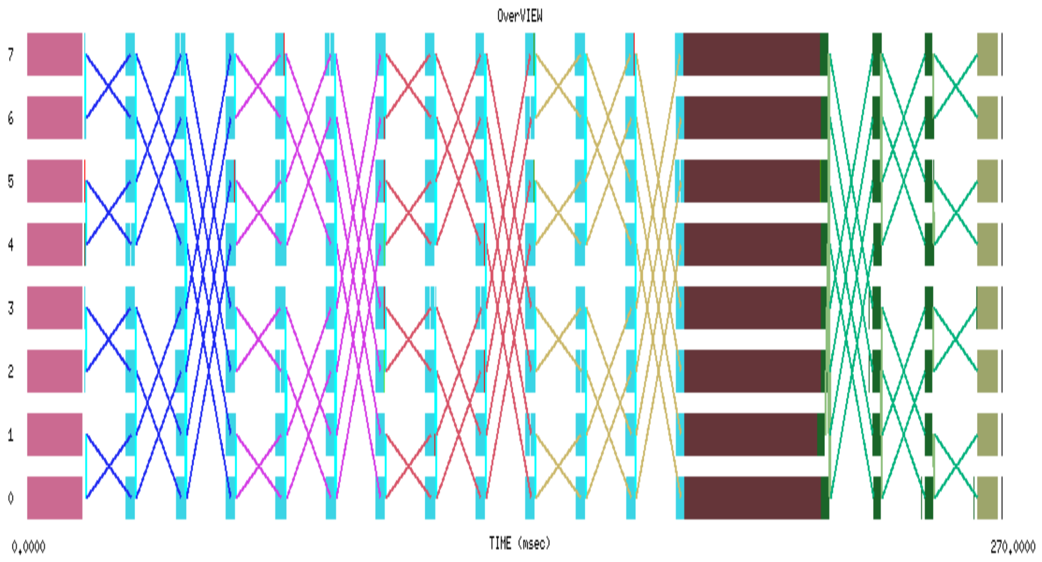


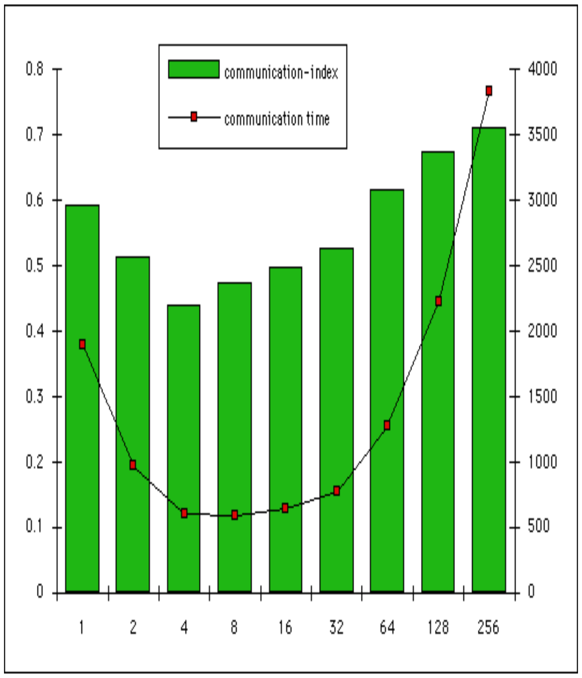
Figure 8: A time-line diagram of the tri-diagonal solver, with data structure information indicated by the color of the communication lines.

on eight processors. The X axis corresponds to time and the Y axis corresponds to processors. The state of each processor over time is indicated by the color of the horizontal bar associated with that processor. Interprocessor communication is indicated by lines between the communicating processors' bars. Data structure information is encoded in the color of the communication lines; all communications involving the same pair of sending and using data structures have the same color. For example, in the tridiagonal solver, the first 75% of the computation is dominated by communication. During this time four different arrays are transposed (each in $log(p) = 3$ stages); this is reflected in the view by four communication phases, each with a different color. Encoding data movement information in views such as this one provides the user with a richer data set to use in understanding program execution and performance.

The examples in this section were primarily used to demonstrate the flexibility of our methodology and to illustrate the detailed statistics that are generated for array interactions in FORTRAN programs. We have shown that with little additional space and time overhead at run time, it is possible to collect data that can be used to define a whole class of performance indices which can help in pinpointing the data structures with poor performance in specific functions, as well as in determining the causes of poor performance within those functions.

4.4 A Pipelined Version of Gaussian Elimination

In this section we consider a different version of the tridiagonal solver executed with a problem size of 256×256 on 16 processors. Here, instead of an explicit transpose of various matrices, the Gaussian elimination phase of the computation is parallelized. A pipelined Gaussian elimination algorithm is used



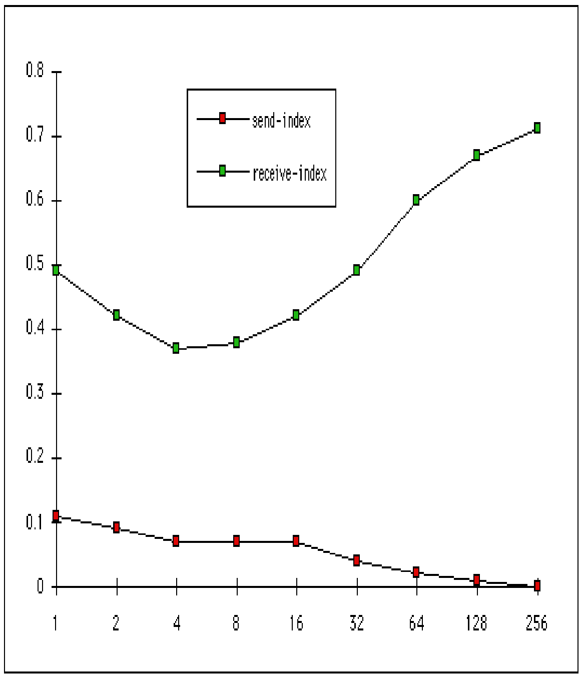


Figure 9: Left: Communication time and communication index for varying block sizes of communication from data structure *sbuf* to *rbuf*. Right: Send and receive indices for communication of *sbuf* to *rbuf*. Receive index increases more than send index.

for this purpose. Pivot elements of each column are determined in sequence and these values are used for updating succeeding columns. Processor i has the columns $\frac{(i-1)\times n}{p}$ to $\frac{i\times n}{p}$. Processors are arranged in a ring, so that each processor receives the updated pivot element from its left neighbor, updates its own column, and then transmits the data to its right neighbor.

There are a number of decisions that need to be made with respect to communicating the arrays around the ring, in order to obtain good performance. One obvious optimization is to pack the pivot elements from all the arrays into a single message (called *sbuf* on the sending end and *rbuf* on the receiving end), thus reducing the number of messages to be transmitted. With this optimization, the number of pivot elements that need to be transmitted in a message governs the execution time of the program. The number of pivots from each array, in each message from *sbuf* to *rbuf*, is the *block size*, which can range anywhere from 1 to 256. The maximum block size of 256 corresponds to all pivot updates of all arrays being sent in a single message from *sbuf* to *rbuf*. So as we increase the block size from 1 to 256, the number of messages reduces from 256 messages to a single message.

Consider Figure 9, which shows the communication time and the communication index of the program as the number of pivots transmitted in a message is increased. An interesting feature is that the communication time starts to decrease initially as the block size is increased, but when the block size is increased above 4, the communication time starts to increase again. By analyzing the trace files for various block sizes, we should be able to explain the reason for this performance problem automatically.

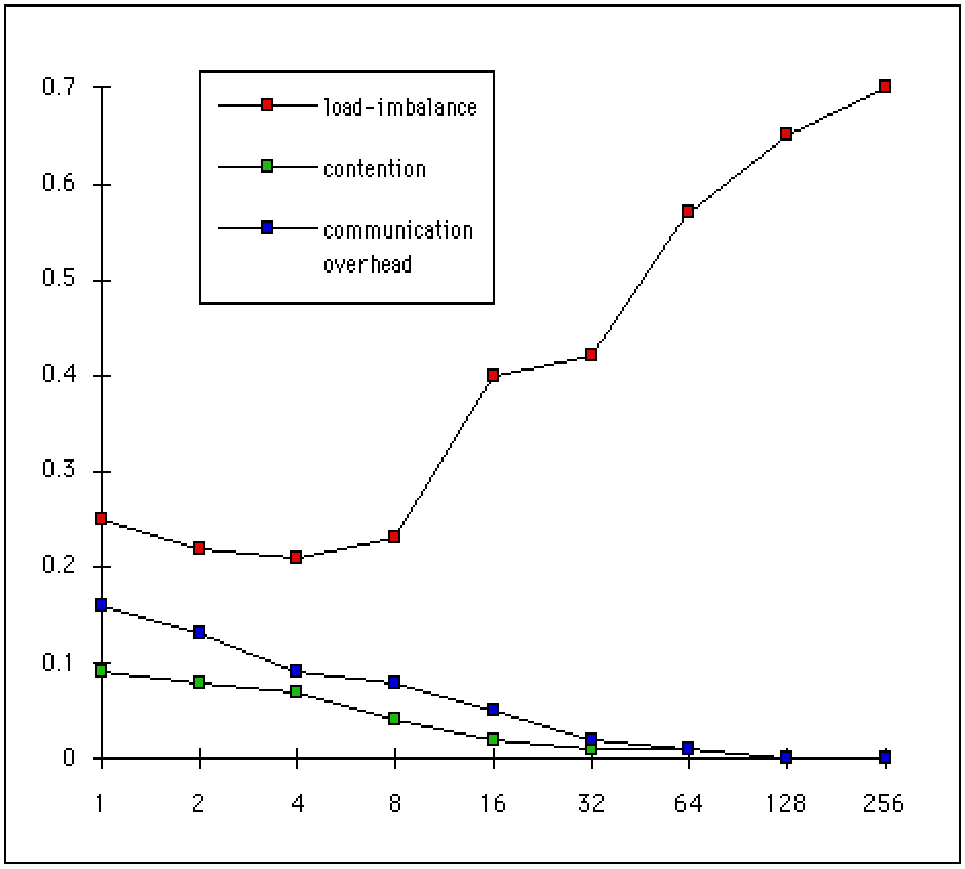


Figure 10: Performance indices for communication from *sbuf* to *rbuf*: For small block sizes, communication overhead, contention and load-imbalance are all significant issues, while for large block sizes, load-imbalance becomes significant.

Figure 9 compares the send index and receive index for varying block sizes. We see that the receive index reduces for block sizes from 1 to 4, but starts increasing beyond that, while the send index reduces (though not in proportion to the receive index) for increasing block sizes. This suggests that a problem in the receiving end becomes significant as the block sizes are increased.

The reason for this behavior is exposed by the performance indices used for studying the data-structure interactions. These indices show that load imbalance, communication overhead and contention are the most significant performance factors for this program. Figure 10 compares these indices for different block sizes. (The values of these performance indices, for this data-structure interaction, are obtained automatically from the trace events by the data movement analyzer, and will be described in a future paper.) We can see in this figure that for a block size of 4, all the indices have low values, beyond which the load-imbalance increases while the other indices continues to decrease. This increase in load imbalance index indicates that the receive into *rbuf* is initiated before the communication of *sbuf* is initiated, resulting in larger idle times (and hence larger receive index values shown in Figure 9). This results in an imbalance where processors have completed their work and are idling for data dependencies to be satisfied, before proceeding. When the block size is 4, a delicate balance between the communication overhead and idle times in processors has been achieved, for this problem size and number of processors, resulting in the lowest execution time for the program.

5 Conclusion

While it has long been understood that improving performance of a parallel program is dependent on understanding the communication characteristics of data structure interactions, there have been very few tools that provide means of studying program performance in terms of these interactions. In this paper, we have presented a novel scheme for generating such statistics automatically, through instrumentation, monitoring and postmortem analysis.

We detect data structure definitions and uses with the help of static flow analysis (without the need for inter-procedural analysis). This information, supplemented with some run time information, enables the generation of trace data tracking data movements, with minimal additional time overhead at run time. This data is then interpreted postmortem by a suite of graphical and statistical tools that output information with respect to source code data structures.

With the help of several examples, we have demonstrated that performance can be effectively analyzed by tabulating performance indices based on data structure interactions. The performance indices hide raw numbers from the application developers, instead highlighting the significance of various performance problems to the reduction of the program's lifetime. The causes of performance degradation are attributed to specific data structure pairs. This enables programmers to identify ways to rearrange the code and/or algorithm to eliminate the performance problem, or to change the alignments and distributions of the relevant data structures, in order to improve performance.

There are still a number of issues which are areas of on-going research: currently our implementation does not obtain information about the specific array subsections that are actually sent when communication takes place. This information may be essential for determining the applicability of certain types of communication optimizations: for example, the replacement of all to all communications with a broadcast. We have implemented our approach for F77 with message passing and static data structures; we have not yet considered some more complex issues that arise in efficiently tracking dynamic data structures. We are currently testing our approach and the robustness of our implementation on larger applications, and hope to present our experiences with these codes at a later date.

References

[1] Thomas E. Anderson and Edward D. Lazowska, " Quartz: A Tool for Tunning Paralel Program Performance," *Proceedings of the 1990 Conference on Measurement and Modeling of Computer Systems*, May 1990.

[2] David Bailey, John Barton, Thomas lasinski and Horst Simon, " The NAS Parallel Benchmarks," *Report RNR-91-002, NASA Ames Research Center*, January 1991.

[3] S. H. Bokhari, " Communication Overhead on the Intel iPSC/860 Hypercube," *ICASE Interim Report 10*, May 1990.

[4] H.Davis, S.R. Goldschidt and J. Hennessy, "Tango: A Multiprocessor Simulation and Tracing System," *Proceedings of the International Conference on Parallel Processing*, August 1991.

[5] D. Gannon, J.K.Lee, B. Shei, S.R.Sarukkai, S.Narayana, N.Sundaresan, D.Atapattu, F.Bodin, " SigmaII: A toolkit for Building Parallelizing Compilers and Performance Analysis Systems ," , *Proceedings of the Programming Environments for Parallel Computing Conference*, Edinburgh, April 1992.

[6] S. L. Graham, P.B. Kessler and M. K. McKusick, " An Execution Profiler for Modular Programs," *Software Practice and Experience,* August 1983.

[7] Michael T. Heath, Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software,* September 1991.

[8] High Performacne Fortran Forum, " High Performance Language Specification," *Rice University,* 1993.

[9] Jenq Kuen Lee and Dennis Gannon, "Object Oriented Parallel Programming Experiments and Results," *Proceedings of Supercomputing '91*, 1991.

[10] Margaret Martonosi and Anoop Gupta, " MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proceedings of the 1992 ACM International Conference on Measurement and Modeling of Computer Systems,* June 1992.

[11] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim and Timothy Torzewski, "IPS-2: The second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems,* April 1990.

[12] Sekhar R. Sarukkai and Allen Malony, " Perturbation Analysis of High Level Instrumentation of SPMD Programs, *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* San Diego, May 1992.

[13] Sekhar R. Sarukkai and Dennis Gannon, "SIEVE: A Performance Debugging Environment for Parallel Programs," *Journal of Parallel and Distributed Computing,* June 1993.

[14] Sekhar R. Sarukkai and Jerry Yan, "Integration of Perturbation Compensation and Application Monitoring Tools for Message Passing Parallel Porgrams," *Submitted to IEEE Transactions on Parallel and Distributed Systems*

[15] Eileen Kraemer and John T. Stasko, " The Visualization of Parallel Systems: An Overview," *Journal of Parallel and Distributed Computing,* June 1993.

[16] Jerry Yan, Charles Fineman, Phil Hontalas, Melisa Schmidt, Sherry Listgarten, Pankaj Mehra, Sekhar Sarukkai and Cathy Schulbach, " The Automated Instrumentation and Monitoring System (AIMS) Reference MAnual," *NASA Ames Research Center,* June 1993.

[17] "UNICOS Performance Utilities Reference Manual," *Cray Research Inc.,* 1991.