

Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies

Jeffrey Vetter

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, USA 94551
+1.925.424.6284
vetter3@llnl.gov

ABSTRACT

We present a technique for performance analysis that helps users understand the communication behavior of their message passing applications. Our method automatically classifies individual communication operations and it reveals the cause of communication inefficiencies in the application. This classification allows the developer to focus quickly on the culprits of truly inefficient behavior, rather than manually foraging through massive amounts of performance data. Specifically, we trace the message operations of MPI applications and then classify each individual communication event using decision tree classification, a supervised learning technique. We train our decision tree using microbenchmarks that demonstrate both efficient and inefficient communication. Since our technique adapts to the target system's configuration through these microbenchmarks, we can simultaneously automate the performance analysis process and improve classification accuracy. Our experiments on four applications demonstrate that our technique can improve the accuracy of performance analysis, and dramatically reduce the amount of data that users must encounter.

1 INTRODUCTION

Message passing serves as an effective programming technique for exploiting coarse-grained concurrency on distributed computers as evidenced by the popularity of the Message Passing Interface (MPI) [9, 22]. Nowadays, most applications for terascale computing environments, such as the systems at the NSF Alliance Centers, the DOE ASCI Labs, and the computational grid [6], rely totally on MPI for inter-nodal communication. Often, users even employ MPI for intra-nodal communication because many implementations of MPI provide highly optimized communication operations that use shared memory rather than networking protocols for message transfers.

Appears in ACM International Conference on Supercomputing 2000 (Santa Fe, NM USA).

The performance of these distributed applications can be challenging to comprehend, as an application's performance stems from three factors: application design, software environment, and underlying hardware. This comprehension is even more complex when considering computer systems with hundreds, if not thousands, of processors. One strategy for optimizing the performance of these applications is to eliminate the application's communication inefficiencies. These inefficiencies arise under many scenarios; one common scenario occurs when processors are staled for long periods waiting to receive a message or when the application loses the opportunity to perform computation simultaneously with communication. Traditionally, programmers infer explanations for these types of inefficiencies manually, basing their conclusions on knowledge of the source code, message tracing, visualizations, and other customized performance measurements. This manual analysis is time-consuming, error-prone, and complicated, especially if the developer must analyze a large number of messages, or if the messages are non-deterministic.

1.1 Key Insights and Contributions

Simply put, given a reliable technique for automatically evaluating an application's message transfers, users could expect the performance analysis system to converge on inefficient communication operations. Such automation would typically reduce the data volume dramatically and would highlight specific operations in the application that are suspected of communication inefficiency. Additionally, for any performance analysis technology to help users understand their application's performance, the technology should be able to explain performance phenomena in terms of decisions a user makes while constructing their application.

This paper contributes new ideas in regards to both concerns. First, we automate a portion of the performance analysis process by categorizing the performance data, emphasizing only those communication operations that appear unusual in the context of the current software and hardware configuration. Our method automatically classifies individual communication operations for an application and it attempts to reveal the specific causes of inefficiencies in the application's communication by mapping these classifications to source code. This automation allows the developer to focus quickly on the culprits of truly inefficient behavior, rather than manually foraging through massive amounts of performance data.

Second, we show that an extensible, trainable technique is both practical and valuable for analyzing message behavior across

different hardware and software configurations. Accordingly, our technique is rooted in machine learning, and, in particular, decision tree classification (DTC). Using this method, we *train* our classification system with MPI benchmarks that exhibit both efficient and inefficient communication behaviors. Since our classification technique adapts to the target software and hardware configuration, we provide users with precise evaluations of their messaging activity while reflecting differences across configurations. Also, our choice of decision tree classification is a considerable advantage in that a user can easily understand how the filter produced each specific classification.

We demonstrate our ideas with an operational prototype by applying it to four message-passing applications. We use the classification system to reveal the specific location of communication inefficiencies in the source code, as well as an explanation for the inefficiency.

1.2 Paper Organization

The remainder of this paper discusses these issues in more detail. Section 2 introduces MPI and trace-based performance analysis. Section 3 explains decision tree classification. Section 4 provides an implementation overview, and in particular, describes how we integrated trace-based performance analysis with decision tree classification. Next, Section 5 uses several MPI applications to evaluate the goals our approach. Section 6 outlines related work. Finally, Section 7 states our conclusions.

2 USING MPI FOR DISTRIBUTED APPLICATIONS

We focus our evaluation on the message-passing interface (MPI) [9, 22] because MPI serves as an important foundation for a large group of applications, and because the elimination of communication inefficiencies from MPI applications is a well-known technique for improving application performance.

Concisely, MPI provides a wide variety of communication operations including both blocking and non-blocking sends and receives, and collective operations such as broadcast and global reductions. We concentrate on basic message operations: blocking send, blocking receive, non-blocking send, and non-blocking receive. Note that MPI provides a rather comprehensive set of messaging operations and we do not evaluate every combination of these operations. We focus our attention on this subset of operations because they are well understood and widely used. We believe that our strategy is applicable to other MPI operations, and we are beginning to evaluate them.

2.1 MPI Blocking Send–Receive

MPI’s primitive communication operation is the blocking send to blocking receive. The first message operation in Figure 1 illustrates one message transfer from task 0 to task 1 using a blocking send and blocking receive, respectively. A *blocking send* (MPI_Send) does not return until both the message data and

Task 0	Task 1
<pre>#define size 1024 int sdata[size]; int rdata[size]; MPI_Status status; MPI_Request request; int tag = 30; /* initialization */ /* ---- blocking send-recv 0 -> 1 */ /* fill sdata */ MPI_Send (sdata, size, MPI_INT, 1, tag, MPI_COMM_WORLD); /* use or overwrite sdata */ /* ---- non-blocking send recv 1->0 */ MPI_Irecv (rdata, size, MPI_INT, 1, tag, MPI_COMM_WORLD, &request); /* computation excluding rdata */ MPI_Wait(&request,&status); /* use rdata */ /* finish */</pre>	<pre>#define size 1024 int sdata[size]; int rdata[size]; MPI_Status status; MPI_Request request; int tag = 30; /* initialization */ /* ---- blocking send-recv 0 -> 1 */ MPI_Recv (rdata, size, MPI_INT, 0, tag, MPI_COMM_WORLD, &status) /* use rdata */ /* ---- non-blocking send recv 1->0 */ /* fill sdata */ MPI_Isend (sdata, size, MPI_INT, 0, tag, MPI_COMM_WORLD, &request); /* computation excluding sdata */ MPI_Wait(&request,&status); /* use or overwrite sdata */ /* finish */</pre>

Figure 1: Example message operations with MPI.

envelope have been safely stored. When the blocking send returns, the sender is free to access and overwrite the send buffer. Note that these semantics allow the blocking send to complete even if no matching receive has been executed by the receiver. Task 0, in Figure 1, uses MPI_Send to transfer the contents of sdata to task 1. Once MPI_Send returns, task 0 is free to use or overwrite the data in sdata.

A *blocking receive* (MPI_Recv) returns when a message that matches its specification has been copied to the buffer. The receive operation specifies three parameters that identify which message it wishes to match: a source identifier, a message tag, and a communicator. In Figure 1, task 1 continues only when MPI_Recv returns, which assures task 1 that rdata is ready to use. Both task 0 and task 1 must use the same communicator and message tag for this message transfer.

2.2 MPI Non-blocking Send–Receive

As an alternative to blocking communication operations, MPI provides non-blocking communication to allow an application to overlap communication and computation. Usually, this overlap improves application performance, albeit at the cost of some software complexity. In non-blocking communication, initiation and completion of communication operations are distinct. The second message operation in Figure 1 illustrates one message transfer from task 1 to task 0 using a non-blocking send and non-blocking receive, respectively.

A non-blocking send has both a send start call and a send complete call. The send start call (MPI_Isend) initiates the send operation and it may return before the message is copied from the send buffer. The send complete call (MPI_Wait) completes the non-blocking send by verifying that the data has been copied from the send buffer. It is this separation of send start and send complete that provides the application with the opportunity to perform computation. Task 1, in Figure 1, uses MPI_Isend to initiate the transfer of sdata to task 0. During the time between MPI_Isend and MPI_Wait, task 1 cannot modify sdata because the actual copy of the message data from sdata is not guaranteed until the MPI_Wait call returns. After MPI_Wait returns, task 1 is free to use or overwrite the data in sdata.

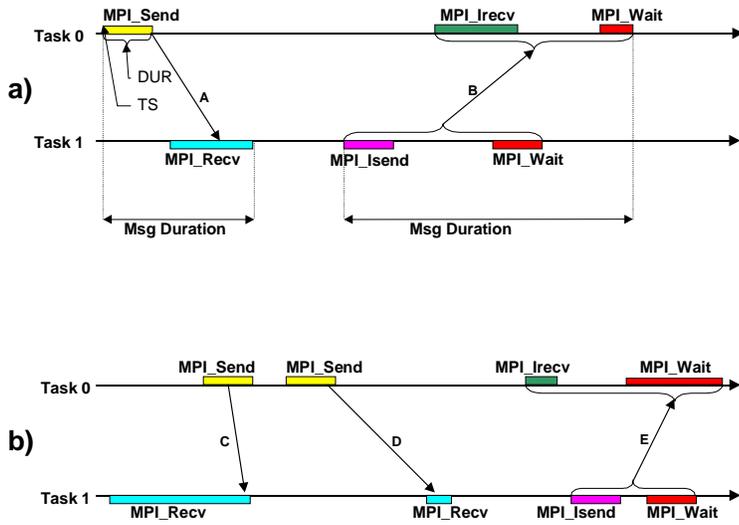


Figure 2: Space-time diagrams of message operations.

Similarly, a non-blocking receive has both a receive start call and a receive complete call. The receive start call (`MPI_Irecv`) initiates the receive operation and it may return before the incoming message is copied into the receive buffer. The receive complete call (`MPI_Wait`) completes the non-blocking receive by verifying that the data has been copied into the receive buffer. As with non-blocking send, the application has the opportunity to perform computation between the receive start and receive complete calls. Task 0, in Figure 1, uses `MPI_Irecv` to initiate the receive of `sdata` from task 1. During the time between `MPI_Irecv` and `MPI_Wait`, task 0 cannot read or modify `rdata` because the message from task 1 is not guaranteed to be in this buffer until the `MPI_Wait` call returns. After `MPI_Wait` returns, task 0 is free to read `rdata`.

2.3 Communication Efficiency

Poor communication efficiency can restrict both the performance and scalability of distributed applications [5]. Qualitatively, we define poor communication efficiency as excessive cost for sending, receiving, or transferring messages, where we define excessive cost as greater than the cost for a normal message transfer with a similar configuration and message size. From the performance perspective, several timings can help reveal the efficiency of MPI communication.

Figure 2(a) shows the normal flow of messages for the code segment in Figure 1. Each MPI operation has a start time, $\text{MPI_Op}_{\text{TS}}$, and duration, $\text{MPI_Op}_{\text{DUR}}$. We define the duration of a complete message transfer to be the difference between the end of the receive and the beginning of the send. For blocking receives, this message duration is $(\text{MPI_Recv}_{\text{TS}} + \text{MPI_Recv}_{\text{DUR}}) - \text{MPI_Send}_{\text{TS}}$. For nonblocking receives, we must use the completion time of the matching `MPI_Wait`, so the message duration is $(\text{MPI_Wait}_{\text{TS}} + \text{MPI_Wait}_{\text{DUR}}) - \text{MPI_Send}_{\text{TS}}$. Send start applies to both blocking and non-blocking sends. Messages A and B in Figure 2(a) illustrate a normal blocking message transfer and a normal non-blocking message transfer, respectively. Although this definition does not capture the actual departure and arrival of

messages in hardware, it does capture sufficient information from the viewpoint of the application to make decisions about communication efficiency.

Put simply, our notion of efficiency expects MPI tasks to expend only a normal amount of time for any message transfer, where a *normal* message transfer on the target system is empirically measured. Any deviation from this standard causes the task to idle unnecessarily while waiting on the communication operation to complete.

Figure 2(b) shows several types of communication inefficiencies. For example, message C has a high $\text{MPI_Recv}_{\text{DUR}}$ when compared to the normal transfer for message A. Further examination reveals that both the message duration and the $\text{MPI_Send}_{\text{DUR}}$ are normal. Based on this information, we label this inefficiency a late send. We identify these inefficiencies with labels that have direct meaning to the user in terms of software design. In this example, a user could improve the efficiency of this message transfer by moving the

`MPI_Send` forward in the control flow of task 0. Using this process, we initially developed seven types of message transfers as shown in Table 1.

We selected and named these inefficiencies with the goal of helping users improve the performance of their applications. Many other categories would be valid and could provide similar types of diagnostic information; however, our categories are consistent and they prescribe to the user straightforward modifications to the application.

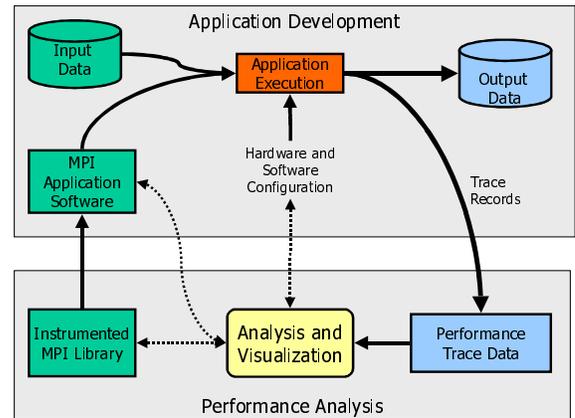


Figure 3: MPI Application Performance Tracing.

2.4 Tracing MPI Applications

To capture the respective timings of MPI operations, we trace application execution. As illustrated in Figure 3, using an instrumented library (or an otherwise instrumented application), tracing captures information about an application component in the form of discrete events over a period of time. Users may analyze these events at runtime or they may write these events to a log file for post-mortem analysis. Most trace-based performance

Type	Applies to	Performance diagnosis
Normal	All operations	Message operation appears normal.
Late Send	MPI_Send	Send operation appears late. Move forward in control flow.
Late Receive	MPI_Recv	Receive operation appears late. Move forward in control flow.
Late Send Post	MPI_Isend	Same as Late Send.
Late Send Wait	MPI_Wait for matching MPI_Isend	Wait operation appears late. Move forward in control flow to allow MPI task earlier access to the data in send buffer.
Late Receive Post	MPI_Irecv	Same as Late Receive.
Late Receive Wait	MPI_Wait for matching MPI_Irecv	Wait operation appears late. Move forward in control flow to allow MPI task earlier access to the data in receive buffer.

Table 1: Communication inefficiencies.

analysis systems including PICL, Pablo, and Tau [8, 18, 21] use this approach. We choose tracing because it provides a chronological description of application events and consequently, it is more general than techniques such as profiling. This detailed description of message activity is necessary because we must be able to reconcile specific message sends with their receives. Our tracing system takes advantage of MPI’s profiling layer by capturing information about each MPI call into an event structure, periodically flushing the event buffer to local disk.

These benefits of tracing in mind, several shortcomings can limit tracing’s usefulness. First, instrumentation is necessary in either the application itself or a library that the application calls. Second, the perturbation introduced by tracing can change the results of the analysis [7]. Third, tracing generates a tremendous volume of data. Hence, users must extract useful information from these large traces.

These shortcomings are manageable for our current implementation. First, users can trace applications transparently by using an instrumented library, which, in turn, uses the MPI profiling layer. Second, tracing is especially useful for capturing message passing and I/O activity because they are generally high-latency operations and the amortized costs for tracing are relatively small when compared to communication and I/O operations. During our evaluation of MPI applications, the change in overall application runtime with tracing was statistically insignificant relative to the normal runtime. Finally, although the third challenge has prompted significant research efforts, especially in the area of visualization [10, 23], we advocate a different approach, which is the subject of this paper.

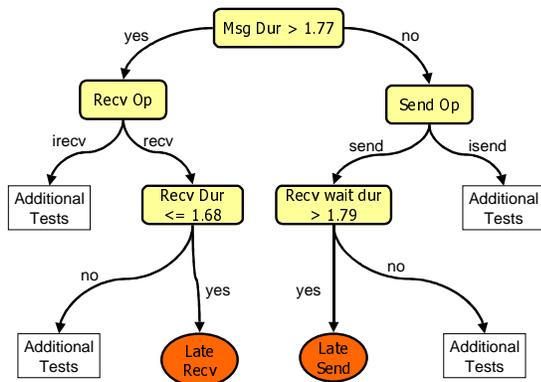


Figure 4: Example decision tree.

3 DECISION TREE CLASSIFICATION

Decision tree classification, as defined in machine learning literature [16], is a classification technique that discovers and analyzes patterns in input feature vectors. This technique maps the input feature vector into one of several predefined classes by applying a series of tests to the feature vector. Very simply, a decision tree is an unbalanced tree with internal nodes representing tests, and leaf nodes signifying classes as represented in Figure 4. Classification begins with the test at the root of the tree, and continues along branches of the tree until a leaf (or class) is encountered. Each test has mutually exclusive and exhaustive outcome. In our domain of communication performance analysis, Figure 4 illustrates two branches of an example decision tree for classifying a message transfer. In Figure 4, the classification procedure must apply three tests to arrive at either of the two classes depicted, late recv and late send. As a supervised learning technique, decision trees generate their series of tests inductively from a set of representative examples provided by the user.

We focus on decision trees for three reasons [4]. First, models developed by decision trees are intelligible to human users. Unlike neural networks and genetic algorithms, decision trees allow users to verify how the classification arrived at its answer simply by looking at the set of tests applied to an input feature vector. Second, decision trees are relatively efficient in both modeling and classification when compared to other supervised learning techniques. Finally, other techniques such as neural networks may lower the error rate of the classification procedure; however, we exchange this possible inaccuracy of decision trees

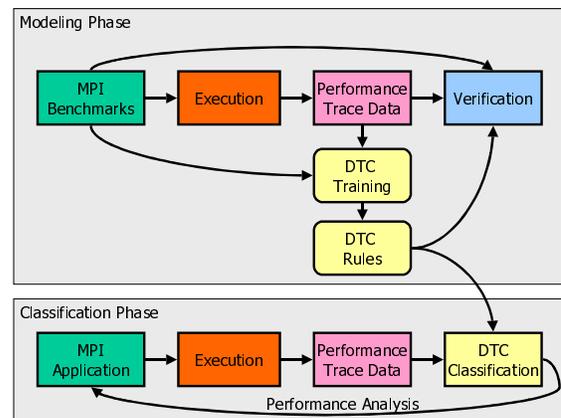


Figure 5: Integration of decision tree classification with performance analysis.

Send Type	Send Dur	Send Wait Dur	Recv Type	Recv Dur	Recv Wait Dur	Msg Size	Msg Dur	Class
send	1.06	0.00	recv	0.98	0.00	16	0.97	normal
send	1.00	0.00	recv	0.92	0.00	16	0.93	normal
isend	1.77	1.22	recv	41.49	0.00	2	0.70	late send post
isend	1.26	1.14	recv	4.50	0.00	2	0.73	late send post
isend	0.65	1.84	irecv	1.12	0.82	32768	1.83	late recv post
isend	0.93	3.15	irecv	1.29	0.87	32768	3.05	late recv post

Figure 6: Examples of user classified training records (input feature vectors) for modeling phase.

Send Type	Send Dur	Send Wait	Recv Type	Recv Dur	Recv Wait	Msg Size	Msg Dur	Class	Conf Factor
send	2.90	0.00	irecv	2.70	1.86	1250	2.07	late recv post	0.92
send	3.35	0.00	irecv	1.21	1.60	1250	2.36	late recv wait	0.99
send	3.36	0.00	irecv	1.21	1.45	1250	2.34	late recv wait	0.99

Figure 7: Trace records for CG-Heat messages after classification.

for their superior understandability.

As noted earlier, the use of decision trees has two distinct phases: the modeling phase and the classification phase. In the modeling phase, the user provides the decision tree with representative examples of feature vectors *along with the user-specified classification*. The decision tree constructs its set of tests from these samples. In the classification phase, the user provides the decision tree with unclassified feature vectors and the *trained* decision tree, then, classifies each vector.

4 IMPLEMENTATION OVERVIEW

Our experiment architecture consists of a MPI tracing tool that captures a trace file for each MPI task and a post-mortem analysis filter that merges and classifies the MPI message activity of the application. As illustrated in Figure 5, our process divides into two parts: the modeling phase and the classification phase.

4.1 Modeling Phase

In the modeling phase, we train the decision tree by providing it with examples of efficient and inefficient MPI behavior. During this phase, the decision tree generates the series of rules (or tests) it will later apply to unclassified performance data. As Figure 5 depicts, the modeling phase begins with the execution of MPI microbenchmarks on the target platform using the same software

Task 0	Task 1
<pre> phase("late send post"); for (size = 1; size <= maxMsgSize; size *= 2) { for (i = 0; i < maxIterations; i++) { delay = delayVariance * drand48 (); MPI_Barrier (MPI_COMM_WORLD); usleep (delay); MPI_Isend (sdata, size, MPI_INT, 1, tag, MPI_COMM_WORLD, &request); MPI_Wait (&request, &status); } } </pre>	<pre> phase("late send post"); for (size = 1; size <= maxMsgSize; size *= 2) { for (i = 0; i < maxIterations; i++) { MPI_Barrier (MPI_COMM_WORLD); MPI_Recv (rdata, size, MPI_INT, 0, tag, MPI_COMM_WORLD, &status); } } </pre>

Figure 8: Microbenchmark for late send post of isend-recv transfer.

and hardware configuration as the user's environment. We use multiple microbenchmarks to train our decision tree; they are normal MPI programs with one simple procedure call delineating the different phases of performance behaviors. The benchmarks reproduce these behaviors for message activity in a way that maps easily to the decisions a user makes about application design. These benchmarks currently create the message transfer behaviors introduced in Section 2.3; these behaviors are normal, late send, late recv, late send post, late send wait, late recv post, and late recv wait.

Figure 8 illustrates the microbenchmark for a late send post of an isend-recv message transfer

between task 0 and task 1. After each task marks its phase, it loops over a range of message sizes for several iterations. Since the benchmark in Figure 8 is emulating a late send post, it delays the MPI_Isend and generates the delay time with a uniform distribution function over a predefined range. To prevent any biases in the underlying synchronization mechanism, we execute each benchmark with multiple pairs of tasks.

During microbenchmark execution, the tracing system captures an event for each MPI call and writes it to a trace file. Each microbenchmark labels each phase of their execution with a stamp describing the behavior that it is emulating. Later, a filter merges multiple trace files and reconciles sends with receives. The result of this reconciliation is a series of records containing fundamental information about each message transfer.

In Figure 6, all times are normalized to the average of those times measured for the normal class for a range of message sizes. Each record contains two attributes identifying the type of send operation and receive operation, and five durations relating to the transfer: send duration, send wait duration, receive duration, receive wait duration, and message duration. All durations except the message duration are the length of the time required to complete the respective MPI call as noted in Section 2.3.

With these training records (or examples) in hand, we use the DTC generator to create the decision tree. The output of this process is a decision tree and a set of production rules as outlined in Section 3. Quinlan [16] provides complete details on the algorithm for decision tree generation. We save this decision tree for the classification process. Take note that each decision tree applies to only one software and hardware configuration. When this configuration changes, the user must regenerate the decision tree.

To verify the tests generated by the decision tree, we apply it to the original training data used to develop the tree. For this training set, Table 2

shows the confusion matrix. This matrix provides a notion of how the actual classifications compare to the predictions made by the decision tree. Our confusion matrix demonstrates that DTC does not provide perfect classification; however, several techniques, such as boosting, can improve this error rate. We are currently exploring other data cleaning techniques that may improve the predictive power of DTC for performance analysis data.

		Prediction						
		normal	late send	late rcv	late rcv post	late rcv wait	late send post	late send wait
Actual	normal	786	4	3	3	2	14	28
	late send	13	406	0	0	0	0	0
	late rcv	21	0	399	0	0	0	0
	late rcv post	48	1	0	314	55	0	2
	late rcv wait	48	0	0	127	245	0	1
	late send post	20	0	0	0	0	397	3
	late send wait	16	0	1	0	0	9	394

Table 2: Confusion matrix.

4.2 Classification Phase

In the classification phase, we feed trace data from MPI applications into the decision tree classifier, and it classifies all of the application’s message activity based on the decision tree generated during the modeling phase. As with the modeling phase, the classifier merges the trace files, reconciles message sends with receives, and normalizes the durations. At this point, the classifier reads the unclassified records and applies the decision tree rules to each message transfer. Figure 7 presents several classifications for the example application CG-Heat. The last two columns show the classification and the confidence factor that the DTC filter selected for each message transfer.

4.3 Mapping Classifications to Source Code

This initial classification of all the message transfers is very useful; however, ultimately, we want to map these classifications back to the original source code. Many tracing tools provide only details on message sends and receives based on node identifiers (e.g., task 0 sends to task 1) without regard to their location in the source code. This limitation forces users into an arduous procedure of first, gathering more information about the send-receive pair, such as tag and communicator information, and then, searching for those locations in the source code.

To improve this situation, our tracing tool captures the address of the caller routine and a relative offset from the beginning of the caller routine to the called MPI function. With this additional information, we can discriminate among the MPI calls and summarize the decision tree classifications based on the location of the sender and the receiver in the source code. In most cases, this summary reduces all message transfers to a handful of MPI calls that the user can immediately investigate. Then, given the predominant class of the communication operations, the user knows the performance rating of each particular MPI call. Figure

9 presents an example of the caller-dependent classification summary for sample application NAS SP. Columns 1 and 2 provide the names of the subroutines calling MPI functions. The next seven columns furnish the classifications for the messages transferred between the sender and receiver. For instance, row 2 of Figure 9 reveals that our system classified over 99.8% of the messages transferred from `x_solve+7716` to `x_solve+888` as late send posts. With this evidence in hand, a user could investigate the software hoping to move forward this send’s initiation in the control flow of `x_solve`. We investigate this process of communication optimization in more detail later in Section 5.2.

Location		Class						
Sender	Receiver	Normal	Late Send	Late Rcvc	Late Send Post	Late Send Wait	Late Rcvc Post	Late Rcvc Wait
<code>copy_faces+4268</code>	<code>copy_faces+3980</code>	2	0	0	34	0	0	1572
<code>x_solve+7716</code>	<code>x_solve+888</code>	0	0	0	1600	0	0	4

Figure 9: Example of sender-receiver classification summary for NAS SP.

5 EVALUATION

We focus our evaluation on the three primary goals: data reduction, portability, and accuracy. We measure data reduction by comparing our classification information to raw trace file size. We subjectively gauge accuracy by investigating the underlying causes of the abnormal message behavior in the application. If the classification system misidentified the message behavior, we provide an explanation when possible. This is exactly the same procedure that a user would follow to examine application performance. Our choice of decision trees as the classification mechanism simplifies this evaluation.

5.1 Hardware and Software Configurations

To evaluate our classification system, we used an IBM SP2. Each node had 2GB main memory and 4 processors. Our experiments used the IBM’s native 32-bit MPI implementation. We performed several tests with two different software configurations for the MPI communication subsystem: IP mode and US mode. IP mode uses standard Internet protocol for messages, while US mode exploits the SP’s high-performance switch. Like most platforms, the SP2 has dozens, if not hundreds, of configuration parameters that directly impact performance. Our experiments revealed that

Application	Trace File Size (MB)	Number of Trace Records	Normal Runtime (seconds)	MPI_Send	MPI_Rcvc	MPI_Isend	MPI_Ircvc
CG-HEAT	2.7	39698	70	6828	148	0	6680
NAS BT	5.6	88522	1344	0	0	9672	9672
NAS SP	1.6	31394	840	0	0	19272	19272
sPPM	0.2	2066	287	0	0	480	480

Table 3: Profiles of example applications.

US mode performs better than IP mode for all message sizes up to the maximum size for our tests, 4MB. This simple configuration change in software invalidates techniques that rely on fixed thresholds for analyzing performance. These differences underscore the need for a performance analysis framework that adapts to the target platform configuration to improve accuracy. We return to this subject later in Section 5.4.

5.2 Applications

We selected two benchmarks and two scientific applications for our experiments. As described earlier, we ran these applications on the SP2 and collected trace files of their message passing activity. Table 3 lists the applications and their basic characteristics: 4-processor runtime, trace files size, number of trace records, and number of send and receive operations. Since these codes are reasonably mature, we expect them to be highly

Application	Sender Location	Receiver Location	Class						
			normal	late send	late rcv	late send post	late send wait	late rcv post	late rcv wait
NAS BT	copy_faces+4056	copy_faces+3636	795	0	0	2	0	11	0
	copy_faces+4000	copy_faces+3696	0	0	0	755	0	0	53
	copy_faces+4168	copy_faces+3756	2	0	0	22	0	0	784
	copy_faces+4112	copy_faces+3816	3	0	0	6	0	0	799
	copy_faces+4280	copy_faces+3876	4	0	0	35	0	0	769
	copy_faces+4224	copy_faces+3936	3	0	0	27	0	0	778
	x_send_solve_info+612	x_receive_solve_info+124	804	0	0	0	0	0	0
	x_send_backsub_info+432	x_receive_backsub_info+120	0	0	0	28	162	0	614
	y_send_solve_info+608	y_receive_solve_info+124	804	0	0	0	0	0	0
	y_send_backsub_info+420	y_receive_backsub_info+120	0	0	0	35	208	0	561
	z_send_solve_info+908	z_receive_solve_info+124	804	0	0	0	0	0	0
z_send_backsub_info+348	z_receive_backsub_info+120	0	0	0	21	183	0	600	
9672			3219	0	0	931	553	11	4958
NAS SP	copy_faces+4100	copy_faces+3680	1575	0	0	6	0	27	0
	copy_faces+4044	copy_faces+3740	0	0	0	1474	0	1	133
	copy_faces+4212	copy_faces+3800	6	0	0	22	0	0	1580
	copy_faces+4156	copy_faces+3860	3	0	0	17	0	0	1588
	copy_faces+4324	copy_faces+3920	0	0	0	33	0	0	1575
	copy_faces+4268	copy_faces+3980	2	0	0	34	0	0	1572
	x_solve+7716	x_solve+888	0	0	0	1600	0	0	4
	x_solve+17864	x_solve+8272	0	0	0	1604	0	0	0
	y_solve+7632	y_solve+872	0	0	0	1602	0	0	2
	y_solve+17236	y_solve+8180	0	0	0	1604	0	0	0
	z_solve+7612	z_solve+912	0	0	0	1603	0	0	1
z_solve+17224	z_solve+8184	0	0	0	1604	0	0	0	
19272			1586	0	0	11203	0	28	6455
CG-HEAT	snd_int+80	rcv_int+128	141	3	4	0	0	0	0
	snd_r8+80	rcv_asynch_r8+136	1396	2321	0	4	0	819	2140
6828			1537	2324	4	4	0	819	2140
sPPM	xbdrys+1248	xbdrys+152	40	0	0	0	0	0	0
	xbdrys+1184	xbdrys+212	40	0	0	0	0	0	0
	xbdrys+6028	xbdrys+288	29	0	0	0	10	0	1
	xbdrys+5960	xbdrys+356	6	0	0	12	12	0	10
	ybdrys+1248	ybdrys+152	40	0	0	0	0	0	0
	ybdrys+1184	ybdrys+212	40	0	0	0	0	0	0
	ybdrys+6588	ybdrys+288	31	0	0	0	9	0	0
	ybdrys+6520	ybdrys+356	8	0	0	10	9	0	13
	zbdrys+6268	zbdrys+288	19	0	0	0	19	0	2
	zbdrys+6200	zbdrys+356	14	0	0	16	5	0	5
	zbdrys+10920	zbdrys+432	33	0	0	0	7	0	0
zbdrys+10852	zbdrys+500	12	0	0	17	4	0	7	
480			312	0	0	55	75	0	38

Table 4: Results of classifications on example applications.

optimized. Currently, we are applying our tool to applications in earlier stages of development.

5.2.1 NAS SP and BT

The benchmark applications NAS SP and BT [2] represent computational fluid dynamics (CFD) applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of the Navier–Stokes equations. The SP and BT algorithms have a similar structure; each solves three sets of uncoupled systems of equations. BT solves block-tridiagonal systems of 5x5 blocks; SP solves scalar pentadiagonal systems resulting from full diagonalization of the approximately factored scheme.

NAS BT, as Table 4 shows, has seven subroutines that communicate using MPI. The routine `copy_faces` exchanges boundary values between neighboring nodes. The routines `{xyz}_send_solve_info`, `{xyz}_recv_solve_info`, `{xyz}_send_backsub_info`, and `{xyz}_recv_backsub_info` form the core of the algorithm. Each routine performs line solves in the XYZ direction by first factoring the block-tridiagonal matrix into an upper triangular matrix, and then performing back substitution to solve for the unknown vectors of each line.

Of the 9672 message transfers in BT, 3219 appear normal. The other two major classes are late send post (931) and late recv wait (4958). Most of the late recv waits occur in the last five of six transfers. Interestingly, the structure of this routine issues all six `MPI_recv`s immediately before all six `MPI_send`s. The routine then uses a `MPI_Waitall` function to complete these twelve operations. The first transfer is instantaneous, but then the following transfers appear to complete quickly. This strategy overlaps multiple communication operations, but it does not overlap any communication with computation. Both before and after this cluster of communication, BT uses deeply nested loops to pack and unpack the communication buffers. Decomposing these loops to take advantage of communication overlap could improve the communication performance, but risks disabling compiler loop optimizations and disrupting efficient access to the memory hierarchy.

The routines `{xyz}_send_solve_info` and `{xyz}_recv_solve_info` appear normal, but `{xyz}_send_backsub_info` and `{xyz}_recv_backsub_info` often have late receive waits. Closer investigation of these routines revealed that they are part of a loop that computes and then communicates for each cell allocated to the MPI task. This compartmentalization of the messaging activity improves the software design of the code, but it also limits the possible opportunities for overlap of computation and communication.

As expected, the results for NAS SP are strikingly similar to NAS BT. The first message transfer of the `copy_faces` routine is normal, but the succeeding transfers are consistently classified as late recv wait. The classifications also reveal the different structure of the solver in the results for `{xyz}_solve`. The majority of message transfers for this subroutine are classified as late send post. With this evidence, a user could try to move the send's initiation forward in the control flow of the solver. Upon further examination, this solver does overlap communication and computation; however, the intricate code for this solver separates the respective sends and receives with data dependencies that prohibit shifting the location of the send initiation.

5.2.2 CG-Heat

CG-Heat is a heat diffusion simulation that solves the implicit diffusion partial differential equation (PDE) using a conjugate gradient solver over each timestep. This code is built on FORTRAN90 and MPI. CG-HEAT has a grid structure and its data access methods are designed to support one type of adaptive mesh refinement (AMR), although the benchmark code as supplied does not handle anything other than a single-level AMR grid.

Of the 6282 messages transferred by CG-HEAT, 1537 are normal while the majority of classifications fall between late send and late recv wait. Unfortunately, our analysis of CG-HEAT reveals that its MPI communication subroutines are encapsulated within FORTRAN subroutines. This design limits the usefulness of our strategy for locating the application function responsible for particular message transfers; however, it does provide some insight into CG-HEAT's software design. Only two user routines call `snd_r8` and `rcv_asynch_r8`. Both of these routines, `faceget` and `faceput`, exchange information with neighbor tasks. As with the earlier inspection of NAS BT, close examination of CG-HEAT shows its communication primitives buried in three levels of abstraction to improve portability and comprehension. Nonetheless, this abstraction makes it impractical to restructure the code to overlap this communication with nearby computation.

5.2.3 sPPM

The sPPM application [15] solves a 3D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the Piecewise Parabolic Method. The algorithm makes use of a split scheme of X, Y, and Z Lagrangian and remap steps, which are computed as three separate sweeps through the mesh per timestep. Message passing provides updates to ghost cells from neighboring domains three times per timestep. The sPPM code is written in Fortran 77 with some C routines. The code uses the asynchronous message operations: `MPI_Irecv` and `MPI_Isend`.

The results for sPPM are reasonably well balanced. The classifier judges 312 of the 480 (65%) messages as normal. After inspecting the code, we found that sPPM automatically creates the routines `{xyz}bdrys` from a skeleton file and consequently, they have identical sender-receiver structure. Our classifications exhibit this regularity as well; the very last send-recv pair performs poorly when compared to the other three message transfers. These classifications make sense when one reviews the code structure of `{xyz}bdrys`. The code posts all `MPI_recv`s immediately, and stages the send initiations in four steps. Between consecutive sends, the routine packages the next send's buffer with deeply nested loops.

5.3 Data Reduction and Portability

Besides accuracy, data reduction and portability are two additional goals of our approach. Numerous researchers have developed techniques to reduce the volume of information that a user must encounter while investigating and navigating a tracefile [19]. Measuring the data reduction gained with these other techniques is relatively easy because they sift the trace records, keeping only those records that pass some criteria. Furthermore, most of these trace reduction techniques do not consider the chore of mapping these results back to source code. Our technique, on the other hand, transforms the trace file from a series of trace records into classified sender-receiver pairs, so a mathematical calculation of the information reduction for this transformation is awkward. Subjectively however, we consider the reductions of

information to Table 4; Table 3 presents the size of the trace file and the number of trace records in the file. Although these trace files are relatively small because they are example executions on only a 4-node SP2, our ideas and technique remain valid for larger configurations. NAS BT, for example, has 9672 messages and 88522 trace records. Our classification technique transforms all these records into seven classes across twelve sender-receiver locations in the source code as Table 4 shows. A user can quickly review this information and then, investigate modifications to the source code.

Our work is highly extensible and portable. That is, to extend the classes of recognizable performance phenomena, users need only construct a microbenchmark that exhibits the problem, trace the microbenchmark, and regenerate the tests for the decision tree. It is unclear how a user would extend these other tools to recognize new classes of performance problems. As to portability, our approach is extremely portable because it adapts to target software and hardware configurations by observing communication operations empirically and by generating a new decision tree.

5.4 Evaluation Summary

Our experience with these four applications was promising. In each case, our classification system summarized a huge amount of information so that we could immediately investigate each application. With the location information and the predominant classification of the send-receiver pair, we could quickly determine if we could modify the communications operations to improve performance. In light of these benefits, several issues linger. In the case of CG-HEAT, our location-mapping technique did not provide enough information to distinguish the calling user subroutines. Also, our location mapping technique currently marks MPI calls using only subroutine names with a byte offset. For usability, our system should map the performance information directly to lines of source code.

6 RELATED WORK

Researchers have proposed numerous techniques and tools for performance analysis. Although many of these efforts have focussed on the underlying instrumentation and data collection frameworks for performance analysis, only recently have researchers addressed the higher-level goal of automating the task of performance analysis, per se.

Many performance analysis techniques provide some level of help for users to locate performance problems including Carnival [13], a performance debugger by Rajamony and Cox [17], the Paradyn performance tool [14], Tmon [11], Quartz [1], performance indices by Sarukkai, Yan, and Gotwals [20], and Cray's MPP Apprentice and ATExpert. In addition, numerous researchers have applied statistical techniques to performance data in an effort to reduce data volume or to automate tasks for the user. These techniques include covariance analysis, discriminant analysis, principle component analysis, and clustering analysis [3, 19]. In the knowledge discovery field, Lee, Stolfo, and Mok [12] have focused techniques for machine learning on traces of Internet network activity to provide automated support for intrusion detection in computer networks. To assist with the analysis of large trace files, numerous researchers have investigated visualization techniques [10, 23]. Visualizations provide users with clever images of their application execution, but, in general, these tools do not necessarily guide the user to performance

problems.

In contrast to this previous work, the novelty of our approach is the use of a portable, extensible technique to provide automated guidance to users, so that they can easily reason about the underlying causes of their application's communication inefficiencies. We do not claim a novel approach to instrumentation or application tracing. We know, however, of no other work that applies these types of classification techniques to performance analysis. Our methodology not only locates and classifies performance problems, but it also prescribes a strategy for eliminating inefficiencies.

7 CONCLUSIONS

Our overall goal was to provide users with automated guidance for discovering performance problems in their applications. From the outset, we wanted to focus on technologies that were portable and extensible, because MPI exists on many platforms and has many different implementations and configuration parameters. Extensibility offers users the opportunity to refine the classification scheme in a fashion that suites their programming requirements.

Our method automatically classifies individual communication operations and it reveals the cause of communication inefficiencies in the application. This automation allows the developer to focus quickly on the culprits of truly inefficient behavior, rather than manually foraging through massive amounts of performance data. Our technique traces the message operations of MPI applications and then, classifies each individual communication event using a supervised learning technique: decision tree classification. We train our decision tree using microbenchmarks that demonstrate both efficient and inefficient MPI behaviors. Because our technique learns normal MPI behavior, it can adapt to the target system's configuration; this improves the technique's predictive accuracy. Our experiments on four applications demonstrated that this technique improves the accuracy of performance analysis, and dramatically reduces the amount of data that users must encounter.

Several goals remain. We plan to add hardware performance information to our tracing system, and then integrate that information into the classification framework. This additional information, such as CPU busy-idle ratios, could improve the technique's predictive capabilities. Also, we are designing a runtime version of the decision tree classification system. Such a system would help reduce data volume at runtime and thereby decrease the size of the resulting trace files and the perturbation on the target system.

ACKNOWLEDGEMENTS

This paper has benefited from the detailed comments of the ICS reviewers, and my colleagues at LLNL: Mary Zosel, John May, Bronis de Supinski, Alane Achorn, Terence Critchlow, and Chandrika Kamanth.. This work was performed under the auspices of the U.S. Dept. of Energy at LLNL under contract W-7405-Eng-48. LLNL Document Number UCRL-JC-136200.

REFERENCES

- [1] T.E. Anderson and E.D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," Proc. 1990 SIGMETRICS Conf. Measurement and Modeling Computer Systems, 1990, pp. 115-25.
- [2] D. Bailey, E. Barszcz et al., "The NAS Parallel Benchmarks (94)," NASA Ames Research Center, RNR Technical Report RNR-94-007, 1994.
- [3] M. Calzarossa, L. Massari et al., "Medea: A Tool for Workload Characterization of Parallel Systems," *IEEE Parallel & Distributed Technology*, 3(4):72-80, 1995.
- [4] U.M. Fayyad, G. Piatetsky-Shapiro et al., Eds., *Advances in knowledge discovery and data mining*. Menlo Park, CA: AAAI Press: MIT Press, 1996, pp. xiv, 611.
- [5] I. Foster, *Designing and building parallel programs: concepts and tools for parallel software engineering*. Reading, MA: Addison-Wesley, 1995.
- [6] I. Foster and C. Kesselman, Eds., *The Grid: blueprint for a new computing infrastructure*. San Francisco: Morgan Kaufmann Publishers, 1999, pp. xxiv, 677.
- [7] J.A. Gannon, K.J. Williams et al., "Using perturbation tracking to compensate for intrusion in message-passing systems," Proc. 14th Int'l Conf. Distributed Computing Systems, 1994, pp. 414-21.
- [8] G.A. Geist, M.T. Heath et al., "A Users' Guide to PICL - A Portable Instrumented Communication Library," Oak Ridge National Laboratory, P.O.Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-8083 1991.
- [9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.
- [10] M.T. Heath, A.D. Malony, and D.T. Rover, "Parallel performance visualization: from practice to theory," *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):44-60, 1995.
- [11] M. Ji, E.W. Felten, and K. Li, "Performance Measurements for Multithreaded Programs," Proc. 1998 ACM Int'l Conf. Measurement and Modeling of Computer Systems, SIGMETRICS 98, 1998, pp. 161-70.
- [12] W. Lee, S. J.Stolfo, and K. W.Mok, "Mining in a data-flow environment: experience in network intrusion detection," Proc. Fifth ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining, 1999, pp. 114-24.
- [13] W. Meira, Jr., T.J. LeBlanc, and A. Poulos, "Waiting Time Analysis and Performance Visualization in Carnival," Proc. ACM SIGMETRICS Symp. on Parallel and Distributed Tools, 1996, pp. 1-10.
- [14] B.P. Miller, M.D. Callaghan et al., "The Paradyn parallel performance measurement tool," *IEEE Computer*, 28(11):37-46, 1995.
- [15] A.A. Mirin, R.H. Cohen et al., "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," Proc. SC99, 1999.
- [16] J.R. Quinlan, *C4.5: programs for machine learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.
- [17] R. Rajamony and A.L. Cox, "Performance debugging shared memory parallel programs using run-time dependence analysis," *Performance Evaluation Review (Proc. 1997 ACM Int'l Conf. Measurement and Modeling of Computer Systems, SIGMETRICS 97)*, 25(1):75-87, 1997.
- [18] D.A. Reed, R.A. Aydt et al., "An Overview of the Pablo Performance Analysis Environment," Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, IL 61801 1992.
- [19] D.A. Reed, O.Y. Nickolayev, and P.C. Roth, "Real-Time Statistical Clustering and for Event Trace Reduction," *J. Supercomputing Applications and High-Performance Computing*, 11(2):144-59, 1997.
- [20] S.R. Sarukkai, J. Yan, and J.K. Gotwals, "Normalized performance indices for message passing parallel programs," Proc. 8th ACM Int'l Conf. Supercomputing, 1994, pp. 323-32.
- [21] S. Shende, A.D. Malony et al., "Portable profiling and tracing for parallel, scientific applications using C++," Proc. SIGMETRICS Symp. Parallel and Distributed Tools (SPDT), 1998, pp. 134-45.
- [22] M. Snir, S. Otto et al., Eds., *MPI--the complete reference*, 2nd ed. Cambridge, MA: MIT Press, 1998.
- [23] J. Stasko, J. Domingue et al., Eds., *Software Visualization: Programming as a Multimedia Experience*,. Cambridge, MA: MIT Press, 1998.