Fast Classification of MPI Applications using Lamport's Logical Clocks

Zhou Tong
Department of Computer Science
Florida State University
Tallahassee, Florida, USA
tong@cs.fsu.edu

Scott Pakin, Michael Lang
Computer, Computational, and Stat. Sci. Div.
Los Alamos National Laboratory
Los Alamos, NM
pakin, mlang@lanl.org

Xin Yuan
Department of Computer Science
Florida State University
Tallahassee, Florida, USA
xyuan@cs.fsu.edu

***Abstract*—We present a novel trace-based analysis tool that rapidly classifies an MPI application as bandwidth-bound, latency-bound, load-imbalance-bound, or computation-bound for different interconnection networks. The tool uses an extension of Lamport's logical clock to track application progress in the trace replay. It has two unique features. First, it predicts application performance for many latency and bandwidth parameters from a single replay of the trace. Second, it infers the performance characteristics of an application and classifies the application using the predicted performance trend for a range of network configurations instead of using the predicted performance for a particular network configuration. We describe the techniques used in the tool and its design and implementation, and report our performance study of the tool and our experience with classifying nine applications and mini-apps from the DOE Design Forward project as well as the NAS Parallel Benchmarks.**

***Keywords*-Parallel application; performance analysis; tool**

I. Introduction

The performance of message-passing applications is often studied through application tracing and trace replay and analysis. Tracing tools such as the DUMPI library [5] have been developed for trace collection; and tools such as Scalasca [6] and the Structured Simulation Toolkit (SST) [1] have been implemented for trace analysis. These tools enable the application and the system to be studied in great detail by replaying the collected traces and simulating the application and the underlying architecture. Simulating an application on a large-scale system can take a significant amount of time and computing resources but can be useful for understanding the performance characteristics of the application on the simulated platform. Such studies, however, offer little insight into how an application will perform when run on a different system.

We present a new trace-based performance analysis tool for fast classification of MPI [18] applications. Unlike conventional trace-based performance analysis tools that focus on uncovering the application performance characteristics for a particular system configuration, our tool can rapidly reveal the performance characteristics when the application runs on many different interconnection networking technologies. In our tool, each interconnect technology is represented by two terms, bandwidth and latency, measured by commonly-used methods such as netperf or Ping Pong [17].

Our tool classifies an application as bandwidth-bound, latency-bound, communication-bound (both bandwidth- and latency-bound), load-imbalance bound, and computation-bound for different latencies and bandwidths by *replaying the application trace only once*. The tool is implemented in MPI and incurs less communication in the trace replay than the original MPI program. It is at least as scalable as the MPI program under study. The tool can be used by application developers to rapidly forecast communication-related performance bottlenecks and by system designers to quickly gauge the relative benefits of various networking options. It thereby complements slower but more accurate approaches such as cycle-accurate simulation.

The tool uses an extension of Lamport's logical clock [13] to keep track of time progress in the trace replay. Such a logical clock not only maintains the happened-before relationship but also tracks the predicted application execution time. The tool maintains multiple sets of logical clock counters, one set for each network configuration. Each set of counters record different types of predicted times including computation time, wait time, latency time, and bandwidth time of the application for the particular interconnect configuration. During the one pass of the trace replay, logical clocks for all network configurations are maintained simultaneously and all of the counters are computed. In other words, by simulating the application once, the predicted execution time for many interconnect configurations is obtained. With the predicted execution times for carefully selected network configurations, application classification is performed based on the performance trend across a range of interconnect configurations instead of the predicted performance for a particular interconnect configuration. This approach can tolerate the inaccuracy in the prediction for individual network configuration and make the classification robust since the sensitivity of an application to latency, bandwidth, or load imbalance can be easily observed from the performance trend over a range of network configurations.

We will discuss the techniques used and describe the design and implementation of the tool. We will then validate the tool, report the performance of the tool, and describe our experience with using the tool to classify nine Department of Energy (DOE) applications and miniapps in the Design Forward project as well as the NAS parallel benchmarks.

II. Related work

Performance analysis has always been a critical component of high-performance computing (HPC). A large number of profiling, trace-collection, simulation, analysis, and visualization tools have been developed. Many tracing tools such as DUMPI [5] and the Intel Trace Analyzer and Collector [9] are available for collecting traces. These traces can be analyzed by various trace analysis/replay tools such as the Structured Simulation Toolkit (SST) [1] and BigSim [20]. In addition, compression techniques can further reduce the volume of communication traces while preserving structural information for replay as in ScalaTrace [15].

These performance analysis infrastructures usually build upon a low level event-driven interconnect simulator such as BookSim [11] and ROSS [3]. Other tools including Scalasca [6] and DIMEMAS [7] provide performance prediction and analysis with source-level instrumentation and MPI routines with extensive execution time. In particular, Scalasca's trace-replay analysis enables automatic wait-state search and identifies the root cause of parallelization bottlenecks such as load imbalance. These end-to-end simulation tools can simulate the behavior of a full application on a target system and generate performance summaries. Furthermore, the performance data can be visualized by tools such as Vampir [14], Paraver [12], Jumpshot [4], and Ravel [10], the latter harvesting communication pattern based on logical happen-before relationships. Our tool is different from these existing tools in that it does not focus on obtaining detailed performance for a particular system. Rather, it tries to rapidly uncover the performance characteristics of an application by quantifying its sensitivity to increases and decreases in bandwidth and latency. It is therefore similar in that regard to analytical performance modeling [2] but represents an automated rather than a manual approach.

III. Fast Classification Tool

A. Overview

The tool takes DUMPI [5] traces of an application as input, simulates the events in the traces, and classifies the application for different interconnect technologies. Our techniques can be applied to any MPI tracing format. We select the DUMPI format because it is widely used in DOE laboratories; and DUMPI traces for many applications and mini-apps are available from the DOE Design Forward Web site.[1] One limitation of DUMPI traces is that they lack information about the machine topology, task mapping, and anything else below the MPI level.

Each DUMPI trace file records all MPI events in one rank of the application. In replaying the traces, our simulation tool creates one MPI process for each DUMPI trace file. An extension of Lamport's logical clock scheme [13] is used to keep track of the progress of the processes in the simulation of the application. Multiple sets of counters are maintained to record the predicted time for different network configurations. For each network configuration, four counters, *computation time*, *wait time*, *bandwidth time* and *latency time*, are used to record the different types of time that a process experiences. For the communication times (wait time, bandwidth time, and latency time), only the time that contributes to the process execution time is recorded. The communication time that overlaps with computation time is excluded. For example, if a message is sent and delivered before the receiver enters the MPI_Recv operation, there is no wait, bandwidth, or latency time experienced at the receiver. The times recorded in the four counters are defined as follows:

1) *Computation time*: the time consumed by computation or local (non-communicating) MPI routines. Computation time is measured as the gap from the exit time of an MPI routine to the entry time of the next MPI routine as well as the gap from the entry time of a local MPI routine to the exit time of that routine. Note that the enter and exit time of each MPI routine is recorded in the DUMPI trace.
2) *Wait time*: the time spent waiting for a corresponding party to start its communication. It occurs in both point-to-point and collective MPI routines. For point-to-point communication, the receiver may be blocked waiting for the sender to start the communication. For a collective communication, a process may be blocked waiting for the last process in the operation to enter the collective communication.
3) *Latency time*: the fixed latency for the first byte of a message to reach the receiver, independent of message size. The latency time counter records the total latency time that is not overlapped with computation.
4) *Bandwidth time*: the time needed for a message to reach the receiver, starting from when the first byte arrives. It is measured as a per-byte time multiplied by the number of bytes transmitted. The bandwidth time counter records the total bandwidth time that is not overlapped with computation.

To investigate the performance characteristics of an application and classify the application for a particular interconnect technology, a set of network configurations is carefully selected with the four counters for each configuration. The performance statistics across the set of configurations is then used to classify the application for the targeted interconnect technology. In the text that follows we discuss the details of the techniques used in the tool including how the communication is modeled, how the logical clock is maintained for each configuration, how the counters are computed, how the predicted time for multiple network configurations are progressed simultaneously, how to select network configurations for a targeted interconnect technology, and how the classification is performed based

[1] http://portal.nersc.gov/project/CAL/designforward.htm

on the performance statistics across the configurations.

B. Modeling communication

The tool currently supports two levels of communication: intra-node and inter-node. For each type of communications the tool supports two models of point-to-point communication: a more accurate table look-up based model that provides more accurate estimation about the communication time for a given system, and a simple Hockney's model [8] where an n-byte message takes $\alpha+n\beta$ time to complete, where α is the communication latency (the time to communicate a zero byte message) and β is the per-byte cost (reciprocal bandwidth).

A collective operation is modeled as a global synchronization for all processes to be ready for the operation, and then Thakur and Gropp's models [19] are used to estimate the time for different collective operations. For example, in collective operations such as MPI_Bcast and MPI_Reduce, the communication time is modeled as $(\alpha+n\beta)\log_2 P$, where $\alpha\log_2 P$ is the latency time and $n\beta\log_2 P$ is the bandwidth time.

In the rest of the paper, we will assume Hockney's model to ease exposition. We note that these models are not entirely accurate. However, because our tool classifies an application based on its performance on a range of interconnect configurations, even when each individual configuration is not modeled accurately, the classification can still be quite robust.

C. Lamport's logical clock

Lamport's logical clock [13] provides a clock synchronization mechanism that honors the happen-before relationship. Each process represents time with a local counter. Before an "event" (computation or a communication operation), the process increments its counter. Counters are transmitted with each message. A receiver sets its counter to the maximum of its current value and one unit of latency plus the value received. Figure 1 illustrates how a logical clock enforces the ordering of three causally-related events (a, b, and c) across three processes (P_0, P_1, and P_2). That is, an event that happens before another event will have a smaller timestamp.

P_1 sends message (a) at time 2, and it is received at P_0 at time $\max(3,2+1)=3$. P_2 sends message (b) also at time 2, but because P_0 already observed a few events, it is not received until time $\max(5,2+1)=5$. Although P_1 is only at time 4 when it receives P_0's message, it advances its counter to time $\max(3,6+1)=7$ because P_0 sent the message at time 6.

D. Maintaining the extended Lamport's logical clock

Our tool extends Lamport's logical clock to not only preserve causal relationships, but also maintain predicted application time by incorporating non-unit computation time and non-unit communication time.

There is one logical clock for each network configuration. The logical time for each event represents the predicted time for the event for the given network configuration. The tool

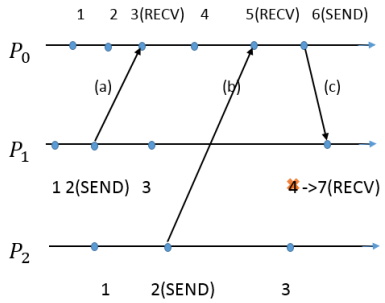


Figure 1: Example of Lamport's logical clock

either assumes a global clock that is synchronized across all ranks or specifies a starting event to start the logical clock and synchronize the physical clock, which is usually a globally synchronized collective operation such as an MPI_Barrier. The tool focuses primarily on communication performance and by default assumes that computation time is the same as reported by the traces. While not evaluated in this work, the tool can also scale computation time evenly to mimic faster or slower processors.

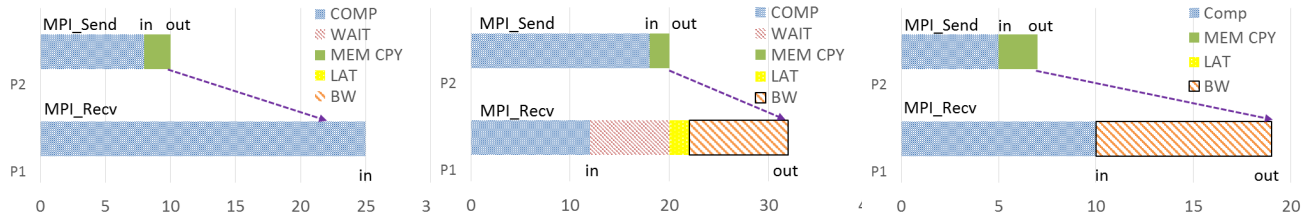
Let there be N communication events in a trace numbered from 0 to $N-1$. We denote r^i_{in} to be the physical entry time of the ith event, $0 \le i < N$, logged in the DUMPI trace, r^i_{out} to be the physical exit time, t^i_{in} to be the logical entry time, and t^i_{out} to be the logical exit time. The tool maintains the logical clock and calculates t^i_{in} and t^i_{out} for all events in the DUMPI trace. The logical entry time is calculated as follows:

$$t^0_{in}=0 \text{ and } t^i_{in}=t^{i-1}_{out}+r^i_{in}-r^{i-1}_{out}.$$

Basically, we initialize the starting logical time to be 0 and the logical entry time of an event is the logical exit time of the previous event plus the computation time between the two events. This allows the logical clock to keep track of the computation time of the application. The logical exit time of an event depends on the type of events. For an MPI non-blocking operation such as MPI_Isend, the operation is treated as computation and the logical exit time is the logical entry time plus the time in the event:

$$t^i_{out}=t^i_{in}+r^i_{out}-r^i_{in}.$$

For an MPI point-to-point blocking send operation, the exit time may be affected by the MPI protocol implemented. In most MPI implementations, point-to-point communication is realized by two protocols: an eager protocol for small messages and a rendezvous protocol for large messages. With an eager protocol, the sender copies the message into a system buffer and exits the operation while the MPI library's progress engine asynchronously transfers the message to the receiver. With a rendezvous protocol, the sender has to wait for an acknowledgment from the receiver before initiating data transfer. Consequently, the rendezvous protocol introduces an additional dependence from the receiver to the sender, which restricts our tool to processing only one



(a) Early Sender (b) Early Receiver (c) Concurrent Sender & Receiver

Figure 2: Counting wait time, latency time, and bandwidth time in a point-to-point communication

configuration at a time, thereby greatly limiting its utility. We design our tool to be MPI-implementation oblivious. Because the eager protocol better reflects the inherent communication characteristics (there is a dependence from the sender to the receiver but not the other way around), communication assumes an eager protocol by default although the rendezvous protocol is also supported by the tool.

With the eager protocol, the blocking send operation completes when the message is copied into system memory:

$$t^i_{out} = t^i_{in} + memcopytime(msg\_size).$$

For a blocking, point-to-point receive operation (e.g., MPI_Recv or MPI_Wait), the logical exit time will also depend on the time for the matching send when the communication starts. Call the logical entry time for the matching send operation T. The logical exit time for the blocking wait is therefore

$$t^i_{out} = \max(t^i_{in}+1, T+\alpha+n\beta).$$

In the formula, $\alpha+n\beta$ is the time to transfer the message from the sender to the receiver. This calculation captures both the happens-before relation and the predicted communication time. In the implementation, in order for the receiver to obtain the value of T (the logical time for the matching send event) in the trace replay, each send event transmits its logical timestamp to the receiver, and each blocking receive event performs the matching receive and updates its logical clock based on the timestamp received. One shortcoming of our trace-replay tool is that it preserves the original message order that appears in the trace, even when MPI_ANY_SOURCE is used.

We treat MPI_Waitall as an array of blocking wait requests. The simulated process is blocked until all matching messages are received and requests are completed. For instance, if there are k wait requests, each one estimates the logical exit time for an individual blocking wait. The maximum logical exit time of all blocking waits is the exit time of MPI_Waitall:

$$t^i_{out} = \max(t^{i,j}_{out}), j \in [1,k].$$

If the ith event is a collective operation, the logical exit time is computed as

$$t^i_{out} = \max(t^c_{in}) + model\_time(n).$$

The term $\max(t^c_{in})$ is the largest logical entry time of the collective operation among all processes. $model\_time(n)$ is the modeled time for the collective operation with message size n as in Thakur and Gropp [19]. In the trace replay, an MPI_Allreduce operation is performed for each collective operation to obtain the largest logical entry time $\max(t^c_{in})$ among all processes. Then, the exit time of each process can be computed locally.

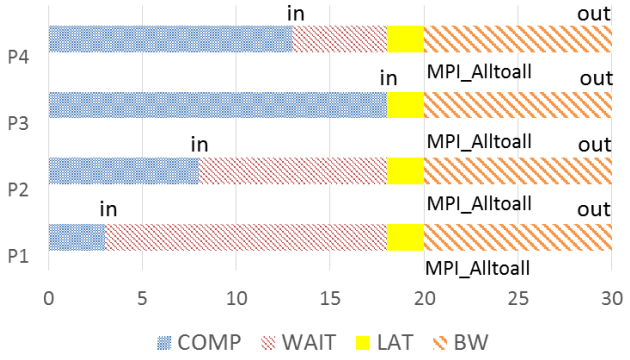


Figure 3: Counting wait time, latency time, and bandwidth time in a collective communication

E. Recording different types of communication times

As discussed earlier, the classification tool maintains four time counters for each network configuration: compute time, wait time, latency time, and bandwidth time. Figure 2 depicts three distinctive scenarios in MPI point-to-point operations. The logical enter times t_{in} of each MPI routine are marked and the exit time t_{out} are estimated based on the expected timing that P_1 received the message. This example illustrates how the four time counters capture both communication dependencies and the predicted communication time. We assume two units of time for latency, ten for bandwidth and two for the memory copy of message into system buffer.

Early Sender: Figure 2a shows that P_2 enters MPI_Send at logical time 8. With the predicted delivery time of 22 (2 units of time for latency and memory copy, 10 units of time for bandwidth). P_1 enters MPI_Recv at logical time 25. Therefore, P_1 experiences zero wait time, latency time, and bandwidth time.

Early Receiver: In Figure 2b, P_1 enters MPI_Recv at logical time 12 while P_2 enters MPI_Send at 18. P_1 spends 8 units of waiting time for P_2 to start sending a message at 20.

The estimated delivery time is 32 at P_1 resulting in 10 units of bandwidth time and 2 units of latency time experienced at P_1.

Concurrent Sender/Receiver: In Figure 2c, P_2 enters MPI_Send at logical time 5 while P_1 is still busy computing until time 10. This leads to 5 units of overlapped computation and communication time. With an estimated delivery time of 19, P_1 experiences no wait time, no latency time, and 9 units of bandwidth time (latency time is counted before bandwidth time). In our model, computation that overlaps communication is considered by default to overlap first latency time then bandwidth time. This can lead to applications being categorized as bandwidth-bound when most or all of the latency is overlapped, even though improvements to latency could still improve overall application performance.

Figure 3 illustrates an example for collective communication. Different processes may start the operation at different times. All processes that arrive before the last one must wait until the last process arrives. After that, the modeled latency and bandwidth time starts to count. In the figure, P_1 starts at time 3. It first waits for 15 units of time (until the last process P_3 arrives at time 18 and then experiences 2 units of latency time and 10 units of bandwidth time.

F. Progressing logical times for multiple network configurations

With the default eager protocol for point-to-point communication, there are only two situations in which the replay of a process can be blocked: blocking receive operations and collective operations. In the blocking receive operation a process must receive a logical timestamp from the matching sending process in order to determine the logical exit time for the blocking receive operation. In a collective operation, each process must perform an MPI_Allreduce operation to obtain the largest logical enter timestamps from all processes. Since the traces are generated for a correctly executed MPI program: a blocking receive operation always has a matching send operation, and all collective operations are called by all participating processes with no point-to-point routine across the collective operation boundary: all point-to-point communications before a collective operation must be completed before the collective operation is performed. Hence, in the trace replay, a blocking receive operation guarantees to receive the timestamp from its matching send operation without causing deadlock. Similarly, in the replay of a collective operation, the all-gather operation is guaranteed to be successful and the logical exit time of the collective operation guarantees to be updated. This is independent of the network configuration to be simulated. Hence, the tool can maintain any number of logical clocks (for any number of different network configurations) and communicate and update all of the logical timestamps for each communication event in a single pass. This ability to predict the performance for many network configurations is the key for fast classification of applications.

Table I: Interconnect in current production systems.

Configurations	Latency (μs)	BW (Gbps)
1G Ethernet (E1G)	50	1
10G Ethernet (E10G)	5	10
InfiniBand QDR (QDR)	1.3	32

G. Selecting network configurations and classification

Each interconnect technology is characterized by its bandwidth (BW) and latency (L). We will denote the interconnect by (BW, L). The tool has a default setting that supports the investigation of some commonly used interconnection network technologies, such as 1Gbps Ethernet, 10Gbps Ethernet, and InfiniBand QDR. In Table I, the MPI latency of 1G and 10G Ethernet are the measured minimum latency reported by QLogic [17], and the latency for InfiniBand QDR is as measured by Panda [16].

The tool uses the predicted performance on three sets of network configurations. Each set consists of seven configurations to determine whether an application is bandwidth-bound, latency-bound, communication-bound, load-imbalance bound, or computation bound for the interconnection network (BW,L).

One set of configurations tests the application's sensitivity to bandwidth at the given latency. This set consists of configurations $(BW/8,L)$, $(BW/4,L)$, $(BW/2,L)$, (BW,L), $(2BW,L)$, $(4BW,L)$, $(8BW,L)$. By examining the performance trend for the range of bandwidth from $BW/8$ to $8 \cdot BW$, one can determine whether an application is sensitive to bandwidth changes. Moreover, even though the communication model does not consider network contention, bandwidth-sensitive applications will also be sensitive to contention.

The second set of configurations tests the application's sensitivity to latency. It consists of configurations $(BW,8L)$, $(BW,4L)$, $(BW,2L)$, (BW,L), $(BW,L/2)$, $(BW,L/4)$, and $(BW,L/8)$. The third set tests the application's sensitivity to both bandwidth and latency, consisting of configurations $(BW/8,8L)$, $(BW/4,4L)$, $(BW/2,2L)$, (BW,L), $(2BW,L/2)$, $(4BW,L/4)$ and $(8BW,L/8)$. With the predicted performance on the three sets of configurations, the classification of the application can be done as follows:

C1. **Computation-bound**: Observed computation time accounts for a large percentage of total execution time. In addition, a program's predicted total execution time is insensitive to speed-up or slow-down of bandwidth and latency. The exact threshold values for the large percentage of total time, and the insensitivity are parameters decided by the user. In Section IV, we give examples to show how the threshold values are set.

C2. **Load-imbalance-bound**: The wait time accounts for a significant percentage of the total execution time. In addition, the wait time is insensitive to the speed-up and slow-down of bandwidth and latency. Wait time

can be introduced by computational load imbalance, network contention, system noise, algorithm design, etc.

C3. **Bandwidth-Bound**: Bandwidth bound is related to a given bandwidth (BW). Given BW, communication time (latency time + bandwidth time + wait time) accounts for a significant percentage of the total execution time and is insensitive to latency changes. The communication time decreases significantly as the bandwidth speed increases across the range of BW (from BW/8 to 8BW). Again, the exact threshold values can be set by the user.

C4. **Latency-Bound**: Latency-bound is related to a given latency (L). Given L, the communication time accounts for a significant percentage of the total execution time and is insensitive to bandwidth changes. The communication time decreases significantly as the latency speed improves across L (e.g. from 8L to L/8).

C5. **Communication-bound**: Communication bound is related to a given bandwidth (BW) and a given latency (L). Given BW and L, communication time (latency time + bandwidth time + wait time) accounts for a significant portion of the total execution time. The communication time decreases significantly as the bandwidth speed increases across BW, or as latency time improves across L, or both.

The classification method can be further fine tuned. For example, if one knows that the communication pattern in an application is unlikely causing contention, then targeted (BW,L) can be used as the anchor for observing the performance trend. If however, one knows that the communication in an application has significant contention, then some slower configuration (e.g., half the bandwidth) may be used as the anchor.

IV. Empirical Evaluation

We report the results of various experiments with our fast classification tool. We first briefly describe the set of parallel benchmarks and applications in the experiments. Then, we present our validation results of the tool by showing that our approach can model that application performance accurately when appropriate information is provided. After that, we report the classification results and demonstrate how applications are classified. Finally, we evaluate the simulation performance of our classification tool.

A. Benchmarks & Applications

The parallel programs used in this study include extracted kernels, miniapps and full-sized applications from the Department of Energy (DOE)'s Design Forward project.[2] the extracted kernels used are Big FFT, Crystal Router (CR); the mini-apps used are AMG, MiniFE, and CLAMR; and the full-sized applications are MultiGrid(MG), AMR Boxlib, PARTISN, and FillBoundary (FB). These applications represent the diverse communication patterns observed in DOE's production systems. More information about these programs as well as their DUMPI traces are in the DOE Design Forward project Web site. In addition, we also use the NAS Parallel Benchmarks[3] (NPB). DUMPI traces of the NPB and timing measurements were generated on Cielito, a 64-node (1024-core) Cray XE6 system at Los Alamos National Laboratory (LANL).

[2]http://portal.nersc.gov/project/CAL/designforward.htm

Table II: Predicted and measured communication and total application time (in second) of 64-rank CLAMR, CR and FB on Cielito

	CLAMR		CR		FB		
	eag.	eag.*	eag.	eag.*	eag.	eag.*	rend.
Comp.	0.23	0.23	0.26	0.26	0.86	0.86	0.86
Pred. Comm.	0.76	0.90	0.07	0.05	0.15	0.19	0.35
Act. Comm.	0.89	0.89	0.06	0.06	0.36	0.36	0.36
Comm. Err. %	-14.61	+1.12	+16.67	-16.67	-58.33	-47.22	-2.78
Pred. Tot.	0.99	1.13	0.33	0.31	1.01	1.05	1.21
Act. Tot.	1.12	1.12	0.32	0.32	1.22	1.22	1.22
Pred. Err. %	-11.61	+0.89	+3.13	-3.13	-17.21	-13.93	-0.82

B. Validation

We show that when accurate latency and bandwidth information is available, our classification tool can accurately predict application performance. 64-rank runs of three programs, CLAMR, CR and FB, which represent different types of communications are used in this study. To obtain accurate communication performance information, we used Intel's MPI Pingpong benchmark[4] with two communicating processes mapped onto either the same node or separate ones, over hundreds of repetitions. We built a look-up table that allows for accurate extrapolation of message communication times (latency plus bandwidth times) of any size on this machine. For comparison, the parameters in the Hockney model for Cielito are derived using least-squares fitting from the measured data. Memory copy speed is obtained with a microbenchmark.

Table II presents the predicted and measured communication and total application time as well as the prediction errors. Each column compares the predicted performance with different models with the measured time. For each benchmark, the first column (eag.) compares the predicted performance using the default eager protocol with the Hockney's model; the second column (eag.*) compares the predicted performance using the default eager protocol with the more accurate look-up table. For FB, there is a third column that contains prediction results with the rendezvous protocol. With the default eager protocol and the Hockney's model, the baseline results show that the communication prediction errors of CLAMR, CR and FB with Hockney's model are -14.61%, 16.67% and -58.33%, respectively; and that the total application time prediction errors are -11.61%, 3.13%, and -17.21% respectively. A negative prediction error means under-estimation while a positive prediction error means over-estimation. With the look-up

[3]https://www.nas.nasa.gov/publications/npb.html

[4]https://software.intel.com/en-us/articles/intel-mpi-benchmarks

table, the communication prediction errors of CLAMR, CR, and FB are reduced to 1.12% and -16.67%, -47.22% while the predictions for overall execution time are -0.89%, -3.13%, and -13.93%. FB poses a significantly larger prediction error of -47.22% for communication and -13.93% overall. This discrepancy is caused by the fact that message sizes in FB are up to 4 MB (so the rendezvous protocol dominates the communications in the program) whereas message sizes in CLAMR and CR are under 4 KB. When the model for the rendezvous protocol is assumed, the prediction accuracy improves to -2.78% for communication time and -0.82% overall for FB.

These results indicate that although our tool does not consider network contention, when accurate communication information is used and modeled, the tool can predict the performance of applications fairly accurately, which is more than sufficient to classify applications and understand the performance limiting factors in the applications. In the classification, we use the Hockney's model. Although this model is not as accurate as the look-up table used in the validation, it provides close approximations. Moreover, our tool classifies applications based on the performance trend over a range of network configurations, which alleviates the problems caused by the inaccurate modeling. The inaccuracy raised due to the specific MPI implementations (e.g., eager protocol vs. rendezvous protocol) poses more challenges. We believe that rendezvous protocol is an implementation choice, and does not represent the inherit communication characteristics. We therefore use exclusively the eager protocol in the classification. Note that observing the performance trend with a range of network speeds also alleviates the impact of such inaccuracy.

C. Classification

We classify the applications on three interconnect technologies: 1G Ethernet, 10G Ethernet and InfiniBand QDR. Each network configuration is represented by $(BW, Latency)$ where BW is in the unit of Gbps and $Latency$ is in the unit of microseconds. Memory copy time is computed with the assumption of 32GB/s memory bandwidth. Hence, 1G Ethernet is represented as $(1{,}50)$, 10G Ethernet is represented as $(10{,}5)$, and InfiniBand QDR is represented as $(32{,}1.3)$. The configurations used in the experiments are shown in Table III. Based on the prediction results of these configurations, the applications are classified as follows:

C1. **Computation-bound (Comp.)**: Observed computation time accounts for 90% of total execution time. In addition, a program's predicted total execution time is insensitive (less than 5% difference) to the factor of 8 speed-up or slow-down of bandwidth and latency.

C2. **Load-imbalanced-bound (Imb.)**: The wait time accounts for 25% of the total execution time. In addition, the wait time is insensitive (less than 5% difference) to the factor of 8 speed-up and slow-down of bandwidth and latency.

Table III: Configurations for classification

Model	ENET-1G	ENET-10G	QDR
Latency	(1, 6.25)	(10, 0.625)	(32, 0.1625)
	(1, 12.5)	(10, 1.25)	(32, 0.325)
	(1, 25)	(10, 2.5)	(32, 0.65)
	(1, 50)	(10, 5)	(32, 1.3)
	(1, 100)	(10, 10)	(32, 2.6)
	(1, 200)	(10, 20)	(32, 5.2)
	(1, 400)	(10, 40)	(32, 10.4)
BW	(.125, 50)	(1.25, 5)	(4, 1.3)
	(.25, 50)	(2.5, 5)	(8, 1.3)
	(.5, 50)	(5, 5)	(16, 1.3)
	(1, 50)	(10, 5)	(32, 1.3)
	(2, 50)	(20, 5)	(64, 1.3)
	(4, 50)	(40, 5)	(128, 1.3)
	(8, 50)	(80, 5)	(256, 1.3)
COMM	(0.125, 400)	(1.25, 40)	(4, 10.4)
	(0.25, 200)	(2.5, 20)	(8, 5.2)
	(0.5, 100)	(5, 10)	(16, 2.6)
	(1, 50)	(10, 5)	(32, 1.3)
	(2, 25)	(20, 2.5)	(64, 0.65)
	(4, 12.5)	(40, 1.25)	(128, 0.325)
	(8, 6.25)	(80, 0.625)	(256, 0.1625)

C3. **Bandwidth-Bound (BW)**: For the targeted bandwidth, communication time (latency time + bandwidth time + wait time) accounts for at least 25% of the total execution time. The communication time is insensitive (less than 5% difference) to the factor of 8 latency change. The communication time decreases by a factor of 2 when the bandwidth increases from 1/2BW to 2BW.

C4. **Latency-Bound (Latency)**: Given a latency L, the communication time accounts for at least 25% of the total execution time. The communication time is insensitive (less than 5% difference) to 8X BW. The communication time decreases by a factor of 2 as the latency improves from 2L to L/2.

C5. **Communication-bound (Comm.)**: Given BW and L, communication time (latency time + bandwidth time + wait time) accounts for 25% of the total execution time. The communication time decreases by a factor of 2 as the bandwidth speed increases across BW by a factor of 4, and as latency time improves across L by a factor of 2.

For some applications, the total communication accounts for less than 25%, but more than 10% of the total time. We further classify the applications as load-imbalance-sensitive (Imb.-s), bandwidth-sensitive (BW-s), latency-sensitive (Latency-s), and communication-sensitive (Comm.-s) following the similar definitions as the load-imbalance-bound, bandwidth-bound, latency-bound, and communication-bound.

Table IV summaries classification results of 9 DOE applications and the NAS IS benchmark for 1G Ethernet, 10G Ethernet, and InfiniBand QDR. Traces marked by "†" are collected on Cielito while the rest are from the Design Forward Web site. For NPB, we include only the IS benchmark because traces collected on Cielito for other NAS benchmarks contain external user-defined MPI datatypes that are not

Table IV: Classification results (number of MPI ranks shown in parenthesis)

	ENET-1G	ENET-10G	QDR
AMG(8)	Comp.	Comp.	Comp.
AMG(27)	Imb.-s	Imb.-s	Imb.-s
AMG(216)	Imb.-s	Imb.-s	Imb.-s
AMG(1728)	Imb.	Imb.	Imb.
AMR(64)	Latency-s	Comp.	Comp.
BigFFT(100)	Comm.	Comm.	Imb.
CLAMR(64)†	Latency	Latency	Imb.
CR(64)†	Comm.	Comm.	Comp.
FB(64)†	BW-s	Comp.	Comp.
FB(125)	BW-s	Imb.-s	Imb.-s
MG(1000)	Imb.-s	Comp.	Comp.
MiniFE(1152)	Comp.	Comp.	Comp.
PARTISN(168)	Imb.	Imb.	Imb.
IS(64)†	Comm.	Imb.-s	Imb.-s

recorded in the DUMPI traces. Without knowing message sizes, we cannot correctly model communication time.

With the slow 1Gbps Ethernet, communication has significant impacts on applications. Many programs are communication-, latency-, or bandwidth-bound or sensitive. As the network becomes faster, in InfiniBand QDR, the latency is sufficiently small and bandwidth is sufficiently large such that these programs become computation-bound or load imbalance-bound. AMG, bigFFT, CLARMR, Fillboundary, PARTISN, and IS are either load-imbalance-bound or sensitive for InfiniBand QDR. For instance, PARTISN might be diagnosed with communication issues based on its extensive wait time, but our tool can classify it as load-imbalanced due to its 2D wavefront communication pattern with data dependencies. Improving communication performance in the absence of other program-configuration changes will not improve overall application performance. Instead, improving their load balance property is the key to improving the scalability of these programs on modern HPC systems. We can see that communication as the performance limiting factor is eliminated for some applications as the interconnection network becomes faster. For example, CR(64) is communication-bound on 1G Ethernet, but becomes computation-bound on the faster InfiniBand QDR; FB(64) is bandwidth-sensitive on 1G Ethernet, but becomes computation-bound on InfiniBand QDR. Next, we will use selected results to show how some applications are classified.

Figure 4 shows the sensitivity of MiniFE(1152) to communication (bandwidth and latency) on 10G Ethernet. In this case, the 1X configuration is (10Gbps, 5μs); the X/2 configuration is (5Gbps, 10μs); the X/4 configuration is (2.5Gbps, 20μs); the 2X configuration is (20Gbps, 2.5μs); and so forth. It clearly shows that this program is dominated by computation and is not sensitive to bandwidth

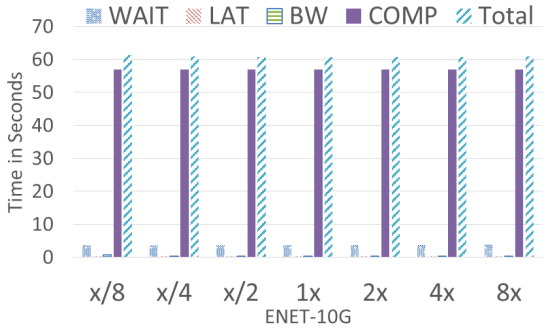


Figure 4: Sensitivity of MiniFE (1152) to communication (bandwidth and latency) on 10G Ethernet

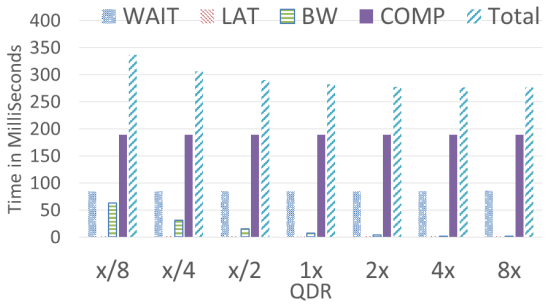


Figure 5: Sensitivity of BigFFT(100) to communication (bandwidth and latency) on InfiniBand QDR

or latency. As a result, MiniFE(1152) is classified as a computation-bound program on 10G Ethernet.

Figure 5 shows the sensitivity of BigFFT(100) to communication (bandwidth and latency) on InfiniBand QDR. We observe that the wait time accounts for 25% of the total execution time. Moreover, the wait time is insensitive (less than 5%) to the factor of 8 speed-up and slow-down of bandwidth and latencies. This is in agreement with the communication pattern of BigFFT(100), which consists of a small number of all-to-all and global barrier operations. This results in a significant amount of time spent waiting for peers to synchronize. The results also show that the program can be sensitive to bandwidth when the bandwidth decreases. At the targeted QDR, the bandwidth time is small. As such, the program is classified as a load-imbalance-bound program for InfiniBand QDR based on our classification criteria. On slower 1G and 10G Ethernet, this program becomes communication-bound. As can be seen from this example, examining the performance trend across the range of network configuration reveals significant performance information about the application.

Figure 6 shows the sensitivity of AMG(216) to communication on 10G Ethernet. This program has about 15% wait time, but little bandwidth and latency time for the range of the network configurations. It is thus classified as load-imbalance-sensitive. This indicates that the program may have scalability issues as the number of ranks increases. Figure 7 shows the performance of AMG with different numbers of ranks (8, 27, 216, 1728) on 10G Ethernet. From

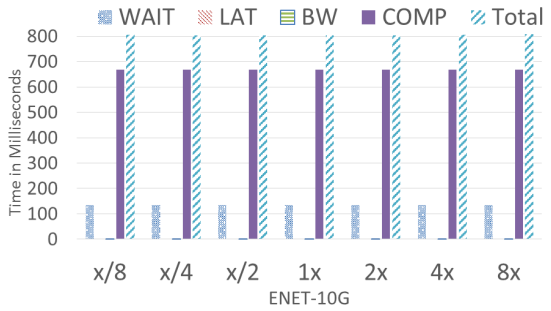


Figure 6: Sensitivity of AMG(216) to communication on 10G Ethernet

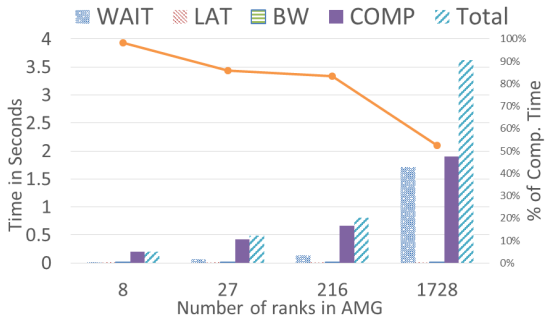


Figure 7: Scalability of AMG on 10G Ethernet

the figure, we can see that the program is dominated by computation for smaller scales as AMG(8) is classified as computation-bound. Moreover, wait time accounts for more than 10% of the total time in AMG(27) and AMG(216), which are classified as load-imbalance sensitive. When the program runs on 1728 ranks, the wait time is about 48% of the total application time, and AMG(1728) is a load-imbalance-bound program for 10G Ethernet.

As can be seen from the experiments, by examining the performance trend for an application on a range of network configurations, important performance characteristics of the application is revealed. The tool is effective in classifying applications even with the inaccuracy in the performance prediction for individual networks.

D. Performance of the classification tool

We study the performance of our classification tool using the NPB with both 64 and 4096 ranks. Note that some of these programs contain communications in which the message sizes are not recorded in the DUMPI traces (so our tool cannot classify the programs correctly). The DUMPI traces of these programs still log all communications in the program, and thus can be used to evaluate the performance of our tool since the traces have all MPI events that are replayed. The 64-rank runs use Class C while the 4096-rank runs use Class D. For this study, both our tool and the original programs are executed on Cielito with the same numbers of ranks per node.

Figure 8 shows the speed-up of simulation time of NPB benchmarks on with 16 ranks running on one node in the

Figure 8: Simulation speedup vs. application runtime of NPB benchmarks (class C, 64 ranks)

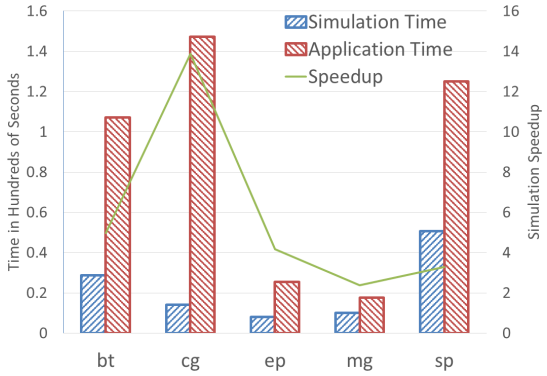


Figure 9: Simulation time vs. application runtime of 5 NPB benchmarks on 4,096 ranks

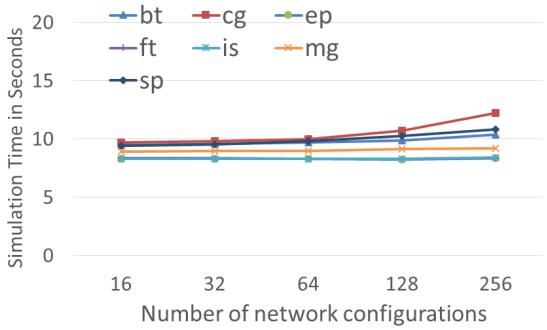


Figure 10: Simulation time of NPB benchmarks for 16, 32, 64, 128 and 256 network configurations

simulation. The speed-up is computed by the total execution time (sum of the execution time for all ranks) for the applications divided by the total simulation time (sum of the execution time for all ranks). As can be seen from the figure, the simulation time speed up ranges from 3 to 45 in comparison to the application time. Figure 9 depicts the simulation time and the application execution time on for 4096-rank runs. The time reported is the actual execution time of the program and our tool on Cielito. We see a speed-up ranging from 2 to 14, similar to the smaller runs.

Figure 10 shows the simulation time for the benchmarks when different numbers of network configurations are

simulated in each run. This experiment uses the 64-rank runs. With more network configurations, more timestamps need to be computed and communicated among processes. We can see that the execution time increases only slightly when the number of network configurations increases from 16 to 256 as a result of the extra communication cost of calculation and communication of timestamps based in our MPI-based tool. This indicates that the number of network configurations (up to 256) is not a major performance limiting factor in the trace replay; and the tool can effectively predict the performance for many network configurations in one simulation.

V. Conclusions

We have presented a trace-based and communication-centric fast classification tool for MPI programs. Our innovation is to use a modified Lamport logical clock scheme that uses non-unit computation and communication times to predict overall time. By maintaining multiple independent logical clocks that are parameterized differently but honor the same happens-before relationship, the tool can predict execution time on many network configurations in nearly the same time needed to predict time for a single configuration.

This multiple-prediction capability enables new analyses to be performed that would be too computationally expensive to perform with traditional, one-configuration-at-a-time simulation. In particular, overall application time can be attributed to the time spent in computation, communication, and load imbalance; and applications can be analyzed for their sensitivity to compute speed, latency, and bandwidth.

The importance of this work is that application developers finally have the information needed to determine if load imbalance in their codes is limiting performance more than hardware speeds; and system architects finally have the information needed to determine what hardware upgrades would yield the greatest improvement in application performance.

References

[1] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen. A simulator for large-scale parallel computer architectures. *Int. J. Distrib. Syst. Technol.*, 1(2):57–73, Apr. 2010.

[2] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Using performance modeling to design large-scale systems. *Computer*, 42(11):42–49, 2009.

[3] C. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low memory, modular time warp system. In *Parallel and Distributed Simulation, 2000. PADS 2000. Proceedings. Fourteenth Workshop on*, pages 53–60, 2000.

[4] A. Chan, D. Ashton, R. Lusk, and W. Gropp. Jumpshot-4 users guide. [Online; accessed 14-AUG-2015].

[5] S. Corporation. SST: The structural simulation toolkit. 2014. [Online; accessed 14-DEC-2012].

[6] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurr. Comput.: Pract. Exper.*, 22(6):702–719, Apr. 2010.

[7] S. Girona and J. Labarta. Sensitivity of performance prediction of message passing programs. In *in Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 620–626, 1999.

[8] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.*, 20(3):389–398, Mar. 1994.

[9] Intel. Introduction to Ethernet latency. 2014. [Online; accessed 14-AUG-2015].

[10] K. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the communication hairball: Visualizing parallel execution traces using logical time. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):2349–2358, Dec 2014.

[11] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 86–96, April 2013.

[12] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A parallel program development environment. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 665–674, London, UK, UK, 1996. Springer-Verlag.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. In *Supercomputer*, volume 12, January 1996.

[15] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.*, 69(8):696–710, Aug. 2009.

[16] D. Panda. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. 2014. [Online; accessed 6-Oct-2015].

[17] QLogic Corp. Introduction to Ethernet latency. an explanation of latency and latency measurement. 2014. [Online; accessed 6-Oct-2015].

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*, volume 1, The MPI Core. The MIT Press, Cambridge, Massachusetts, 2nd edition, Sept. 1998.

[19] R. Thakur and W. D. Gropp. Improving the performance of MPI collective communication on switched networks. 11/2002 2002.

[20] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. *Int. J. Parallel Program.*, 33(2):183–207, June 2005.