Performance analysis of parallel programs via message-passing graph traversal

Matthew J. Sottile[1], Vaddadi P. Chandu[2], and David A. Bader[2]

[1]Los Alamos National Laboratory
matt@lanl.gov

[2]Georgia Institute of Technology
{vchandu, bader}@cc.gatech.edu

Abstract

The ability to understand the factors contributing to parallel program performance are vital for understanding the impact of machine parameters on the performance of specific applications. We propose a methodology for analyzing the performance characteristics of parallel programs based on message-passing traces of their execution on a set of processors. Using this methodology, we explore how perturbations in both single processor performance and the messaging layer impact the performance of the traced run. This analysis provides a quantitative description of the sensitivity of applications to a variety of performance parameters to better understand the range of systems upon which an application can be expected to perform well. These performance parameters include operating system interference and variability in message latencies within the interconnection network layer.

1. Introduction

The primary causes of performance degradation within distributed memory parallel computers are the latency of the interconnection network and perturbations to applications due to interactions with the operating system and other tasks. One technique for analyzing the performance characteristics of a distributed memory parallel program is to simulate perturbations in message latency and processor compute time, and propagate these perturbations through subsequent messages and computations to observe their effect on application runtime. This is easily modeled as a discrete event simulation, and many well defined techniques exist for building and analyzing such models [7, 5]. Unlike a general discrete event model, we chose to directly analyze the message-passing graph that results from the execution of the program on a set of nodes. In this paper, we introduce a performance analysis methodology that is developed to study these perturbations. This allows us to greatly simplify the model and analysis code, and provides a simple framework for defining the constraints under which the analyzer can model perturbations while still guaranteeing correctness and message order of the parallel program.

We perform and present this research in the context of the Message Passing Interface (MPI) library [8], but the work itself is not bound to MPI. Parallel programs for distributed memory systems generally are implemented via primitives for passing data between processors and synchronizing computations between pairs of processors and collective processor groups. The MPI implementation of this programming model is widely used and currently very popular. Other implementations exist, such as the older Parallel Virtual Machine (PVM) [18] and the Aggregate Remote Memory Copy Interface (ARMCI) [12]. Our performance analysis methodology is applicable to all of these message-passing implementations by simply defining the primitives of the implementation in the context of the framework presented here.

1.1. Related work

Several researchers have developed model and trace-based systems for analyzing the performance of parallel programs. Petrini et al. [14] relied on modeling the parallel program and the parallel computer before performing the analysis. This method was used to predict the performance of programs on machines prior to their construction, and to identify the causes of performance discrepancies from the predictions once the machine was constructed.

1-4244-0054-6/06/$20.00 ©2006 IEEE

Unlike the model-based approach, other techniques are driven by traces of actual program runs. Trace driven methods have the advantage that they capture nuances in execution that arise from unique data conditions at runtime that cannot be modeled purely by examining the static program code itself. Unfortunately, this specificity is not as flexible as model-based approaches with respect to performance prediction and extrapolation. In a trace, one loses the statistical properties of the control flow branch and join structure of the original code, limiting the potential for performance extrapolation.

Dimemas [1, 3], a commercial tool developed at CEBPA-Centro Europeo de Paralelismo de Barcelona, is one such tool for performance prediction of parallel programs using trace-based analysis. The user specifies the communication parameters of the target machine. A simple model [1, 3, 15] is assumed for communication which consists of (a) machine latency, (b) machine resources contention, (c) message transfer (message size/bandwidth), (d) network contention, and (e) flight time (time for message to travel over the network). Given a trace-file and the user's selection of network parameters, Dimemas simulates the parallel program's execution using the communication model. While Dimemas captures most of the parameters that affect the impact of the network on a parallel program's execution time, the model does not have similar capabilities for analyzing the operating system's interference with the application's performance.

Users who are familiar with trace analysis tools such as Vampir [4] and Dimemas will find the concept of a message-passing graph essentially identical to the visualizations that they create of parallel program execution. While Dimemas and our work are both trace-based performance analyzers for parallel programs, several key differences between Dimemas and our framework exist. 1) We seek to parameterize both the on-node noise and cross-node messaging using empirically-derived distributions from microbenchmarks. Dimemas provides an API for this purpose, but Dimemas itself does not actually perform this type of parameterization. 2) We do not require a global resolution of clocks in the trace files required by the Vampir-like trace format used by Dimemas. 3) In our framework, we handle arbitrarily large trace files by streaming the trace through the simulator instead of loading it all in core. In comparison, Dimemas can handle large traces by reducing their information content in a preprocessing step. 4) We also seek to compare the effects of various architectures by using experimentally-derived parameter distributions to construct empirical distributions for deriving simulation parameters. Additionally, Dimemas provides a 'plug-in' mechanism that can be used to simulate delays both at the interconnect layer due to latency and contention, and the compute node to simulate operating system noise. As future work, we will investigate the use this plug-in mechanism to parameterize simulations using experimentally-derived empirical distributions, instead of scalar constants or idealized probability distributions. In this paper, we extend the concept of trace-based analysis beyond the static messaging graph to a framework in which the graph is modified in a disciplined manner to model performance perturbations and their effect.

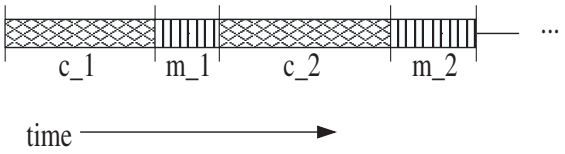

Figure 1. Alternating phases of computation (c_i) and messaging (m_i) over time.

2. The message-passing graph concept

Consider a parallel program using a distributed memory programming model via message-passing. On a given processor, the program alternates between periods of local computation and resource usage, and interaction with remote processors via message-passing events for both data movement and control synchronization (see Fig. 1). If each of these periods has a time stamp at the beginning and a small amount of meta-data indicating what occurred during the period, one can easily determine what the processor was doing at any given time. We begin constructing the message-passing graph by creating a set of "straight-line" graphs (one per processor) with nodes at the beginning and end of each computation of the messaging period and an edge between successive events labeled with the duration of the period. Given these straight-line graphs, we now must consider message-passing activity to create the program's overall message-passing graph. The edges used to construct the straight-line graphs are referred to as *local edges* in this paper.

During periods of message-passing activity, the processor interacts with one or more other processors depending on which message-passing primitives are invoked. Given the ordering of the events on each processor and some simple knowledge about the blocking semantics of message-passing primitives, we can easily perform a single pass over all events to decide

which events on remote processors correspond to local message-passing events. Using this information, we can create edges between the coupled events on interacting processors representing the initiation and termination of a message-passing primitive. These edges that represent processor interactions via message-passing are referred to here as *message edges*.

It is vitally important for modeling consistency to create a pair of message edges for each message-passing event, although where one places the edges depends on the event being modeled. The importance of the edge pair is in recognition of the effect of local perturbations on remote nodes on the completion time of local message-passing events. The message-passing edges must capture not only latency variations between the nodes, but also allow for the propagation of remote perturbations back to all affected processors.

For modeling consistency and clarity, the model specified in this paper embeds the semantics of the message-passing operations and their perturbations within the graph itself. We avoid pushing the semantics of the operations to the level of the algorithm that walks the graph, as this both complicates the algorithms and makes verification and validation of the simulation more difficult. For future research, we will investigate pushing such responsibility to the algorithms instead of the graph representation for performance optimization of the analysis tool itself.

In the next section, we will show how to define this graph for a subset of MPI-1 message-passing primitives [8] based on the send-receive model. Many of the remaining MPI operations share characteristics with those we describe, and our definitions can be easily extended to include them. Our methods currently do not attempt to capture the put-get semantics of other message-passing models such as ARMCI and those introduced in MPI-2 [9].

3. Graph primitives for a subset of MPI-1

A common method to classify message-passing primitives is to partition them into two sets based on the number of interacting processors, and partition these into two further sets based on the blocking semantics of the events. The first partition separates *pairwise* events from *collective* events. A simple send operation is pairwise, while a reduction is collective. The second partition separates *blocking* events from *nonblocking* events. The simple synchronous send operation is blocking, while an MPI `MPI_Isend` is nonblocking.

A third class of primitives exist for single node operations that are necessary, but straightforward with respect to this work. These include functions such as `MPI_Init`, which appear in the trace files and graph, but given the fact that they do not interact with other nodes, are trivial to model.

3.1. Pairwise primitives

The first set of primitives are the pairwise primitives. For a set of parallel processors, a pairwise event is defined as one that involves two processors exchanging a (potentially empty) data set.

3.1.1 Blocking

A blocking operation will not return control to the caller until it has successfully completed or encounters an error condition from which it cannot recover or proceed. The `MPI_Send` operation is a blocking primitive that sends a block of data to a receiver who posts a matching `MPI_Recv` receive operation. The MPI specification provides three forms of blocking send: the synchronous send, the buffered send, and the ready send. Each blocks until some condition has been met.

Pairwise blocking operations are easy to model in the graph, as they require a simple matching of the pair of events and the blocking nature of the operation requires a well defined begin and end relationship between the nodes. The result is that perturbations propagate through the graph to each node and preserve the pairwise relationship and event ordering.

3.1.2 Blocking send/receive pair

Here we present a simple graph representation of the paired send/receive operation `MPI_Send` and `MPI_Recv`.

In the presence of modeled perturbations, the end times of each operation after perturbation are determined by Eq. (1).

$$\begin{aligned} t'_{s_e} &= max(\, t_{s_e}, \\ &\qquad t_{s_s} + \delta_{os_1}, \\ &\qquad t_{s_s} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{os_2} + \delta_{\lambda_2}) \\ t'_{r_e} &= \qquad t_{r_s} + \delta_{os_2} + \delta_{\lambda_1} + \delta_{t(d)} \end{aligned} \tag{1}$$

As we can see, due to the possibility of on-node interference (δ_{os}), messaging latency (δ_{λ}), and perturbations that are proportional to the amount of data sent ($\delta_{t(d)}$), the completion time of send operations is dependent on the maximum of three values. These represent the original completion time (t_{s_e}), the completion time delayed by local perturbations on the sender alone ($t_{s_s} + \delta_{os1}$), or the delay due to latency in sending

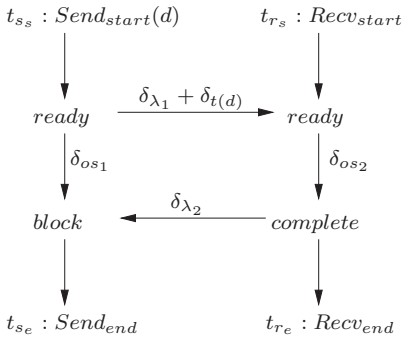

Figure 2. Subgraph representing a blocking send and receive pair of d bytes of data. Locations are indicated where operating system noise (δ_{os}), latency (δ_λ), and bandwidth ($\delta_{t(d)}$) are modeled.

the message, processing it on the receiving end with potential receiver-local perturbations, and latency in acknowledging completion ($t_{s_s} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{os2} + \delta_{\lambda_2}$).

3.1.3 Nonblocking

It is widely recognized that significant performance gains up to some limit can be made by hiding latency to slow resources such as memory and I/O by overlapping additional computation with the resource request. As such, parallel programs often take advantage of nonblocking messaging primitives to overlap inter-processor communication with local computation to hide the high latency of the interconnection network. MPI provides primitives such as `MPI_Isend` for this purpose. These nonblocking primitives return immediately (hence the "I") to the caller, and their status can be checked at a later time. This allows the program to post data for transmission to a receiver as soon as the data is ready, and perform additional computation until the sender must block (if at all) pending the completion of the send operation.

Due to the fact that nonblocking calls immediately return, variations in latency and local perturbations on the receiving end of the transaction are not immediately apparent to the sender. We are faced with two possible situations with different consequences. First, we have a situation where the transaction is semi-synchronous. The send is nonblocking, but at a later time the sender invokes a blocking routine such as `MPI_Wait` that forces the sender to not proceed further until the communication is complete. This is easy to simulate, as it can be considered similar (not necessarily equivalent) to a synchronous send operation that has been separated into two phases. The lack of equivalence is due to the fact that multiple instances of the operation may be interleaved.

The second situation is trickier, and represents a truly asynchronous interaction between processors. In this case, the sender posts nonblocking send operations, and never blocks on the successful completion of the transaction before posting subsequent sends to the same receiver. In Fig. 3 we illustrate the first case of a paired send and receive followed at some later point by a pair of wait operations.

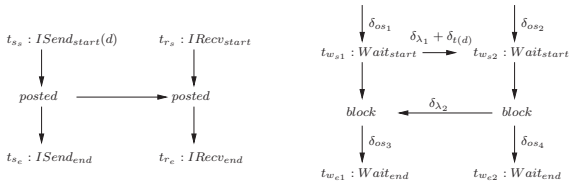

Figure 3. Subgraph representing a nonblocking send and receive pair of d bytes of data, and the corresponding wait operations. The send/receive pair is matched with a wait pair by matching the *status* flags that uniquely identify the send/receive transaction.

Eq. (2) shows the modified end times for the wait operations. Note that the end times of the send and receive operators are not modified due to their immediate return semantics.

$$\begin{aligned} t'_{w_{e1}} &= max(\quad t_{w_{e1}} + \delta_{os_1} + \delta_{\lambda_1} + \delta_{t(d)} + \delta_{\lambda_2} + \delta_{os_3}, \\ &\qquad\quad t_{w_{e1}} + \delta_{os_2} + \delta_{\lambda_2} + \delta_{os_3}) \\ t'_{w_{e2}} &= max(\quad t_{w_{e2}} + \delta_{os_2} + \delta_{os_4}, \\ &\qquad\quad t_{w_{e2}} + \delta_{os_1} + \delta_{\lambda_1}) \end{aligned} \tag{2}$$

3.2. Collective primitives

Collective operations are used in nearly all parallel programs that require each processor to receive some amount of global state during the execution of the parallel program. These include synchronization primitives such as a *barrier*, data distribution primitives such as *broadcast*, and global application of associative operators such as a *reduction*. The presence of collective operations is often a primary source of performance degradation in a parallel program because a single slow processor will induce idle time in all other processors. In particular, local perturbations can have a global effect on the overall program behavior.

Fortunately, modeling this is easily accomplished in the graph framework. Consider a set of p processors participating in a collective operation. Each processor has incurred some amount of simulated delay up to this point due to local perturbations and message latency. What must be decided is what the delay on each processor should be after the collective operation has occurred. A simple approach is to choose the maximum delay from the set of processors, and propagate it across all others. This is not necessarily accurate beyond a rough first approximation. The collective operation requires a sequence of network transactions to occur, and between each exists periods of local computation. This means that there is a possibility that local perturbations and network latency may cause the delay on each processor after the transaction to actually be greater than the maximum delay entering the collective.

Consider an all-reduce operation (`MPI_AllReduce`) such as a global summation. One can easily show that a butterfly messaging topology can be used to require each processor to send and receive $O(log(p))$ messages [6, 13]. This can be explicitly constructed in the graph, which allows for analysis to be performed without any special knowledge of the operation. Unfortunately, this is not space or time efficient given the fact that we know a-priori that a single collective operation can be considered equivalent to $log(p)$ periods of local computation and pairwise messaging. As such, we can simply model the collective as an edge from all p processors to a single processor, on which the $log(p)$ communication and computation perturbations are propagated, and a set of edges from this processor to all others that induces no additional perturbations, but simply communicates the maximum of this set of perturbations to every other processor.

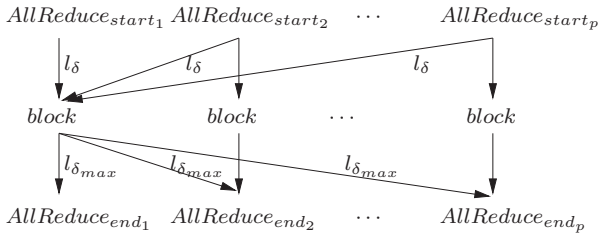

Figure 4. An AllReduce operator subgraph. The abbreviated noise annotations on edges are described in the text.

In Fig. 4 we show how an AllReduce operator is modeled. In the AllReduce operation, each node must contribute local data to a global operation, the result of which is then sent to all processors. Instead of modeling the communication topology precisely, we approximate it by sampling operating system noise and latency $log(p)$ times for each processor, and labeling the edge from each i^{th} processor to the first with this value called l_δ. The maximum value of all l_δ values is computed, and propagated back out to all nodes along the return edge labeled $l_{\delta_{max}}$. This has the effect that is frequently observed in practice of forcing the slowest node (or in this case, the most perturbed node and link) to dominate the performance of the entire collective.

A simplification of this graph can be used to model a simpler Reduce operator in which only one processor holds the result after completion. In this case, three modifications are necessary. First, the message edges labeled l_δ are simplified to only sample latency once. Second, each processor has a local edge from the start node to the blocking node labeled with local operating system noise. Finally, the $l_{\delta_{max}}$ edges become unlabeled, as they are do not contribute additional perturbations themselves, but are simply required to carry the contribution of noise on the processor receiving the reduction result to those providing data to the operation.

4. Creation of message-passing graph

The message-passing graph that we create for analysis is generated using trace data from an execution of the program on a parallel system. Each processor creates an event trace that records the local timestamp, the event type, and event metadata for each event that occurs. This is done via the standard PMPI interface defined by the MPI specification. Each MPI primitive to be recorded is wrapped with a lightweight PMPI wrapper that records the event in a memory resident buffer. The buffer is dumped to an event trace file when it becomes full, and is then reset to empty for future events. The size of this buffer can be tuned to compensate for event frequency and overhead for I/O to dump the trace information to a file. It is unavoidable that tracing will introduce performance perturbations not present in the non-instrumented version of the parallel program. We have taken care to minimize this perturbation, but must recognize that it is present and must be kept in mind during later analysis of the program performance. For future work we will use more robust tracing tools that already exist as discussed later.

4.1. Avoiding clock synchronization

It is important to recognize that constructing the graph only requires pairing events across processors. The execution order on each processor makes this possible using execution ordering *only*. It is tempting, although misleading, to infer information about two processors using their local timestamps and clocks. This is related to a difficult problem in distributed systems to synchronize a set of clocks that are separated by links with non-trivial, and most importantly, unknown and statistically-defined latencies and clock drifts [2].

We take advantage of the fact that a trace of a program that ran to completion represents a message pattern that was sufficiently correct for a proper run. Each message event is guaranteed to have a counterpart, and this counterpart can be found simply by processing each event in order on each processor. If an event is encountered and the counterpart must be found, the algorithm must simply find the next event on the counterpart processor that has not already been found that matches. This is different, and significantly simpler than deriving the messaging graph from static code, by recognizing the fact that the run occurred and the message ordering is fixed as a result. Although attempting to resolve clocks across the traces is also a possible way to align and match events, using the message ordering on each processor to regenerate the messaging pattern makes this unnecessary.

4.2. An implementation of the graph construction algorithm

We have designed and implemented a prototype program to process trace data into the message-passing graph structure, and introduce simulated perturbations in order to analyze the sensitivity of an application to message-passing latency and operating system noise. The graph is created according to the message-passing primitive semantics specified by the MPI standard and implementation specification, some of which were illustrated in Section 3. The trace files are generated using a C library conforming to the PMPI standard with timestamp data provided by the high resolution, cycle-accurate timers available on all modern microprocessors.

We now present how to construct the message-passing graph from trace data. An event is split into two subevents: a start subevent and an end subevent, which correspond to entry and exit from the message passing operation that produced the event. For finer granularities, more subevents can be added without much effort to capture implementation specific details of how the processors interact during the message-passing primitive.

Fig. 5 in Appendix A shows a message-passing graph that our model generated from a set of trace data. For simplicity and clarity in this example, we used reduced trace data and only blocking MPI primitives. Each edge connects two subevents with an edge weight equal to the delay incurred between its source and sink subevents. The source and sink subevents need not be necessarily the start subevent and end subevent, but may be anything depending on whether the edge is a local edge or a message edge.

In order to simulate the operating system noise, the weight of a local edge connecting two subevents in the same trace is altered and the change is additively propagated through the graph to all graph nodes reachable from the sink node of the modified edge. Likewise, to simulate network latency, the weight of a message edge connecting two subevents in different traces is altered and the change is again propagated through the graph. Thus, behavior of the program under study with varying operating system circumstances and network parameters can be studied quantitatively by modifying edge weights and carrying their cumulative effect through the graph as it is traversed. This information gives a firm base on which the degree of suitability of a parallel program to a particular platform can be determined. We also can explore how varying parameters affects not only overall runtime, but regions within the graph where perturbations are absorbed or fully propagated, corresponding to tolerant or highly sensitive code, respectively.

4.3. Correctness

Correctness of the graph and its modification during the analysis process is vital. The process of taking traces and merging them into a single message-passing graph has the benefit of using the fact that the program did run correctly in the first place in order to create the traces. Constructing the graph based on this is simply a matter of associating events to match message end points, and this has been shown to be possible in the past as evidenced in tools such as Vampir. Correctness is important to consider though when modifying the timings of events in the process of analyzing the noise sensitivity of the program.

The key question in this process is whether the modified timings of events causes events to occur prematurely with respect to their counterparts on other processors. In a purely synchronous program, this is impossible, as the delays are propagated along the lo-

cal and message edges, and all events on interacting processors are delayed in a quite straightforward manner. Nonblocking, asynchronous interactions are the complicating factor. For example, a processor that initiates a send that does not block on the successful completion of the transmission does not immediately see delays on the receiving end before it proceeds to additional events. In MPI, this is realized in the `MPI_ISend` primitive. Fortunately, in most codes that have been examined, these nonblocking calls have a corresponding blocking event that causes the sender to block on a check for the completion of the send. In essence, the nonblocking send allows the programmer to implement a synchronous send operation with the ability to inline code that does not depend on the completion of the send in between the initiation of the transfer and the check that it completed. In MPI-1, this is realized as the pairing of `MPI_ISend` with a blocking `MPI_Wait` (with `WaitAll` and `WaitSome` existing for similar blocking semantics on sets of `ISend` operations) primitive.

In the worst case, one processor issues a sequence of nonblocking sends without checking that any have completed before issuing more to the same processors. If the receiver posts blocking receives or `MPI_Wait` operations, correctness is preserved by ensuring that delays in the sends are propagated to the receiver and push the wait operations ahead to match the difference in time due to the delay. In the event that this is not possible, and both sides use only asynchronous calls with no synchronization (a possible, although questionable practice for most programs), the tool cannot guarantee that an arbitrarily perturbed graph is correct and produces a warning that this situation has been identified.

5. Parameterizing simulated perturbations

Given application traces, the questions that we wish to answer using the framework and tools presented here deal with how well one can expect a program to perform on a parallel computer under the influence of a set of performance influencing parameters. For example, one can execute a parallel program on a system with a minimal, lightweight kernel running on compute nodes, and then explore what amount of operating system overhead the application can tolerate before significant performance degradation occurs. The previous sections discuss the methodology for exploring the application performance under varying parameters. To best study these questions, one must also have a disciplined approach to determining how to parameterize the simulation and analysis tools.

We propose that in the initial phase of this research, parameters be determined using *microbenchmarks* that are carefully constructed to probe very specific performance parameters. Each parallel platform has a signature that is defined by the set of metrics determined by various microbenchmarks, and this signature is provided to the analysis tools, along with an application trace, to estimate the behavior of the program on the new platform. Our current work treats parameters as random variables with a distribution parameterized by the microbenchmarks.

Two methods can be used to generate parameters for analysis given the output of microbenchmarks. First, one can estimate parameters for assumed distributions of the parameters. For example, it is generally assumed that queueing time can be modeled as an exponential distribution, and the parameter of the distribution can be estimated from experimental measurements. The second method for generating parameters is to use the data itself to build an empirical distribution. This method relies on gathering a sufficiently large number of samples such that the shape of the actual distribution is accurately captured. It is a simple exercise to show that the resulting empirical distribution approaches the actual distribution as the sample size increases, as stated by the law of large numbers [17].

5.1. Operating system noise

Operating system noise is the result of time lost to non-application tasks due to operating system kernel or daemons requiring compute time. A "noisy" operating system will frequently take time from applications for its own operations, while a "noiseless" operating system will allow applications to use as many cycles as possible. The effects of this noise can be quite severe, as exemplified by experiences with the ASCI Q supercomputer [14].

Microbenchmarks are available to probe systems to infer the perturbation due to operating system noise, and the data from these microbenchmarks can be used to generate empirical distributions from which our analysis tool can sample. The fixed time quantum (FTQ) microbenchmark described in [16] probes for periodic perturbations in a large number of fine grained workloads. The point-to-point messaging microbenchmark described by Mraz [11] uses a simple message-passing program to probe the effect of noise on message-passing programs. As discussed in Section 3, noise is represented in the message-passing graph via edge weights on local edges. This models the additional time a processor requires to complete a fixed amount of work due to preemption for operating system tasks.

5.2. Interconnection network performance

The interconnection network on a parallel computer has two parameters that influence performance the most: bandwidth (how much data can be transmitted in a quantum of time), and latency (how much time is required to move a minimal quantum of data between two nodes). These parameters are easy to determine, and well known; simple benchmarks for bandwidth and latency exist for MPI and other communication protocol layers. A latency benchmark measures the variation in the time taken to send a message between two nodes. Given the lack of an accurate, high-precision global clock across communicating processors, the latency benchmark uses a traditional ping-style message exchange between two processors. A bandwidth benchmark is similar, except with messages of a significant size in one direction, with an acknowledgment returned to the sender. The size of the large message must be sufficiently large so as to make the latency component negligible in the overall time.

Two assumptions are made regarding this benchmark. First, the connection between the nodes has symmetric performance characteristics with the distribution of message latencies (from sender to receiver and vice-versa) both independent and identically distributed (*iid*). Second, two separate messages from one host to another have latency distributions that are also iid. Systems where routing adaptation and "warming up" of links occurs will violate this second assumption, and a suitable alternative tool must be employed to measure and model the appropriate statistical distribution.

Variations in message latency and bandwidth are modeled within the graph as edge weights on message edges. Latency noise is modeled independently of the size of the message, while variations in bandwidth must be modeled as a function of the message size. Interconnect noise is also simulated using empirical distributions derived from sampled data.

6. Implementation and example application

The initial implementation of the tools for analyzing traces includes a simple PMPI-based tracing generation library and an analyzer that inputs these traces, constructs the message-passing graph, and allows for a very simple parameterization of edge weight modifications to explore noise and latency variations. The analysis tool uses the algorithm described in Section 4 to connect individual traces for each processor with message edges. To avoid the obvious limitations imposed by memory constraints, the analysis tool uses a windowed approach to building the graph. This is particularly important to consider given the number of events in a long running, high processor count job.

Given the set of performance parameters related to noise in the operating system on processors and the interconnection network connecting them, the analysis tool processes the graph in the following manner. As the graph is created using subgraphs as described in Section 3 the δ values that are indicated as edge weights are generated by sampling the empirical distributions associated with the parameters. The original message-passing trace has edge weights on local edges corresponding to the time intervals observed in the run that generated the trace. Message edges are weighted zero originally, as the effects of latency and bandwidth are already embedded in the timings of the actual events that occurred. Simulating additional delays in messaging is achieved by marking message edges with non-zero, positive values. As the graph is streamed through the tool, the $max()$ operators defined in Section 3 are applied to modify the times of each node in the graph based on the simulated perturbation deltas added to both message and local edges. The end result is a final modified timestamp on the final node for each processor corresponding to the `MPI_Finalize` event.

From this new completion time, we can observe how running times for the overall program and individual processors increase in the presence of varying degrees of noise. For example, if we generate a trace on a system with relatively low noise (such as a bproc cluster as discussed in [16]), we can parameterize the simulation with performance parameters measured on a system with higher noise to explore how the program can be expected to perform on a system composed of higher noise processors.

It should be clear that we do not currently explore the possibility of determining how a trace taken on a high noise system would run on a system with lower noise. A similar methodology could be applied by introducing negative edge weights, but this sort of analysis is being left for future work.

6.1. Token ring

A token ring is one of the simplest messaging topologies found in realistic parallel programs. In n-body simulations, it is occasionally true that the n^2 particle interactions must be computed directly instead of using approximation algorithms that require $O(nlogn)$ or $O(n)$ computations. For p processors, it is possible then to divide up the n particles into sets of $\frac{n}{p}$

on each processor. Each processor p_i then packages up the set of particles that it "owns", and passes it to the $(i+1 \ mod \ p)^{\text{th}}$ processor. This processor computes the interactions between its local particles and those contained in this "token" containing a particle set from some other processor. This set is then passed on to the next processor as before, and this is repeated p times until each processor receives the token containing its local particle set, at which time each processor has computed the influence of all n particles on their local set.

Our initial experiments verify the intuitive behavior that one would expect from a fully synchronous program as this. We performed a traced run on 128 processors of a ring-based program, and varied the degree of perturbations from none to a mean of 700 cycles worth of perturbation at 100 cycle increments. The resulting change in running times increases for each processor that matches the 100 cycle increments multiplied by the number of traversals of the ring. For example, if the ring was traversed 10 times with each processor injecting 100 cycles of noise for each message, the runtime of each processor increased by approximately 10*100*128 cycles.

7. Future work and conclusions

The current set of tools we designed and implemented are developed to explore the feasibility and algorithmic aspects of this method of performance exploration. Two major areas of work are in need of immediate attention. First, we plan to use existing tracing libraries that provide a more complete treatment of the MPI specification, in addition to allowing traces to be generated for other message-passing and shared memory parallel programming tools. The library we are exploring, KOJAK [10], provides the EPILOG tracing format and accessor library. The second area of work is to provide a mechanism to provide a richer set of parameters to the simulation, and maintain a history of analysis experiments that are performed using our tools. We would also like to investigate modeling reduced noise from that observed in the traced runs to explore how performance could be expected to change if the run was performed on a system with *less* noise.

We have presented an analysis methodology and prototype of a performance analysis tool driven by message-passing traces, which is scalable and ensures correctness of the analysis that preserves message ordering true to the trace-generating run. We discussed how operating system and interconnect parameters can be generated and integrated into our analysis methodology. We model the application as a message-passing graph, which is traversed in the same order as the execution order of the original parallel program. Enforcing no changes in the order of execution ensures correctness of the model in the presence of blocking and nonblocking message-passing primitives. Our windowed graph generation technique allows us to analyze traces of arbitrarily large size on systems with limited memory, thus making it fully scalable. Since trace-based simulation reflects application behavior on real machines under internal data states for real runs, the results are expected to be more accurate for a given processor count than an idealized model at the cost of restricting extrapolation abilities.

While the tools are still early in development, currently supporting only a subset of blocking, nonblocking and collective MPI primitives, this work introduces a promising methodology for analyzing parallel program performance taking into account their *actual* runtime behavior for real problems. In the future, we also plan to expand this performance analysis to support more of the MPI-1 primitives, in addition to other parallel programming paradigms including but not limited to extensions present in MPI-2 and other distributed memory models such as ARMCI. These primitives represent what are known as one-sided communications operations.

As the analysis tools mature, we plan to focus on studying a number of regular and irregular parallel applications over different systems using this tool. We ultimately aim to provide a methodology and a set of tools to assist in the process of analyzing the performance of large applications on a variety of parallel architectures in order to characterize their performance and guide users and system procurements to determine the best platform for applications of interest to the user community.

8. Acknowledgments

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR No. 05-7914. Bader was supported in part by NSF Grants CAREER ACI-00-93039 and CCF-06-11589, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; and DARPA Contract NBCH30390004.

References

[1] R. M. Badia, J. Labarta, J. G., and F. Escalé. DIMEMAS: Predicting MPI applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools, 8th Global Grid Forum (GGF8)*, pages 50–60, Seattle, WA, June 2003.

[2] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, second edition, 1994.

[3] S. Girona, J. Labarta, and R. M. Badia. Validation of dimemas communication model for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 7th European PVM/MPI Users' Group Meeting*, pages 39–46, Balatonfüred, Hungary, Sept. 2000.

[4] Intel Corporation. HPC Products. http://www.pallas.com/e/products/index.htm.

[5] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, New York, 1991.

[6] J. JáJá. *Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992.

[7] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. M-H series in industrial engineering and management science. McGraw-Hill, Inc., second edition, 1991.

[8] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, 1994.

[9] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[10] B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 1301–1304, Klagenfurt, Austria, Sept. 2003.

[11] R. Mraz. Reducing the variance of point to point transfers in the IBM 9076 parallel computer. In *Proceedings of the Conference on Supercomputing*, pages 620–629, Washington, DC, Nov. 1994.

[12] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[13] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, CA, 1997.

[14] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. Supercomputing*, Phoenix, AZ, Nov. 2003.

[15] G. Rodríguez, R. M. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference*, pages 183–188, Pisa, Italy, Aug. 2004.

[16] M. Sottile and R. Minnich. Analysis of Microbenchmarks for the Performance Tuning of Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 371–377, San Diego, CA, Sept. 2004.

[17] H. Stark and J. W. Woods. *Probability and Random Processes with Applications to Signal Processing*. Prentice Hall, third edition, 2002.

[18] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, Dec. 1990.

A. An example message-passing graph

We show a message-passing graph generated from a real trace generated by a simple sequence of blocking communications between a small set of processors. The graph was generated using our framework and visualized using Graphviz.

Figure 5. A Message-Passing graph for Trace Data containing Blocking MPI Primitives.