MPInside: a Performance Analysis and Diagnostic Tool for MPI Applications

Daniel Thomas
SGI
21 rue Albert Calmette
78350 Jouy en Josas, France
+33 1 34 88 80 00
dthomas@sgi.com

Jean-Pierre Panziera
SGI
21 rue Albert Calmette
78350 Jouy en Josas, France
+33 1 34 88 80 00
jpp@sgi.com

John Baron
SGI
2750 Blue Water Road
Eagan, MN 55121 USA
+1 (651) 683 3544
jbaron@sgi.com

ABSTRACT
Performance analysis and prediction of parallel applications using the Message-Passing Interface (MPI) standard is a challenging task. Collecting, organizing, and making sense of profiling data for MPI jobs of even modest scale is difficult and time-consuming. The task is further complicated by the inherent difficulty in interpreting the resulting communication measurements. In this paper we introduce MPInside, a new profiling and diagnostic tool that overcomes these constraints with carefully considered choices for measurement techniques, capabilities, and output formats. Using examples from real-world applications, we illustrate the innovative features of the tool—including late senders for point-to-point calls and unaligned collective calls—all in an instrumentation-free framework. We also demonstrate the in-flight modeling capabilities of MPInside with several "what if" experiments.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Measurement techniques, Modeling techniques, Performance attributes.

General Terms
Algorithms, Measurement, Performance, Experimentation.

Keywords
Parallel applications, MPI, Performance analysis, Performance modeling.

1. INTRODUCTION
Analysis tools for MPI [1] programs are a critical requirement for both application developers and end users, providing a means to understand program behavior, identify performance bottlenecks, and predict performance on a variety of system configurations. Numerous tools available today attempt to satisfy these needs with varying degrees of instrumentation, tracing, and post-processing capabilities. In our experience these tools provide either too little information concerning key parameters of the MPI communications or too much information to be summarized easily. In addition, the majority of tools require application recompilation and/or re-linking in order to obtain sufficiently detailed communication statistics.

In this paper we first describe the motivations that led us to develop yet another MPI profiling tool, MPInside. We then present the basic profiling features and the advanced methods used to measure the wait time related to unaligned MPI calls for both collective and point-to-point communications. Section 4 describes the modeling capabilities developed within the tool framework to estimate application behavior on a perfect interconnect, thus providing a best-case signature for the application. We also show how more sophisticated models can be applied to predict application performance on arbitrary system configurations. Finally, we close with a case study illustrating how MPInside modeling can identify why a real-world application, LAMMPS, scales poorly on one platform and runs well on another.

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.

2. MPINSIDE MOTIVATIONS
The MPInside project began as an investigation on parallel applications using the MPI standard. With a classical profiling tool, one measures the time an application spends in the user code versus the MPI library. Usually, when the computation time dominates, the application scales well. On the other hand, a large percentage of communication time typically indicates a poor parallel efficiency. One would naively believe that better communication hardware directly translates into reduced communication time and better parallel efficiency.

Surprisingly, our measurements showed that applications with a high communication rate did not always benefit from a hardware interconnection network upgrade. Conversely, some communication-intensive MPI applications appeared to scale noticeably well even on low-performance networks. Finally, faster processors reduced not only the computation component of an application but also, unexpectedly, the communication time.

It became clear that simple inspection of the profile (the timing breakdown) was insufficient to analyze and predict the performance of MPI applications. One needed to check what was happening "inside" the application, "inside" the MPI library and how the two (application and MPI) were interacting together.

2.1 MPI Profiling and Tracing Tools
When the MPI standard specifications were first released in 1994, they included the description of a profiling interface (see chapter

8 in reference [1]). Each MPI function calls a PMPI internal function (e.g., MPI_Send calls PMPI_Send). This simple mechanism enables a portable method for MPI profiling and performance tools and is used for example by the perfcatch utility in the SGI Message Passing Toolkit (MPT) [2]. In addition to the accumulated time, some simple statistics are gathered for each MPI function call (number of calls, message sizes, etc.) and are dumped at the end of the run. This high-level information provides a quick summary of the MPI activity, but is disconnected from the application itself. For a given MPI call, it is not possible to identify the contribution of different call sites (source lines from which the function is called).

A second class of tools identifies the contributions of MPI calls at the line level. For example, mpiP is a "lightweight profiling library for MPI applications" [3] that requires re-linking the application. More sophisticated tools (e.g., Dynaprof [4]) instrument an MPI application at runtime without any recompilation or re-linking. These tools enable sampling of the various call sites for each MPI function.

The ultimate level of detail is provided by tracing tools such as the Intel Trace Analyzer [5] (based on the original Vampir tool [6]) or the new Vampir tool [7]. Using the PMPI interface, each MPI call is recorded and dumped into files. After post-processing the raw data and with the help of an interactive visual tool, the developer can replay the entire run and check every single MPI transfer.

Other tools attempt to find a middle ground in the degree of detail that is provided. For example, Scalasca [8] takes an incremental approach to performance analysis, using successive measurement refinements that can combine summary profiles with trace information. But Scalasca requires an initial recompilation of the code, which is not always possible or convenient.

In summary, the available tools will incrementally

- report the time spent in each MPI function,
- identify where in the code the MPI function is called,
- detail individual calls to that function.

The information that can be extracted with such tools is very useful to developers, but the sophistication of the graphical interface requires a certain level of expertise when dealing with a large number of cpus. In addition the size of the traces is an issue, especially at run time. In order not to interfere with the timing of the communications the trace must remain small enough to fit in a memory buffer. Reducing the scope of a job is absolutely necessary to make tracing manageable, and this reduction entails extra tedious effort and risk.

2.2 Bandela: modeling MPI communications

For a one millisecond MPI_Recv call, the hardware performance and network stress will be quite different if the message size is 8 Mbytes versus 8 bytes. In the first case, the limitation will certainly be the network hardware. In the latter case, no hardware characteristic can explain such a long duration for a one-word message. But this behavior could be easily understood if the message is posted late, well after the receiver process is blocked in MPI_Recv. Similarly, an MPI_Barrier or MPI_Allreduce call will be at least as long as the time difference between the first and the last calls. For imperfectly scheduled communications, there is an incompressible amount of time spent in MPI which does not depend on the communication network characteristics.

The majority of profiling tools we have discussed measure the duration of MPI calls, but it is also important to evaluate the impact of the communication synchronicity: the relative arrival times of the different calls involved in an MPI communication.

Although that information is available to the tracing tools for each individual MPI communication, the tools are not configured to estimate the global contributions. For this purpose we developed a research tool, Bandela [9], using the same tracing tool approach, but with the ability to play back MPI event traces using a crude bandwidth/latency model for point-to-point communications. The model assumes that all messages of length Nbytes are transferred in the time

$$T(Nbytes) = T_{latency} + Nbytes / Bandwidth$$

By adjusting the bandwidth and latency, one can estimate the importance of these basic network hardware parameters. In the extreme case, with zero latency and infinite bandwidth—the so-called perfect interconnect—all transfer times are reduced to zero. Yet, even in that ideal case, the unsynchronized MPI calls can generate substantial MPI time as seen in Figure 1.

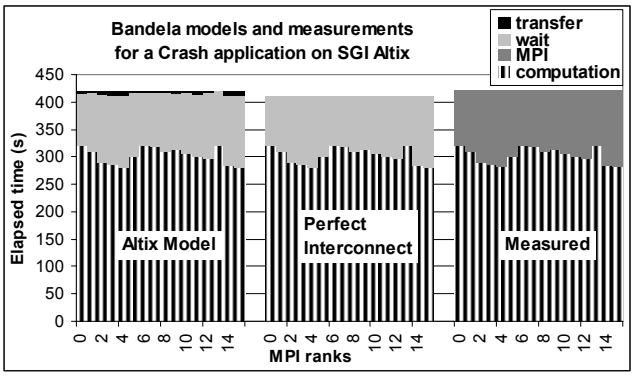

Figure 1. Comparing Bandela SGI Altix 4700 model (left), Bandela perfect interconnect (middle) and measurements (right) for a 16 CPU run.

The Bandela tool, even with its simple communication model, successfully explained the behavior of real-world MPI applications. By varying the communication latency and bandwidth, their relative importance could be evaluated. More often than not, we found the hardware communication parameters to be less important than commonly thought.

Bandela remained a research tool difficult for the average user to master. The trace files generated became unwieldy when running a large number of ranks. For iterative applications, the trace file sizes could be reduced by restricting the trace collection to a smaller time window, but that feature was too awkward to be practical. Although Bandela had an advantage compared to other tracing tools in that runtime overhead did not degrade the validity of the measurements, it still suffered from painfully long trace playbacks even for reduced jobs.

3. MPINSIDE DIAGNOSTIC FEATURES

3.1 MPInside specifications

Our experience with Bandela suggested that a new MPI performance tool was required—one that would provide the same insight as Bandela, but without the overhead. Thus was born MPInside, a tool designed to:

- be useable with thousands of ranks without overhead.
- work without traces and without post-processing. All processing should be performed in-flight.
- not require re-compilation or re-linking.
- use a simple command line interface, without the need for an interactive GUI.
- produce simple text output to be fed into spreadsheets or other scripts.
- be portable to any MPI library for its basic features.
- support the full MPI 1.2 specifications and the MPI-2 one-sided communications, MPI_Put, MPI_Get, MPI_Win_xx.
- handle various communication models, in particular the perfect interconnect (zero latency, infinite bandwidth).

3.2 Basic run

The MPInside tool is activated simply by inserting MPInside in the mpirun command line, e.g.:

```
mpirun -np 128 MPInside ./a.out args…
```

By default only a basic light-weight wrapping of the MPI functions is performed. The MPI profiling information is gathered into a single text file. The sample output for a 4-rank run in Figure 2 contains five tables: timing breakdown, total size sent, number of "send" calls, total size received, number of "recv" calls. Each table contains one entry per rank.

>>>>	Elapse	times	in	(s)	<<<<		
CPU	Comput	init	wait	send	irecv	bcast	overhead
0	7.7442	0.0486	2.1606	0.0000	0.0034	0.0146	0.0033
1	9.6735	0.0471	0.0000	0.2357	0.0000	0.0150	0.0020
2	7.7667	0.0458	0.2484	0.0000	0.0042	1.9045	0.0023
3	7.7251	0.0450	0.0000	0.2633	0.0000	1.9361	0.0016
>>>>	Kbytes	with	send	attribute	<<<<		
CPU	Comput	init	wait	send	irecv	bcast	overhead
0	------	0	0	0	0	0	0
1	------	0	0	400000	0	4000	0
2	------	0	0	0	0	0	0
3	------	0	0	400000	0	0	0
>>>>	Number	of	requests	with	Send	attribute	<<<<
CPU	Comput	init	wait	send	irecv	bcast	overhead
0	------	1	0	0	0	0	0
1	------	1	0	1000	0	1000	0
2	------	1	0	0	0	0	0
3	------	1	0	1000	0	0	0
>>>>	Kbytes	with	Recv	attribute	<<<<		
CPU	Comput	init	wait	send	irecv	bcast	overhead
0	------	0	0	0	400000	4000	0
1	------	0	0	0	0	0	0
2	------	0	0	0	400000	4000	0
3	------	0	0	0	0	4000	0
>>>>	Number	of	requests	with	Recv	attribute	<<<<
CPU	Comput	init	wait	send	irecv	bcast	overhead
0	------	0	1000	0	1000	1000	0
1	------	0	0	0	0	0	0
2	------	0	1000	0	1000	1000	0
3	------	0	0	0	0	1000	0

Figure 2. MPInside statistics basic output

Thanks to its simple format, the raw data is readily imported into a spreadsheet where it can be easily plotted (see Figure 3) and where derivative metrics such as average message size can be computed. We believe this provides a convenient and scalable solution to the data management problem inherent to large parallel jobs, and therefore describe the various array outputs in more detail.

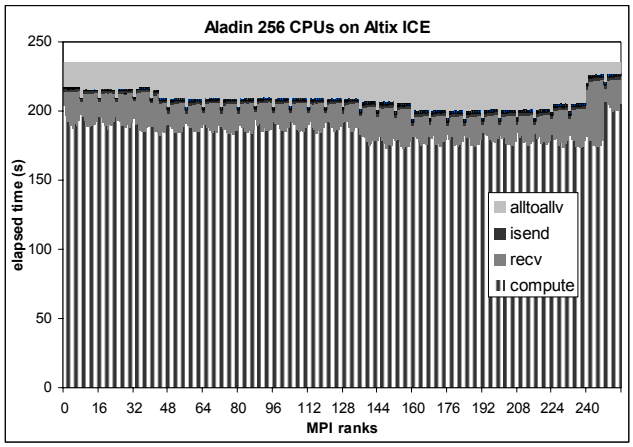

Figure 3. MPI profile from the MPInside elapsed time statistics

As shown in Figure 2, sizes and requests are split into a "send" set and a "receive" set. Note that for point-to-point communications, the tool reports the number of bytes physically transferred, not the size specified on the receive side.

For a collective operation such as MPI_Bcast, the size and increment are assigned as a send for the root of the broadcast and as a receive for the other ranks participating in the operation. To better match the actual data transfer for collective operations, e.g. MPI_Alltoall, the sizes reported are the buffer size multiplied by the number of ranks participating in the function.

Finally, the user may optionally request a report of the following transfer matrices, with one row and one column per rank representing

- TIME(i,j): the aggregate time rank "i" spent receiving data from rank "j";
- SIZE(i,j): the amount of data transferred from "i" to "j";
- REQUEST(i,j): the number of calls involved in these transfers.

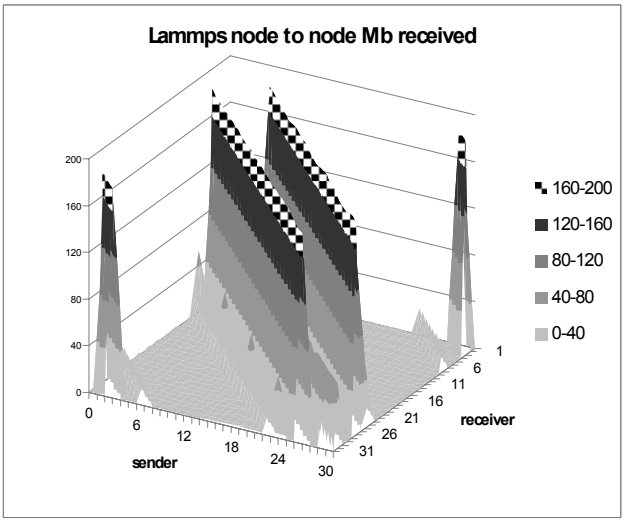

Figure 4. MPI Byte Transfer Matrix (x= sender rank, y=receiver rank, z=Mbytes transferred)

For example, Figure 4 plots the "SIZE" matrix as a 3D surface, revealing the cyclic pattern of the data transfers in the LAMMPS application.

3.3 Collective Wait time

As mentioned in section 2.2, a fraction of the time in MPI collective functions (e.g., MPI_Barrier or MPI_Allreduce) is spent waiting for the last rank to reach the rendezvous point. To evaluate the cost of these timing misalignments a call to MPI_Barrier is inserted before each MPI collective, as illustrated in Figure 5. Since it synchronizes all ranks, the inserted MPI_Barrier measures the collective wait time. The time in the subsequent MPI collective is assumed to be spent entirely in the physical transfer of data.

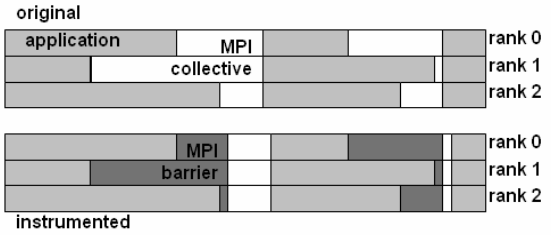

Figure 5. MPI_Barrier insertion for collective wait time

Such a strategy could be considered counter-productive and highly intrusive since it doubles the number of collective calls, and hence it is not enabled by default. In practice, however, the associated overhead generally is not noticeable and seldom adds more than a few percent to the application elapsed time.

In the particular cases of functions like MPI_Gather, MPI_Scatter or MPI_Reduce, the forced synchronization with the MPI_Barrier predecessor introduces some distortion at the root rank. All messaging is concentrated in a shorter time instead of being spread across the whole duration of the call. This can be notably seen for MPI_Reduce on figure 6. However, this is not a major drawback for a diagnostic tool compared to the valuable insight provided.

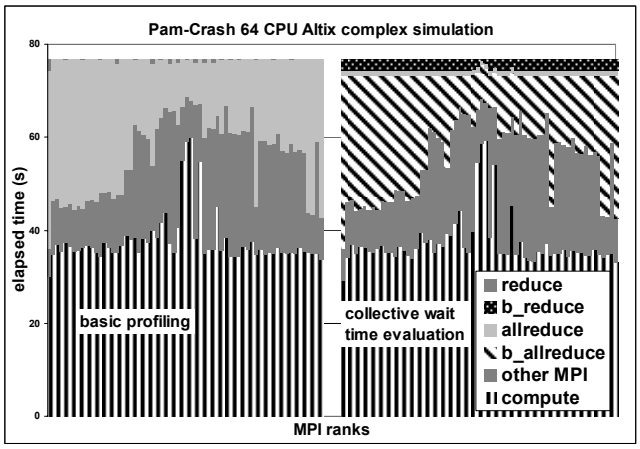

Figure 6. Basic profiling and Collective Wait Time example

Although the PAM-CRASH [12] application generally scales well to hundreds of ranks, the corner case shown in Figure 6 exhibits a high percentage of time in MPI_Allreduce (light grey on the left). With the scheme we just detailed most of that time is shown to be "collective wait" time ("b_allreduce", dark downward diagonal on the right), while the actual MPI_Allreduce time is insignificant. Hence this MPI "collective" time is primarily the result of application workload imbalance, which not even the mythical perfect interconnect would remedy.

3.4 Late Senders

3.4.1 Send Late Time

The MPI communication time, T_{comm}, for blocking functions such as MPI_Wait, MPI_Recv, or MPI_Send can be modeled as

$$T_{comm} = T_{wait} + T_{trans}$$

where T_{trans} is the physical transfer time and T_{wait} denotes any remaining time between the posting of the operation and the time it completes.

For MPI_Send or MPI_Isend/MPI_Wait couplets, T_{wait} represents "late" receivers. With sufficient buffers, however, the impact of such late receivers can be minimized, particularly on shared memory systems such as the SGI Altix 4700 [10]. This is less true on InfiniBand (IB) clusters such as the SGI Altix ICE [11], where the locked memory buffers allocated for IB are limited.

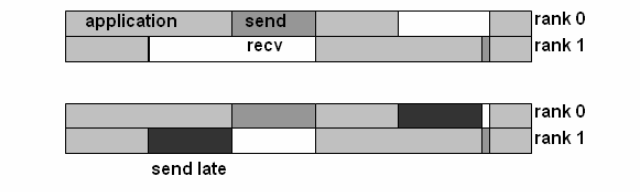

Figure 7. Transfer with Late Sender arrival

Conversely, for MPI_Recv or MPI_Irecv/MPI_Wait couplets T_{wait} is nonzero when the matching sender is late. This situation, which we refer to as "send late time" (SLT), is illustrated in Figure 7. Unlike the case for sends, this wait time cannot be avoided with any kind of buffering and hence is important to monitor.

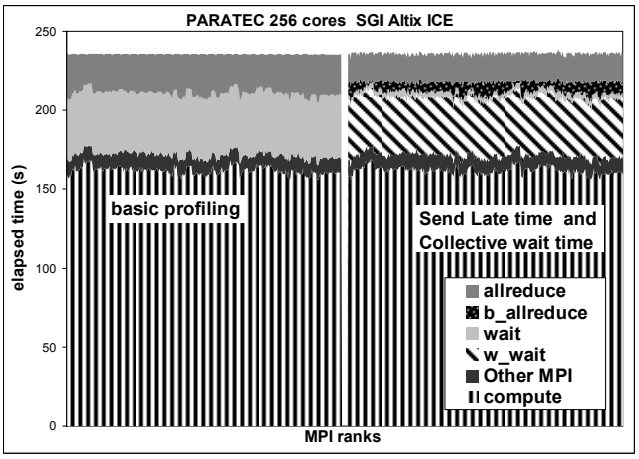

Figure 8. Basic MPI profiling and Send Late Time evaluation

Figure 8 shows MPInside results for Paratec [13] on a 256-core SGI Altix ICE system. The profile on the left was obtained with default MPInside settings and indicates that MPI_Allreduce and MPI_Wait are the most time-consuming MPI functions. The profile on the right illustrates both the collective wait time

evaluation described in section 3.3 and the send late time. The two charts have similar characteristics despite the changes the instrumentation introduced. In this case, most of the MPI_Allreduce time is spent transferring data (labeled "allreduce" on the right chart) rather than synchronizing (labeled "b_allreduce"). The MPI_Wait time is separated between the time ostensibly spent transferring (labeled "wait") and the send late time (labeled "w_wait"), the latter of which is dominant.

With this additional information, the Paratec developer will know that the application spends most of its time for point-to-point communications waiting rather than transferring. At this stage, she might also conclude that some micro-load imbalance in the computation is responsible for this staggered arrival on the MPI_Wait functions. The MPInside modeling capabilities (see section 4.2) will help identify the real cause of the problem.

Several additional metrics of interest can be derived from the transfer time estimations. For example, the average MPI_Recv *raw* bandwidth may be computed as:

$$BW_{raw} = \text{bytes\_received \_MPI\_Recv} / \text{recv\_transfer\_time}$$

This should be compared with the average *effective* bandwidth:

$$BW_{eff} = \text{bytes\_received \_MPI\_Recv} / \text{recv\_total\_time}$$

3.4.2 Clusters lack a universal clock

To estimate the send late times, MPInside initially relied on time stamps inserted into the point-to-point messages. For a MPI_Send/MPI_Recv couplet, the send late time was simply

$$SLT = \text{Max}(0; Tsend - Trecv)$$

This approach works well on a single system image (SSI) system such as the SGI Altix 4700 where a very precise (40 ns resolution) real-time synchronized clock is available. On clusters, however, the clocks for the different nodes are typically not synchronized with sufficient precision. To compensate for this deficiency, we attempted to estimate both the time difference between nodes and the time drift on each node. Unfortunately the precision of such calibration was unacceptably large, ranging from 2 to 20 µs depending on the number of ranks: the larger the MPI communicator size, the larger the imprecision of the "universal" clock.

3.4.3 Stuttering communications

To overcome the lack of universal clock we implement a "stuttering" technique in which a short tagged message is posted in front of each point-to-point message. Thus the receiver will see two messages: the short one, followed by the original payload. The send late time is simply estimated as the time the receiver had to wait for the tagged message minus a fixed delay calibrated during initialization. This method uses only local time, thereby removing the necessity of a universally synchronized clock.

Consider the following simple sequence:

Rank 0	Rank 1
1-MPI_Isend(to 1)	MPI_Isend(to 0)
2-MPI_Recv(from 1)	MPI_Recv(from 0)
3-MPI_Wait (Isend)	MPI_Wait(Isend)

The stuttering technique will transform it into:

Rank 0	Rank 1
1a-MPI_Isend(tag to 1)	MPI_Isend(tag to 0)
1b-MPI_Isend(data to 1)	MPI_Isend(data to 0)
2a-MPI_Recv(tag from 1)	MPI_Recv(tag from 0)
2b-MPI_Recv(data from 1)	MPI_Recv(data from 0)
3a-MPI_Wait (Isend tag)	MPI_Wait(Isend tag)
3b-MPI_Wait (Isend data)	MPI_Wait(Isend data)

In applications for which communication times are predominantly spent waiting for unsynchronized arrivals, the distortion introduced by this approach is minimal. Under these conditions the supplemental zero-size messages have negligible impact. The supplemental messages have a similarly small effect on highly synchronized applications provided that the communications are primarily bandwidth-limited. For latency-sensitive applications, on the other hand, the stuttering approach may bias the measurements. Hence a good practice when using the advanced features of MPInside is to compare the results with those obtained with basic profiling to ensure that no unexpected artifacts were introduced.

3.5 Coupling MPInside with application hardware counter profiling

A complete application analysis requires not only a communications profile, but also detailed information on the computational portion of the job. Hence for this task MPInside must be coupled with an application profiling tool. If simply gathering timing information with a PC sampling experiment, one can easily exclude the sampled MPI or MPInside routines from the profiling results. But it is not such a straightforward matter to exclude the MPI calls in a hardware performance counter event experiment. MPInside has been integrated with the NCSA Perfsuite [14] tool to do just that—hardware event counting will stop when entering an MPI function and then restart upon returning to the application.

4. MPINSIDE APPLICATION RUNTIME MODELING

There are a number of possible approaches to parallel applications performance modeling. Some use an analytical method to model the application computation and communication in order to predict performance on future systems, e.g. Kerbyson et al. [16]. Others, such as the Performance Modeling and Characterization framework (PMac) [17], record an application signature which can then be convolved with the target machine profile.

MPInside uses an in-flight modeling approach to estimate the application behavior on any hypothetical platform. Each input case and cpu count of interest requires a separate simulation in order to account for differences in MPI event sequences. Since the modeling calculations are incorporated into the MPI library calls, the computational portion of the application still runs undisturbed at full speed and in parallel. With this strategy the modeling overhead is constrained to the MPI calls, such that even for the most complex models the application runtimes are typically not increased by more than 20%.

Although the principles described in the following sections are relatively simple and general, the implementation requires

detailed knowledge of MPI library internals and can therefore become quite complex. As of this time the MPInside modeling capability is available only with the SGI MPI library, but our investigations suggest that this capability could be extended to other MPI libraries such as MPICH2 [18].

4.1 Virtual clocks for virtual systems

As mentioned in section 3.4.3, MPInside maintains a local clock which measures the application events as they occur. The MPInside modeling feature adds one *virtual* clock per simulated system. At every MPI event in the application, each virtual clock is incremented according to its platform-specific model.

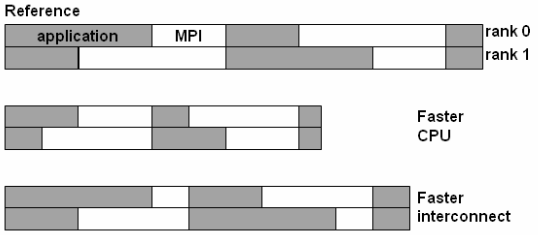

Figure 9. MPI application modeling

Figure 9 illustrates the principle of the MPInside modeling for a two-rank MPI run. The topmost timeline displays the events as they are recorded by the real clock; underneath are two virtual timelines for systems with the following characteristics:

- (middle) a twice faster processor than the reference, but with the same interconnect;
- (bottom) the same processor as the reference, but with a twice faster interconnect.

Because virtual timestamps are inserted into each MPI message, the receivers can easily compute the point-to-point send late times as the difference between the virtual send and receive times. Similarly the collective wait time is the difference between the maximum of all timestamps with those of each rank.

Computation time outside MPI is measured using hardware clocks which are typically not synchronized across ranks, but this poses no problems since application runtime modeling does not require an initial synchronization at time 0. Time drifts may matter, but virtual clocks are one-to-one synchronized at each MPI send/recv exchange, and all clocks are synchronized at most collective MPI operations.

4.2 Perfect interconnect

MPInside uses the Bandela formula from section 2.2 to estimate MPI transfer time as the sum of a latency component and the message size divided by the bandwidth.

For a perfect interconnect the latency is zero and the bandwidth is infinite, thus all message transfer times are null. However, even in this ideal case the MPI wait time can still be substantial (recall Figure 1).

We now return to the Paratec example from section 3.4. Figure 10 compares the measured MPI profile with the simulated perfect interconnect. As expected from the "collective wait" estimation, most of the time in MPI_Allreduce disappears in the perfect case. However, comparison to Figure 8 reveals that the MPI_Wait time has been significantly reduced, thus pointing to a hardware limitation rather than any micro-load imbalance in the application.

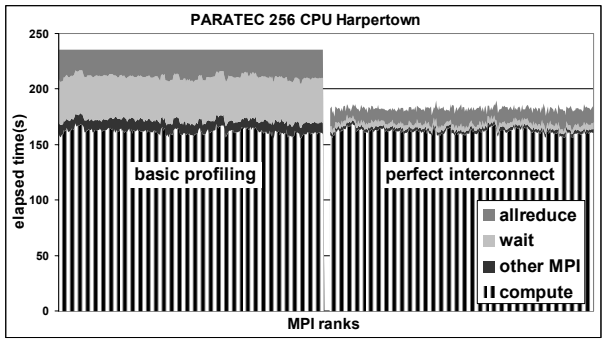

Figure 10. Paratec measured basic MPI profiling and Perfect Interconnect model

The perfect interconnect, though physically unrealizable, is an extremely useful modeling tool. Because the transfer time requires no approximations, the profile can be computed with high accuracy. It provides an upper bound for application performance and is therefore quite valuable in assessing the impact of the communication network.

4.3 Changing MPI latency and Bandwidth

The bandwidth-latency transfer model introduced in section 2.2,

$$T(Size) = T_{Latency} + Size / Bandwidth$$

works well on SGI Altix shared memory systems. One can easily adjust the two parameters to fit the measurements, and then modify them to simulate various configurations.

On gigabit Ethernet or InfiniBand clusters, we found that this model required substantial refinements. First, a cluster is quite non-uniform for MPI communications—it is much more efficient within the node (shared memory) than across nodes (network). Second, the inter-node transfer bandwidth is a function of the message size. Finally, the latency and, more importantly, the raw bandwidth are impacted by the network load. Neglecting the effect of the load on latency, this is summarized with

$$T(Size) = T_{Latency} + Size / Bandwidth\ (Size,load)$$

The network load is a highly variable global component: the effect on a certain transfer will depend on all other communications using the same network subset. As an approximation we assume that the global load will vary as the number of live requests on the local node. The transfer time formula becomes

$$T(Size) = T_{Latency} + (Size / Bandwidth\ (Size)) * LR * DR$$

where LR represents the number of local live requests and DR is a fudge factor, the "degradation ratio".

Depending of the application, the degradation ratio typically ranges between 0.1 (minimal effect) to 20 (saturated network). Multiple DR values can be specified in a single modeling run, thereby providing a convenient means to bracket the actual measurements. The DR itself has no absolute meaning, only a relative one: it allows the model to be tuned to match the measurements. This tuned model can then serve as a basis for studying configurations different than that on which the application was executed. The next section provides an example of this.

5. CASE STUDY

LAMMPS [19] is a well-known scalable molecular dynamics code. On SGI Altix ICE systems, a particular LAMMPS

benchmark with a small fixed-size problem scales up to 128 ranks; beyond that it slows down and demonstrates negative scaling.

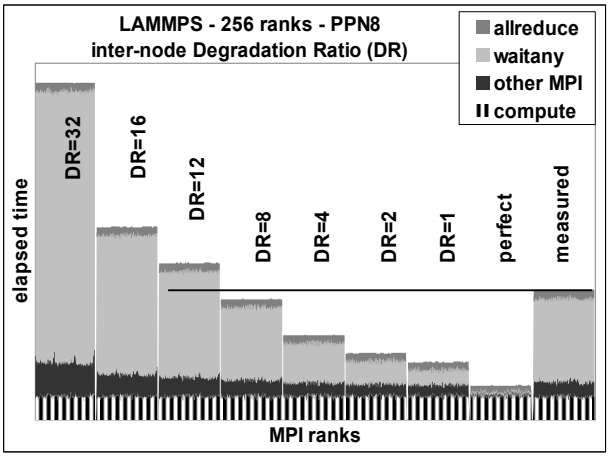

Figure 11. Small test case model outputs for LAMMPS on the SGI Altix ICE cluster, 8 processes per node (PPN8)

Figure 11 shows several MPInside model profiles (the left charts) and the measured profile (the rightmost chart) for this application at 256 processors using 8 processes per node (PPN8). In this case the MPI engine accounts for a significant portion of the overall runtime: a perfect interconnect would provide a four-fold improvement in overall performance. All the model scenarios in the figure use the same experimentally-derived Bandwidth(Size) function; what varies are the inter-node degradation ratios. A degradation ratio of approximately 10 provides the best fit to the measured profile.

Figure 12 shows the 256-process LAMMPS model results using 4 processes per node (PPN4); note the marked reduction in the measured elapsed time compared to the PPN8 case. For many applications PPN4 substantially improves the computation time due to reduced memory contention. However, since this LAMMPS benchmark demonstrates very good cache locality, there is negligible difference in the computation time between the PPN4 and PPN8 cases.

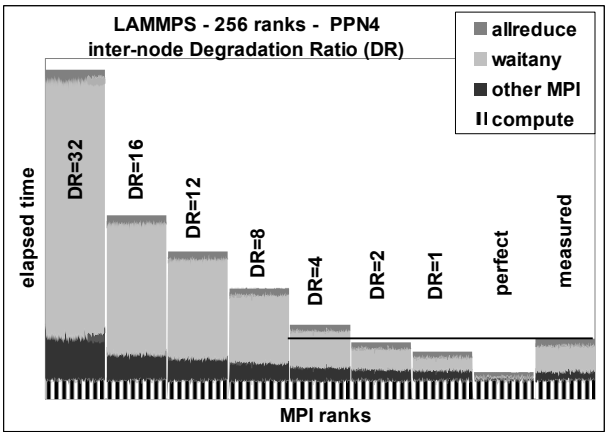

Figure 12. Model outputs for LAMMPS on the SGI Altix ICE cluster, 4 processes per node (PPN4)

The drastic drop in elapsed time in the PPN4 case is due entirely to improved communication performance, as the MPInside model best-fit degradation ratio drops substantially from 10 to 2. This correlation between processes per node and MPI engine load merits further investigation.

Table 1. LAMMPS Node 0 off-node activity

	PPN4	PPN8
Mb received from remote nodes	533	542
Number of remote requests per node	325 581	372 231

Table 1 summarizes the off-node communications extracted from the MPInside transfer matrices. Although from the perspective of a single node there is slightly less off-node activity with PPN4 than with PP8, at the global level there is far more off-node activity with PPN4 than with PP8 because there are double the number of nodes in the PPN4 case. Hence with the same computational profile and the same nominal interconnect performance, the application should run at least as well if not better with PPN8 than with PPN4.

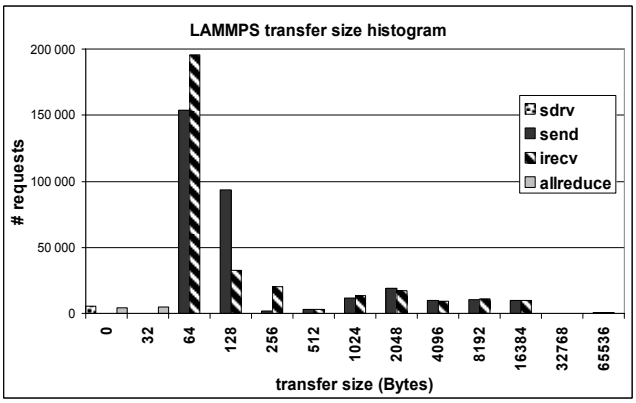

Figure 13. LAMMPS rank 0 size distribution

The size histogram extracted from the MPInside statistics output file (Figure 13) reveals that most of the message exchanges are very short (<128 bytes). This suggests that the large MPI overhead with PPN8 is due to inefficiency in the MPI engine when processing a flurry of small messages.

Following this analysis we modified the LAMMPS code in order to coalesce the inter-node messages, thereby minimizing the number of MPI requests on the interconnection network. This approach resulted in a two-fold reduction in the global elapsed time for the PPN8 case.

As a final experiment, we ran the original LAMMPS code on an SGI Altix 4700 with Intel Itanium2 (Montecito) processors (Figure 14). Compared to the SGI Altix ICE system with Intel Xeon (Nehalem) processors, the global performance on the Altix 4700 is the same despite the fact that the application's computational routines run three times faster on Nehalem than on Montecito. On the shared-memory Altix 4700, the communications do not suffer from the same bottlenecks that the InfiniBand hardware components present.

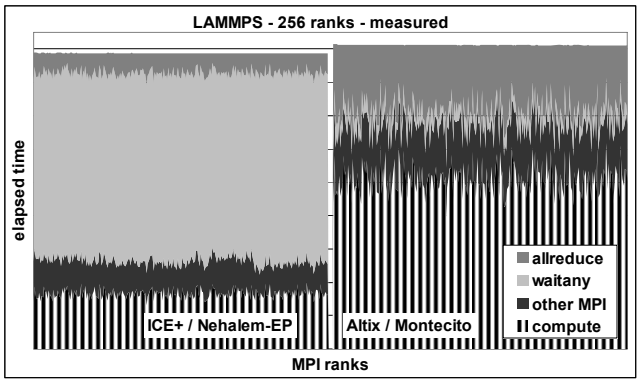

Figure 14. LAMMPS fixed-size 256 CPU relative performance on ICE/Nehalem and Altix/Montecito

Based on the Altix 4700 measurements, the MPInside model indicates that a hypothetical future system with a Nehalem processor and an interconnect that is twice as fast as the Altix NUMAlink4 will run this unmodified LAMMPS test case with a performance close to that of the perfect interconnect (Figure 15).

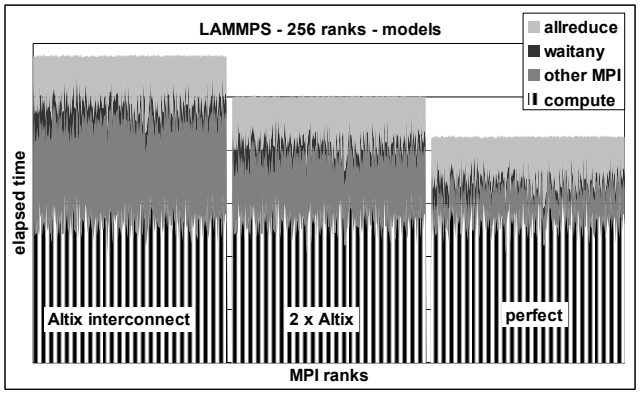

Figure 15. Expected performance on LAMMPS standard for a hypothetical shared memory system

6. CONCLUSION

In this paper we have introduced MPInside and described its basic features, the primary of which allow the impact of unaligned MPI events to be measured without distortion.

Further development work, already in progress, allows:

- characterization of communication event dependencies,
- profiling for unaligned MPI events, and
- cross-referencing of MPI call stacks across ranks.

Together with its ease of use, low overhead, and modeling capabilities, MPInside constitutes a powerful analysis and diagnostic tool to meet the challenges of peta-scale MPI application development.

7. REFERENCES

[1] MPI: a Message-Passing Standard

http://www.mpi-forum.org/docs/mpi-10.ps

[2] Perfcatch utility tool for SGI's Message Passing Toolkit http://www.sgi.com/products/software/mpt/

[3] Vetter, J.S. and M.O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP), 2001

[4] Mucci, P. DynaProf http://www.cs.utk.edu/~mucci/dynaprof/

[5] Intel® Trace Analyzer and Collector http://software.intel.com/en-us/intel-trace-analyzer

[6] Nagel, W., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. 1996. "VAMPIR: Visualization and Analysis of MPI Resources". *Supercomputer* 12(1):69–80.

[7] Vampir - Performance Optimization http://www.vampir.eu

[8] Scalasca (Scalable Performance Analysis of Large-Scale Applications), http://www.fz-juelich.de/jsc/scalasca/

[9] Tanasescu, C. "Scalability Considerations for Compute Intensive Applications on Clusters". LCI, 2003. http://www.linuxclustersinstitute.org/conferences/archive/2003/PDF/P02-Tanasescu_C.pdf

[10] SGI Altix 4700 servers http://www.sgi.com/products/servers/altix/4000/

[11] SGI Altix ICE servers http://www.sgi.com/products/servers/altix/ice/

[12] PAM-CRASH http://www.esi-group.com/products/crash-impact-safety/pam-crash

[13] Paratec (PARAllel Total Energy Code http://www.nersc.gov/projects/paratec/

[14] PerfSuite is a collection of tools, utilities, and libraries for software performance analysis: http://perfsuite.ncsa.uiuc.edu/

[15] MPT (Message Passing Toolkit) http://www.sgi.com/products/software/mpt/

[16] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, M. Gittings, "Predictive Performance and Scalability Modeling of a Large-Scale Application," sc, pp.39, ACM/IEEE SC 2001 Conference (SC 2001), 2001

[17] PMaC: Performance Modeling and Characterization http://www.sdsc.edu/pmac/index.html and L. Carrington, A. Snavely, X.Gao, and N. Wolter. A performance prediction framework for scientific applications. In International Conference on Computational Science Workshop on Performance Modeling and Analysis (PMA03), pages 926–935, June 2003.

[18] MPICH2 a high-performance and widely portable implementation of MPI http://www.mcs.anl.gov/research/projects/mpich2/

[19] LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator.): http://lammps.sandia.gov/