Impact of Noise on Scaling of Collectives: An Empirical Evaluation

Rahul Garg and Pradipta De

IBM India Research Laboratory, Hauz Khas, New Delhi 110 016,
grahul@in.ibm.com, pradipta.de@in.ibm.com

Abstract. It is increasingly becoming evident that operating system interference in the form of daemon activity and interrupts contribute significantly to performance degradation of parallel applications in large clusters. An earlier theoretical study has evaluated the impact of system noise on application performance for different noise distributions [1]. Our work complements the theoretical analysis by presenting an empirical study of noise in production clusters. We designed a parallel benchmark that was used on large clusters at SanDeigo Supercomputing Center for collecting noise related data. This data was fed to a simulator that predicts the performance of collective operations using the model of [1]. We report our comparison of the predicted and the observed performance. Additionally, the tools developed in the process have been instrumental in identifying anomalous nodes that could potentially be affecting application performance if undetected.

1 Introduction

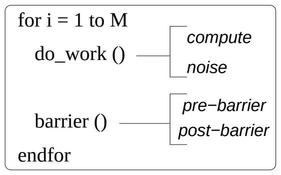
Scaling of parallel applications on large high-performance computing systems is a well known problem [2–4]. Prevalence of large clusters, that uses processors in order of thousands, makes it challenging to guarantee consistent and sustained high performance. To overcome variabilities in cluster performance and provide generic methods for tuning clusters for sustained high performance, it is essential to understand theoretically, as well as using empirical data, the behavior of production mode clusters. A known source of performance degradation in large clusters is the *noise in the system* in the form of daemons and interrupts [2, 3]. Impact of OS interference in the form of interrupts and daemons can even cause an order of magnitude performance degradation in certain operations [2, 5].

A formal approach to study the impact of noise in these large systems was initiated by Agarwal et al. [1]. The parallel application studied was a typical class of kernel that appears in most scientific applications. Here, each node in the cluster is repetitively involved in a computation stage, followed by a collective operation, such as barrier. This scenario was modeled theoretically, and impact of noise on the performance of the parallel applications was studied for three different types of noise distributions.

In this paper, our goal is to validate the theoretical model with data collected from large production clusters. Details revealed through empirical study helps in fine-tuning the model. This allows us to establish a methodology for predicting the performance of large clusters. Our main contributions are: (i) We have designed a parallel benchmark that measures the noise distribution in the cluster; (ii) Using data collected from the production clusters we make performance predictions made by the theoretical model proposed earlier. This validation step enables us to predict the performance of large clusters. We report the prediction accuracy against measurements at the SanDiego Supercomputing Center (SDSC). We discovered that measurements of noise distributions also help in identification of misbehaving nodes or processors.

In addition to making performance predictions, our study could be useful in performance improvements. Traditional techniques for performance improvement either fall in the category of *noise reduction* or *noise synchronization*. Noise reduction is achieved by removing several system daemons, dedicating a spare processor to absorb noise, and reducing the frequency of daemons. Noise synchronization is achieved by explicit co-scheduling or gang scheduling [6–8]. Most of these implementations require changing the scheduling policies. Our work gives insight into another technique for improving performance, that can be called *noise smoothing*. If the model predicts the actual performance reasonably well, then the systems can be tuned to ensure that the noise does not have heavy tail (i.e. infrequent interruptions that take long time). This technique may complement the other approaches currently used in large high-performance systems.

The rest of the paper is organized as follows. The theoretical model for capturing the impact of noise on cluster performance, based on [1], is presented in Section 2. In Section 3, we present the details of the parallel benchmark that we have designed. Section 4 presents the analysis of the data collected on the SDSC clusters. Finally, we conclude in Section 5.



```
for i = 1 to M
  do_work ()
  barrier ()
endfor
```

Fig. 1. *Typical code block in a parallel scientific application.*

2 Theoretical Modeling of System Noise

In this section, we briefly introduce the theoretical model of collective communication, as described earlier in [1]. In this model, a parallel program consists of a sequence of iterations of a *compute phase* followed by a *communicate phase*, as shown in Figure 1. In the compute phase, all the threads of the program locally carry out the work assigned to them. There is no message exchange or I/O activity during the compute phase. The communicate phase consists of a collective operation such as a barrier. We are interested in understanding the time it takes to perform the collective operation as a function of the number of threads in the system.

Consider a parallel program with N threads running on a system that has N processors. We assume, for simplicity of analysis, that $N = 2^k - 1$ for some positive integer

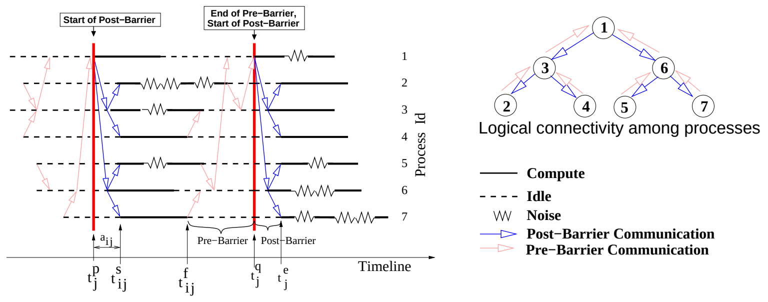


Fig. 2. *The diagram shows a typical computation-barrier cycle, along with pre-barrier and post-barrier phases for a barrier call, and interruptions in compute phase due to system noise.*

k. Figure 2 shows the sequence of events involving one iteration of the loop in Figure 1. In this figure, we used 7 processors which are logically organized as a binary tree to demonstrate the operation of one iteration of the parallel code. Figure 2 shows the decomposition of the communicate phase into pre-barrier and post-barrier stages, and the interruptions introduced during compute phase by the triggering of noise. In the figure, t_{ij}^s denotes the start time of the compute phase, t_{ij}^f denotes the finish time of the compute phase, and t_{ij}^e denotes the end of the barrier phase of the iteration.

2.1 Modeling the Communication Phase

The barrier operation comprises of two stages: a *pre-barrier* stage succeeded by a *post-barrier* stage. We assume that these stages are implemented using message passing along a complete binary tree, as shown in Figure 2. The basic structure does not change for any implementation based on a fixed tree of bounded degree, such as a k-ary tree. A process is associated to each node of the binary tree. A special process, called the *root* (Process 1 in Figure 2) initiates the *post-barrier* stage and concludes the *pre-barrier* stage. In the post-barrier stage, a start-compute-phase message initiated by root is percolated to all leaf nodes. At the end of computation each node notifies its parents of completion of work. This stage ends when the root finishes its computation and receives a message from both its children indicating the same. An iteration of the loop in Figure 1 would thus consist of a compute stage, followed by a pre-barrier and a post barrier stage. Let t_j^p denote the start of the post-barrier stage just preceding the j-th iteration, and t_j^q denotes the time at which the post-barrier stage of the j-th iteration concludes, as shown in Figure 2. Following t_j^q, the iteration can continue with other book-keeping operations before beginning the next iteration. Also, the time taken by a barrier message to reach node i in the post-barrier phase is denoted by a_{ij}.

For simplicity, we assume that each message transmission between a parent and a child node takes time τ, which is referred to as the *one-way latency*. Thus, for the root process (process id 1) this value is zero, i.e. $a_{1j} = 0$ in Fig 2. For all the leaves i, $a_{ij} = \tau(\log(N+1) - 1)^1$. Thus, for the case of $N = 7$ in the figure, $a_{2j} = 2\tau$, $a_{3j} = \tau$, $a_{4j} = 2\tau$, $a_{5j} = 2\tau$, $a_{6j} = \tau$, and $a_{7j} = 2\tau$, for all j.

2.2 Modeling the Compute Phase

Let W_{ij} represent the amount of work, in terms of number of operations, carried out by thread i in the compute phase of j-th iteration. If the system is *noiseless*, time required by all processors to finish the assigned work will be constant, i.e. time to complete work W_{ij} is w_{ij}. The value of the constant typically depends on the characteristics of the processor, such as clock frequency, architectural parameters, and the state of the node, such as cache contents. Therefore, in a noiseless system the time taken to finish the computation is, $t^f_{ij} - t^s_{ij} = w_{ij}$.

Due to presence of system level daemons that are scheduled arbitrarily, the wall-clock time taken by processor i to finish work W_{ij} in an iteration varies. The time consumed to service daemons and other asynchronous events, like network interrupts, can be captured using a variable component δ_{ij} for each thread i in j-th iteration. Thus, the time spent in computation in an iteration can be accurately represented as,

$$t^f_{ij} - t^s_{ij} = w_{ij} + \delta_{ij}$$

where δ_{ij} is a random variable that captures the overhead incurred by processor i in servicing the daemons and other asynchronous events. Note that δ_{ij} also includes context switching overheads, as well as, time required to handle additional cache or TLB misses that arise due to cache pollution by background processes. The characteristics of the random variable δ_{ij} depends on the work W_{ij}, and the system load on processor i during the computation. The random variable δ_{ij} models the *noise* on processor i for j-th iteration, shown as the noise component in Figure 2.

2.3 Theoretical Results

The theoretical analysis of [1] provides a method to estimate the time spent at the barrier call. We will show here that the time spent at the barrier can be evaluated indirectly by measuring the total time spent by a process in an iteration, that consists of the compute and communicate phases. The time spent by a process in an iteration may be estimated by the amount of work, noise distributions and the network latencies.

In this analysis we make two key assumptions of stationarity and spatial independence of noise. Since we assume that our benchmark is run in isolation, therefore only noise present is due to system activity. This should stay constant over time, giving stationarity of the distribution. Secondly, the model in [1] assumes that the noise across processors is independent (i.e. δ_{ij} and δ_{kj} are independent for all i, j, k). Thus there cannot be any co-ordinated scheduling policy to synchronize processes across different nodes.

The time spent idling at the barrier call is given by,

$$b_{ij} = t^e_{ij} - t^f_{ij} \tag{1}$$

[1] From here on, log refers to log_2 and ln refers to log_e

We first derive the distribution of $(t_j^q - t_j^p)$, and then use it to derive the distribution of b_{ij}. From the figure it can be noted that, $(t_j^q - t_j^p)$ depends on a_{ij}, w_{ij}, and the instances of the random variable δ_{ij}. Now, if the network latencies are constant (τ), it is easy to verify that, if $a_{ij} \leq \tau \log((N+1)/2), \forall (i,j)$,. Thus we have,

Lemma 1. *For all j, $\max_{i\in[1\ldots N]} t_{ij}^f - t_{ij}^s \leq t_j^q - t_j^p \leq \max_{i\in[1\ldots N]}(t_{ij}^f - t_{ij}^s) + 2\tau \log((N+1)/2)$.*

We model $(t_j^q - t_j^p)$ as another random variable θ_j. Now, Lemma 1 yields,

Theorem 1. *For all iterations j, the random variable θ_j may be bounded as*[2]*,*

$$\max_{i\in[1\ldots N]}(w_{ij} + \delta_{ij}) \leq \theta_j \leq \max_{i\in[1\ldots N]}(w_{ij} + \delta_{ij}) + 2\tau \log((N+1)/2).$$

For a given j, all δ_{ij} are independent for all i. Thus if we know the values of w_{ij} and the distributions of δ_{ij}, then the expectations as well as the distributions of θ_j may be approximately computed as given by Theorem 1. For this, we independently sample from the distributions of $w_{ij} + \delta_{ij}$ for all i and take the maximum value to generate a sample. Repeating this step a large number of times gives the distribution of $\max_{i\in[1\ldots N]}(w_{ij} + \delta_{ij})$. Now, b_{ij} can be decomposed as (see Figure 2)

$$b_{ij} = (t_j^q - t_j^p) - (t_{ij}^f - t_{ij}^s) - (a_{i(j+1)} - a_{ij}) \tag{2}$$

Since we have assumed a fixed one-way latency τ, $a_{ij} = a_{i(j+1)} = \tau$, therefore distribution of barrier time b_{ij} is given by,

$$\theta_j - (w_{ij} + \delta_{ij})$$

Using Theorem 1, θ_j can be approximately computed to within $2\tau \log((N+1)/2)$ just by using w_{ij} and δ_{ij}. Therefore, the barrier time distribution can be computed just by using noise distribution and w_{ij}. If w_{ij} are set to be equal for i, then w_{ij} cancels out, and barrier time distribution can be approximated just by using δ_{ij}.

In this paper we attempt to validate the above model by comparing the measured and predicted performance of the barrier operations on real systems. We evaluate if Theorem 1 can be used to give a reliable estimate of collectives performance on a variety of system. For this, we designed a micro-benchmark that measures the noise distributions $\delta_i(w)$. The benchmark also measures the distribution of θ_j, by measuring $t_{ij}^e - t_{ij}^s$. We implemented a simulator that takes the distributions of $w_{ij} + \delta_{ij}$ as inputs, and outputs the distribution of $\max_{i\in[1\ldots N]}(w_{ij} + \delta_{ij})$. We compare the simulation output with the actual distribution of θ_j obtained by running the micro-benchmark. We carry out this comparison on the Power 4 cluster at SDSC with different values of work quanta, w_i.

3 Methodology for Empirical Validation

Techniques to measure noise accurately is critical for our empirical study. This section presents the micro-benchmark kernel used to measure the distributions. We first tested this benchmark on a testbed cluster and then used it to collect data on the SDSC cluster.

[2] For random variables, X and Y, we say that $X \leq Y$ if $P(X \leq t) \geq P(Y \leq t), \forall t$.

Algorithm 1 The Parallel Benchmark kernel

```
1: while elapsed_time < period do
2:     busy-wait for a randomly chosen period
3:     MPI_Barrier
4:     $t_{ij}^s$ = get_cycle_accurate_time
5:     do_work ( iteration_count )
6:     $t_{ij}^f$ = get_cycle_accurate_time
7:     MPI_Barrier
8:     $t_{ij}^e$ = get_cycle_accurate_time
9:     store $(t_{ij}^f - t_{ij}^s), (t_{ij}^e - t_{ij}^f), (t_{ij}^e - t_{ij}^s)$
10:    MPI_Bcast (elapsed_time);
11: end while
```

3.1 Parallel Benchmark (PB)

The Parallel Benchmark (PB)[3] aims to capture the compute-barrier sequence of Figure 1. The kernel of PB is shown in Algorithm-1. The PB executes a compute process (Line 5) on multiple processors assigning one process to each processor. The *do_work* can be any operation. We have chosen it to be a Linear Congruential Generator (LCG) operation defined by the recurrence relation,

$$x_{j+1} = (a * x_j + b) \; mod \; p. \tag{3}$$

A barrier synchronization call (Line 7) follows the fixed work (W_i). The time spent in different operations are collected using cycle accurate timers and stored in Line 9. In the broadcast call in Line 10 rank zero process sends the current elapsed time to all other nodes ensuring that all processes terminate simultaneously. Daemons are usually invoked with a fixed periodicity which may lead to correlation of noise across iterations. The random wait (Line 2) is intended to reduce this correlation. The barrier synchronization in Line 3 ensures that all the processes of the PB commence simultaneously.

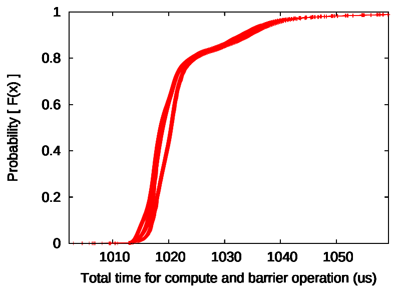
Since the benchmark measures the distributions $\delta_i(W)$ for a fixed W we omit the subscript j that corresponds to the iteration number in the subsequent discussion.

3.2 Testing the Parallel Benchmark

We first study the PB on a testbed cluster. The testbed cluster has 4 nodes with 8 processors on each node. It uses identical Power-4 CPUs on all nodes. IBM SP switch is used to connect the nodes. The operating system is AIX version 5.3. Parallel jobs are submitted using the LoadLeveler[4] and uses IBM's Parallel Operating Environment (POE). The benchmark uses MPI libraries for communicating messages across processes.

The goal in running on the testbed cluster was to fine-tune the PB for use on larger production clusters. We first execute the code block as shown in Algorithm 1 on the

[3] We refer this benchmark as PB, acronym for Parallel Benchmark, in the rest of the paper.
[4] The LoadLeveler is a batch job scheduling application and a product of IBM.



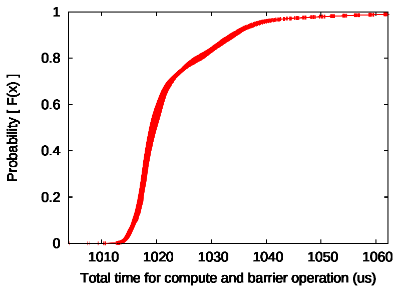


Fig. 3. *Distribution of time taken in an iteration for computation and barrier using the Parallel Benchmark code shown in Fig 1.*

Fig. 4. *Distribution of time taken in an iteration for computation and barrier operation using random wait, but* no broadcast *during execution.*

testbed cluster, and record the distributions of $(t_i^f - t_i^s) \sim w_i + \delta_i$, $t_i^e - t_i^f = b_i$, and $(t_i^e - t_i^s) \sim \theta$. Figure 3 shows the distribution of $\theta \sim (t_j^e - t_j^s)$ (Section 2.3) for PB. Ideally, the distributions for all processes (i.e. for all i) should exactly match, assuming $a_{i(j+1)} = a_{ij}, \forall i, j$. Interestingly, whenever broadcast was enabled in our experiment, the distributions were different. There were 4 bands formed, which we correspond to the 4 different nodes of the cluster. Figure 4 shows the distributions are identical when the broadcast in Line 10 of Figure 1 was omitted.

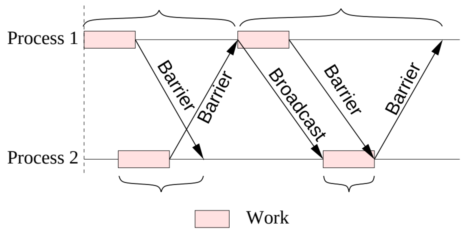


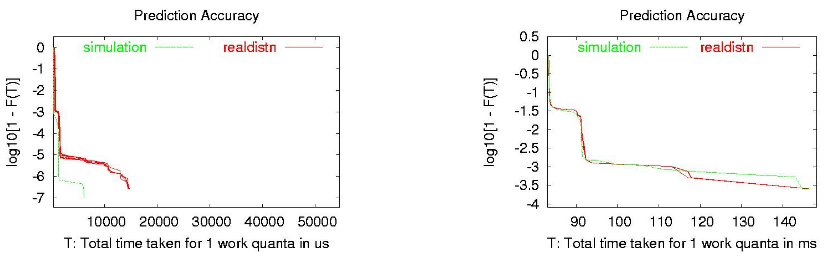
Fig. 5. *Sequence of message exchange in a work-barrier-broadcast loop.*

In order to explain this discrepancy we closely looked at the implementation of barrier. Barrier is implemented in two steps: a shared memory barrier which synchronizes all the processes on a node, followed by message passing implementation that synchronizes across nodes. The message passing implementation across the nodes is not based on binary tree as assumed in the model described earlier. We illustrate the message passing mechanism for barrier synchronization in Figure 5 for a case with two processes. As soon as a barrier is called on a process, a message is sent to the other process. The barrier call ends when the process receives a message from the other process. Broadcast is implemented by sending a single message. Figure 5 shows the message exchange between the two processes, and the calculation of total time for an iteration. It can be seen that if process 2 starts its work after process 1 then it consistently measures smaller time per iteration. Adding random wait, desynchronizes the start time and mitigates the above anomaly.

Finally, we also verified the stationarity assumption by executing PB multiple times at different times of the day. As long as the PB is executed in isolation without any other user process interrupting it the results stay unchanged.

3.3 Predicting Cluster Performance

We implemented a simulator that repeatedly collects independent samples from N distributions of $w_i + \delta_i$ (as measured by the PB), and computes the distribution of $\max_{i\in[1\ldots N]}(w_i + \delta_i)$.



(a) (b)

Fig. 6. Comparing distribution of $\max_{i\in[1\ldots N]}(w_i + \delta_i)$ against distributions of $t_i^e - t_i^s$ computed from empirical data on 32 processors of a testbed cluster, for (a) $w = 300\mu s$ and (b) $w = 83ms$.

Figure 6(a) and Figure 6(b) show the distributions of the time taken by an iteration (θ) on 32 processors in our testbed cluster, along with the output of the simulator ($\max_{i\in[1\ldots N]}(w_i + \delta_i)$). Two different quanta values (w) of $300\mu s$ and $83ms$, were used to model small and large choice of work respectively. Since, in the simulation the communication latency involved in the collective call is not accounted, hence the output of the simulator is lower than the real distribution.

Interestingly, the accuracy of the prediction with larger quanta values is better even without accounting for the communication latency. This is because when the quanta value (W) is large, the noise component, $\delta_i(W)$ is also large thereby masking the communication latency part.

4 Benchmark Results on SanDiego Supercomputing Center (SDSC)

This section presents our observations from the data collected on the SanDiego Supercomputing Center's DataStar cluster [9]. The DataStar compute resource at SDSC [9] offers 15.6 TFlops of compute power. For our experiments, we used two combination of nodes: (a) 32 8-way nodes with 1.5 GHz CPU, giving a total of 256 processors, and (b) 128 8-way nodes with a mix of 1.5 GHz and 1.7 GHz processors, giving a total of 1024 processors. The nodes are IBM Power-4 and Power-4+ processors. Each experiment is executed for about 1 hour in order to collect a large number of samples. However, in order to avoid storage of this large time series, we convert the data into discrete probability distributions with fixed number of bins. The distributions are used to compute different statistics related to each experiment.

4.1 Benchmark Results on SDSC (256 processors on 32 nodes)

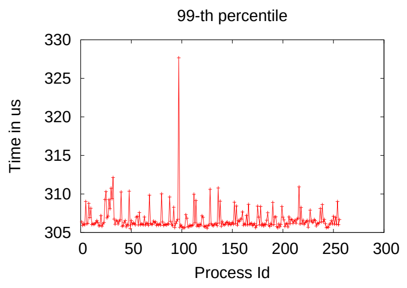


Fig. 7. This plot shows the values corresponding to 99-th percentile in the distribution of $w_i + \delta_i$ for $w = 300\mu s$.

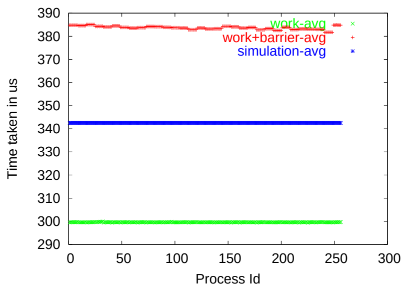


Fig. 8. The plot shows the mean for three distributions for each process: $w_i + \delta_i$, which is the work-avg, $t_i^e - t_i^s$, which is the total time for iterations including barrier wait, and $\max_{i\in[1\ldots N]}(w_i + \delta_i)$, for a quanta of $300\mu s$.

The PB was run on 256 processors spawning 32 nodes. All nodes and processors in this experiment were identical.

The value corresponding to 99-th percentile in the distribution of $w_i + \delta_i$ with $w = 300\mu s$ is plotted in Figure 7. Next, we plotted the distribution of $t_i^e - t_i^s$ for all the processes in Figure 9, along with the (predicted) distribution of $\max_{i\in[1\ldots N]}(w_i + \delta_i)$ obtained using the simulator. It is seen that the simulation predicts the real distribution very closely on this production cluster, except in the tail part of the distribution.

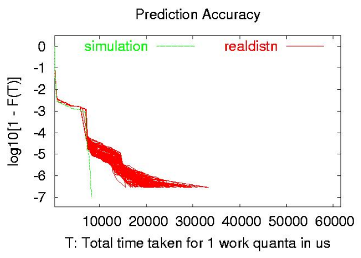
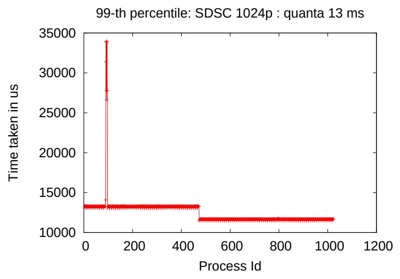


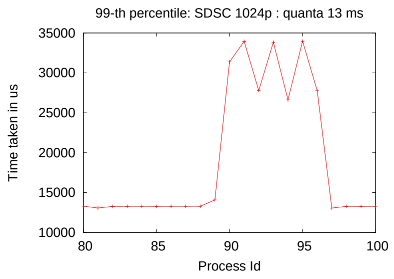
Fig. 9. Comparing distribution of $\max_{i\in[1...N]}(w_i + \delta_i)$ against distributions of $t_i^e - t_i^s$ for 256 processes, for $w_i = 300\mu s$.

Further insight is revealed in Figure 8, which shows the average time for the distributions of $(w_i + \delta_i)$, $t_i^e - t_i^s$, and $\max_{i\in[1...N]}(w_i + \delta_i)$. It shows that the mean value of $\max_{i\in[1...N]}(w_i + \delta_i)$ is about $50\mu s$ less than the mean of $t_i^e - t_i^s$ distributions. This is accounted by the communication latency, $2 * \tau * log(N + 1)/2$, which is calculated to be $2 * 5\mu s * log(32)/2 = 40\mu s$ for the 32 node cluster in DataStar.

4.2 Benchmark Results on SDSC (1024 processors on 128 nodes)

We repeated the experiments on a larger cluster of 128 nodes with 1024 processors. However, in this experiment there were 2 different sets of processor types.





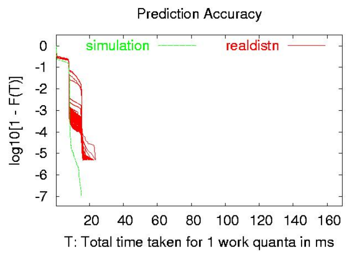
(a) *The graph shows the 99-th percentiles from the distributions of $(w_i + \delta_i)$ for 1024 processors. There is a noticeable spike indicating that some processors take significantly longer to complete the computation phase.*

(b) *Zooming into the spiked area of Figure 10(a) shows that there are 8 processes on 1 particular node that is taking up significantly longer to finish the work.*

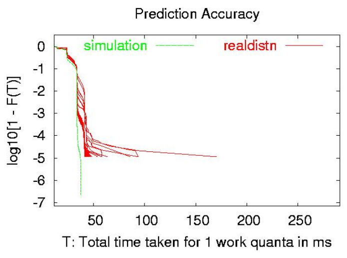
Fig. 10.

In Figure 10(a), we have plotted the 99-th percentile of $(w_i + \delta_i)$ distribution for each processor. It shows a spike around processor id 100. A zoom-in of the region between processor id 75 and 100 is shown in Figure 10(b). There are a set of 8 processors starting from id 89 to 96 which takes significantly longer to complete its workload. All these processors belong to a single node. This indicates that one node is anomalous and slowing down rest of the processes in this cluster. We discussed this with the SDSC system administrator who independently discovered problems with the same node (possibly after receiving user complaint). Our run on the 256 processor system had the same problem (see Figure 7) due to the same node.

Finally, in Figures 11(a) and Figures 11(b) the prediction made by the simulator is compared against the observed distribution. In this experiment, the match between the predicted distribution of $\max_{i\in[1...N]}(w_i + \delta_i)$ and the observed distribution is not as good as in the previous experiment (for both the values of $w = 300\mu s$ and $w = 13ms$). For the $300\mu s$ case, the mean of the $\max_{i\in[1...N]}(w_i + \delta_i)$ was found to be $1.93ms$, while the mean of the $t_i^e - t_i^s$ was in the range $2.54ms$ to $2.93ms$ (for different processes i); while, for the $13ms$ case, the mean for $\max_{i\in[1...N]}(w_i + \delta_i)$ distribution was $23.191ms$ and the mean of the $t_i^e - t_i^s$ ranged from $24ms$ to $28.36ms$. At present, we are unable to explain this anomaly. We are conducting more experiments on different systems to pinpoint the cause of this.



(a)



(b)

Fig. 11. *Comparing distribution of* $\max_{i\in[1...N]}(w_i+\delta_i)$ *against distributions of* $(t_i^e - t_i^s)$ *for 1024 processes for* $w_i = 300\mu s$ *and* $w_i = 13ms$ *on SDSC cluster.*

5 Conclusion

High performance computing systems are often faced with the problem performance variability and lower sustained performance compared to the optimal. It has been noticed that system activities, like periodic daemons and interrupts, behave as noise for the applications running on the large clusters and slows down the performance. If a single thread of a parallel application is slowed down by the Operating System interference, the application slows down. Hence it is important to understand the behavior of noise in large clusters in order to devise techniques to alleviate them. A theoretical analysis of the impact of noise on cluster performance was carried out by Agarwal et al. [1]. A model for the behavior of noise was designed to predict the performance of collective operations in cluster systems. In this paper, we have attempted to validate the model using empirical data from a production cluster at SanDiego Supercomputing Center. We have designed a benchmark for collecting performance statistics from clusters. Besides providing the means to validate the model, the measurements from the benchmark proved useful in identifying system anomalies, as shown in the the case of the SDSC cluster.

6 Acknowledgment

Firstly, we would like to thank SanDeigo Supercomputing Center (SDSC) who provided us substantial time on their busy system for our experiments. We would like to thank Marcus Wagner for helping us in collecting the data from the SanDiego Supercomputing Center. Thanks to Rama Govindaraju and Bill Tuel for providing us with insights on the testbed cluster we have used for fine-tuning the parallel benchmark and helping us in collecting the data.

References

1. S. Agarwal, R. Garg, and N. K. Vishnoi, "The Impact of Noise on the Scaling of Collectives," in *High Performance Computing (HiPC)*, 2005.
2. T. Jones, L. Brenner, and J. Fier, "Impacts of Operating Systems on the Scalability of Parallel Applications," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-MI-202629, Mar 2003.
3. R. Giosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of System Overhead on Parallel Computers," in *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2004.
4. F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q," in *ACM Supercomputing*, 2003.
5. D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-grained Parallel Applications," in *ICS*, 2005.
6. J. Moreira, H. Franke, W. Chan, L. Fong, M. Jette, and A. Yoo, "A Gang-Scheduling System for ASCI Blue-Pacific," in *International Conference on High performance Computing and Networking*, 1999.
7. A. Hori and H. Tezuka and Y. Ishikawa, "Highly Efficient Gang Scheduling Implementations," in *ACM/IEEE Conference on Supercomputing*, 1998.
8. E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll, "STORM: Lightning-Fast Resource Management," in *ACM/IEEE Conference on Supercomputing*, 2002.
9. "DataStar Compute Resource at SDSC." [Online]. Available: http://www.sdsc.edu/user_services/datastar/