Measuring Performance Impacts of SMM on Parallel MPI Benchmarks

Thomas Van Klaveren, Veronica Sneha, Sai Krishna Yerneni, Andrew Rogers

June 7, 2015

1 Abstract

System Management Mode (SMM) is a processor mode that, while kept secure and not accessible by the operating system, has functionality to handle safety functions, hardware emulation, and power management. Studies have been formed and conducted to look at the effects of this special mode on the operating system and hypervisor. Additionally, effects on applications in general have been tested; however, more work is desired to understand the effects on specific application types. In particular, the effects on Message Passing programs is of interest because of the growing popularity in parallel programming. We show that, on different lengths of interrupts into SMM, varying effects are noted. Further, we see that the major impact is seen with longer interrupts as the processes-to-node ratio increases.

2 Introduction

System Management Mode (SMM) is a processor mode that is obscured from the operating system(OS). Moreover, the OS is unable to access this mode. SMM is accessed through the System Management Interrupt(SMI). Execution of the OS is ceased during an interrupt and the special SMM gains control to handle power management, hardware control, and OEM designed code[3]

Past studies have shown that SMI and entrance into the special mode have impacts on the hypervisor and the kernel[1]. Upon the occurrence of SMI, the SMM gains control of all CPU processes. Among the tests that were conducted by Delgado and Karavanic was a study of the timer interrupt, which of significance in the tests performed in this paper. They found that on short SMI, when control returned from SMM, the timer interrupt that would have taken place during that SMI is pushed to the return time causing a longer time frame than would normally have occurred between timer interrupts.

Subsequently, the following timer interrupt occurs at its regularly scheduled CPU cycle time regardless of the SMI; this truncates the size of the next scheduled interval. Moreover, their study proved that long SMI occurrences effectively froze active tasks because some timer interrupts were missed completely.

The popularity of parallel programming is growing as a means of computing bigger problems more efficiently. Message Passing Interface parallelism, in particular, is a method of parallel computing that utilizes multiple and selectable nodes in a cluster to compute the problem at hand. The purpose of our work in this paper is to determine the effects of SMM on Message Passing Interface(MPI) programming. Further, the effects of oversubscribing is observed by overloading processes on the nodes specified for the application. We utilize the findings from Delgado and Karavanic and advance to determine and explain the findings that we see when applying regularly scheduled SMIs when the MPI program is running.

3 Related work

Delgado, et al. measured the impact of SMIs on a number of various modules within the computing environment[1]. They demonstrated that the CPU cycles lost to SMIs affected all core and that degradations of the performance benchmarks was nearly proportional to the duration of the SMI. While the longer SMIs provided more dramatic impacts, short SMIs added system jitter due to timer interrupts[1]. This was observed in Figure 1.

Power management also demonstrated impacts related to SMIs. The CPU tries to reduce power usage by switching from the fully active C0 state into lower power states when it is able to idle or when less CPU performance is required. The tickless kernel in Linux reduces power by removing an unnecessary clock interrupt and allowing the CPU to remain in a less active state. Frequent SMIs wake up an idle CPU and undo the benefits of this power-saving strategy[1].

User applications and devices also experienced impacts related to the increase of SMIs. Unreal Tournament was used to examine a visual impact of the CPU cycles lost to SMIs. Unreal Tournament is a video game which utilizes a 3d modeling engine to render a world in which characters move around in real time. Video frames displayed per second (FPS) is the metric by performance is measured[1]. 30 FPS is the threshold value wherein any instances below that rate generate an experience where humans can visibly detect a loss in performance[1]. The tests performed demonstrated that the size of the SMI was directly related to the drop in frame rate[1]. USB driver performance was analyzed using a USB audio device. They were able to capture the instances lost by driver empirically and they were additionally able to physically detect audio distortion once the SMIs exceeded 20ms[1]. Performance of TCP impacts is closely related to our work with Message Passing Interface. They found, matching their other tests, that SMI size negatively impacted the throughput.

4 System Management Mode

Intel first introduced System Management Mode with the Intel® 386 SL processor[1]. The designers intended this to be used for firmware as it operates transparent to the OS and applications. The processing only enters into SMM through the use of a System Management Interrupt. The SMI moves processing to a separate operating environment with a new address space (SMRAM)[3]. The data and code for the SMI handler reside in this protected space. These SMI's were designed utilize SMM to support system-wide functions like power management or OEM-designed proprietary code[3].

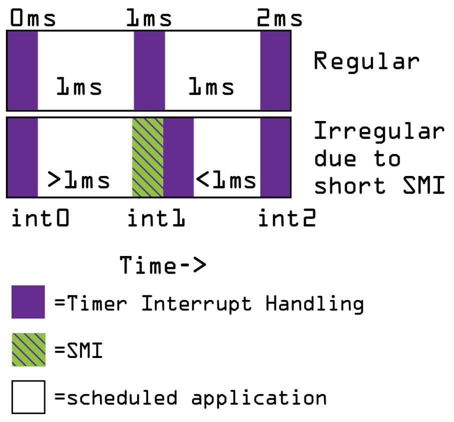

Figure 1: SMI Effect on CPU Cycles

The SMI handler code can reduce the system performance while supporting other hardware needs. The code can idle hard-disk drives, turn off monitors, run OEM firmware code or even place the whole system in a suspended state[3]. The SMIs operate at a higher priority level than Non-Masking Interrupts(NMIs) and device interrupts[1]. The higher priority level dictates that these SMIs cannot themselves be interrupted by NMIs or other interrupts. These lower priority interrupt that were paused by the trap to SMM can only resume and complete processing once the SMI handler has finished and returned operation back to the normal state[1]

5 MPI

MPI was being defined in 1993 ”for distributed memory concurrent computers and networks of workstations”[4]. The MPI-1 standard was published in 1994 and the MPI-2 standard was later published in 1996[6] and MPI 3.0 was approved in 2012[5]. MPI-2 and MPI 3.0 added additional extensions onto the core of MPI-1[6, 5]. MPI was designed to provide various communication routines to support different use cases. It did not initially contain fault tolerance and it was not specifically designed for parallelizing compilers[4]. MPI has point-to-point and collective communication methods, however, that form a strong foundation for a parallel processing algorithms.

5.1 MPI Communication Methods

Process groups in MPI allow the introduction of task parallelism. These groups are an ordered collection of processes and each process uses its rank within the order for identification[4]. Tasks can be loaded with executable code and data; different executable code would provide Multiple Instruction, Multiple Data(MIMD) task parallelism and Single Program, Multiple Data(SPMD) task parallelism can be generated by tasking groups different conditional branches the same executable code[4].

An MPI send operation can be created in one of three different modes. *Standard* mode allows the function to return without waiting for an acknowledgment from the receiver[4]. This prevents the sender from stalling when the extra synchronization is not required. *Ready* mode messages specify that the destination process needs to have previously posted a *receive* or the outcome is indeterminate[4]. A send using *synchronous* mode does not return until a matching *receive* is returned. This does explicitly mean that *synchronous* mode sends lock up the process. The send operations can either be blocking or non-blocking[4]. Blocking sends do not return until the memory is safe to reuse without corrupting the message and non-blocking operations return a handle to a communication object that can be used to check for completion of the send[4].

5.2 OpenMPI

OpenMPI was organized in 2004 to provide an open source implementation of MPI-1 and MPI-2 standards. Their intention and goals are to include the HPC community in the growth and development of a "production-quality complete MPI-2 implementation" with [6]. They represent a merger of work done with: FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, and LAM/MPI from Indiana University and this represent the next iteration of all of these efforts.

6 Design of the Experiment

Design and testing of the SMI impacts on parallelism through MPI were conducted in High Performance Computing lab at Portland State University, in Portland, Oregon. It was our intention to create a testing benchmark that would highlight delays in the execution of MPI message transfers in the presence of controlled SMIs running in the background. One challenge that we faced relative to traditional benchmarks is that we were highly focused on the performance degradation when SMIs occur as message passing was taking place and not simply the total completion time. A good MPI benchmark needs to account for the overlap of the computation and communication[2]. Time loss to SMI during computation phases has been measured[1] and it was our intent to eliminate as much of that impact as possible from our tests.

6.1 Black Box SMI

Delgado created a device driver that could generate SMIs on command; it writes specific values to an IO port that normally would be assigned to generate SMI's[1]. This is a valuable tool for the purpose of measuring the performance impacts on the operating system and applications. We are able to produce SMIs of two different durations with Delgado's driver. It is able to produce an interrupt of a consistent $5ms$ ("short SMIs") or $104ms$ ("long SMIs") on a steady one per second frequency[1]. There is also functionality in the driver to count the number of interrupts that have occurred since the interrupt was activated[1]. This level of control gives us the ability to effectively isolate the SMIs and measure the impact on performance in a controlled environment.

6.2 Hardware Setup

The Portland State University HPC is made up of the Wyeast cluster. We accessed the head node of the Wyeast cluster through SSH from the Computer Science department's Linux server, Ada. The head node, *wyeasthead* is a Dell PowerEdge T410 unit that is accessible through the PDX.edu intranet. The head is connected via a Cisco SLM2024 router and Cat 5e interconnects to 18 child nodes (wyeast01 - wyeast18) running on Dell PowerEdge R410 units. The units have Fedora 21 and OpenMPI version 1.8.3 installed.

6.3 Tests Performed

The testing algorithm we used was designed to identify variations in the performance of the MPI send/receive calls within the application. The algorithm measures the time between when the MPI_Ssend() function is called and returned as a benchmark for SMI impact to the MPI activity. The function returns when the receiving process sends an acknowledgment. We felt that a call to wall time made both before and after the function would be the most effective method to isolate delays in the communication protocol from any computation done by the process. This measurement required us to use a synchronous blocking communication method to ensure that the transmission was complete prior to collecting the second time stamp.

Messages of size 10^5 elements of type integer, *Message1*, were sent from the head node to p processes that were equally divided over n nodes. If 16 processes were allocated to four nodes, then four communication pipelines would be established from the head to each node. Total processing time would also be measured to be compared to the message transfer time. Each node then sends a message of a single element of type double integer, *Message2*, back to the head node.

Tests were conducted by measuring the application numerous times under various conditions. The number of nodes varied from 2 to 4 due to restrictions to manage contention over the lab resources during the application of the testing period. We conducted 10 tests for each value of n and p where n = 2 or 4 and where p = 2^i for all i from 1 to 7 and p >= n. These rounds of 10 tests were each conducted three times. The first round was SMIs turned off. The second was with the short SMIs occuring once per second. The final round included the use of long SMIs occurring once per second.

We later ran further testing once we had dedicated access to 16 nodes. We decided to run a smaller set of tests on 8 and 16 nodes with the same range of processes due to time constraints. We collected various data points in each test. First there was total processing time of the program. Next, we collected the accumulated time sending the message from the head process to all other processes in the variable *M1*. Finally, we used *M2* to measure the accumulated time spent by the head process receiving messages from all other processes. This gives a view of both one-to-many and many-to-one message synchronization types.

7 Results and Discussion

Tests were run to specifically look at the timing effects of the SMIs on both message passing portions of the OpenMPI programs as well as the total runtime. We also looked at how adding additional nodes and processes impacted the results. In general, the effects were consistent across the board. We saw similar effects in each type of SMI (none, short, and long) over all levels of nodes and all levels of processes. We found that the impact of the SMIs on the message passing portion of the algorithm was identical to total processing time of the program.

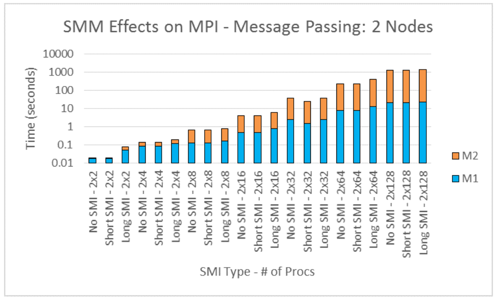

Figure 2: Message Time, 2 Nodes

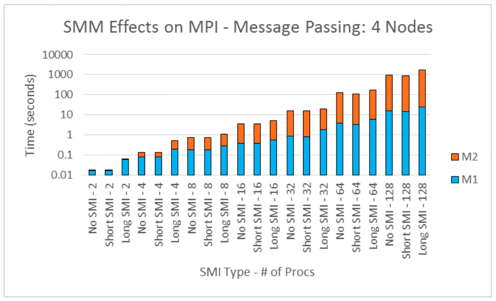

Figure 3: Message Time, 4 Nodes

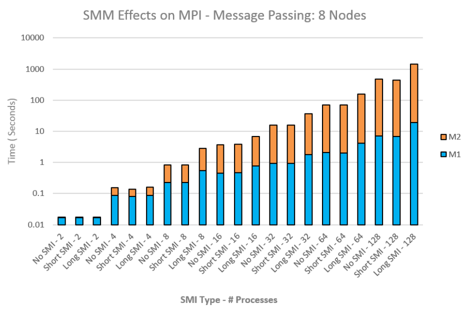

Figure 4: Message Time, 8 Nodes

7.1 Processes

Before discussing the SMI effects, it is important to understand the impact of adding additional nodes as well as additional processes. When the number of nodes is less than or equal to the number of processes, the time spent on passing *Message1* and *Message2* is very similar. As discussed above, *Message1* is one-to-many and *Message2* is many-to-one. The reason for the similarity is that they are essentially the same process. For *Message1*, the head node sends some message in a loop to each of the other processes, each of which is on their own node. Since each process is on its own node, it is not sharing time with any other process and the total time spent sending the message is the time it takes for the head node to finish its loop. Similarly, when each of the other processes sends a message back to the head node, because each process is running on its own node, the total time spent in this

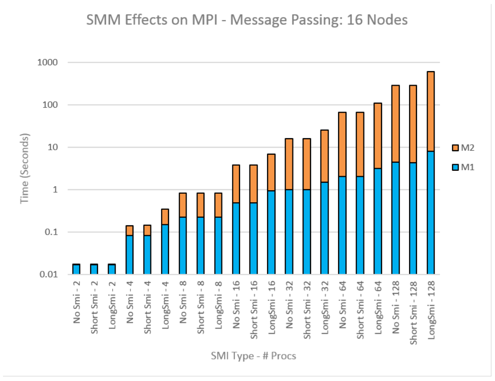

Figure 5: Message Time, 16 Nodes

message passing call is just the time it takes for the head process to finish receiving each message in order. The only time savings is the fact that *Message2* is sending a single integer whereas *Message1* is an array of 100,000 elements.

Once the number of processes exceeds the number of nodes, *Message2* is impacted. When each process is sending its message to the head, the head process may or may not be an active process to receive the message and the sender will need to wait for the head to receive a context switch and collect the message. Assume, for example, the head process and 3 other processes are on the same node. In turn on a context switch, each process sends its message to the head process. Each senders message timer is running and does not stop until the message is received, which will only be done on the head process's turn. Because each process has a turn, the sender's timer runs for longer than it would otherwise. The result of overloading the nodes was an exponential time increase not only in the total runtime, but also in the time spent in message passing calls.

7.2 SMI

It was hypothesized that both the short and long SMIs would create a time delay both for the runtime of the program and for the time spent in the message passing calls. The reasoning behind this theory comes from the tests outlined by Delgado and Karavanic.

In their research, they noted that successive timer interrupts, which occurred with a constant delta of 1ms, and a change in that delta after the introduction of the SMI[1]. They further explained that when an SMI occurs when a timer interrupt was expected, the delta was increased. The delay in timer interrupt handling results in the greater... delta, that in turn results in the next timer interrupt occurring in a shorter time than the normal delta[1]. Our hypothesis was partially confirmed after viewing the results of the timing output for each SMI type compared to that of No SMI, however a secondary effect of the shorter delta was initially overlooked.

As it turns out, there is relatively little effect of the short SMI. In fact, we consistently, though not exclusively, saw a minor time decrease of less than 1% for processes up to 64 on 2 and 4 nodes and a slightly greater decrease for more than 64 processes as evidenced in 6. These short SMI results were quite surprising, though explainable. When the SMI is called, all processor cores enter the System Management Mode[1]. That means that the SMM is using a processs CPU cycles and explains why one timer interrupt delta will be greater than expected and the following delta will be shorter. When a process in the application is blocked and cannot move forward in its message passing call, the small takeover of SMM eats the CPU cycles in the blocked processs context turn. This can be seen in 1. More importantly, when control returns to the executing program from an SMI, the next processs turns CPU cycles may have already started and been eaten by the SMM meaning that its blocked time is preemptively cut short due to the SMI. When the receiver gets the message, the wall clock time is shorter than it would have been if no SMI had taken over the CPU because its timer starts later and switches to a new process at its regularly scheduled time causing a smaller than expected delta. In this small interrupt duration, we actually see a time savings, though very small, due to the hijacking of the send processs waiting CPU cycles by SMM before the timer is called. The effects on the message passing calls translate to the program runtime as a similarly small time savings is identified and shown in Figures 10,11,12, and 13. Based on the line graph, the effect of short SMI is minimal regardless of the process-to-node ratio.

We observed that while the results for test with short SMIs were generally lower than tests with no SMIs, we did see some individual runs where short SMI times exceeded the result of the no SMI run. This appears to be in line with the "jitter" that was observed by Delgado and Karavanic in regard to the performance of timer interrupts[1]. We were unable to identify the cause of this variance in our testing and believe addition work in this area is warranted.

On the other hand, the thought that the long SMI would negatively impact the message passing calls and the program runtime was confirmed. Long SMI can have major delay consequences as the nodes are more overloaded with process. As processes are added exceeding the nodes available, the delay becomes exponentially worse. In this case, the SMI is so long that the timer interrupt following the delayed timer interrupt is not decreased, but actually missed. In essence tasks are completely frozen for extended periods of time [1] as the timer interrupt is missed. This hurts performance because, not only is the message passing timer continuing to count if the SMI is called during that process, but the following processes is missed until its next turn. The big effect in the worst case scenario is when the receiver is skipped, particularly when the nodes are overloaded with processes. When the receiver is missed, the timers for all of the other processes keep running for another round of context switches. Again, these problems in the message passing also translate to the program execution as a whole as evidenced in the graph displaying the total runtime in Figures 6, 7, 8, and 9. In addition, note the growing gap between the No/Short SMI lines and the Long SMI line in Figures 10,11,12, and 13 showing the heightened impact of Long SMI on hyperthreading MPI.

8 Future Work

We would like to expand our measurements in further testing. The convention of our benchmark only allowed us to perform basic send/receive operations and we would like to explore further communication topologies. We would like to conduct examination

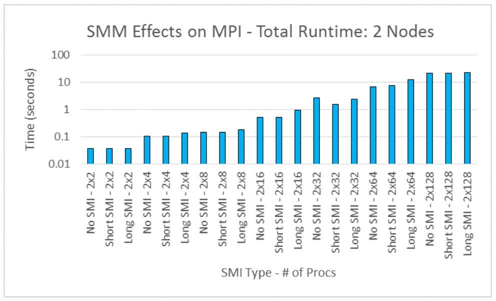

Figure 6: Total Run Time, 2 Nodes

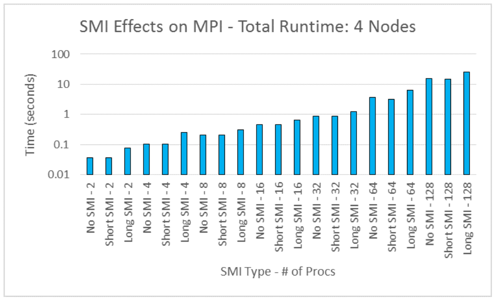

Figure 7: Total Run Time, 4 Nodes

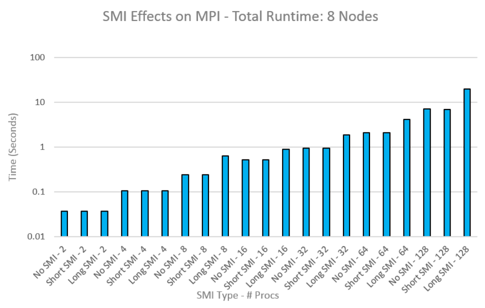

Figure 8: Total Run Time, 8 Nodes

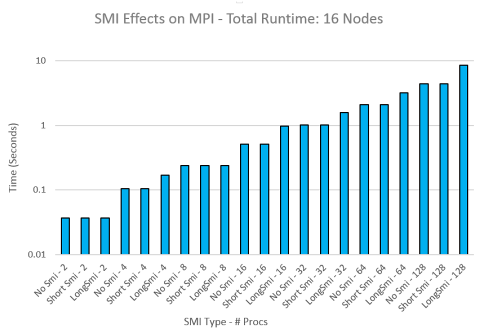

Figure 9: Total Run Time, 16 Nodes

into more of the OpenMPI message passing calls such as Scatter, Gather, Broadcast, and Reduce. Due to the timer that was used, MPI_Wtime(), and how this call worked, we were limited to using synchronous sends so that start and end time could be collected in the same process and so that the base time was guaranteed to be the same in the start and end call to this routine. Perhaps using a different timer method would allow for a test of non-synchronous sends. The effects specific to message passing may be different in these cases and it would be interesting to do further testing. An area that was not specifically timed and tested was the computation portion of the processes. While our main focus was on the message passing calls, a comparison of the effects on computation code would help to give further understanding into SMI effects on the application. One final piece of testing that would advance our study would be to look at scaling the length of the SMIs to find where the tipping point is where the application begins to see a negative impact.

9 Conclusions

The performance implications of System Management Mode and System Management Interrupts on MPI programs were analyzed in depth in our study. The effects of these interrupts were exacerbated by overloading the cluster nodes with additional processes. Findings indicate an impact of the SMIs in this case. We observed minimal decrease in processing times with the introduction of regularly scheduled short SMIs in the message passing calls and total processing time of the application. In contrast, the long regularly scheduled SMI negatively impacted the performance of the application. The impact became more severe as the number of processes assigned to the nodes increased, showing a strong correlation be-

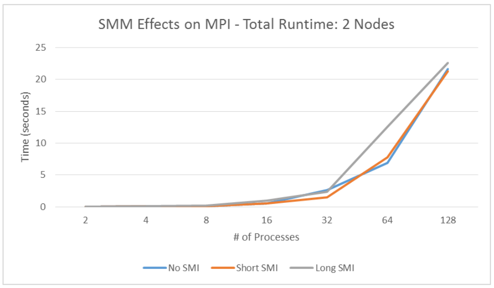

Figure 10: Different SMI Intervals, 2 Nodes

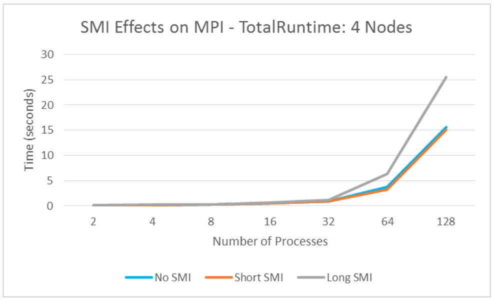

Figure 11: Different SMI Intervals, 4 Nodes

tween the two.

10 Acknowledgments

We would like to thank Forrest Alexander for his technical assistance with the High Performance Lab servers and in configuring the OpenMPI software onto the test system as well with guidance in the design of the experiment. Brian Delgado also provided assistance with the use of his black box SMI generating software.

References

[1] Delgado, B. and Karavanic, K. Performance Implications of System Management Mode. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization.* Sept. 22-24, Portland, OR USA

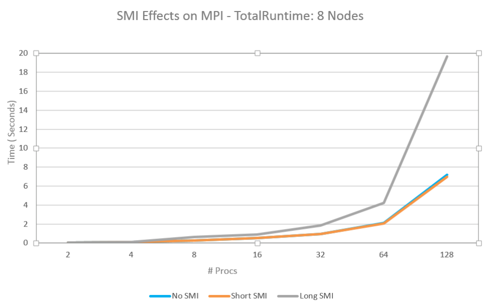

Figure 12: Different SMI Intervals, 8 Nodes

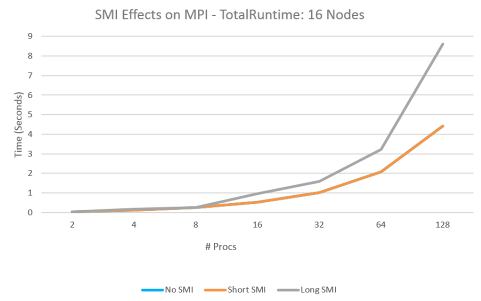

Figure 13: Different SMI Intervals, 16 Nodes

[2] Gropp, W. and Lusk, E. Reproducible Measurements of MPI Performance Characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* September 26, 1999. London, UK pages 11-18

[3] Intel, Intel®64 and IA-32 Architectures Software Developer's Manual (Vol. 3)

[4] The MPI Forum. MPI: A Message Passing Interface. 1993. ACM

[5] The MPI Forum. http://www.mpi-forum.org

[6] "Open MPI V1.8.4 Documentation." *Open MPI V1.8.4 Documentation.* The Open MPI Project, 20 Dec. 2014. Web. 5 May 2015.

[7] Li, J. Message Passing Interface(MPI). *CS515 Parallel Programming*. Portland State University, May 2015, Lecture

[8] Wikimedia Foundation. http://en.wikipedia.org/System_Management_Mode