Asymmetric Interactions in Symmetric Multi-core Systems: Analysis, Enhancements and Evaluation

T. Scogland
Dept. of Computer Science
Virginia Tech
njustn@cs.vt.edu

P. Balaji†
Math. and Computer Science
Argonne National Lab
balaji@mcs.anl.gov

W. Feng, G. Narayanaswamy
Dept. of Computer Science
Virginia Tech
{feng, cnganesh}@cs.vt.edu

***Abstract*—Multi-core architectures have spurred the recent rapid growth in high-end computing systems. While the vast majority of such multi-core processors contain symmetric hardware components, their interaction with systems software, in particular the communication stack, results in a remarkable amount of asymmetry in the *effective capability* of the different cores. In this paper, we analyze such interactions and propose a novel management library called *SyMMer* (Systems Mapping Manager) that monitors these interactions and dynamically manages the mapping of processes on processor cores to transparently improve application performance. Together with a detailed description of the *SyMMer* library, we also present performance evaluation comparing SyMMer to a vanilla communication library using various micro-benchmarks as well as popular applications and scientific libraries. Experimental results demonstrate *more than a two-fold improvement* in communication time and 10-15% improvement in overall application performance.**

I. Introduction

Multi- and many-core architectures [1]–[5] have been one of the primary driving factors in the recent rapid growth of HEC systems. The great majority of these architectures are symmetric (i.e., homogeneous) in that the cores have equal and interchangeable computational parameters. Consequently, many application and systems software developers assume that such symmetry is a direct indication of the *effective capability* of each core in the system. Unfortunately, as we have shown in our previous work [6], this is often untrue.

The reason for this, is that while the processor hardware itself is symmetric, the rest of the system is not. For example, most network hardware has *not* been designed to maintain state concerning which application process is running on which core of the system; thus hardware interrupts corresponding to data packets for different processes are either randomly distributed across all cores or are statically directed to a single core. This additional interrupt processing can cause one core to be more overloaded than others, thereby affecting performance. Further, on processors where each core has its own dedicated L1 and/or L2 cache, the core handling the interrupt, fetches the data being communicated into its local cache for processing it. Thus this data will not be *equidistant* from all cores. Consequently, further processing of the data is faster on some cores, than on other cores. Finally, as we will show in the later sections, assumptions made within a systems software stack (such as the MPI library) with respect to the symmetry of the different cores can propagate as skew for end applications, resulting in unintended *out-of-sync* communication patterns. In these and other examples, there can be a marked asymmetry in the *effective capability* of the different cores with respect to what each core can offer to the overall parallel application.

So, how can such asymmetry be dealt with? There are at least two different options to deal with this: (1) we can design a programming model specific to multi-core systems that exposes the asymmetry in the different cores to the application or systems software developer, allowing them to explicitly manage the workload of different processes; (2) we can provide an automated approach, either through a library or an external system daemon, which monitors the processes and dynamically maps different processes to different cores to improve performance. Unfortunately, neither option is perfect. For the first option, even if we build a new programming model, having application writers port their applications is cumbersome and impractical. Also, the exact amount of asymmetry depends on various factors specific to the processor hardware, and is difficult to abstract through a programming interface. For the second option, neither a library nor a daemon has explicit information from the application. Thus, any decision made will be based on *monitoring* the interactions and *inferring* the application's computation and communication patterns.

In this paper, we take the second approach to handling asymmetry in multi-core environments by designing a

This research is funded in part by the Mathematical, Information, and Computational Sciences Division of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and the Department of Computer Science in the College of Engineering at Virginia Tech.

†This author's work was also supported in part by the National Science Foundation Grant #0702182.

novel Systems Mapping Manager (SyMMer) library. Specifically, we first perform a detailed analysis of the asymmetric interactions between multi-core architectures and the communication stack. Next, we utilize this analysis to efficiently monitor these interactions and dynamically manage the mapping of processes onto processor cores so as to improve performance, while remaining completely transparent to the programmer and end user. To achieve this, SyMMer uses several heuristics to identify the types of asymmetry between the effective capabilities of cores and the processing requirements of the application. If substantial asymmetric behavior is detected, SyMMer transparently re-maps the processes, thereby achieving improved performance. While SyMMer is a generic framework that can fit into any communication library, in this paper our descriptions refer to a version plugged into the MPICH2 implementation [7] of the Message Passing Interface (MPI) [8].

Together with the detailed design of the SyMMer library, we also present performance evaluations comparing SyMMer to vanilla MPICH2 using various micro-benchmarks as well as the popular scientific libraries and applications, GROMACS [9], LAMMPS [10] and the FFTW [11] library. Our experimental results demonstrate that our approach can provide *more than a two-fold improvement* in communication time and 10-15% improvement in overall application performance.

II. Overview of Multi-core Architectures

For many years, hardware manufacturers have been replicating structures on processors to create multiple pathways allowing multiple instructions to run concurrently. Duplicate arithmetic and floating point units, co-processing units, and multiple thread contexts (SMT) on the same processing die are examples of such replication. Multi-core processors are the next step in such hardware replication where two or more independent execution units are combined on the same integrated circuit.

At a high level, multi-core architectures are similar to multi-processor architectures. The operating system handles multiple cores in the same way as multiple processors: by allocating one process at a time to each core. Arbitration of shared resources between the cores happens completely in hardware, with no intervention from the OS. However, multi-core processors are also very different from multi-processor systems. For example, in multi-core processors, both computation units are integrated on the same die. Thus, communication between these computation units does not have to go outside the die, and hence is independent of the die pin overhead, making intra-die communication much faster than inter-die. Further, architectures such as the current

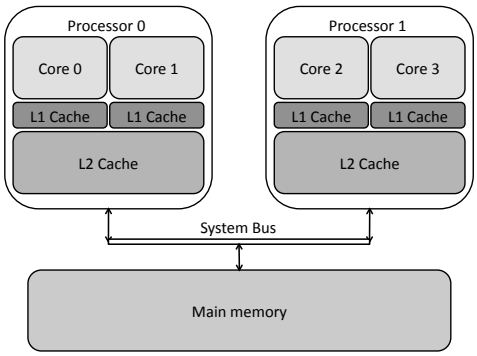

Fig. 1. Intel Dual-core Dual-processor System

Intel multi-core processors, as shown in Figure 1, provide a shared cache between the cores on the same die. This makes communication even simpler by eliminating the need for explicit communication protocols; the data is simply there in the local cache.

However, multi-core processors also have the disadvantage of more shared resources as compared to multi-processor systems. That is, multi-core processors might require different cores on a processor die to block waiting for a locally shared resource when the resource is being used by a different core. Such contention increases as the ratio of cores to other resources (e.g., memory controllers or shared caches) increases.

III. Asymmetric Interactions in Multi-core Systems

Given the rise of multi-core architectures, its interactions with applications and systems software is important. This section describes such interactions.

A. Application and Developer Viewpoint of the Communication

The communication stack described in this section consists of the MPICH2 implementation of MPI, using Linux TCP/IP sockets, though the behavior is similar to other implementations of MPI as well.

On the transmission side, the application formulates a message and hands it over to MPI to transmit. MPI buffers this data, appends a header, and passes it to the TCP stack for transmission. On the receiver side, the network adapter transfers received packets to the socket buffer, where they are assembled, checked for validity, and held until requested by the application. When the application calls an MPI receive operation, this data is copied out of the socket buffer into the application designated receive buffer.

B. Architectural Viewpoint of the Communication Stack

While the path the message follows is straightforward when viewed from an application or developer standpoint, there are many hidden side effects below the clean abstraction provided by the higher layers of the communication stack. In this section, we present the impact these effects can have on a cores effective capability to perform computation and communication.

1) Processing Impact: As described in Section III-A, when a packet arrives, the network adapter places the data in memory to be handled by TCP, after which an interrupt is raised to inform the communication stack that there is a message to process. For most system architectures, the processing core to which the interrupt is directed is either statically or randomly chosen using utilities such as *IRQ balance*. However, in both approaches, the *chosen core* to which the interrupt is assigned is not guaranteed to be the same core on which the process performing the relevant communication resides. Thus, all of the processing done on the packet at the receiver takes place on the chosen core including data integrity checks, connection demultiplexing, and other such compute intensive operations that can significantly impact the computational load on the *chosen core*.

Note that this protocol processing computational load is *in addition to* whatever computation the application process is performing. Thus, as far as the application processes are concerned, the *chosen core* tends to have a reduced *effective* computational capability as compared to the remaining cores.

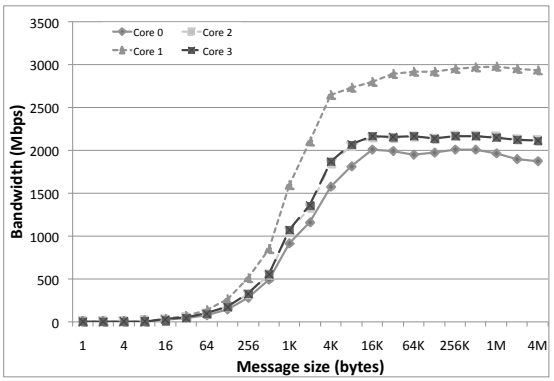

Fig. 2. MPI Bandwidth on Intel

2) Cache Transaction Impact: Aspects of protocol processing such as data copies and checksum-based data integrity require the communication stack to touch the data before handing it over to the application (through the socket buffer). For example, on the receiver side, the network adapter places the incoming data in memory and raises an interrupt to the device driver. However, when the TCP/IP stack performs a checksum of this data, it has to fetch the data into its local cache. Once the checksum is complete, when the application process has to read this data, it has to fetch this data to its local cache. That is, if the application process resides on the same die as the core performing the protocol processing, then the data is already on the die and can be quickly accessed. Instead, if the application process resides on a different die, then the data has to be fetched using a cache-to-cache transfer.

To demonstrate these effects (processing impact and cache transaction impact), we measured the communication bandwidth between two processes on two Intel dual-processor/dual-core machines, with the processes bound to different cores of the system. Figure 2 demonstrates these measurements. The interrupts (and hence the protocol processing) are always directed to core 0 in this case. As shown in the figure, when the application processes are also bound to core 0, they have to compete with the protocol processing that is occurring on the same core and hence suffer a performance penalty. On the other hand, when the application processes are bound to core 1, they do not have to face this additional protocol processing overhead. Further, since core 0 performs the protocol processing, the communication data is available on the local dies shared L2 cache (Figure 1), and hence, can be easily used by core 1. Thus, core 1 does not face any of the protocol processing overheads but has the benefits of in-cache access to its data data through a *free ride* from core 0. This results in the best possible performance for this situation. Cores 2 and 3 do not have to the face the protocol processing overheads, but do not have the benefits of increased cache hits either. Thus, their performance is in between that of cores 0 and 1. This behavior is confirmed in Figures 3 (a) and 3 (b), using the actual performance counters.

C. Identifying the Symptoms of Communication Stack and Multi-core Architecture Interaction

Directly understanding the actual interactions between the communication stack and the multi-core architecture is complicated, and requires detailed monitoring of various aspects of the kernel and hardware as well as correlation between the various events. Therefore, we take an indirect approach to understanding these interactions by monitoring for *symptoms* in the application behavior that are triggered by known interactions, instead of monitoring the interactions themselves. We should note that while a certain interaction can result in a symptom, the occurrence of the symptom does not necessarily mean that the interaction has taken place. That is, each symptom can have a number of *causes* that could have triggered it.

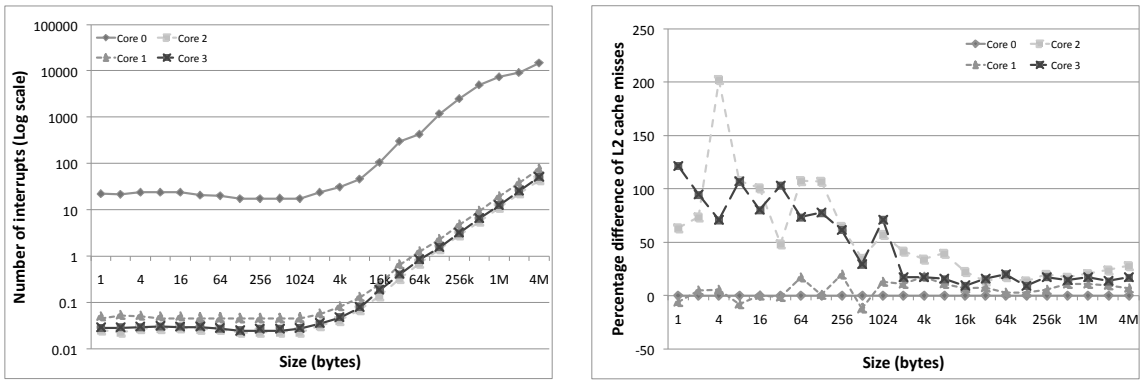

Fig. 3. (a) Interrupts Per Message and (b) Cache Analysis

In this section, we discuss the various symptoms that we need to monitor in order to infer that an interaction has taken place. In Section IV, we describe our approach to monitor these symptoms and minimize the impact of the interactions through appropriate metrics.

1) Symptom 1: Communication Idleness: As noted in Section III-B, if a core is busy performing protocol processing, the number of compute cycles it can allocate to the application process is lower than other cores, thus slowing down the process allocated to this core. Therefore, a remote process that is trying to communicate with this *slower* process would observe longer communication delays and be idle for longer periods as compared to other communicating pairs. This symptom is referred to as *communication idleness*. Again, as noted earlier, communication idleness can occur for a number of reasons, including native imbalance in the application's communication model.

2) Symptom 2: Out-of-Sync Communication: Communication middleware such as MPI performs internal buffering of data before communicating. Assuming both the sender and receiver have equal computational capabilities, there would not be any backlog of data at the sender, and the MPI internal buffering would not be utilized. Let us consider a case where process A sends data to process B and both processes compute for a long time. Then process B sends data to process A and again both processes compute for a long time. Now, suppose process B is *slower* than process A, and A wishes to send data to B. In this case, process A's MPI library would have to buffer the data since process B is not ready to receive more. From process A's perspective the send has completed once the data has been handed over to the MPI library; thus, it moves on to perform its computation.

After its computation, when it tries to receive data from process B, it sees that the previous data that it attempted to send is still buffered and tries to send it out again. By now, B is ready to receive more data and the send is successful. After receiving the data, process B goes off to perform its computation, while process A is waits to receive its data. This behavior is caused because despite the fact that processes A and B are performing similar tasks, they are slightly *out-of-sync* due to the difference between their effective computational capabilities.

3) Symptom 3: Cache Locality: As mentioned in Section III-B, when a core performs protocol processing, it fetches the data to its cache in order to perform the data integrity check, among other things. Thus, if the process which is waiting for that data is either on that core, or another on the same die, it will have the data without any further delay. On the other hand, if the process is on another die, it will require a costly inter-die data transfer to obtain it. Thus, a process which is transferring large amounts of data will gain a performance benefit from sharing a die with the core processing the communication interrupts. In addition to this, we find that processes which share data locally also present this symptom. For example, if one process is frequently sharing data with another process on the same machine, the data will be transferred between their caches on a regular basis, and can benefit from a shared L2 cache.

D. Our Focus

As shown above, the effective capabilities of logically identical cores are frequently not the same. The result being that where a process is currently running can greatly effect its performance. The appropriate response to this issue is to map the processes on to the correct cores. Traditionally the role of making sure that processes are on the best possible processing unit has fallen to the system scheduler, but current schedulers do not take the *effective capability* into account. Another option is to map the processes manually as we did in our previous work [6], but the time and difficulty of that method makes it impractical for many situations, such

as, projects where the code base changes frequently, or applications where the work done per process changes frequently. As such, an automatic or dynamic approach is necessary. In the rest of this paper we will describe and evaluate our solution, the dynamic mapping framework known as SyMMer.

IV. The SyMMer Library

This section describes our novel Systems Mapping Manager (SyMMer) library and its associated monitoring metrics. The SyMMer library, shown in Figure 4, is an interactive monitoring, information sharing, and analysis framework that can be tied into existing communication middleware such as MPI or OpenMP. The library directly implements the monitoring, communication, and analysis components, while the actual metrics that are used for the decision making are separately pluggable as we will describe in Section IV-A. This allows the design of metrics and the programming environment where they are used to be completely independent of each other.

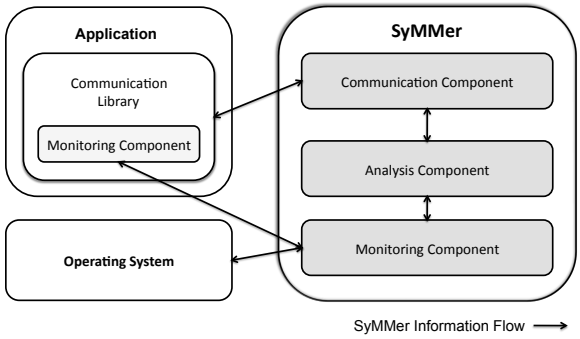

Fig. 4. The SyMMer Component Architecture

Interaction Monitoring: The interaction monitoring component is responsible for monitoring the system for information. This includes system specific information (hardware interrupts, software signals), communication middleware specific information (MPI data buffering time and other internal stack overheads) and processor performance counters (cache misses, interrupts). This component utilizes existing libraries and the operating system for as much of the instrumentation as possible, while relying on in-built functionality for the rest. For example, processor performance counters are measured using the Performance Application Programming Interface (PAPI [12]) and system specific information through the *proc* file-system. While MPI specific information can be monitored through libraries such as PERUSE [13], several of the current MPI implementations do not support these libraries yet. Thus, we use in-built profiling functionality to obtain such information.

In order to minimize monitoring overhead, the monitoring component dynamically enables only the components which are required for the metrics being used. For example, if no metric makes use of processor performance counters, such information is not monitored.

Communication: The communication component is responsible for the exchange of state or data between different processes in the application. Several forms of communication are supported, including point-to-point sharing models (for sending data to a specific process), collective sharing models (for sending data to a group of processes) and bulletin board models (for publishing events that can be asynchronously read by other processes). For each of these models, both intra-node communication (between cores on the same machine) and inter-node communication (between cores on different machines) are provided. Inter-node communication is designed to avoid out-of-band communication by making use of added fields in the communication middleware's existing headers. Whenever a packet is sent, the sender adds the information which needs to be shared to the outgoing header. The receiver, on receiving the header, shares this information with other local processes using regular out-of-band intra-node communication. This approach has the advantage that any single inter-node communication can share information about all processes on the node. Intra-node communication, on the other hand, has been designed and optimized using shared memory without requiring locks or communication blocking of any kind. This provides a great deal more flexibility and reduces the overhead of our framework significantly.

Analysis: The information collected by the monitoring component in each process and shared with other processes is *raw*, in the sense that there is no correlation between the different pieces of information. Further, the monitored information is low-level data which needs to be processed and summarized into higher level and more compact information before it can be processed by the various metrics. The *information analysis* component performs all the analysis and summarization of the data before passing it along. This component also allows the various pluggable metrics to be arbitrarily prioritized for cases where an application may show multiple symptoms. Finally, each monitoring event has a certain degree of inaccuracy or burstiness associated with it. Thus, some monitoring events have more *data noise* than others. To handle such issues, the analysis component allows different monitors to define *confidence levels* for their monitored data. Thus, depending on the number of events that are received, the analysis component can accept or discard different events based

on their confidence levels, using appropriate thresholds.

In addition to its duties in preparing data for the metrics to use, the analysis component takes action when a metric determines that it is required. Once the analyzed data identifies two processes which are currently scheduled on cores that are not best suited for them, but can potentially improve performance by swapping the processes between the cores, the analysis component is responsible for performing the swap as quickly and efficiently as possible. The communication component comes in to play in a handshake phase used to minimize the time the processes spend on the same core during a swap. For the purpose of this paper the actual mapping and swapping is accomplished using the get and set affinity functions available on Linux to set each process to have an affinity with only one core, and when swapping simply changing the affinity of each process to their new target core. The design does not mandate that this be the method used, and in fact would benefit from the availability of an interface which would allow one to get and set the current core without affinity being set.

A. Metrics for Mapping Decisions

In this section, we discuss different metrics that can be plugged into the SyMMer library. Specifically, we focus on metrics that solve the symptoms noted in Section III-C. Since our current reference implementation makes use of MPI, these metrics will be described in relation to its facilities, but they are equally applicable to other implementations and models. In all cases, the default mapping is assumed to be random with respect to the capabilities of the cores corresponding to the demands of the processes. In practice this is supported by the fact that while the scheduler generally puts processes on cores in order, the capabilities of each core per machine, the demands of the processes, and the order of launching the process is sufficiently variable for it to be considered random.

1) Communication Idleness Metric: This metric is defined based on the *communication idleness* symptom defined in Section III-C. The main idea of this metric is to calculate the ratio of the idle time (waiting for communication) and computation time of different processes. This metric utilizes the MPI monitoring capability of the SyMMer framework to determine this ratio. The idle time is measured as the time between the entry and exit of each blocking MPI call within MPI's progress engine. Similarly, the computation time is measured as the time between the exit and entry of each blocking MPI call. The computation time, thus represents the amount of computation done by the application process, assuming that the process does not block or wait on any other resource.

Hence, the computational idleness metric represents the idleness experienced by each process. For example, a process which has a high communication idleness can allow for other computations such as protocol processing. Processes with very little idle time on the other hand cannot tolerate even a small amount of protocol processing without seeing material performance degradation. Comparison of this idleness factor between different processes provides an idea of which processes are more suited for sharing the protocol processing overhead. Clearly, the idleness metric needs to be compared only for processes running on the same node. Hence this metric only uses the intra node communication channel discussed above.

Once the idleness is compared, if a process on a core other than the protocol processing core is experiencing a high idleness ratio, and the process on the protocol processing core is experiencing very low or no idleness, this metric determines that they are suitable for a swap. Figure 5 illustrates this process by depicting each state a process may be in, and the core or cores it might wish to switch to, from that state. Once the swap is made the idleness of the two processes should be more similar, both to each other as well as to the average idleness of all local processes. While this on the surface appears similar to the cache locality metric, there is a significant difference in that the locality metric tries to improve locality between local processes based on cache misses, whereas the idleness metric attempts to match the computation or communication demands of a process with a core whose capabilities will help balance the uneven load.

2) Out-of-sync Communication Metric: The out-of-sync metric captures the amount of computation performed by a process while having unsent data buffered in its internal MPI buffers, and compares that with the wait time of other processes. As described in Section III-C, this metric represents the case where unsent data followed by a long compute phase results in high wait times on the receive process. When the computation time (with buffered data) is above a threshold, a message is sent to all processes informing them of this symptom. Similarly, when the wait time of a process is above a threshold, this information is distributed as well. If communicating peer processes observe these conditions, a leader process is dynamically chosen, which then analyzes the data and decides on the best mapping. It then informs the new mapping to all processes concerned.

The mapping decision works to move each process to

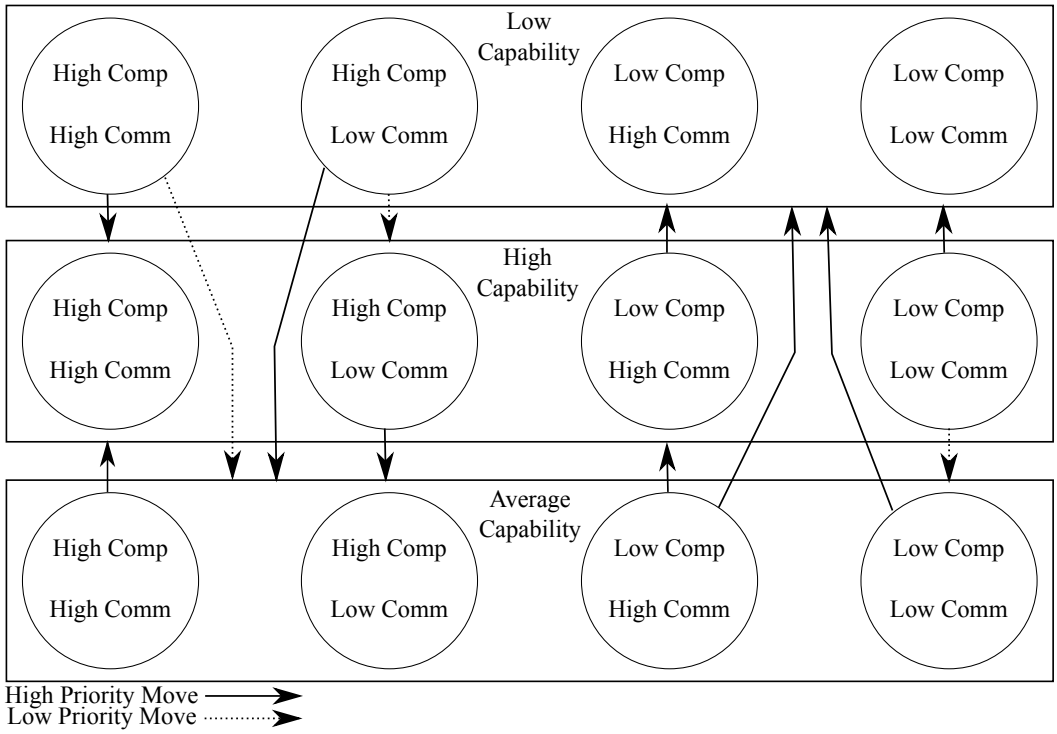

Fig. 5. Process states: Each circle is a possible process behavior, each box a level of core capability, and each line a potential destination to improve the performance of the process (based on the Idleness Metric)

a core that is similar to the one its peer is running on. In some cases a disparity between only two processes can cause all processes in an application to see out-of-sync communication, and fixing that one pair repairs this behavior for all the others as well. This metric is unique among the three discussed in this paper, in that it makes a completely distributed decision. In fact, it occurs between a minimum of two nodes, and all nodes presenting the symptom make a decision that one or more nodes should change their mappings.

3) Cache Locality Metric: The cache locality metric utilizes the L2 cache misses monitored by the SyMMer library. This metric is specific to the processor architecture, and relies on the locality of cache between cores on the same die. If the number of cache misses for a process is sufficiently greater than those observed by another process on the same node, then these two processes can be swapped as long as the communicating process is moved *closer* to the core performing the protocol stack processing. In some cases it can even determine that the protocol processing core is the best location to place a process which has high enough cache misses, since that core will have no misses, the same as the other core on the same die, allowing two processes to be mapped in this manner rather than just one. This is an example of prioritized metrics, in that in these cases, following the communication idleness metric would result in a below optimal result. Again, this metric only relies on intra-node communication as it is only used to switch processes to the respective cores within the same node.

V. Experimental Evaluation

In this section, we evaluate our proposed approach with multiple microbenchmarks in Section V-A and the GROMACS and LAMMPS applications and FFTW Fourier Transform library in Section V-B. The analysis of each microbenchmark uses a variable component which we found to have an effect on the performance of the benchmark and the asymmetry we have been monitoring, the applications lack this because they do not expose such malleable characteristics. Thus the applications are presented as a simple comparison of time with or without the SyMMer library.

The cluster testbed used for our evaluation consisted of two Dell PowerEdge 2950 servers, each equipped with two dual-core Intel Xeon 2.66GHz processors. Each server has 4GB of 667MHz DDR2 SDRAM. Each processor has a 4MB L2 cache which is shared

between two cores. The machines run the Fedora Core 6 operating system with Linux kernel version 2.6.18 and were connected using NetEffect NE010 10-Gigabit Ethernet network adapters.

A. Microbenchmark Evaluation

In this section, we evaluate the three microbenchmarks we developed specifically to test each of the symptoms described in Section III-C individually, and the benefits achievable by the SyMMer library in these cases. Each microbenchmark was thoroughly tested to ensure that it stressed its symptom and not the others, since cache locality can always effect performance, it should be noted here that while it does make a difference in the first two, the difference is completely eclipsed by the symptom they are designed to test, and is lost as statistical noise. In Section V-B, we will evaluate the SyMMer library with real applications and scientific libraries to show its impact in the *real world*.

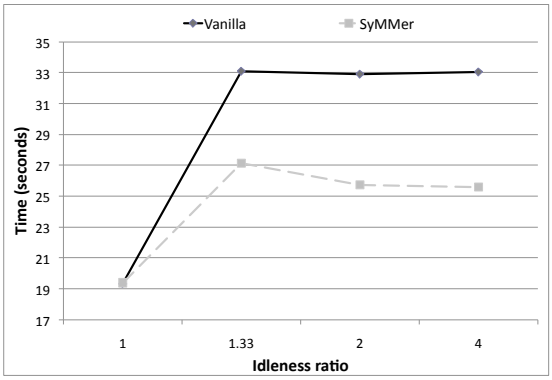

Fig. 6. Communication Idleness Benchmark Performance

1) Communication Idleness Benchmark: The communication idleness benchmark stresses the performance impact of delays due to irregular communication patterns. In this benchmark, processes communicate in pairs using MPI_Send and MPI_Recv, with each pair performing *different* ratios of computation between each communication step. Thus, a pair that is performing less computation spends more time in an idle state waiting for communication. Such processes are less impacted by the protocol processing overhead on the same core as compared to other processes which spend more of their time doing computation.

Figure 6 shows the performance of the communication idleness benchmark using SyMMer-enabled MPICH2 as compared to vanilla MPICH2. We define idleness ratio to be the ratio between the computation done by the pair doing the most computation and the pair doing the least. This ratio hence represents the amount of computational irregularity in the benchmark. Thus an idleness ratio of 1 represents that all the processes in the benchmark perform the same amount of computation, while a value of 4 represents one communicating pair performs up to 4 times more computation than another one.

In Figure 6, we plot the time taken for the benchmark to execute with various idleness ratios. We observe that both vanilla and SyMMer-enabled MPICH2 perform the same for an idleness ratio of 1. This is expected given that an idleness ratio of 1 represents a completely symmetric benchmark with no irregularity in computation. Thus there is no scope for SyMMer to improve performance in this case. We do observe however that as the idleness ratio increases, with vanilla MPICH2, the performance of the benchmark increasingly depends on the mapping of processes to cores. That dependence makes it possible for SyMMer to use the disparity in computation to achieve a more optimal arrangement, reducing runtime by up to about 30%.

To further analyze the behavior of the benchmark, we show the distribution of time spent in the different parts of communication in Figures 7(a) and 7(b) for vanilla MPICH2 and SyMMer-enabled MPICH2 respectively. For consistency both are based on the same initial mapping of processes to cores with an idleness ratio of 4. Figure 7(a), shows that the wait times for the different processes are quite variable. This means, that some processes spend a lot of time waiting for their peer processes to respond, while other processes are overloaded with the application computation as well as protocol processing overhead. Figure 7(b), on the other hand, illustrates the distribution for SyMMer-enabled MPICH2. As shown in the figure, SyMMer keeps the wait time more even across the varied processes, thus reducing the overhead seen by the application.

2) Out-of-Sync Communication Benchmark: This benchmark emulates the behavior where two application processes perform a large synchronous data transfer and a large computation immediately thereafter. Because some of the user data may be buffered, usually due to a full buffer in a lower level communication layer, there is a possibility that the processes may go *out-of-sync*. In this case that refers to the situation where the sending and receiving processes are not assigned to cores with equal effective capabilities. This would result in data being buffered at the sender node, causing the send to be delayed, which results in the receiver process waiting not only for the amount of time it takes to send the data, but also for the time needed to complete the computation which is logically after the send and receive pair completes.

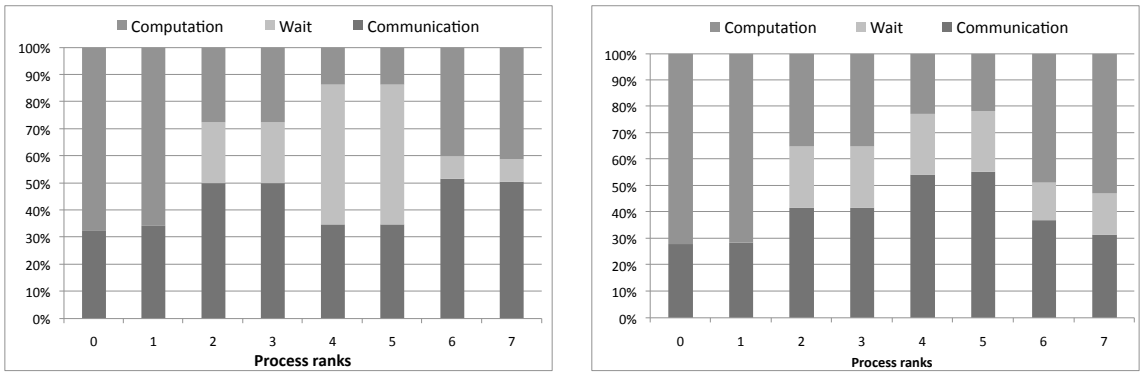

Fig. 7. Communication Idleness Benchmark devision of time: (a) Vanilla MPICH2 (b) SyMMer-enabled MPICH2

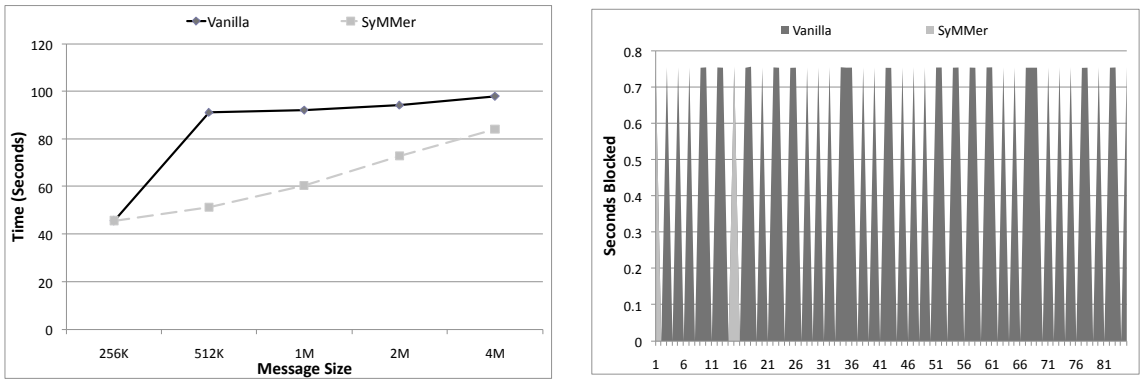

Fig. 8. Out-of-Sync Communication Benchmark: (a) Performance (b) MPI data buffering time

Figure 8(a) shows the performance of the out-of-sync communication benchmark using vanilla MPICH2 and SyMMer-enabled MPICH2, it compares the total time in seconds to execute the benchmark with various message sizes used in the communication step. Similar to the communication idleness benchmark, SyMMer-enabled MPICH2 performs as well as or better than vanilla MPICH2 in all cases. At message sizes up to 256 KB, both vanilla and SyMMer MPICH2 have the same performance. This is because messages at and below 256 KB are sent out without buffering by MPICH2 (*eager* messages). Because of the absence of buffering until 256 KB, there can be no out-of-sync behavior, which represents our base case where there is no scope for performance improvement. For message sizes above 512 KB however, we observe that SyMMer enabled MPICH2 consistently outperforms vanilla MPICH2 by up to 80%. As the message size continues to rise above 512 KB however, the performance gap between SyMMer and vanilla begins to close. We believe this to be caused by MPI's tendency to resist buffering larger messages when possible. A detailed analysis of this is left for future work.

The internal workings of the benchmark are demonstrated in Figure 8(b), which shows the times when data is buffered within the MPI library running under the same initial conditions with and without SyMMer. As shown in the figure, the data buffering time is almost an order-of-magnitude less when using SyMMer. It should be noted that, when an out of sync message occurs with SyMMer, the time taken is the same, but since SyMMer corrects the error in synchronization, it doesn't happen further times. This ultimately results in the performance improvement demonstrated in Figure 10.

3) Cache Locality Benchmark: This benchmark stresses the cache locality of processes by doing the majority of the network communication in *certain* processes in the application. Thus, depending on which core the communicating processes are mapped to, they may present the cache locality symptom discussed in Section III-C, which lets us know that there is an impact on the performance of the application. Hence, when the communicating processes are not on the same processor die as the core performing protocol processing, they can potentially take a severe performance hit.

In our benchmark, processes communicate in pairs wherein two pairs of processes are engaged in heavy inter-node communication, while the other pairs perform computation and exchange data locally. We measure the

performance as the time it takes to complete a certain number of iterations of the benchmark. This is portrayed in Figure-9(a) which showcases performance by comparing the total execution time with a computational load factor, which is effectively a measure of the amount of work done per run. We find that as the computational load factor increases, SyMMer is able to outperform vanilla MPICH2 by up to 29%.

We profile the benchmark and count the number of L2 cache misses observed by each process to gain a further understanding of SyMMer's behavior. Figure 9(b) shows the total number of cache misses observed by the processes performing inter-node and intra-node communication respectively. We observe that the number of cache misses is significantly decreased for the inter-node communicating processes, but despite the migration and other overhead incurred in the process, the cache misses fall for the intra-node communicating processes as well. As it turns out, the intra-node communicating processes gain two benefits from SyMMer that we didn't initially anticipate. Firstly, their locality with respect to the local processes they communicate with improves. Secondly, moving them away from the die doing the inter-node processing reduces the amount of cache thrashing they have to contend with. Thus SyMMer is not only able to improve the cache locality of the inter-node communicating processes, but also that of the intra-node communicating processes as well.

B. Evaluating Applications and Scientific Libraries

In this section, we evaluate the performance of two molecular dynamics applications, GROMACS and LAMMPS, and the FFTW Fourier Transform library and demonstrate the performance benefits achievable using SyMMer.

1) GROMACS Application: GROMACS (GROningen MAchine for Chemical Simulations) [9] is a molecular dynamics application developed at Groningen University, primarily designed to simulate the dynamics of millions of biochemical particles in a molecular structure. GROMACS is optimized towards locality of processes. It splits the particles in the overall molecular structure into segments, distributes different segments to different processes, and each process simulates the dynamics of the particles within its segment. If a particle interacts with another particle that is not within the process' local segment, MPI communication is used to exchange information regarding the interaction between the two processes. The overall simulation time is broken into many steps, and performance is reported as the number of nanoseconds per day of simulation time. For our measurements, we use the GROMACS LZM application.

2) LAMMPS Application: LAMMPS [10] is a molecular dynamics simulator developed at Sandia National Laboratory. It uses spatial decomposition techniques to partition the simulation domain into small 3D subdomains, one of which is assigned to each processor. This allows it to run large problems in a scalable way wherein both memory and execution speed scale linearly with the number of atoms being simulated. We use the Lennard-Jones liquid simulation with LAMMPS scaled up 64 times for our evaluation and use the communication time for measuring performance.

3) FFTW Scientific Library: Fourier Transform libraries are extensively used in several high-end scientific computing applications, especially those which rely on periodicity of data volumes in multiple dimensions (e.g., signal processing, numerical libraries). Due to its high computational complexity, scientists typically use Fast Fourier Transform (FFT) algorithms to compute the Fourier transform and its inverse. FFTW [11] is a popular parallel implementation of FFT.

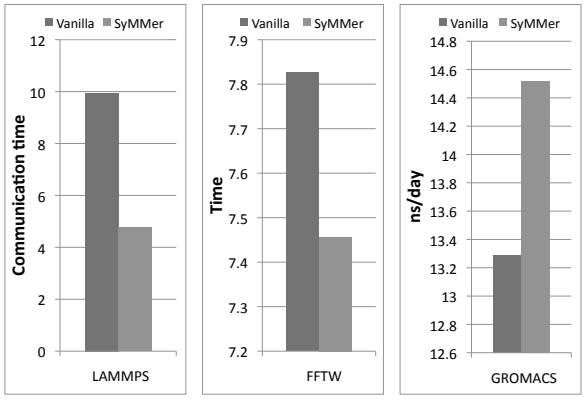

Fig. 10. Performance Evaluation: (A) LAMMPS (B) FFTW, lower is better (C) GROMACS, higher is better

Figure 10 illustrates the performance achieved by SyMMer-enabled MPICH2 as compared to vanilla MPICH2 for GROMACS, LAMMPS and FFTW. As shown in the figure, SyMMer-enabled MPICH2 can remap processes to the right cores so as to maximize performance, resulting in a performance improvement across the board. For GROMACS, the overall application execution time is presented, in the form of its own internal performance measure ns/day, which shows an improvement of about 10-15%. For LAMMPS, communication overhead is presented due to the fact that communication in LAMMPS tends not to scale with problem size making it a more stable measure, which shows nearly a *two-fold* improvement in performance. For FFTW, execution time is presented based on the internal average execution time provided by the benchmark, we

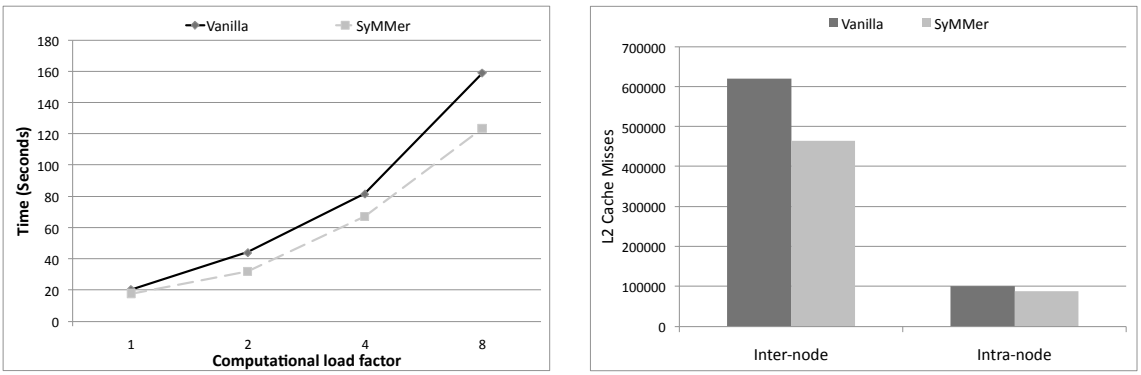

Fig. 9. (a) Cache locality performance (b) Cache miss analysis

noticed only about 3-5% performance difference in our experiments. This is due to the small communication volumes that are used for FFTW in our experiments. Given that SyMMer monitors for interactions between the communication protocol stack and the multi-core architectures, small data volumes mean that such interaction would be small as well.

In summary, we see a noticeable improvement in performance with SyMMer-enabled MPICH2 for all three cases. This shows that dynamic process-to-core mapping is a promising approach to minimize the impact of interactions of the communication protocol stacks with the multi-core architecture and allow us to improve application performance significantly in some cases.

VI. Discussion on Alternative Multi-core Architectures

While we demonstrated our approach on the Intel multi-core architecture, the idea of dynamically mapping application processes to the cores in a system is relevant to most current and next-generation multi- and many-core systems. For example, many-core accelerator systems such as GPGPUs provide complicated hierarchical architectures. The new NVidia Tesla system, for instance, has up to 16 GPUs, with each GPU having up to 128 processing cores. The cores in a GPU are grouped together into different multi-processor blocks where cores within a block communicate via fast shared memory, while cores across blocks share lower-speed global device memory. Cores between different GPUs can only communicate through the system bus and host memory. Such architectures clearly are highly sensitive to the placement of processes on the different computation units, and would certainly benefit from the knowledge gained in development of a mapping library such as SyMMer, if not the facilities of the library itself.

AMD-based multi-core architectures are quite similar to Intel-based multi-core architectures, and as such one might think they would behave the same. There are however, a few key differences which need to be addressed before they can utilize SyMMer-like process management libraries to full effect. Specifically, the non-uniform memory access (NUMA) model which they use makes process management more complicated as compared to Intel systems. For example, on an Intel system, SyMMer could freely move any process to any core in the system with the only migration cost being re-populating the new cache from main memory. However, on an AMD system, as soon as a process touches a memory buffer, this buffer is allocated on its local memory. At this time, if the process is migrated to a different die, all of its memory access will be remote, resulting in significant overhead. For example, we noticed that for the LAMMPS application, process migration can increase the non-local memory transfers by 17-31 fold in some cases. While the SyMMer architecture is still valid for AMD systems, aspects such as page migration between different memory controllers become important concerns in achieving good performance. In its absence, SyMMer would be restricted to process mapping only to cores within a die, which would result in a more limited improvement in performance.

Finally, with the upcoming network-on-chip (NoC) architectures such as Intel Xscale (a.k.a. Intel terascale) processors and Tilera processors, dynamic process-to-core mapping will become increasingly important. Primarily owing to the huge number of cores that will reside on the same die. For example, the Intel Xscale processors would accommodate 80 cores per die, while Tilera processors accommodate 64. In such cases, assigning a process to the wrong die could lead to a significant amount of inter-die communication which would be largely limited by the die pin overhead. A library like our novel process-to-core mapping framework, could dynamically *search* for the right assignments and use them to achieve better and more consistent performance.

VII. Related Work

While both multi-core architectures and communication protocol stacks are heavily studied topics, to the best of our knowledge, there is no existing literature that directly studies the interactions between these two aside from our previous paper [6].

In [14], the authors quantify the performance difference in asymmetric multi-core architectures and define operating system constructs for efficient scheduling on such architectures. This closely relates to the *effective capability* of the cores (similar to the current paper), but the authors only quantify asymmetry that already exists in the architecture (i.e., different core speeds). In our work, we study the impact of the interaction of multi-core architectures with communication protocol stacks, which externally creates such asymmetry through aspects such as interrupts and cache sharing.

In [15], the authors study the impact of the Intel multi-core architecture on the performance of various applications by looking at the amount of intra-node and inter-node communication performed. Some of the performance problems of multi-cores are identified in that paper and the importance of multi-core-aware communication libraries is underlined. They also discuss a technique of *data tiling* for reducing the cache contention which is dependent on the cache size. While the underlying principle and approach of this work significantly differs from our approach, we believe that these two techniques can be utilized in a complementary manner to further improve performance.

The authors of [16], [17] look at the impact of shared caches on scheduling processes or threads on multi-core architectures wherein certain mappings of processes to cores would work better in terms of utilizing it. We improve upon their work by generalizing the extraneous factors that affect application performance on multi-core systems and by devising a framework for dynamically performing the ideal mapping of processes to cores.

In summary, our work differs from existing literature with respect to its capabilities and underlying principles, but at the same time, forms a complementary contribution to other existing literature that can be simultaneously utilized.

VIII. Concluding Remarks and Future Work

In this paper, we demonstrated that the interactions between the communication stack and multi-core architectures can result in heavy asymmetry in the *effective capabilities* of the different cores; this results in significant performance degradation for various applications. We further presented the design and evaluation of a novel systems software stack, known as *SyMMer* (Systems Mapping Manager) library, that monitors such interactions and dynamically manages the mapping of processes onto processor cores so as to improve performance. Our evaluation of the SyMMer library demonstrated nearly a *two-fold* improvement in communication time and 10-15% improvement in overall performance for various applications.

As future work, we plan to study other multi- and many-core architectures such as AMD NUMA and GPGPU systems to understand the implications of novel process-to-core mapping on such systems.

References

[1] "Intel Core 2 Extreme quad-core processor," http://www.intel.com/products/processor/core2xe/qc_prod_brief.pdf.

[2] "AMD Quad-core Opteron processor," http://multicore.amd.com/us-en/quadcore/.

[3] "Sun Niagara," http://www.sun.com/processors/UltraSPARC-T1/.

[4] "Intel Terascale Research," http://www.intel.com/research/platform/terascale/teraflops.htm.

[5] "Tilera TILE64 Processor family," http://www.tilera.com/pdf/ProBrief_Tile64_Web.pdf.

[6] G. Narayanaswamy, P. Balaji, and W. Feng, "An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments," in *15th International Symposium on High-Performance Interconnects (HotI 2007)*, Palo Alto, California, August 2007.

[7] A. N. Laboratory, "Mpich2: High performance and portable message passing."

[8] "Message Passing Interface Forum," http://www.mpi-forum.org.

[9] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1-3, pp. 43–56, September 1995. [Online]. Available: http://dx.doi.org/10.1016/0010-4655(95)00042-E

[10] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995.

[11] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization and Platform Adaptation".

[12] "PAPI," http://icl.cs.utk.edu/papi/.

[13] "PERUSE," http://www.mpi-peruse.org/.

[14] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures," in *SC '07*, 2007.

[15] L. Chai, Q. Gao, and D. K. Panda, "Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System," in *CCGrid '07*, 2007.

[16] J. Anderson, J. Calandrino, and U. Devi, "Real-time scheduling on multicore platforms," *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pp. 179–190, 04-07 April 2006.

[17] A. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair thread scheduling for multicore processors," Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06, October 2006.