# On the Interaction of High-Performance Network Protocol Stacks with Multicore Architectures

Ganesh Chunangad Narayanaswamy

Thesis submitted to the faculty of the
Department of Computer Science
at the
Virginia Polytechnic Institute and State University

In partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

April 18, 2008
Blacksburg, Virginia

Keywords: Multicore Architectures, High-Performance Networking, Process-to-Core Mapping, Network Contention, Runtime Adaptation

# On the Interaction of High-Performance Network Protocol Stacks with Multicore Architectures

Ganesh Chunangad Narayanaswamy

## ABSTRACT

Multicore architectures have been one of the primary driving forces in the recent rapid growth in high-end computing systems, contributing to its growing scales and capabilities. With significant enhancements in high-speed networking technologies and protocol stacks which support these high-end systems, a growing need to understand the interaction between them closely is realized. Since these two components have been designed mostly independently, there tend to have often serious and surprising interactions that result in heavy asymmetry in the *effective capability* of the different cores, thereby degrading the performance for various applications. Similarly, depending on the communication pattern of the application and the layout of processes across nodes, these interactions could potentially introduce network scalability issues, which is also an important concern for system designers.

In this thesis, we analyze these asymmetric interactions and propose and design a novel systems level management framework called *SIMMer* (Systems Interaction Mapping Manager) that automatically monitors these interactions and dynamically manages the mapping of processes on processor cores to transparently maximize application performance. Performance analysis of SIMMer shows that it can improve the communication performance of applications by more than two-fold and the overall application performance by 18%. We further analyze the impact of contention in network and processor resources and relate it to the communication pattern of the application. Insights learnt from these analyses can lead to efficient runtime configurations for scientific applications on multicore architectures.

To my parents

To my family and friends

To the thirty–two Hokies
who lost their lives on 04/16/07

# Acknowledgements

I have had the fortune of being part of an academically stimulating community here at Virginia Tech for the past two years. Many people have given me intellectual support and stimulation in developing my ideas, and while it may never be possible to thank them enough for their contributions, its a pleasure to acknowledge those people who have made this thesis possible.

Its difficult to overstate my gratitude to my advisor, Dr. Wu-chun Feng for his kind support, probing critiques and remarkable patience. He has always motivated me to think independently and believed in my capabilities. I would like to thank him for nominating me for the Outstanding Master's Student Award in the College of Engineering. I owe a huge amount of gratitude to Dr. Pavan Balaji, my committee member, for his guidance and never ending willingness to help me. It truly has been an exceptionally rewarding experience working with him. My sincere thanks to Dr. Dimitrios S. Nikolopoulos, for his valuable suggestions and contributions towards this thesis and also for the gratifying experience of working with him as a teaching assistant during my first year.

My sincere appreciation to all the other faculty in the Department of Computer Science at Virginia Tech with whom I have had the good fortune to interact with and work on their projects. My sincere thanks are due to Dr. William Gropp, Dr. Rajeev Thakur, and Dr. Darius Buntinas at Argonne National Laboratory for the intellectually stimulating experience as a research intern at ANL.

I am indebted to Tom Scogland for working with me and helping me out with various aspects of this work. My thanks also to Ashwin Aji, Jeremy Archuleta and other members of the Synergy Laboratory for their valuable comments and suggestions. I am also grateful to my numerous other friends who have helped me directly or indirectly in my research work.

Finally, I dedicate this thesis to my parents for their love and for providing me with the means to learn and understand as an individual. I cannot thank them enough for the values they have inculcated in me, without which I would not the person I am today. I am grateful to God for providing me with the strength and intellectual capability to complete this thesis.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Statement of the Problem

## 1.1 Motivation

As scientific applications grow in scale and complexity, high-end computing (HEC) systems that are required to meet their demands have grown accordingly. Such systems are increasingly being characterized by nodes built out of commodity components, which enables rapid deployment and performance scaling. Two of the significant trends in the HEC domain over the past few years have been the dramatic improvements in processor technology (with the advent of multicore architectures) and in networking technology (using high-performance networks).

Multicore architectures have established themselves as a major step to the growing scales and capabilities of modern HEC systems and contributed significantly to its rapid growth [30, 21]. Increasing power consumption, heat dissipation, and cooling requirements with complex single-core processors have forced processor designers to turn to multicore architectures as a solution for keeping up with Moore's Law. Multicore processor architectures are bringing the performance of a large multiprocessor system down to the chip level, providing a significant level of throughput in a small package with lower power consumption. The commodity market already has quad-core architectures from Intel [25] and AMD [6]. Processors with larger core counts, such as the IBM

Cell [12], Intel Terascale [26], and Sun Niagara [32], are also gaining in popularity.

On the other hand, high-performance networks such as 10-Gigabit Ethernet (10GE) [19,18,16], Myrinet [31], and InfiniBand (IB) [24] are increasingly becoming an integral part of large-scale systems with respect to scalability and performance. These network technologies use highly optimized protocols and hardware implementations to realize low-latency and high-bandwidth communications. As these two trends emerge, the interaction between them and the its impact on application performance needs to be understood clearly. Also, potential bottlenecks caused by this interaction needs to be identified and possible solutions must be explored.

While many of the multicore architectures have symmetric computational capabilities and physical parameters, the high-performance communication protocol stacks running on them frequently fails to maintain that symmetry up to the application level. Specifically, it handles incoming packets independent of the application receiving the packets and the processing core running the application process. A significant amount of processing is statically fixed to a single core in the system, resulting in processing imbalance and consequently adverse effects on applications in two primary aspects. First, the *effective capability* that the overloaded core can provide to the application is reduced. Second, the data that is processed by the protocol stack is now localized to this core rather than to the process to which it belongs, thus resulting in cache misses for the process. *Accordingly, depending on which application process is assigned to which core, the performance of the application can vary significantly.*

While multicore architectures improve the computational throughput of a node by increasing the number of processes that reside on the same physical node, they share the same physical network resulting in *increased* network utilization. At the same time, however, given the increase in intra-node shared-memory communication for processes residing on the same node, network usage can potentially *decrease* significantly. Based on these two conflicting possibilities, we need to understand whether modern multicore architectures add extra requirements on networks requiring future HEC systems to scale up network capacity further, or whether the increase in intra-node shared memory communication compensates for the increase in network contention, thus not requiring any changes. Therefore, depending on the application communication pattern and the layout of processes across nodes, interesting questions arise about network resource contention and scalability. Also, the impact of network contention needs to be analyzed against that of contention in processor resources itself (e.g. shared caches, issue queues etc).

Thus, in the first part of this thesis, we investigate the interactions caused by the asymmetry between multicore architectures and application processing requirements with microbenchmarks and real life applications. Then, we utilize the lessons learned from this analysis to design an intelligent Systems Interaction Mapping Manager (SIMMer) framework within the Message Passing Interface (MPI) communication library, which is the *de facto* parallel programming model in high-end computing systems for a vast majority of scientific applications. SIMMer efficiently monitors these interactions and dynamically manages the mapping of processes onto processor cores so as to maximize performance, thereby transparently hiding the details to the programmer or the end user. Specifically, the SIMMer library uses several heuristics to identify asymmetry between the effective capabilities of cores (i.e. either computational capabilities or cache behavior) and the processing requirements of the application. If substantial asymmetric behavior is detected, SIMMer transparently re-maps the processes, thereby achieving improved performance. In cases where significant asymmetric interactions are not detected within a node, we study various interactions across nodes that result in network contention. Thus, in the second part of this thesis, we investigate the influence of contention in network and processor resources in application performance. We then relate our results with various process allocation schemes and application communication patterns. Both these analyses, when looked at holistically, can lead to intelligent layout of processes both within a node and across nodes and hence result in better run-time execution environments for applications on multicore architectures.

## 1.2 Contributions

This thesis has the following significant contributions:

- Demonstration and evaluation of various asymmetric interactions between multicore architectures and high-performance network protocol stacks.
- Synthesis and evaluation of intelligent mappings of processes to cores based on insights learnt from the types of these interactions.
- Proposal, design and implementation of SIMMer, a framework which profiles parallel applications and applies intelligent process-core mappings automatically to eliminate possible asymmetric interactions between multicore architectures and application requirements.
- Validation of SIMMer against microbenchmarks and scientific applications and show how it can improve application communication performance by more than two-fold.
- Analysis of network and processor resource contention issues and relate it to process layout schemes and application communication patterns.
- The insights gained from all of the above can be applied in designing better run-time configurations for applications on multicore architectures.

## 1.3 Outline

The rest of this thesis is arranged as follows. In Chapter 2, we present some background on multicore architectures and the operating system and architectural viewpoint of the communication stack. In Chapter 3, we analyze the asymmetric interactions between multicore architectures and the protocol stack and synthesize intelligent mapping of processes to cores for various applications. In Chapter 4, we present the design of the SIMMer library and evaluate it against a few microbenchmarks and applications. Chapter 5 studies the impact of resource contention on the processor and the network caused by protocol stacks on multicore architectures. Chapter 6 presents some related work and we follow it up with our concluding remarks in Chapter 7.

# Chapter 2

# Background

In this section, we provide a detailed description of the interaction of multicore architectures with the TCP/IP communication stack residing in the operating system (OS). We start with an overview of the relevant architectural features of multicore processors in Section 2.1. Then, we provide the operating system and architectural viewpoints of the behavior of the TCP/IP stack and their interactions in Sections 2.2 and 2.3. We also provide some background on process allocation schemes and their importance in Section 2.4.

## 2.1 Overview of Multicore Architectures

For many years, hardware manufacturers have been replicating components on processors to create multiple pathways allowing more than one instruction to run concurrently with others. Duplicate arithmetic and floating point units, co-processing units, and multiple thread contexts (SMT) on the same processing die are examples of such replication. Multicore processors are considered to be the next step in such hardware replication where two or more (mostly) independent execution units are combined onto the same integrated circuit.

Multicore architectures are at a high level similar to multi-processor architectures. The operating system deals with multiple cores in the same way as multiple processors by allocating one process to each core at a time. Arbitration of shared resources between the cores happens completely in hardware with no intervention from the operating system. However, multicore processors are also very different from multi-processor systems. For example, in multicore 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. Further, architectures such as the current Intel multicores, as shown in Figure 2.1, provide a shared cache between the different cores on the same die. This makes communication even simpler by eliminating the need for complicated cache-coherency protocols found in multi-cache systems (although level 1 cache coherency cannot be avoided).

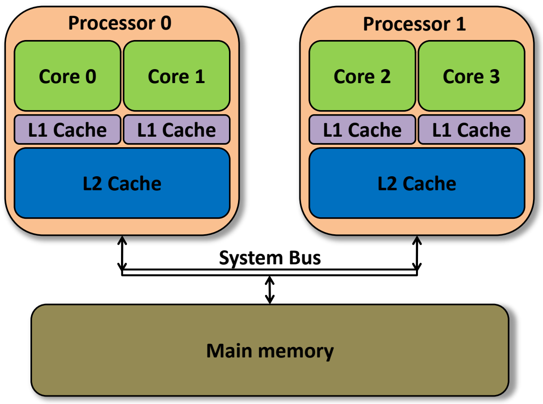


Figure 2.1: Intel Dual-core Dual-processor System

However, multicore processors also have the disadvantage of more shared resources as compared to multi-processor systems. That is, multicore processors might require different cores on a processor die to block waiting for local shared resources to get freed when it is being used by a different core. Such contention is even higher when the ratio of number of cores on the system increases as compared to the other resources (e.g., multicore systems with multiple thread contexts). Further, for architectures such as AMD NUMA, each processor in a multi-processor system has access to its own memory and hence overall memory bandwidth essentially doubles with the number of processors. However, for multicore systems, the overall memory bandwidth does not change.

## 2.2 Operating System Viewpoint of the Communication Stack

Like most communication protocol suites, the TCP/IP protocol suite is a combination of different protocols at various levels, with each layer responsible for a different facet of the communications.

To allow standard Unix I/O system calls such as `read()` and `write()` to operate with network connections, the file-system and networking facilities are integrated at the system-call level. Network connections represented by sockets are accessed through a descriptor in the same way an open file is accessed through a descriptor. This allows the standard file-system calls such as `read()` and `write()`, as well as network-specific system calls such as `send()` and `recv()`, to work with a descriptor associated with a socket.

On the transmission side, the message is copied into the socket buffer, data integrity ensured through checksum computation (to form the TCP checksum) and passed on to the underlying IP layer. The checksum computation on the sender side is usually performed during the copy operation to maximize the cache effect (Jacobson's optimization). The IP layer fragments the data to MTU-sized chunks, constructs the IP header, and passes on the IP datagram to the device driver. The device driver then makes a descriptor for the packet and passes the descriptor to the network adapter using a PIO (Programmed I/O) operation. The network adapter performs a DMA operation to move the actual data indicated by the descriptor from the socket buffer to the network adapter buffer and raises an interrupt to inform the device driver that it has finished moving the data. The network adapter then ships the data with the link header to the physical network.

On the receiver side, the network adapter DMAs received segments to the socket buffer and raises an interrupt to inform the device driver. The device driver hands it over to the IP layer using a software interrupt mechanism. The interrupt handler for this software interrupt has a higher priority compared to the rest of the kernel. The IP layer verifies the IP checksum, and if the integrity is maintained, defragments the data segments to form the complete TCP message and hands it over to the TCP layer. The TCP layer verifies the data integrity of the message and places it in the socket buffer. When the application calls the `read()` operation, the data is copied from the socket buffer to the application buffer.

## 2.3 Architectural Viewpoint of the Communication Stack

While the TCP/IP data and control paths are relatively straightforward with respect to the operating system viewpoint, they have a number of implications from an architectural viewpoint, specifically relevant to multicore architectures. In this section, we present the compute processing and cache-related impact that the architecture can have on the stack.

**Processing Impact:** As described in Section 2.2, when a packet arrives, the network adapter places the data in memory and raises an interrupt to inform the device driver about the arrival of the packet. 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 need not be the same core on which the process performing the relevant communication resides. Further, the core which receives the hardware interrupt performs the relevant protocol processing for the incoming data as well. This includes data integrity checks, connection demultiplexing, and other such compute intensive operations that can significantly impact the computational load on the *chosen core*. Also, the interrupt scheduling granularity of these utilities is many times coarser than the granularity with which interrupts are received. Thus, any interrupts received between scheduling periods will still be delivered to the same core.

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

**Cache Transaction Impact:** Aspects of TCP/IP processing such as data copies and checksum-based data integrity require the protocol 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 (DMAs) 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 may have 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 again 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. However, if the application process resides on a different die, then the data has to be fetched using a cache-to-cache transfer (over the system bus in the Intel architecture).

At this point, the impact of protocol offloaded stacks such as Internet Wide Area RDMA Protocol (iWARP) needs to be understood clearly. Such stacks use the hardware capabilities of the network interface card (NIC) to offload the protocol stack onto the NIC and also bypass the operating system in delivering the packet directly to the application. Thus, such stacks may not have any of the impacts discussed above because of extra hardware capabilities provided by the NIC. Nevertheless, the interaction of such stacks with multicore architectures introduce a different kind of interaction — that of network contention. Since such offloaded NICs do not have similar hardware capabilities as the computational cores themselves, they may face some contention when handling all the network packets for all the computational cores in the system. We study the impact of this interaction in the second part of the thesis in Chapter 5.

## 2.4 Process Allocation Schemes

In a multicore cluster, the processes can be arranged among the nodes in several ways. Applications typically have fixed communication patterns, and allocation schemes provide us the flexibility of modifying which processes get collocated on the same node. Thus, depending on the allocation scheme, the amount of network sharing might increase or decrease. We look at two common allocation schemes in this thesis: cyclic and blocked allocation.

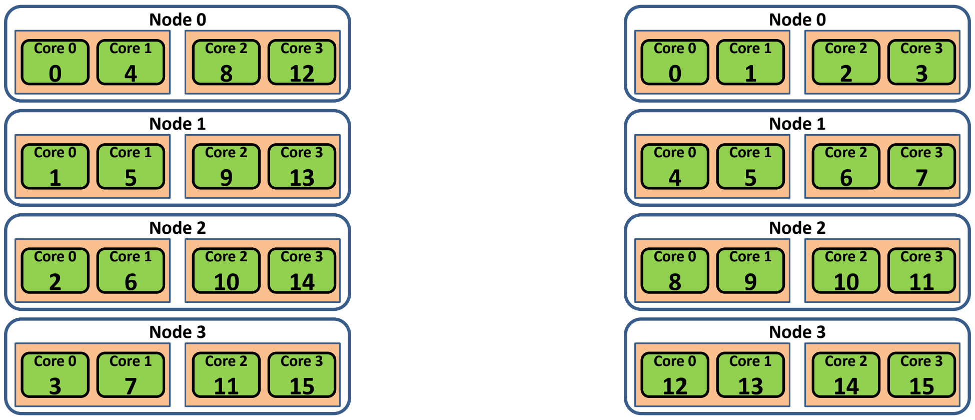

Node 0
Core 0 0 Core 1 4 Core 2 8 Core 3 12
Node 1
Core 0 1 Core 1 5 Core 2 9 Core 3 13
Node 2
Core 0 2 Core 1 6 Core 2 10 Core 3 14
Node 3
Core 0 3 Core 1 7 Core 2 11 Core 3 15
Node 0
Core 0 0 Core 1 1 Core 2 2 Core 3 3
Node 1
Core 0 4 Core 1 5 Core 2 6 Core 3 7
Node 2
Core 0 8 Core 1 9 Core 2 10 Core 3 11
Node 3
Core 0 12 Core 1 13 Core 2 14 Core 3 15

Figure 2.2: Process Allocation Schemes: (a) Cyclic and (b) Blocked

Cyclic allocation allocates each subsequent process cyclically to the next node in the ring of nodes. For example, with a total of 16 processes and 4 nodes, process ranks 0, 4, 8 and 12 will get assigned to node 0, ranks 1, 5, 9 and 13 to node 1, and so on. This allocation ensures good load balance among all nodes. In blocked allocation, blocks of processes are assigned to each node in turn. For example, with 16 processes, 4 nodes and a block size of 4, process ranks 0, 1, 2 and 3 get assigned to node 0, ranks 4, 5, 6 and 7 to node 1, and so on. Blocked allocation typically enables full utilization of all available resources of a node. Figures 2.2 (a) and (b) show cyclic and blocked allocation with an example of 16 processes and 4 nodes with 4 cores per node.

The process allocation scheme can play an important role in the kind of communication performed by a process. For example, for an application that does mostly neighbor communication in a 1-D chain of processes, blocked allocation will perform better. The reason is that the neighbor processes that a process communicates with are more likely to be on the same node. The result can be significant reduction in network communication, thereby potentially improving performance. Since the number of neighbors is constant, adding more cores to a node does not have scope for improving locality further.

In a 2-D grid of $N \times N$ processes performing neighbor communication with $M$ cores in a node, again blocked allocation works better than cyclic allocation in localizing more neighbors when $N > M$. When $M$ and $N$ are equal, the same number of neighbors co-exist with both cyclic and blocked allocation. The same holds true for a 3-D grid of processes as well. Thus, for neighbor communication, more neighbors will co-exist with blocked allocation.

As another example, for an application which performs tree-like regular long distance communication, a cyclic allocation strategy might be a better choice, as it might localize many of the communicating processes within a node. For applications running on large clusters with hierarchical layers of switches, allocation schemes that localize branches of trees within the lowest hierarchy might be more beneficial.

# Chapter 3

# Intelligent Mapping of Processes to Cores

In this chapter, we investigate the performance impact of the interaction of multicore architectures with network communication stacks as introduced in Section 2.3 . This analysis will provide a better understanding about these interactions and how we can design solutions to prevent them from adversely affecting application performance. We use the analysis developed in this section to design the intelligent mapping of processes to cores and, in the next section, develop a library to transparently handle these interactions. Our analysis in this section is with the TCP/IP stack over 10-Gigabit Ethernet.

After describing our experimental setup in Section 3.1, we perform a microbenchmark based analysis in Section 3.2. We then show the impact of this interaction on the FFTW Fast Fourier Transformation library in Section 3.3. In Section 3.4, we follow this up with results on the GROMACS and LAMMPS molecular dynamics applications and design intelligent process-core mappings which can improve application performance.

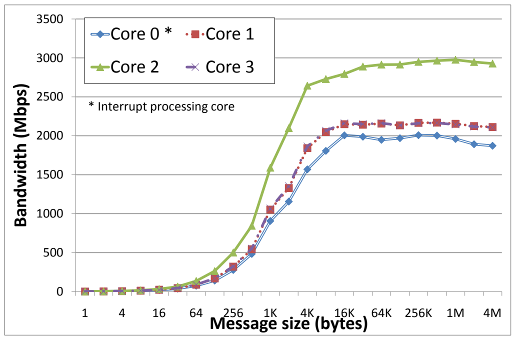
## 3.1 Experimental Testbed

Before looking at the evaluation results, we describe the two cluster setups that we used in our study.

**Intel Cluster:** Two Dell Poweredge 2950 servers, each equipped with two dual-core Intel Xeon 2.66-GHz processors. Each server has 4 GB of 667-MHz DDR2 SDRAM. The four cores in each system are organized as cores 0 and 2 on processor 0, and cores 1 and 3 on processor 1 (as reported by `/proc/cpuinfo`). Each processor has a 4-MB shared L2 cache. The operating system used is Fedora Core 6 with kernel version 2.6.18.

**AMD Cluster:** Two custom-built, dual-processor, dual-core AMD Opteron 2.6-GHz systems. Each system has 4 GB of DDR2 667-MHz SDRAM. Each core has a separate 1-MB L2 cache. Both machines run SuSE 10 with kernel version 2.6.13. In this case, the cores are numbered as cores 0 and 1 on processor 0 and cores 2 and 3 on processor 1.

**Network and Software:** Both clusters used NetEffect 10GE adapters installed on a x8 PCI-Express slot and connected back-to-back. For our evaluation, we used the MPICH2 (version 1.0.5p4) implementation of MPI.



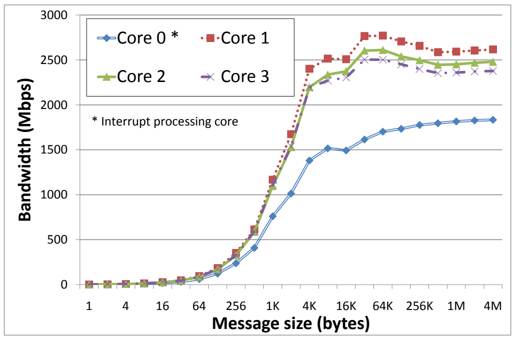


Figure 3.1: MPI Bandwidth: (a) Intel Cluster and (b) AMD Cluster

## 3.2 Analysis with Microbenchmarks

Here we evaluate performance of the clusters with respect to bandwidth and latency, using the appropriate microbenchmarks from the OSU MPI benchmark suite. We choose the OSU benchmark suite for their simplicity and ease of analysis. These evaluations are discussed in Sections 3.2.1 and 3.2.2. We then analyze the processing and cache transaction impacts of these results in Section 3.2.3. Each benchmark was run at least 5 times, and the average of all runs is reported.

### 3.2.1 MPI Bandwidth Evaluation

In the bandwidth microbenchmark, the sender sends a single message of size $S$ to the receiver many times. On receiving all the messages, the receiver sends back one small message to the sender informing that it has received the messages. The sender measures the total time and calculates the amount of data that it had transmitted per unit time.

Figures 3.1(a) and 3.1(b) show the MPI bandwidth achieved by TCP/IP on the Intel and AMD clusters respectively, when scheduled on each of the four cores in the system. Both the sender and the receiver process are scheduled on the same core number but on different servers. For the Intel cluster, Figure 3.1(a) shows several trends that are of interest to us. First, when the communication process is scheduled on core 0, bandwidth performance barely reaches 2 Gbps. Second, the benchmark performs slightly better when the communication process is scheduled on either core 1 or core 3, that is, cores on the second CPU of the Intel cluster. In this case, the benchmark achieves about 2.2 Gbps. Third, the benchmark achieves the best performance when the communication process is scheduled on core 2, that is, the second core of the first CPU. In this case, the benchmark achieves about 3 Gbps bandwidth, about 50% better than when the processes are scheduled on core 0.

The trends with the AMD cluster, as shown in Figure 3.1(b), are very similar to those observed in the Intel cluster. The interrupt processing core on the first CPU (core 0) achieves low performance, the cores on the second CPU (cores 2 and 3) achieve moderate performance, and the second core of the first CPU (core 1) achieves the best performance.

These results indicate that the interaction of the communication protocol stack with the multi-core architecture can have significant impact on performance.

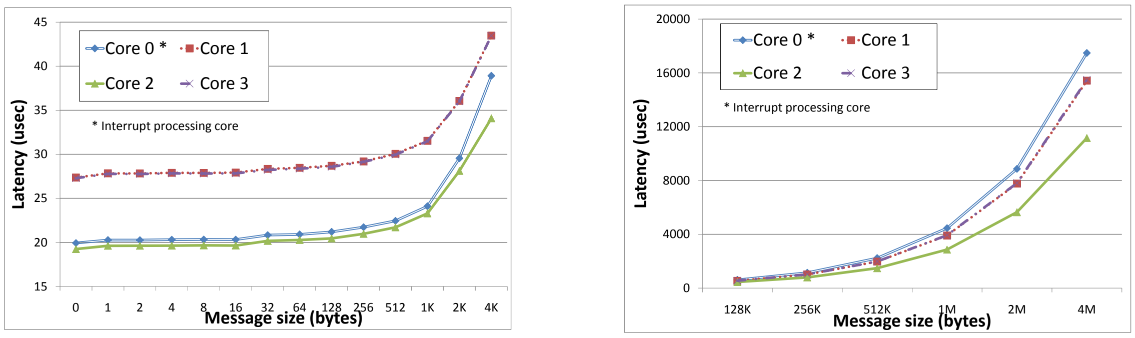


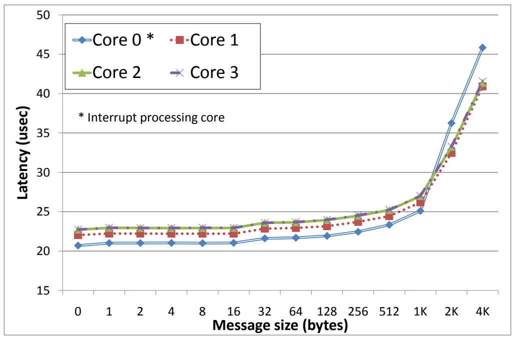
Figure 3.2: MPI Latencies with TCP/IP (Intel Cluster): (a) Small Messages and (b) Large Messages

### 3.2.2 MPI Latency Evaluation

In the MPI latency benchmark, the sender transmits a message of size *S* to the receiver, which in turn sends back another message of the same size. This is repeated several times and the total time averaged over the number of iterations – this gives the average round-trip time. The ping-pong latency reported here is one-half of the round-trip time.

Figure 3.2 illustrates the MPI latency achieved over TCP/IP when scheduled on each of the four cores in the Intel cluster. Again, both the sender and receiver processes are scheduled on the same core number but on different servers. To better illustrate the results, we have separated them into two groups. Figures 3.2(a) and 3.2(b) show the measurements for small and large messages, respectively.

Figure 3.2(a) shows that the best performance is achieved when the communication process is



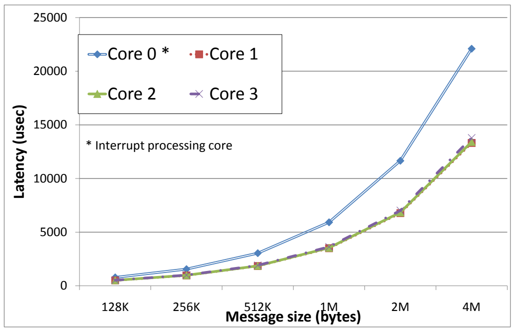


Figure 3.3: MPI Latencies with TCP/IP (AMD Cluster): (a) Small Messages and (b) Large Messages

on core 2. When the communication process is scheduled on core 0, however, performance drops only slightly, unlike the MPI bandwidth results. When the communication process is scheduled on cores 1 or 3, we see that the performance achieved is the worst.

The difference in the performance of core 0 for the latency test compared to the bandwidth test is attributed to the synchronous nature of the benchmark. That is, for small messages, data is sent out as soon as `send()` is called. By the time the sender receives the pong message, the TCP/IP stack is idle (no outstanding data) and ready to transfer the next message. On the receive side, when the interrupt occurs, the application process is usually waiting for the data. Thus, the interrupt does not interfere with other computation and hurt performance. Also, core 0 has the data in cache after the protocol processing; thus, if the application is scheduled on the same core, it can utilize this cached data, resulting in higher performance for core 0 as compared to cores 1 and 3. For large messages, however, the benchmark is no longer synchronous. That is, as the data is being copied into the sockets buffer, the TCP/IP stack continues to transmit it. Thus, both the asynchronous kernel thread (which is always statically scheduled on core 0) and the application thread might be active at the same time, resulting in loss of performance. This is demonstrated in Figure 3.2(b).

Figures 3.3(a) and 3.3(b) show the MPI latencies for small and large messages, respectively on the AMD cluster. We see that the AMD cluster also follows the same trends as the Intel cluster. The interrupt processing core (core 0), results in the best performance for small messages, while its performance quickly deteriorates for large messages. Core 1, the second core on the first processor, achieves good performances for both small and large messages.

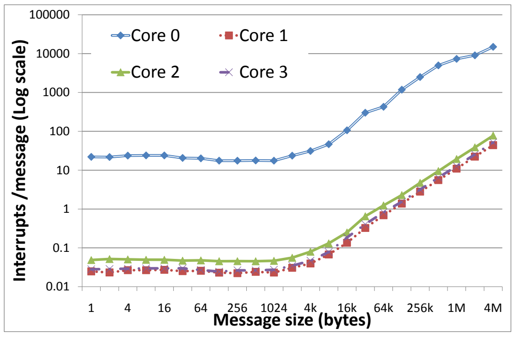
Both the MPI bandwidth and latency results show the presence of significant interaction between multicore architectures and the communication stack, and this interaction also results in considerable performance difference. This showcases the need to further analyze these results to understand these interactions in-depth. With this goal in mind, we analyze the microbenchmark results to gain further insights in the next section.

### 3.2.3 Analysis of Results

To further understand the microbenchmark results, we analyze in this section, the processing capability and cache transaction impact of the system while running the MPI bandwidth benchmark. To measure the impact on processing capability, we profile the number of hardware interrupts received by each core. This will represent the amount of protocol processing overhead incurred by each core, and hence the reduction in the *effective capability* of each core. The number of L2 cache misses incurred by each core will suffice to give us a very good idea of the cache sharing impact. Since both the Intel and AMD clusters show similar performance behavior, we look only at results on the Intel cluster.

#### Analysis of Processing Impact

To measure the interrupts generated by TCP/IP during the execution of the MPI bandwidth benchmark, we used the Performance Application Programming Interface (PAPI) [3] library (version 3.5.0). Figure 3.4 (a) illustrates the number of interrupts per message observed during the execution of the MPI bandwidth benchmark, which was scheduled on the different cores. As shown in



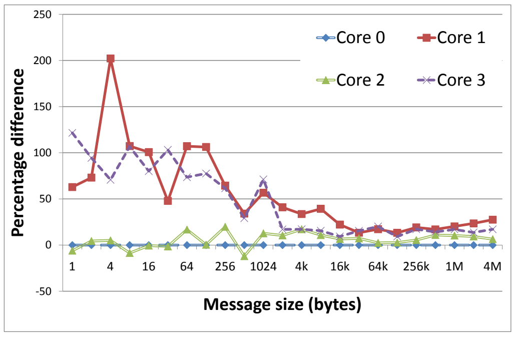


Figure 3.4: Analysis of MPI Bandwidth: (a) Interrupts and (b) Cache Misses

the figure, core 0 gets more than 99% of all the interrupts. This observation is in accordance with the processing impact of the communication stack as discussed in Section 2.3. That is, the *effective capability* of the interrupt processing core gets drastically reduced.

Based on the large number of interrupts, coupled with the asynchronous processing of the TCP/IP stack by the interrupt processing core, its capability to perform application processing gets affected. This results in reduced performance of the MPI bandwidth benchmark when the application process is scheduled on the interrupt processing core.

### Analysis of Cache Transaction Impact

As described in Section 2.1, multicore architectures provide opportunities for core-to-core data sharing either through shared caches (e.g., Intel architecture) or separate on-chip caches with fast connectivity (e.g., AMD architecture). In the case of TCP/IP, when interrupt processing is performed by a particular core, the data is fetched to its cache to allow for data-touching tasks such as checksum verification. Thus, if the application process performing the communication is scheduled on the same CPU but a different core, it can take advantage of the fast core-to-core on-die communication. In the Intel architecture, since the L2 cache is shared, we expect this to be reflected as substantially fewer L2 cache misses.

We verify our hypothesis by using PAPI to measure L2 cache misses. Figure 3.4 (b) shows the percentage difference in the number of L2 cache misses observed on each core compared to that on core 0. We observe that cores 0 and 2 (processor 0) have significantly lower L2 cache misses than do cores 1 and 3 (processor 1). These cache misses demonstrate the reason for the lower performance of the MPI bandwidth benchmark when the process is scheduled on either core 1 or core 3, as compared to when it is scheduled on core 2. The percentage difference in cache misses drops with larger message sizes because the absolute number of cache misses on the cores increases with message size as they cannot fit in the cache.

## 3.3 FFTW Scientific Library

The analysis and results for microbenchmarks explained in the previous section serve as good starting points for our study. In order to understand the implications of the interactions at the user level, we evaluate the performance of user-level libraries and applications. Towards this effect, we investigate a heavily used scientific library in this section.

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 [20] is a popular parallel implementation of FFT.

For our evaluation, we use the benchFFT benchmark on the Intel cluster and utilize all cores of both nodes to consider a more realistic scenario. Since 4 cores exist in each node, we have a total of $4! \times 4! = 576$ different configurations of mapping each process to every possible core. Figure 3.5 shows the sorted performance of the benchFFT benchmark of the FFTW library for all configurations of process-core mappings. Here we define performance as the time taken for the benchmark to finish execution. We can observe a significant variation in performance between

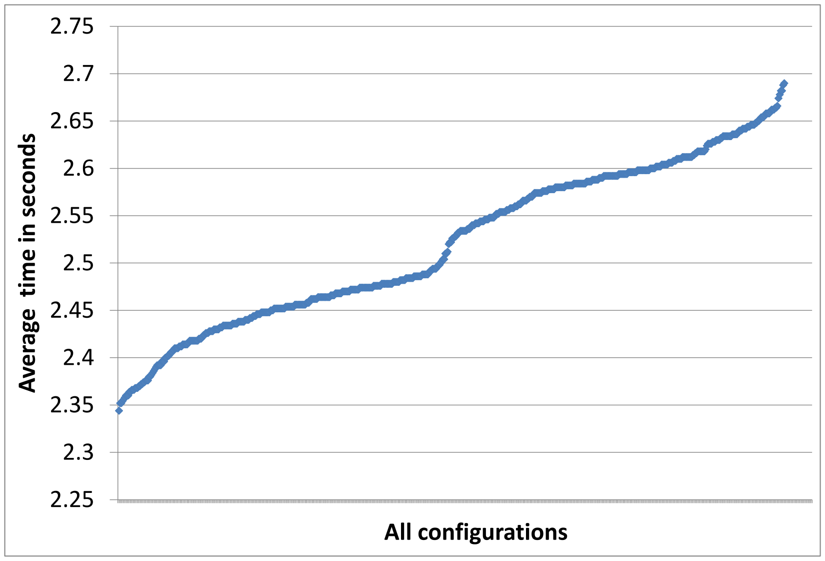


Figure 3.5: FFTW Performance for All Configurations

different configurations. The worst-case performance difference is 14.8% between the best and worst configuration. This result shows that the exact mapping of processes to cores can affect the performance of user-level libraries also significantly. Given that FFT methods have been adopted extensively by many applications, even a small performance degradation of the FFT library can impair the performance of applications to a large extent.

## 3.4 Intelligent Static Mapping of Processes to Cores

The results and analysis developed in previous sections shows the scope for performance improvement if we can design mappings of processes to cores in an intelligent and informed manner. With the backdrop of analysis developed in Section 3.2.3 for the microbenchmarks, in this section, we embark upon the job of identifying the characteristics of different processes of real applications. Towards this goal, we describe factors and techniques for determining intelligent process-core mappings which can deliver better performance. We use this analysis to synthesize optimal process-core mappings for two real-life molecular dynamics simulation applications, GROMACS and LAMMPS, described in Sections 3.4.1 and 3.4.2, respectively. We show that we can achieve noticeable performance improvement with such synthesized mappings versus a random mapping.

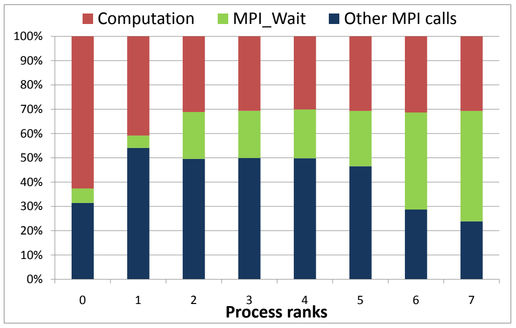
### 3.4.1 GROMACS Application

**Overview:** GROMACS (GROningen MAchine for Chemical Simulations) [11] 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 (ns/day), hence higher values denote better performance. For our measurements, we use the GROMACS LZM application.

**Analysis and Evaluation:** We start by observing that, similar to the FFTW benchmark, several different combinations of process-to-core mappings are possible. Some of these combinations perform worse as compared to the others. To understand the reasoning behind this, we analyze two such combinations (combinations A and B in Table 3.1). We profile the GROMACS LZM application using mpiP [2] and MPE [1] to get statistical analyses of the time spent in different MPI routines. Figure 3.6(a) shows the application time breakdown when running GROMACS with combination A. To simplify our analysis, we show the main components of computation and MPI_Wait, while grouping together all the other MPI calls into a single component. We observe several trends from the graph. First, process 0 (running on core 0) spends a substantial amount of time in computation (more than 60%) while spending minimal amount of time in MPI_Wait. At the same time, processes 6 and 7 spend a large amount of time (more than 40%) waiting. That is, load imbalance occurs in the application.

Table 3.1: Process-Core Mappings Used in GROMACS LZM

| | Machine 1 Process Ranks | | | | Machine 2 Process Ranks | | | |
|---|---|---|---|---|---|---|---|---|
| Mapping | Core 0 | Core 1 | Core 2 | Core 3 | Core 0 | Core 1 | Core 2 | Core 3 |
| A | 0 | 4 | 2 | 6 | 7 | 3 | 5 | 1 |
| A’ | 6 | 4 | 2 | 0 | 7 | 3 | 5 | 1 |
| B | 0 | 2 | 4 | 6 | 5 | 1 | 3 | 7 |
| B’ | 2 | 0 | 4 | 6 | 5 | 1 | 3 | 7 |



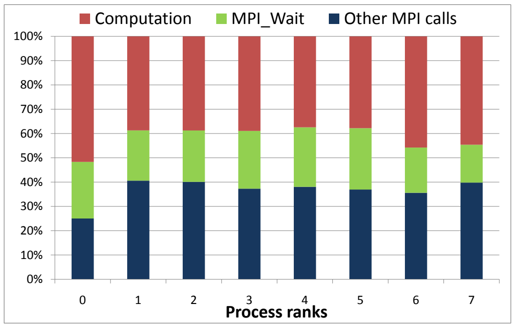


Figure 3.6: GROMACS Application Time Breakdown with TCP/IP: (a) Combination A and (b) Combination A’

To rectify this load imbalance, we swap the core mappings for processes 0 and 6 to form combination A’ (Table 3.1). In this new combination, since process 6 is idle for a long time (in MPI_Wait), we expect the additional interrupts and protocol processing on the core to avoid affecting this process too much. We notice, however, that process 7 has a large idle time in spite of being scheduled on core 0 of the second machine. We attribute this to the inherent load imbalance in the application. Figure 3.6(b) shows the application time breakup with combination A’. We notice that the load imbalance is less in this new combination. Figure 3.7 shows the overall performance of GROMACS with the above process-core mappings. We observe that the performance of the intelligently scheduled combination (A') is nearly 11% better as compared to combination A. The trend is similar for combination B as well.

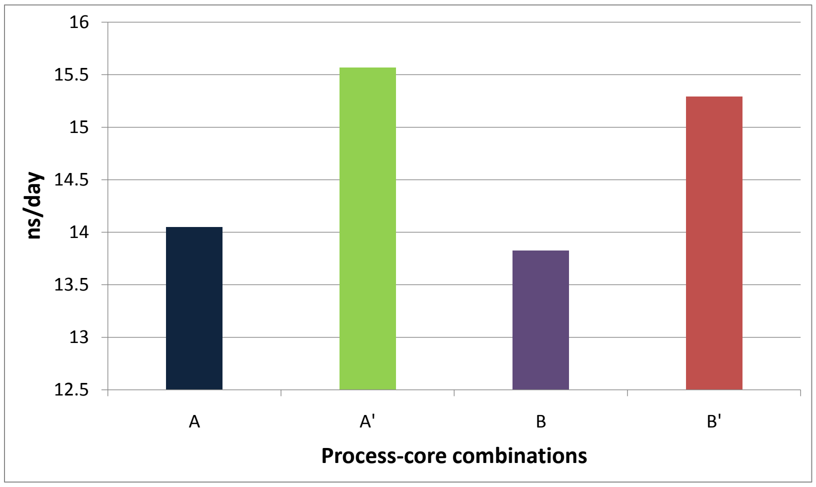


Figure 3.7: GROMACS LZM Protein System Application

This demonstrates that with an intelligent mapping of processes to cores, we can significantly improve the performance of the application when executing on TCP/IP.

### 3.4.2 LAMMPS Application

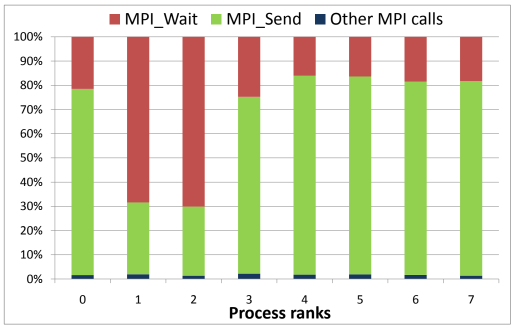
**Overviews:** LAMMPS [33] is a molecular dynamics simulator developed at Sandia National Laboratory. It uses spatial decomposition techniques to partition the simulation domain into small 3D sub-domains, one of which is assigned to each processor. This allows it to run large problems in a scalable way 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.

**Analysis and Evaluation:** We again analyze two difference combinations (Table 3.2). Fig-

Table 3.2: Process-Core Mappings Used in LAMMPS Application

|  | Machine 1 Process Ranks |  |  |  | Machine 2 Process Ranks |  |  |  |
|---|---|---|---|---|---|---|---|---|
| Mapping | Core 0 | Core 1 | Core 2 | Core 3 | Core 0 | Core 1 | Core 2 | Core 3 |
| A | 2 | 0 | 4 | 6 | 1 | 3 | 5 | 7 |
| A' | 0 | 2 | 4 | 6 | 1 | 3 | 5 | 7 |
| B | 0 | 4 | 2 | 6 | 7 | 3 | 5 | 1 |
| B' | 6 | 4 | 2 | 0 | 7 | 3 | 5 | 1 |

ure 3.8(a) illustrates the split-up in the communication time spent by LAMMPS while running on processes-to-core combination A. As shown in the figure, processes 1 and 2 (which run on core 0) spend about 70% of the communication time in MPI_Wait while the other processes spend about 80% of the communication time in MPI_Send. This result is completely counterintuitive as compared to GROMACS: we expect the processes *not* running on core 0 to spend a long time waiting, while processes running on core 0 to perform a lot of computation.



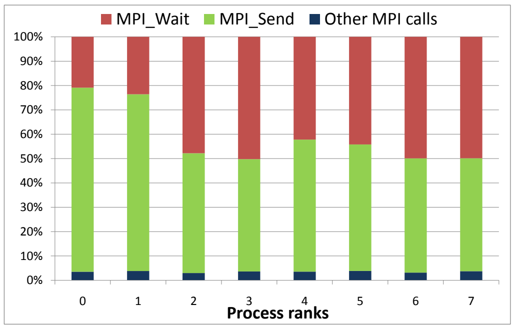


Figure 3.8: LAMMPS Communication Time Breakdown with TCP/IP: (a) Combination A and (b) Combination B

To understand this behavior, we further profile the communication code. We observe that all processes regularly exchange data with only three other processes (Figure 3.9) and that the sizes of the messages exchanged are quite large (around 256 KB). Figure 3.10 illustrates the communication timeline for LAMMPS. As shown in the figure, process X is running on the slower core (which receives most of the interrupts), while process Y is running on a different core. We describe the communication timeline in different steps (broken up in the figure using dotted horizontal lines).

**Step 1:** Initially, both processes post receive buffers using MPI_Irecv() and send data to each other using MPI_Send(). On MPI_Send(), data is copied into a temporary MPI send buffer. As the data is being copied, if space exists in the TCP/IP socket buffer, this data is also handed over to TCP/IP. If not, the data is buffered in the MPI temporary send buffer till more space is created.

**Step 2:** After returning from MPI_Send(), all processes call MPI_Wait() to wait until all the data from their peer processes have been received. While waiting for data to be received, if any data is buffered in the MPI temporary send buffer and has not been sent out yet, MPI attempts to send that out as well. Now, if the receiver is able to read the data fast enough, the TCP/IP socket buffer is emptied quickly, and the sender can hand over all the data to be sent to TCP/IP. If the receiver is not able to read the data fast enough, however, the TCP/IP socket buffer fills up and all the data to be transmitted cannot be handed over to TCP/IP before returning from MPI_Wait(). In our example, since process X is slower, it does not read the incoming data fast enough, thus causing process Y to return from MPI_Wait() without handing over all the data to be sent to TCP/IP.

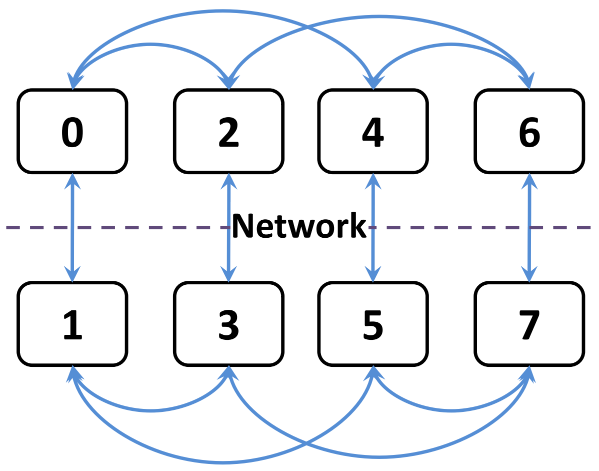


Figure 3.9: LAMMPS Communication Pattern (8 processes)

**Step 3:** Once out of MPI_Wait(), process Y proceeds with its computation. However, since it did not hand over all the data that needs to be transmitted to TCP/IP, some of the data is left untransmitted. Thus, process X cannot return from its MPI_Wait() and has to wait for process Y to flush the data out.

**Step 4:** After completing the computation, when process Y tries to send the next chunk of data, the previous data is flushed out. Process X receives this flushed-out data, returns from MPI_Wait(), and goes ahead with its computation. Now, since process X is not actively receiving data (since it is performing computation), the TCP/IP socket buffer, and eventually process Y's MPI temporary send buffer gets filled up. At this stage, since process Y does not have enough buffer space to copy the application data, it has to wait before returning from MPI_Send().

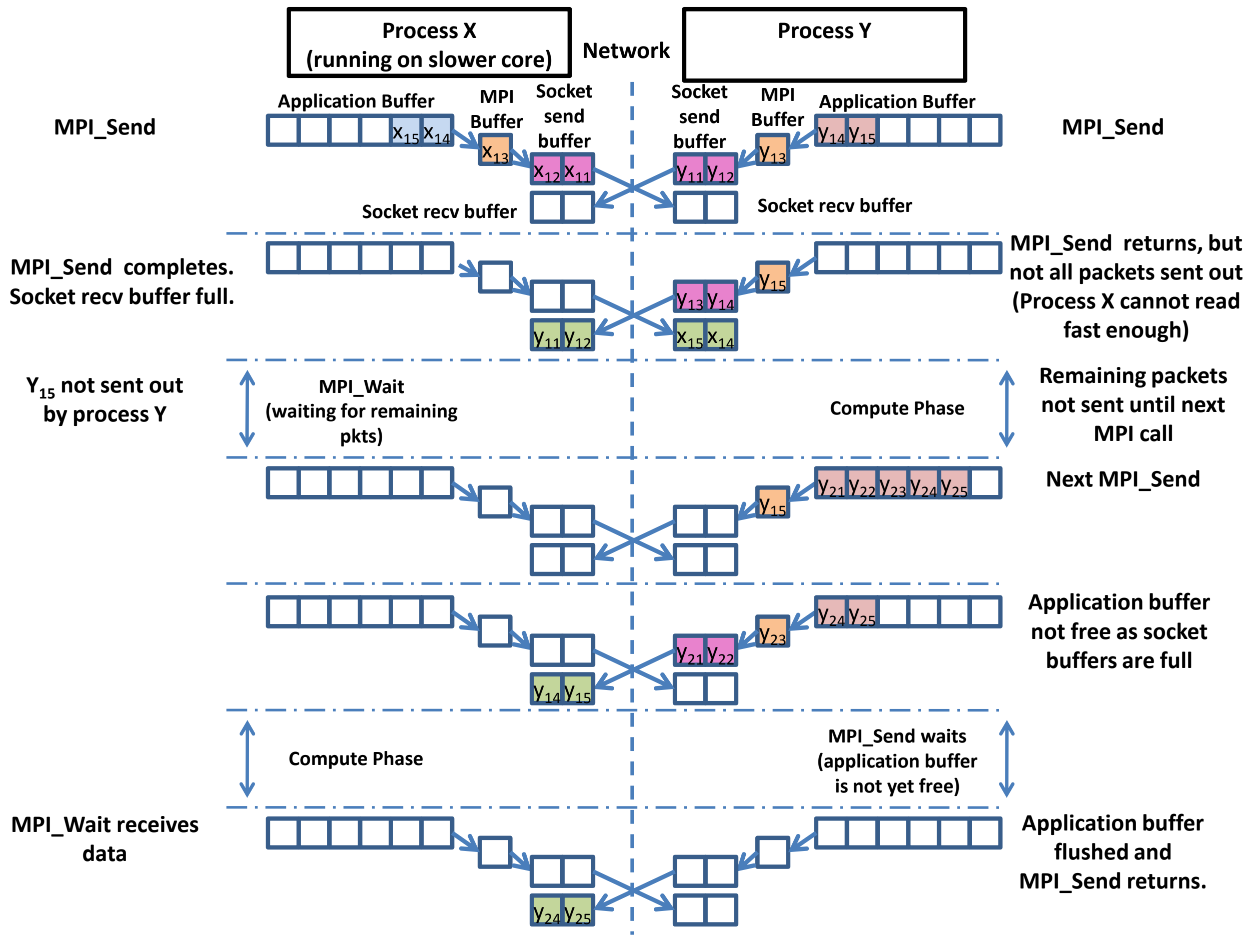

Process X (running on slower core)
Network
Process Y
MPI_Send
Application Buffer
MPI Buffer
Socket send buffer
Socket recv buffer
MPI_Send
MPI_Send completes. Socket recv buffer full.
MPI_Send returns, but not all packets sent out (Process X cannot read fast enough)
Y15 not sent out by process Y
MPI_Wait (waiting for remaining pkts)
Compute Phase
Remaining packets not sent until next MPI call
Next MPI_Send
Application buffer not free as socket buffers are full
Compute Phase
MPI_Send waits (application buffer is not yet free)
MPI_Wait receives data
Application buffer flushed and MPI_Send returns.

Figure 3.10: LAMMPS Timeline

**Step 5:** After process X returns from its computation, when it calls MPI_Wait(), it starts receiving data allowing process Y to complete its MPI_Send().

From the above description, we observe that the processes X and Y are running *out of phase*. That is, when process Y performs computation, X waits in MPI_Wait, and when X performs computation, process Y waits in MPI_Send. This out-of-phase behavior causes unnecessary waits, resulting in loss of application communication performance. We note that this behavior happens because the *effective capability* of the cores on which run processes X and Y execute do not match. To rectify this situation, we need only ensure that the cores that execute processes X and Y match in capability.

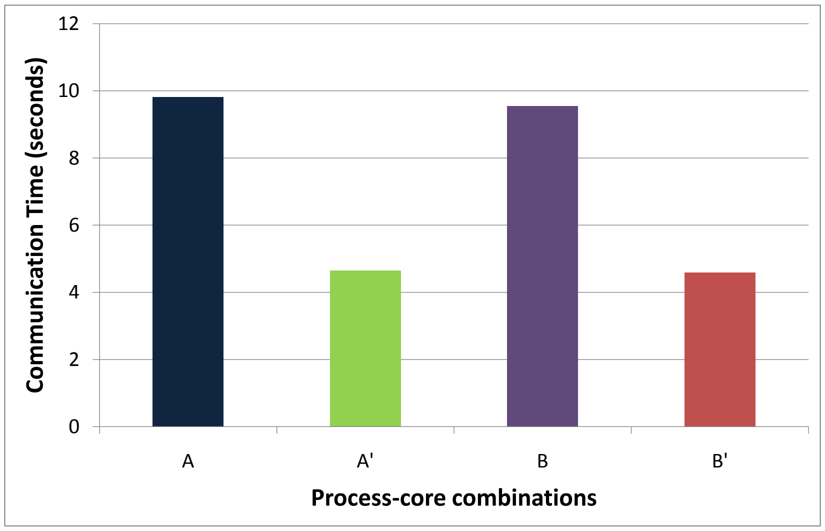


Figure 3.11: LAMMPS Performance

In Table 3.2, for combination A, we see that swapping processes 0 and 2 gives us the desired effect (note that each process communicates with only one process outside its node). Figure 3.8(b) demonstrates that this new intelligent combination can reduce the imbalance to a large extent.

Figure 3.11 shows the communication performance of LAMMPS with the above core mappings. We observe more than two-fold performance difference between combinations A and A' as well as combinations B and B'.

In summary, with both GROMACS and LAMMPS, we identified application characteristics that caused pronounced interaction between the communication stack and the architecture. We formulated intelligent process-core mapping configurations for these applications, which were able to substantially improve application performance.

# Chapter 4

# The SIMMer Framework

In this chapter, we formalize the intelligent mapping strategies developed in the previous chapter and implement them in a framework that monitors and manages the interactions. Based on the processing and cache transaction impacts which we discussed in Section 2.3, we expand them into *symptoms* of interaction perceivable in application behavior. We describe these symptoms in Section 4.1. In section 4.2, we present our approach to mitigate such effects and the design of the SIMMer framework. We perform a thorough evaluation of SIMMer with microbenchmarks and applications in Section 4.3. We conclude this chapter with a discussion on alternative multicore architectures and the relevance of SIMMer in such architectures.

## 4.1 Identifying the Symptoms of Protocol Stack and Multicore Architecture Interaction

Directly understanding the actual interactions between the communication protocol stack and the multicore architecture is complicated and requires detailed monitoring of various aspects of the kernel and hardware as well as correlation between the various events. Instead, we take an indirect approach to understand the 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 a certain interaction can result in a symptom. However, the occurrence of the symptom does not mean that the interaction has taken place. That is, each symptom can have a number of *causes* that could have triggered it.

In this section, we discuss the various symptoms that we have to identify in order to catch the different interactions. In Section 4.2, we describe our approach to monitor these symptoms and minimize the impact of the interactions through appropriate metrics.

***Symptom 1 (Communication Idleness):*** As noted in Section 2.3, if a core is busy performing TCP/IP protocol processing, the number of compute cycles it can allocate to the application process is lower, thus slowing down the process allocated to this core. So, 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 due to a number of reasons including native imbalance in the application communication model itself.

***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. In this case, process A would have to buffer the data since process B cannot receive it as fast as it is sent out by process A. Thus, process A *attempts* to send the data and goes off 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 sends it out. During the time when process A was computing, process B just waits to receive the data. Now, when the data is flushed out, process B receives it and goes off to perform its computation, when process A is still waiting to receive its data. This behavior is caused because though processes A and B are performing similar tasks, they are slightly *out-of-sync* because of the difference in their effective computational capabilities, resulting in large wait times.

***Symptom 3 (Cache Locality):*** As mentioned in Section 2.3, when a core performs TCP/IP protocol processing, it fetches the data to its cache in order to perform the checksum data integrity. Thus, if one process is actively working on the data that is being communicated, while other processes are not (suppose they are working on some other data), the first process would likely notice more cache misses than the rest. That is, the data needed by this process is not locally available and has to be fetched from another core's cache.

## 4.2 Intelligent Process-to-Core Mapping with SIMMer

This section describes our intelligent Systems Interaction Mapping Manager (SIMMer) framework and its associated monitoring metrics.

### 4.2.1 The SIMMer Framework

The SIMMer library (Figure 4.1) is an interactive monitoring, sharing, and analysis framework that can be tied into existing communication middleware such as MPI. The framework itself deals with the monitoring, sharing, and analysis components while the actual metrics that are used for the decision making are separately pluggable, as we will describe in Section 4.2.3.

**Interaction Monitoring:** The interaction monitoring module is responsible for monitoring the different components within the system. This includes system-specific information (hardware interrupts, software signals), communication middleware-specific functionality (data buffering time and other internal stack overheads) and processor performance counters (cache misses). The monitoring module utilizes existing libraries for performing some of the instrumentation, while relying on in-built functionality for the rest. For example, processor performance counters are measured

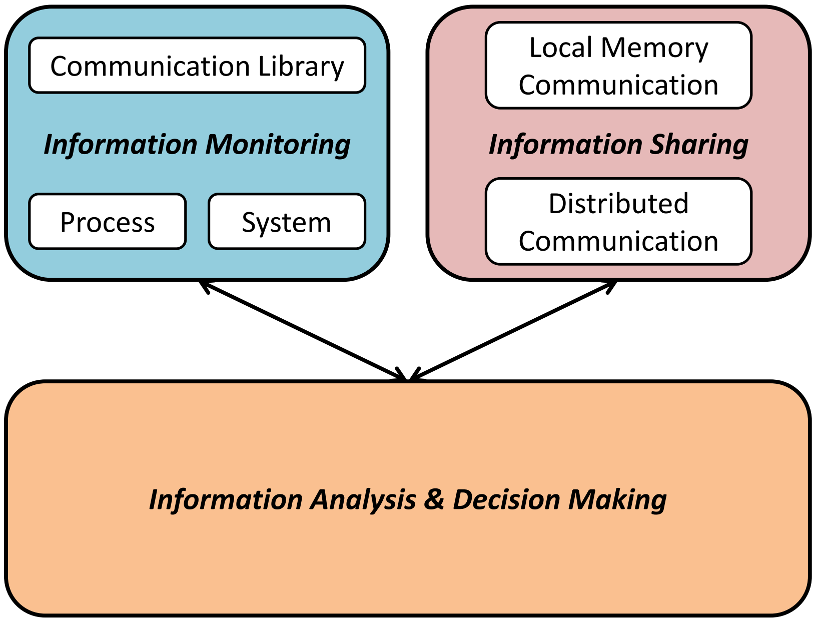

Communication Library
Information Monitoring
Process
System
Local Memory Communication
Information Sharing
Distributed Communication
Information Analysis & Decision Making

Figure 4.1: The SIMMer Component Architecture

using the Performance Application Programming Interface (PAPI [3]) and system-specific information through the *proc* file-system. Communication library-specific information can be monitored through specific profiling libraries if available. For example, MPI-specific information can be monitored through libraries such as PERUSE [4], but several of the current MPI implementations do not support them yet. Thus, we utilize in-built profiling functionality for such information.

In order to minimize monitoring overhead, the monitoring module dynamically enables only portions of monitoring that are required for the metrics being used. For example, if no plugged in metric requires processor performance counters, such information is not monitored to reduce the overhead.

**Information Sharing:** The information sharing module is responsible for the exchange of state or data between different processes in the application. Several forms of information sharing 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 header itself. Whenever a packet is sent, the sender adds the information that needs to be shared within the header. The receiver upon receiving the header shares this information 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 completely using shared memory without requiring locks or communication blocking of any kind. This provides a great deal more flexibility and reduces the communication overhead of our framework significantly.

**Information Analysis:** The monitored interaction information collected by each process and shared with other processes is *raw* in the sense that no correlation exists between the different pieces of information by comparing locally monitored events with those from other processes. Further, the monitored information is low-level data that needs to be summarized to more abstract information before they can be used by the different metrics. The *information analysis* module performs such analysis and summarization of the monitored information. This module also allows for prioritization between the different plugged in metrics for cases where an application shows multiple symptoms whose independent analyses often times conflict with each other. Finally, each monitoring event has a certain degree of inaccuracy associated with it. Thus, some monitoring events have more *data noise* than others. To handle such issues, the analysis module allows different monitors to define *confidence levels* for their monitored data through environment variables. Thus, depending on the number of events that are received, the analysis module can accept or discard different events based on their confidence levels, using appropriate thresholds.

### 4.2.2 Current Implementation

SIMMer is a generic framework that can be implemented for any architecture or communication library. Referring to Figure 4.1, the Information Monitoring component is the only component that is specific to the system or middleware. This is because the information that is monitored can be exclusive to a communication library or architecture. Given that MPI is the *de facto* parallel programming model for a majority of scientific applications in high-performance computing, we focus our implementation on the MPI programming model on the Intel and AMD multicore architectures. Specifically, we implement SIMMer within MPICH2 [28] because of prior expertise with MPICH2 and also because it is open source. For the rest of this chapter, SIMMer refers to our implementation on the MPICH2 communication middleware.

### 4.2.3 Metrics for Mapping Decisions

In this section, we discuss different metrics that can be plugged into the SIMMer library. Specifically, we focus on metrics that address the symptoms noted in Section 4.1.

**Communication Idleness Metric:** This metric is defined based on the *communication idleness* symptom defined in Section 4.1. 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 communication library monitoring capability of the SIMMer 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.

Therefore, 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. Comparison of this idleness factor between different processes provides the idea of which processes are more suited for sharing the protocol processing overhead. So, the idleness metric needs to be compared only for processes running on the same node, and this metric only uses the intra-node communication channel as discussed in Section 4.2.1.

**Out-of-sync Communication Metric:** The out-of-sync metric captures the time of computation performed with unsent data in each process' internal MPI buffers, and compares that with the wait time of other processes. As described in Section 4.1, this metric represents the case where unsent data followed by a long compute phase results in high wait times on the receive process. Whenever the computation time (with buffered data) is above a user-defined threshold, a message is sent to all processes informing them of this symptom. A similar message is sent whenever the wait time of a process is above a user-defined threshold. If communicating peer processes observe these conditions on either side, a leader process is dynamically chosen, which then analyzes the data and intelligently decides on the best mapping possible. It then informs the new mapping to all processes concerned.

**Cache Locality Metric:** The cache locality metric utilizes the L2 cache misses monitored by the SIMMer library. This metric is specific to the processor architecture and relies on the locality of cache between different core pairs. If the number of cache misses for a process is more than the user-defined confidence level while that of another process is smaller, then these two processes can be swapped as long as the communicating process moves *closer* to the core performing the protocol stack processing. 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.

## 4.3 Experimental Evaluation

In this section, we evaluate our proposed approach with multiple microbenchmarks in Section 4.3.1 and the GROMACS and LAMMPS applications and FFTW Fourier Transform library in Section 4.3.2. The platform we used for our evaluation is the same Intel cluster that we described and evaluated in Section 3.1. Each microbenchmark possesses 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. Performance results of all microbenchmarks and applications are presented as a comparison of the MPI library without SIMMer (henceforth called vanilla MPICH2) and the SIMMer-enabled MPI library.

### 4.3.1 Microbenchmark Evaluation

We designed three microbenchmarks to illustrate the impact of each of the symptoms described in Section 4.1 and to represent common processing patterns seen in real-world applications. These microbenchmarks are then used to demonstrate the benefits achievable by the SIMMer library in these cases. We then evaluate the SIMMer library with real applications and scientific libraries to show its impact in the *real world*.

#### 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 4.2 shows the performance of the communication idleness benchmark using SIMMer-enabled MPICH2 as compared to vanilla MPICH2. We define the 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

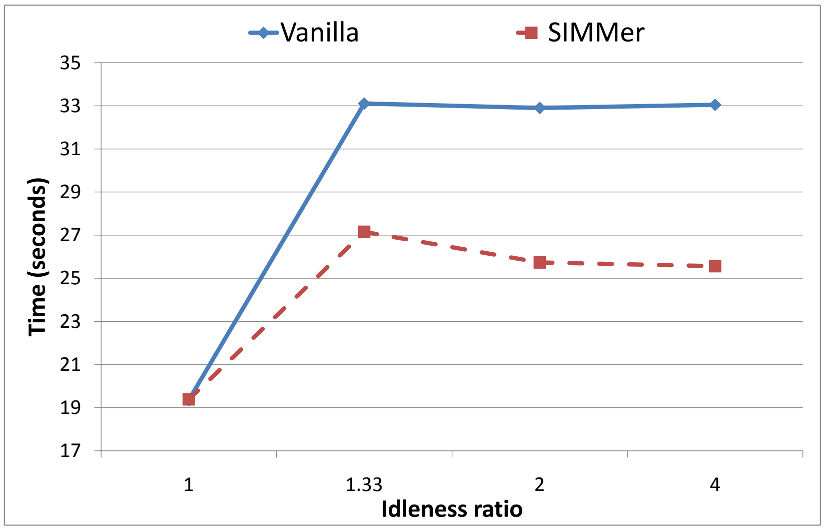
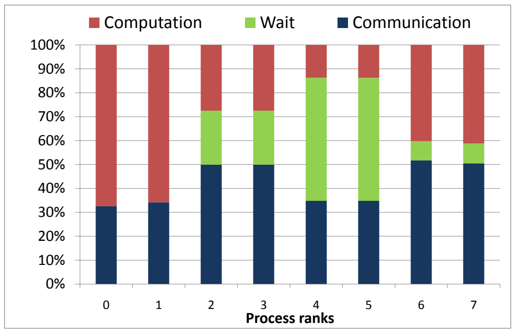


Figure 4.2: Communication Idleness Benchmark Performance

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 4.2, we plot the time taken for the benchmark to execute against various idleness ratios. We observe that both vanilla and SIMMer-enabled MPICH2 have the same performance when the idleness ratio is 1. This is expected given that an idleness ratio of 1 represents a completely symmetric benchmark with no irregularity in computation or communication. Thus there exists no scope for SIMMer to improve performance in this case. But we observe that as the idleness ratio increases with vanilla MPICH2, the performance of the benchmark heavily depends on the mapping of processes to cores. That dependence makes it possible for SIMMer to use the disparity in computation to achieve a more optimal arrangement, reducing runtime by nearly 40%.

To further analyze the behavior of the benchmark, we show the distribution of time spent in the different parts of communication in Figures 4.3(a) and 4.3(b) for vanilla MPICH2 and SIMMer-enabled MPICH2 respectively. For consistency, both are based on the same initial mapping of



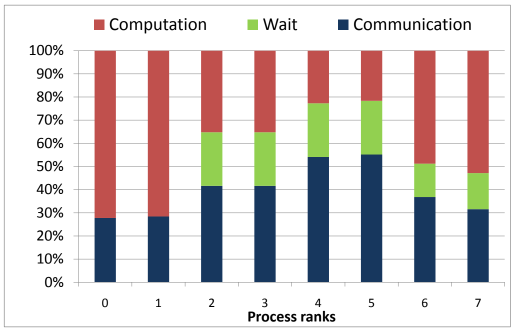


Figure 4.3: Communication Idleness Benchmark Division of Time: (a) Vanilla MPICH2 and (b) SIMMer-enabled MPICH2

processes to cores with an idleness ratio of 4. Figure 4.3(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 application computation as well as protocol processing overhead. Figure 4.3(b), on the other hand, illustrates the distribution for SIMMer-enabled MPICH2. SIMMer keeps the wait times more even across the varied processes, thus reducing the overhead seen by the application.

**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, 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.

Figure 4.4 shows the performance of the out-of-sync communication benchmark using vanilla

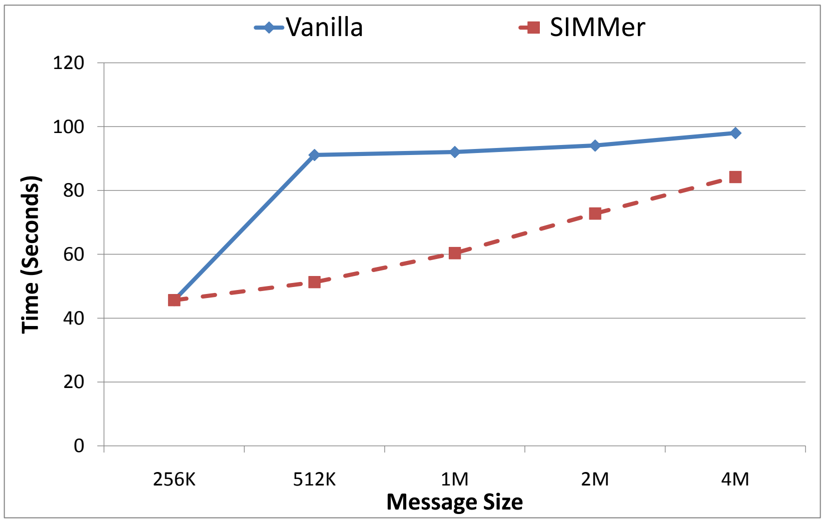


Figure 4.4: Out-of-Sync Communication Benchmark Performance

MPICH2 and SIMMer-enabled MPICH2, by comparing the total time in seconds to execute the benchmark with various message size used in the communication step. Similar to the communication idleness benchmark, SIMMer-enabled MPICH2 performs as well as or better than vanilla MPICH2 in all cases. At message sizes up to 256 KB, both vanilla and SIMMer 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 with no scope for performance improvement. But for message sizes above 512 KB, we observe that SIMMer-enabled MPICH2 consistently outperforms vanilla MPICH2 by up to 80%.

Figure 4.5 analyzes the number of times data is buffered within the MPI library running under the same initial conditions with and without SIMMer. As shown in the figure, the data buffering time is almost an order-of-magnitude less when using SIMMer. When an out-of-sync message occurs with SIMMer, the time taken is the same, but because SIMMer corrects the error in synchronization, the out-of-sync behavior does not happen further times. This ultimately results in the

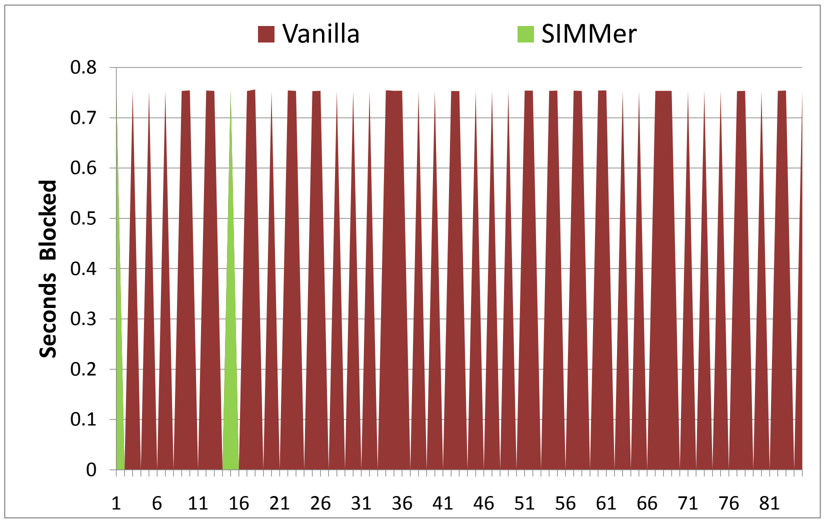


Figure 4.5: MPI Data Buffering Time with the Out-of-Sync Communication Benchmark

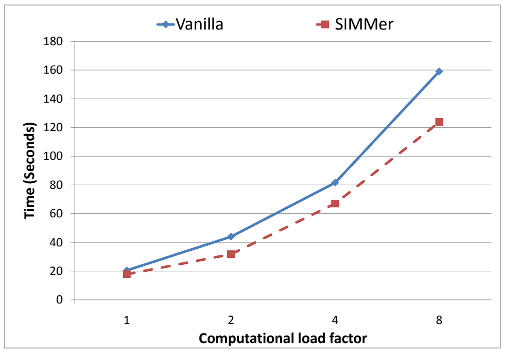
performance improvement demonstrated in Figure 4.4.

### 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. Depending on which core the communicating processes are mapped to, they may present the cache locality symptom discussed in Section 4.1, which lets us know that the performance of the application is impacted. 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 taken to complete a certain number of iterations of the benchmark. This is portrayed in Figure 4.6(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, SIMMer outperforms vanilla MPICH2 by as much as 29%.

We profile the benchmark and count the number of L2 cache misses observed by each process to gain a further understanding of SIMMer's behavior. Figure 4.6(b) shows the total number of cache misses observed by the processes performing inter-node and intra-node communication, respectively. For inter-node communication, we observe that the number of cache misses decreases significantly when using SIMMer-enabled MPICH2 despite the migration and other overhead incurred in the process. The number of L2 cache misses fall in the case of intra-node communication as well. This is because the intra-node communicating processes gain two benefits from SIMMer that we did not initially anticipate. Firstly, their locality with respect to the local processes that 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 SIMMer not only improves the cache locality of the inter-node communicating processes, but it also improves that of the intra-node communicating processes.



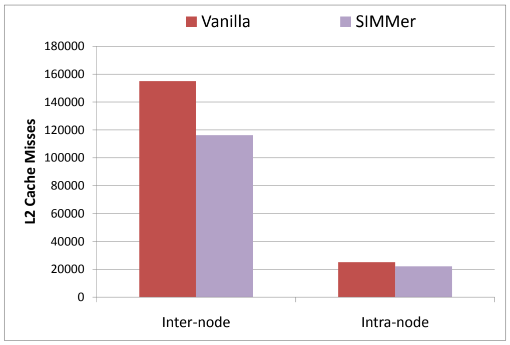
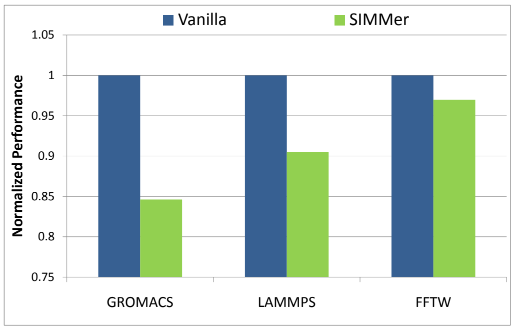


Figure 4.6: Cache Locality Benchmark: (a) Performance and (b) Cache Miss Analysis

### 4.3.2 Evaluating Applications and Scientific Libraries

Here 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 SIMMer. For an explanation of the applications, refer to Section 3.4.



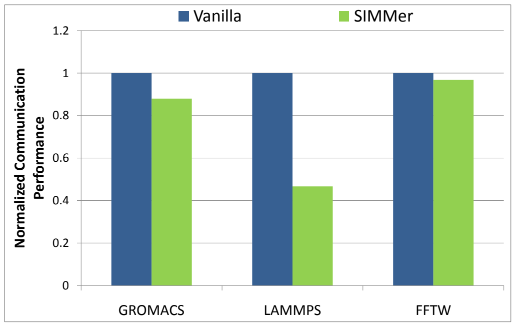


Figure 4.7: Normalized Application Performance: (a) Overall Execution Time and (b) Communication Time

Figure 4.7(a) illustrates the normalized overall execution time of SIMMer-enabled MPICH2 for GROMACS, LAMMPS and FFTW, as compared to vanilla MPICH2. SIMMer-enabled MPICH2 can remap processes to the right cores so as to maximize performance resulting in performance improvement across the board. The figure shows an 18% performance improvement with GROMACS when using SIMMer. With LAMMPS, the overall performance improvement is about 10%. For FFTW, we 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 SIMMer monitors for interactions between the communication protocol stack and the multicore architectures, small data volumes mean that such interaction would be small as well.

To better understand the benefits achieved by SIMMer, we profile the applications to measure the time spent in communication by these applications. This comparison would provide us a better understanding of the performance improvement in the communication stack that SIMMer can achieve. Figure 4.7(b) shows the normalized communication time of SIMMer-enabled MPICH2 for GROMACS, LAMMPS and FFTW, as compared to vanilla MPICH2. SIMMer achieves more than a *two-fold* improvement in communication performance with LAMMPS. For GROMACS, the improvement is about 13%, while for FFTW, it is more than 3%.

In summary, we see a noticeable improvement in both overall performance and communication performance with SIMMer-enabled MPICH2 for all three applications. This shows that intelligent process-to-core mapping is a promising approach to minimize the impact of interactions of protocol stacks with the multicore architecture, and in turn, to improve application performance significantly in some cases.

### 4.3.3 Evaluation of NAS Parallel Benchmarks

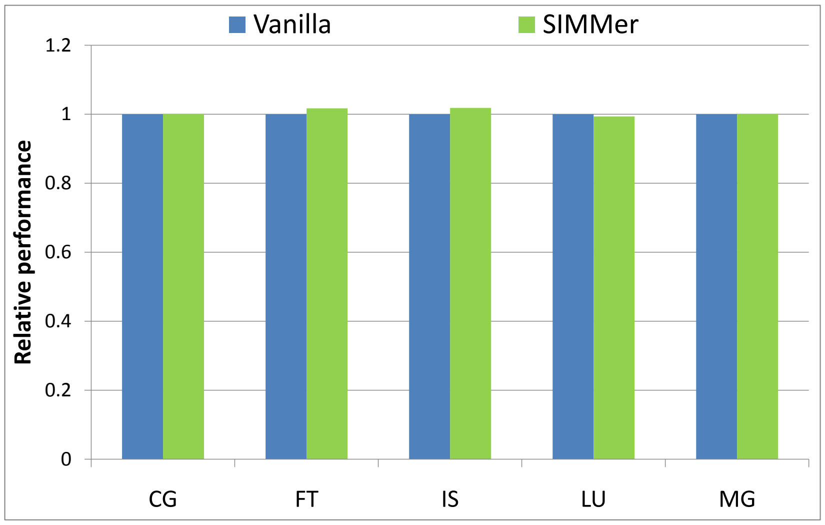


Figure 4.8: Performance Evaluation of NAS Parallel Benchmarks

Next we evaluate the performance of the NAS Parallel Benchmarks [8] with SIMMer and motivate the need for a broader investigation of interactions of high-performance networks with multicore architectures. Figure 4.8 shows the relative performance of SIMMer-enabled MPICH2 for various NAS Parallel Benchmarks (Class B) relative to vanilla MPICH2. No significant performance difference is observed when using SIMMer with the NAS Parallel Benchmarks. We believe this is because of the absence of any significant asymmetric interactions between application processing and multicore architectures within a node. This leads us to perform a broader investigation of interactions across nodes by analyzing the impact of network resource contention. This analysis is presented in Chapter 5.

## 4.4 Discussion on Alternative Multicore Architectures

While we demonstrated our approach for the Intel multicore architecture, the idea of intelligently mapping application processes to the different cores in the 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 are highly sensitive to the placement of processes on the different computational cores and would likely benefit from a process-to-core mapping manager library such as SIMMer.

While AMD-based multicore architectures are quite similar to Intel-based multicore architectures, a few key differences need to be addressed before they can utilize SIMMer-like process management libraries to improve performance. Specifically, the non-uniform memory access (NUMA) model makes process management more complicated for AMD systems as compared to Intel systems. For example, on an Intel system, SIMMer could freely move any process to any core in the system. 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 core that does not sit on the same die, all of its memory accesses will be remote, thus adding a significant performance overhead.

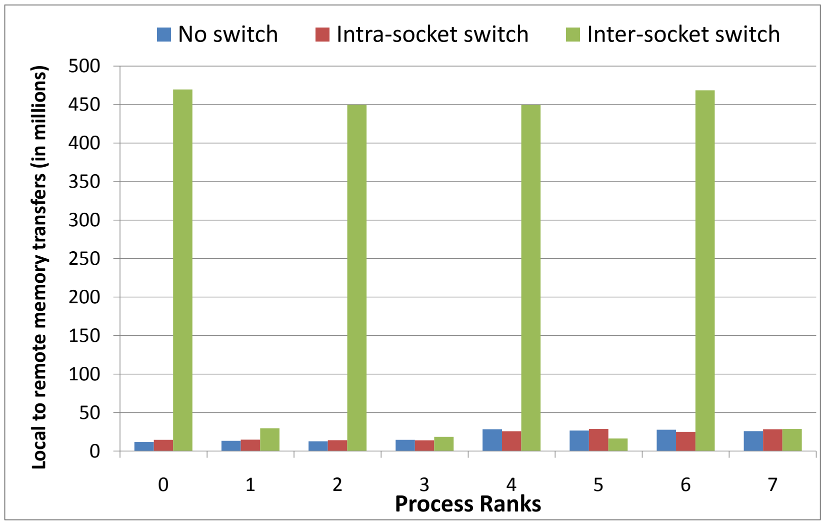


Figure 4.9: Local to Remote Memory Transfers for Various Types of Process Switches

To analyze this, we profile the LAMMPS application with PAPI to count various memory access performance counters in our AMD cluster. Figure 4.9 shows the number of local to remote memory transfers for process switches within the same processor and between processors. We induce these types of switches in SIMMer in one of the nodes (the one running processes 0, 2, 4 and 6) by running the application with the relevant process-core mapping. We also compare it to the baseline case where no switch is performed. The graph shows a 31-fold increase in non-local memory transfers with an inter-socket process switch. This confirms that the NUMA model can cause side effects that can nullify the potential performance improvement gained by SIMMer.

While the SIMMer-enabled MPICH2 can still be applied to AMD systems, aspects such as page migration between different memory controllers need to be addressed from an implementation perspective to achieve good performance. In its absence, SIMMer would be restricted to process mapping only to cores within a die, which would potentially result in only limited improvement in performance.

Finally, with the upcoming network-on-chip (NoC) architectures such as Intel XScale and advanced architectures such as the Intel Terascale and Tilera chips, intelligent process-to-core mapping will become especially important because of the huge number of cores that will reside on the same die. For example, the Intel Terascale chip would accommodate 80 cores on the same die, while a Tilera chip accommodates 64 cores on the same die. In such cases, an incorrect assignment of 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. Intelligent process-to-core mapping, on the other hand, can dynamically move towards for the *right* assignments and use them to achieve the best performance.

# Chapter 5

# Network and Processor Resource Contention in Multicore Architectures

In this chapter, we investigate the interaction of high-performance network protocol stacks with multicore architectures from a different perspective by analyzing the impact of network resource contention on application performance. We analyze this impact with three approaches. First we look at the effect on performance due to sharing of network resources. Next, we perform a complementary analysis of contention of processor resources. Finally, we look at how changes in process allocation schemes can affect the amount of network contention.

We choose the Myrinet network and the MX protocol for our study and use the NAS Parallel Benchmark suite to evaluate performance. We present an overview of the Myrinet network in Section 5.1. Sections 5.2 and 5.3 explain the experimental setup and the various configurations in which we run our experiments. The analysis of network and processor contention are presented in Sections 5.4 and 5.5, respectively. In Section 5.6, we show the analysis with different process allocation schemes.

## 5.1 Overview of Myrinet Network

Myri-10G [31], the latest generation Myrinet developed by Myricom, is a low-latency wormhole-routed high-speed interconnect that supports link speeds of 10 Gbps. The Myrinet network interface card (NIC) has a user-programmable processor and DMA engines that ease the design and customization of software communication stacks. MX (Myrinet Express) is a high-performance, low-level, message-passing software interface tailored for Myrinet, which exploits the processing capabilities embedded in the Myrinet NIC. The Myri-10G NICs, switches, and the associated software support both Ethernet and Myrinet protocols at the link level. Users can run applications using MX-10G over Myrinet (MOM) with Myrinet switches or using MX-10G over Ethernet (MODE) with 10-Gigabit Ethernet switches. The basic MX-10G communication primitives are non-blocking send and receive operations.

Our network consists of the Myri-10G NICs connected by a 24-port Myrinet switch. The NICs are connected to the host via a 133-MHz/64-bit PCI-X bus. They have a programmable LANai processor running at 300-MHz with 2-MB onboard SRAM memory.

## 5.2 Experimental Setup

To study the impact of network resource contention, we used a bigger setup than used in previous sections. Our setup consisted of a 16-node cluster, of which each node is a custom-built, dual-processor, dual-core AMD Opteron 2.6-GHz system with 4 GB of DDR2 667-MHz SDRAM. The four cores in each system are organized as cores 0 and 1 on processor 0 and cores 2 and 3 on processor 1. Each core has a separate 1-MB L2 cache. All machines run Ubuntu Fiesty with kernel version 2.6.19 and are equipped with Myri-10G network interface cards connected to a Myrinet switch. The MPI library used is MPICH2-MX v1.0.6. All experiments were run at least three times with the processor affinity of each process set to a fixed core to remove the impact of operating system scheduling anomalies.

We use version 3.2.1 of the NAS benchmark suite. The results shown are for class B of the NAS benchmarks, but we achieved similar results for classes A and C.

## 5.3 Configurations Used in Experiments

This section describes the configurations on which we ran our experiments. We use 16 processes for all the NPB benchmarks because this covers the maximum number of benchmarks and configurations for our setup. We note that 16 processes can be run on different configurations on a multicore architecture with four cores. Picking only those with constant number of processes on a node, we end up with three configurations:

- 16X1 – 16 nodes, one process on one of the four cores
- 8X2 – 8 nodes, 2 processes, on two of the four cores
- 4X4 – 4 nodes, 4 processes, one on each core

We observe that between each of the three configurations have increased levels of network contention. With 16X1, no network contention is present since each node runs only one application process. With 8X2, however, two processes in each node use the same network interface card. Hence, the network contention with 8X2 is two times more than with the 16X1 case. With 4X4, four processes use the same NIC, thus making the network contention four times greater than with the 16X1 case. In our experiments, we ran the 4X4 configuration with cyclic allocation of processes between nodes.

To consider the effects of processor contention, we split the 8X2 into two cases again. Our setup consists of a dual-core dual-processor system, and hence, the two processes can be run in two different modes:

- 8X2 co-processor mode – two processes, each running on a different processor
- 8X2 virtual processor mode – two processes, both run on the same processor

We note that in the virtual processor mode, increased contention of processor resources is present because both processes are run on the same processor.

We now briefly describe the various benchmarks in the NPB suite:

**BT and SP:** The NAS BT and SP benchmarks are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite difference discretization of three-dimensional Navier-Stokes equations [8]. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme. SP and BT use a skewed-cyclic block distribution known as multipartitioning in which each processor is responsible for several disjoint sub-blocks of points (cells) of the grid. In both BT and SP, the granularity of communications is kept large, and fewer messages are sent [8].

**CG:** The CG kernel benchmark solves an unstructured sparse linear system by the conjugate gradient method. It uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzero values. The MPI CG code accepts a power of two number of processors that are mapped onto a grid of row by column processors and tests irregular long distance communication.

**FT:** The FT benchmark solves a Poisson partial differential equation using a 3-D discrete fourier transformation. The processes are arranged in a 1-D grid, and the global array is distributed along its last dimension. The forward 3-D FFT is then performed as multiple 1-D FFTs in each dimension, first in the $x$ and $y$ dimensions, which can be done entirely within a single processor, with no interprocessor communication. An array transposition is then performed, which amounts to an all-to-all exchange, wherein each processor must send parts of its data to every other processor. The final set of 1-D FFTs is then performed [8].

**IS:** This benchmark performs a sorting operation based on bucket sort. It performs many all-to-all exchange communication.

**LU:** LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block lower and upper triangular systems. Communication of partition boundary data occurs after the completion of computation on all diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method and is called a wavefront method by its authors [31]. The LU benchmark is very sensitive to the small messages and is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages.

**MG:** The MG benchmark uses a V-cycle multigrid method to compute the solution of the 3-D scalar Poisson equation. It performs both short- and long-range communications that are highly structured.

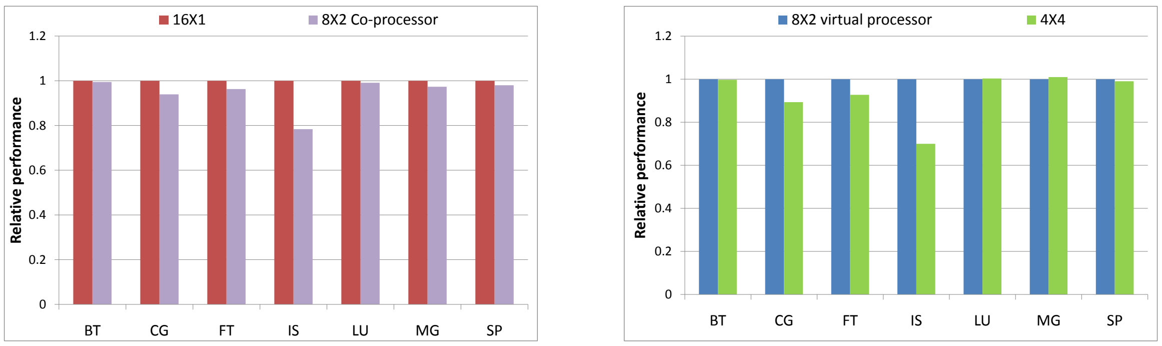


Figure 5.1: Evaluation of Network Contention: (a) 16X1 vs 8X2 Co-Processor and (b) 8X2 Virtual Processor vs 4X4

## 5.4 Impact of Network Contention

We evaluate the impact of network contention by running the NPB benchmarks over each of the three configurations described in Section 5.3. Figure 5.1 shows the impact network resource contention can have on performance of applications. The figures show the normalized performance of the NAS benchmarks in the three aforementioned configurations. In Figure 5.1(a), as we move from 16X1 to 8X2 co-processor mode, the performance of all the benchmarks drops (as much as 27% for IS). The reason is the increased network contention in the 8X2 configuration, where two processes have to share the network resources. Since only one process has been added to every node, the chances that a process will communicate predominantly with the process collocated in its node are slim.

In Figure 5.1(b) on the other hand, the performance drop is seen mainly for CG, FT, and IS, while the other benchmarks perform similarly or show improved performance (in the case of MG and LU) between the two configurations. Here we see mixed benefit in moving to 4X4 because more number of processes are collocated in the same node. Thus, potentially, more shared memory communication can happen reducing the possibility of network contention.

Table 5.1: Normalized Total Network Communication Time for 16X1 and 8X2 Co-Processor

| | ***Communication time (seconds)*** | | |
|---|---|---|---|
| ***NAS Benchmark*** | ***16X1*** | ***8X2 co-processor*** | ***Percentage increase*** |
| *BT* | 0.090 | 0.098 | 7.94 % |
| *CG* | 1.319 | 1.757 | 33.12 % |
| *FT* | 0.051 | 0.056 | 9.56 % |
| *IS* | 0.015 | 0.023 | 56.49 % |
| *LU* | 0.721 | 0.772 | 6.97 % |
| *MG* | 0.098 | 0.127 | 30.00 % |
| *SP* | 0.165 | 0.197 | 19.38 % |

To analyze the level of network contention in the above results, we profile the network communication time in each of these configurations. Since we are using Myrinet's MX protocol, we profile the time spent in the `mx_isend()` and `mx_test()` calls. For small messages, the time spent in `mx_isend()` and `mx_test()` is towards copying the message data from application memory to the NIC memory, queueing the message request and waiting for the message to be sent out. This time represents the time spent by the network in sending the data out and thus is an indicator of the overhead of network contention.

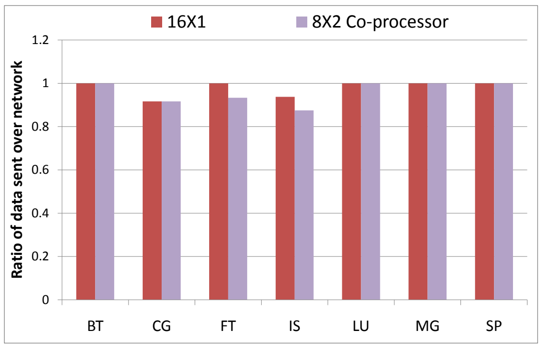
Table 5.1 shows the normalized total time spent in `mx_isend()` and `mx_test()` calls for the various configurations and the percentage difference between them. We observe an increase in the network communication time for all the benchmarks between 16X1 and 8X2 co-processor mode. In other words, moving to the 8X2 co-processor mode results in more time being spent for network communication because the network resources are being shared. Also, the amount of intra-node communication remains comparatively low, hence making it difficult to observe any significant benefit from the reduced latency. Of 15 other possible processes with which a process can communicate, only one results in intra-node communication, which means a 93% chance that a process will communicate over the network with another process. These results mimic the performance results where all benchmarks observe a decrease in performance when moving to 8X2 co-processor mode.

In Table 5.2, however, the network communication increases only for the CG, FT, and IS benchmarks, while for all others it drops. This again clearly mimics the performance results as seen in Figure 5.1(b). In this case, moving from the 8X2 virtual processor mode to 4X4 mode results in two processes getting added to the same node. This denotes an increased capability to perform intra-node communication. Compared to a 93% chance of network communication with the 8X2 case, the chances that a process will communicate over the network with another process in the 4X4 case is 80%.

We analyze our results further by profiling the amount of data sent over the network as compared to intra-node communication for all the benchmarks. Figure 5.2(a) shows the ratio of data sent over the network for 16X1 and 8X2 co-processor modes for all the benchmarks. Except for FT and IS, where the amount of network communication drops slightly (6.7%), all the benchmarks have the same amount of network communication as compared to 16X1. Figure 5.2(b) shows the same graph for 8X2 virtual processor mode and 4X4 configurations. Here we observe that BT, LU, MG, and SP experience drops in network data communicated (up to 50% in the case of LU and MG), while CG, FT, and IS show very low reductions in the amount of network data communicated. In fact, CG does not observe any drop in network communication when moving from 16X1 to 8X2 to 4X4. This result also exactly mimics the network communication time results we observed in Table 5.1 and corroborates the performance results we get.

Table 5.2: Normalized Total Network Communication Time for 8X2 Virtual Processor and 4X4

| | ***Communication time (seconds)*** | | |
|---|---|---|---|
| ***NAS Benchmark*** | ***8X2 virtual processor*** | ***4X4*** | ***Percentage increase*** |
| *BT* | 0.110 | 0.083 | -24.68 % |
| *CG* | 1.804 | 2.716 | 50.54 % |
| *FT* | 0.058 | 0.063 | 8.32 % |
| *IS* | 0.024 | 0.066 | 172.70 % |
| *LU* | 0.803 | 0.395 | -50.83 % |
| *MG* | 0.185 | 0.144 | -22.24 % |
| *SP* | 0.223 | 0.198 | -11.40 % |



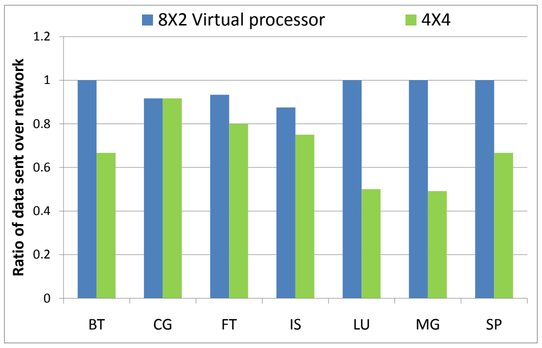


Figure 5.2: Network Communication Data Size: (a) 16X1 vs 8X2 Co-Processor and (b) 8X2 Virtual Processor vs 4X4

## 5.5 Impact of Processor Contention

To make our analysis of resource contention more comprehensive, we also need to analyze the effect of processor contention. This would allow us to compare the relative effects of network contention and processor contention. To do this, we compare the performance of 8X2 co-processor and 8X2 virtual processor modes. For the co-processor mode, we run the processes in cores 0 and 2, while for the virtual processor mode we run the processes on cores 2 and 3. We expect that with more processor contention, the performance of applications would take a hit.

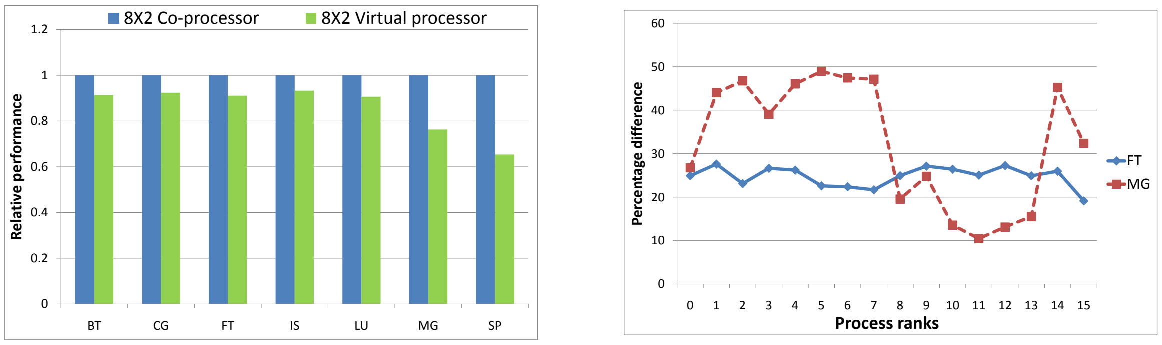


Figure 5.3: Analysis of Processor Contention: (a) Performance and (b) L2 Cache Misses

Figure 5.3(a) shows the normalized performance of co-processor and virtual processor modes in the 8X2 configuration for all the benchmarks. We observe a substantial performance difference between the two modes for all the benchmarks (up to 53% as in the case of SP). This confirms our hypothesis that contention of processor resources can be very detrimental for applications.

We verify our results with processor contention by analyzing the architectural effects of sharing of processor resources. For this, we use PAPI [3] to count various hardware performance counters on the AMD Opteron processor. One of the resources which can create contention in a multicore processor is the shared L2 cache. We use PAPI to count the number of L2 cache misses observed by each process in each mode. We present the percentage difference in the number of L2 cache misses between the co-processor and the virtual processor modes in Figure 5.3(b). As shown in the figure, the virtual processor mode sees increased L2 cache misses ranging from 27% more misses in the case of FT to 48% more in the case of MG. This increase in L2 cache misses confirms the existence of high contention for the shared L2 cache while running in the virtual processor mode.

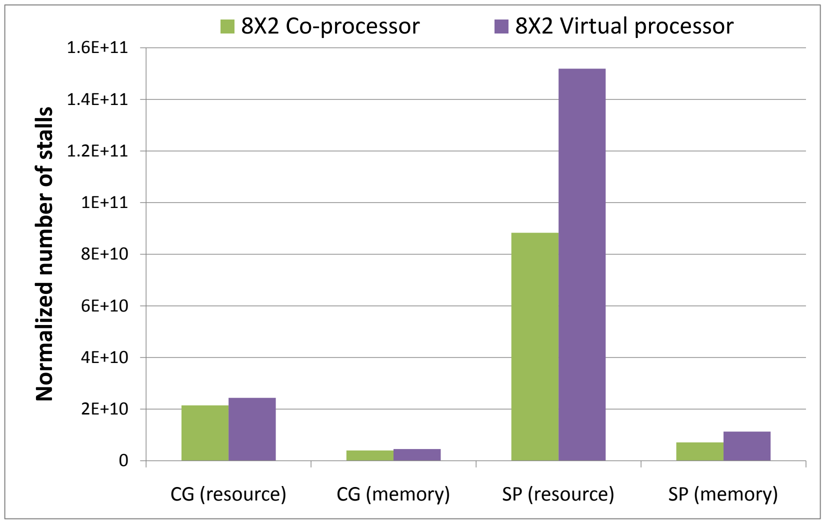


Figure 5.4: CPU Stall Cycles

We further analyze our results by profiling the benchmarks for two types of CPU stall cycles: those stalling for any resource and those stalling for memory accesses. These stalls represent the number of useful CPU cycles wasted due to contention in processor resources. Here we show results only for the CG and SP benchmarks; the results for the other benchmarks are similar. Figure 5.4 shows the normalized number of CPU stall cycles waiting for resource and memory for CG and SP benchmarks. From the graphs, we can see that the virtual processor mode has more stall cycles than does the co-processor mode. SP observes up to 73% more resource stalls cycles and 66% more memory stalls, whereas in the case of CG, it is 14% and 17%, respectively. These results also confirm our observation that processor resource contention can affect performance of applications to a large extent.

## 5.6 Analysis of Allocation Schemes

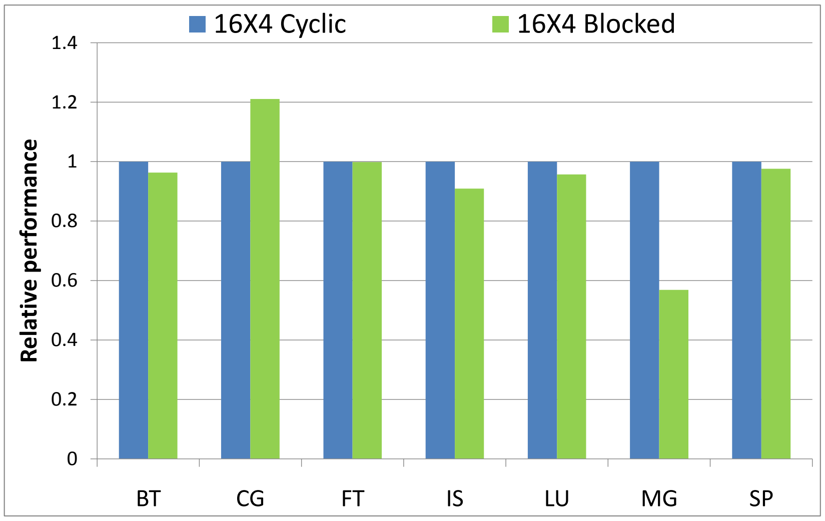


Figure 5.5: Cyclic vs Blocked: Performance

Here we take a different approach for investigating the impact of network contention, by performing a comparative study of two process allocation schemes. The main idea behind this study is to investigate whether changes in process allocation schemes can impact the amount of network contention among processes. We look at two such schemes commonly used in parallel computing environments involving multicore architectures — cyclic and blocked allocation schemes and use the NPB benchmarks to evaluate the performance of these allocation schemes. We run the experiments on 64 processes, with four processes on each of the 16 nodes.

Figure 5.5 shows the performance of various NPB benchmarks with cyclic and blocked allocation on class B data sizes. The results show that the CG benchmark sees an improvement in performance (21%) while for the other benchmarks, performance remains the same or drops. In particular, the performance of MG benchmark drops by more than 43%. Thus, we observe a substantial variation in performance for applications depending on which allocation scheme is used during runs.

Table 5.3: Network Communication Time for Cyclic and Blocked Allocation

| | ***Communication time (seconds)*** | | |
|---|---|---|---|
| ***NAS Benchmark*** | ***Cyclic allocation*** | ***Blocked allocation*** | ***Percentage increase*** |
| *BT* | 0.086 | 0.136 | 57.87 % |
| *CG* | 2.962 | 2.445 | -17.50 % |
| *FT* | 0.020 | 0.059 | 191.80 % |
| *IS* | 0.026 | 0.042 | 62.71 % |
| *LU* | 0.747 | 0.761 | 1.92 % |
| *MG* | 0.049 | 0.305 | 527.70 % |
| *SP* | 0.396 | 0.460 | 16.21 % |

To further understand the reasons behind the trends observed, we profile the network communication time of the benchmarks similar to the profiling done in Section 5.4. Table 5.3 shows the normalized total communication time for each of the benchmarks for cyclic and blocked cases and the percentage difference between them. From the table, we observe that CG realizes a reduction in communication time when running in blocked allocation mode. For all other benchmarks, the network communication time increases. We note here that MG observes more than a five fold increase in communication time, which explains why the performance of MG drops heavily when using blocked allocation.

Figure 5.6 shows the data size communicated over the network for the NPB benchmarks between blocked and cyclic allocation. With CG, the amount of data communicated over the network halves when moving from cyclic to blocked allocation. This result explains CG's increased performance with blocked allocation. FT and IS see no reduction in network data size communicated, whereas MG sees a slight increase. These results also agree well with our other performance results.

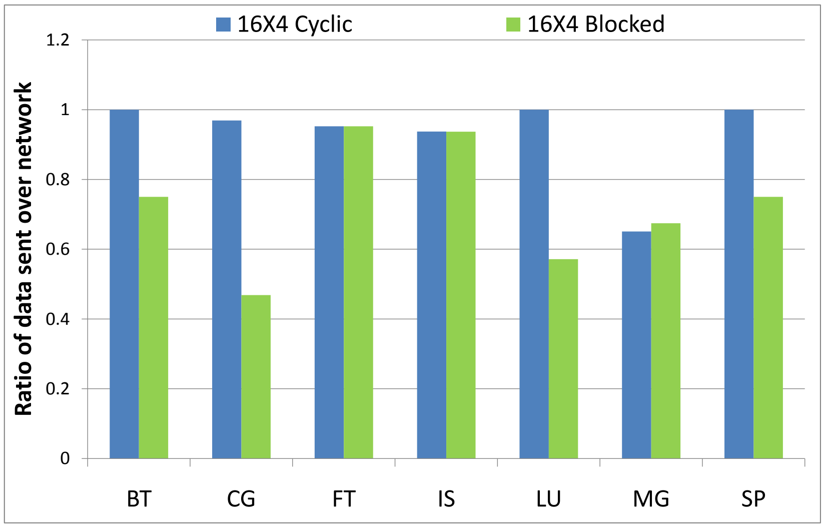


Figure 5.6: Cyclic vs Blocked: Network Data Size

## 5.7 Application Processing Pattern Analysis

The previous sections evaluated application performance from the viewpoint of system and network characteristics. In this section, we tie in the analysis developed in previous sections to the application communication patterns. This will help in deriving insights into better performing process configurations based on the general communication pattern of the application.

The CG benchmark performs communication within groups of four processes with certain boundary nodes communicating between the groups. As an example, Figure 5.7 shows the communication pattern of CG with 16 processes. This pattern shows that any allocation scheme that localizes the groups of four processes within a node will achieve improved performance. For example, if each of the group of four processes are localized within a node, the only network communication is between the boundary nodes. Thus any allocation scheme that optimizes this strategy will get better performance. We see this result with blocked allocation in the 16X4 case, which performs better than the cyclic allocation (see Figure 5.5).

The FT benchmark performs an all-to-all exchange within subcommunicators along the row and column in a processor grid. Thus, having more cores in a node allows some processes in either the row or the column subcommunicators to be local to a node. But the communication as part of the other sub-communicator still has to go through the network. Although some amount of network communication is saved, sufficient contention of network resources still exists. Similarly, choosing an appropriate allocation scheme might help in localizing all the nodes of a sub-communicator, but enough network traffic exists between the other subcommunicator to nullify this advantage. In our results, we see a similar behavior, where the performance drops for FT when moving from 16X1 to 4X4 because of the increased contention of the network but remains the same for the cyclic and blocked allocation strategies. The IS benchmark has a similar analysis as FT as it also does predominantly all-to-all exchanges. This analysis for FT and IS ties in well with the network data size analysis results shown in Figures 5.2 and 5.6. Designing efficient network topologies for FT and IS can be a challenging task given the all-to-all pattern.

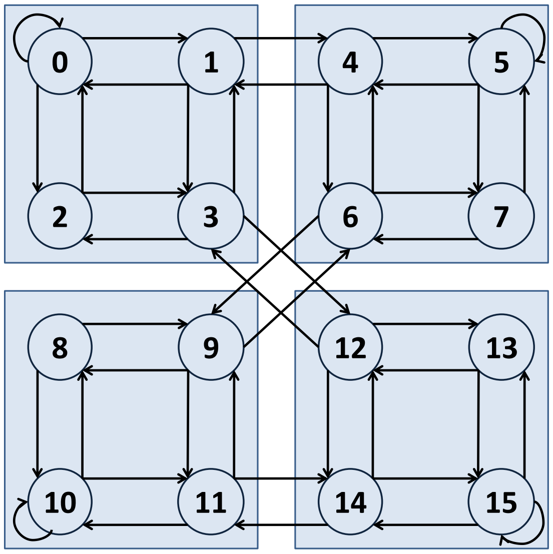


Figure 5.7: CG Communication Pattern

MG has an interesting pattern wherein some clustered communication happens in groups of 4, but these clusters themselves are grouped in clusters of 16. Each process communicates with another process which is at increasing distances of increasing powers of two from it. Thus, any process allocation strategy that puts processes at distances of powers of two on the same node will be beneficial for the application. For example, when the number of nodes is a power of two, cyclic allocation will put such processes on the same node. This situation explains why MG performs better with cyclic allocation than blocked allocation with 64 processes and also why the 4X4 cyclic configuration performs better than the 8X2 configuration.

BT, LU, and SP follow complex communication patterns that make analysis from the processing pattern difficult. Changes in configurations or allocation schemes may not significantly affect the amount of network contention. For example, our results in previous sections do not seem to follow any major trends for these benchmarks.

## 5.8 Summary of Resource Contention Analysis

We saw in Section 5.4 that network contention affects the performance of applications. Hence users need to be careful before utilizing many cores of a multicore architecture for their application. We also saw that contention in processor resources has farther implications than network contention on performance. This is expected given the higher impact of sharing resources closer to the processing core as compared to sharing of peripheral resources such as the network. Although the effects of network contention pale in comparison with processor contention, knowledge of network contention effects can provide valuable insights for users when designing run-time strategies for applications. Thus, both these types of contention are important factors that system designers and users should keep in mind when running applications. Another important result we observed in this section is that using a different process allocation strategy for parallel applications has the potential to reduce the effects of network contention. Finally, knowledge of the application communication and processing patterns can give better ideas for designing better run-time configurations for applications.

# Chapter 6

# Related Work

While there has been a lot of previous research on high-performance networks protocol stacks such as 10 Gigabit Ethernet [22,19,23,10,18,9,16] as well as design aspects on multicore architectures, to the best of our knowledge, no previous work has studied the interaction of high-performance network protocol stacks with multicore architectures. Nevertheless, in this section, we discuss various papers which relate closely to the broader topic of multicore architectures and communication performance of applications on multicores.

In [29], the authors quantify the performance difference in asymmetric multicore architectures and define operating system constructs for efficient scheduling on such architectures. This closely relates to the *effective capability* of the cores (similar to our current work), 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 multicore architectures with communication protocol stacks, which externally creates such asymmetry through aspects such as interrupts and cache sharing.

In [13], the authors study the impact of the Intel multicore architecture on the performance of various applications based on the amount of intra-CMP, inter-CMP and inter-node communication performed. Some of the performance problems of multicores are identified in that paper and the importance of multicore-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. We investigate the problem with a different approach by synthesizing intelligent process-to-core mappings and the amount of sharing of network resources. 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.

In [14], Curtis-Maury et al. look at resource contention with OpenMP communication on multicore processors. The authors identify very similar architectural bottlenecks in their paper. But their paper does not look at resource contention of network resources and how it compares to contention in processor resources, which we address in this thesis. Another paper by Sondag et al., which deals with threads on asymmetric multicore architectures, is [35], wherein the authors discuss scheduling strategies to perform intelligent thread-to-core assignment by using information from static analysis of application profiles and execution information. Our work is related to the work done by Sondag et al. in dealing with the intelligent process-to-core mappings, but significantly differs from their work by considering symmetric multicore architectures and the role of the network protocol stack.

The authors of [7, 17] look at the impact of shared caches on scheduling processes or threads on multicore architectures wherein certain mapping of processes to cores can potentially deliver higher performance by benefiting from better cache locality. We improve upon their work by generalizing the extraneous factors that affect application performance on multicore and by devising a framework for dynamically performing the ideal mapping of processes to cores.

In [5], Alam et al. perform extensive characterization of various scientific workloads on the AMD multicore processor. But their work looks only at a single multicore node, whereas we look at a cluster of nodes and at the impact of the network as well. From the communication pattern viewpoint, many articles and papers have investigated the processing patterns of various applications and benchmarks [36, 15, 34, 27] and suggested improvements for delivering higher performance. But none of these papers focus on multicore architectures in their evaluation, which we address here.

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. Thus, this thesis makes a novel and interesting contribution in this domain.

# Chapter 7

# Conclusions

We present our concluding remarks in this section and identify potential future work that can be done to improve upon this thesis.

## 7.1 Finishing Remarks

With increased adoption of multicore architectures in high-end computing systems, coupled with high-performance network protocol stacks in supporting these architectures, a towering need to analyze the interaction of these two components in the context of application performance can be observed. Since the operating system and systems software that manage each of the different system components have been designed mostly independently, the task of investigating them can be quite challenging. Some of these interactions can reduce the *effective capability* of cores by statically assigning protocol processing to a single core and this leads to surprisingly asymmetric behavior that impact the effective throughput of applications. With growing number of cores in a multicore processor, investigation of network scalability and resource contention is also very important.

In this thesis, we take on the challenge of analyzing the asymmetric interaction of these two components and design efficient solutions for them. We first demonstrate the impact of these interactions and design intelligent mappings of processes to cores for applications and microbenchmarks. We then propose, implement, and verify SIMMer (Systems Interaction Mapping Manager), a framework for managing these irregular interactions automatically. SIMMer enabled applications can benefit from more than two times better communication performance and 18% improved overall application performance.

We have made significant contributions in this thesis in understanding the implications of using multicore architectures with high-speed network protocol stacks. To the best of our knowledge, no other framework/library that has been designed to automatically detect and manage these network interactions. We reckon that the analysis presented in this thesis can be applied to produce better run-time application configurations that can transparently improve application performance without the developer/end user having to explicitly manage these interactions. The results of such studies can also be applicable on other commodity use protocol stacks as well.

## 7.2 Future Work

Our work can be extended in several different directions. We summarize some of the most relevant and important future extensions to our work here:

- We have identified three common symptoms of asymmetry in multicore architectures in this thesis. There can be many more such symptoms which can manifest for certain architectures or applications. SIMMer could be expanded to include these symptoms when considering intelligent mapping of processes onto cores.

- Another important part where our work can be extended is to incorporate the network contention analysis into MPI process managers and process spawners (such as `mpd`, `mpiexec`, `lamboot` etc. This can be done by providing hints to the process managers about the application communication pattern, which can take intelligent decisions about laying out the processes across nodes.

- We can extend this thesis to explore the interaction of protocol stacks with more exotic multicore architectures such as the Intel Terascale, Sun Niagara 2, Tilera and so on. Each of these architectures have a large number of cores on them, and hence, intelligent mapping of processes to cores becomes more critical.

- On NUMA architectures, the benefits of intelligent process-to-core mappings can be potentially reduced by impacts caused by non-uniform memory accesses. Towards this, SIMMer can be improved to incorporate well-informed page migration techniques to counter the NUMA effects on these architectures.

- Another direction in which our work can be extended is to study hybrid and asymmetric multicore architectures which add a new dimension to the problem by having cores of different physical capabilities itself. Such architectures include IBM Cell, Intel IXP network processor, AMD Fusion, and even conventional multicore architectures running on different frequencies.

# Bibliography

[1] MPE : MPI Parallel Environment. http://www-unix.mcs.anl.gov/perfvis/download/index.htm.

[2] mpiP. http://mpip.sourceforge.net.

[3] PAPI. http://icl.cs.utk.edu/papi.

[4] PERUSE. http://www.mpi-peruse.org.

[5] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of Scientific Workloads on Systems with Multi-Core Processors. In *IISWC*, pages 225–236, 2006.

[6] AMD Quad-Core Opteron Processor. http://multicore.amd.com/us-en/quadcore.

[7] J.H. Anderson, J.M. Calandrino, and U.C. Devi. Real-Time Scheduling on Multicore Platforms. *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 179–190, 04-07 April 2006.

[8] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, December 1995.

[9] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High Performance Sockets over Protocol Offload Engines. In *IEEE Cluster*, Boston, MA, Sep 27-30 2005.

[10] P. Balaji, H. V. Shah, and D. K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT)*, San Diego, CA, Sep 20 2004.

[11] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A Message-Passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications*, 91(1-3):43–56, September 1995.

[12] IBM Cell processor. http://www.research.ibm.com/cell.

[13] 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 *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 471–478, 2007.

[14] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In *First International Workshop on OpenMP*, Eugene, Oregon, June 2005.

[15] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements Of Parallel Scientific Applications With Explicit Communication. *20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[16] D. Dalessandro, P. Wyckoff, and G. Montry. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *RAIT '06*, 2006.

[17] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-Fair Thread Scheduling for Multicore Processors. Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University, October 2006.

[18] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *IEEE Hot Interconnects*, Palo Alto, CA, Aug 17-19 2005.

[19] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *SC '03*, 2003.

[20] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[21] P. Gepner and M. F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. In *PARELEC*, pages 9–13, 2006.

[22] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro '04*.

[23] J. Hurwitz and W. Feng. Analyzing MPI performance over 10-Gigabit ethernet. *Journal of Parallel and Distributed Computing*, pages 1253–1260, 2005.

[24] InfiniBand Trade Association. http://www.infinibandta.org.

[25] Intel Core 2 Extreme Quad-Core Processor. http://www.intel.com/products/processor/core2XE/łinebreak[0]qc_prod_brief.pdf.

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

[27] J. Kim and D. J. Lilja. Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs. In *CANPC '98: Proceedings of the Second International Workshop on Network-Based Parallel Computing*, pages 202–216, London, UK, 1998. Springer-Verlag.

[28] Argonne National Laboratory. MPICH2: High Performance and Portable Message Passing.

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

[30] Multicore Technology. http://www.dell.com/downloads/global/power/ps2q05-20050103-Fruehe.pdf.

[31] Myricom. Myrinet Home Page. http://www.myri.com.

[32] Sun Niagara. http://www.sun.com/processors/UltraSPARC-T1.

[33] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[34] R. Riesen. Communication Patterns. In *Workshop on Communication Architecture for Clusters (CSC 2006)*, Rhodes Island, Greece, April 2006.

[35] T. Sondag, V. Krishnamurthy, and H. Rajan. Predictive Thread-to-Core Assignment on a Heterogeneous Multi-core Processor. In *PLOS '07: ACM SIGOPS 4th Workshop on Programming Languages and Operating Systems*, October 2007.

[36] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, 2003.