An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments[*] [†]

G. Narayanaswamy
Dept. of Computer Science
Virginia Tech
cnganesh@cs.vt.edu

P. Balaji[‡]
Math. and Computer Science
Argonne National Laboratory
balaji@mcs.anl.gov

W. Feng[§]
Dept. of Computer Science
Virginia Tech
feng@cs.vt.edu

Abstract

This paper analyzes the interactions between the protocol stack (TCP/IP or iWARP over 10-Gigabit Ethernet) and its multicore environment. Specifically, for host-based protocols such as TCP/IP, we notice that a significant amount of processing is statically assigned to a single core, resulting in an imbalance of load on the different cores of the system and adversely impacting the performance of many applications. For host-offloaded protocols such as iWARP, on the other hand, the portions of the communication stack that are performed on the host, such as buffering of messages and memory copies, are closely tied with the associated process, and hence do not create such load imbalances. Thus, in this paper, we demonstrate that by intelligently mapping different processes of an application to specific cores, the imbalance created by the TCP/IP protocol stack can be largely countered and application performance significantly improved. At the same time, since the load is a better balanced in host-offloaded protocols such as iWARP, such mapping does not adversely affect their performance, thus keeping the mapping generic enough to be used with multiple protocol stacks.

1 Introduction

Multicore architectures have recently established themselves as a major step forward for high-end computing (HEC) systems [10, 18]. Their increasing popularity is of particular importance given the growing scales and capabilities of modern HEC. The commodity market already has quad-core architectures from Intel [5] and AMD [1]. Processors with larger core counts, such as the IBM Cell [2], SUN Niagara [13] and Intel Terascale [6] are also gaining in popularity.

On the other hand, high-performance networks such as 10-Gigabit Ethernet (10GE) [17, 16, 15], Myrinet [19], and InfiniBand (IB) [4] are increasingly becoming an integral part of large-scale systems with respect to scalability and performance. While all these networks aim at achieving the best communication performance, each network splits its protocol stack differently with respect to the amount of processing that is done on the host and the amount that is done on the network interface card (NIC). For example, IB performs almost all of its processing on the NIC. Myrinet (specifically, Myri-10G) performs almost all of its processing on the host. The 10GE family has NICs with different offload capabilities (e.g., regular 10GE, TCP-offloaded 10GE, iWARP-offloaded 10GE). Thus, depending on the amount of processing on the host, it is critical that we understand its interaction with applications running in multicore environments.

In this paper, we study such interaction using two high-performance communication stacks: (i) 10GE with host-based TCP/IP and (ii) 10GE offloaded with iWARP. In the first part of the paper, we provide detailed analysis of these stacks on multicore systems. We notice that, for host-based TCP/IP, 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. For iWARP, however, most of the protocol processing is done by the NIC. The portions of the communication stack that are performed by the host, such as data buffering and memory copies, are done by the application process and its associated libraries, thus localizing it to the process itself and resulting in reduced cache misses.

This leads us to believe that for host-based TCP/IP, based on which process is mapped to which core, application performance can vastly vary. On the other hand, for host-offloaded protocol stacks, such mapping would show no difference in performance. Thus, in the second part of the paper, we utilize this analysis to intelligently map processes-to-core for various applications. Our experiments reveal significant improvement in performance for some applications based on such mapping when using TCP/IP, while showing minimal performance difference when using iWARP. Hence, we conclude that an intelligent mapping of processes-to-cores can significantly improve application performance for TCP/IP while retaining the generality of the application by not affecting its performance for other host-offloaded protocol stacks.

2 Background

In this section, we present an overview of multicore architectures and the NetEffect 10GE iWARP network adapters.

[*]We would like to thank NetEffect Inc., particularly Allan Torres and Gary Montry, for providing us with equipment and technical support for this work.

[†]This paper is published in the proceedings of the IEEE International Symposium on High Performance Interconnects (HotI), Palo Alto, California, August 2007.

[‡]This author's work is supported by the National Science Foundation Grant #0702182 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

[§]This author's work is supported by the Department of Computer Science at Virginia Tech.

2.1 Overview of Multicore Architectures

On-chip hardware replication has been around for many years, providing the CPU with parallelization capabilities for various code segments. Multicore architectures extend these by replicating the microprocessing unit itself (referred to as cores), together with additional portions of the on-chip hardware. While these architectures are similar to multiprocessor systems, they differ in two primary aspects. First, not all of the CPU hardware is replicated. For example, in the Intel architecture, multiple cores on the same die share the same L2 cache, issue queues, and other functional units. Thus, if a core is already using one of these shared hardware resources, another core which needs this resource has to stall. Second, core-to-core data sharing is much faster than the processor-to-processor case since the cores reside on the same die, making cache coherency simpler and faster, and avoiding the die pin when communicating between cores.

2.2 Overview of NetEffect 10GE iWARP

Figure 1 shows the architecture of the NetEffect NE010 10GE iWARP NIC. The NE010 offloads the entire iWARP and TCP/IP stacks to the NIC. So, in theory, these adapters can support all versions of the 10GE network family, i.e., regular 10GE, TCP, and iWARP offload engines. However, the offloaded TCP/IP stack is not directly exposed to applications, and hence these adapters only allow applications to use them as either regular 10GE or iWARP offload engines.

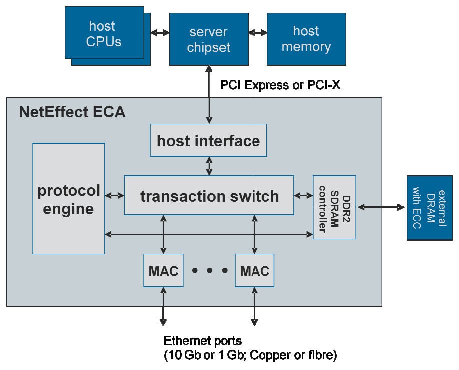


Figure 1: NetEffect iWARP NIC Architecture

The NE010 consists of a protocol engine integrating iWARP, TOE, and regular Ethernet logic in hardware using a structured ASIC. It also consists of a RAM-based transaction switch, which operates on *in-flight* data, and a local memory controller to access NIC memory (256MB DDR2) for buffering non-iWARP connections. These adapters support a number of programming interfaces including a hardware-specific verbs and the OpenFabrics verbs interfaces. These adapters also support a Message Passing Interface (MPI) [7] implementation which is a derivative of MPICH2.

3 TCP/IP and iWARP Processing

In this section, we describe the protocol processing done by TCP/IP and iWARP.

3.1 TCP/IP Protocol Processing

TCP/IP performs many aspects of communication including data buffering, message segmentation, routing, ensuring data integrity (using checksum) and communication reliability. The processing of host-based TCP/IP can broadly be broken down into two components, viz., the synchronous and asynchronous components. The synchronous component refers to the portions of the stack that are either performed in the context of the application process or in the context of the kernel thread corresponding to the application process (e.g., checksum on the sender side, data copies). The asynchronous component, on the other hand, refers to the portions of the stack that are performed in the context of a completely different kernel thread or kernel tasklet (e.g., reliability, data reception, and in some cases, the actual data transmission).

Let us consider the following example to better understand TCP/IP processing. Suppose the sender wants to send a 64KB message. On a `send()` call, this data is copied into the sender socket buffer, segmented into MTU-sized chunks and the checksum for each chunk calculated. Now suppose the TCP window permits the sender to transmit 32KB of data. The first 32KB of the buffered data is handed over to the NIC after which the `send()` returns. The processing so far is done during the `send()`, and thus is a part of the synchronous component. After the `send()` returns, the application can go ahead with its other computation. At this time, suppose the receiver sends an acknowledgment of its data receipt. The sender NIC raises a hardware interrupt to awaken a kernel thread to handle it. The kernel thread sees this acknowledgment and initiates the transfer for the remaining data. Since this part of the processing is done independently from the application, it is referred to as the asynchronous component. On the receiver, the synchronous and asynchronous components are similar.

The important aspect is that the asynchronous component is independent of the application processes. The processing of a asynchronous kernel thread is common for the entire system. Further, in the x86 architecture, hardware interrupts are statically mapped to a single core in the system. Therefore, the kernel process that handles this interrupt also gets mapped to a single core. That is, irrespective of how many processes in the system are performing communication, the asynchronous component of these communications is statically handled by a single core in the system.

3.2 iWARP Processing

iWARP is a relatively new initiative by the Internet Engineering Task Force (IETF) [3] and the RDMA Consortium (RDMAC) [12]. It implements most of the protocol processing relevant to transmission and reception of data on the network hardware. However, aspects such as data buffering and memory copies of the data to final application buffers are not handled by it – upper layers residing on top of iWARP are expected to handle them. For example, in the case of MPI, the incoming data is received in a zero-copy manner into intermediate temporary buffers and later copied into the final buffers by MPI.

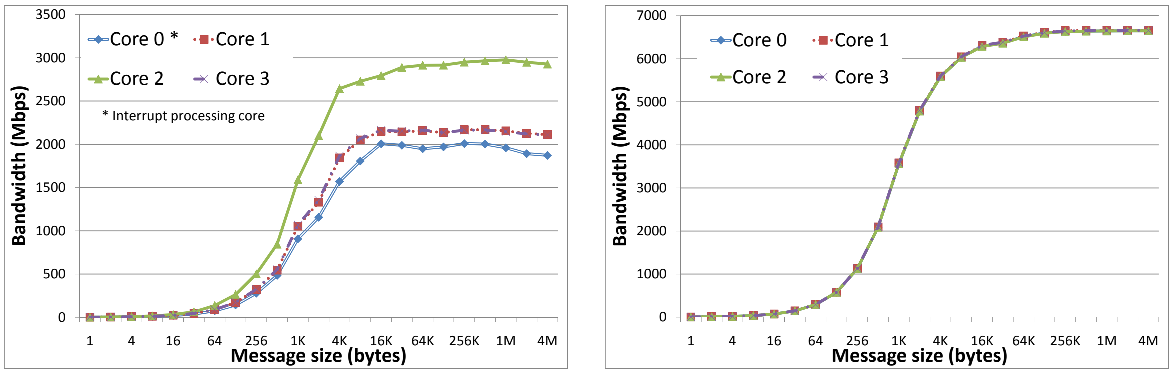


Figure 2: MPI Bandwidth (Setup A): (a) TCP/IP and (b) iWARP

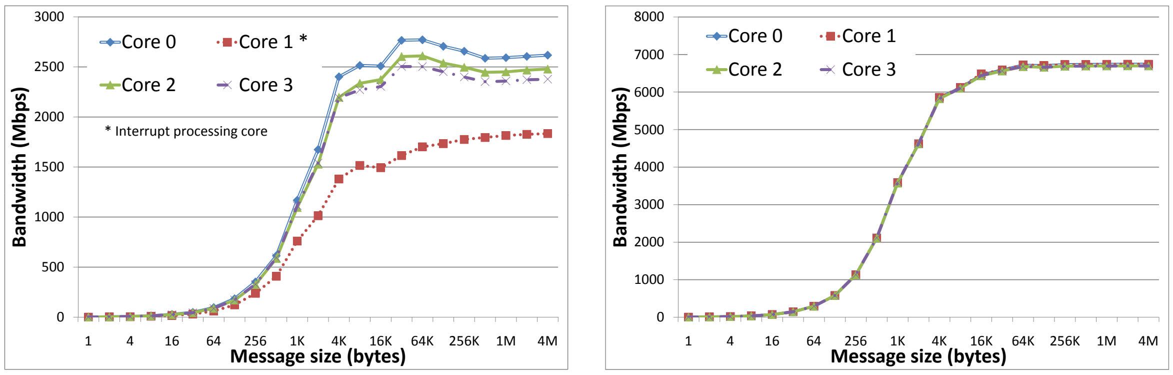


Figure 3: MPI Bandwidth (Setup B): (a) TCP/IP and (b) iWARP

The overall communication stack can be broken down into two portions. The actual transmission and reception of data, that is performed by iWARP, is completely implemented on hardware and is not associated with any specific processing core in the system. The remaining communication aspects (message buffering and data copies) are synchronously handled by communication libraries such as MPI when the application makes a send or receive call. Thus, there is no *application independent* component in the communication processing of host-offloaded protocol stacks such as iWARP and consequently no reason to statically allocate any processing to a fixed core in the system.

4 Experimental Testbed

We used two cluster setups in this paper.

Setup A: Two Dell Poweredge 2950 servers, each equipped with two dual-core Intel Xeon 2.66GHz processors. Each server has 4GB of 667MHz 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. Each processor has a 4MB shared L2 cache. The operating system used is Fedora Core 6 with kernel version 2.6.18.

Setup B: Two custom-built, dual-processor, dual-core AMD Opteron 2.55GHz systems. Each system has 4GB of DDR2 667MHz 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 1MB L2 cache. Both machines run SuSE 10 with kernel version 2.6.13.

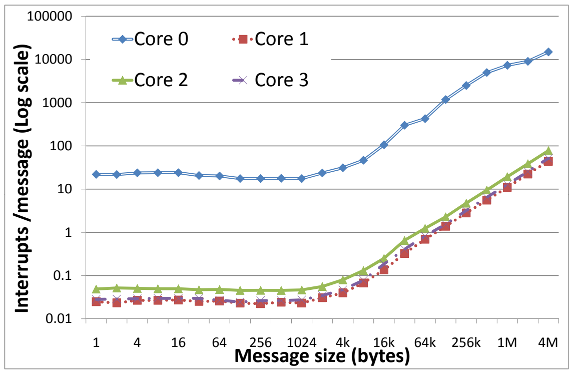
Network and Software: Both setups used the NetEffect 10GE iWARP adapters installed on a x8 PCI-Express slot and connected back-to-back. For TCP/IP evaluation, we used the MPICH2 (version 1.0.5p4) implementation of MPI. For iWARP, we used a derivative of MPICH2 by NetEffect (based on MPICH2 version 1.0.3), that was built using the NetEffect verbs interface.

5 Microbenchmark-Based Analysis

In this section, we analyze the interactions of the TCP/IP and iWARP stacks over 10GE in multicore systems. Specifically, we analyze different microbenchmarks to understand how they are affected in a multicore environment. We present analysis of MPI bandwidth in Section 5.1 and MPI latency in Section 5.2. Both these benchmarks are taken from the OSU MPI microbenchmark suite. Each benchmark was measured at least five times and the average of all runs is reported.

5.1 Analysis of MPI Bandwidth

Figures 2(a) and 2(b) show the MPI bandwidth achieved by TCP/IP and iWARP on setup A, 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. In this experiment, 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 it had transmitted per unit time.



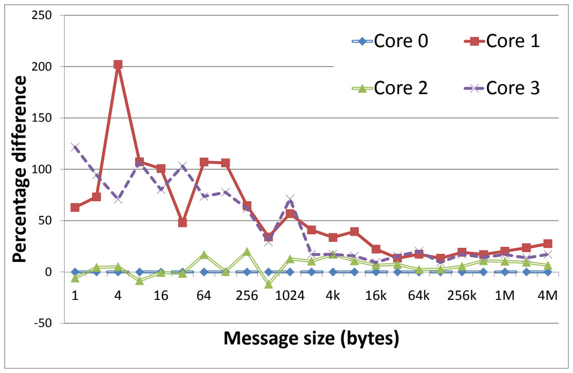
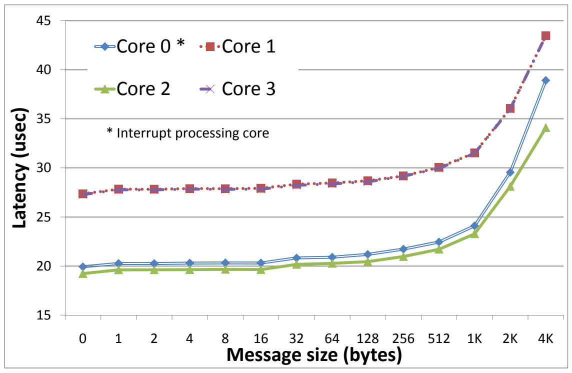


Figure 4: Analysis of MPI Bandwidth: (a) Interrupts and (b) Cache Misses



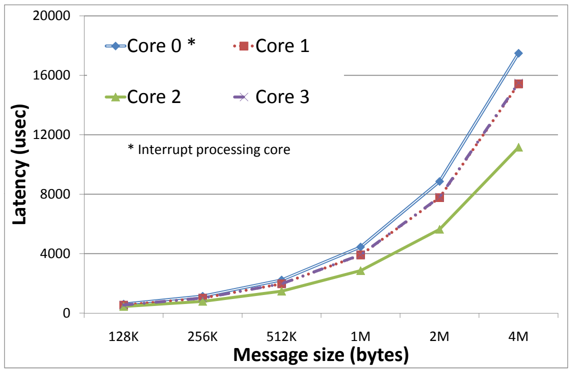


Figure 5: MPI Latencies with TCP/IP: (a) Small Messages and (b) Large Messages

Figure 2(a) shows several trends for TCP/IP. First, when the communication process is scheduled on core 0, bandwidth performance barely reaches 2Gbps. Second, the benchmark performs slightly better when the communication process is scheduled on either core 1 or core 3, i.e., cores on the second CPU. In this case, the benchmark achieves about 2.2Gbps. Third, the benchmark achieves the best performance when the communication process is scheduled on core 2, i.e., the second core of the first CPU. In this case, the benchmark achieves about 3Gbps bandwidth, i.e., about 50% better than the case where the processes are scheduled on core 0. On the other hand, Figure 2(b) shows that, for iWARP, there is no impact on the performance irrespective of which core the communication process is scheduled on.

Figures 3(a) and 3(b) show the MPI bandwidth results on setup B for TCP/IP and iWARP. The trends observed in these figures are very similar to those observed in setup A. That is, for TCP/IP, the interrupt processing core on the first CPU (core 1 in this case) 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 0) achieves the best performance. For iWARP, all core mappings achieve the same performance.

These results indicate that the interaction of the TCP/IP protocol stack with the multicore architecture can have significant impact on performance. To further understand these results, we analyze the interrupt processing and L2 cache misses of the system while running this benchmark in sections 5.1.1 and 5.1.2, respectively. Since both setups A and B show similar performance behavior, we only look at results on setup A in the rest of the paper.

5.1.1 Interrupt Analysis

In order to measure the interrupts generated by TCP/IP during the execution of the MPI bandwidth benchmark, we utilized the Performance Application Programming Interface (PAPI) [11] library (version 3.5.0). Figure 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 the figure, core 0 gets more than 99% of all the interrupts. This observation is in accordance with the description of the asynchronous component in Section 3. That is, the hardware interrupt and the asynchronous component of the TCP/IP stack are statically mapped to a single core in the system.

Based on the large number of interrupts, coupled with the processing of the asynchronous component of the TCP/IP stack by core 0, its capability to perform application processing is drastically reduced. This results in reduced performance of the MPI bandwidth benchmark when the application process is scheduled on this core.

5.1.2 Cache Analysis

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 (as described in Section 3.1), 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 4 (b) shows the percentage difference of number 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 cores 1 and 3 (processor 1)[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.

5.2 MPI Latency Evaluation

Figure 5 illustrates the MPI latency achieved when scheduled on each of the four cores in the system for TCP/IP. Again, both the sender and receiver processes are scheduled on the same core number but of different servers. In this experiment, 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. To better illustrate the results, we have separated them into two groups. Figures 5(a) and 5(b) show the measurements for small and large messages, respectively.

Figure 5(a) shows that the best performance is achieved when the communication process is on core 2. This is similar to the bandwidth test and is attributed to the better cache locality for the process (section 5.1.2). However, when the communication process is scheduled on core 0, there is only a slight drop in performance which is 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 and 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 5(b).

Figures 6(a) and 6(b) show the MPI latencies for small and large messages respectively with iWARP. Similar to the MPI bandwidth benchmark, it can be observed that performance is not affected by the core on which the communicating process is scheduled.

6 Mapping Processes to Specific Cores

In this section, we utilize the analysis provided in section 5 to identify the characteristics of the different processes of real applications and appropriately map them to the best core. We perform such analysis on two applications, GROMACS and LAMMPS, which are described in Sections 6.1 and 6.2.

6.1 GROMACS Application

Overview: GROMACS [14], developed at Groningen University, is primarily designed to simulate the molecular dynamics for millions of biochemical particles. A topology file consisting of the molecular structure is distributed across all active nodes. The simulation time is broken into many steps, and performance is reported as the number of nanoseconds per day of simulation time. For our measurements, we use the GROMACS LZM application.

	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

Table 1: Process-Core Mappings Used in GROMACS LZM

Analysis & Evaluation: There are several different combinations of process-to-core mappings that 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 1). We profile the GROMACS LZM application using mpiP [9] and MPE [8] to get statistical analysis of the time spent in different MPI routines. Figure 7(a) shows the application time break down when running GROMACS with combination A. To simplify our analysis, we show the main components of computation and MPI_Wait, while clubbing 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, a load imbalance occurs in the application.

To rectify this load imbalance, we swap the core mappings for processes 0 and 6 to form combination A' (Table 1). In this

[1]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.

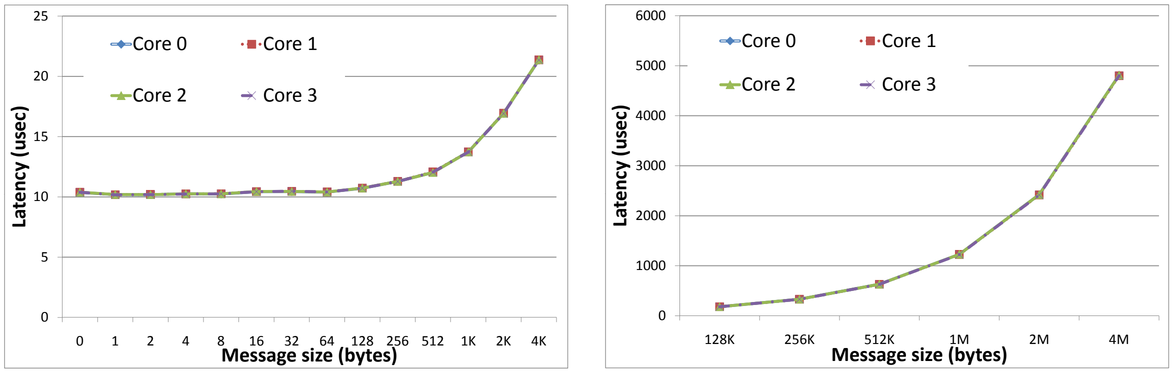


Figure 6: MPI Latencies with iWARP: (a) Small Messages and (b) Large Messages

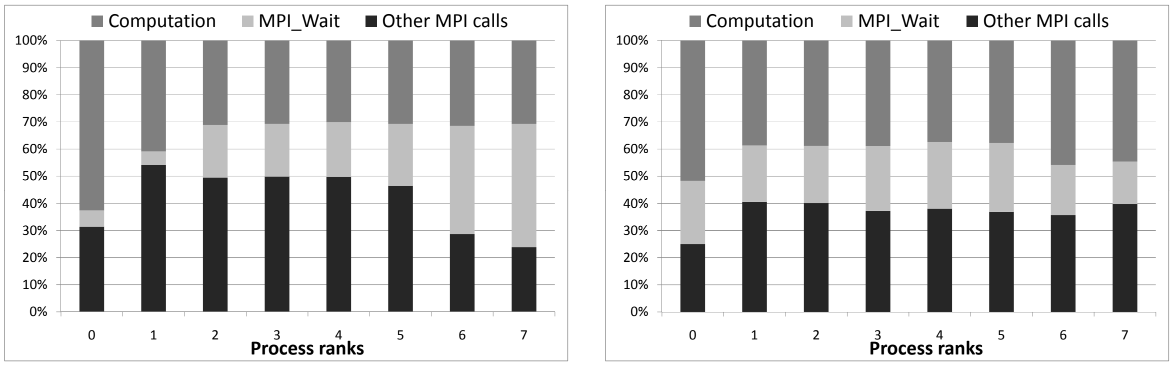


Figure 7: GROMACS application time split up with TCP/IP (a) Combination A (b) Combination A'

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 not affect this process too much. For process 7, however, we notice that it has a large idle time inspite of being scheduled on core 0 of the second machine. We attribute this to the inherent load imbalance in the application. Figure 7(b) shows the application time break up with combination A'. We notice that the load imbalance is lesser in this new combination. Figure 8 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.

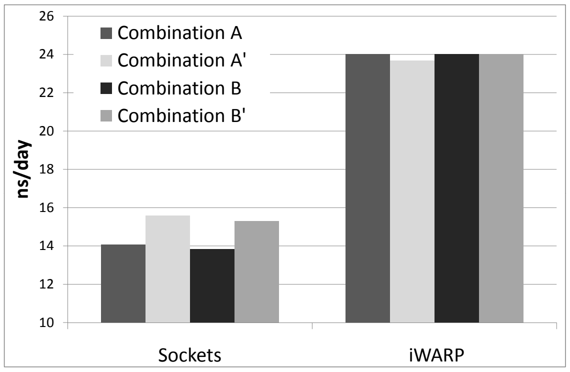


Figure 8: GROMACS LZM Protein System Application

We also notice that, with iWARP, the performance on all core mappings is similar. The maximum standard deviation of the performances with iWARP is only 1.9%. 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, while not adversely affecting its performance on host-offloaded protocols such as iWARP, thus maintaining generality.

6.2 LAMMPS Application

Overviews: LAMMPS [20] is a molecular dynamics simulator developed at Sandia National Lab. 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 linearly scale with the number of atoms being simulated. We use the Lennard-Jones liquid simulation with LAMMPS scaled up 64 times for our evaluation.

	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

Table 2: Process-Core Mappings Used in LAMMPS Application

Analysis & Evaluation: Figure 11(a) illustrates the split up in the communication time spent by LAMMPS while running on processes-to-cores combination A (Table 2). 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 is completely counterintuitive as compared to GROMACS, because 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.

To understand this behavior, we further profile the communication code. We observe that all processes regularly exchange data with only three other processes (Figure 9), and the sizes of the messages exchanged are quite large (around 256KB). Figure 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 time-line in different steps (broken up in the figure using *dotted* horizontal lines).

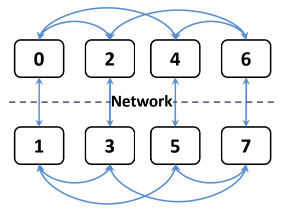


Figure 9: LAMMPS Communication Pattern (8 processes)

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 there is space 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 till all the data from their peer process has 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. On the other hand, if the receiver is not able to read the data fast enough, 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.

Step 3: Once out of MPI_Wait(), process Y goes ahead 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().

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 can see 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 only need ensure that the cores which execute processes X and Y match in capability.

In table 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 11(b) demonstrates that this new intelligent combination can dramatically reduce the imbalance.

Figure 12 shows the communication performance of LAMMPS with the above core mappings. We observe about 50% performance difference between combinations A and A' as well as combinations B and B'. Similar to GROMACS, there is no performance difference while running LAMMPS with iWARP.

7 Conclusions and Future Work

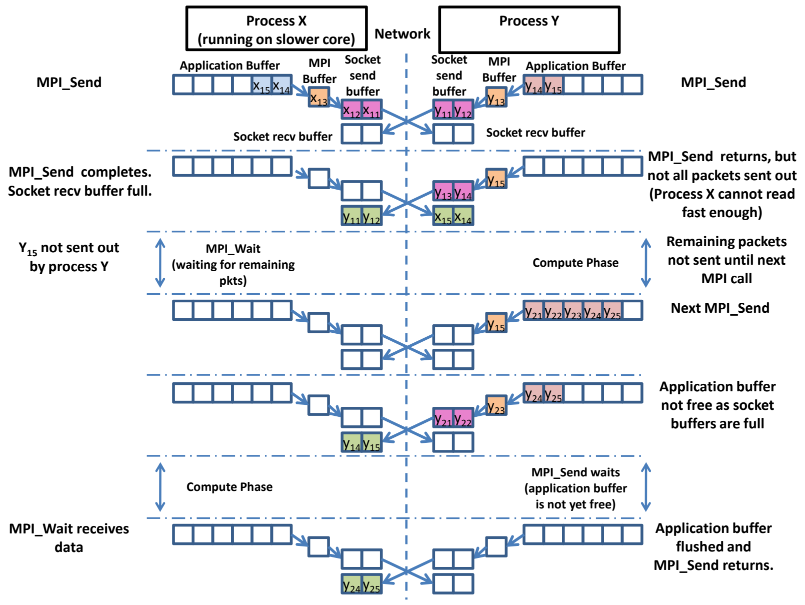
Multicore architectures have been growing in popularity as a significant driving force for high-end computing (HEC). At the same time, high-performance networks such as 10-Gigabit Ethernet (10GE) have become an integral part of large-scale HEC systems. While both of these architectural components have been vastly studied independently, there has been no work which focuses on the interaction between these components. In this paper, we studied such interaction using two protocol stacks of 10GE, namely TCP/IP and iWARP. We first utilized microbenchmarks to understand these interactions. Next, we leveraged the lessons learned from this analysis to demonstrate that intelligently mapping processes-to-cores based on simple rules can achieve significant improvements in performance. Our experimental results demonstrated more to than a two-fold improvement for the LAMMPS application. For future work, we plan to provide a system daemon which would dynamically pick appropriate process-to-core mappings based on the behavior of the processes.

References

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

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

[3] IETF. http://www.ietf.org.



Process X (running on slower core)
Network
Process Y
MPI_Send
Application Buffer
MPI Buffer
Socket send buffer
Socket recv buffer
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
MPI_Send waits (application buffer is not yet free)
MPI_Wait receives data
Application buffer flushed and MPI_Send returns.

Figure 10: LAMMPS Timeline

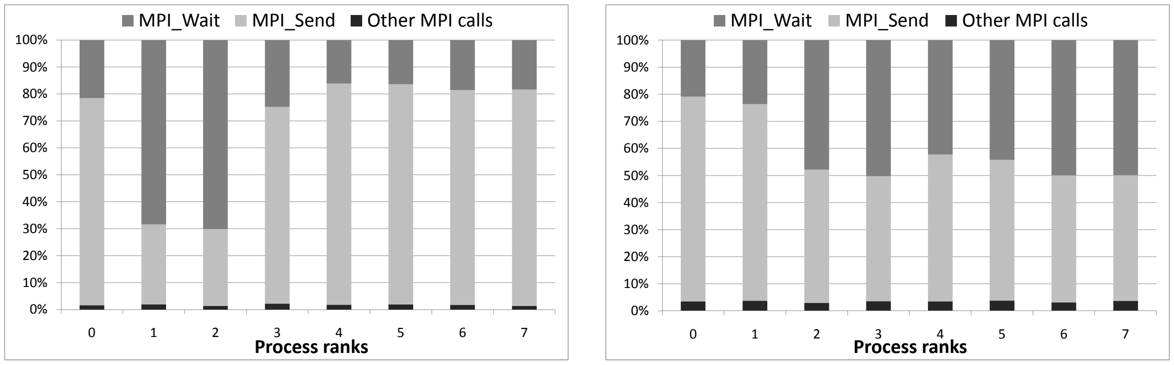


Figure 11: LAMMPS Communication Time Split Up with TCP/IP: (a) Combination A (b) Combination B

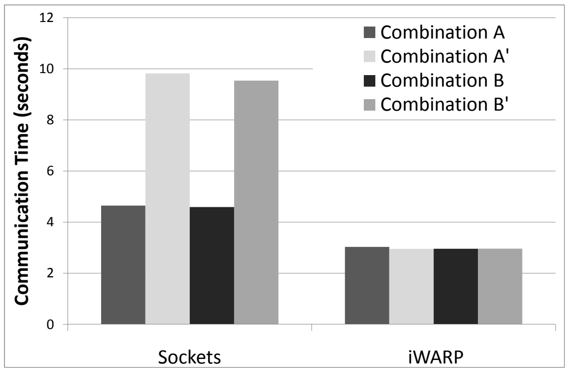


Figure 12: LAMMPS Performance

[4] Infiniband Trade Association. http://www.infinibandta.org/.

[5] Intel Core 2 Extreme quad-core processor. http://www.intel.com/products/processor/core2XE/qc_prod_brief.pdf.

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

[7] Message Passing Interface Forum. http://www.mpi-forum.org.

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

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

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

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

[12] RDMA Consortium. http://www.rdmaconsortium.org.

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

[14] 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.

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

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

[17] 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*.

[18] P. Gepner and M. F. Kowalik. Multi-core processors: New way to achieve high system performance. In *PARELEC*, pages 9–13, 2006.

[19] Myricom. Myrinet home page. http://www.myri.com/.

[20] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.*, 117(1):1–19, 1995.