Blurring the Computation-Communication Divide: Extraneous Memory Accesses and their Effects on MPI Intranode Communications

Wilson M. Tan and Stephen A. Jarvis

Performance Computing and Visualisation Group
Department of Computer Science
University of Warwick, United Kingdom
Email: wilson.tan@warwick.ac.uk

Abstract

Modern MPI simulator frameworks assume the existence of a Computation-Communication Divide: thus, they model and simulate the computation and communication sections of an MPI Program separately. The assumption is actually sound for MPI processes that are situated in different nodes and communicate through a network medium such as Ethernet or Infiniband. For processes that are within a node however, the validity of the assumption is limited since the processes communicate using shared memory, which also figures in computation by storing the application and its associated data structures.

In this work, the limits of the said assumption's validity were tested, and it is shown that Extraneous Memory Accesses (EMAs) by a compute section could significantly slow down the communication operations following it. Two general observations were made in the course of this work: first, more EMAs cause greater slowdown; and second, EMAs coming from the compute section of the processes containing the MPI_Recv are more detrimental to communication performance than those coming from processes containing MPI_Send.

1998 ACM Subject Classification Modeling techniques

Keywords and phrases High performance computing, Message passing, Multicore processing, Computer simulation, Computer networks, Parallel programming, Parallel processing

Digital Object Identifier 10.4230/OASIcs.ICCSW.2012.135

1 Introduction

Many current MPI simulator frameworks such as BSIM[10], WARPP[5] and SST-Macro[7] work on the assumption of a *Computation-Communication Divide*. This assumption states that a message passing program could be divided into two components, the computation section and the communication section, each of which could be simulated independently of each other. This assumption is currently being applied both to processes that are within a single node, and those that are located in different nodes.

The Computation-Communication Divide is actually a reasonable assumption for processes that are between nodes: computation would be the program segments that a processor would handle, while communication would be taken care of by the NIC, routers, and interconnects. The two segments perform independently of each other.

For processes that are in the same node however, the divide is not as clear-cut. The issue lies with the fact that in contrast to internode communication that relies on a dedicated interconnect such as Ethernet or Infiniband, intranode communication relies on shared memory. Aside from facilitating intranode communication, the memory subsystem is also

2012 Imperial College Computing Student Workshop (ICCSW'12).
Editor: Andrew V. Jones; pp. 135–141

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

involved in the process of computation: it stores data and instructions for the application being executed.

Given the importance of intranode MPI in future systems, it is of interest to many to know up to what extent currently held assumption about intranode communication holds. This is the focus of this work: in particular, it investigates how Extraneous Memory Accesses(*EMAs*) affect the communication between MPI processes executing in a single node setup. By "Extraneous Memory Accesses", the authors mean memory accesses that do not figure in the communication process itself, and thus involve data structures that are not being sent between processes. Two possible locations from which EMAs could come from are tested: the sending process and the receiving process.

The rest of the paper is organized as follows: the methodology and system used are described in Section 2, the results are presented and discussed in Section 3, and paper is concluded in Section 5.

2 Methodology

A custom microbenchmark called *PaMPIck* was developed for this work. PaMPIck is primarily based on a simple loop enclosing a send-receive operation pair between Process 0 and Process 1. The loop iterates 100 times. The data being transferred in each send-receive is composed of the first 100 elements of a 1 million element integer array. Each process has 2 1-million member integer arrays: a send array and a receive array. Given that each integer is 4 bytes, 400 bytes were transferred in every iteration, from the sending process' send array to the receiving process' receive array. Other sizes for the data being transferred(aside from 100 elements) were tested, but were not included in this paper for conciseness. The data being transferred between the sending process and the receiving process is never changed between send-receive iterations. The values for the 100 elements are assigned before the very first send-receive, but never changed after that. In a way, this is to actually encourage maximum cache reuse in the send-receive pair.

```
for(iterator = 0; iterator < 100; iterator = iterator + 1)
{
    MPI_Barrier
    PAPI_start(EventSet)

    if (my_rank==0) MPI_Send
    else MPI_Recv

    PAPI_read(EventSet, values)

    //---accumulation of PAPI event
    counts from the iteration---//

    PAPI_stop(EventSet, values)
%%%%%%%---variable section, depending on setup---%%%%%%%
%%%%%%%---either---%%%%%%%
    if(my_rank == 0)
    {
        for(iterator2 = 0; iterator2 < limit;
        iterator2 = iterator2 + 1)
        {
            receivearray[iterator2] = iterator;
        }
    }
%%%%%%%---or---%%%%%%%
    if(my_rank == 1)
    {
        for(iterator2 = 0; iterator2 < limit;
        iterator2 = iterator2 + 1)
        {
            sendarray[iterator2] = iterator;
        }
    }
}
```

Depending on the setup being tested, EMAs were done between send-receive operations. Two kinds of EMAs were tested: those at the sending side(Process 0), and those at the receiving side(Process 1).

Inducing EMAs consists of changing the values of some members of the array not being used by the process for the send-receive operation: this is the receive array for Process 0(the sending side), and the send array for Process 1(the receiving side). These represent accesses done by processes participating in the send-receive on memory elements that do not figure directly with the data being sent or received: in real situations, these could correspond to intermediate or scratch variables.

Each send receive operation was measured using PAPI[9], and the following events and parameters were recorded: virtual cycle time, number of instructions, number of cycles, L1 data cache misses, L2 data cache misses and LLC(last level cache) misses. To make sure that the values being read by PAPI are accurate, each send-receive is preceded by a barrier: this is necessary so that the times and cycles being spent for doing EMAs would not reflect on the values measured by PAPI. The number of cache misses incurred during program execution is very sensitive to many factors such as other programs concurrently running in the system. Therefore, utmost care was taken to ensure that all experiments were carried out in identical conditions as much as possible.

The processor used in this study is an Intel Core i5-2430M, running at 2.40GHz. The i5 is a dual core processor, with three levels of cache memory. Each core has two 32 KB first level caches, one for instruction and one for data. The L2 cache is shared between data and instructions, and is sized at 256 KB. It is core specific. The 3MB 3rd level cache is shared among all cores in the processor.

The operating system of the platform is Linux kernel version 3.0.0-15. Programs were compiled using gcc 4.6.1, with the *-O0* optimization flag. The MPI implementation utilized was OpenMPI[4] 1.4.3, and programs were ran with the "–bind-to-core" flag.

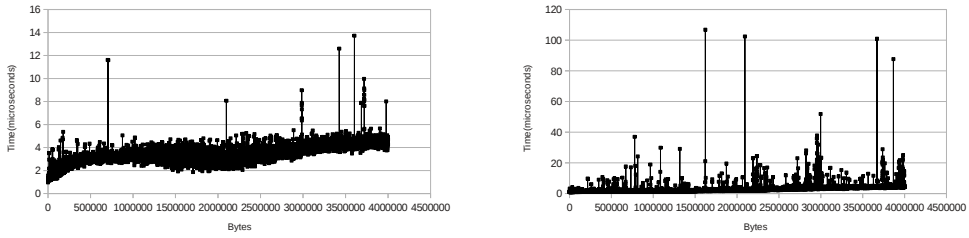
3 Results and Discussion

3.1 Latency: Send-side EMAs and Receive-side EMAs

For the EMAs coming from the Send-side and the Receive-side, several values were tested. The values ranged from no extraneous array entries(0 bytes) modified between iterations to the entire extraneous array being modified(4 Mbytes) between iterations. Values were separated by increments of 100 elements or 400 bytes, resulting in a total of 10,000 runs for each side.

The resulting average virtual/system times(or *latencies*) taken by the MPI_Recv for setup with the Send-side EMAs are shown in Figure 1a, while those with the Receive-side EMAs are shown in Figure 1b. Only the MPI_Recv data is shown primarily for the purpose of brevity. It is nevertheless definitive of the send-receive pair, since a send-receive pair could only be considered finished upon the successful completion of the receive half. Also, according to measurements, an MPI_Recv operation significantly takes up more time than its MPI_Send counterpart.

An immediate observation that could be made about the two graphs already presented is the scattering of the values: the relationship between the number of bytes modified and the operation time is definitely not linear for either setup. Nevertheless, despite the lack of a perfectly linear relationship, it is clear that the time values do tend to increase as the number of bytes modified between iteration increases. Of particular interest to us is the development of the *latency lower bound line*, or the line defining lower boundary of the region



(a) **(b)**

Figure 1 Average latencies for MPI_Recv, send-side EMA setup and receive-side EMA setup.

formed by the aggregation of data points. For instance in Figure 1b, while the average time values fluctuate, the graph shows that beyond 238,400 bytes modified between iterations, the latency would no longer go lower than a microsecond; beyond 2,337,000 bytes, it would no longer go below 2 microseconds.

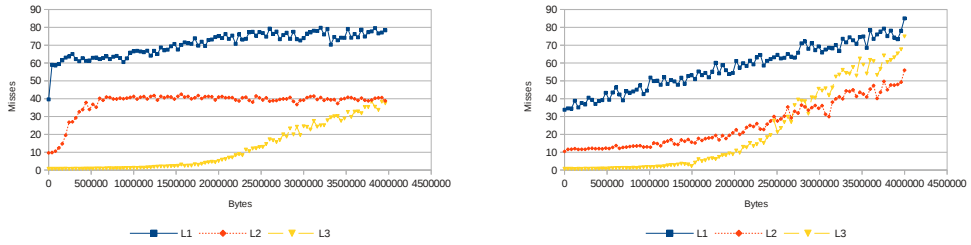
The **First General Observation** of the paper could now be stated: *while the latency of a receive operation could fluctuate up and down, there is always a lower bound value below which it will never go lower than, and that value increases as the number of bytes modified by the preceding compute section increases.*

It is interesting to note that most MPI microbenchmarks also utilize repeated send-receive pairs when measuring latency or bandwidth, not unlike what was utilized in this study. The send-receive pairs in many of these benchmarks are usually separated by very little if any computation, and thus correspond nicely with the leftmost part(bytes = 0) of Figure 1a and Figure 1b.

3.2 Cache Misses: Send-side EMAs and Receive-side EMAs

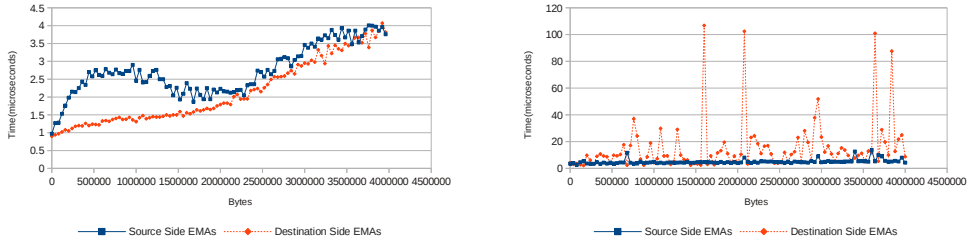
As for the underlying reason behind the general increase in latency as the the number of modified bytes increases, results indicate that the latency trend follows the trend of the average LLC or Level 3 cache miss very closely. This makes is to be expected, since the latency for cache misses in the last-level cache is several times larger than those in higher level caches[6]. Like the latency, the number of cache misses also tend to fluctuate and form dense scatter graphs. For ease of presentation, scatter graphs for the average cache misses were no longer plotted. Instead, the lower bound of the region formed by the conglomeration of data points was derived and plotted. This was done for all three cache levels. The extraction process consisted of taking the minimum of 100-data point exclusive windows, with the first window covering data points 1 to 100, the second window covering 101 to 200, etc. The lower bound lines for the 3 cache levels of the receive side are plotted in Figure 2a.

It could be observed that on the side of the receive side, the number of L1 cache misses always dominate and plateau early, followed by the 2nd level cache(Figure 2a). The number of 3rd level cache misses start rising very slowly and plateaus much later than the other two caches. Take note that the 3rd level cache is shared between all cores, so ascribing the number the misses at that level to a specific core or process with absolute certainty is difficult.



(a) (b)

Figure 2 Average cache misses for MPI_Recv, receive-side EMA setup and send-side EMA setup.



(a) (b)

Figure 3 Lower bound and Upper bound values for the average latencies of MPI_Recv.

3.3 Comparison: Send-side EMAs vs Receive-side EMAs

To be able to compare the effects of Send-side EMAs and Receive-side EMAs, the upper bound and lower bound lines of Figure 1a and Figure 1b were plotted in Figure 3a and Figure 3b. The technique was the same as the one used in Section 3.2, with the exception that the window maximum was taken for the upperbound instead of minimum. The minima(one from each side) are then plotted in Figure 3a, and the maxima in Figure 3b.

Figure 3a, shows that for an equivalent number of EMAs, those from the Receive side actually result in better(lower) bound values than those from the Send side. However, from Figure 3b, it is apparent that in many instances, the Receive side has higher upper bound values than the Send side.

Distribution-wise, the latency values produced by the Send-side EMAs(Figure 1a) are significantly more clustered than the values from the Receive-side EMAs(Figure 1b): the variance of the average latencies from the Receive-side EMAs is 2.95, for that from the Send-side is just 0.6. Nevertheless, the average of the average latencies is higher in the setup with Send-side EMAs: 3.43 microseconds, against 2.92 microseconds.

All these signify that while the lower bounds are better for setups with Receive-side EMAs, in practice, most of the latencies experienced by the send-receive pairs are far higher than the lower bound. In comparison, the lower bounds for setups with Send-side EMAs are worse, but most of the latencies experienced by the send-receive pairs are closer to lower bounds.

These results lead to this paper's **Second General Observation**: *in general, Receive-side EMAs are more detrimental to send-receive performance than Send-side EMAs; at the very least, they make the latency much less predictable than Send-side EMAs.*

4 Related Works and Future Plans

Several proposals have been put forward before with the aim of improving intranode communications. Some, such as [2] focused on user-level mechanisms, while some such as [8] focused on techniques that leverage kernel-level privileges. [3] proposed a hybrid of the two, using different mechanisms depending on message size. This paper is different from all of these works in a sense that it does not propose any modification to existing intranode communications mechanisms; instead, it studied the behavior of one specific intranode communication subsystem(that of OpenMPI), and how it compares with an assumption about it frequently made by simulator framework systems.

The closest previous work to this paper is probably [1], where separate intranode communication mechanisms were compared in terms of latency, bandwidth, and effect on cache. Part of the said work studied the effect of the data transfer mechanisms on the L2 cache. In a sense, this work is the *opposite* of that work, since while the said work focused on the effect of transfer mechanisms on the rest of the application, this paper focused on how the non-data transfer portions(or compute sections) of the application affect the communication section performance. This aspect is emphasized by the fact that in PaMPIck, sends and receives were just repeated in every iteration of the test program: the data being transferred was never changed.

As for future work, the authors intend on carrying out the study on processors that feature more than 2 cores. There is one potential source of EMA that was not tested in this work: *processes that do not figure in the send-receive pair, but are concurrently active with the pair carrying out the send-receive*. The third source of EMA was not tested in the study covered by this paper because the Core i5-2430M is a *dual*-core processor, and testing the third EMA source would have needed 3 processes. The Core i5-2430M could actually support up to four MPI processes, but we would end up oversubscribing one of the cores, so the results many not be conclusive.

In the long run, the authors hope that this work would lead to the development of a better intranode communication model that takes into account the effects of EMAs, wherever they may come from. Such a model is key to projecting optimal setups for future MPI-based HPC systems, which would probably feature more intranode communications than internode ones.

5 Conclusion

The assumption of a *Computation-Communication Divide* is very convenient when reasoning about message passing programs. Unfortunately, it is not always reasonable for cases wherein the processes are located in the same node and communicate through shared memory. Experiment results showed that memory-related activities of the compute section could significantly slow down intranode communications. Two general observations were made in the course of this work: first, the more EMAs made by the preceding compute section, the greater the slowdown for the intranode communication would be; and second, EMAs made at the side of the receiving node cause greater slowdown than those from the side of the sending node. Simply put, when the interconnect has a "memory"(because it *is* memory), the boundary *blurs*. At the level of the node, if the communication section performance is to be successfully predicted, the behavior of the preceding compute section must be taken into account.

While the failure of the Computation-Communication Divide assumption could not be tagged as the source of *all* simulation inaccuracies, the authors nevertheless recommend modellers to consider it as a "suspect" when reality-simulation discrepancies arise.

References

1. D. Buntinas, G. Mercier, and W. Gropp. Data transfers between processes in an smp system: Performance study and application to mpi. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 487 –496, aug. 2006.
2. Lei Chai, A. Hartono, and D.K. Panda. Designing high performance and scalable mpi intra-node communication support for clusters. *Cluster Computing, IEEE International Conference on*, 0:1–10, 2006.
3. Lei Chai, Ping Lai, Hyun-Wook Jin, and D.K. Panda. Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 222 –229, sept. 2008.
4. E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, Ralph H. Castain, D. J. Daniel, R. L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation mpi implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, page 97–104, Budapest, Hungary, 09/2004 2004.
5. S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. Warpp: a toolkit for simulating high-performance parallel scientific codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 19:1–19:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
6. John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.
7. Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. A simulator for large-scale parallel architectures. *International Journal of Parallel and Distributed Systems*, 1(2):57–73, 2010.
8. Hyun-Wook Jin, S. Sur, L. Chai, and D.K. Panda. Lightweight kernel-level primitives for high-performance mpi intra-node communication over multi-core systems. In *Cluster Computing, 2007 IEEE International Conference on*, pages 446 –451, sept. 2007.
9. Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.
10. Ryutaro Susukita, Hisashige Ando, Mutsumi Aoyagi, Hiroaki Honda, Yuichi Inadomi, Koji Inoue, Shigeru Ishizuki, Yasunori Kimura, Hidemi Komatsu, Motoyoshi Kurokawa, Kazuaki J. Murakami, Hidetomo Shibamura, Shuji Yamamura, and Yunqing Yu. Performance prediction of large-scale parallell system and application using macro-level simulation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 20:1–20:9, Piscataway, NJ, USA, 2008. IEEE Press.