Implications of a PIM Architectural Model for MPI*

Arun Rodrigues⋆ Richard Murphy⋆ Peter Kogge⋆ Jay Brockman⋆ Ron Brightwell‡
Keith Underwood‡

University of Notre Dame⋆
Computer Science and Engineering Department
384 Fitzpatrick Hall
Notre Dame, IN 46545

Sandia National Labs† ‡
PO Box 5800
MS-1110
Albuquerque, NM 87185-1110

{arodrig6,rcm,kogge,jbb}@cse.nd.edu, {rbbrigh, kdunder}@sandia.gov

Abstract

Memory may be the only system component that is more commoditized than a microprocessor. To simultaneously exploit this and address the impending memory wall, processing in memory (PIM) research efforts are considering ways to move processing into memory without significantly increasing the cost of the memory. As such, PIM devices may become the basis for future commodity clusters. Although these PIM devices may leverage new computational paradigms such as hardware support for multi-threading and traveling threads, they must provide support for legacy programming models if they are to supplant commodity clusters. This paper presents a prototype implementation of MPI over a traveling thread mechanism called parcels. A performance analysis indicates that the direct hardware support of a traveling thread model can lead to an efficient, lightweight MPI implementation.

1. Introduction

With the memory wall looming for microprocessors, there is a variety of efforts underway to determine how best to avoid it. One active area of research is processing in memory (PIM) technology. The PIM approach integrates processors in the memory device. The advantages are a dramatic increase in memory bandwidth and a significant decrease in memory latency. Many researchers believe that these devices can yield a large improvement in performance with a modest increase in cost. This is particularly true if PIMs are based on commodity DRAM technology. Mass market applications for such a commodity DRAM PIM range from systems-on-a-chip, as in a cell phone or PDA, to the memory or entire node in a future commodity cluster. As such, PIMs have the potential to exceed the volumes of the current microprocessor market. Thus, they may influence the design of next-generation microprocessors and memory systems and have the potential to be the basis of future commodity clusters.

One approach to PIM technologies is to architect the PIM device specifically for parallel computing. This seems counter to the use of such devices in cell phones; however, most PIM approaches are considering multiple processors per DRAM device. Thus, the only difference is that the multiprocessing capabilities would need to be extended across multiple chips. Indeed, some PIM researchers believe that single chip and massively parallel systems should use the same parallelism mechanisms. Multi-threading and traveling threads (based on parcels) are two of the techniques that are being proposed as parallelism mechanisms. In architectures such as PIM Lite[6] or MIND[23], hardware support will be provided to switch between multiple threads in a single cycle. More interestingly the communication mechanism, parcels, will contain thread state to allow extremely lightweight remote invocation (or migration) of threads.

Achieving maximum performance from such PIM based systems may require a change from the message passing model that is widespread today; however, it is critical for PIM based systems to provide reasonable performance on current codes. This paper discusses a prototype implementation of MPI using traveling threads. The prototype implementation contains a minimal number of commonly used operations, but it serves as a platform to evaluate MPI implementation issues on PIMs. Initial analysis of this prototype has indicated that PIMs perform some MPI operations more efficiently than current commodity technologies.

The remainder of this paper is organized as follows. The

*The bulk of the work described here was done at Sandia National Labs. The PIM were concepts developed at Notre Dame, and funded by JPL under the HTMT project, by DARPA under the DIVA project, and by DARPA through Cray Inc. as part of the HPCS program.

†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

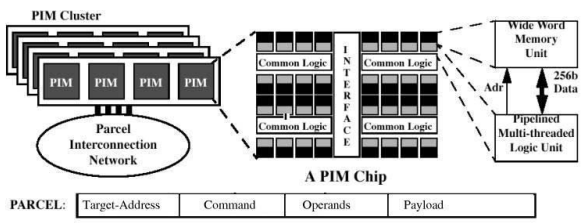


Figure 1. PIM System

PIM computational model used for this work is presented in Section 2. This is followed by details of the MPI implementation for this model in Section 3. The evaluation methods used are discussed in Section 4, and some preliminary results are shown in Section 5. Section 6 presents a comparison to related work. Finally, conclusions and future work are presented in Section 7.

2. Processing-In-Memory

Processing-in-Memory (PIM) [14, 7, 21] also known as Intelligent RAM[22], embedded RAM, or merged logic and memory, combines both high speed logic and dense DRAM on the same die. This arrangement exploits the tremendous amounts of available on-chip bandwidth (potentially terabytes per second) and provides very low latency access (10 ns or better) to a large amount of local state (up to 2048 bits at a time). While this has clear application to systems-on-a-chip, its impact on the high performance community is just beginning to be felt. To date the chips that have been built for standalone high end applications have for the most part supported *conventional* execution models - pure SIMD (e.g. Terasys) to pure MIMD (e.g. Execube[12]). Other experimental systems have still relied on a conventional CPU for overall control or centralized processing (e.g. DIVA[11]), or looked to the outside world as an *SMP on a chip* with message-passing link interfaces between chips (Blue Gene[1]).

This paper assumes a newly emerging view of PIM-based supercomputers (Figure 1) [13, 15, 7]. In this *All-PIM* system, there is nothing but PIM chips, with each PIM chip containing multiple self-contained memory units (MUs). Each MU has an attached logic unit (LU) on which is implemented a specially designed processor whose ISA can take advantage of the wide words that can be fetched from the MUs, and the high bandwidth and low latency with which such accesses can be performed (e.g. PIM Lite). By 2007, for example, such chips could contain dozens of such LU/MU nodes, each running at several GHz and hosting 10s of MB in its MU.

Equally important is how any of these nodes view the other nodes. First, there is nothing other than these chips. Second, they all *look like* memory; requests for access is through *memory addresses*, with the LU that handles the processing for any particular request simply an *anonymous processor* that just happens to be *nearby* the designated memory location. An All-PIM system is thus a sea of memory about which requests for access and/or processing flit on the basis of memory addresses. Third, these requests are much more than dumb memory accesses. They can be requests for serious processing to occur at the targeted address, ranging from atomic memory operations (AMOs) (e.g. add to memory), to remote method, or even complete program invocations. Finally, because there are many MUs in a system (potentially millions), such a system appears as some sort of very large multi-threaded shared memory machine.

We term the communication mechanism that transports these requests as *parcels* (PARallel Computing Elements). As with a traditional dumb memory request, a parcel holds at least a target memory address (used for routing), a command, and/or a few pieces of data (simple *operands*). There may also be a larger data *payload* to transport blocks such as cache lines. In an All-PIM system, however, the command may also be interpreted as a program counter, and the operands as a small set of *registers*. Novelty comes into play in considering computational *threads* in size between the single operation AMOs and the arbitrarily long node-resident threads resulting from an RPI. In particular, small *threadlets* can represent some short sequence of processing that can be done at one node, and then *move* to some other MU associated with the next piece of data. The *command* and *operands* capture the thread state, while the payload may be used for either blocks of data, or a *cache* of sorts that can minimize movements to retrieve previously visited data[20].

For this paper we assume that individual MU/LU are running conventional node level applications programs as one or more of their threads. When such programs perform an MPI function, some mix of parcel-based threads performs the bulk of the work of moving the message from one node to another. We assume that the LU nodes are much like a second generation PIM Lite - 4 stage pipelined, with each stage supporting a separate thread.

3. MPI Implementation using Parcels

The goal of MPI for PIM is to provide a viable "proof of concept" and a testbed for exploring the issues of implementing MPI on a PIM system. Specifically, it explores the effects of a highly multi-threaded programming model on MPI's complexity and performance. As a limited testbed, MPI for PIM implements only a subset of the MPI-1.2 standard[18]. `MPI_Barrier()`, and point-to-point com-

munication were implemented. Support for user datatypes, and multiple communicators were not.

3.1. Effects of threading

MPI for PIM uses pervasive multi-threading to achieve concurrency, reduce the complexity of the implementation, and hide latency. To avoid the traditionally high costs of thread synchronization and programming, MPI for PIM leverages two features of the PIM programming model: fine-grain locking and thread migration.

Conventional single thread implementations of MPI often have difficulty achieving true concurrency with non-blocking communication. After requests are enqueued, the status of the request can only be advanced when a call is made to MPI. Thus, whenever any MPI call is made, a single thread MPI must iterate through its list of outstanding requests and attempt to update their status. This can result in significant overhead as the MPI implementation must "juggle" all outstanding requests whenever an MPI call is made [24]. By using threads, MPI for PIM avoids juggling requests. Requests are assigned a thread which can advance the request without having to wait for an MPI call.

Fine-grain interwoven threads can also reduce or hide latency. For example, a call to `memcpy()` can be divided among several threads, allowing the parallelizing of the copy and fully utilizing the processor pipeline by avoiding stalls.

Traveling threads allow the communication of not just "dumb" data, but also a thread of execution. In MPI for PIM, this means that a receiving process does not have to dedicate resources to monitoring incoming messages and responding to them. Instead a sent message causes a thread migration to the destination process. Once there, the sending thread continues execution, performing any required resource management or copying. Because each incoming message is a thread, it can "look after itself." This avoids having to "juggle" multiple MPI requests.

3.2. Key Data Structures

Each MPI process has two main queues which coordinate communication between the threads on that node:

- `Posted Queue:` contains MPI requests for receive operations which have posted a buffer to be received into, but which are not yet completed. Calls to `MPI_Irecv()` add to this list.
- `Unexpected Queue:` contains requests from messages which arrived at an MPI process, but could not find a posted buffer to be copied into. These messages will allocate a buffer and copy their data to it (for eager messages) or post a "dummy" entry (for rendezvous messages).

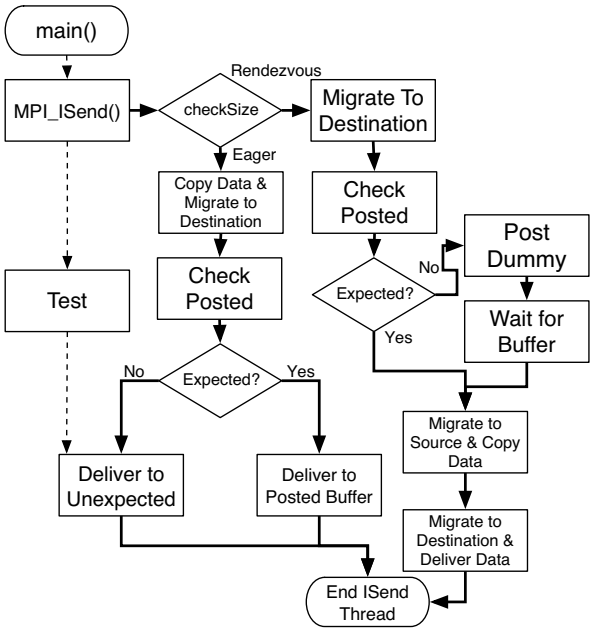


Figure 2. Implementation of `MPI_Isend()` in MPI for PIM

Each queue member is protected by a full empty bit lock, instead of a single lock for the entire queue. This allows multiple threads to traverse the queue concurrently, and the individual locks allow modifications to queue elements to be atomic.

3.3. Implementation of `MPI_Isend()`

All calls to `MPI_Isend()` spawn a new thread. This thread takes one of two paths of execution, depending on the message size, as illustrated in Figure 2. Dashed lines show the flow of the calling thread with solid lines illustrating the flow of the Isend thread.

Data buffers for "Eager" messages (below 64K) are immediately assembled into a parcel for transfer across the network. Once assembled, the `MPI_Isend()` request can be marked as "done" and the thread will migrate to the destination process. Upon arriving, the Isend thread checks the `posted queue` for a matching buffer. If it finds a match, it delivers the message data. Otherwise, the thread allocates a suitable buffer and places a request on the `unexpected queue`.

Messages larger than 64K utilize a rendezvous protocol. The Isend thread migrates to the destination node and

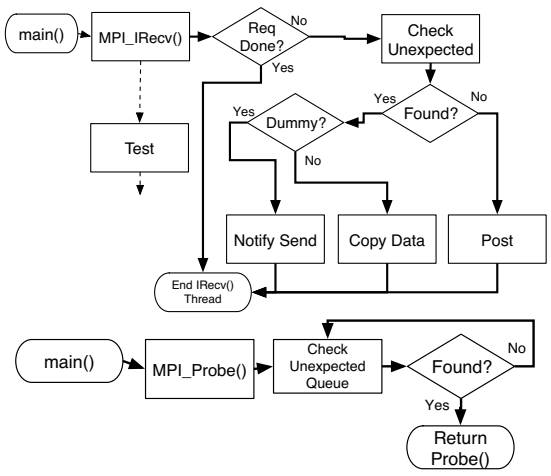


Figure 3. Implementation of `MPI_Irecv()` and `MPI_Probe()` in MPI for PIM

checks for a posted buffer. If it finds such a buffer the thread will claim the buffer and prevent other threads copying data into it by removing it from the `posted queue`. The Isend thread will then return to its source node, and transfer the message data back to the waiting buffer.

If a rendezvous send cannot find a posted buffer, it will post a "dummy" message envelope to the `unexpected queue` and wait for a buffer to become available. By posting to the `unexpected queue`, calls to `MPI_Probe()` are made aware of the rendezvous message. Also, the "dummy" request preserves ordering semantics by ensuring that receive requests will find and match incoming messages in the correct order. When a recieve matches this request it modifies the "dummy" request to inform the waiting unexpected message thread where the recieve buffer is.

3.4. Implementation of `MPI_Irecv()` and `MPI_Probe()`

`MPI_Irecv()` and `MPI_Probe()` both follow somewhat similar paths (Figure 3). Because `MPI_Irecv()` is nonblocking, it begins with a thread spawn. `MPI_Probe()` is blocking, so it is does not execute in another thread.

`MPI_Irecv()` first checks the status of its request, as the request may have already been completed by an incoming send. If the request has not been completed, the Irecv thread checks the `unexpected queue` for a match. If no match is found, it posts its request to the `posted queue` and exits. To preserve MPI ordering semantics, the `unexpected queue` is locked while it is being checked and the receive is posted.

`MPI_Probe()` repeately checks the `unexpected queue` for a match.

4. Evaluation Methods

The initial comparison of MPI for PIMs and commodity processors is focused on measurements of the complexity of the code paths for some core MPI routines. Thus, it is based on a simplistic microbenchmark. Traces of this microbenchmark under a variety of possible usage scenarios were taken and compared for MPICH 1.2.5 and LAM-MPI 6.5.9 on a PowerPC and for MPI for PIM on a simulated PIM architecture. This section describes the benchmark and the methodology that was used for tracing and simulation.

4.1. Benchmark

The microbenchmark[5] used for this evaluation was written at Sandia National Labs to consider the impact of posted versus unexpected receives. The code uses a combination of `MPI_Irecv`, `MPI_Send`, `MPI_Recv`, `MPI_Barrier`, `MPI_Probe`, and `MPI_Waitall` to control the percentage of messages that are unexpected. The test sends 10 messages of parameterizable size in each direction (for a total of 20 sequential sends). This benchmark was used for this analysis because it effectively exercised a small set of the most commonly used MPI routines under varying usage scenarios. This allowed us to vary the code paths taken and study the impact of those code paths on instruction count, memory references, and instructions per cycle (IPC).

4.2. Trace Based Analysis

Traces for the baseline conventional implementations (LAM and MPICH) were gathered on an Apple Macintosh Power Mac with a PowerPC MPC7450 (G4+) processor running at 1Ghz. This platform was running Darwin kernel version 6.6 (Mac OS X 10.2.6). The `amber` utility [2] was used to gather instruction traces of the microbenchmark described in Section 4.1 using both LAM and MPICH implementations of MPI. These instruction traces were then converted to an architecture independent format called TT7 for further analysis.

Execution of MPI for PIM was performed on a PIM Architectural simulator, which can also generate traces. The architecture simulated mirrored a possible 2nd generation PIM Lite. The MPI for PIM source code was instrumented with special tracing functions so instructions in the trace could be categorized into broad categories (see Section 5.2). To generate execution times for MPI for PIM, the traces from the architectural simulator were simulated on a PIM Trace-based simulator.

Table 1. Configurations used for simulation

Variable	simg4	PIM
memory lat., open page	20 cycles	4 cycles
memory lat., closed page	44 cycles	11 cycles
L2 latency	6 cycles	NA
Pipelines	7	1
Pipeline Depth	4 (integer)	4 (interwoven)

To provide a fair comparison between MPI for PIM and other implementations, sections of the LAM and MPICH traces that concerned functionality not implemented in MPI for PIM were discounted. These include functions that dealt with specifics of the network interface, bookkeeping, debugging, datatype or communicator lookup, byte ordering, and parameter checking. Such functions were identified and any instructions in the trace which executed in these functions were removed. To accomplish this, a disassembler was used to find mappings between instructions in the TT7 Trace and functions in LAM or MPICH.

4.3. Simulation Based Analysis

Cycle counts for execution on the PowerPC were obtained using the simg4 cycle accurate simulator from Motorola [19]. This simulator produced accurate cycle counts, instruction mixes, pipeline stall counts, and cache performance data. Cycle count estimates for the instruction categories for each function shown in Section 5.2 were estimated using output from simg4. Pipeline stall counts for memory instructions were used to calculate an approximate IPC for memory instructions. Given this number, the number of memory instructions, and the overall number of cycles to execute the function trace, it was possible to estimate the average IPC of non-memory instructions for that function. The relative number of memory to non-memory instructions belonging to each instruction category were combined with the IPC estimates to produce a cycle estimate for each category.

The PIM Architectural simulator is based off of the SimpleScalar tool set [8]. It extends the PISA ISA to add extra PIM functionality such as thread migration and the manipulation of Full/Empty Bits. It can simulate multiple PIMs and includes support for adjusting several architectural features (Table 1).

5. MPI Performance Impact

This section presents results comparing various aspects of the performance of the MPI for PIM prototype and MPI implementations on commodity platforms. As described in Section 4, only the aspects of MPI that were implemented in MPI for PIM were analyzed. Comparisons are presented for eager (256 bytes) and rendezvous (80 KB) transfers.

5.1. Overhead Reduction

An important aspect of MPI for PIM is a reduction in the overhead of MPI calls. MPI overhead includes time spent performing tasks other than network communication and buffer copies. With a pervasively multi-threaded implementation, MPI for PIM can avoid much of the MPI state swapping, or "juggling", that must occur in a single thread MPI. MPI for PIM executes fewer overhead instructions than LAM, and usually fewer instructions than MPICH (Figure 4(a-b)). The PIM implementation also makes fewer memory references (Figure 4(c-d)). The reduction in memory references is compounded because the PIM processor is "closer" to the memory. So, memory references on PIMs tend to be lower latency than on conventional machines. Combining the reduction in memory references with the improvement in memory access time yields a significant reduction in the time spent accessing memory.

Because MPI for PIM's memory references are fewer and faster, its overall IPC tends to be high. MPICH suffers from a high branch misprediction rate (up to 20%), which usually limits its IPC to less than 0.6. LAM's IPC for eager messages is high, however, for longer messages it suffers from data cache misses which limit its performance. These differences in IPC and executed instructions result in an overall cycle count which is lower than the conventional MPIs. For eager sends, MPI for PIM averages 57% less overhead than MPICH and 42% less than LAM. For rendezvous sends, MPI for PIM averages 58% less overhead than MPICH and 78% less than LAM.

The actual time spent in MPI would depend on the fabrication process used in a PIM processor. However, a PIM pipeline would generally be much simpler than a conventional processor and would probably be able to run at a similar clock rate. Additionally, as conventional processor speeds grow, the latency between memory and processor would also increase further limiting conventional performance.

5.2. MPI Function Analysis

To explain the performance differences between MPI for PIM and conventional single threaded MPIs, it is useful to examine several of the major MPI calls. The overhead in these calls can be classified into one of four behaviors:

- **State Setup/Update:** Initialization and update of MPI Requests and internal state dealing with the progress of a function.

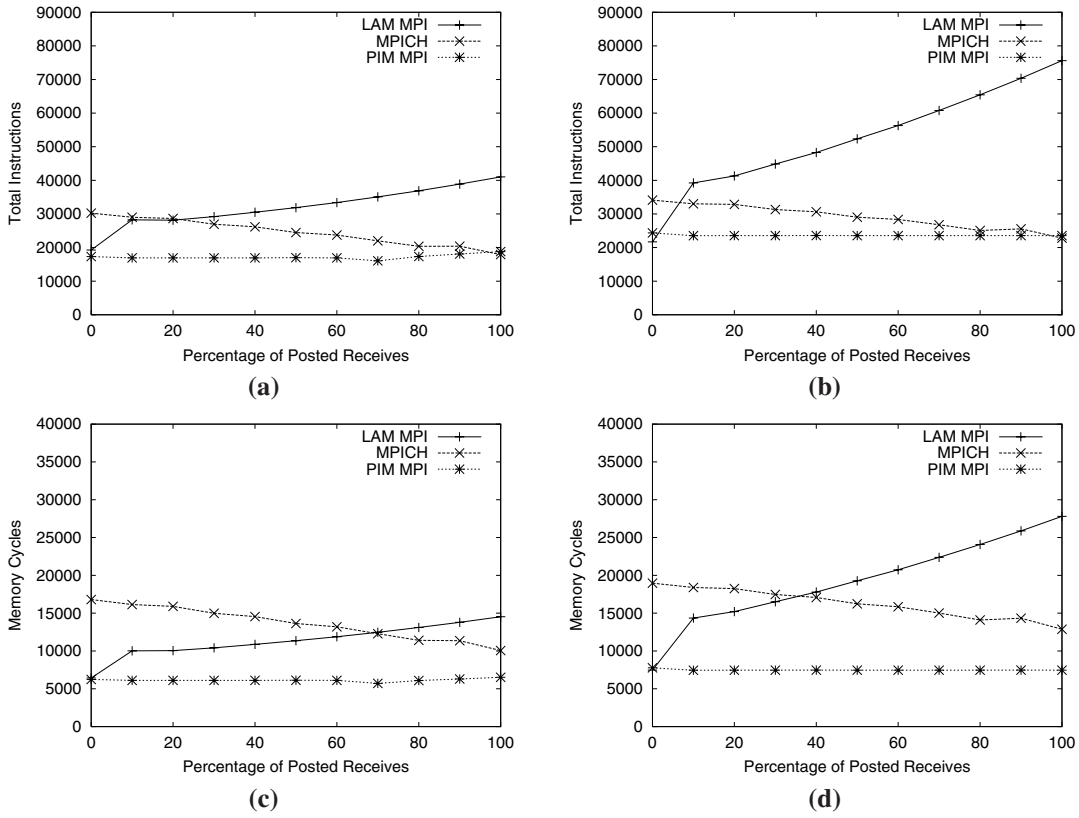


Figure 4. Total instructions (excluding network instructions) executed in MPI routines for benchmark application using **(a)** eager sends and **(b)** rendezvous sends; Number of memory accesses (excluding network instructions) by MPI routines for benchmark application using **(c)**eager sends and **(d)** rendezvous sends.

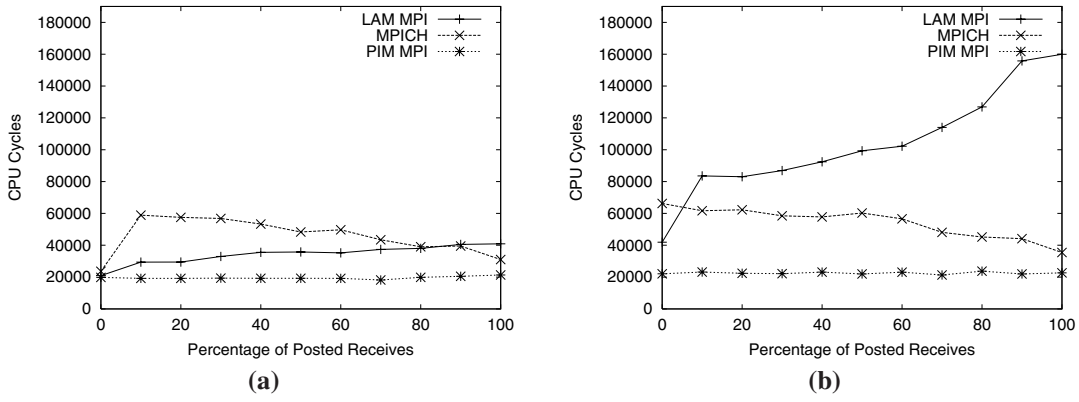


Figure 5. Total CPU cycles spent in MPI routines for benchmark application for **(a)** eager sends and **(b)** rendezvous sends, excluding network instructions.

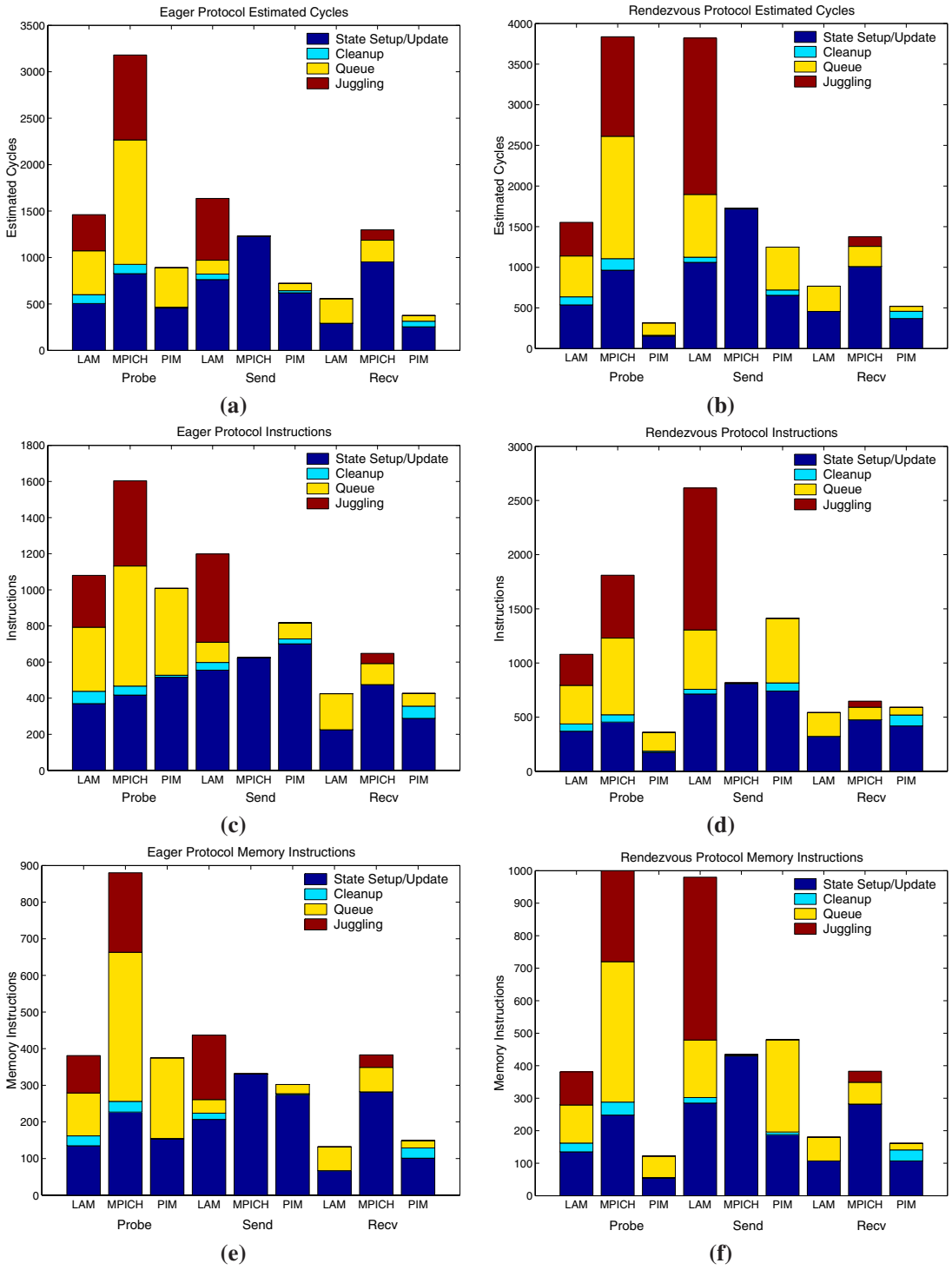


Figure 6. A breakdown of the CPU cycles spent in each of three routines for **(a)**eager sends and **(b)** rendezvous sends; A breakdown of the instructions executed in each of three routines for **(c)**eager sends and **(d)** rendezvous sends; A breakdown of the memory access instructions executed in each of three routines for **(e)**eager sends and **(f)** rendezvous sends. All breakdowns exclude network and memory copy instructions.

- **Cleanup:** Deallocation of data structures, unlocking of synchronization controls, removal of requests from lists or queues.
- **Queue Handling:** Iterating through lists or queues to advance requests or match envelopes. May also include searching hash tables for matches (LAM) and acquiring synchronization locks (MPI for PIM).
- **Juggling:** Time spent switching from the MPI context of one request to another in single threaded MPIs. This generally occurs when there are multiple outstanding non-blocking requests and MPI must check each to see if progress can be made on them.

MPI for PIM generally executes MPI functions with less overhead than single threaded MPIs. This improvement comes from several sources such as: faster memory accesses, reduced state setup for the rendezvous protocol, and elimination of the need to "juggle" multiple requests.

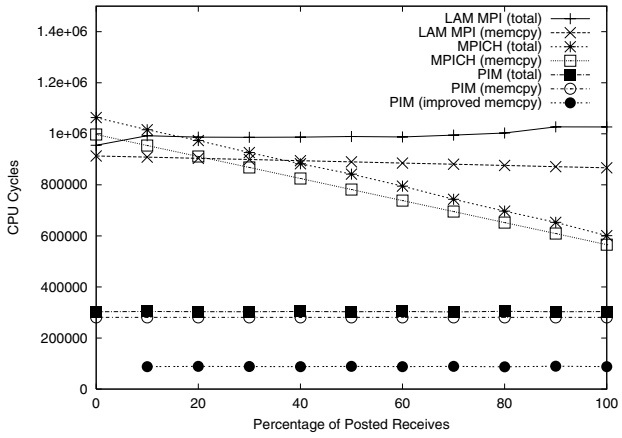
MPI for PIM requires fewer cycles to setup and maintain state in several key MPI functions, especially when comparing the rendezvous protocol (Figure 6(a-b)). This is due to the use of "intelligent" traveling threads to perform sends. A conventional MPI must expend cycles initializing and updating a send request, and then interpreting the incoming data and dispatching it based upon protocol. In effect, a conventional MPI must setup the state information for send twice. In contrast, an incoming thread in an MPI based upon traveling threads is already initialized and can "dispatch itself".

Another advantage of MPI for PIM is that MPI functions do not have to switch contexts from one MPI request to another to advance pending requests. The overhead of this "juggling" of requests can be quite significant (Figure 6(c-d)), especially since this class of behavior tends to require a large number of memory accesses (Figure 6(e-f)). In LAM, it accounted for 14% to 60% of MPI overhead instructions, depending on the number of outstanding requests. In MPICH, it accounted for between 18% and 23%.

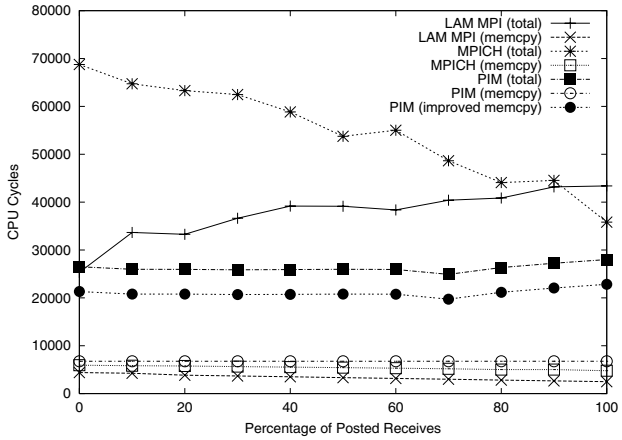
There are some cases where MPI for PIM performs poorly compared to LAM or MPICH. MPICH's `MPI_Send()` can outperform MPI for PIM with rendezvous sized messages. It appears that MPICH's send performs a "short-circuit" type optimization and bypasses the normal queuing and device checking procedures. Lastly, MPI for PIM often requires more instructions in cleanup activities, probably due to the extra queue unlocking which is required for synchronization.

5.3. Other Performance Impacts

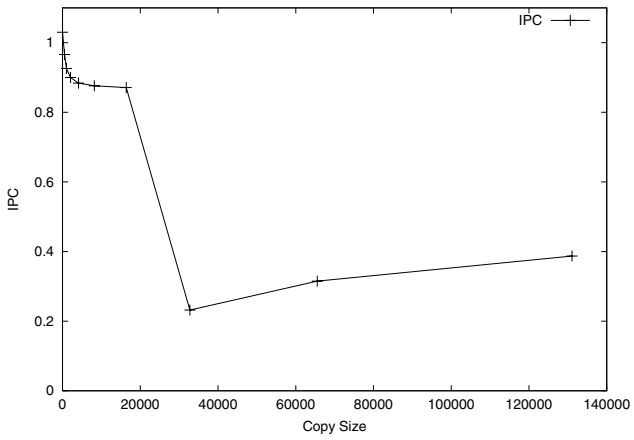
MPI frequently requires memory copies to handle unexpected messages, pack data, assemble messages, and



(a)



(b)



(c)

Figure 7. Total MPI cycles, including `memcpy`s for **(a)** rendezvous sends;**(b)** eager sends (at a much more detailed scale); **(c)** Conventional `memcpy` IPC for varying copy sizes.

perform shared memory communication. These memory copies can account for a significant percentage of the total time spent in MPI, especially for large message sends. Conventional processors suffer significant performance degradation when performing memory copies which exhaust their cache. This effect is shown in Figure 7(c). A PowerPC G4 (32K L1 data cache) can perform a `memcpy()` of less than 32K at an IPC close to 1.0. However, larger memory copies show a performance drop, with IPCs less than 0.4. This drop in performance is a graphic depiction of hitting the "memory wall" and will only become more pronounced as the gap between memory and processor speeds grows.

PIM processors have several advantages when performing memory copies. The first is that a PIM processor is "closer" to memory. It does not have to go through several layers of cache, but is connected directly to the memory macro. Additionally, it is possible to copy a full DRAM row at a time, which gives it dramatically higher bandwidth. By utilizing the architectural features of PIM to reduce memory copy times, MPI time could be considerably reduced (Figure 7(a) and (b)).

6. Related Work

The prototype MPI implementation that we have described in this paper is very similar to other MPI implementations on top of active message layers, such as those described in [9, 16, 3]. It is also similar to implementations that have been built on networks that have remote DMA (RDMA) capability, such as those described in [4, 10, 17]. However, the traveling thread model is able to support some features of MPI much more efficiently.

For example, most of the implementations of MPI on top of active messages require the process to poll the network in order to process messages and activate message handlers. This can lead to inefficiencies when the receiving process is not running, and, in some cases, may violate the progress rule of MPI. Hardware support for traveling threads increases the ability of remote processing to occur on the arrival of messages without interference from the operating system and without requiring the receiving process to waste processor cycles polling the network.

Existing RDMA-based implementations of MPI also suffer from similar issues. Messages can arrive without explicitly polling by the receiver, but the MPI library must actively notice incoming messages and process them. For example, a short message is typically written into a buffer that is managed by the MPI library, and is later copied into a receive buffer. This can only occur after the MPI library notices that it has arrived. Traveling threads allow for this processing to happen immediately upon thread arrival.

7. Conclusions and Future Work

This work is based on a PIM architecture that could form the basis for commodity cluster computing in the future. As such, it is important to consider the implications of this technology for current computing paradigms. This work presents an analysis of an initial implementation of MPI on PIM architectures. Although the PIM architecture is explicitly designed for parallelism, it is not explicitly designed to support MPI. Despite this, the preliminary analysis indicates that a PIM architecture will support MPI very well, and may reduce the complexity of the MPI implementation via inherent multi-threading. In terms of performance, MPI for PIM requires fewer CPU cycles than equivalent commodity implementations for many of the operations. This is attributable to a significant reduction in total instructions (through the use of special features in the PIM) and an increase in instructions per cycle (IPC). Overall, this work demonstrates that an MPI implementation for PIM is not only possible, but is likely to perform at least as well as what is found on commodity systems.

Only a preliminary analysis is presented here. Future work will focus on implementing more of the MPI standard to permit application simulation on the architectural simulator and to further study MPI performance improvements achievable with PIMs. For example, PIM instruction sets will likely provide vector types of operations on extremely wide words. Additionally, the extremely high memory bandwidth provided by PIMs may offer a significant win for applications using MPI derived datatypes. Also, PIMs can offer extremely fine grained synchronization methods that will allow automated exploitation of opportunities for communication and computation overlap. For example, it may be possible to allow an `MPI_Recv` to return before all of the data has arrived. Fine grained synchronization could then block the application if it attempted to access a portion of the data that has not arrived. Finally, PIMs may also support the MPI-2 one-sided communication functions very efficiently, especially the accumulate operation, which allows for operations to be performed on remote data.

References

[1] F. Allen and *et. al*. Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer. *IBM Systems Journal*, 4(2), 2001.

[2] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.

[3] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1081–1093, Oct. 2001.

[4] R. Brightwell and A. Skjellum. MPICH on the T3D: A case study of high performance message passing. In IEEE, editor, *Proceedings. Second MPI Developer's Conference: Notre Dame, IN, USA, 1–2 July 1996*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[5] R. B. Brightwell and K. D. Underwood. Evaluation of an eager protocol optimization for mpi. In *EuroPVM/MPI 2003*, September 2003.

[6] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang. Pim lite: On the road towards relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, 384 Fitzpatrick Hall, Notre Dame IN 46545, February 2003.

[7] J. B. Brockman, P. M. Kogge, V. Freeh, S. K. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ICS*, 1999.

[8] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[9] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.

[10] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In *Proceedings of the Third MPI Developers' and Users' Conference*, pages 15–24, March 1999.

[11] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.

[12] P. M. Kogge. The execube approach to massively parallel processing. August, 1994.

[13] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. H. Sha. Pursuing a petaflop: Point designs for 100tf computers using pim technologies. October 25-31, 1996.

[14] P. M. Kogge, J. B. Brockman, and V. Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.

[15] P. M. Kogge, J. B. Brockman, and V. W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.

[16] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[17] J. Liu, J. Wu, S. P. Kinis, P. Wyckoff, and D. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing*, June 2003.

[18] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[19] Motorola System Performance Modeling and Simulation Group. *Sim_G4 v1.4.1 User's Guide*, 1998. Available as part of Apple Computer's CHUD tool suite.

[20] R. C. Murphy and P. M. Kogge. Trading Bandwidth for Latency: Managing Continuations Through a Carpet Bag Cache. In *Proceedings of the International Workshop on Innovative Architecture 2002 (IWIA02)*. IEEE Computer Society, January 10-11, 2002.

[21] R. C. Murphy, P. M. Kogge, and A. A. Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *Proceedings of the Second Workshop on Intelligent Memory Systems, held in conjunction with ASPLOS-IX, Cambridge, MA*. ACM Press, November 12-15, 2000.

[22] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[23] T. Sterling and H. Zima. Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing. In *SC2002, Baltimore, MD*.

[24] T. L. Team. Porting the lam-mpi 6.3 communication layer. Technical Report TR00-01, Univesity of Notre Dame, 2000.