Message Passing On Communication-Exposed Multi-Core Processors

James Psota and Anant Agarwal
M.I.T. Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, 02139
{jim,agarwal}@csail.mit.edu

ABSTRACT
Next-generation microprocessors will increasingly rely on parallelism, as opposed to frequency scaling, for improvements in performance scalability. Microprocessor designers are attaining such parallelism by placing multiple processing cores on a single silicon die. Current commercial multi-core processors such as the POWER5 and AMD Opteron 800 force inter-processor communication to go through the memory system. However, some multi-core processors such as the MIT Raw processor offer first-class network support that exposes network resources at the ISA level, providing opportunities to interact with hardware resources in novel ways.

This paper presents rMPI, which leverages the on-chip network of multi-core processors to build an abstraction with which many programmers are familiar: the MPI programming interface. This study uses the MIT Raw processor as an experimentation and validation vehicle, although the findings presented are applicable to multi-core processors with on-chip networks in general. Likewise, this study uses the MPI API as a general interface which allows parallel tasks to communicate, but the results shown in this paper are generally applicable to message passing communication. Overall, rMPI’s design constitutes the marriage of message passing communication and on-chip networks, allowing programmers to employ a well-understood programming model to a novel high performance multi-core processor architecture. rMPI offers the following features: robust, deadlock-free, and scalable programming mechanisms; an interface that is compatible with current MPI software; an easy interface for programmers already familiar with high-level message passing paradigms; and fine-grain control over their programs when automatic parallelization tools do not yield sufficient performance.

This paper compares rMPI running on such a multi-core processor to hand-coded applications running on one of the processor’s low-level on-chip networks. rMPI is also compared to a commercial-quality MPI implementation running on a cluster of Ethernet-connected workstations. This paper evaluates various performance metrics such as latency, bandwidth, and performance scalability on a number of kernel benchmarks and applications. Results show that rMPI provides speedups of 4x to 15x for 16 processor cores, depending on the application, which equal or exceed performance scalability of the MPI cluster system. Furthermore, rMPI achieves overhead as low as 5% for real applications relative to hand-coded applications.

Keywords
On-Chip Networks, Multi-Core Processor Architectures, MPI, parallel computing, parallel programming, Message Passing

1. INTRODUCTION
Next-generation microprocessors will increasingly rely on parallelism, as opposed to frequency scaling, for improvements in performance scalability. Microprocessor designers are attaining such parallelism by placing multiple processing cores on a single piece of silicon, a feat now achievable thanks to the technology scaling described by Moore’s Law [3]. Current commercial multi-core processors such as the Power 5 and AMD Opteron 800 force interprocessor communication to go through the memory system, which can be slow. However, some multi-core processors offer first-class on-chip inter-core network support. Multi-cores with on-chip networks are analogous to traditional cluster-based systems in many ways: both offer processing nodes connected by a network. However, technology scaling is enabling such network-interconnected parallel systems to be built on a chip, offering users extremely low latency networks. The MIT Raw processor builds on this idea and provides a prototype of such a system to evaluate these ideas. Raw includes first-class instruction set architecture (ISA) support for inter-processor communication, enabling orders of magnitude improvement in communication latency.

As the architecture of modern computer systems evolves from single monolithic cores to multiple cores, its programming models simultaneously continue to evolve. Programming parallel computer systems has historically been quite challenging because the programmer must orchestrate both computation and communication. However, there has been a surge in new programming models and tools for multi-cores. A number of different models have evolved to help the programmer with this arduous task, from standardized shared-memory and message-passing application programming interfaces, to automatically-parallelizing compilers that attempt to achieve performance and correctness similar to that of hand-coded programs. One of the most widely-used standard programming interfaces today is the Message Passing Interface (MPI), which allows programmers to construct message passing programs that are portable across MPI-compliant parallel architectures.

This paper investigates the merits of tightly integrated on-chip networks, especially in light of their programmability. This paper introduces rMPI, which attempts to provide a scalable interface that
allows transparent migration of the large extant legacy code base which will have to run on multi-cores. rMPI leverages the on-chip network of the Raw multi-core processor to build an abstraction with which many programmers are familiar: message passing via the MPI programming interface. The processor cores that constitute multi-core architectures (MCAs) such as Raw are tightly coupled through fast integrated on-chip networks, making such MCAs quite different from more traditional heavily-decoupled parallel computer systems. Additionally, some MCAs eliminate many layers of abstraction between the user program and underlying hardware, allowing programmers to directly interact with hardware resources. Because of the removal of these layers, MCAs can have extremely fast interrupts with low overhead. Removing standard computer system layers such as the operating system both represents an opportunity for improved performance but also places an increased responsibility on the programmer to develop robust software. These and other novel features of multi-core architectures motivated designing rMPI to best take advantage of the tightly-coupled networks and direct access to hardware resources that many MCAs offer. Overall, rMPI offers the following features:

1. robust, deadlock-free, and high performance mechanisms with which to program MCAs
2. an interface to MCAs that is compatible with current MPI software
3. gives programmers already familiar with the message passing paradigm an easy interface with which to program MCAs
4. gives programmers fine-grain control over their programs when automatic parallelization tools do not yield sufficient performance

rMPI was implemented for the MIT Raw multi-core processor architecture [37, 35, 34, 36]. The Raw processor prototype embodies current trends in multi-core chip designs, while also providing a forward-looking on-chip network. Raw contains an array of 16 identical programmable tiles, each containing a MIPS-style processing pipeline, instruction and data caches, and communication routers for Raw's on-chip networks. Raw has a number of unique features that provide an interesting platform on which to implement MPI. For instance, Raw does not have the standard stack of computer system layers that user programs have to go through to interact with the hardware so user programs can interact with hardware directly. Because of this, coupled with the networks being on-chip, inter-core communication is very low latency. Furthermore, unlike on many other computer systems, interrupts on Raw are fast (less than 100 cycles). This paper discusses how these features were leveraged to implement message passing (compliant to the MPI standard) on Raw.

The evaluation of rMPI presented in this paper attempts to understand how well it succeeds in offering the above-mentioned features. rMPI is evaluated in comparison to two references. To develop a qualitative intuition about the scaling properties of rMPI, it is compared against LAM/MPI, a highly optimized commercial MPI implementation running on a cluster of workstations. Additionally, it is compared against hand-coded and hand-orchestrated applications running on the General Dynamic Network (GDN), one of Raw's low-level on-chip dynamic networks on top of which rMPI was built. The comparison against the GDN is an attempt to determine the overhead imposed by features that rMPI offers, which include the MPI programming interface, removal of sub-application-level deadlock potential, and automatic message packetization/reassembly. The sources of rMPI's overhead are determined by analyzing where cycles are spent in enacting both a send and a receive using MPI function calls for both short and long messages. Overall, we show that rMPI running on Raw can provide performance scalability that is comparable to a commercial MPI implementation running on a cluster of workstations by leveraging the underlying network architecture of the Raw processor. Additionally, rMPI's overhead relative to the GDN for real applications is as low as 5% when compared using cycle counts.

The rest of this paper is organized as follows. Section 2 provides an overview of the Raw architecture, focusing on the resources that are especially relevant to rMPI's design and operation. Section 3 discusses rMPI's design at a high level, and describes some of its optimizations. Section 4 provides detailed results. Section 5 discusses other work related to message passing on parallel computer systems. Finally, Section 6 concludes the paper.

2. BACKGROUND

2.1 Raw Processor

Before investigating rMPI's design and implementation, a brief overview of the Raw processor must be given. The Raw processor consists of 16 identical tiles, each contain a processing core and network components that allow for interprocessor communication. The processing cores each have an 8-stage in-order single-issue MIPS-style processing pipeline and a 4-stage single-precision pipelined FPU. The Raw chip also has four register-mapped on-chip networks which are exposed to the programmer through the Raw ISA. Additionally, tiles contain 32KB of hardware-managed data cache, 32KB of software-managed instruction cache, and 64KB of software-managed switch instruction memory. The Raw prototype was implemented in hardware with an ASIC process, and has been shown to perform well on a variety of application types [37].

Raw's software-exposed ISA allows programmers to directly control all of the chip's resources, including gates, wires, and pins. As the on-chip interconnection networks are also exposed through the Raw ISA, the programmer has the ability to carefully orchestrate data transfer between tiles simply by reading and writing registers. Raw has four 32-bit full-duplex on-chip networks, two static (routes specified at compile time) and two dynamic (routes specified dynamically at run time). The two static networks allow for scalar operands to be efficiently passed between tiles with no limit on simultaneous communication patterns that can be supported in a computation. The dynamic networks, on the other hand, handle all communication that occurs at run time, such as cache misses, interrupts, and dynamic messages. rMPI leverages one of Raw's dynamic networks, the General Dynamic Network (GDN), for all communication between tiles prompted by an MPI communication routine. The GDN is a suitable communication substrate for MPI, as MPI semantics require the capability to send and receive messages dynamically.

rMPI relies on several key features of the GDN, so discussion in some level of detail is important for understanding rMPI's design. The GDN is a dimension-ordered wormhole routed network on which messages containing 32-bit header words are sent. In addition to containing routing information, the GDN header contains a length field which is only 5 bits in length. Thus, the maximum GDN mes-
message size is 32 words\textsuperscript{1} including the header. The network guarantees that GDN messages arrive at the destination tile atomically and in-order. That is, the GDN guarantees that messages sent from a particular sender to a particular receiver will be received in the same order in which they were sent. However, GDN messages from different senders sent to the same receiver may be interleaved and received in a different order relative to the absolute time when each was sent. The GDN’s atomicity constraint does guarantee, though, that the contents of individual GDN messages from different sources will not be interleaved with each other.

If the sending tile must communicate more than 32 words, it must break the message into pieces, which then must be re-assembled by the receiver. Managing many senders and many receivers in an all-to-all communication pattern clearly becomes challenging using the low-level GDN. Additionally, it is trivial to construct a communication pattern on the GDN which deadlocks the network – the GDN’s input and output network ports are both blocking, and contain space for only four words and sixteen words of data, respectively. Thus, the programmer must construct programs which manage buffer space and communication patterns carefully to avoid deadlock.

Raw also offers programmers interrupts that take less than 60 clock cycles of overhead for both call and return. These interrupts are extremely fast relative to many modern computer systems, and allow programmers to use interrupts in situations where they may otherwise not. Raw’s GDN_AVAIL interrupt, for instance, can be used to trigger receiving from the GDN input queue of a particular tile. Given that the program’s control flow can switch to the interrupt handler so quickly, this interrupt can be used as an alternative to blocking reads from the network.

Such direct access to hardware resources provides a great amount of flexibility. However, orchestrating both computation and communication in a parallel computer can be an arduous task. In addition to the standard tasks that parallel programmers must carry out to implement a parallel algorithm, programmers must also effectively manage the low-level resources that the Raw ISA exposes. On Raw’s GDN, for instance, the programmer must break up large messages into small packets on the sender side, and reassemble them on the receiver side. Programmers also have to take care to avoid deadlock. rMPI implements the high-level abstractions that the MPI standard provides and frees up the programmer from such low level details. Section 3 discusses the high-level design of rMPI, including how it leverages the low-level resources that Raw provides.

2.2 MPI

In the parallel computing domain, MPI has become the de-facto standard for writing parallel applications. MPI is not an implementation or a language, but a standard with which implementations on different parallel computers must comply. Thus, programs written using MPI are portable: they can be moved from one parallel system to another, assuming both systems offer a valid MPI implementation. Overall, such portability is a key goal of MPI, providing a virtual computing model that hides many architectural details of the underlying parallel computer. MPI implementations exist for most high performance parallel computer systems, with LAM/MPI \cite{lam-mpi, mpich} and MPICH \cite{mpich, mpi} being two of the most popular. As mentioned previously, rMPI implements the MPI standard for the Raw processor.

\textsuperscript{1}A “word” here is an arbitrary 32-bit datatype.

The MPI standard \cite{mpi, mpi2} includes primitives for blocking and non-blocking point-to-point communications, collective operations (e.g., broadcast, scatter and gather), process groups, communication topologies, and bindings for C, C++, and Fortran. The MPI standard is large, containing over 200 function calls. rMPI implements the blocking point-to-point communication routines (but not the non-blocking routines), collective operations, MPI datatypes, process groups, and communication topologies for C programs. More information about the MPI standard is available in \cite{mpi2, meier, steve, steve2}.

Figure 1 shows a simple MPI user program where one process sends a short message to another process. Both processes run the same source code, and perform different tasks depending on their rank, or process number. The receiving process must explicitly specify which message it wishes to receive by specifying a message source, tag, and other parameters.

3. DESIGN

This section describes the design, architecture, and implementation of rMPI from a high level. Further detail is available in \cite{rmpi}. rMPI is a runtime system that enables users to run MPI programs on Raw. rMPI leveraged many ideas from well-known open source MPI libraries, such as MPICH \cite{mpich} and LAM/MPI \cite{lam-mpi, mpich}, but also attempted to implement the MPI standard in a way that leverages the unique resources that Raw provides. Indeed, multi-core processors with low-latency on-chip networks and fast interrupts serve as very different hardware platforms compared to a cluster of workstations interconnected via TCP/IP, and rMPI reflects these differences.

Figure 2 shows rMPI’s system architecture. Upon invoking an MPI routine, the user’s program calls into the high-level MPI layer, which implements the public MPI API functions. This layer is responsible for preparing the MPI request for processing by lower layers and handles tasks such as argument checking and data buffer management. This top layer also determines how to best utilize the low level communication routines, and directly invokes them for point-to-point communication. The point-to-point layer implements basic communication primitives through which all communication takes place. This layer interacts directly with Raw’s hardware mechanisms,
including reading and writing from the GDN network ports. The collective communication layer is invoked by the high level MPI layer for collective communications operations. This layer implements high-level collective communication algorithms such as broadcast and scatter/gather, and ultimately also calls down into the point-to-point layer for communication. The high-level MPI layer and collective communication layer leverage some of the code base from the LAM/MPI implementation, although much of it was re-implemented for rMPI.

To better understand the function of each of these layers, consider an example where the user’s program makes a call to MPI_Bcast. This call simply broadcasts the same message from the calling process (the root) to all processes in the specified communication domain (which, for this example, can be considered the entire process group of N processes). Note that all processes participating in the broadcast invoke MPI_Bcast – one process does the sending, and the rest receive their copy of the message. The high-level MPI layer implements this function’s public API, and checks arguments and packs data buffers if strided datatypes are used. This layer then calls into the collective communications layer, which performs one of two actions depending on the caller’s rank. When the caller is the root process, this layer invokes the point-to-point send routine, receiving a message from the root. The point-to-point layers deal with actually sending and receiving the messages. When all messages have been sent and received, the call to MPI_Bcast returns.

The point-to-point layer interacts with Raw directly, and is responsible for sending and receiving all data between tiles. As alluded to in Section 2, rMPI takes care of breaking up messages larger than 32 words into packets that are sequentially injected into the network as GDN messages. It does this by prepending an rMPI header to each packet to encode metadata required by the MPI standard (e.g., tag, size, etc.) and also so the receiver can associate incoming packets with logical MPI messages. Figure 3 shows how a logical message with 65 payload words is broken up into three packets, each with appropriate headers as just described. Each receiving tile registers all outstanding receive requests, and is therefore able to reassemble messages using the minimal rMPI headers. The rMPI header length was heavily optimized to improve effective network bandwidth for message payloads; the first packet of any logical MPI message includes a four word header (source, tag, length, communication context), and subsequent packets just contain a one-word header which encodes the source of the message. Such small headers attempt to mitigate MPI overhead relative to the GDN for short messages, and maximize the benefit of Raw’s low-latency on-chip networks.

Messages are received in the point-to-point layer by using Raw’s fast interrupt handler routine. That is, when a message arrives on the GDN input port of a tile, an interrupt fires, and the receiving tile’s control flow immediately jumps into the interrupt handler. The interrupt handler proceeds to drain the GDN network, keeping track of header information and sorting packet payloads appropriately. The interrupt handler is arguably the most complex component in rMPI. As packets can be received over multiple invocations of the interrupt handler, and packets from multiple senders can be interleaved, the interrupt handler must carefully sort them out and keep track of the status of each incoming message. Further, the interrupt handler must share some of its data structures with the rest of the system, as the user-level thread must be able to access the buffer where the handler stores messages.

An interrupt-driven design was chosen over a standard blocking receive design for a number of reasons. First, an interrupt-driven design allows MPI programs to make asynchronous progress on both communication and computation. Messages are received as soon as they arrive, and otherwise each processor can continue computation. The interrupt-driven design also reduces the potential for network congestion, since Raw’s internal network buffering is minimal (4 words per input and output port per tile) and sends are blocking. Because of this, deadlock could easily occur in a blocking receive design. Since the interrupt handler always drains the network of its contents immediately, deadlock can not occur at the network level\(^2\). The interrupt-driven design may not have made sense in some other contexts where interrupts must go through the operating system and are therefore slow, but Raw’s interrupts take on the order of 60 cycles of overhead, and therefore made sense in this context. Finally, the interrupt-driven design was straightforward from an algorithmic standpoint; all tiles are able to continue computing unless data is available to be received, thereby always allowing forward progress to be made.

A number of optimizations in the interrupt handler improved the overall performance of rMPI. In the case where a particular mes-

---

\(^2\)Of course, application-level deadlock could still occur with a poorly-constructed program.


<table>
<thead>
<tr>
<th>Message Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 word</td>
<td>2.47</td>
</tr>
<tr>
<td>100 words</td>
<td>1.80</td>
</tr>
<tr>
<td>10000 words</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table 1: Speedup resulting from posted receive entry optimization for various message sizes. This optimization drastically reduces bookkeeping overhead and memory copy operations by copying incoming messages directly into the receiver's user-level buffer.

message arrives at a tile before the MPI receive call for that message was called by the user program, the interrupt handler must buffer the message contents. However, when the user program requests a message before it arrives, rMPI registers a posted receive entry with the user’s buffer address. When the message finally arrives, the interrupt handler places the payload directly into the user’s buffer instead of storing it into temporary memory, thereby eliminating a memory copy operation, which can be quite expensive for very large messages. Table 1 shows the speedups that this optimization yields for send/receive pairs for various message sizes. Note that this optimization significantly improves performance of small messages because the receiver does not have to instantiate the data structures and bookkeeping mechanisms that are normally necessary to buffer an unexpected message. Larger messages also benefit from not having to perform a memory copy operation. Raw’s direct access to low-level hardware resources made such optimizations straightforward to implement.

rMPI also optimizes for the case where a process sends to itself by circumventing the typical data structures and network hardware that are normally traversed. In the send-to-self case, the message payload is simply copied into the appropriate location at the time of the receive call. rMPI also optimizes zero-payload messages by circumventing all of the logic and data structure initialization that typically occurs for messages with payloads. This optimization is useful in cases where users send short sentinel messages to signal completion of a particular task, a common occurrence in many MPI programs.

4. EVALUATION AND ANALYSIS

This section presents experimental results that show rMPI provides a scalable and low overhead MPI implementation for the Raw architecture. It discusses various performance metrics such as latency, bandwidth, and performance scalability on a number of kernel benchmarks and applications. This section also analyzes the overhead that implementing the MPI standard imposes relative to the underlying GDN. The evaluations use two bases for comparison in evaluating rMPI: hand-programmed native GDN, running on Raw, and LAM/MPI, running on a Beowulf cluster of workstations.

4.1 Methodology

We focus on evaluating the MPI-compliant rMPI library running on the Raw processor in an attempt to investigate how well MPI maps to multi-core architectures. Throughout this section, performance is presented using results collected from programs written using the rMPI library and from hand-coded programs running on the general dynamic network (GDN) running on Raw. Additionally, programs written using the LAM/MPI library running on a 128-node Beowulf cluster were used in this comparison. While tedious and time-consuming to write, hand-coded GDN programs generally provide a performance upper-bound for a given algorithm that uses the Raw GDN. Thus, experiments comparing rMPI programs to native GDN programs offer an assessment of rMPI’s overhead and performance scalability. The experiments that compare rMPI programs to MPI programs running on a cluster of Ethernet-interconnected workstations also give insight into the scalability of rMPI, but relative to a drastically different hardware platform using a different MPI implementation. The combination of the GDN and cluster comparisons provide a thorough look at rMPI’s performance characteristics.

The Raw architecture does not have a traditional hardware instruction cache; the instruction cache is managed in software. While Raw’s software-managed instruction caching system provides an array of benefits and much flexibility, it has not yet been optimized, and therefore had a degrading effect on the performance of rMPI programs because of the rMPI library’s large size. Thus, the results presented in this section were collected using the Raw simulator with a 256kB instruction cache (as opposed to Raw’s normal instruction cache size of 32kB), large enough to obviate Raw’s software instruction caching system. While all of the presented experiments were verified to run on the Raw hardware, removing the effects of the software instruction caching system leveled the playing field for comparison with the native GDN, which does not need software instruction caching. Furthermore, running on the simulator provided the added benefit of debugging support and deterministic results across simulations. In the experiments comparing rMPI to the native GDN, performance was evaluated using raw clock cycles.

One of the most common parallel processing platforms today that is used to run MPI programs is a “Beowulf” cluster: a collection of workstations interconnected by some commodity interconnect such as Ethernet. As such, this section compares the scalability of MPI programs running on rMPI and on LAM/MPI. The results presented here were collected on a cluster containing 128 nodes, each containing a 2GHz 64-bit AMD Opteron processors. Each node contains 4GB of memory, and they are connected using 10GB/sec Ethernet over TCP/IP. For this experiment, only one processor from each node was used at any given time, forcing inter-node communication to always occur over the Ethernet connection as opposed to between two processors inside a particular node. The speedups of four MPI applications were computed relative to a single processor on each respective platform running a serial implementation of each application. In other words, the rMPI speedup was computed relative to a single Raw tile running the serial version of a particular application, and the LAM/MPI speedup was computed relative to a single processor from the cluster node running the serial version of the same application. The same serial and MPI source codes were run on both platforms, and the speedups were calculated using cycle counts. While the cluster’s hardware is highly optimized and contains a more modern memory system and process technology, evaluating speedups relative to single-processor performance seemed to be a valid basis for comparison. Normalizing each platform by a single processor of itself essentially removed many system-dependent variables such as processor clock frequency and memory system performance, so the evaluation could focus on the scalability of the system’s interconnects and software.

4.2 End-to-End Overhead Evaluation

While sophisticated parallel programs can be developed using rMPI much more easily than by hand-coding programs using the low-level GDN, developers would most likely not use rMPI if its overhead was prohibitively high. However, experiments show that for
message sizes ranging from 100 words to 10 million words in length, the overhead of rMPI is reasonably low. Figure 4 shows the overhead of messages of length 1 word to 10 million words transmitted from one tile on the Raw chip to an adjacent tile for an rMPI application. The overhead was calculated using total latency of the rMPI send/receive pair relative to a hand-coded send/receive pair on the native GDN. The rMPI overhead for very short messages is, as expected, quite high; rMPI’s overhead is 481% for a one-word message, and 230% for a ten-word message. As the message size grows from 10 to 100 words, however, the difference of end-to-end latencies narrows; rMPI’s overhead is only 32% for a 10000-word message. The overhead for a 100000-word message jumps up to 60%, most likely because rMPI’s receiving logic and data structures tax Raw’s memory system, causing extra cache misses.

In an attempt to further understand the cause of rMPI’s overhead, experiments were run to capture where rMPI spends its time during an MPI_Send and MPI_Recv for the same single-sender, single-receiver latency test. Table 2 shows the resulting cycle breakdowns for the one-word message length case and 10000-word message length case for MPI_Send. This data makes it clear why rMPI performs so poorly for very small message sizes relative to the GDN. For the 1-word case, an overwhelming 67% of the time is due to argument checking (to preserve MPI semantics) and function call overhead, and another 5.5% is spent constructing the rMPI message envelope that precedes all MPI messages. However, both of these actions only occur once in any MPI_Send call. Hence, virtually all of the time in sending a 10000-word message is spent breaking the messages into packets and pushing them out to the GDN. Thus, the fixed overhead due to MPI_Send is amortized for large message sends, explaining the overhead drop from 481% for a one-word message to 32% for a 10000-word message.

Similarly, Table 3 shows the resulting cycle breakdowns for the one-word message length case and 10000-word message length case for MPI_Recv. In the one-word message case, 33% of the cycles were spent calling into the rMPI library and checking arguments (to uphold the MPI specification), and over 50% was spent managing bookkeeping data structures and sorting packets. About 8% of the time was spent calling into the interrupt handler (saving and restoring registers, etc.), and only 7% of the time was spent actually receiving and storing data from the network. Contrasting this to the 10000-word message case, it is once again clear that some of the overhead is fixed argument checking and calling in and out of the interrupt handler accounts for about 0.1% of the total cycles. Over one third of the time is spent receiving data from the network, and nearly two-thirds of the cycles are spent managing the packet sorting and data structure bookkeeping. In fact, only 0.02% of cycles were spent managing data structures before the data was available, and only 0.04% of cycles were spent managing data structures after the message was completely received. Thus, sorting and demultiplexing packets while they are arriving consumes a relatively large portion of the time in an MPI_Recv.

Finally, it is important to recognize that the overhead of MPI_Send is largely fixed, and it is therefore able to be amortized for a large message. On the other hand, MPI_Recv has both fixed and variable overhead; the argument checking, function call overhead, and bookkeeping before and after the interrupt handler can all be amortized over the receive of a large message, but the packet sorting and bookkeeping during the interrupt scales with the size of the message.

### 4.3 Performance Scaling

To investigate the performance scalability of rMPI, a number of experiments were run for four parallel applications using rMPI, the native GDN, and LAM/MPI. In these evaluations, speedup was computed relative to a serial version of the application running on a single processor of the respective system. This normalization allowed comparison of performance scalability despite the drastically different hardware specifications.

#### 4.3.1 Jacobi Relaxation

The jacobi relaxation algorithm was evaluated on all three platforms. While programming this application using the native GDN directly was tedious and time-consuming, it is seen as an upper bound for performance scalability because of its extremely low overhead for sending and receiving. The algorithm was run on 2-dimensional

---

**Table 2:** Cycle breakdown for an MPI_Send for 1 word and 10000 words for three primary actions. Presented as a fraction of total cycles used to enact send.

<table>
<thead>
<tr>
<th>Action</th>
<th>1 word</th>
<th>10000 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument Checking</td>
<td>67.26%</td>
<td>0.09%</td>
</tr>
<tr>
<td>Envelope Construction</td>
<td>5.52%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Sending headers and payload, packetization</td>
<td>27.22%</td>
<td>99.90%</td>
</tr>
</tbody>
</table>

**Table 3:** Cycle breakdown for an MPI_Recv for 1 word and 10000 words for four primary actions. Presented as a fraction of total cycles used to enact receive into user’s buffer.

<table>
<thead>
<tr>
<th>Action</th>
<th>1 word</th>
<th>10000 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiving from network</td>
<td>7.29%</td>
<td>36.16%</td>
</tr>
<tr>
<td>Call into and out of interrupt handler</td>
<td>7.75%</td>
<td>0.02%</td>
</tr>
<tr>
<td>Argument Checking, function call overhead</td>
<td>32.82%</td>
<td>0.09%</td>
</tr>
<tr>
<td>Bookkeeping and packet sorting</td>
<td>52.14%</td>
<td>63.74%</td>
</tr>
</tbody>
</table>
floating point matrices ranging from $16 \times 16$ to $2048 \times 2048$. Figure 5 shows the results for the $16 \times 16$ case. With very small input data sizes, the low overhead GDN is the only configuration that actually yields a speedup. rMPI and LAM/MPI are both slower than the serial version of the code for this input data size. In fact, LAM/MPI slows down more when 8 and 16 processors are used. The slowdown for rMPI and LAM/MPI is caused by the low computation-to-communication ratio; there is simply not enough computation to effectively amortize the cost imposed by the MPI semantics. On the other hand, the extremely low overhead of the GDN can achieve a non-trivial speedup despite the small input matrix.

For an input matrix size of $2048 \times 2048$, the three configurations scale very similarly, as seen in Figure 6. This is congruent with intuition: the low-overhead GDN outperforms both MPI implementations for small input data sizes because its low overhead immunizes it against low computation to communication ratios. However, the MPI overhead is amortized over a longer running program with larger messages in this case. rMPI even outperforms the GDN in the 16 processor case, which is most likely a due to memory access synchronization on the GDN, as the GDN algorithm is broken into phases in which more than one processor accesses memory at the same time. Contrastingly, the interrupt-driven approach used in rMPI effectively staggered memory accesses, and in this case, such staggering provides a win for rMPI.

Figure 7 summarizes the speedup characteristics for the GDN, rMPI, and LAM/MPI. Again, the GDN achieves a 3x speedup immediately, even for small input data sizes. On the other hand, both MPI implementations have slowdowns for small data sizes, as their overhead is too high and does not amortize for low computation to communication ratios. rMPI starts to see a speedup before LAM/MPI, though, with an input data matrix of size $64 \times 64$. One potential reason rMPI exhibits more speedup is its fast interrupt mechanism and lack of operating system layers through which to go.

One clearly interesting input data size for the GDN and rMPI graphs is $512 \times 512$: both show a significant speedup spike. Figure 8, which shows the throughput (computed elements per clock cycle) of the serial Jacobi implementation running on Raw, sheds some light on why this speedup jump occurs. Up until the $512 \times 512$ input size, the entire data set fit into the Raw data cache, obviating the need to go to DRAM. However, the $512 \times 512$ data size no longer fit into the cache of a single Raw tile. Hence, a significant dip in throughput occurs for the serial version for that data size. On the other hand, since the data set is broken up for the parallelized GDN and rMPI versions, this cache bottleneck does not occur until even larger data sizes.

Figure 5: Speedup (in cycles) for Jacobi Relaxation for a $16 \times 16$ input matrix. Speedups shown for application using GDN running on Raw, rMPI running on Raw, and LAM/MPI running on a cluster of workstations. The GDN and rMPI were compared against serial version on Raw, and LAM/MPI was compared against serial version running on a single compute node of the cluster.

Figure 6: Speedup (in cycles) for Jacobi Relaxation for a $2048 \times 2048$ input matrix. Speedups shown for application using GDN running on Raw, rMPI running on Raw, and LAM/MPI running on a cluster of workstations. The GDN and rMPI were compared against serial version on Raw, and LAM/MPI was compared against serial version running on a single compute node of the cluster.

Figure 7: Speedup (in cycles) for Jacobi Relaxation for various input matrix sizes. The x-axis shows N, where the input matrix is $N \times N$. Speedups shown for application using GDN running on Raw, rMPI running on Raw, and LAM/MPI running on a cluster of workstations. The GDN and rMPI were compared against serial version on Raw, and LAM/MPI was compared against serial version running on a single compute node of the cluster.
sizes, which explains the jump in speedup. It should be noted that this distributed cache architecture evident in many multi-core architectures can be generally beneficial, as it allows fast caches to be tightly coupled with nearby processors.

Figure 9 characterizes the overhead of rMPI for a simple send/receive pair of processors on the Raw chip. To characterize rMPI's overhead for real applications, which may have complex computation and communication patterns, the overhead of rMPI was measured for jacobi. The experiment measured the complete running time of jacobi experiments for various input data sizes and numbers of processors for both the GDN and rMPI implementations. Figure 9 shows the results of this experiment. Here, rMPI overhead is computed as follows:

$$\text{overhead}_{\text{rMPI}} = \frac{\text{cycles}_{\text{rMPI}}}{\text{cycles}_{\text{GDN}}} - 1$$

As can be seen, rMPI's overhead is quite large for small data sets. Furthermore, its overhead is particularly high for small data sets running on a large number of processors, as evidenced by the N=16 case for 2 processors, which has an overhead of nearly 450%. However, as the input data set increases, rMPI's overhead drops quickly. It should also be noted that for data sizes from 16 × 16 through 512 × 512, adding processors increases overhead, but for data sizes larger than 512 × 512, adding processors decreases overhead. In fact, the 1024 × 1024 data size for 16 processors has just a 1.7% overhead. The 2048 × 2048 for 16 processors actually shows a speedup beyond the GDN implementation. This is most likely due to memory access synchronization, as described above, and does not generally hold true for all applications.

4.3.2 Scalability of Other Applications

In an attempt to further characterize the performance scalability of rMPI, three other applications were run. The applications were run using both rMPI on Raw and LAM/MPI on the cluster. As before, the sequential versions of each application were run on each respective architecture to determine baseline performance. Note that a GDN version of these applications was not evaluated, as developing non-trivial applications on the GDN is quite time-consuming and tedious, and the above experiments with jacobi served as a solid characterization. This is most likely due to memory access synchronization, as described above, and does not generally hold true for all applications.
a one-word message send/receive is roughly 100 cycles on rMPI. However, the shape of the LAM/MPI latency curve is notably different, as can be seen in Figure 11. First of all, the overhead for sending a single-word message from one process to another on LAM/MPI is quite high, taking almost 1 million cycles to complete. This can be attributed to the many layers of abstraction that LAM/MPI must go through to access the network. Additionally, the latency to send and receive a 10000-word message is roughly the same as sending a 1-word message. Not until message sizes reach 100000-words and above is the round trip latency affected.

This empirical result explains why rMPI's speedup is greater than LAM/MPI's speedup for large numbers of processors. In the 16 processor case, the system had to send 8 times as many messages compared to the 2 processor case. On the one hand, rMPI's latency increases proportionately with increasing message sizes, so it scaled well as more processors were introduced. On the other hand, the latency for sending smaller messages on LAM/MPI is not much different than the latency for sending 10000-words. Thus, while rMPI reaps the savings of smaller messages which ensue in the 16 processor case, LAM/MPI's large fixed overhead is the same and therefore does not benefit from smaller messages.

Figure 12 and Figure 13 show the speedups for a parallelized trapezoidal integration application and a parallelized pi estimation algorithm, respectively. Both applications are similar in terms of their computation-to-communication ratios, which are both quite large relative to an application such as jacobi relaxation. As can be seen in the figures, LAM/MPI generates a larger speedup for a small number of processors, but rMPI has larger speedups for larger numbers of processors. rMPI's performance on trapezoidal with 16 processors is over double LAM/MPI's speedup. This result can again be explained by the argument given above for the matrix multiplication application. In these cases, the messages are quite small (30 words or less), so sending many more, as in the 16 processor case, affects LAM/MPI's performance more drastically than it does rMPI's.

In general, though, both applications achieve speedups ranging from approximately 6x – 14.5x on both rMPI and LAM/MPI. These speedups are larger than that of matrix multiply and jacobi, which algorithmically have significantly lower computation-to-communication ratios. The results for all four applications evaluated agree with intuition: rMPI and LAM/MPI both exhibit better performance scal-

5. RELATED WORK
There have been a large number of other projects relating to message passing communication on parallel computer systems, many of which have influenced this work. The iWarp system [6, 17], designed and built jointly by Carnegie Mellon University and Intel Corporation attempted to integrate a VLIW processor and fine-grained communication system on a single chip. iWarp supported message passing communication between processor elements that contained elements similar to rMPI, including support for dynamically-generated messages where routes were specified at runtime, and which were received asynchronously by the message recipient. The INMOS transputer [5] was another early parallel microprocessor developed in the 1980s whereby computing elements could send messages to one another. The MIT Alewife machine [21, 20, 19] also contained support for fast user-level messaging.

A number of parallel multiprocessors on a chip have recently emerged to take advantage of the increased amount of silicon processor de-
signers have at their disposal. Other multi-core microprocessors include VIRAM [18], Wavescalar [33], TRIPS [37], Smart Memories [25, 26], and the Tarantula [8] extension to Alpha. Some commercial chip multiprocessors include the POWER 4 [9] and Intel Pentium D [2]. This work is applicable to many newer architectures that are similar to Raw in that they contain multiple processing elements on a single chip. rMPI should provide useful insights to users of these machines, hopefully encouraging them to implement their own fast messaging interface.

This paper primarily concentrated on Raw's dynamic network, but much work has been done using Raw's static network, which operates on scalar operands. Prior work [37, 13] shows considerable speedups (up to two orders of magnitude for some applications) can result using the static network for stream computation. Additionally, there exist a number of compiler systems for Raw that automatically generate statically-scheduled communication patterns, allowing the programmer to develop applications at a high level. CFloW [14], for instance, is a compiler system that enables statically-scheduled message passing between programs running on separate processors. Raw's rawcc [22, 23] automatically parallelize C programs, generating communication instructions where necessary.

Finally, there exist a large number of MPI implementations for a variety of platforms and interconnection devices. MPICH [38], LAM/MPI [7], and OpenMPI [12] are general MPI implementations that can be used on a variety of systems, employing Ethernet, InfiniBand [1], Myrinet [4], RDMA [24] and a variety of other communication substrates. MPI currently has a lot of momentum in the high-performance computing industry, and will most likely find its way into new computer systems as they develop.

6. CONCLUSION
This paper presented rMPI, an MPI-compliant message passing library for multi-core architectures. rMPI introduces robust, deadlock-free, and high-performance mechanisms to program multi-cores, offering an interface that is compatible with current MPI software. Likewise, rMPI gives programmers already familiar with MPI an easy interface with which to program Raw which enables fine-grain control over their programs. rMPI was designed to leverage the unique architectural resources that a multi-core processor architecture with on-chip networks and direct access to hardware resources provides. A number of evaluations using the rMPI implementation show that its overhead relative to the native on-chip network is low for real applications, and that it allows performance scalability for real applications that is comparable to both the on-chip network and a cluster of workstations running a commercial-quality MPI implementation. The authors hope that this work provides guidance and useful insights to application developers of future multi-core processors.

7. ACKNOWLEDGEMENTS
Thanks to Jeff Squyres for his support, and the Indiana University Computer Science Department for access to their cluster. Also, thanks to Rodric Rabbah for support with results analysis, and Jason Miller and Alexey Radul for helpful feedback.

This work was supported by an NDESG Graduate Fellowship. The Raw chip was fabricated in cooperation with IBM. The Raw project is supported by Darpa, NSF, ITRI, and the Oxygen Alliance.

8. REFERENCES


