The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer

Sameer Kumar
sameerk@us.ibm.com

Gabor Dozsa
gdozsa@us.ibm.com

Gheorghe Almasi
gheorghe@us.ibm.com

Dong Chen
chendong@us.ibm.com

Mark E. Giampapa
giampapa@us.ibm.com

Philip Heidelberger
philiph@us.ibm.com

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA

Michael Blocksome
blocksom@us.ibm.com

Ahmad Faraj
faraja@us.ibm.com

Jeff Parker
jjparker@us.ibm.com

Joseph Ratterman
jratt@us.ibm.com

Brian Smith
smithbr@us.ibm.com

Charles Archer
archerc@us.ibm.com

IBM Systems and Technology Group
Rochester, MN, 55901

ABSTRACT

We present the architecture of the *Deep Computing Messaging Framework* (**DCMF**), a message passing runtime designed for the Blue Gene/P machine and other HPC architectures. DCMF has been designed to easily support several programming paradigms such as the Message Passing Interface (MPI), Aggregate Remote Memory Copy Interface (ARMCI), Charm++, and others. This support is made possible as DCMF provides an application programming interface (API) with active messages and non-blocking collectives. DCMF is open source software that has a layered component based architecture with multiple levels of abstraction, allowing the members of the community to contribute new components to its design at the various layers. The DCMF runtime can be extended to other architectures through the development of architecture specific implementations of interface classes. The production DCMF runtime on Blue Gene/P takes advantage of the direct memory access (DMA) hardware to offload message passing work and achieve good overlap of computation and communication. We take advantage of the fact that the Blue Gene/P node is a symmetric multi-processor with four cache-coherent cores and use multi-threading to optimize the performance on the collective network. We also present a performance evaluation of the DCMF runtime on Blue Gene/P and show that it delivers performance close to hardware limits.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Languages, Measurement, Performance

Keywords

Parallel computing, Message passing, Application Programmer Interface, MPI, Blue Gene, Collective Communication, Active Messages, Sockets, ARMCI, Charm++

1. INTRODUCTION

The Blue Gene/L [2] (BG/L) machine is a low-power highly scalable supercomputer that has achieved nearly 480 TF on the LINPACK benchmark [7]. Blue Gene/P [10] (BG/P) is the successor to Blue Gene/L and can scale to over 3 PF of peak performance. Both Blue Gene machines have low-frequency embedded cores and multiple network devices for application message processing.

In this paper, we present the architecture of the Deep Computing Messaging Framework (DCMF). This framework is the product message passing runtime on BG/P. The main contributions of this framework are:

- a minimalist application programming interface (API) with active messages, one-sided operations, non-blocking collectives and explicit support for distributed memory consistency and ordering requirements
- a generalized messaging stack that supports many programming paradigms and contexts simultaneously,
- features that enable low-latency, high throughput API calls for architectures with low-frequency cores and multiple network devices,

- high performance message passing on BG/P.

The DCMF runtime is a generalization of the message layer [3] stack on BG/L that was initially designed to support the Message Passing Interface (MPI) [8]. The message-layer stack was extended to support the Aggregate Remote Memory Copy Interface (ARMCI) [15] and Global Arrays. Since then, there have been research explorations to study programming paradigms such as Charm++ [13] and Unified Parallel C [5] on the BG/L machine. This has motivated the design of a new application programming interface (API) to support different programming paradigms. The DCMF API is architecture independent and provides a minimalist set of message passing services, which include a point-to-point message send, non-blocking one-sided get and put operations, and an optional set of non-blocking collective calls. To support different programming paradigms such as Global Arrays, Charm++, and MPI, only a portability layer (e.g., the ADI in MPICH [17, 9] and the machine layer in Charm++) needs to be developed on top of the DCMF API.

A key feature of the DCMF API is the support for active messages (similar to LAPI [4] and Charm++/Converse [12, 11]), where the header packet carries the identifier of the handler function to be executed on the arrival of the header packet. This is different from a send-receive model where an explicit receive has to be posted for each send. However, in this work, we show that send-receive programming models can be developed with good performance on top of the active message model.

Programming paradigms often have different semantics for the various operations they support. For example, in ARMCI some operations can be synchronous (blocking) and require all succeeding operations to wait for that operation to finish. The DCMF architecture supports different consistency levels for these different messaging semantics. In addition, the different consistency levels can also enable different ordering schemes for the messages; for instance, since the Charm++ runtime does not require message ordering, it can take advantage of the relaxed consistency model allowing the DCMF runtime to send and receive messages out of order.

The Global Arrays runtime can issue both ARMCI and MPI calls resulting in different message handlers being invoked on the destination processors. This has motivated support for multiple programming contexts to co-exist at the same time in DCMF. This feature is more general than queues in the ELAN API [16] and queue-pairs in Infiniband VERBS [1]. The multiple contexts can enable ARMCI, MPI and even DCMF calls to be made in the application and be processed by the framework independent of each other.

An important feature of the DCMF design is supporting architectures with multiple network devices that have different interfaces. On BG/P, for example, there are three network devices: DMA, collective network, and global interrupt network. The DCMF runtime provides device classes to handle each network device independently and support any new network devices. This design facilitates porting DCMF to another architecture.

Similar approaches to DCMF have been used in the VERBS API for Infiniband [1], ELAN for Quadrics [16], Myrinet Express [14] and LAPI [4] in IBM SP architectures. Our approach differs from these as it has a unique set of features which include active messages, multiple programming contexts and consistency levels for messages. Moreover our stack is designed and optimized for architectures of low frequency cores and multiple network devices with different interfaces.

The BG/P DCMF runtime has been open-sourced [6] to allow users to program at any of its layers and explore a variety of message passing optimizations. The next sections detail the different levels of the DCMF stack and their implementations on BG/P.

2. DCMF ARCHITECTURE

We begin with a brief discussion on the DCMF application programmer interface (API).

2.1 DCMF API

The hierarchical structure of the DCMF runtime is presented in Figure 1. The majority of applications are expected to use MPI or other common middlewares supported by the stack (e.g. ARMCI or Converse/Charm++). The DCMF stack builds upon and co-exists with Lower Level Network APIs. Note that, for example, applications can simultaneously use MPI, ARMCI or Charm++, DCMF API and low-level network APIs to minimize overheads in latency critical regions.

The DCMF API serves as the primary interface for higher level messaging systems (or custom applications). This API defines a minimal set of functions to initialize, query, configure and utilize the communication hardware.

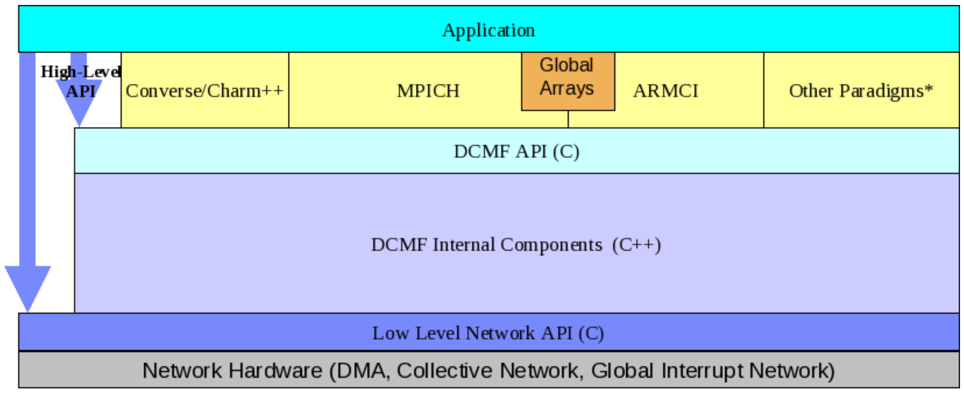
To ensure message progress, the DCMF API defines four possible thread levels that coincide with their MPI equivalents. Mutual exclusion is ensured for all critical sections within DCMF when the configuration requires it. The runtime system can also be configured to enable interrupts on packet arrival and have a dedicated lightweight communication thread make progress on messaging. Current MPICH implementation on Blue Gene/P takes advantage of interrupt support to be fully compliant with the MPI progress semantics.

The DCMF API exposes three basic types of message-passing operations: two-sided point-to-point send (DCMF_Send), one-sided put (DCMF_Put) and get (DCMF_Get), and multi-send. All three have non-blocking semantics to facilitate overlapping of computation and communication. The user is notified of completion of communication events through callback functions.

Figure 2 illustrates the messaging context and active message concepts of the DCMF API. It is summarized as follows.

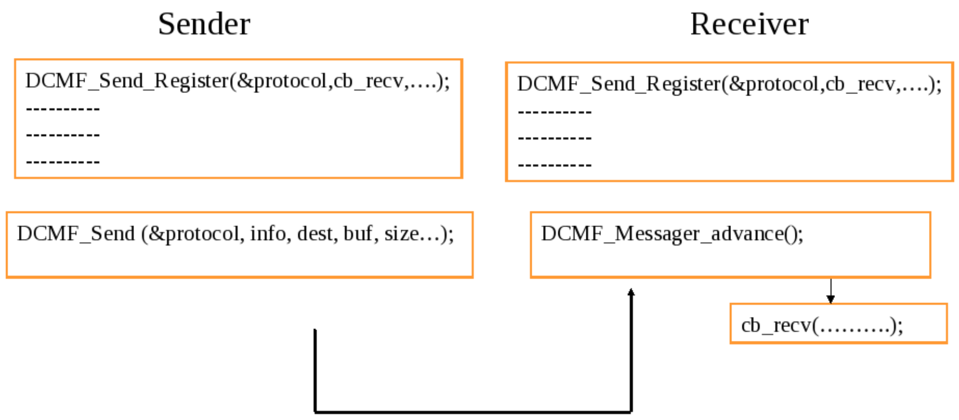
Multiple messaging contexts: To prepare a message transfer, the processes create messaging contexts. For two-sided communication, the context is created via the DCMF _Send_register() call . This call takes the reception callback (the cb_recv in Figure 2) as an argument and associates it with a new context stored in the opaque persistent protocol object. The context can then be used for initiating subsequent send operations. Multiple contexts can co-exist and be used simultaneously in the same application.

Active message model: Two-sided send operations can be initiated through the DCMF_Send call (see Figure 2), which takes the context and the message *info* argument. The latter can deliver meta-data information along with the payload to enable message matching at the receiver side. When the receiver is processing the first packet of the message, the DCMF runtime invokes the reception callback associated with the context and passes it the meta-data (the info



Application
High-Level API
Converse/Charm++
MPICH
Global Arrays
ARMCI
Other Paradigms*
DCMF API (C)
DCMF Internal Components (C++)
Low Level Network API (C)
Network Hardware (DMA, Collective Network, Global Interrupt Network)

Figure 1: Overview of the DCMF Messaging Stack



Sender
Receiver
DCMF_Send_Register(&protocol,cb_recv,….);
DCMF_Send_Register(&protocol,cb_recv,….);
DCMF_Send (&protocol, info, dest, buf, size…);
DCMF_Messager_advance();
cb_recv(……….);

Figure 2: Active message DCMF API for two-sided communication

argument) and the size of the message as parameters. The callback must return a buffer where the incoming message needs to be stored.

Multisend protocol: DCMF provides a novel multisend protocol. In a multisend, many point-to-point messages to different destinations can be sent through a single operation, thus amortizing the software startup overheads. This call is of high importance for low frequency core architectures. Multisend also enables network hardware specific optimizations for groups of messages (e.g., depositing packets along straight lines on the torus network). DCMF provides two flavors of the multisend protocol: multicast and many-to-many. Many-to-many allows different messages to be sent to different processes through a single API call. Multicast is a special case of many-to-many where the same message is delivered to all target processes.

Collective protocols: The API defines function prototypes for optimized non-blocking collective operations including broadcast, reduce/allreduce, barrier and all-to-all, most of which are implemented via multisend.

2.2 Internal Components of DCMF

DCMF has a component based design with four abstract base components (manifested as C++ classes): device, protocol, sysdep and messager as shown in Figure 3.

An abstract device component manages hardware and software resources associated with a corresponding network device. The abstract device layer in DCMF is important to support architectures which have several different communication networks. It also aids porting the system across various architectures. We have explored a DCMF design on the BG/L hardware where torus is accessed directly (i.e. there is no DMA hardware). When we moved to BG/P, we added a new abstract device class to handle DMA resources.

The messager object encapsulates devices used by a particular manifestation of DCMF. Each DCMF implementation can have its own messager implementation that manages only those devices which exist on the particular architecture. The messager initializes the devices and provides a unified advance method for the higher layers to ensure message progress.

Portability of DCMF is enhanced by the sysdep component, which provides an abstract interface for all kernel dependent services. These services include thread-management, intra-node synchronization, personality information lookup (e.g., local rank and global torus mapping), etc.

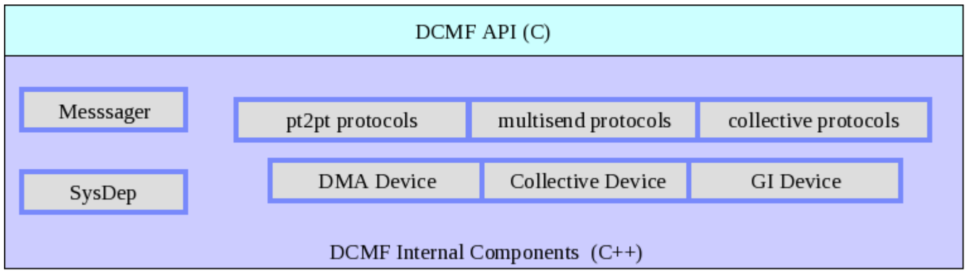


Figure 3: Overview of DCMF Basic Components

Messaging algorithms (like eager or rendezvous two-sided send) are implemented as protocol classes. The base protocol class provides the generic registration framework for managing multiple messaging contexts. A protocol object typically creates a few device dependent and interrelated message objects, and then posts them in the proper sequence to the appropriate device queues. Devices signal the protocol object through callbacks when posted messages complete. These callbacks may trigger creation and posting of more messages until the required data transfers have been completed.

The abstract device and protocol layers facilitate building protocol classes that use more than one device to implement complex messaging algorithms. The component based design of DMCF results in high extendability: novel algorithms and optimizations can easily be explored by adding new protocol objects on top of existing device and message classes.

A fully functional generic DCMF runtime is available via the sockets device, socket send protocol, and the PutOverSend and GetOverSend protocols. These components implement the core functionality of the DCMF runtime and provide a base for porting DCMF to other platforms. A new DCMF runtime implementation optimized to a particular network architecture simply needs to implement a network device and send protocol. The GetOverSend and PutOverSend protocols implement the DCMF_Get and DCMF_Put APIs through the DCMF_Send interface. This allows the implementor to optimize a new DCMF runtime implementation in stages without dropping functionality during the bringup process. The GetOverSend and PutOverSend are intended to ease portability to new architectures and not provide optimal performance for any specific platform.

The sockets version of the DCMF runtime allows portability to generic platforms such as Linux, Mac OS X, Z/OS, or other architectures. The reason to port to the sockets platform is twofold. The first is that it is a test environment for the DCMF developers to build and test protocols and algorithms in a portable, low cost, highly available environment. Second, users of the DCMF API who do not have ready access to Blue Gene hardware can test their application code on a non-Blue Gene environment such as workstation Linux.

The sockets version of the code currently only scales to a small number (32 process ranks) of nodes, but it can be used as a proof of concept for the portability of the platform and to test the APIs. The result is that we have an rDMA implementation over sockets, as well as the traditional send/recv model with active messages, that will work on virtually any workstation environment.

This "DCMF over sockets" API currently has been tested and passes all the MPICH test suite, using the same DCMF code as the Blue Gene version, so it can be used as an underlying transport for MPI applications. While the Blue Gene version of DCMF uses the compute node kernel and it's facilities for process management, the sockets version uses a portable process manager interface based on the MPICH PMI interface. This allows for portable and scalable job launch in a cluster environment. In addition, the sockets version works in a 64 bit environment as a test vehicle as we port DCMF to 64-bit architectures.

3. DCMF ON BLUE GENE/P

In this section, we discuss the DCMF implementation on BG/P, starting with a brief overview of the BG/P hardware. Then we focus on the device and protocol components.

3.1 BG/P Hardware Architecture

The Blue Gene/P node has four 850 MHz embedded PowerPC 450 cores on a single ASIC and can achieve a peak floating point throughput of 13.6 GF/node. The software stack supports three modes: symmetric multi-processing mode (or SMP mode) with one process and up to four threads, dual mode with two processes, each with up to two threads and quad mode (also known as virtual node mode, or VN mode) with four processes. Similar to BG/L, BG/P nodes are connected with three networks that the application may use: a 3D torus network that is deadlock-free and provides reliable delivery of packets, a collective network which implements global broadcast and global integer arithmetic operations, and a global interrupt network for fast barrier synchronizations.

On the 3D torus, packets are routed on an individual basis with either deterministic or adaptive routing. With deterministic routing, all packets between a pair of nodes follow the same path along x,y,z dimensions in that order. With adaptive routing, each packet can choose a different minimalist path based on the load on the torus router ports. The raw throughput of each link is 425 MB/s, while the achievable throughput is about 374MB/s due to hardware and software header overheads [10]. The BG/P architecture also adds a Direct Memory Access (DMA) engine to facilitate injecting packets to the network and receiving packets from the torus network. This allows the cores to offload packet management and enables better overlap of communication and computation. The DMA is also used for local intra-node memory copies.

The collective network provides reliable delivery at a raw throughput of 850 MB/s, while the achievable peak throughput is about 824 MB/s [10]. On BG/P, the collective network cannot be accessed through the DMA engine. So, the cores must handle packet management. This was a design choice for BG/P mainly based on complexity, resources, and verification requirements for enabling the DMA for the collective network. In addition, the collective network operates on the global communicator (MPI_COMM_WORLD) and, collective calls are all blocking operations in the current MPI standard. This suggested that there was little to be gained from DMA access to the collective network.

The major constructs in the DMA are injection and reception memory FIFOs (first-in, first-out buffers), injection and reception byte counters, and message descriptors. There are 32 Injection FIFOs, eight reception FIFOs, 64 injection and 64 reception counters per DMA group, and four groups per node.

There are three message types: memory FIFO, direct-put, and remote-get. For a memory FIFO message, the packets (including torus headers) are placed in a reception FIFO on the destination node. For a direct-put, the payloads of the packets are deposited directly into an arbitrary memory buffer whose physical address is known to the source. For a remote-get, the payload of the message is one or more message descriptors that are placed into an injection FIFO on the destination node to be eventually processed by the DMA. For example, the payload descriptor could be a direct-put of a long message back to the source node.

Each message must have an injection counter, and all direct-put messages also have a reception counter. The counters keep track of the number of bytes of the message sent or received. The DMA decrements counters when packets are injected or received. To test for message completion, software can poll a counter to see if it has reached an appropriate value (usually 0) or the DMA can be programmed to trigger an interrupt when a counter hits zero. Additionally, software can examine the DMA head pointer to see if a message has been completely injected. This feature can enable several thousands of messages to be in flight at the same time (see Section 3.2.1).

3.2 DMA Device: Managing DMA Resources

The software device layer in the DCMF stack manages the hardware resources and presents message passing primitives to the higher layers. This device uses the DMA System Programming Interface (SPI) [10] calls to allocate and manage resources in the DMA hardware. It has components for the different functionalities of the DMA hardware. It is event-driven with callbacks registered for each DMA event. The components InjFifoGroup and RecFifoGroup use SPI calls to program and manage DMA hardware injection and reception FIFOs. The InjFifoGroup defines a push method to inject descriptors in the DMA. The RecFifoGroup uses an SPI routine to poll the DMA reception FIFOs for packets. The poll routine calls the registered handlers for the packets in reception FIFOs.

The CounterGroup initializes counters and maintains a list of free and in-use counters. For each hardware DMA counter there is a software counter-client associated with it. When the counter hits zero, a callback is called in the counter-client. The DMA device categorizes counters as shared or exclusive. A shared counter can have several clients that are notified when the counter hits zero, while an exclusive counter can have only one counter-client.

As counters are a limited resource, sharing counters can allow a very large number of messages to be sent.

The DMA software provides message classes, which are the basic building blocks for high-level protocols. These message classes create descriptors and call the InjFifoGroup object to inject them on the DMA. Some of these message classes are :

1. *SendMessage* injects a memory FIFO descriptor with a shared injection counter, tracks completion by observing the DMA head and calls a callback on completion. If the injection FIFO is full, it buffers all input parameters and retries when the FIFO has space.

2. *PutMessage* injects a direct put descriptor with a shared injection and a shared reception counter. It tracks completion on the source by observing the DMA head and then calls a callback when the data has been fully injected on the network. If the injection FIFO is full, it buffers all input parameters and retries when the FIFO has space.

3. *GetMessage* provides a basic remote-get functionality by injecting a remote-get descriptor with a direct-put descriptor as the payload. It uses an exclusive reception counter on the source node, and a shared injection counter on the source and destination nodes. When resources such as a reception counter and space in the injection FIFO are unavailable, input parameters are buffered, and the remote-get is re-initiated when the resources become available.

4. *MulticastMessage* is more general than the send as it allows the message to be sent to many destinations. This message allocates an exclusive injection counter which hits zero when the data is multicast to all destinations.

5. *ManyToManyMessage* is similar to multicast as data is sent to several destinations, but different data is sent to each destination. It also allocates an exclusive injection counter to track completion of the messages.

The Device master-class constructs all the above mentioned objects and provides an advance method that polls the DMA hardware for event notifications and calls the registered callbacks. Message objects are posted to the Device to initiate a message passing operation. The Device queues up messages when the injection FIFO on the DMA is full. It also maintains a common shared counter, which is used by messages that track completion through the progress of the DMA head pointer. This allows a very large number of messages to be in flight.

3.2.1 DMA Point-to-point Protocols

The point-to-point protocols instantiate the DMA message classes with inputs passed in from the DCMF API and set up callbacks to be invoked on message completion. Currently five protocols are implemented: eager, get, put, rendezvous, and multisend.

The Eager protocol instantiates the SendMessage and sends meta-data and application payload to the destination's reception memory FIFO. The MPI ordering semantics are ensured by the use of deterministic routing and injecting all descriptors to the same destination on one injection FIFO.

This protocol assumes that the receiver is able to receive the message. On the receiver, the first packet of the message results in the application handler being invoked (with the meta-data as a parameter). This handler allocates a buffer for the incoming message payload. For a short message, the application meta-data (see Section 2) and the payload is copied to the same buffer and only one descriptor is posted to optimize overheads.

The Get protocol implements the one-sided get interface. The application will initialize and exchange opaque memory region objects prior to issuing a one-sided operation. The memory region object is opaque to the application and contains platform specific attributes necessary to complete a one-sided operations. These attributes may index into a memory segment table to translate virtual to physical addresses, specify an offset from a base memory region address and a length, or other implementation specific attributes such as authentication and protection schemes. The source and destination memory regions are passed to the DCMF_Get interface with the offsets into the source and destination memory region and the number of bytes to transfer. The protocol instantiates a GetMessage object to move the data from the destination processor back to the source of the get. When the remote-get completes, the reception counter hits zero and the application completion callback is invoked. The DMA injection FIFOs can overflow when several remote-gets are issued to the same node resulting in an interrupt. We have implemented an interrupt handler that allocates a larger injection FIFO to allow the more remote-get requests to be processed.

The Put protocol implements the one-sided put interface. Memory regions are required to address the source and destination buffers, as required by the Get protocol. The protocol instantiates a PutMessage object to move data from the local processor to the remote processor. Similar to the send, completion on the source is tracked by observing the DMA head. The destination DMA uses a shared reception counter to process all incoming direct-put packets.

The Rendezvous protocol is implemented in the MPICH ADI with the DCMF_Send and DCMF_Get API calls. The sender sends a rendezvous packet to the receiver with the source buffer, offset and size of the message. Message ordering between eager and rendezvous messages is preserved between the send-receive pair by using the DCMF_Send API. The receiver then initiates a remote-get operation. If a receive has not been posted for the message, the ADI can delay the DCMF_Get until the receive is posted and a buffer is allocated for the message. On completion of the remote-get, the receiver sends an acknowledgment back to the sender. Other programming paradigms can have similar rendezvous implementations. We are also exploring a rendezvous protocol at the DCMF level. Even with rendezvous messages that use exclusive reception counters, in the first phase several rendezvous headers can be sent out. As reception counters become available, the remote-get calls are issued and completed in the DMA units without the sender CPU's participation.

The two flavors of Multisend, namely Multicast and Many-to-many instantiate the DMA Device MulticastMessage and ManyToManyMessage classes respectively. When the message completes, the application completion callback is called on the source. The receiver processes the message similar to the eager point-to-point message. The multisend multicast also provides the ability to multicast data along a line of a torus using the deposit-bit feature of the torus router.

The protocols presented above are MPI progress compliant. In the eager protocol, the data is moved on the network to the receiver's reception memory FIFO as soon as it has been initiated on the sender. To make the rendezvous protocol progress compliant, we need to enable interrupts on packet arrival. The interrupt will awaken the communication thread to initiate the remote-get. We also need to enable interrupts when reception counters hit zero to make progress on remote-get requests that have been queued due to unavailability of counters.

3.3 Collective Device

Similar to BG/L, the collective network on BG/P requires the processor core to inject and receive packets. However, the throughput of the collective network has almost doubled, but the clock speed of the core only increased about 20%. This has motivated the exploration of a multi-threaded design for the collective network software *device*. The collective network software device supports message classes for integer allreduce, floating point allreduce, and broadcast.

For short collective calls, only one thread is used and the collective network blocks for a finite amount of time. If the collective does not finish in this time-frame, the collective call returns and is completed in the next advance call. This optimization reduces the latency of the collective operation as it avoids polling the other devices such as the DMA and the global interrupt network. The collective device can signal a communication thread to accelerate packet processing on the collective network for large collective operations in SMP mode.

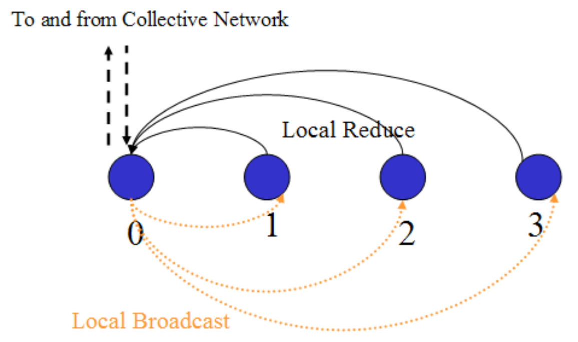
However, in quad mode (or virtual node mode), intra-node communication for global collectives has to be executed in software. We have explored two solutions for global allreduce in quad mode, one where the local reduce is performed over a shared memory segment and another where the DMA is used to move data within the node. These optimizations are also examples of protocols that can utilize multiple network devices on BG/P.

3.3.1 Quad Mode Allreduce DMA Optimization

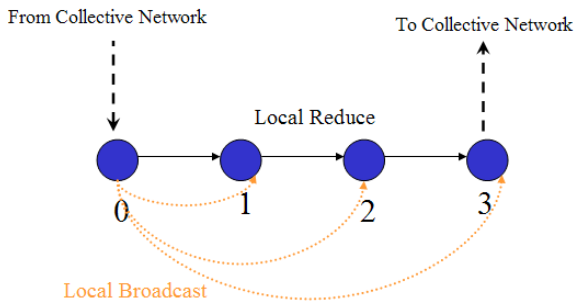
In quad mode, there are four processes (on the four cores) that have to perform both local computation and packet processing on the collective network. Figures 4(a) and 4(b) present two different collective network algorithms in quad mode for short and long messages.

With short allreduce calls, each core sends its local contribution directly to a designated core (core 0 in the figure) on each node, where all local data is reduced and then globally reduced on the collective network. Once the data is globally reduced, it is broadcast locally by core 0 using the DMA. The data movement from other cores to core 0 is overlapped to optimize latency. The local broadcast is an optimized multisend-multicast call which amortizes software overheads between message sends to all of the cores.

The large message scheme uses a pipelined load-balanced ring scheme to do the local reduction. To achieve better throughput, two cores are used to exclusively inject and receive packets on the collective network. As in the direct-reduce scheme, the DMA is used for intra-node communication between the cores. The local reduction starts with core 0 sending a fragment of its local contribution to core



(a) Short Optimization: Direct Reduce



(b) Long Optimization: Ring Reduce

Figure 4: Quad Mode Optimization across two networks

3 along the logical ring, via cores 1 and 2. At each core along the ring, the local contribution is reduced with the incoming message. At core 3 the data is injected onto the collective network for the global reduction, and the output of the collective network is received at core 0 and broadcast to all local cores using a DMA local multicast.

We have explored both DMA memory FIFO and direct-put messages for these two schemes and the performance results are presented in Section 4.

3.3.2 Quad Mode Allreduce Shared Memory Optimization

On BG/P, the channel between the DMA and the L3 cache is shared for all intra-node communication. As this channel can become a bottleneck, we explore shared memory optimization. In this scheme, the four MPI tasks perform a local reduction to a temporary local result buffer that is then injected by a single core onto the collective network device. The local reduction is accomplished with a pipelined lockless circular queue in shared memory. The first core copies its local source buffer into the pipeline, the second and third cores will each reduce every pipeline segment with its local source buffer back into the pipeline, and the fourth core will reduce each pipeline segment with its local source buffer into the temporary local result buffer. When the entire local result buffer is available, a core is dedicated to inject the local result buffer onto the collective network while a second core is dedicated to receive the result buffer from the collective network.

For an allreduce operation, a local broadcast of the result from the collective network is needed. The core dedicated to receive the result from the collective network will receive the result buffer directly into its global result buffer. Once the entire result buffer is received, the receive core will copy this buffer into a pipelined lockless circular queue in shared memory and the other three cores will copy the result into their individual global result buffers. This constitutes a shared memory local broadcast of the collective network allreduce operation.

Protocol	Latency(μs)
DCMF Eager One-way	1.9
MPI Eager One-way	3.2
MPI Rendezvous One-way	6.5
Converse MPI Driver	8.7
Converse DCMF Driver	4.5
DCMF Put	1.2
DCMF Get	2.0
MPI Get	3.5
ARMCI non-blocking Get	4.7
ARMCI blocking Get	6.4

Table 1: Latency measurements in various programming paradigms in SMP mode

4. PERFORMANCE RESULTS

We ran several micro-benchmarks to measure the latency and throughput of the various levels of the software stack. Table 1 shows the half-round trip time of the ping-pong benchmark written using DCMF, MPI and Converse APIs respectively. For short messages, the half round-trip latency of the Eager protocols is about 1.9μs at the DCMF level and about 3.2μs at the MPI level. The MPI performance results are with thread-mode-multiple, where all threads can make MPI calls at the same time. Overheads in the the MPICH and MPICH ADI levels of the software stack and acquiring and releasing locks contribute to the 1.3μs performance difference. Table 1 also shows the latency of some one-sided operations at the DCMF, MPI and ARMCI levels. The MPI and ARMCI layers have higher overheads than DCMF.

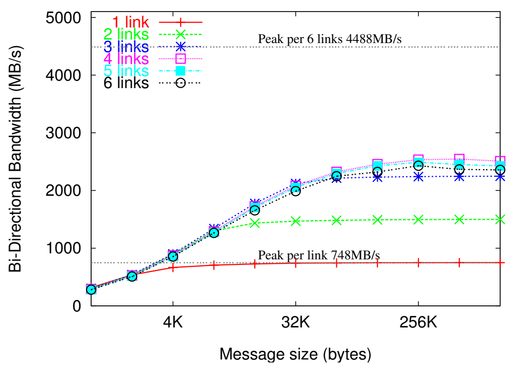


Figure 5: Throughput of eager protocol for near-neighbor exchange

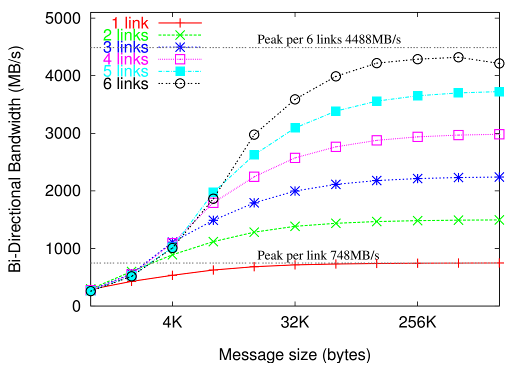


Figure 6: Throughput of rendezvous protocol for near-neighbor exchange

In addition, Table 1 presents the latency of the Converse runtime with an MPI driver and a direct DCMF driver. The MPI driver in Charm++ makes calls to MPI_Iprobe to poll for incoming messages. When a message has arrived, a blocking receive is posted to pull the message from the network. However, the DCMF driver takes advantage of active messages in DCMF to achieve lower overheads.

Figures 5 and 6 show the throughput of near-neighbor exchange for MPI eager and MPI rendezvous protocols, respectively. In this benchmark, each node sends and receives a message from up to 6 neighbors. With one neighbor, both eager and rendezvous protocols achieve close to the peak bi-directional performance with achieved throughputs of 745 MB/s and 748 MB/s, respectively. While the rendezvous protocol scales with the number of neighbors, the eager protocol saturates at 3 links. This is likely due to memory copy overheads of moving the message payload from the memory FIFO to the application buffer. In the rendezvous protocol, the DMA engine directly moves payload to the application buffer resulting in the scalable performance. So, MPI messages with sizes greater than 1KB use the rendezvous protocol.

4.1 Overlap of Computation and Communication

Because of the DMA engine on BG/P, it is possible to get better overlap of communication and computation than on BG/L. Therefore applications using the non-blocking point-to-point calls in the various supported paradigms can get better performance in regions where overlap is possible.

We define an overlap-ratio as :

$$(compute\_time + communication\_time)/compute\_time$$

An overlap-ratio of *one* indicates the communication time was completely buried in the computation. An overlap-ratio greater than one indicates the communication time was not buried in the computation.

We measured the overlap-ratio on BG/P through MPI benchmarks. In Figure 7, the sender posts a non-blocking MPI_Isend, enters a compute loop, and then calls MPI_Wait to ensure the send has completed. At the same time, the receiver blocks on an MPI_Recv call. In the plot, the black regions represent an overlap-ratio close to one, while grey and white regions have higher overlap ratios. Only small

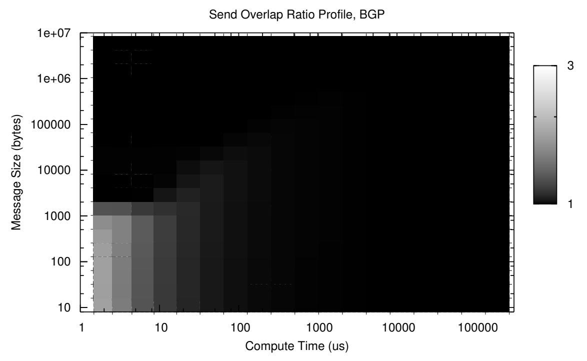


Figure 7: Send overlap performance (no interrupts)

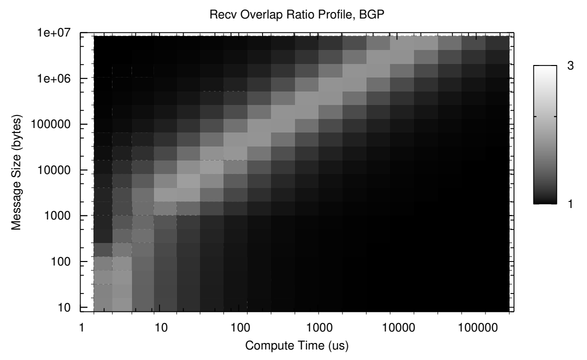


Figure 8: Receive overlap performance (no interrupts)

messages overlapping with low compute times show high overlap-ratios, possibly due to the software overheads of a short message. Figure 8 shows overlap in a benchmark where the receiver computes after posting an MPI_Irecv and then calls wait while the sender blocks on an MPI_Send. With short messages the overlap-ratio is possibly higher than one due to software overheads. With large messages, the rendezvous header is not processed while the core computes, resulting in a higher overlap-ratio. Figures 9 and 10 show the overlap-ratio profile with interrupts enabled. Interrupts improve the receive overlap with rendezvous messages, as the rendezvous header is processed by the interrupt handler which issues a remote-get to finish all data movement while the cores compute.

4.2 Global Allreduce Performance

Figures 11 and 12 show the performance of the integer sum allreduce operation in SMP mode. The latency for short messages is about $3.2\mu s$ on 32 nodes. Two threads are enabled at 4096 integers, and the peak throughput achieved is about 816MB/s (about 96% of the raw network throughput).

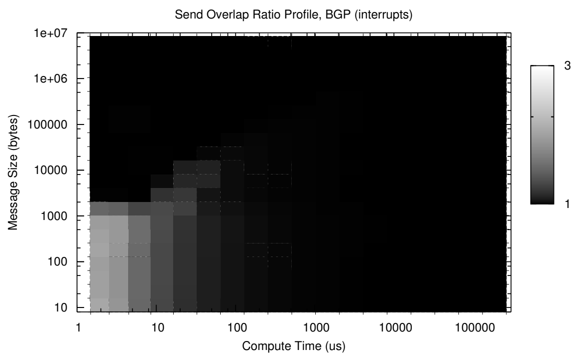


Figure 9: Send overlap performance with interrupts

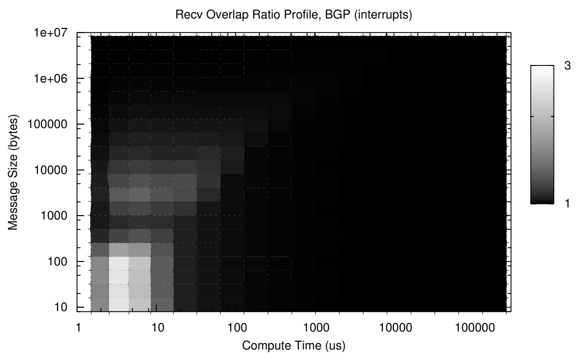


Figure 10: Receive overlap performance with interrupts

The performance of integer sum allreduce in quad mode (or VN mode) is shown in Figures 13 and 14. The switch from direct-reduce to ring-reduce (as discussed in Section 3.3) occurs at 512 integers. The maximum achieved throughput is 282 MB/s with the DMA direct-put scheme. The throughput is restricted by the rate at which the PowerPC 450 core on BG/P can compute a vector integer sum. The shared memory optimization has better throughput for intermediate sized messages than memory FIFO DMA, but has lower throughput than the direct-put version. The shared memory optimization has higher overheads as data has to be copied to a shared memory buffer before it is reduced, and then the final result has to be copied from the shared memory buffer to the application output buffers. Moreover, as the shared memory segment size is fixed, the throughput peaks at about 32K integers.

5. SUMMARY AND FUTURE WORK

We presented the motivation, architecture and BG/P implementation of the Deep Computing Messaging Framework. We explored a C++ device and protocol interfaces which can easily support multiple networks with different interfaces.

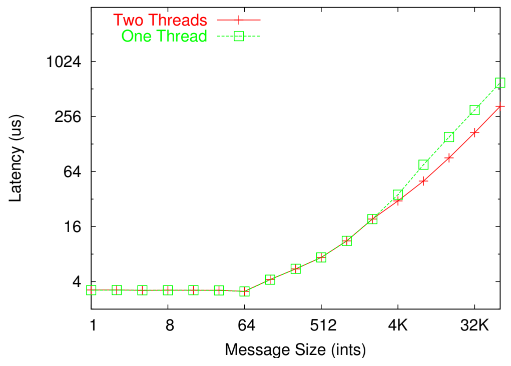


Figure 11: Latency (μs) of MPI integer sum allreduce on 32 nodes in SMP Mode

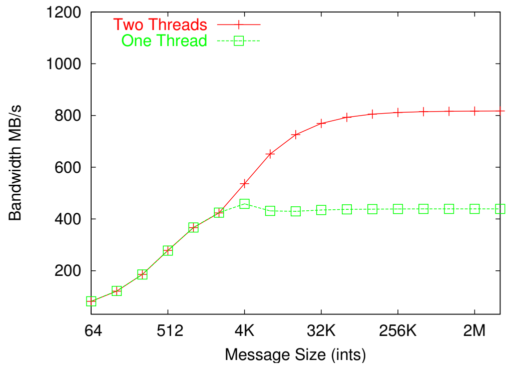


Figure 12: Throughput (MB/s) of MPI integer sum allreduce on 32 nodes in SMP Mode

We presented a non-blocking active messaging API with multiple contexts and consistency levels, multisend and collective calls to easily support many programming paradigms on architectures with low-frequency cores. We also presented preliminary performance results with the ARMCI, MPI and Charm++ paradigms, that show relatively low overheads and near peak throughputs for point-to-point messages, one-sided communication and the allreduce collective operation. We are exploring a generalized collective framework developed on top of the multisend calls to provide low-latency high throughput collectives on subsets of processors.

6. ACKNOWLEDGMENTS

We would like to thank Jeremy Berg, Jose G. Castanos, Bob Cernohous, Thomas Gooding, Douglas Miller, Carl Obert, Craig Stunkel, and Robert Wisniewski for their support in the design and development of DCMF on the Blue Gene/P machine.

The work presented in this paper was funded in part by the US Government contract No. B554331.

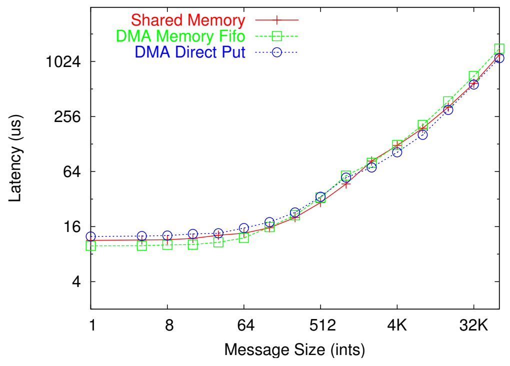


Figure 13: Latency (μs) of MPI integer sum allreduce on 32 nodes in Quad Mode

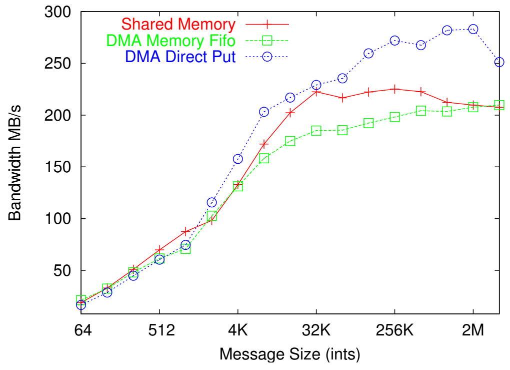


Figure 14: Throughput (MB/s) of MPI integer sum allreduce on 32 nodes in Quad Mode

7. REFERENCES

[1] Open Fabrics Alliance. http://www.openfabrics.org.

[2] N. R. Adiga et al. Blue Gene/L torus interconnection network. *IBM J. Res. Dev.*, 49:265–276, (2005).

[3] G. Almasi et al. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM J. Res. Dev.*, 49:393–406, (2005).

[4] M. Banikazemi, R. 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, 2001.

[5] C. Barton, C. Cascaval, S. Chatterjee, G. Almasi, Y. Zheng, M. Farreras, and J. Amaral. Shared memory programming for large scale machines. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006.

[6] DCMF. http://dcmf.anl-external.org/wiki, 2008.

[7] J. Dongarra, E. Strohmaier, H. Simon, and H. Meuer. www.top500.org, 2007. Date retrieved: 10 Jan 2008.

[8] M. P. I. Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. Mpich: A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[10] IBM Blue Gene Team. Overview of the Blue Gene/P project. *IBM J. Res. Dev.*, 52(1/2), January (2008). http://www.research.ibm.com/journal/rd/521/team.html.

[11] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[12] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[13] S. Kumar, C. Huang, G. Almasi, and L. V. Kalé. Achieving strong scaling with NAMD on Blue Gene/L. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[14] Myrinet Inc. *"Myrinet Express (MX), A High Performance Low Level Message Passing Interface for Myrinet"*, January 2006.

[15] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.

[16] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network: high-performance clustering technology. *IEEE Micro*, 22(1):46 –57, 2002.

[17] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*, August 1995.