CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm

Eric Bohm, Sayantan Chakravorty, Pritish Jetley,
Abhinav Bhatelé, Laxmikant V. Kalé

Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana, IL 61801
{ebohm, schkrvrt, pjetley2, bhatele2, kale}@uiuc.edu

Abstract

A significant fraction of parallel scientific codes are iterative with barriers between iterations or even between phases of the same iteration. The sender of a message is assured that the receiver is executing exactly the same iteration or phase as it. This opens up the opportunity to use one-sided communication without synchronization, explicit or implicit, between the sender and receiver of every message. The synchronization inherent in the application is sufficient to ensure correctness. We present CkDirect, an interface for such one-sided communication in the message driven Charm++ runtime system. CkDirect helps avoid unnecessary synchronization and message copying as well as scheduling overhead in iterative scientific codes. We describe the interface as well as its implementations on two different interconnects: Infiniband and Blue Gene/P. We evaluate CkDirect through a micro-benchmark, two simple scientific codes: stencil computation and matrix multiplication, as well as a full fledged quantum chemistry application called OpenAtom.

1 Introduction

The message-driven parallel programming model has been found to be suitable for a number of different application scenarios [15, 16, 7]. In the message-driven model, the runtime system is responsible for the buffer management necessary for receiving and sending messages. So, the user code can send a message without the receiver having to explicitly know about the message. Each message is associated with a handler method and when a message is received on a processor, the corresponding handler is invoked with the message as an argument. Frequently, a message-driven model is implemented with a scheduler on each processor maintaining a queue of received messages. The scheduler selects a message from the queue and invokes its handler. This message-driven model of parallel programming makes it easy to adaptively overlap computation and communication [5]. Moreover, a message-driven model has benefits for developing parallel modules and composing them seamlessly into one application [5].

CHARM++ is a runtime system (*RTS*) that combines a message-driven programming model with the equally powerful idea of object-based virtualization [7]. With object-based processor virtualization, users view computation as a collection of interacting objects without regard to the actual number of physical processors. It is the responsibility of the CHARM++ RTS to map these objects, referred to as virtual processors (or *chares*), to physical processors. These chares are message-driven objects with special methods called *entry methods* that act as handlers for incoming messages sent by other chares. A chare sends a message to another by remotely invoking an entry method on the receiving chare with the desired message. The CHARM++ RTS is responsible for locating the receiver chare and transmitting the message to the receiver's location. The RTS makes sure that a buffer is allocated on the receiving processor to receive the message. The CHARM++ RTS then enqueues each received message on a processor in a scheduler queue. The scheduler selects a message from the queue and invokes the corresponding entry method on the appropriate chare with the message as an argument. CHARM++ has been ported to a large number of systems; the actual message transmission scheme, therefore, varies according to the interconnect being used. The CHARM++ RTS has enabled the scaling of applications in very different fields to large numbers of processors such as NAMD [2] for molecular dynamics, ChaNGa [6] for cosmological simulations, OPENATOM [3] for Car-Parrinello ab initio molecular dynamics and ParFUM [9] for finite element methods.

In spite of this considerable success of the message-driven paradigm, we noticed a scope for improvement that could potentially benefit a number of categories of scientific applications. We observed that most scientific codes are iterative. A number of such iterative codes exchange the same amount of data between the same partners in each iteration, e.g. QM/MM, non-adaptive finite element simulations, etc. Moreover, for many other applications the amount of data communicated between partners during an iteration changes infrequently and slowly.

Iteration boundaries are implicit (or explicit) synchronization points for this style of communication. When a sender sends out a message during a particular iteration, it is assured that the receiver has finished the previous iteration and is performing the same iteration as it. Therefore, there is no reason to enforce synchronization again during the communication itself, either explicitly in user code (matching sends/receives) or within the runtime implementation (rendezvous). The application's own synchronization is sufficient to guarantee that the communication occurs correctly. The sender can safely write data into a receiver's buffer without fear of overwriting useful data from the previous iteration. In fact, the receiver need not be involved at all except to be informed when the data has been completely received.

Prior to this effort, the CHARM++ RTS did not provide users with any facility for such one-sided communication. We present a one-sided persistent communication interface called *CkDirect* for the CHARM++ RTS. CkDirect is an extension to the CHARM++ RTS that can be used by iterative applications with stable communication patterns to avoid some of the unnecessary overheads of the message-driven model.

2 CkDirect

CkDirect extends the CHARM++ RTS by providing a persistent, one-way, one-sided memory to memory communication interface between two chares. The goals of this extension are to minimize the overhead involved for a specific class of communication and to exploit RDMA features where available. Using CkDirect, chares can define channels through which they can send contiguous blocks of data to remote memory regions without wrapping them in messages. However, the CkDirect interface constrains the manner in which it can be used in a CHARM++ program. It is not designed to be general purpose, but instead targets iterative applications with stable communication patterns.

CkDirect was designed to allow such communication patterns to be implemented with the minimum possible overhead. On machines capable of RDMA, it lets a sender write directly into the receiver's buffer since the receiver is guaranteed to be ready. This lets the receiver get the data in the location exactly where it is needed, for example a row in the middle of a matrix. In a normal CHARM++ program, either the receiver would receive the data in a message and copy it into the desired location (the row in the matrix), or the computation code would have to be changed to operate on this received message rather than the matrix. CkDirect lets users avoid the performance and productivity costs of these additional operations. Once the data has been written into the receiver's buffer, the receiver is informed through a simple function call back. Thus the receiver can easily find out when it has received the data without having to pay the scheduling overhead incurred by a CHARM++ message.

The remote write (*put*) operation, which is initiated by the sender, was selected because it closely matches the message driven programming model wherein message senders entirely drive the flow of control. A chare knows when some local data, resulting from a local computation, is ready for use by a remote chare. The sender packs the data into a message and sends it to the receiver, where a message handler gets invoked once the message has been received. This message send can be replaced by a put without altering the flow of control on the sender or receiver, as long as the receiver is informed of the message's arrival. As discussed earlier, in an iterative code the receiver is guaranteed to be ready to receive data. On the other hand, a receiver initiated remote read call (*get*), requires that the receiver, through some synchronization, gain the knowledge that the source is ready to send it data. The receiver then initiates the get process and must be prompted to continue when the get is complete. Thus, using a get operation forces us to deviate considerably from the message driven model in which communication is initiated by the sender whenever it is ready. Therefore, we selected the put operation for the CkDirect extension to the message driven CHARM++ RTS.

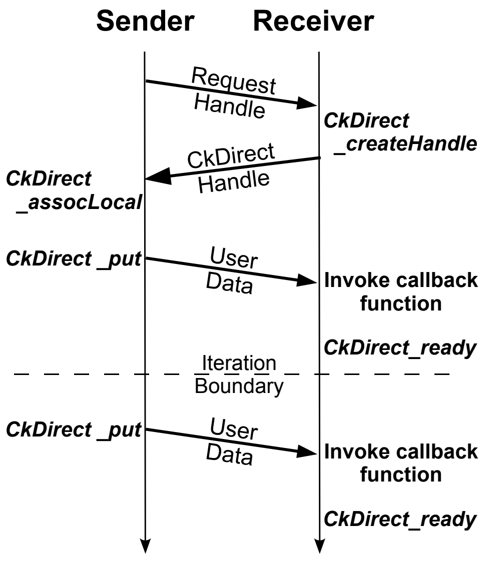


Figure 1: The different steps in setting up and using a CkDirect channel.

Figure 1 illustrates the different functions that make up the CkDirect interface and how they are used to establish a CkDirect channel between two chares and send data along that channel. A CkDirect channel needs to be set up before it can be used for communication between two chares. The setup consists of two steps.

In the first step, the receiver calls **CkDirect_createHandle** to create a handle representing the channel. The arguments to CkDirect_createHandle include the address of the receiving buffer, its size, an *out-of-band* pattern as well as a function callback and its data to inform the receiver of the message's arrival. The out-of-band pattern is an double word pattern that the user is sure will never appear as received data, for example: NaN in an array of doubles, -1 in an array of positive integers, etc. As we shall see in Section 2.1, this out-of-band pattern can be used to detect when a CkDirect message has been received.

The second step during the setup of a CkDirect channel consists of sending the handle created on the receiver to the sender. On the sender side, the sender calls **CkDirect_assocLocal** to associate a local sender buffer with this particular handle. This completes the setup of a CkDirect channel. The same local send buffer can be associated with multiple different handles. This allows the same data to be sent to different receivers along different CkDirect channels without creating multiple copies of it.

The sender can send data along a CkDirect channel by simply invoking **CkDirect_put** on the corresponding handle. Once the RTS on the receiving processor detects that the data on a particular CkDirect channel has arrived, it invokes the call back function passed to CkDirect_createHandle with the callback data as argument.

When an iteration is over the receiver can signal its readiness to receive data for the next iteration by calling **CkDirect_ready**. It must be noted that this does not perform any synchronization between the sender and receiver. It is just used to let the RTS know that it should expect new data on this particular CkDirect channel. The synchronization to make sure that the sender does not send a second message before the receiver calls CkDirect_ready is the user's responsibility and is achieved through synchronization in other parts of the application.

2.1 Implementation on Infiniband

The Infiniband port of CHARM++ is implemented on top of the Reliable Connection protocol of Infiniband. Hence, the CHARM++ implementation can assume that the Infiniband layer guarantees message delivery. Moreover, the data within a message arrives in order. So, if the last byte has been received one can be sure that the rest of the message has also been received. The Reliable Connection protocol makes the use of RDMA features of Infiniband relatively effortless. Since the sending and receiving buffers are both known in CkDirect, it is a nice fit for the RDMA capabilities of Infiniband.

CkDirect is implemented on top of Infiniband through a *polling queue*, maintained on each processor. The polling queue is a list containing the handles of all CkDirect messages expected on that processor. When a CkDirect handle is created with a call to CkDirect_createHandle, it is automatically added to the polling queue. The call to CkDirect_createHandle also sets the last 8 bytes of the receiving buffer to the out-of-band value provided by the user. It also registers the receiving buffer with the Infiniband layer as a memory chunk that might receive a remote write.

Similarly, on the sender side, during the call to CkDirect_assocLocal, the local sending buffer is registered with the Infiniband layer as a memory chunk from which a remote read can be performed.

Once the user calls CkDirect_put on the sender, an RDMA instruction is issued to the Infiniband layer. This RDMA instruction reads the data from the local sending buffer corresponding to the handle and writes it into the remote receiving buffer.

The RTS on the receiving processor checks periodically scans the polling queue for outstanding received data. It checks whether CkDirect data has been received by verifying that the last double word of the receiving buffer is no longer equal to the out-of-band value provided by the user. Upon detecting the receipt of CkDirect data, the corresponding handle is removed from the polling queue and its associated callback is invoked.

When the receiver is ready for the next block of data at the same location, it calls **CkDirect_ready** on the corresponding handle. This causes the RTS to again set the last 8 bytes of the receiving buffer to the out-of-band value provided during handle creation by the user. The RTS also enqueues the CkDirect handle into the polling queue. It should be noted that during a call to CkDirect_ready, the receiver does not send any message to or participate in any synchronization with the sender.

For performance reasons, the task of CkDirect_ready is actually performed through two separate operations by the user: 1) Mark the handle as having processed the current iteration and ready for the next iteration and 2) Start polling the handle for new data. The Infiniband implementation allows us to split CkDirect_ready explicitly into two calls: **CkDirect_ReadyMark** and **CkDirect_ReadyPollQ**. CkDirect_ReadyMark sets the of out-of-band byte pattern so that a put can be detected. CkDirect_ReadyPollQ inserts the CkDirect handle into the polling queue if new data has not already been received for that handle. The CkDirect_ReadyMark call can be used as soon as the receiver side user code is done with the buffer. The CkDirect_ReadyPollQ call is invoked at a later time when the user code expects that activity will occur on the channel. Using the two separate calls allows the user code to shorten the time span during which a CkDirect handle needs to be polled, without missing any message. This is particularly useful if an iteration has multiple phases. The user can make sure that the RTS polls a CkDirect handle only in the particular phase in which it is being used. However, if the application does not have such separate phases the user can always use the single CkDirect_ready call.

2.2 Implementation on Blue Gene/P

IBM's Blue Gene/P (BG/P) machine provides the Deep Computing Messaging Framework [8] (DCMF) for the implementation of messaging systems such as MPI, CHARM++, ARMCI and GASnet. DCMF uses active message semantics, wherein the first header packet of a message designates a handler function for the entire message. The one-sided primitives in DCMF were in flux at the time of this writing and therefore, based on the advice of DCMF implementers, we elected to use DMCF's two-sided interface for the initial implementation. This means that the current implementation of CkDirect on BG/P is not zero-copy. Nevertheless this does allow the CHARM++ RTS to avoid copying within its implementation of CkDirect.

The DCMF two-sided primitive, DCMF_Send, requires that handler functions be registered for receipt of short (< 224 B) and normal (≥ 224 B) messages. A normal message receipt handler must provide: a pointer to a receive buffer large enough to contain the message, a callback function pointer and data for completion notification, as well as a buffer for storing the message transaction state in a data structure of type DCMF_Request_t. The completion callback is invoked after the message has been delivered into the buffer provided by the handler. The short message handler is similar to the normal handler but must itself copy the data from the received message into the destination buffer. This active message approach dovetails nicely with the CkDirect semantic.

Each DCMF_Send invocation also includes a local send completion callback, and an Info header of up to 7 quad words (a quad word is 16 B) which accompanies the message payload. DCMF_Send, like the receipt handlers, must also provide a message transaction state buffer.

CkDirect_createHandle allocates a message transaction state buffer for the receive side and stores it along with information about the registered user buffer in the CkDirect handle returned to the user. Similarly, the sender side message transaction state buffer is allocated in CkDirect_assocLocal and stored in its copy of the CkDirect handle. Since a CkDirect channel can have at most one message in flight, these state buffers can be reused during subsequent CkDirect_puts.

The meta information capacity of the Info header and the receive side completion callback greatly simplify the implementation of CkDirect on BG/P. At each CkDirect_put, the sender "reminds" the receiver of the necessary DCMF context by sending the user application's receive buffer pointer, put callback, callback data, and the message transaction state buffer pointer, in the Info packet. This removes the need for any lookup tables to fill in the DCMF context based on the CkDirect handle. Sending all the DCMF context in the Info header does increase its size to two quad words. However, even if we had used look up tables to store the DCMF context for CkDirect handles and sent just the CkDirect handle in the Info, we would still have ended up with a Info of 1 quad word. Our implementation trades off the increased Info header size for a simpler implementation that delivers better performance.

The receiver side completion callback required by DCMF is set to the CkDirect user application's callback, further simplifying our implementation. This eliminates the need for polling receive buffers for completion as found in the Infiniband implementation. The CkDirect_Ready calls have no effect in the current Blue Gene/P implementation.

2.3 Related Work

MPI_Put provides a one-sided communication primitive similar to CkDirect. However, the receiver detects completion of a message very differently for *MPI_Put*. *MPI_Put* requires the use of one of three synchronization schemes (fence, post-start-complete-wait or lock-unlock) to signal the completion of operations at the receiver. The fence mechanism is a collective operation on all processors associated with an *MPI_Win*. If all we need is to detect completion on the receiver, this scheme is overkill since it forces the sender as well as other processors to synchronize needlessly. While the post-start-complete-wait scheme allows the definition of groups within which synchronization must take place, it is not free of overhead. The sender and receiver are forced to synchronize so that the receiver can detect the completion of all outstanding one-sided operations. Finally, the lock-unlock mechanism involves synchronization between the two parties in the form of lock acquisition and release.

CkDirect, in contrast, informs the receiver of received data through a simple callback function registered at setup time. The RTS on the receiving processor detects when a message has been completely received. The sender is not involved in any way and does not have to participate in any extra synchronization This reduces overhead in the large class of iterative codes in which the received data is assuredly not overwritten before it has been processed by the receiver in the current iteration. The flexibility and abstraction provided by the MPI one-sided interface comes at the cost of increased complexity and additional overhead. Indeed, as reported in [13], these overheads mean that even optimized implementations suffer longer latencies than point-to-point messages. As we shall see in Section 3, this is not the case with CkDirect.

ARMCI [11] is another messaging library that provides a variety of RMA operations. Unlike CkDirect, ARMCI puts support strided and I/O vector-specified data layouts. ARMCI also specifies the *ARMCI_Fence* and related operations to signal the global completion of outstanding puts. CkDirect avoids the overhead of such calls by relying on synchronization at the application level.

Message Size(10^3 B)	0.1	1.0	5.0	10.0	20.0	30.0	40.0	70.0	100.0	500.0
Default CHARM++	22.924	25.110	47.340	66.176	96.215	160.470	191.343	271.803	353.305	1399.145
CkDirect CHARM++	12.383	16.108	29.330	43.136	68.927	93.422	120.954	195.248	275.322	1294.358
MPICH-VMI	12.367	19.669	37.318	60.892	102.684	127.591	201.148	322.687	332.690	1396.942
MVAPICH	12.302	19.436	37.311	56.249	88.659	119.452	144.973	236.545	315.692	1386.051
MVAPICH-Put	16.801	22.821	51.750	64.202	94.250	120.218	146.028	232.021	308.942	1369.516

Table 1: Round trip time (μs) for the pingpong micro benchmark on CHARM++ without and with CkDirect as well as for two different MPI versions on Infiniband.

Message Size(10^3 B)	0.1	1.0	5.0	10.0	20.0	30.0	40.0	70.0	100.0	500.0
Default CHARM++	14.467	20.822	44.822	72.976	128.166	186.771	240.306	400.226	560.634	2693.601
CkDirect CHARM++	5.133	11.379	33.112	60.675	115.103	169.552	223.599	383.732	543.491	2677.072
MPI	7.606	13.936	39.903	66.661	120.548	173.041	226.739	386.712	546.740	2680.459
MPI-Put	14.049	17.836	39.963	67.972	122.693	178.571	232.629	392.388	552.708	2685.972

Table 2: Round trip time (μs) for the pingpong micro benchmark on CHARM++ without and with CkDirect as well as MPI on Blue Gene/P.

LAPI [12] provides an asynchronous infrastructure for the development of parallel applications. It is based on an active-message framework. Whereas CkDirect handles are bound to callback functions on the receiver during a registration process, a LAPI message includes the address of a user-specified function to be invoked at the receiver on message delivery. The two are similar in that they both expect the user to manage buffer space for arriving data. However, CkDirect is a higher level interface that, as we shall see, can be implemented on very different kinds of interconnects.

3 Microbenchmark

We used a simple pingpong microbenchmark to evaluate the effectiveness of our implementation of CkDirect. The benchmark measures the round trip time for various sizes of messages. For each message size, the reported time is averaged over a thousand iterations.

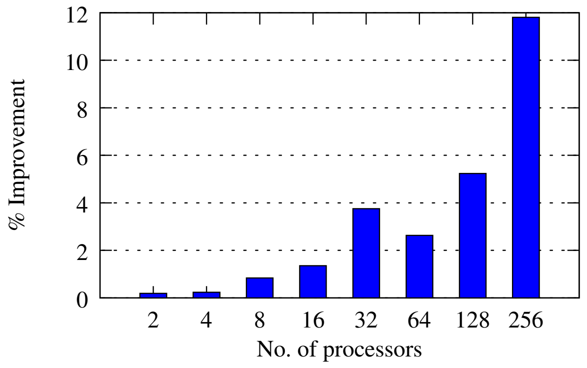
Table 3 shows pingpong times on Infiniband. This experiment was run on the Abe cluster at NCSA (Dual-socket quad-core 2.33 GHz Intel 64 Clovertown nodes connected by Infiniband.) We show the round trip time for the default implementation of CHARM++, CHARM++ using CkDirect as well as MPICH-VMI (2.2.0) and MVAPICH2 (0.9.8). For MVAPICH2, we show two times: one using two-sided communication and the other using one-sided communication (*MPI_Put*.) Message size was varied from 100 to 500,000 bytes. The message size in this case refers to the amount of user data being sent. The default version of CHARM++ sends an additional CHARM++ message header.

The round trip time for CHARM++ using CkDirect is lower than that of the default version of CHARM++ for all user message sizes. The performance gains for smaller user messages ($\leq$ 1 KB) are accounted for by two observations: 1) data sent using CkDirect_put do not need the transmission of the CHARM++ header ($\approx$ 80 bytes long) and 2) CkDirect data does not incur the CHARM++ scheduling overhead.

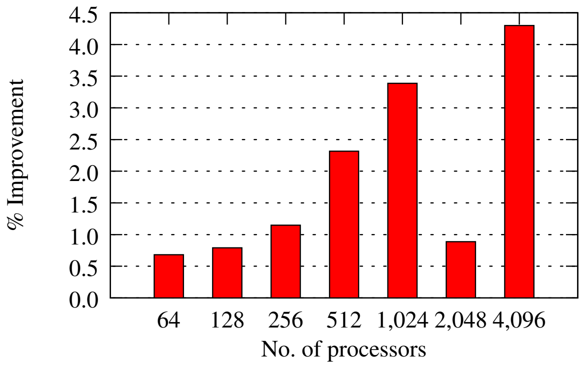
For messages sized between 1 KB and 20 KB, the default version uses a packet-based two-sided communication protocol whereas CkDirect uses just a RDMA put. Since the default version of CHARM++, unlike CkDirect, requires additional synchronization to use RDMA, it is cheaper for it to use the packet based protocol in this range of message sizes. However, the per-byte transmission cost for a RDMA put is lower than that for a packet based version. This explains the growing difference in performance between the CkDirect and default versions in this range.

Between 20 KB and 30 KB, the default CHARM++ version switches to a RDMA-based protocol. So, the only differences between the two versions are the initial rendezvous necessary for the RDMA-based default version and the fact that the CkDirect message avoids the CHARM++ scheduler. The scheduler cost is constant, particularly for a pingpong benchmark. The rendezvous has a constant cost synchronization component as well as a memory component whose cost increases slowly with message size. This is why the difference between the default and CkDirect versions grows slowly for messages bigger than 30 KB.

The CkDirect version of CHARM++ also performs better than both versions of MPI available on the machine. This might be because of the much simpler semantics of CkDirect compared to MPI. The simpler semantics of CkDirect means that, unlike MPI, there is very little subsidiary processing, such as tag matching, done by the RTS. MPI one-sided communication performed better than MPI two-sided for message sizes larger than 70 KB. However, it was still significantly slower than CHARM++ with CkDirect. This difference can be attributed to the extra synchronization (post-start-complete-wait) required while using *MPI_Put* to detect when a message has been successfully received. The lack of synchronization, explicit or implicit, in CkDirect affords it an advantage even over one-sided MPI communication primitives. Thus, CkDirect derives its benefit not just from one-sided communication but also its lack of synchronization.



(a) Results on the NCSA T3.



(b) Results on the ANL Blue Gene/P.

Figure 2: Improvement in average iteration time for CkDirect over CHARM++ messages.

Similar results were found on Argonne National lab's Blue Gene/P (Surveyor) as shown in Table 3. The one-way latency time for the DCMF message layer on BG/P is reported at $1.9\mu s$ [8]. Therefore, CkDirect is running quite close to the best performance available. We compared the vendor (IBM) MPI and CHARM++ with and without CkDirect. The CkDirect version is faster than the default message-based one for all message sizes tested, initially by $\approx 9\mu s$. This difference grows with message size to $\approx 16\mu s$. There is no cut-over point where RDMA is used on BG/P, because the supporting rendezvous protocol was not installed on Surveyor at the time of this writing. The scheduler cost remains constant and the copying cost for the default message based ping pong increases very slowly in proportion to message size.

The performance difference between CkDirect and MPI starts at $\approx 2.5\mu s$, jumps to $\approx 6\mu s$ at 5 KB and drops down to $\approx 3\mu s$ at 30 KB and greater. We surmise that there may be some kind of buffering threshold which results in a slightly different behavior within MPI. We assume that various MPI semantics account for the $\approx 3\mu s$ difference found at all message sizes, because no underlying one-sided semantics are being used in this BG/P implementation of CkDirect.

4 Simple applications

Experiments were performed to assess the effectiveness of the CkDirect interface for simple applications whose communication patterns are typical of many scientific codes. We present results from a stencil computation that avoids redundant copies and a matrix multiplication algorithm that optimizes communication volume.

4.1 Stencil Computation

By performing halo-exchanges interspersed with chunks of computation, stencil computation programs closely mimic the communication pattern of many real-world applications. This explains their significance as benchmarking tools. In our implementation, a three-dimensional domain is partitioned into cuboids, with one such cuboid assigned to each chare. We use the Jacobi method to converge to a solution. The two versions of the code we compare here are based on CHARM++ messages (MSG) and the CkDirect API (CKD) respectively.

Considerable effort was invested in ensuring a fair comparison between the two versions. In particular, we avoid copying overhead at the receiving side in both versions. Therefore, gains in performance are harder to come by and are solely the result of bypassing the CHARM++ scheduler in the one-sided communication version. As we shall show in the following subsections, this does lead to considerable performance advantages; however it also entails a significant increase in the application codebase which is both tedious to write and hard to debug.

The semantics of the stencil computation closely match the intended use of the CkDirect interface. Once a chare has set up CkDirect channels with each of its six communication neighbors, the computation proceeds in an iterative manner. Each chare sends its boundary (*ghost*) faces to the corresponding neighbors. A chare commences computation when it has all the ghost faces it needs. We ensure that there is only one CkDirect transaction in flight by having a global barrier after all chares have called CkDirect_ready.

We performed strong scaling tests on the ANL Blue Gene/P and NCSA's T3 (dual-socket, dual-core Intel64 Woodcrest nodes connected with Infiniband.) We observed that the program benefited greatly from processor virtualization. However, with an increase in the number of chares, the number of messages increases, with each message shrinking in size. This translates into greater scheduling overheads because of increased queue occupancy. In such a scenario, CkDirect helps improve performance by avoiding message creation as well as scheduling overheads.

We present results for a domain with $1024 \times 1024 \times 512$ elements. Figure 2(a) shows the improvement in performance gained by using CkDirect instead of CHARM++ messages on Infiniband. Best execution times were observed for a virtualization ratio of 8. As expected, we see greater percentage gains at finer granularities on higher numbers of processors. Note in particular the $\approx 12\%$ savings in execution time over the message-based version on 256 processors.

We performed similar tests on the ANL BG/P. Figure 2(b) shows improvements over the message-based version from 64 through 4,096 processors. As with the T3, we placed 8 chares

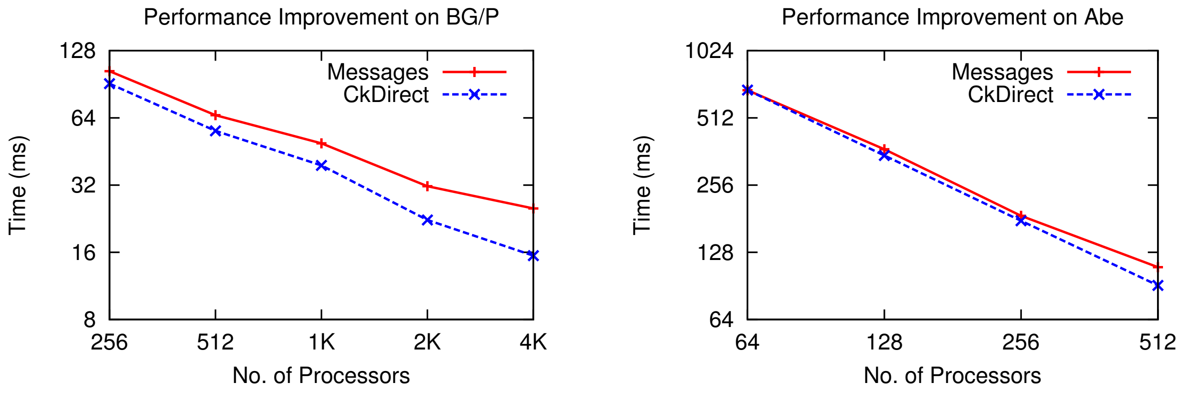


Figure 3: Execution time for Matrix Multiplication on Blue Gene/P and NCSA's Abe

per processor. As before, the percentage gains become more significant on more processors. There is, however, a sharp dip in improvement on the 2,048 processor configuration, something we noted over repeated runs on that processor count. We are still investigating this effect.

We see higher gains on Infiniband, since that implementation of CkDirect uses true one-sided synchronization free communication, unlike BG/P. Thus, in halo-exchange applications both implementations of CkDirect improve performance, particularly on large numbers of processors.

4.2 Matrix Multiplication

We implemented a parallel matrix multiplication algorithm that uses a 3D decomposition for 2D matrices [1]. Our implementation divides the input matrices A and B among a 3D array of chares. Before each chare can do its local computation, A should be replicated along the Z dimension of the array (chosen arbitrarily) and B along the X dimension. For this, every chare sends its portion of A and B to other chares which share its Z and X coordinate respectively. Once the computation is done, C needs to be distributed among the chares and hence every chare communicates with others having the same Y coordinate.

When CHARM++ messages are used for communication, the data needs to be copied into the correct locations in the local copy of A and B. Using CkDirect helps avoid this copying by directly placing the data in the appropriate locations. It also helps avoid the scheduling overhead of multiple incoming messages on each chare.

We present results for input matrices sized 2048×2048 on the Blue Gene/P and NCSA's Abe cluster. Figure 3 shows the performance of the implementations using messages and CkDirect respectively on the two machines. CkDirect outperforms the message-based implementation on both machines with the absolute difference in iteration times increasing with higher numbers of processors. The CkDirect version scales much better than the default version. This happens because in this application the number of messages per processor increases as the cube root of the number of processors. So on large numbers of processors, the lower overheads of CkDirect pay off even more. On 4K processors of BG/P, the performance improvement is close to 40%. Thus, CkDirect can help applications with similar communication characteristics scale to a higher number of processors.

5 OpenAtom

OpenAtom is a highly scalable implementation of the Car-Parrinello method [4] written in Charm++ using the PINY physics engine. Applications of this type perform molecular dynamics simulations by modeling the electronic structure explicitly (often called *ab initio*), with sufficient accuracy to model chemical bond formation and destruction. OpenAtom (the production name for the LeanCP code) is used in a variety of computational chemistry and material science contexts [17, 10]. It has previously been tuned on the Blue Gene/L [3], scaling efficiently to the entire 40,960 processor Blue Gene/L installation at T. J. Watson. As described elsewhere [3], the computation is implemented in a series of overlapping phases with various data dependencies. For the purpose of this paper only the PairCalculator phases used in orthonormalization will be discussed in detail. The underlying physics and mathematics for OpenAtom can be found in [14].

5.1 PairCalculator

Electronic states are typically represented as a 3D collection of complex double precision floating point values. These are naturally decomposed into $states$, and further decomposed into $planes$ and held by the $GS(s,p)$, a two-dimensional chare-array. For the purpose of orthonormalization, a pair of states is multiplied together to form an overlap matrix which is then used to maintain the orthogonality constraint. Previous optimizations for this process have implemented the "block-pairs" algorithm wherein the blocks of two states meet at a third object, the PairCalculator (PC). This object can be decomposed along states, and also along points of the plane, thereby maximizing parallelism while balancing various communication and multicast tradeoffs [3].

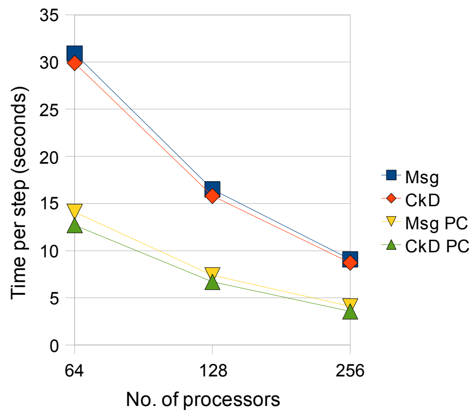
It is the communication of these points from each $GS(s,p)$ to $PC(s,s',p,p')$, which most easily lends itself to further optimization via CkDirect. This communication is repeated each iteration and the number of points is fixed. Further, data dependencies ensure that the sender and receiver are always on the same iteration . All the constraints of CkDirect as described in Section 2, are met.

The default implementation for this communication from GS to PC copies the points into a message and sends them to the PC, which copies the points into a contiguous data buffer and increments a counter. When all the states computed in a PC have arrived, it multiplies the data using DGEMM. Efficient execution by DGEMM requires that the data in the three matrix operands $(C = A \times B)$ be held in contiguous buffers. Orthonormalization updates the state data to maintain orthogonality and this updated data is returned to the GS chares for use in the remainder of the iteration.

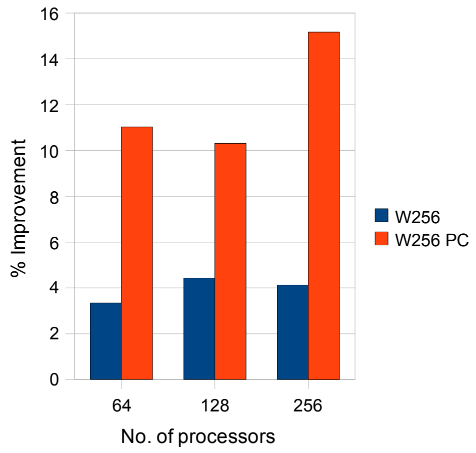
Using CkDirect, the points in GS and their destination in PC are registered as the sender and receiver buffers. The receiver side also registers a CkDirect callback which counts the number of states that have sent their points to the PC. When the data from all needed states has arrived, the callback enqueues a Charm++ entry method to perform the multiplication. The callback itself is a simple C function call, not a Charm++ entry method. Therefore, the scheme allows accumulation of all the data to occur without incurring entry method scheduling overhead on the receiver side. After the multiply is complete, the CkDirect_Ready function is called to prepare for the next iteration. The OpenAtom version that uses CkDirect is the same for both Blue Gene/P and Infiniband.

5.2 OPENATOM Performance with CkDirect

Although OPENATOM's use of the CkDirect API is architecture independent, the underlying implementation of polling for detection does impact an optimal vs non-optimal use of CkDirect. Our initial implementation experienced worse performance when using CkDirect than normal messaging. To see why, we first must consider that OPENATOM's coarsest decomposition requires $4 \times nstates \times nplanes$ CkDirect channels. In the benchmark, which has 1024 states, this requires thousands of CkDirect channels, a number which increases further each time the PairCalculator computation is further decomposed, as is done at higher processor counts. This typically requires tens or hundreds of channels per processor, with commensurate overhead to poll each channel. Each PairCalculator spends most of the time step ready for input, which can inflict the polling overhead on many unrelated phases. The simplest solution for this problem is to use CkDirect_ReadyMark when the data is ready and CkDirect_ReadyPollQ at the end of the phase prior to the PairCalculator.



(a) Scaling on NCSA Abe

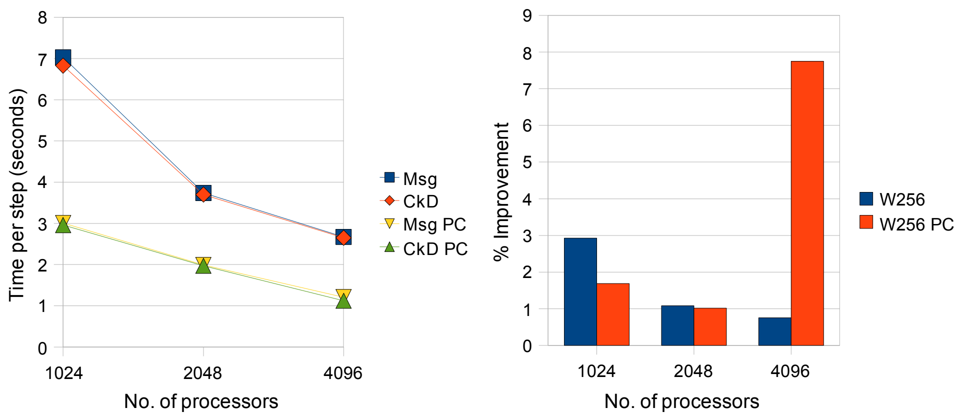


(b) Imp. on NCSA Abe

Figure 4: OPENATOM performance for CkDirect and CHARM++ messages on Abe.

With this optimization in place we obtained performance results for the 256 water molecule benchmark with a 70 Rydberg cutoff on Abe. Figures 4(a) and 4(b) indicate that the performance improvement on the NCSA Abe cluster is around 4%. Figures 4(a) and 4(b) indicate that the performance improvement on the NCSA Abe cluster is around 4%. To highlight the phases optimized using CkDirect, we also ran a version of the benchmark which disables all phases except for the PairCalculator phases, while retaining all PairCalculator-related communication. These timing results are differentiated by "PC" in Figure 4. The PairCalculator-only version results shows an even greater difference across architectures reaching as high as 14% on Abe. Although CkDirect performs better at all points, strong scaling efficiency for OPENATOM drops off earlier on Abe, particularly when running on many cores per node, due to limited bandwidth and computational noise (further discussion of these issues is outside the scope of this paper), so we truncated this graph at 256 processors and used 2 cores per node to simplify analysis and highlight network effects.

Figures 5(a) and 5(b) present performance results for the same Rydberg cutoff on Blue Gene/P. The CkDirect version is slightly faster for all processor counts. The application itself is already highly tuned to maximize overlap of computation with communication and to exploit available latency. This initial Blue Gene/P implementation of CkDirect is effectively only removing the already low CHARM++ overheads. Although the PairCalculator phases dominate performance for this large benchmark, computation and communication are well overlapped in OPENATOM which provides enough latency tolerance that the reduction in communica-



(a) Scaling on ALCF BGP (b) Imp. on ALCF BGP

Figure 5: OPENATOM performance for CkDirect and CHARM++ messages on BGP.

tion overhead from CkDirect in one phase does not translate into the same gains for the whole time step. The net effect of the latency and communication overhead reduction is an improvement in PairCalculator-only runs that is similarly slight except for the 4096 run, where communication is more critical and the resulting benefits are more substantial.

We anticipate further improvements in OPENATOM's performance when the CkDirect optimization is integrated into other phases of the computation. The greater benefit on Infiniband is largely due to the benefits of RDMA and, to a lesser extent, from increased latency sensitivity in the application, which is due to the pairing of Abe's faster processors with a higher latency interconnect than found in Blue Gene/P. The general trend for all the benchmarks run is for the amount of benefit to increase with processor count, so we predict that this technique will prove highly beneficial in codes like OPENATOM on large scale machines, provided two conditions are met: (a) efficient one-sided primitives are used in the implementation of CkDirect and (b) the architecture has a higher communication to computation ratio than is achieved by 8-way multicore clusters with a single Infiniband connection per node.

6 Conclusion

We have described a new one-sided communication interface, CkDirect, which relies on pre-existing synchronization in applications. Though it places constraints on the conditions under which it can be used, we illustrated its benefits in several use cases common to science codes. We demonstrated considerable performance gains over CHARM++ messages and *MPI_put* in microbenchmarks, simple applications and a production science code. These benefits were demonstrated on both the x86_64 Infiniband and Blue Gene/P architectures.

Although CkDirect provides several performance benefits, it is a simplistic and low-level interface for memory to memory one-way operations. As such, it is more labor-intensive to use and debug than standard Charm++ messaging techniques. We are considering several extensions to simplify its use, including support for multicasts, reductions, strided communication patterns and the eventual inclusion of CkDirect into an automatic learning framework which will create persistent channels where appropriate.

References

[1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[2] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[3] E. Bohm, G. J. Martyna, A. Bhatele, S. Kumar, L. V. Kale, J. A. Gunnels, and M. E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[4] R. Car and M. Parrinello. Unified approach for molecular dynamics and density functional theory. *Phys. Rev. Lett.*, 55:2471, (1985).

[5] A. Gursoy and L. Kalé. Performance and Modularity Benefits of Message-Driven Execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.

[6] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[7] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[8] S. Kumar, G. Dozsa, G. Almasi, D. Chen, M. E. Giampapa, P. Heidelberger, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *To appear, Proceedings of the 22nd ACM International Conference on Supercomputing*, 2008.

[9] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[10] Y. Mantz, H. Gerard, R. Iftimie, and G. Martyna. Isomerization of a peptide fragment studied theoretically in vacuum and in water solvent at finite temperature. *J. Am. Chem. Soc.*, 130:130, (2004).

[11] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, 2006.

[12] G. Shah, J. Nieplocha, J. H. Mirza, C. Kim, R. J. Harrison, R. Govindaraju, K. J. Gildea, P. DiNicola, and C. A. Bender. Performance and experience with lapi - a new high-performance communication library for the ibm rs/6000 sp. In *IPPS/SPDP*, pages 260–266, 1998.

[13] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in message passing interface one-sided communication. *International Journal of High Performance Computing Applications*, 19(2):119–128, May 2005.

[14] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16):2006–2022, Oct. 2004.

[15] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[16] D. A. Wallach, W. Hsieh, K. Johnson, M. F. Kaashoek, and W. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, pages 217–226, July 1995.

[17] T. Whitfield, G. Martyna, and J. Crain. Liquid NMA: A surprisingly realistic model for hydrogen bonding motifs in protens. *Chem. Phys. Lett.*, 414:210, (2005).