RC22300 (W0201-058) January 8, 2002
Computer Science

IBM Research Report

A C++ Implementation of the Co-Array Programming Model for Blue Gene/L

Maria Eleftheriou, Siddhartha Chatterjee, José E. Moreira
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A C++ Implementation of the Co-Array Programming Model for Blue Gene/L

Maria Eleftheriou Siddhartha Chatterjee José E. Moreira

{mariae,sc,jmoreira}@us.ibm.com
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598-0218

Submitted for publication

Abstract

Blue Gene/L (BG/L) is a 65,536-node massively parallel computer being developed at the IBM T. J. Watson Research Center that promises to revolutionize large-scale scientific computing. However, its size alone will make programming BG/L a major challenge, both from a correctness and from a performance perspective. This has motivated us to look into new programming models that have been proposed as an alternative for MPI on parallel systems. We find the Co-Array programming model attractive for its simplicity and high performance. Instead of implementing this model in the usual way, as language extensions to Fortran, we have chosen to implement it as a C++ library. This allowed us to prototype and deploy this implementation with a relatively small effort, and to experiment with it on existing machines. Users also benefit, for they can start experiment with this programming model early and without learning a new language. Our preliminary results indicate that parallel programs written with our C++ Co-Array library can reach good levels of speedup.

1 Introduction

Blue Gene/L (BG/L) [1] is a massively parallel computer system being developed at the IBM T. J. Watson Research Center. A fully-configured BG/L machine will consist of 65,536 dual-processor nodes interconnected as a $32 \times 32 \times 64$ three-dimensional torus, with message passing as the sole means of inter-node communication. Programming such a massively parallel machine will be a major challenge, from the perspective of both correctness and performance. Therefore, we wish to support high-level parallel programming models that support common patterns of parallel interaction and simplify the task of parallel programming.

We expect BG/L to support a subset of MPI-2 [4, 8] to allow programmers to write explicit message-passing programs. However, we believe that MPI as a programming model for massively parallel systems is too low-level, making it error-prone to code, complex to read and understand,

and difficult to debug. Other authors have recognized these deficiencies of MPI and have proposed alternative higher-level parallel programming models for large parallel systems. Examples of such programming models include Global Arrays [5], Titanium [9], Unified Parallel C (UPC) [3], and Co-Array Fortran [6, 7]. These various models have the following characteristics in common.

1. They are based on the Single Program Multiple Data (SPMD) model of computation. In this model, parallelism comes from running n images of the same program, each with its own set of data objects. Execution is asynchronous for the most part, except for well-defined global synchronization points.

2. They provide a global address space for placement of the shared data structures in the computation. This address space is intended to be used largely in a partitioned manner. Coherence of global data is usually guaranteed only at global synchronization points.

3. They provide convenient mechanisms for any image to access any piece of shared data. However, different parts of the global address space may have different access latencies.

4. They aim at providing levels of application performance that are competitive with direct programming using MPI.

Of these four programming models, Global Arrays is implemented as a library for C and Fortran, whereas Titanium, UPC, and Co-Array Fortran are implemented as language extensions to Java, C, and Fortran, respectively. Co-Array Fortran is unique in that, for every remote data reference, the remote image where the data resides is explicitly identified. The other models provide a degree of transparency when accessing data, the location of data being derivable from the element being accessed. In the case of Global Arrays, a single data reference may span multiple images.

Although we believe all four programming models can be valuable in the context of BG/L, and they all have been shown to achieve good performance, we chose to initially pursue an implementation of Co-Arrays. Two main factors influenced our decision. First, regardless of the actual language binding, the run time systems for all these models need to address some common issues: mapping between local and global address spaces, managing point-to-point communication and global synchronization, and overlapping computation and communication. Developing a run time system for one model provides important information for the development of run time systems for the other models. Second, we judged Co-Array to be the simplest of the programming models to implement. Both Global Arrays and UPC support complex data distributions, whereas Titanium supports general inter-node pointers.

Instead of implementing the Co-Array programming model as a language extension to Fortran, we implement it as a C++ library, for several reasons. First, the class definition, operator overloading, and generic programming mechanisms of C++ allow much of the Co-Array syntax to be implemented without the need of language extensions. Second, a library-only implementation is faster to prototype and deploy than a language extension, which requires compiler support. Third, it is easier to motivate users to experiment with a new library for a standard and familiar language than to convince them to try a new programming language. Finally, a C++ library implementation of the Co-Array programming model is portable across a variety of systems, allowing us to experiment with it before the BG/L machine is ready.

The remainder of this paper is organized as follows. Section 2 covers background material, including the main features of the Co-Array programming model and the relevant characteristics of BG/L. Section 3 describes our design of the C++ library for the Co-Array programming model. Section 4 illustrates programming with the library through some examples, and presents performance results. Section 5 concludes and discusses future work.

2 The Co-Array Programming Model and BG/L

In this section, we review the main features of the Co-Array programming model implemented by Co-Array Fortran. We also describe those features of the BG/L machine that had the most impact in our design of the C++ library for the Co-Array model.

2.1 Co-Array Fortran

Co-Array Fortran is a small set of extensions to Fortran 95 for SPMD parallel processing. With Co-Array Fortran, a single program is replicated a fixed number of times for execution. Each replica is called an *image*. Images execute asynchronously with respect to one another, and each image has its own set of data objects. Ordinary Fortran data objects declared in the program are replicated across all images, but each copy is private to its image.

Co-Array Fortran introduces a new kind of object, the *co-array*, which is formed by the aggregation of objects replicated across the images. A co-array is an array of objects that spans all images and allows cross-image accesses. A co-array has both local dimensions and co-dimensions. Local dimensions apply within each image and are declared with parentheses. Co-dimensions apply across images and are declared with square brackets. For example,

```
REAL u(nrows)[*]
```

declares a two-dimensional co-array `u`, consisting of a one-dimensional local array of `nrows` elements replicated across all images. Parentheses are used to index elements within an image, while square brackets are used to select an image. For example,

```
a = u(i)[p]
u(j)[p] = b
```

assigns the value of the `i`-th element of co-array `u` in image `p` to variable `a` in the local image, and assigns the value of variable `b` in the local image to the `j`-th element of co-array `u` in image `p`. Such inter-image assignments will, in general, result in inter-node communication when implemented on a message-passing machine. When a co-array variable is used without an explicit image index, it refers to the local image. Thus,

```
u(k) = u(j)
```

copies the value of the `j`-th element of `u` in the local image into the `k`-th element of `u` also in the local image. An efficient implementation should perform this assignment without communication.

In addition to co-arrays, Co-Array Fortran provides a set of intrinsics for parallel programming. The most important intrinsics are `num_images()`, `this_image()`, and `sync_all()`. The

`num_images()` intrinsic returns the number of images in the execution. The `this_image()` intrinsic returns the index of the local image in the set of all images, which is a number between 1 and `num_images()`. Finally, `sync_all(INTEGER list(*))` is a group barrier that requires all operations before the call on images in `list` to complete before any image in `list` advances beyond the call.

2.2 BG/L

As previously mentioned, the BG/L machine consists of 65,536 dual-processor nodes interconnected in a torus topology. The interconnection network supports high-bandwidth, low-latency delivery of variable-sized packets of data. Each node can send and receive data at an aggregate rate of more than 4 GB/s when running at 700 MHz. To support such high data rates, the preferred mode of operation for a BG/L node is to dedicate one of the processors to handle inter-node communication, while the other processor runs the user application. The two processors share the memory in the node, but they are not cache coherent. Therefore, data exchanges between the computation and communication processors are carried through a non-cacheable region of the address space.

3 The C++ Library for the Co-Array Model

This section describes our implementation of the Co-Array model for BG/L, as well as the binding of the Co-Array model to the C++ language. The description has four main components: the implementation of threads (Section 3.1), the implementation of co-arrays (Section 3.2), the implementation of point-to-point communication (Section 3.3), and the implementation of group synchronization (Section 3.4). In addition to exploiting the type system of C++, our implementation also uses the pthreads library to implement threads and the MPI library as the underlying messaging layer.

3.1 Implementing computation and communication agents

Each node of the BG/L architecture consists of two processors—the computation processor and the communication processor. From a programming perspective, each image of the user program is mapped to a node and is organized internally as two threads: a computation thread that runs on the computation processor, and a communication thread that runs on the communication processor. We implement these threads using the *pthreads* library [2]. The appropriate number of threads is spawned at program start-up. The computation thread begins by executing the top-level function of the user program; the communication thread runs in a reactive loop, processing communication events from its associated computation thread and from other communication threads. The two threads on a node communicate with each other through a pair of queues that reside in a non-cacheable region of the shared address space—one directed from the computation thread to the communication thread (the `comp2comm` queue), the other from the communication thread to the computation thread (the `comm2comp` queue). Both queues are protected by locks. This approach avoids complications resulting from the non-coherent caches in the processors.

3.2 Implementing co-arrays

A co-array of elements of type `T` is implemented as a C++ template class `CoArray<T>`. On encountering the definition

```
CoArray<double> A(100);
```

the computation thread in each image allocates a `LocalArray<double>` object capable of holding 100 `double`s, and an array of pointers to `RemoteArray<double>` objects that reference the `LocalArray` components of this `CoArray` in each of the other images. Figure 1 illustrates the design. Note that a remote pointer in one image is a local pointer in another image; an image can therefore perform address arithmetic on a remote pointer, but may not meaningfully dereference a remote pointer. The images exchange their local pointers in an all-to-all communication step within the `CoArray` constructor to set up the `RemoteArray` references in a consistent manner.

The `CoArray<T>` class overloads `operator()` to allow access to elements of the local portion of a co-array. More interestingly, it overloads `operator[]` to produce a reference to a `RemoteArray<T>` by accessing its cached copy. Applying `operator()` to a `RemoteArray<T>` returns a `RemotePtr<T>`, which is a reference to a specific remote element. One small syntactic difference from Co-Array Fortran is that a remote element of a co-array is written as `A[node](i)` rather than `A(i)[node]`.

The overloaded operators `operator=` and `operator T()` in the `RemotePtr<T>` class initiate inter-image communication. The former operator handles the case `A[node](i) = x` by sending a "remote write" message to the remote image. The latter operator handles the case `x = A[node](i)` by sending a "remote read" message and converting the result into the appropriate local type. Section 3.3 describes the details of the communication actions. The case `A[node1](i) = B[node2](j)` is handled by a combination of the two operators. In our current implementation, the construct `A(i) = A[this_image()](j)` causes an image to send a message to itself, but this could be easily optimized out.

3.3 Implementing point-to-point communication

Point-to-point communication in our implementation involves the computation and communication threads at both the local and the remote end. Figures 2 and 3 show the overall protocols for various communication operations.

The local computation thread initiates a communication operation as follows.

1. It initiates a "remote write" by enqueueing a *Put* request on its `comp2comm` queue and returning.

2. It initiates a "remote read" by enqueueing a *Get* request on its `comp2comm` queue and waiting until it receives the data value.

The local communication thread dequeues requests from its `comp2comm` queue and sends one of two kinds of messages to the remote communication thread over the network (using MPI as the transport mechanism).

1. A *Put* message is a request to store the data contained in the message at the specified address of the remote node, and is sent in response to a *Put* request. It contains the following fields: the remote address, the size of the data, and the data value.

2. A *Get* message is a request for data from a remote node, and is sent in response to a *Get* request. It contains the following fields: the local address where the data will eventually be stored, the remote address from which to read the data, and the size of the data.

The remote communication thread takes the following actions on receiving a message from the network.

1. On receiving a *Put* message, it enqueues a *Put* request on its `comm2comp` queue and sends an *Ack* message back to the local communication thread.

2. On receiving a *Get* message, it enqueues a *Get* request on its `comm2comp` queue.

The remote computation thread dequeues requests from its `comm2comp` thread and takes the following actions.

1. On receiving a *Put* request, it completes the remote write operation by writing the supplied values to the specified memory locations.

2. On receiving a *Get* request, it reads the specified memory location and enqueues a *Put* request containing the data addressed to the local node on its `comp2comm` queue.

Note that the computation thread of an node is responsible for draining the `comm2comp` queue in that node. The computation thread currently performs this action at three points: when it is waiting after having issued a *Get* request, before it issues a *Put* request, and when it engages in a group synchronization operation (Section 3.4).

A "remote write" operation completes at the local node when the local communication thread receives the *Ack* message from the remote communication thread. Note that the local computation thread is not blocked and can proceed with additional computation and communication. The operation completes at the remote node when the remote computation thread has dequeued and processed the *Put* request from its `comm2comp` queue.

A "remote read" operation completes at the local node when the local computation thread has dequeued and processed the returning *Put* request from its `comm2comp` queue. The operation completes at the remote node when the remote communication thread receives the *Ack* message from the local communication thread.

The *Ack* message is required in order to maintain the invariant that an image completes all communication actions prior to entering a group synchronization point. Communication threads decrement a count of pending communication actions on receiving an *Ack* message.

3.4 Implementing group synchronization

The library provides functions for group synchronization, that is, a barrier between all images (`sync_all()`) or a subset of images (`sync_all(list of images)`). As in the case of

point-to-point communication, group synchronization requires coordination among both local and remote computation and communication threads. Figure 4 illustrates the overall protocol.

When the computation thread of an image reaches a `sync_all` call, it enqueues a *Sync* request on its `comp2comm` queue and then repeatedly dequeues and processes requests from its `comm2comp` queue until it encounters a *Sync* request. The local communication thread sends out *Sync* messages to each image on the list and waits for *Sync* messages from each of them. Once it has received a message from each image on the list, it enqueues a *Sync* request on its `comm2comp` queue to inform the local computation thread that synchronization has been achieved.

Since different images can progress at different rates, the local communication thread can receive *Sync* messages from remote images at any time. If it receives a *Sync* message from an image that is not in the list of images with which it is currently synchronizing, it records the identity of the remote image in a pending list. Otherwise, the image is in the current synchronization list, in which case it deletes that image from the list and decrements the count of outstanding messages. When processing the *Sync* request from its local computation thread, it removes from the pending list any images from the current synchronization list that it finds there, and decrements the count of outstanding messages correspondingly.

4 Examples

We illustrate the use of the Co-Array programming model and the performance of our implementation using two examples: Jacobi relaxation and matrix transposition. Performance measurements were conducted on a Linux cluster of dual 600 MHz Pentium III machines, interconnected with a 100 Mbit/s Ethernet switch.

4.1 Jacobi relaxation

The Jacobi relaxation example closely follows the example in Numrich and Reid [6, §2.1]. The two-dimensional problem grid is partitioned across images in the column dimension. Each image is responsible for `ncol` columns, and allocates space for one column of ghost cells on either side.

```
typedef double vector_t[VECTOR_SIZE];
void laplace(int nrow, int ncol, CoArray<vector_t>& u)
{
  int me = u.this_image();
  int images = u.num_images();
  LocalArray<vector_t> new_u(ncol+2);
  int left  = me == 0 ? images-1 : me-1;
  int right = me == images-1 ? 0 : me+1;
  int list[2] = { left, right };
  u[left](ncol+1) = u(1);     // communication (put)
  u[right](0)     = u(ncol);  // communication (put)
  CoArray<double>::sync_all(list,2);
  for (int j = 1; j < ncol+1; j++) {
    new_u(j)[0] = u(j)[nrow-1] + u(j)[1] + u(j-1)[0] + u(j+1)[0];
    for (int i = 0; i < nrow-2; i++)
```

```
      new_u(j)[i+1] = u(j)[i] + u(j)[i+2] + u(j-1)[i+1] + u(j+1)[i+1];
    new_u(j)[nrow-1] = u(j)[0] + u(j)[nrow-2] + u(j-1)[nrow-1] + u(j+1)[nrow-
1];
  }
  for (int j = 1; j < ncol+1; j++)
    for (int i = 0; i< nrow; i++)
      u(j)[i] = new_u(j)[i] - 4.0 * u(j)[i];
}
```

Note that `u(j)[i]` denotes element `i` of column `j` of co-array `u` in the local image, and is a purely local operation.

Figure 5(a) shows the speedup of this code for various grid sizes. Speedup is near-optimal.

4.2 Matrix transposition

The matrix transposition example transposes a square matrix organized as a co-array of blocks that is partitioned across images in the column dimension.

```
typedef double block_t[BLOCK_SIZE][BLOCK_SIZE];
void transposeBlock(block_t& src, block_t& dst)
{
  for (int i = 0; i < BLOCK_SIZE; i++)
    for (int j = 0; j < BLOCK_SIZE; j++)
      dst[i][j] = src[j][i];
}

void MatrixTranspose(CoArray<block_t>&u, int nrow,int ncol)
{
  int me = u.this_image();
  int comm_size = u.num_images();
  CoArray<block_t> b(nrow, ncol, MPI_COMM_WORLD);
  block_t temp;
  CoArray<block_t>::sync_all(MPI_COMM_WORLD);
  for (int I = 0; I < nrow; I++) {
    int i = (I+me*ncol) % nrow;
    for (int j = 0; j < ncol; j++) {
      transposeBlock(u(i,j),temp);         // transpose local block
      b[i/ncol](j+me*ncol, i%ncol) = temp; //communication (put)
    }
  }
  CoArray<block_t>::sync_all();
}
```

Figure 5(b) shows the performance of this code for different block sizes. This code is much more communication-intensive than the previous one, which limits speedup as a function of number of nodes.

5 Conclusions and Future Work

We have discussed the design and implementation of a C++ library supporting the Co-Array parallel programming model on the BG/L machine. We have validated the correctness of the implementation and have characterized its performance using two test codes: Jacobi relaxation and matrix transposition. The initial performance results are promising.

We plan two fronts of future work. First, we need to reduce the run-time copy overheads in our implementation. In particular, it is possible in some circumstances to let the communication processor access the data directly. This would reduce the end-to-end latency of communication operations. Second, we need to better characterize the performance of our implementation, initially on a simulator of BG/L and eventually on the real hardware.

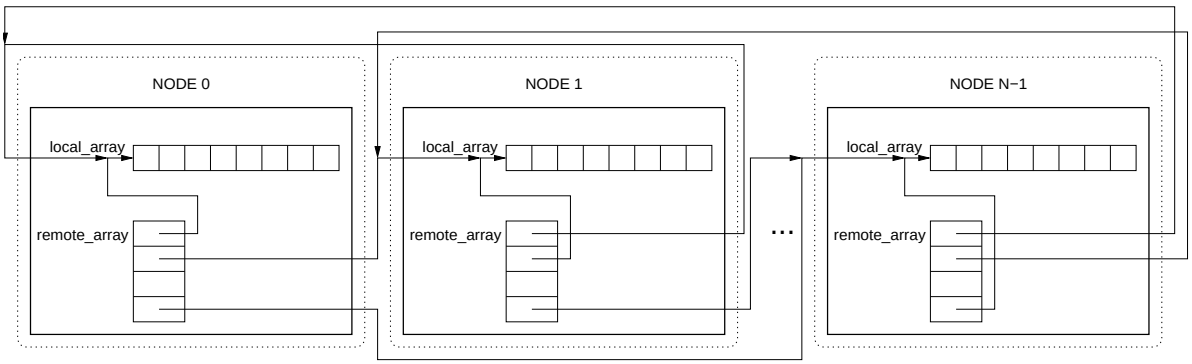


Figure 1: Implementation of a co-array.

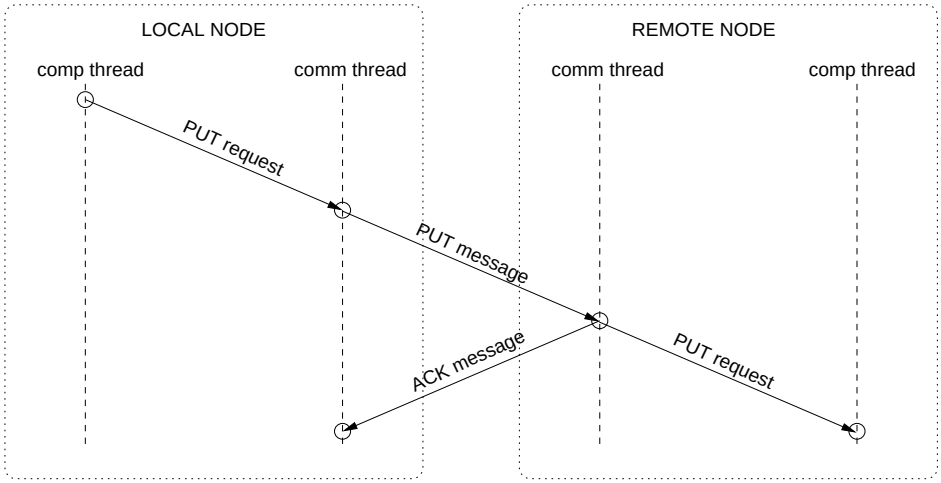


Figure 2: Transactions in a PUT operation.

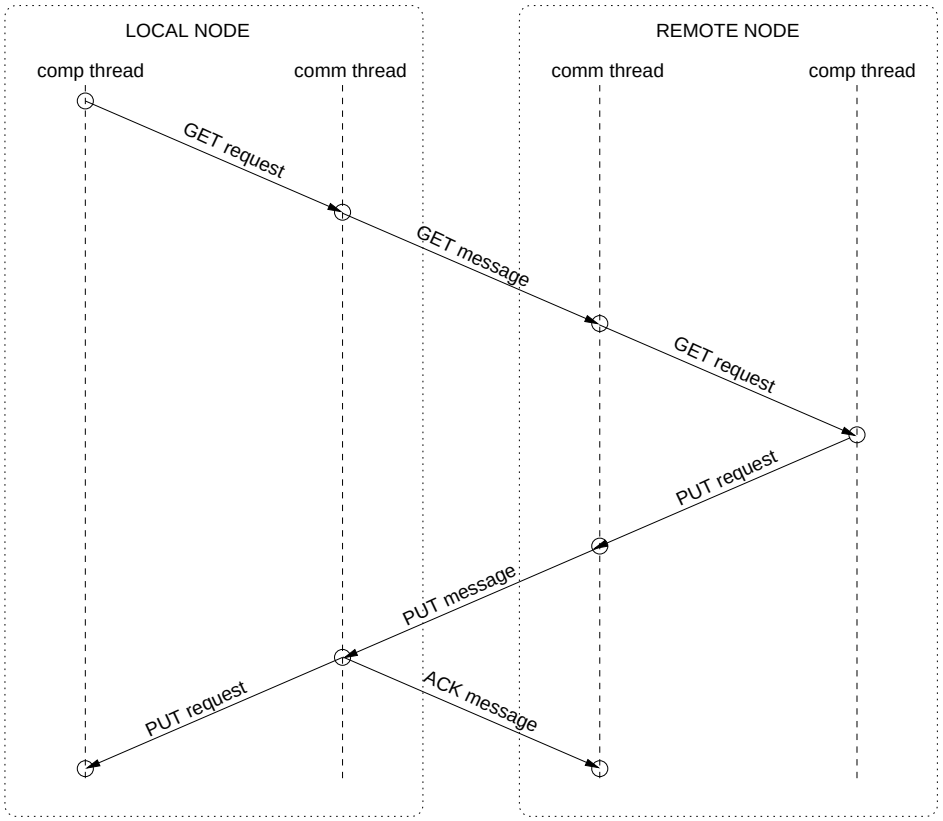


Figure 3: Transactions in a GET operation.

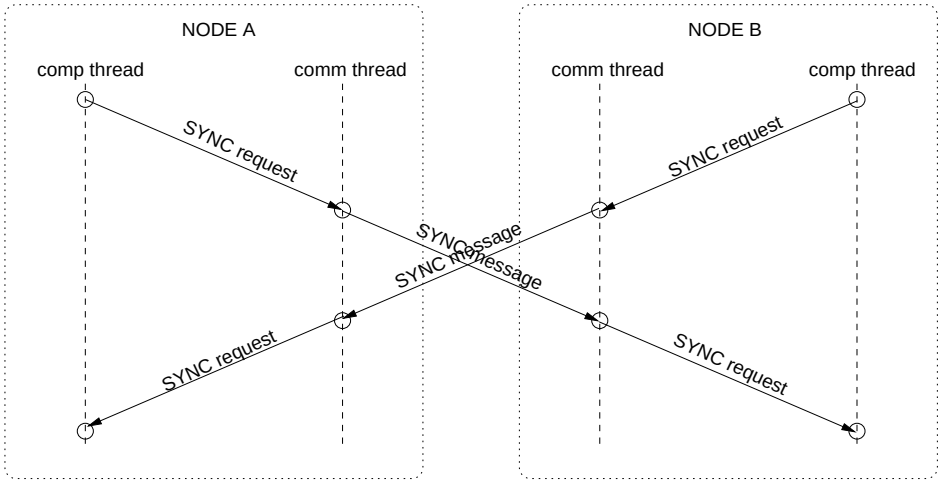
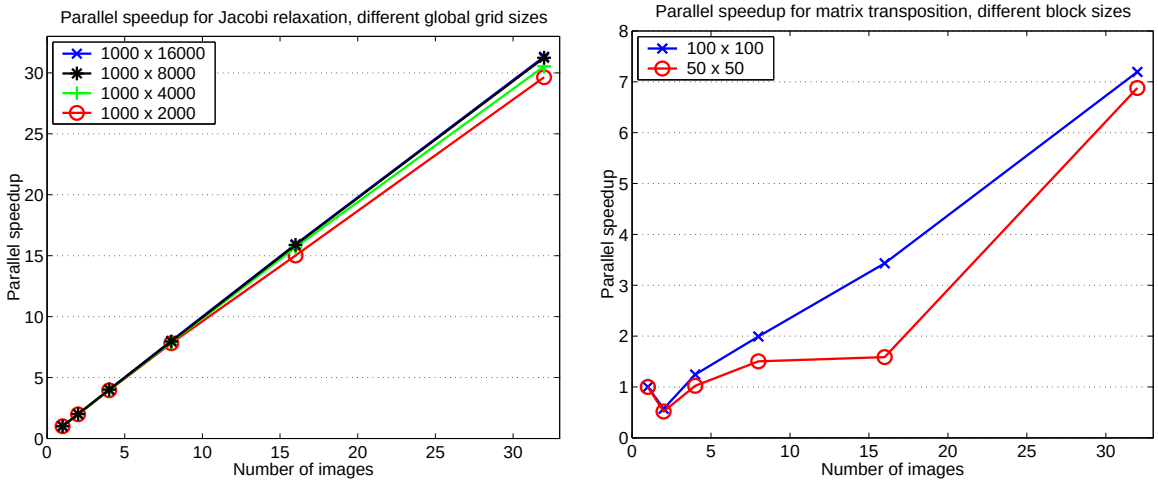


Figure 4: Transactions in a SYNC operation.



(a) (b)

Figure 5: Performance results. (a) Jacobi relaxation. (b) Matrix transposition.

References

[1] G. Almasi, G. S. Almasi, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castaños, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, A. Deutsch, M. B. Dombrowa, W. Donath, M. Eleftheriou, B. Fitch, J. Gagliano, A. Gara, R. Germain, M. E. Giampapa, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, A. P. Lanzetta, D. Lieber, M. Lu, M. Mendell, L. Mok, J. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, R. Rand, R. Regan, R. Sahoo, A. Sanomiya, E. Schenfeld, S. Singh, P. Song, B. D. Steinmacher-Burow, K. Strauss, R. Swetz, T. Takken, P. Vranas, and T. J. C. Ward. Cellular supercomputing with system-on-a-chip. In *Proceedings of ISSCC'02*, 2002.

[2] D. R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, 1997.

[3] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, Cambridge, MA, 2nd edition, 1999.

[5] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[6] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[7] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. *Fortran Forum*, 17, 1998.

[8] M. Snir, S. W. Otto, S. Hess-Lederman, D. Walker, and J. J. Dongarra. *MPI: The Complete Reference*, volume 1. MIT Press, Cambridge, MA, 2nd edition, 1998.

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, CA, Feb. 1998.