HPC Challenge Submission for X10

Ganesh Bikshandi[1] Jose Castanos [2] Sreedhar Kodali[1] Sriram Krishnamoorthy[2]
V. Krishna Nandivada[3] Igor Peshansky[2] Vipin Sachdeva[2] Vijay Saraswat[2] *
Mark Stephenson[2] Sayantan Sur[2] Pradeep Varma[3] Tong Wen[2]

[1] IBM Systems & Technology Group
Indira Nagar
Bangalore, 560071

[2] IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[3] IBM India Research Lab
Vasant Kunj
New Delhi, 110070

1. Introduction

X10 is a modern, high-level OO programming language designed to support high performance computations on a variety of concurrent architectures. Supported by the IBM-DARPA PERCS project, X10 is built on a few simple concepts extending Sequential Java – the notion of places (an encapsulation of data and activities operating on the data), asynchronous computation and communication, atomicity and ordering. X10 also includes some extensions to the sequential language, such as value types, and (value-)dependent types, and provides a framework for user-supplied annotations and compiler extensions, and for linking with natively-developed code.

The core concepts of X10 map well to a variety of modern architectures, e.g. a cluster of SMPs (connected with high-performance fabrics), high-end machines with global shared memory such as the Power7 machine (currently being designed), Blue Gene, as well as systems built from multi-core processors such as the Cell processor.

In this submission we present results for the four HPC Challenge Type 2 programs – Stream, Random Access, FT and LU – on three implementations of X10. We provide numbers for Stream, Random Access and FT running on a cluster of Power5 SMPs and a BG/L rack, and for LU running on a 64-way Power5 SMP on top of a Java runtime (with native BLAS). This submission differs from the X10 submission last year in that the programs are rewritten (the LU program is presented for the first time), and numbers are presented for a new implementation that produces C/C++ code and runs on a cluster of Power 5 SMPs and the Blue Gene machine. We show that all programs scale well, and LU outperforms HPL at 64 processors for large block-sizes.

2. X10 programming language

The X10 programming language is based on Sequential Java (Java less threads, synchronized, volatile, and arrays) and augments it with a few core concepts for concurrency and distribution, and extensions to the type system necessary to support these concepts.

The notion of *place* is central to X10. An X10 computation is run over a collection of places. A place represents a collection of data together with activities operating on the data. An activity is created in a place and stays at the place for the duration of its lifetime. Objects may have mutable state (such objects are called *reference* objects and are created as instances of *reference classes*), or they may not (such objects are called *value* objects are are created as instances of *value* classes). During its execution an activity can create data items (e.g. objects) in the place where it is executing. A reference object stays allocated at the place in which it is created (as long as some activity has a reference to it) and may be operated upon only by activities at that place. Value objects do not contain any mutable state and hence can be copied freely from place to place; the language permits such objects to be referenced from any place.

The collection of places constitutes a *partitioned global address space* since an object at one place can contain fields which reference objects that live in other places.

Special kinds of objects, known as *global objects*, may have state that spans multiple places. In X10 v 1.1 (the language discussed in this submission) the only pre-defined types of global objects are arrays. Arrays in X10 are quite rich – they are defined over a *region* (a collection of index points). Global arrays are defined over *distributions* (a colored region which colors each point in the region with the place where it is assigned). Further, X10 has a very rich *dependent type system* which permits types (such as arrays) to be qualified by constraints (or assertions) over the *properties* of the type. For instance the type `double[:rail]` is the type of all arrays of `double` which are `rail`s, i.e. are one-dimensional, rectangular, and zero-based. Such arrays can be implemented particularly efficiently through contiguous chunks of memory addressed with offset 0 through $N-1$, where N is the size of the array.

Activities may create other activities using the `async (P) S` statement, where `P` names a place and `S` is the state-

* Contact: vsaraswa@us.ibm.com

ment to be executed at that place. Constructs are provided to permit the simultaneous spawning of activities: the `foreach` construct permits an activity to be spawned for every point in a region, and the `ateach` construct permits an activity to be spawned at every point in a distribution (at the place to which the distribution maps that point). Termination of activities may be detected by the `finish` construct: thus the statement `finish S` terminates only when the statement `S` terminates and all activities launched during the execution of `S` themselves terminate (recursively).

X10 does not provide locks for mutual exclusion. Instead a very simple form of atomic transaction is supported. The statement `atomic S` requires that the statement `S` be executed as if in one uninterruptible and indivisible step. A conditional version of the statement – which blocks until the memory reaches a state in which a condition is true – is also provided and is the fundamental synchronization primitive in X10.

A powerful derived construct in X10 is the *clock*. Conceptually, a clock is simply a barrier (familiar from SPMD languages), but modified to work in a context in which activities may be dynamically spawned and may dynamically terminate in such a way that their basic operations are determinate. A new clock is constructed through the `clock.factory.clock()` expression. An activity *A* can be registered on a clock when it is created – indeed this is the only way it can be registered on an old clock (i.e. a clock that exists when it is created). The only other way that an activity can be registered on a clock is when it creates the clock. Activities are automatically deregistered from a clock when they terminate. At any given time an activity may execute a `next;` operation. This is both a signal to all other activities registered on the clock that it has finished whatever work it was supposed to finish in this phase of the clock and an action that causes the activity to suspend until all other activities registered on the clock have signalled that they have completed their work in the current phase. (There is provision for a split-phase barrier as well, but this shall not be needed for the examples in this paper.)

2.1 Global vs. fragmented data-structures

A central theme in the programs discussed in this submission is the notion of global vs fragmented data-structures. A global array is an array defined over a distribution which maps the points in its region to one or more places. For instance the following code fragment creates a global array:

```
region(:rail) R = [0:99];
dist(:rail) D = dist.factory.block(R);
final double[:rail] A = new double[D];
```

First it creates a region `R` defined as the collection of points `0`, `1`, ..., `99`. The variable `R` is declared to be rail. Next, a new distribution `D` is created. It specifies that the 100 points of `R` are to be distributed over the set of places that this computation is executiing in a block-distributed fashion. (That is, the first `N/P` points are allocated to the first place, the next set to the next place, and so on. This allocation is adjusted to evenly distribute points that are left over (if any), to the first few places.) Finally it defines an array `A` that is defined over `D` and allocates a `double` for every point in `D`. Note that the creation of `A` automatically implies communication across all the places to which `D` maps its points. In particular, the given amount of memory is allocated at these places, appropriately initialized and tied to the global representation of `A`. We say that `A` is a global data-structure because any activity spawned after the creation of `A` (in a lexical scope in which `A` is visible) may reference `A`, even if it is executing at a place different from the one at which `A` was created.

Often the programmer may find it convenient to represent an array in a *fragmented* fashion. In this fashion, a single conceptual array is implemented as a collection of local arrays, one per place. Sometimes a second "backbone" array may be allocated with a so-called "unique" distribution (one point mapped to a place). Each element of this array may itself be a local array; thus one gets a two-tiered array, with each leaf being a completely local array.

X10 v 1.1 specifies that arrays may be indexed only with ints, i.e. they may contain only 2GB elements. This has turned out to be a serious limitation on large arrays, particularly when it is desired to scale a computation to a large number of nodes. We shall be removing this limitation in the next major revision of the language (v 1.5, in Summer 2008). For the time being programmers may find it convenient to use local arrays. Since each local array can have upto 2GB elements, and there may be as many local arrays as there are places (there may be upto 2GB places), it becomes possible to have very large arrays.

Below we shall consider programs with both kinds of arays.

3. Implementation

We are developing two implementations of X10, (1) X10 JVM designed for the Java tool-chain, and (2) X10 Flash, designed for the C/C++ tool-chain.

Both implementations are structured as traditional compilers producing code that uses a run-time system. The runtime system is structured as a collection of library classes (X10lang) implementing the X10 object model as well as a core set of classes, and as a runtime (X10lib) responsible for managing multiple activities in a place, and for communication between places.

3.1 X10 compilers

The X10 JVM and X10 Flash compiler share a substantial amount of code. Both are written in Polyglot, a Java-based framework for compiler development, and use a fairly standard multi-pass structure. A parser (developed using LPG, an open-source parser generator) is used to produce Polyglot ASTs from X10 source. These ASTs are then visited in several passes for disambiguation, type-checking, analysis, transformation etc. Finally a code-generation pass is used to output one or more files of source code. The two compilers differ essentially in this phase: the X10 JVM compiler produces Java source files whereas the X10 Flash compiler produces C/C++ source files.

3.2 X10 runtimes

The X10 JVM runtime is a collection of libraries written primarily in Java. Currently this implementation does not support X10 computations across multiple operating system processes, i.e. the X10 program must run in a single JVM instance. Any standard VM may be used. We have run X10 programs on Linux, AIX, Windows and MacOSX machines.

The X10 Flash runtime is written in C/C++ and is based on pthreads and the IBM Low-level API (LAPI) for communication. LAPI is a low-level API that supports the notion of active messages and remote direct memory access across a high-performance network. As such, it is a close match for X10 language constructs. On IBM Power5 SMPs, LAPI is the lowest user-programmable communication interface; MPI is implemented through LAPI. Implementations of LAPI are available on AIX and Linux, supporting the IBM HPS switch and Infiniband.

3.2.1 X10 on Power

Through the X10 Flash implementation, X10 programs may run on a cluster of Power5 SMPs running AIX and POE and connected through an HPS switch. The person running a program may use a configuration file to specify the number of places in a particular run of the program, and the number of threads per place. The compilation and linking process produces a binary that is launched at the selected nodes using POE.

3.2.2 X10 on Blue Gene

The Blue Gene/L and Blue Gene/P port of X10 is built on top of the initial implementation of X10lib. We have implemented a LAPI port on top of a new low level Blue Gene communications library called DCMF ("Deep Computing Messaging Framework"). DCMF is mainly a research code in Blue Gene/L, developed to explore a variety of communications paradigms. DCMF is also the standard low level communications library in Blue Gene/P and supports higher level message libraries such as MPI, ARMCI and the UPC runtime. Like all Blue Gene communications libraries, DCMF is a user-space library, and relies on the characteristics of the Blue Gene networks such as guaranteed delivery of messages by the hardware, partition of messages into small self-contained packets, low latency, torus interconnect and high ratio of bandwidth to processor speed.

The decision to base the X10lib port for Blue Gene on top of the LAPI API has the major advantage of abstracting the X10 development from the hardware. It allows us to run unmodified X10 programs on both Blue Gene and Power systems, and compare their performance. It also simplifies tracking the evolution of the X10 environment. But it also has the disadvantage of not utilizing some Blue Gene features (such as the global network) which are not easily expressed through LAPI calls.

Program	Line count
Stream	47
Fragmented Stream	51
Random Access	79
FT	137
LU	291

Table 1. Line count

On Blue Gene/L, we run X10 programs on the compute nodes, using the light-weight kernel in the "Virtual Node Mode": every node is partitioned into two equivalent processes, because Blue Gene/L does not easily support mulithreaded programs. The results for RandomAccess, Stream and FFT presented later use this mode, which effectively partitions in half the 512MB main memory in each node. We also used the GNU toolchain for compilation, which does not take advantage of several optimizations in the dual floating point unit. This was fundamentally the same environment we used for the UPC submission for Class 2 last year, and the performance results are comparable up to 1 rack.

One of the main challenges we encountered when porting X10 in to Blue Gene was the difficulty of indexing very large global arrays in a 32 bit architecture (such as Blue Gene), and for that reason, we limited the sizes of the matrices we run on large systems. Although it is possible to reconfigure indexing in X10 to use 64-bit integers, this is not a natural representation for Blue Gene. On the other hand, we experimented with versions of HPCC problems that replace the global arrays with set of local arrays ("fragmented arrays"). These versions of the code can access the whole memory of Blue Gene irrespective of the number of nodes as they do not use global addressing, and achieve slightly better performance (10% in Stream for example) than the results presented in this paper.

4. Performance

The line counts for the benchmark programs are given in Table 1.

For this submission we provide numbers on the V20 cluster at the IBM Poughkeepsie Benchmarking Center. The cluster contains over 128 dedicated p575+ Dual core 1.9GHz 16 CPU nodes, each with 64GB DDR2 memory. The nodes are connected by a Dual Plane HPS Switch. The tests are run with SMT off and large page size (2GB). The tests are run on AIX 5.3 TL5, using POE 4.3, Load Leveler 3.4, RSCT 2.4 (LAPI). The code produced by the X10 compiler is compiled by `mpCC` using the flags `-q64 -O3 -qarch=pwr5 -qtune=pwr5 -qhot -qinline`.

For the runtime, the following environment variable settings are used:

```
MP_EUILIB=us, MP_CSS_INTERRUPT=no,  MP_SINGLE_THREAD=yes
```

A full listing of environment variables can be made available to the committee on request.

We also provide numbers on a single rack of BG/L. By the time of SC, we expect to run numbers on larger configurations.

Acknowledgements. We gratefully thank the LAPI team (particularly Hanhong Xue, Chulho Kim and Robert Blackmore) for their support. We thank Calin Cascaval for making available UPC source code for a previous HPC Challenge submission. We thank John Gunnels for informative discussions about LU. This material is based upon work supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

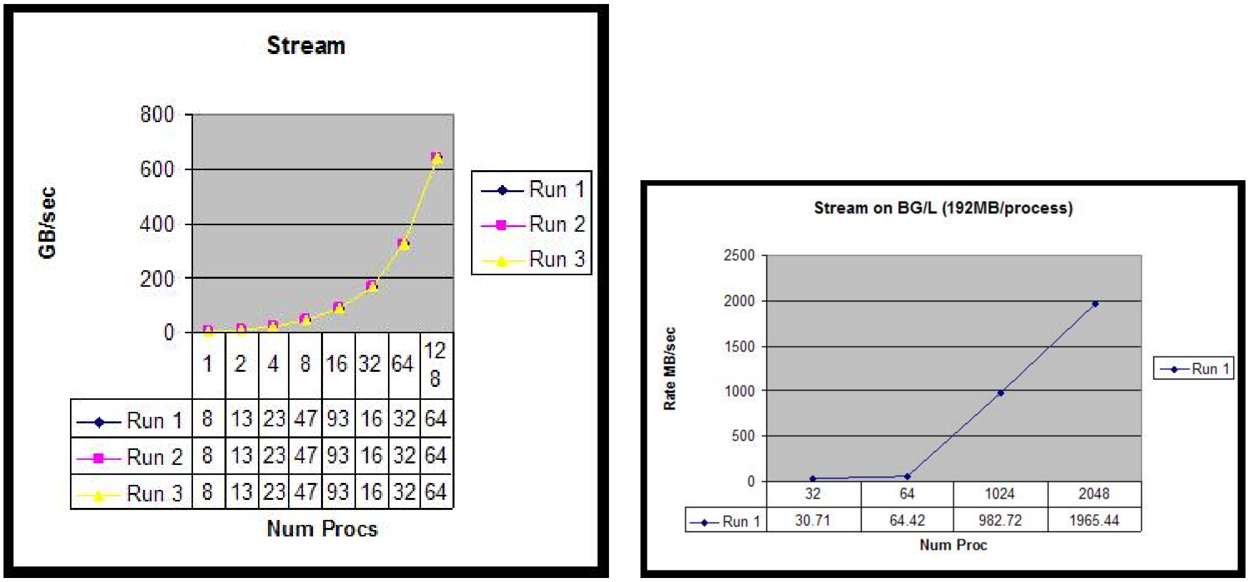

Figure 1. Stream over 32 Power5 nodes and a BG/L rack

A. Stream

We show below two versions of the Stream program, one with global arrays (Stream) and the other with fragmented arrays (FragmentedStream).

A.1 Stream program

The program is quite straightforward. When the program is launched, a single activity is started at place 0 executing the `main` method of the given class. This activity creates three global `double` rails (1-d, zero-based, rectangular arrays) are created, `a`, `b` and `c` and appropriately initialized (lines 12-15). Recall that the creation of these rails may involve communication between places. The activity then launches a child activity at each place (using the `ateach` construct), each of which is registered on a newly created clock (created by an intermediary activity that dies immediately after creating the clock and spawning the `ateach` statement). This is the typical way in which an "SPMD" pattern is expressed in X10.

Each activity now executes the loop (lines 19-24) `NUM_TIMES` times. In each iteration of the loop the given triad operation is performed for every point in the underlying distribution `D` that is mapped to the current place (notated as `here` in X10). After each activity has performed the triad operation for every element in its domain, it enters the barrier. The activity at place 0 measures the time difference between the time it starts to execute its first triad operation and the time at which all activities have finished their triad operations.

The best time is reported across all iterations. The computation is verified by performing the same operation again and comparing the result.

Performance of the program on Power5 cluster and on the BG is given by Figure 1. When using Stream on 2K nodes it was necessary to reduce the size of the arrays, since the global array cannot contain more elements than can be addressed by 32 bits. This problem is not faced with the fragmented array representation discussed below.

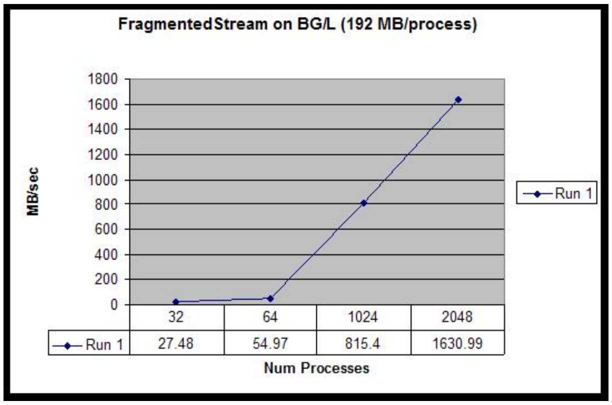

Figure 2. Fragmented Stream on a BG/L rack

```
public class Stream {
  const int MEG=1024*1024, NUM_TIMES=10;
  const double alpha=3.0D;
  public static void main(String[] args) {
    final boolean[] verified = new boolean[] { true };
    long N0=2*MEG;
    if (args.length > 0) N0 = java.lang.Long.parseLong(args[0]);
    final long N=N0*place.MAX_PLACES;
    final region(:rail) R=[0:(int) (N-1)];
    final dist(:rail) D=dist.factory.block(R);
    final double[] times =new double[NUM_TIMES];
    final double[:rail]
      a = new double[D],
      b = new double[D] (point [i]) {return 1.5*i;},
      c = new double[D] (point [i]) {return 2.5*i;};
    finish async {
      final clock clk=clock.factory.clock();
      ateach(point [i]:dist.UNIQUE) clocked (clk) {
        for (int j=0;j<NUM_TIMES; j++) {
          if (i==0) times[j]= -mySecond();
          for (point [p]:D|here) a[p]=b[p]+alpha*c[p];
          next;
          if (i==0) times[j] += mySecond();
        }
        for (point [p]:D|here) // verification
          if (a[p] != b[p]+alpha* c[p])
            async(place.FIRST_PLACE) clocked (clk) verified[0]=false;
      }
    }
    double min=10000000L;
    for (int j=0; j<NUM_TIMES; j++) if (times[j]<min) min=times[j];
    printStats(N, min, verified[0]);
  }
  public static double mySecond() {
    return (double) ((double)(System.nanoTime() / 1000) * 1.e-6);
  }
  public static void printStats(long N, double time, boolean verified) {
    System.out.println("Number of places=" + place.MAX_PLACES);
    long size = (3*8*N/MEG);
    double rate = (3*8*N)/(1.0E9*time);
    System.out.println("Size of arrays: " + size +" MB (total)"
               + size/place.MAX_PLACES + " MB (per place)");
    System.out.println("Min time: " + time + " rate=" + rate + " GB/s");
    System.out.println("Result is "
               + (verified ? "verified." : "NOT verified."));
  }
}
```

A.2 Fragmented stream

The fragmented version of the program is given below. The major difference is that at each place the current activity allocates a separate local array (a on line 17). Since this program involves no communication there is no need for an activity running at one place to reference an array allocated at another place. Otherwise the structure of the program is identical to the Stream program discussed above. Note that each local array can be as big as 2G; therefore the total size of all arrays allocated can grow unboundedly with the number of places.

The code above shows an improvement in performance over Figure 1 because a check in the main loop is dropped.

```
1  public class FragmentedStream {
2      const int MEG=1024*1024;
3      const double alpha=3.0D;
4      const int NUM_TIMES=10;
5      public static void main(String[] args) {
6          final boolean[] verified = new boolean[] { true };
7          final double[] times = new double[NUM_TIMES];
8          long N0 = 2*MEG;
9          if (args.length > 0) N0 = java.lang.Long.parseLong(args[0]);
10         final long N=N0*place.MAX_PLACES;
11         final int LocalSize = (int) N0;
12         System.out.println("LocalSize=" + LocalSize);
13         final region(:rank==1&&zeroBased&&rect)  RLocal=[0:LocalSize-1];
14         finish async {
15             final clock clk=clock.factory.clock();
16             ateach(point [p]:dist.UNIQUE) clocked (clk) {
17                 double[] a = new double[LocalSize],
18                          b = new double[LocalSize],
19                          c = new double[LocalSize];
20                 for (point [i] : RLocal) {
21                     b[i] =1.5*(p*LocalSize+i);
22                     c[i] = 2.5*(p*LocalSize+i);
23                 }
24                 for (int j=0;j<NUM_TIMES; j++) {
25                     if (p==0) times[j]= -mySecond();
26                     for (point [i]:RLocal ) a[i]=b[i]+alpha*c[i];
27                     next;
28                     if (p==0) times[j] += mySecond();
29                 }
30                 for (point [i]: RLocal) // verification
31                     if (a[i] != b[i]+alpha* c[i])
32                         async(place.FIRST_PLACE) clocked (clk) verified[0]=false;
33             }
34         }
35         double min=10000000L;
36         for (int j=0; j<NUM_TIMES; j++) if (times[j]<min) min=times[j];
37         printStats(N, min, verified[0]);
38     }
39     public static double mySecond() {
40         return (double) ((double)(System.nanoTime() / 1000) * 1.e-6);
41     }
42     public static void printStats(long N, double time, boolean verified) {
43         System.out.println("Number of places=" + place.MAX_PLACES);
44         long size = (3*8*N/MEG);
45         double rate = (3*8*N)/(1.0E9*time);
46         System.out.println("Size of arrays: " + size +" MB (total)"
47                            + size/place.MAX_PLACES + " MB (per place)");
48         System.out.println("Min time: " + time + " rate=" + rate + " GB/s");
49         System.out.println("Result is " + (verified ? "verified." : "NOT verified."));
50     }
51 }
```

B. Random Access

The Random Access program is equally straightforward to write in X10. Here we present a global array version of the program.

The core computation is performed by the method `RandomAccessUpdate` in lines 31-46. The main activity launches a `finish ateach` operation, with one activity created per place, after creating the global rail `Table`. Each of these activities runs through a loop creating a new update `ran`. It then computes the destination of this update and launches an `async` to atomically update an element of `Table` at the destination. Note that there is no need for this activity to wait for termination

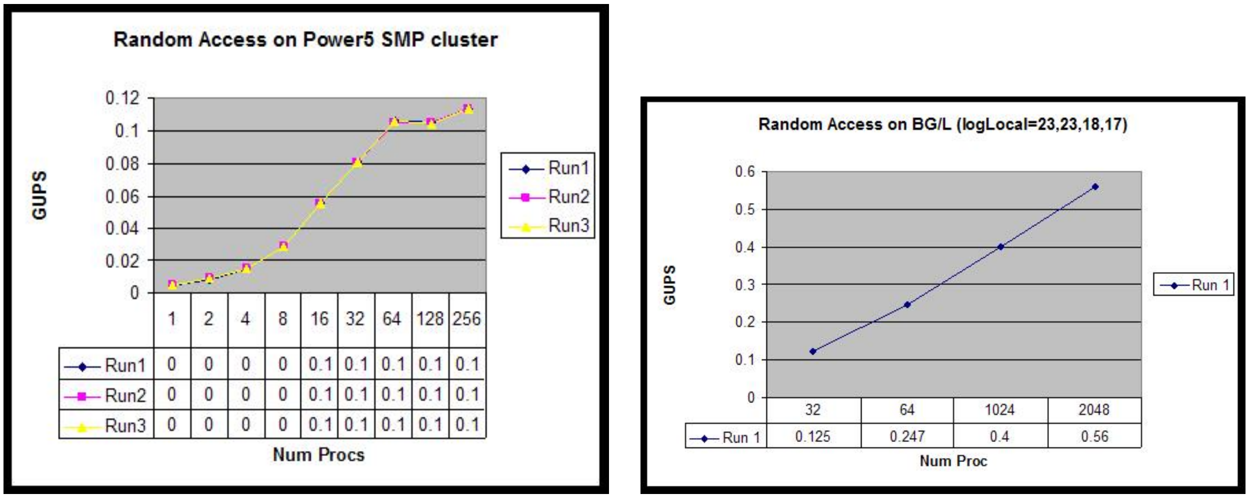

Figure 3. Random Access over 32 Power5 nodes and a BG/L rack

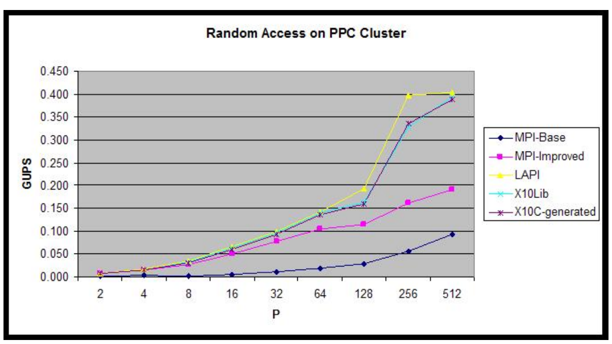

Figure 4. Fragmented Random Access over Power5 (program not included)

of this `async`; it merely goes on to compute the next value of `ran` and launch another `async`. Computation terminates (as detected by the `finish`) when all activities at all places have finished their loops and terminated (and all activities that they have launched have terminated). Thus this simple program specifies quite a complex pattern of communication, ordering and atomicity.

The annotation `@aggregate` is worth a remark. X10 permits a general annotation facility – almost any syntactic element in X10 can be annotated. Annotations can be processed by code supplied by the programmer and run in the compiler and/or runtime system. In this case, the `@aggregate` annotation indicates to the runtime that it may be worthwhile for the runtime to aggregate `asyncs` sent to the same destination. The limit of aggregation is specified by an environment variable.

The performance on BG/L is essentially the same as the performance for UPC reported at SC 06.

We have developed a fragmented array version of this benchmark as an internal milestone. An extensive study has been performed on this application. Performance is detailed in Table 4. Note that the performance is substantially better than MPI performance (of the "unoptimized" algorithm). This is due to better support for small messages in the X10 runtime which is built directly over LAPI. An implementation of the "software routing" scheme of Garg and Sabharwal for BG is under development in X10.

```
package ra;
class RandomAccess {
  static double mySecond() { return (double) ((double)(System.nanoTime() / 1000) * 1.e-6);}
  const long POLY = 0x0000000000000007, PERIOD = 1317624576693539401L;
  const int NUM_PLACES = place.MAX_PLACES, PLACE_ID_MASK = NUM_PLACES-1;
  /* Utility routine to start random number generator at Nth step */
  static long HPCC_starts(long n) {
      int i, j;
      long[] m2 = new long[64];
       while (n < 0) n += PERIOD;
       while (n > PERIOD) n -= PERIOD;
       if (n == 0) return 0x1;
       long temp = 0x1;
       for (i=0; i<64; i++) {
           m2[i] = temp;
           temp = (temp << 1) ^ (temp < 0 ? POLY : 0L);
           temp = (temp << 1) ^ (temp < 0 ? POLY : 0L);
       }
       for (i=62; i>=0; i--) if (((n >> i) & 1) != 0) break;
       long ran = 0x2;
       while (i > 0) {
           temp = 0;
           for (j=0; j<64; j++) if (((ran >> j) & 1) != 0) temp ^= m2[j];
           ran = temp;
           i -= 1;
           if (((n >> i) & 1) != 0)
               ran = (ran << 1) ^ ((long) ran < 0 ? POLY : 0);
       }
       return ran;
   }
   static void RandomAccessUpdate(final int logLocalTableSize, final long[:rail] Table) {
       finish ateach(point [p] : dist.UNIQUE) {
           final long localTableSize=1<<logLocalTableSize,
                TableSize=localTableSize*NUM_PLACES,
                mask=TableSize-1,
                NumUpdates=4*localTableSize;
           long ran=HPCC_starts(p*NumUpdates);
           for (long i=0; i<NumUpdates; i++) {
               final long temp=ran;
               final int index = (int)(temp & mask);
               @aggregate async (dist.UNIQUE[index/(int)(TableSize/NUM_PLACES)])
                 atomic Table[index] ^= temp;
               ran = (ran << 1)^((long) ran < 0 ? POLY : 0);
           }
       }
   }
   public static void  main(String[] args) {
       int  logLocalTableSize= 10;
       if (args.length > 1 && args[0].equals("-m")) {
           logLocalTableSize = java.lang.Integer.parseInt(args[1]);
       }
       /* calculate the size of update array (must be a power of 2) */
       final int LogLocalTableSize = logLocalTableSize;
       final long LocalTableSize = 1<<logLocalTableSize,
            TableSize = LocalTableSize*NUM_PLACES;
       final region(:rail) R = [0:(int)TableSize-1];
       final dist(:rail) D = dist.factory.block(R);
       final long[:rail] Table = new long[D] (point [i]) {return i;};
       System.out.println("Main table size   = 2^" +LogLocalTableSize + "*"
               +NUM_PLACES+" = " + TableSize+ " words");
       System.out.println("Number of places = " + NUM_PLACES);
       System.out.println("Number of updates = " + (4*TableSize)+ "");
       double cpuTime = -mySecond();
```

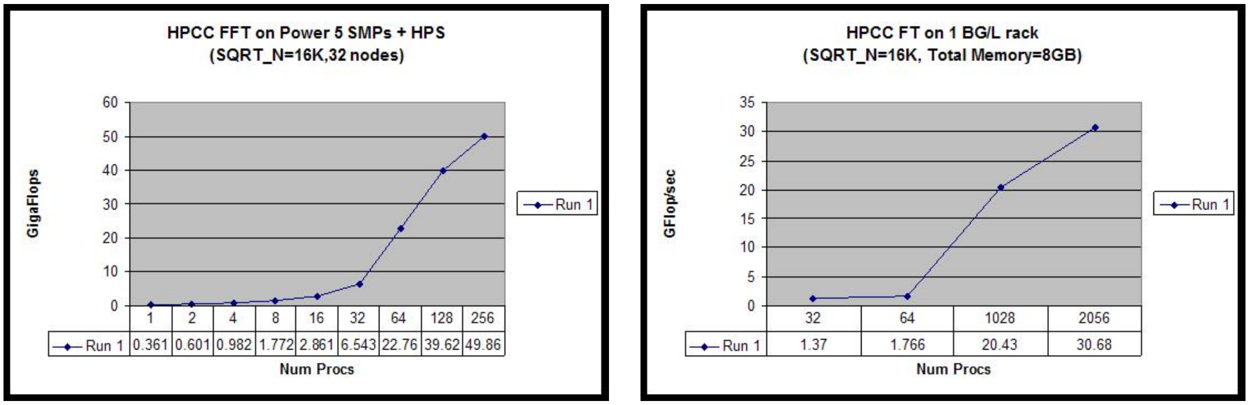

Figure 5. HPCC 1D-FFT over 32 Power5 nodes and a BG/L rack

```
        RandomAccessUpdate(LogLocalTableSize, Table );
        cpuTime += mySecond();
        double GUPs = (cpuTime > 0.0 ? 1.0 / cpuTime : -1.0)*4*TableSize/1e9;
        System.out.println("CPU time used  = "+cpuTime+" seconds");
        System.out.println(GUPs + " Billion(10^9) Updates per second [GUP/s]");
        RandomAccessUpdate( LogLocalTableSize, Table ); // repeat for testing.
        final long[.] t  = new long[dist.UNIQUE](point [p]) {
            long err = 0;
            for(point [q]:Table.distribution|here) if (Table[q] != q) err++;
            return err;
        };
        long temp = t.sum();
        System.out.println("Found " +temp+  " errors in " +TableSize+ " locations.");
        if (temp > 0.01*TableSize) System.err.println("Failure.");
    }
}
```

C. FT

The code for FT is listed below (the HPCC 1D complex FFT) and is essentially straightforward. This code highlights the use of extern libraries (in this case the FFTW library) by X10 code. (See `extern` statements in Lines 6-11). Note that the `bytwiddle` routine does not consult a cached array of twiddle factors, instead these are computed on the fly. (It is quite straightforward in X10 to cache these in an array. This would have a significant impact on performance.)

The only novelty is in the `transpose` method which performs a local transpose of each block before spawning an `async` task to copy the block to the target destination. The actual copying is done by an intrinsic `arrayCopy`; however the activity is controlled with the usual X10 constructs – in this case the `finish` at line 34 will not progress until all the activities launched by all SPMD activities to copy have terminated.

The performance numbers for the HPCC 1D-FFT are given in Figure 5.

```
import java.util.*;
value class fftDist {
    const int NUM_PLACES = place.MAX_PLACES;
    const dist(:unique) UNIQUE = dist.UNIQUE; //no cast
    const int FFTW_MEASURE = 0;
    public static extern void executedft(long plan, double[:rail] A, int i0, int i1);
    public static extern long fftw_plan_dft_1d(int SQRTN, double[:rail] A1,
                    double[:rail] A2, int m1, int what);
    public static extern void start(int sqrtn, int n);
    public static extern double getPI();
    static { System.loadLibrary("fftDist"); } // load fftw library
    static void row_ffts(final int SQRT_N, final int N, final double[:rail] A,
                    final long [.] fftw_plans) {
        finish ateach (point [p]: UNIQUE)
            executedft(fftw_plans[p], A.local(), 0, SQRT_N/NUM_PLACES);
    }
    static void bytwiddle(final int SQRTN, final int N, final double M_PI,
                    final double[:rail] A, final double sign) {
        finish ateach(point [p]: UNIQUE) {
            int numLocalRows = SQRTN/NUM_PLACES;
            region R2d = [p*numLocalRows:(p+1)*numLocalRows-1,0:SQRTN-1];
            double W_N = 2.0*M_PI/N;
            for (point [i,j]: R2d) {
                int ij =i*j, idx =2*(i*SQRTN+j);
                double ar = A[idx], ai = A[idx+1];
                double c = Math.cos(W_N*ij), s=Math.sin(W_N*ij)*sign;
                A[idx] = ar*c+ai*s;
                A[idx+1] = ai*c-ar*s;
            }
        }
    }
    static void transpose(final int SQRTN, final int N, final double[:rail] Y,
                    final double[:rail] Z) {
        finish ateach(point [p]: UNIQUE) {
            final int numLocalRows = SQRTN/NUM_PLACES;
            int rowStartA = p*numLocalRows; // local transpose
            region block = [0:numLocalRows-1,0:numLocalRows-1];
            for (int k=0; k<NUM_PLACES; k++) { //for each block
                int colStartA = k*numLocalRows;
                for (int i=0; i<numLocalRows; i++)
                    for (int j=i; j<numLocalRows; j++) {
                        int idxA = 2*(SQRTN * (rowStartA + i) + colStartA + j),
                            idxB = 2*(SQRTN * (rowStartA + j) + colStartA + i);
                        double tmp0 = Y[idxA], tmp1 = Y[idxA+1];
                        Y[idxA] = Y[idxB]; Y[idxA+1] = Y[idxB+1];
                        Y[idxB] = tmp0; Y[idxB+1] = tmp1;
                    }
                for (int i=0; i<numLocalRows;i++) { // now copy
                    final int srcIdx = 2*((rowStartA + i)*SQRTN+colStartA),
                        destIdx = 2*(SQRTN * (colStartA + i) + rowStartA);
                    async (UNIQUE[k])
                        Runtime.arrayCopy(Y, srcIdx, Z, destIdx, 2*numLocalRows);
                }
            }
        }
    }
    static void check(final int SQRTN, final int N, final double[:rail] A0,
                     final double[:rail] A) {
        final double epsilon = 1.0e-15;
        final double threshold = epsilon*Math.log(N)/Math.log(2)*16;
        finish ateach(point [p]: UNIQUE)
            for (point [q]:A.distribution|here)
                if (Math.abs(A[q]/N-A0[q])> threshold)
                    System.err.println("Error at "+q+" "+A[q]/N+" "+A0[q]);
    }
    static void solve(final int SQRT_N, final boolean reportTime) {
        final int N = (SQRT_N * SQRT_N);
        final int numLocalRows = SQRT_N/NUM_PLACES;
        if (numLocalRows*NUM_PLACES != SQRT_N) {
            System.err.println("SQRT_N must be divisible by NUM_PLACES!");
            System.exit(-1);
        }
```

```
        final region(:rail) R = [0:2*N-1];
        final dist(:rail) D = dist.factory.block(R);
        final double[:rail] A = new double[D], B0 = new double[D], B = new double[D];
        final long [.] fftw_plans = new long [UNIQUE], fftw_inverse_plans = new long [UNIQUE];
        final double M_PI = getPI();
        final long[] tms = new long[7];
        finish ateach (point [p] : UNIQUE) {
            fftw_plans[p] = fftw_plan_dft_1d(SQRT_N, A.local(), A.local(), -1, FFTW_MEASURE);
            fftw_inverse_plans[p] = fftw_plan_dft_1d(SQRT_N, A.local(), A.local(), 1, FFTW_MEASURE);
            final Random rnd=new Random();
            for (point [i]: (D|here).region) {
                A[i]= rnd.nextDouble()-0.5;
                B0[i]=A[i];
            }
            start(SQRT_N, N);
        }
        tms[0]=System.nanoTime(); transpose(SQRT_N, N, A, B);
        tms[1]=System.nanoTime(); row_ffts(SQRT_N, N, B, fftw_plans);
        tms[2]=System.nanoTime(); transpose(SQRT_N, N, B, A);
        tms[3]=System.nanoTime(); bytwiddle(SQRT_N, N, M_PI, A, 1.0);
        tms[4]=System.nanoTime(); row_ffts(SQRT_N, N, A, fftw_plans);
        tms[5]=System.nanoTime(); transpose (SQRT_N, N, A, B);
        tms[6]=System.nanoTime(); // now starting inverse FFT for verification
        transpose(SQRT_N, N, B, A);
        row_ffts(SQRT_N, N, A, fftw_inverse_plans);
        transpose(SQRT_N, N, A, B);
        bytwiddle(SQRT_N, N, M_PI, B, -1.0);
        row_ffts(SQRT_N, N, B, fftw_inverse_plans);
        transpose(SQRT_N, N, B, A);
        check(SQRT_N, N, B0, A);
        if (reportTime) {
            System.out.println("After verification");
            double secs = ((double)(tms[6] - tms[0])*1.0e-9);
            double Gigaflops = 1.0e-9*N*5*Math.log(N)/Math.log(2)/secs;
            double mbytes = N*2.0*8.0*2/(1024*1024);
            System.out.println("execution time = " + secs + " secs"+" Gigaflops = "+Gigaflops);
            System.out.println("SQRT_N = " + SQRT_N + ", p = " + place.MAX_PLACES);
            System.out.println("N = " + N + " N/place=" +N/place.MAX_PLACES);
            System.out.println("Mem = " + mbytes + " mem/place = " +mbytes/place.MAX_PLACES);
            String[] nms = new String[] { "transpose1", "row_ffts1", "transpose2",
                                          "twiddle", "row_ffts2",  "transpose3"};
            for (int i = 1; i < tms.length; ++i)
                System.out.println("Step " + nms[i-1] + " took " +
                                   ((double)(tms[i]-tms[i-1])*1.0e-9)+" s");
        }
    }
    static void printArray(final String name, final int SQRT_N, final double[:rail] A) {
        System.err.println("Array "+name+":");
        finish ateach (point [p] : dist.UNIQUE) {
            for (point [q]:(A.distribution|here)) {
                final int i = q/SQRT_N/2;
                final int j = (q % (2*SQRT_N))/2;
                System.err.println(here.id+" ("+ i +"," + j + ") "+ q + ": " + A[q]);
            }
        }
    }
    public static void main(String[] args) {
        System.out.println("Warming up");
        solve(4, false); // warm up
        System.gc();
        System.out.println("Starting computation");
        solve(256, true);
    }
}
```

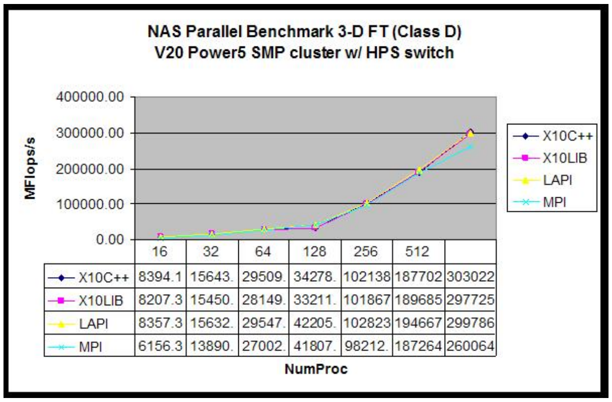

Figure 6. NAS PB FT Class D runtimes on Power5

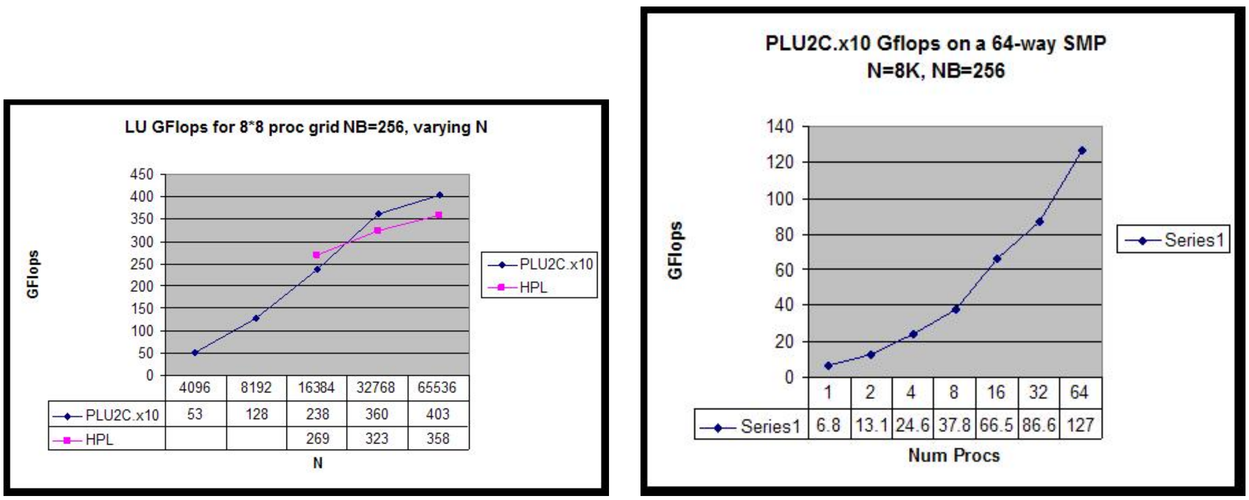

Figure 7. Performance of PLU (N=8K,NB=256)

C.1 NAS parallel benchmark 3D FFT

As a point of comparison, we offer results from a separate X10 implementation of the 3D-NAS PB FFT benchmark. This benchmark was written with fragmented arrays and utilized overlap between computation and communication, using the idea of "pencils" developed by the Berkeley UPC group.

The NAS PB tests were run on 160 compute nodes (only 16 were used). Each node is a 1.9GHz, 16-CPU P575+ with 64GB DDR2 RAM, configured with SMT off, and with 50% large pages. The test was run on AIX 5.3, POE 4.3, VAC 8.0 and LAPI 1.39.16.

D. LU

The LU program in X10 turned out to be remarkably easy to write. LU is notorious for having a data-dependent communication structure. However, the flexibility of X10's `finish` and `async` constructs make it very easy to express this program and – remarkably – obtain quite good performance.

The central idea behind this algorithm is influenced by the work of Parry Husband on the LU implementation in Berkeley UPC. We present a shared memory implementation of this idea in X10 in very few lines of code. The idea is to represent the 2-d matrix as a collection of blocks, block distributed over a px x py grid of *workers*. Each worker is an activity, all workers run in the same place, the entire array is allocated in one place. Each worker examines the blocks allocated to it in a column-first order (this order is important). On examining each block it determines whether there is any work to do on this block. There is work to do if the dependencies that this block has been waiting have been resolved. (In detail, these dependencies are indicated by the status of certain `@shared` variables associated with each block, viz. `ready`, `rbCount` and `LUCol`.) For instance if the corresponding block to the left of a given block and the block above a given block are ready, then the block may perform a DGEMM.

It is critical to the good performance of this algorithm that work on the critical path be given priority over background work. This is accomplished by giving preference to blocks towards the left and top. The worker always examines the blocks that are assigned to it (and known to not be ready – once a block is ready it is never going to be changed and is not examined again by the worker) in top-down left-to-right order. However, as soon as a block is found which has a piece of work associated with it that is ready to performed, the work is performed and then the worker *returns* to the very first block. This gives significant priority to the lower numbered blocks and ensures that they will progress as soon as a piece of work is ready.

Several improvements on the program presented below are possible. For instance it has been suggested to us that chunking many blocks into a single DGEMM call will increase performance. We plan to investigate these in future work.

We note the program presented below deviates from the specification in a few respects. It computes the LU decomposition (leaving L unpivoted), and then validates the result by multiplying L an U to compute A' and computing the difference $A' - A$. This means that the lower order N^2 calculations required by the specification are not performed. We adjust the GFlop count for the X10 program to reflect this discrepency.

D.1 Performance

The resulting X10 program is compiled by the `x10c` compiler and run in the X10 runtime in Java, i.e. on a single JVM instance, running on a 64-way Power5 SMP. On such a machine, the X10 program shows performance comparable to that of an appropriately configured HPL program. We note that the X10 program exhibits very good scaling.

We show two graphs. The first (left graph in Figure 7) shows the performance of the X10 program compared with an HPL program, running on a grid of 8×8 processors, with a block size of 256, varying N. (Note that two measurements for HPL are missing.) The parameters for HPL were chosen according to published guidelines, and included a depth of 1. This graph shows that the X10 program outperforms HPL for larger values of N.

The second graph (right graph in Figure 7) shows the scaling of the X10 program from 1 to 64 processors, for a given choice of N and NB. Similar graphs have been obtained for higher values of N.

Note that the X10 program is fairly small – under 300 lines of code.

```
import java.util.concurrent.atomic.*;
public class PLU2_C(int px, int py, int nx, int ny, int B) {
    public static final int NUM_TESTS=10, LOOK_AHEAD=6, RAND_MAX=100;
    final AtomicIntegerArray pivots;
    final int bx,by,N;
    final Blocks[] A;
    public PLU2_C(int px_, int py_, int nx_, int ny_, int B_) {
        property(px_,py_,nx_,ny_,B_);
        bx=nx*px; by=ny*py; N=bx*B;
        assert bx==by;
        pivots = new AtomicIntegerArray(N);
        for(int i=0; i<pivots.length(); i++) pivots.set(i,i);
        A = new Blocks[px*py];
        for (int i=0; i < px*py; i++) A[i] = new Blocks(nx*ny);
        for(int pi=0; pi<px; ++pi)
            for(int pj=0; pj<py; ++pj)
                for(int i=0; i<nx; ++i)
                    for(int j=0; j<ny; ++j)
                        A[pord(pi,pj)].z[lord(i,j)] = new Block(i*px+pi, j*py+pj);
    }
    public PLU2_C(PLU2_C o) {
        property(o.px,o.py,o.nx,o.ny,o.B);
        bx=o.bx; by=o.by; N=o.N;
        pivots = o.pivots; // pivots are shared.
        A = new Blocks[px*py];
        for (int i=0; i<px*py;++i) A[i] = new Blocks(nx*ny);
        for(int pi=0; pi<px; ++pi)
            for(int pj=0; pj<py; ++pj)
                for(int i=0; i<nx; ++i)
                    for(int j=0; j<ny; ++j)
                        A[pord(pi,pj)].z[lord(i,j)] = new Block(o.getLocal(pi, pj, i, j));
    }
```

```
    static double format(double v, int precision){
        int scale=1;
        for(int i=0; i<precision; i++) scale *= 10;
        return ((int)(v*scale))*1.0/scale;
    }
    static int max(int a, int b) { return a>b?a:b; }
    static double max(double a, double b) { return a>b?a:b;}
    static double fabs(double v){ return  v>0?v:-v; }
    static int min(int a, int b) { return a>b?b:a; }
    static double flops(int n) { return ((4.0*n-3.0)*n-1.0)*n/6.0; }
    boolean in(int x, int low, int high) { return low <= x && x < high;}
    int pord(int i, int j) { return i*py+j;     }
    int lord(int i, int j) { return i*ny+j; }
    Block getBlock(int i, int j) { return A[pord(i%px, j%py)].z[lord(i/px,j/py)]; }
    Block getLocal(int pi, int pj, int ni, int nj) {
        return A[pord(pi,pj)].z[lord(ni,nj)];
    }
    class Blocks {
        final Block[] z;
        Blocks(int n) { z=new Block[n]; }
    }
    PLU2_C applyPivots(boolean lowerOnly) {
        for(int i=0; i<bx; i++)
            for(int j=0; j< (lowerOnly? i : by);  j++)
                for(int r=i*B; r<(i+1)*B; r++) {
                    final int target = pivots.get(r);
                    if(r != target) getBlock(i,j).permute(r, target);
                }
        return this;
    }
    boolean verify(PLU2_C M) {
        double max_diff = 0.0;
        for (int i = 0; i < N; i++) {
            int iB = i%B;
            for (int j = 0; j < N; j++) {
                final int I=i/B, J=j/B, jB = j%B;
                double v = 0.0;
                int k;
                for (k=0; k<i && k <= j; k++) {
                    final int K=k/B, kB=k%B;
                    v += M.getBlock(I,K).get(iB, kB) * M.getBlock(K,J).get(kB, jB);
                }
                if (k==i && k <= j) {
                    final int K=k/B, kB=k%B;
                    v += M.getBlock(K,J).get(kB, jB);
                }
                double diff = fabs(getBlock(I,J).get(iB, jB) - v);
                max_diff = max(diff, max_diff);
            }
        }
        return (max_diff <= 0.01);
    }
    public static void main(String[] a) {
        System.out.print("PLU2_C ");
        if (a.length < 4) {
            System.out.println("Usage: PLU2_C N b px py [verify] [libFile] ");
            return;
        }
        final int N = java.lang.Integer.parseInt(a[0]),
            B= java.lang.Integer.parseInt(a[1]),
            px= java.lang.Integer.parseInt(a[2]),
            py= java.lang.Integer.parseInt(a[3]);
        boolean VERIFY = true;
        String libraryName = "/vol/x10/vj/libPLUInC.so";
        if (a.length > 4) VERIFY=java.lang.Boolean.parseBoolean(a[5]);
        if (a.length > 5) libraryName = a[6];
        final int nx = N / (px*B), ny = N/(py*B);
        assert (N % (px*B) == 0 && N % (py*B) == 0);
        System.out.println("N="+N + " B=" + B + " px=" + px
                           + " py=" + py + " nx=" + nx + " ny="+ny);
        System.load(libraryName);
```

```
        final PLU2_C orig = new PLU2_C(px,py,nx,ny,B);
        int i=0;
        double[] flops = new double[NUM_TESTS];
        while (i < NUM_TESTS) {
            PLU2_C plu = new PLU2_C(orig);
            System.gc(); System.out.print("Starting...");
            long s = - System.nanoTime(); plu.run();  s += System.nanoTime();
            flops[i]=format(flops(N)/(s)*1000, 3);
            System.out.print(" Time="+(s)/1000000+"ms"+" Rate="+flops[i]+" MFLOPS");
            if (VERIFY) {
                System.out.print(" (Verifying...");
                long v = - System.nanoTime();
                boolean correct = new PLU2_C(orig).applyPivots(false).verify(plu.applyPivots(true));
                v += System.nanoTime();
                System.out.print(((v)/1000000) + " ms " + (correct?" ok)":" fail)"));
            }
            System.out.println();
            i++;
        }
        double max = 0.0; for (i=0; i < NUM_TESTS; i++) if (max < flops[i]) max=flops[i];
        System.out.println("Max rate: " + max);
    }
    public void run() {
        finish foreach (point [pi,pj] : [0:px-1,0:py-1]) {
            int startY=0, startX []= new int[ny];
            final Block[] myBlocks=A[pord(pi,pj)].z;
            while(startY < ny) {
                boolean done=false;
                for (int j=startY; j < min(startY+LOOK_AHEAD, ny) && !done; ++j) {
                    for (int i=startX[j]; i <nx; ++i) {
                        final Block b = myBlocks[lord(i,j)];
                        if (b.ready) {
                            if (i==startX[j]) startX[j]++;
                        } else done |= b.step(startY, startX);
                        Thread.yield();
                    }
                }
                if (startX[startY]==nx) { startY++;}
            }
        }
    }
    class Block(int I, int J) {
        final int maxCount;
        final double[:rail] A;
        @shared boolean ready = false;
        @shared private int count=0;
        @shared int LUCol=-1;
        @shared double maxColV; //maximum value in Column LU_col
        @shared int maxRow; //Row with that value
        int visitCount;
        private int rbIndex;
        private int cmCount=0, nextRowToBePermuted;
        Block(final int I, final int J) {
            property(I,J);
            A =  new double[[0:B*B-1]] (point [i]) {
                double d =Math.random();
                return (I==J && i %B==i)? format(20.0*d+10.0/RAND_MAX,4)
                : format(10.0*d/RAND_MAX, 4);};
            maxCount=min(I,J);
            rbIndex=I; visitCount=0; nextRowToBePermuted=I*B;
        }
        Block(final Block b) {
            property(b.I,b.J);
            A = new double[[0:B*B-1]] (point [i]) { return b.A[i];};
            maxCount = min(I,J);
            for(int i=0; i<B*B; i++) A[i] = b.A[i];
            rbIndex = I; visitCount=0; nextRowToBePermuted=I*B;
        }
```

```
    boolean step(final int startY, final int[] startX) {
        visitCount++;
        if (count==maxCount) {
            return I<J ? stepIltJ() : (I==J ? stepIeqJ() : stepIgtJ());
        } else {
            Block IBuddy=getBlock(I, count);
            if (!IBuddy.ready) return false;
            Block JBuddy=getBlock(count,J);
            if (!JBuddy.ready) return false;
            mulsub(IBuddy, JBuddy);
            count++;
            return true;
        }
    }
    nullable<Block> nextBlock=null;
    boolean stepIltJ() {
        final Block diag = getBlock(I,I);
        if(! diag.ready) return false;
        while (nextRowToBePermuted < (I+1)*B) {
            if (nextBlock != null) {
                if (nextBlock.count !=I) return false;
            } else {
                final int target = pivots.get(nextRowToBePermuted);
                if (target != nextRowToBePermuted) {
                    nextBlock = getBlock(target/B, J);
                    if (nextBlock.count != I) return false;
                }
            }
            permute(nextRowToBePermuted, pivots.get(nextRowToBePermuted));
            nextRowToBePermuted++; nextBlock=null;
        }
        backSolve(diag);
        return ready = true;
    }
    boolean stepIgtJ() {
        if(LUCol>=0) {
            Block diag = getBlock(J,J);
            if(!diag.ready && !(diag.LUCol > LUCol)) return false;
            lower(diag, LUCol);
            if(LUCol==B-1) ready=true;
        }
        ++LUCol;
        if(LUCol <= B-1) computeMax(LUCol);
        return true;
    }
    boolean stepIeqJ() {
        if(LUCol==-1) {
            for(;rbIndex<bx && (getBlock(rbIndex, J).count==J); rbIndex++);
            if(rbIndex < bx) return false;
        }
        if(LUCol>=0) {
            if(cmCount==0) cmCount = I+1;
            Block block;
            for(;cmCount<bx && ((block=getBlock(cmCount,J)).LUCol >=LUCol); cmCount++) {
                if(fabs(block.maxColV) > fabs(maxColV)) {
                    maxRow = block.maxRow;
                    maxColV = block.maxColV;
                }
            }
```

```
            if(cmCount < bx) return false;
            final int row = I*B+LUCol;
            pivots.set(row, maxRow);
            if(row != maxRow) permute(row, maxRow);
            LU(LUCol);
            if(LUCol==B-1)  ready=true;
        }
        ++LUCol;
        if(LUCol<=B-1)      {
            computeMax(LUCol, LUCol);
            cmCount=0;
        }
        return true;
    }
    void computeMax(int col) { computeMax(col, 0); }
    void computeMax(int col, int startRow) {
        int ord=ord(startRow,col);
        maxColV = A[ord];
        maxRow = I*B+startRow;
        for(int i=startRow+1; i<B; i++) {
            final double a = A[ord++];
            if(fabs(a) > fabs(maxColV)) {
                maxRow = I*B+i; // write Row first to use write-ordering property of volatiles
                maxColV = a;
            }
        }
        assert in(maxRow, 0, N);
    }
    void permute(int row1, int row2) {
        final Block b = getBlock(row2/B, J); //the other block
        int ord1=row1%B, ord2=row2%B;
        for(int j=0; j<B; j++){
            final double v1 = A[ord1], v2 = b.A[ord2];
            A[ord1]=v2; b.A[ord2]=v1;
            ord1+=B; ord2+=B;
        }
    }
    void lower( Block diag, int col) {blockLower(this.A, diag.A, col, B, diag.get(col,col)); }
    void backSolve(Block diag) { blockBackSolve(this.A, diag.A, B);}
    void mulsub(Block left, Block upper) { blockMulSub(this.A, left.A, upper.A, B);}
    void LU(final int col) {
        for (int i = 0; i < B; ++i) {
            int iord=ord(i,0), jord=ord(0,col), m = min(i,col);
            double r = 0.0; for(int k=0; k<m; ++k) {
                r += A[iord]*A[jord+k];
                iord +=B;
            }
            A[jord+i] -=r;
            if(i>col) A[jord+i] /= A[jord+col];
        }
    }
    int ord(int i, int j) { return i+j*B; }
    double get(int i, int j) { return A[ord(i,j)];  }
    void set(int i, int j, double v) {      A[ord(i,j)] = v; }
    void negAdd(int i, int j, double v) { A[ord(i,j)] -= v; }
    void posAdd(int i, int j, double v) { A[ord(i,j)] += v; }
  }
  static extern void blockLower(double[.] me, double[.] diag, int col, int B, double diagColCol);
  static extern void blockBackSolve(double[.] me, double[.] diag, int B);
  static extern void blockMulSub(double[.] me, double[.] left, double[.] upper, int B);
}
```