

xIUPC/BlueGene Class II Submission to the HPC Challenge Award Competition

October 27, 2006

Călin Caşcaval[†] George Almási[†] Christopher (Kit) Barton[‡] Ettore Tiotto[‡]
Gábor Dózsa[†] Montse Ferreras[§] Philip Luk[‡] Tom Spelce^{*}

[†] IBM TJ Watson Research Center
1101 Kitchawan Rd
Yorktown Heights NY 10598

[‡] IBM Software Group
8200 Warden Avenue
Markham Ontario L6G1C7

[§] Universitat Politècnica de Catalunya
Department of Computer Architecture
08034 Barcelona Spain

^{*}Lawrence Livermore National Laboratory
7000 East Ave.
Livermore, CA 94551

1 Submission Description

We implemented all of the Class 2 HPC Challenge benchmarks: Global HPL, Global FFT, Global Random Access and EP STREAM (Triad), in the UPC language [5]. The code for this implementation is derived directly from the HPC Challenge Official Specification Document:

<http://www.hpcchallenge.org/class2spec.pdf>.

The code sizes for our implementations, for HPL, FFT, Stream and RandomAccess respectively are as follows:

Lines	Blank	Cmnts	NCSL	TPtoks	
48	9	11	30	351	backsolve.upc (UPC)
89	15	26	48	402	main.upc (UPC)
52	6	12	35	296	matgen.upc (UPC)
43	4	25	24	165	panel.upc (UPC)
50	7	13	30	265	pivot.upc (UPC)
45	6	16	23	223	swap.upc (UPC)
45	7	24	15	159	triangular_solve.upc (UPC)
101	14	49	55	617	update.upc (UPC)
63	14	22	28	261	hpl.h (UPC)
536	82	198	288	2739	Total (UPC) ----- (9 files)

Lines	Blank	Cmnts	NCSL	TPtoks	Lines	Blank	Cmnts	NCSL	TPtoks
151	18	43	100	1018	fft.upc (UPC)				
121	24	23	75	637	verify.upc (UPC)				
59	14	23	22	160	fft.h (UPC)				
331	56	89	197	1815	-----	UPC	-----		(3 files)

Lines	Blank	Cmnts	NCSL	TPtoks	
157	31	40	90	654	EPStreamTriad.upc (UPC)

Lines	Blank	Cmnts	NCSL	TPtoks	
162	27	29	107	725	RandomAccess.upc (UPC)

All our code was compiled with an experimental version of the IBM XL UPC compiler targeted to the BlueGene computer family. In our submission we focus mostly on elegance and readability of the code, relying on the compiler, runtime system and machine characteristics for performance.

In order to keep the code small and readable we chose trivial implementations of all the benchmarks, following the description in the specification document. For parallelization, we used a subset of the UPC language features: shared arrays, parallel loops (`upc_forall`), barriers and blocked transfers. For Global HPL, we used a new feature, multidimensional blocking, which we have also proposed as a language extension to the UPC community. Multidimensional blocking allowed us to specify a tiled layout for the matrix, resulting in good locality. For more details, see Section 3.1.

2 Environment

We used several BlueGene installations for our experiments. Most of the development was done on stand-alone BlueGene node cards (32 nodes, i.e. 64 processors each) and a stand-alone rack (1024 nodes, 2048 processors)

located at IBM TJ Watson. Production runs were done on the 20 rack BlueGene located at the IBM TJ Watson facility in New York and on the 64 rack BlueGene/L at Lawrence Livermore National Labs.

We will now discuss the implementation of each benchmark.

3 Benchmark Discussion

3.1 Global HPL

The main kernel of HPL solves a dense linear system of equations using LU factorization with row partial pivoting. The basis of our implementation is single-processor blocked LU factorization code similar to that found in the LAPACK routines `dgetf2` and `dgetrf`. UPC allows the specification of a blocking factor for shared arrays which distributes the array across the threads using a block-cyclic layout; but for LU factorization this default layout is not optimal. We extended the UPC language specification with a 2-dimensional tiled layout similar to [2]. This allows blocked linear algebra routines to be executed in the tiles. We have ongoing discussions with the UPC community to add tiled layouts to the language. Tiled layouts do not affect the semantics of a UPC program, merely enhancing locality in a useful way. An example of a tiled shared array declaration is:

```
shared [B1][B2] double A[M][N];
```

The statement above declares an $M \times N$ array in which $B1 \times B2$ chunks are local to particular threads. For HPL we used square blocks.

The parallelism in the code occurs in several routines:

- in the panel factorization pivots are computed in parallel, each thread computing the maximum on its local elements. A reduction operation is used to compute the global pivot for a column;
- row scaling (“`dscal`”) is executed in parallel. The pivot is read by each thread and we chose to call the ESSL library to do the scaling for the local section of the column;
- row swapping is performed in parallel by all threads that owns sections of the row;
- outer products are executed in parallel by the threads owning blocks that is being updated. We call the ESSL `dger` routine for the local sections. The “`x`” and “`y`” vectors are transferred with `upc_memget` statements.
- triangular solve is executed in parallel by threads that own blocks of the RHS. We call ESSL `dtrsm` after copying the triangular block using a `upc_memget` statement.
- finally, rank-k update is executed in parallel by all threads owning a portion of the update matrix. Remote panels are copied in by `upc_memget`, after which we call ESSL `dgemm`.

In the second part of the benchmark, we backsolve the system of equations by using the ESSL `dgemv` and `dtrsm` routines. Each thread solves its local block. The backsolve uses the `upc_all_broadcast` routine to send locally computed vector elements.

The performance in this benchmark is limited by communication costs, local computation and load balance. Only one thread at a time can factorize a panel. Tile size also affects performance: larger panels lead to better performance because they increase the amount of local computation; however, they potentially cause load imbalance, especially towards the end of the computation when the number of panels is smaller. Based on early experiments chose a blocking factor of 300.

Verifying G-HPL: We initialize the matrix using random numbers. We set the last column in each row to the sum of all other elements. Thus, after triangularization and backsolve all the elements in the last column must be equal to 1.0. We compute the average error norm and check that it is less than epsilon.

Number of Racks	Number of threads	Memory Size (GBytes)	Array Size
1	2048	256	11,453,246,122
2	4096	512	22,906,492,245
4	8192	1024	45,812,984,490
8	16384	2048	91,625,968,981
16	32768	4096	183,251,937,962
32	65536	8192	366,503,875,925
64	131072	16384	733,007,751,850

Table 1: Required memory and array sizes for the EP STREAM benchmark.

3.2 Global FFT

The UPC implementation of Global FFT is conventional. The FFT source array (of length $N = 2^{2 \times M}$, where M is determined by the memory capacity of the machine) is processed as an $N \times N$ matrix. We perform a 2-D FFT involving a sequence of 1-D FFTs and distributed matrix transpositions.

In our implementation the array is distributed in a blocked fashion with a block size of $N \times B$. 1-D FFTs are performed on the rows of $N \times N$ matrices. Because of the data distribution 1-D FFTs are performed on contiguous data, and hence have good cache behavior. We invoke FFTW [4] to execute them.

Matrix transpositions, however, are nonlocal. Moreover, because we are exchanging $B \times B$ chunks of the matrix each row of the chunk is exchanged separately, resulting in relatively fine-grained communication (buffers of B elements exchanged by all threads). UPC does not provide primitives for strided communication. The algorithm requires three transpositions, two for correctness and a third because the HPC challenge problem description requires the results to be in the same order as the input data. This makes communication a dominant factor in performance.

The largest single chunk of code in the Global FFT implementation is the transpose (20 lines of code). It uses a sequence of `upc_memget` calls to implement the transposition. Unfortunately UPC does not supply the right kind of collective primitive to efficiently implement a transposition.

The memory requirements of FFT are such that the IN and OUT arrays together fill one fourth of the total available system memory. In our experiments we use 128MB per processor, which results in exactly one fourth of memory in co-processor mode.

Verification: is performed with a naive, distributed version of the Cooley-Tukey algorithm. The final result has to match the original array.

3.3 Random Access

The Random Access source code is the same as the 2005 HPC Challenge Class II submission.

We used the simplest possible algorithm for RandomAccess, in order to keep source code simple. We significantly improved our communication library and compiler optimizations, resulting in almost doubling the performance on large systems.

Verification: is done by executing the Random update twice, such that the resulting array is identical to the original.

3.4 EP Stream Triad

The EP Stream code is also identical to last year’s submission.

The problem sizes are selected to fill half the memory of the machine, as shown in Table 1.

Verifying STREAM: doing the verification on a single processor for an array of more than 366 billion elements is expensive and would consume all our machine allocation quota. Therefore we chose to do verification by sampling. Each thread randomly selects a set of indices (the set size being the number of threads running the program) and verifies that the array element at that location has the correct value. Note that as opposed to the embarrassingly parallel triad

Processors	Performance	Memory/CPU
	(Gflop/s)	(MBytes)
64	47.17	230
256	117.87	215

Table 2: Global HPL performance results.

Processors	Performance	Array Size
	(Gflop/s)	
4096	158.743	17,179,869,184
65536	1115.64	274,877,906,944

Table 3: Global FFT performance results.

operation, in which each node operates on local data exclusively, the verification step involves communication across the machine.

4 Performance Results

Tables 2, 3, 4 and 5 shows the measured performance of our benchmarks on up to 64 racks of BlueGene/L.

HPL is clearly affected by communication. Rank-k update is slowed down by the fact that the update of N^2 panels requires data from $2 \times N$ processors. In our implementation these $2 \times N$ processors have to serve N `upc_memget` requests each in every iteration and therefore become communication hot-spots, compromising performance. This problem is typically solved by broadcasts on rows and columns of processors; however, UPC does not offer such collective primitives.

FFT performance is dominated by serial FFTW at low thread counts, and by the transpose operation on larger machines. On BlueGene the transpose operation’s bandwidth is also limited by the cross-section bandwidth of the machine. UPC does not have an `all-to-all` type operation that would be useful for matrix transposition, so we used sets of `upc_memget` operations.

For RandomAccess (Table 4) we optimized two regions of optimization. At a relatively small numbers of threads (up to 4 racks) performance is driven by the compiler’s ability to optimize the code. Important optimizations we had to implement included the ability to recognize and propagate constants (such as the number of threads) and to eliminate integer divisions in the main loop. Since UPC treats each shared variable access as potentially remote, a large number of index calculations need to be performed, which in turn lead to modulo operations - an operation that the 440 cores of the BlueGene processor were not designed to perform efficiently.

Going from 1 processor to 2 processors in RandomAccess there is a 50% drop in efficiency, due to the non-locality of the updates on multiple processors. From there on efficiency drops steadily, with larger drops on machine topologies that are farther away from cubic. E.g. note the drop in performance between 2 and 4 racks.

STREAM (Table 5) is embarrassingly parallel, and there is no scaling drop. In the table we left out the intermediate results because they contribute no information.

Acknowledgments

This work was supported in part by DARPA Contract NBCH30390004. We are also grateful to a number of people who offered support and advise throughout the project. In particular, we would like to thank Roch Archambault, Jose Castanos, John Gunnels, Roland Koo, and Fred Mintzer for helping at different stages of this project and with preparing and running our programs on BG/L.

Processors	Performance	Memory Used
	(GUPS)	(TBytes)
2048	0.58	0.250000
4096	1.15	0.500000
8912	2.28	1.000000
16384	4.49	2.000000
32768	8.83	4.000000
65536	14.80	8.000000
131072	28.31	16.000000

Table 4: Random Access performance results.

Processors	Performance	Memory used
	(GB/s)	(TBytes)
2048	1432.70	0.256000
4096	2865.35	0.500000
8912	5730.41	1.000000
16384	11460.65	2.000000
32768	22920.70	4.000000
131072	91627.49	16.000000

Table 5: STREAM Triad performance results.

References

- [1] Christopher Barton, Calin Cascaval, George Almasi, Yili Zhang, Montse Farreras, Siddhartha Chatterjee, and José Nelson Amaral. Shared memory programming for large scale machines. In *Programming Language Design and Implementation (PLDI)*, pages 108–117, June 2006.
- [2] Ganesh Bikshandi, Jia Guo, Dan Hoefflinger, George Almasi, Basilio B. Fraguera, Maria-Jesus Garzaran, David Padua, and Christoph von Praun. Programming for Parallelism and Locality with Hierarchical Tiled Arrays. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 48–58, March 2006.
- [3] BlueGene *The IBM Journal of Research and Development*, 49(2/3), 2005
- [4] Frigo, Matteo and Johnson, Steven G. The Design and Implementation of FFTW3 Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation. 2005, volume 93 No 2 pp. 216-231
- [5] UPC Consortium. *UPC Language Specification, V1.2*, May 2005.

Appendix A: Global HPL

```
/**
 * HPC Challenge Class 2 -- Global HPL
 *
 * IBM Research
 * (C) Copyright 2006, All Rights Reserved
 *
 */
#include "hpl.h"

shared int ipvt[M]; // shared array to store pivots

/**
 * \brief panel factorization of A[col0:M][col0:col1]
 * updates of A[col0:M][col0:col2]
 * generates ipvt[0:col1]
 */
void parallel_panel (int col0, int col1, int col2)
{
    for (int k=col0; k< col1; k+= B)
    {
        int k1 = min (k+B, col1);
        panel (k, k1, k1); // panel factorization on one block
        swap_rows (k, k1-k, k1, col2); // row swap to the right of panel
        triangular_solve(k, k1, col2); // solve to the right of panel
        update(k, k1, col2); // parallel update
    }
}

/**
 * \brief one block panel factorization
 */
void panel (int col0, int col1, int col2)
{
    for(int k= col0; k < col1; k++)
    {
        int pivotRow = max_pivot(k); // compute the pivot row
        ipvt[k-col0] = pivotRow; // update perm. vector
        scale_column(k, pivotRow); // scale column w/ pivot
        swap_row(k, pivotRow, col0, col2); // then swap the rows
        outer_product(k, col2); // update rest of matrix
    }
}

void update(int col0, int col1, int col2)
{
    int p = min(B, col1-col0);
    int blk = B; // lda to pass by ref to ESSL
    upc_forall (int ii = col0+B; ii<M; ii+=B; continue)
        upc_forall (int jj = col0+B; jj<col2; jj+=B; &A[ii][jj])
        {

```

```

double a[B*B], b[B*B]; // local blocks
double alpha=-1.0, beta=1.0; // dgemm coefs
int m = min(M, ii+B) - ii; // size of m x n
int n = min(col2, jj+B) - jj;
upc_memget (a, &A[ii][col0], (B*B*sizeof(double))); // get a locally
upc_memget (b, &A[col0][jj], (B*B*sizeof(double))); // get b locally
double * c = (double *)&A[ii][jj]; // c is local

dgemm("N", "N", &n, &m, &p, &alpha, b, &blk, a, &blk, &beta, c, &blk);
}
upc_barrier;
}

shared [B*B] double x_k[B*B][THREADS];

void parallel_backsolve ()
{
int blk = B;

for(int kk=(M/B)*B;kk>=0;kk=kk-B)
{
int m = min(M-kk,B); // lines in block
if (MYTHREAD == upc_threadof (&A[kk][N-1]))
{
int one = 1;
double alpha = 1.0;
double * y = (double *)&A[kk][N-1];
double a_kk [B*B]; upc_memget(a_kk, &A[kk][kk], B*B*sizeof(double));

dtrsm ("R", "L", "N", "N", &n, &m, &alpha, a_kk, &blk, y, &blk);
}

upc_barrier; /* wait for A[kk:kk+m][N-1] to be ready */

/* broadcast A[kk:kk+m][N-1] everywhere */
upc_all_broadcast(x_k, &A[kk][N-1], B * B * sizeof(double), 0);

/* update A[ii:kk][N-1] vector */
upc_forall (int ii = 0; ii < kk; ii+=B; &A[ii][N-1])
{
double alpha = -1.0;
double beta = 1.0;
int n = min(M-ii,B);
double a_ik [B*B]; upc_memget (a_ik, &A[ii][kk], B*B*sizeof(double));
double * y = (double *) &A[ii][N-1];

dgemv("T", &m, &n, &alpha, a_ik, &blk,
(double *)&x_k[0][MYTHREAD], &blk, &beta, y, &blk);
}
}
}

```


Appendix B: Global FFT

```
/**
 * HPC Challenge Class 2 -- Global 1-D FFT
 *
 * IBM Research
 * (C) Copyright 2006, All Rights Reserved
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "fft.h"

/* ***** */
/*          global and shared variable declarations          */
/* X is the initial data; Z is the result of the transform.  */
/* The transform is size N*N, where N=2^M.                  */
/* ***** */

ComplexArray_t X, A, Z;

/* ***** */
/*          main program          */
/* ***** */

int main()
{
    double t_start, t_init, t_transp1, t_fftcol;
    double t_twiddle, t_transp2, t_fftrow, t_end;
    fftw_plan planZ, planA;

    if ( N % THREADS != 0 ) abort(); // consistency chk
    initRandData(X);                t_start = mysecond(); // init random data
    fftw_init (Z,A, &planZ, &planA); t_init   = mysecond(); // init fftw
    transpose (X,Z);                 t_transp1 = mysecond(); // transpose X
    fftw_execute (planZ);             t_fftcol  = mysecond(); // fftw of Z
    multByTwiddleFactors(Z);         t_twiddle = mysecond(); // adjust twiddle
    transpose (Z, A);                t_transp2 = mysecond(); // transpose Z
    fftw_execute (planA);            t_fftrow  = mysecond(); // fftw of A
    transpose (A, Z);                t_end    = mysecond(); // transpose A
    int rc = verify (X,Z,2*M);       // verify

    /* print results */
    if ( MYTHREAD == 0 )
    {
        printf ("VERIFY %s\n", rc?"FAIL":"SUCCESS");
        double t_sec = t_end - t_start;
        double gflops = (double)( 5.0 * (N*N) * 2.0 * M * 0.000000001 ) / t_sec;
        printf("FFT size:%lld elements, Total time: %g seconds GFlops = %g\n",
            (long long) N*N, t_sec, gflops);
        printf("init:      %gs (%3.1f%)\n", t_init-t_start, PRCNT(t_init-t_start));
        printf("transp1: %gs (%3.1f%)\n", t_transp1-t_init, PRCNT(t_transp1-t_init));
    }
}
```

```

    printf("fftcoll: %gs (%3.1f%)\n", t_fftcoll-t_transp1, PRCNT(t_transp1-t_fftcoll));
    printf("twiddle: %gs (%3.1f%)\n", t_twiddle-t_fftcoll, PRCNT(t_twiddle-t_fftcoll));
    printf("transp2: %gs (%3.1f%)\n", t_transp2-t_twiddle, PRCNT(t_transp2-t_twiddle));
    printf("ffttrow: %gs (%3.1f%)\n", t_ffttrow-t_transp2, PRCNT(t_ffttrow-t_transp2));
    printf("transp3: %gs (%3.1f%)\n", t_end-t_ffttrow, PRCNT(t_end-t_ffttrow));
}
upc_barrier;
return rc;
}

/* ***** */
/* matrix transpose of A->B, arrays of N*N interpreted as (N, N) matrices */
/* ***** */
void transpose (ComplexArray_t A, ComplexArray_t B)
{
    unsigned bsize = N / THREADS;
    ArrayIndex_t i, j;
    upc_barrier;
    upc_forall (i = 0; i < N; i += bsize; &B[ i*N ] )
        for (j = 0; j < N; j += bsize )
            {
                complex_t * la = (complex_t *)&A[i*N+j];
                complex_t * lb = (complex_t *)&B[i*N+j];

                // copy block to dest row by row
                for (unsigned k = 0; k < bsize; k++)
                    upc_memget( lb + k * N, &A[(j+k) * N + i], sizeof(complex_t) * bsize );

                // transpose block in place
                for (unsigned k = 0; k < bsize - 1; k++)
                    for (unsigned l = k + 1; l < bsize; l++)
                        {
                            complex_t c = lb[k*N+l];
                            lb[k*N+l] = lb[l*N+k];
                            lb[l*N+k] = c;
                        }
            }
    upc_barrier;
}

/* ***** */
/* multiplication with twiddle factors between transposes */
/* ***** */

void multByTwiddleFactors(ComplexArray_t Z)
{
    for (ArrayIndex_t i = 0; i < N; i++)
        upc_forall (ArrayIndex_t j = 0; j < N; j++; &Z[ i*N+j ] )
            {
                double x = ( 2 * M_PI * i * j ) / ( N * N );
                double tw_re = cos(x), tw_im = -sin(x);
            }
}

```

```

        Z[ i*N+j ].re = tw_re * Z[ i*N+j ].re - tw_im * Z[ i*N+j ].im;
        Z[ i*N+j ].im = tw_im * Z[ i*N+j ].re + tw_re * Z[ i*N+j ].im;
    }
}

/* ***** */
/* setup initial data for performance measurement (random) */
/* ***** */

void initRandData (ComplexArray_t X)
{
    for (ArrayIndex_t i = 0; i < N; i++ )
        upc_forall (ArrayIndex_t j = 0; j < N; j++ ; &X[ i*N + j ] )
        {
            X[i*N+j].re = (double)random();
            X[i*N+j].im = (double)random();
        }
}

/* ***** */
/* Initialize fftw - create plans for local 1D fft of Z and A's rows */
/* ***** */

void fftw_init (ComplexArray_t Z, ComplexArray_t A,
fftw_plan *planZ, fftw_plan *planA)
{
    int n = N;
    *planZ =
        fftw_plan_many_dft(1, &n, N/THREADS,
            (fftw_complex *)&Z[MYTHREAD*(N/THREADS*N)], NULL, 1, N,
            (fftw_complex *)&Z[MYTHREAD*(N/THREADS*N)], NULL, 1, N,
            FFTW_FORWARD, FFTW_ESTIMATE );
    *planA =
        fftw_plan_many_dft(1, &n, N/THREADS,
            (fftw_complex *)&A[MYTHREAD*(N/THREADS*N)], NULL, 1, N,
            (fftw_complex *)&A[MYTHREAD*(N/THREADS*N)], NULL, 1, N,
            FFTW_FORWARD, FFTW_ESTIMATE );
}

```

Appendix C: Global Random Access

```
/**
 * HPC Challenge Class 2 -- Global Random Access
 *
 * IBM Research
 * (C) Copyright 2005,2006, All Rights Reserved
 *
 */
#include <math.h>
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>
#include <time.h>

typedef unsigned long long uint64;
typedef long long int64;
typedef unsigned long long ArrayIndex_t;

#ifndef N
#define N          (21)
#endif

#define TableSize (1ULL<<N)
#define NUUPDATE  (4ULL * TableSize)

#define POLY       0x0000000000000007ULL
#define PERIOD     1317624576693539401LL

shared uint64 Table[TableSize];
shared unsigned verifyfailed;

double      mysecond      ();
uint64      starts        (int64);
void        RandomAccessUpdate ();
void        RandomAccessVerify ();

/* ***** */
/*          main program          */
/* ***** */
int main()
{
    double time = 0;
    double GUPs = 0;
    verifyfailed = 0;

    if(MYTHREAD == 0)
        printf("Table size = %llu MBytes/CPU, %llu MB/total on %d threads\n",
            (uint64)TableSize*8/1024/1024/THREADS,
            (uint64)TableSize*8/1024/1024,
            THREADS);
}
```

```

/* ----- */
/* begin timed portion */
/* ----- */
upc_barrier;
time = mysecond();
upc_forall (ArrayIndex_t i = 0; i < TableSize; i++; i) Table[i] = i;
upc_barrier;
RandomAccessUpdate();
upc_barrier;
time = mysecond() - time;
GUPs = ((double) 1e-9 * NUPDATE) / time;

/* ----- */
/* end timed portion */
/* ----- */
if (MYTHREAD == 0) printf ("End timed portion: verifying\n");
RandomAccessUpdate(); /* do it again */
upc_barrier;
RandomAccessVerify();
upc_barrier;

if(MYTHREAD == 0)
{
    if ((double)verifyfailed/(double)NUPDATE < 0.01)
printf ("Verification: SUCCESS (%d/%lld)\n", verifyfailed, NUPDATE);
    else
printf ("Verification *FAILED*, %d\n", verifyfailed);
    printf("Default number of updates (RECOMMENDED) = %llu\n", NUPDATE);
    printf("Real time used = %.6f seconds\n", time );
    printf("%.9f Billion(10^9) Updates per second [GUP/s]\n", GUPs);
}

return 0;
}

/* ***** */
/* principal loop (that does all the work) */
/* ***** */

void RandomAccessUpdate()
{
    ArrayIndex_t i;
    uint64 ran = starts(NUPDATE/THREADS * MYTHREAD);
    upc_forall (i = 0; i < NUPDATE; i++; i)
    {
        ran = (ran << 1) ^ (((int64) ran < 0) ? POLY : 0);
        Table[ran & (TableSize-1)] = Table[ran & (TableSize-1)] ^ ran;
    }
}

/* ***** */

```

```

/*          check loop          */
/* ***** */

void RandomAccessVerify()
{
    ArrayIndex_t i;
    int localverifyfailed = 0;
    upc_forall (i = 0; i < TableSize; i++; i)
        if (Table[i] != i) localverifyfailed++;
    if(localverifyfailed) verifyfailed += localverifyfailed;
}

/* ***** */
/* Utility routine to start random number generator at Nth step */
/* ***** */

uint64 starts(int64 n)
{
    int i;
    uint64 m2[64];
    uint64 temp, ran;

    while (n < 0)          n += PERIOD;
    while (n > PERIOD)    n -= PERIOD;

    if (n == 0)          return 0x1;

    temp = 0x1;
    for (i=0; i<64; i++)
    {
        m2[i] = temp;
        temp = (temp << 1) ^ ((int64) temp < 0 ? POLY : 0);
        temp = (temp << 1) ^ ((int64) temp < 0 ? POLY : 0);
    }

    for (i=62; i>=0; i--) if ((n >> i) & 1) break;

    ran = 0x2;
    while (i > 0)
    {
        temp = 0;
        for (int j=0; j<64; j++) if ((ran >> j) & 1) temp ^= m2[j];
        ran = temp;
        i -= 1;
        if ((n >> i) & 1) ran = (ran << 1) ^ ((int64) ran < 0 ? POLY : 0);
    }

    return ran;
}

/* ***** */

```

```
/* Utility routine to measure time */
/* ***** */

double mysecond()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + ((double) tv.tv_usec / 1000000);
}
```

Appendix D: EP Stream Triad

```
/**
 * HPC Challenge Class 2 -- Global EP Stream Triad
 *
 * IBM Research
 * (C) Copyright 2005, All Rights Reserved
 *
 */
#include <stdio.h>
#include <time.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <float.h>
#include <limits.h>

typedef unsigned long long uint64;
typedef unsigned long long ArrayIndex_t;

/*
 * The vector size. The benchmark will allocate 3 * N * sizeof(double)
 * Set in the Makefile for different size machines.
 */
#ifndef N
#define N 2000000
#endif
#define NTIMES 10

shared double a[N], b[N], c[N];
shared int errorCode;

#define Mmin( a_, b_ ) ( ( (a_) < (b_) ) ? (a_) : (b_) )
#define Mmax( a_, b_ ) ( ( (a_) > (b_) ) ? (a_) : (b_) )

double mysecond();

/* ***** */
/* main program */
/* ***** */
int main()
{
    double alpha = 3.0;
    double times[NTIMES];
    double minTime = 100000000.0, maxTime = 0.0, avgTime = 0.0;
    double rateGBs = 0.0;
    const ArrayIndex_t VectorSize = N;
    ArrayIndex_t i;

    errorCode = 0;
}
```

```

    if (MYTHREAD==0)
printf ("Memory in use: %lld MB per CPU, %lld MB total, %d threads\n",
        (uint64)(N*24)/1024/1024/THREADS,
        (uint64)(N*24)/1024/1024,
THREADS);

// -----
// initialize the b and c arrays with a verifiable sequence
// -----
upc_forall(i = 0; i < VectorSize; i++) {
    b[i] = 1.5*i;
    c[i] = 2.5*i;
}

upc_barrier;

// -----
// timing loop, we select the best of NTIMES, excluding the first
// -----
for(int k = 0; k < NTIMES; k++) {
    times[k] = mysecond();

    upc_forall(i = 0; i < VectorSize; i++) {
        a[i] = b[i] + alpha * c[i];
    }

    upc_barrier;

    times[k] = mysecond() - times[k];
}

// -----
// verify
// -----
if(checkTriad()) errorCode++;

upc_barrier;

if(MYTHREAD == 0) {
    if(errorCode != 0) printf("verification failed %d\n", errorCode);
    else printf("verification successful\n");
}

upc_barrier;

// -----
// select the best iteration and compute the rate
// -----
for(int k = 1; k < NTIMES; k++) {
    avgTime = avgTime + times[k];
    minTime = Mmin(minTime, times[k]);
}

```

```

        maxTime = Mmax(maxTime, times[k]);
    }

    avgTime /= (double)(NTIMES - 1); /* note -- skip first iteration */
    rateGBs = (minTime > 0.0 ? 1.0 / minTime : -1.0);
    rateGBs *= 1e-9 * 3 * sizeof(double) * N;

    if(MYTHREAD == 0) {
        printf("Rate (GB/s)   Avg time       Min time       Max time\n");
        printf("%11.4f  %11.4f  %11.4f  %11.4f\n",
            rateGBs, avgTime, minTime, maxTime);
    }

    return 0;
}

/* ***** */
/* Verification function */
/* ***** */
int checkTriad()
{
    double val;
    ArrayIndex_t index;
    const ArrayIndex_t VectorSize = N;

    // -----
    // each thread verifies THREAD elements randomly spread throughout the
    // machine
    // -----
    srand(MYTHREAD+1); // initialize the random generator

    for(int k = 0; k < THREADS; k++) {

        index = random() % VectorSize; // compute the index
        val = (1.5 * index) + 3.0 * (2.5 * index); // compute the expected val

        if ( a[index] != val ) {
            printf("%d: verification failed a[%lld]=%11.4f, val=%11.4f\n",
                MYTHREAD, (uint64)index, a[index], val);
            return 1;
        }
    }
}

return 0;
}

/* ***** */
/* Utility function to get the time */
/* ***** */
double mysecond()

```

```
{
  struct timeval tv;
  gettimeofday(&tv, 0);
  return tv.tv_sec + ((double) tv.tv_usec / 1000000);
}
```