

The UPC/BlueGene Class II Submission to the HPC Challenge Award Competition

October 15, 2005

Călin Cașcaval
IBM TJ Watson Research Center
1101 Kitchawan Rd
Yorktown Heights NY 10598
cascaval@us.ibm.com

Christopher (Kit) Barton
IBM Software Group, Toronto Lab
8200 Warden Avenue
Markham Ontario L6G1C7
kbarnton@ca.ibm.com

George Almási
IBM TJ Watson Research Center
1101 Kitchawan Rd
Yorktown Heights NY 10598
gheorghe@us.ibm.com

Yili O Zheng
Dept. Electrical and Computer Eng
Purdue University
West Lafayette IN 47907
yzheng@purdue.edu

Montse Farreras
Universitat Politecnica de Catalunya
Department of Computer Architecture
08034 Barcelona Spain
mfarrera@ac.upc.es

Philip Luk
IBM Software Group, Toronto Lab
8200 Warden Avenue
Markham Ontario L6G1C7
philipl@ca.ibm.com

Raymond Mak
IBM Software Group, Toronto Lab
8200 Warden Avenue
Markham Ontario L6G1C7
rmak@ca.ibm.com

1 Submission Description

We implemented two of the Class 2 HPC Challenge benchmarks, Random Access and EP Stream Triad in the UPC language. The code for this implementation is derived directly from the HPC Challenge Official Specification Document <http://www.hpcchallenge.org/class2spec.pdf>.

The code size for these two implementations is:

Lines	Blank	Cmnts	NCSL	TPtoks	
178	34	43	105	767	EPStreamTriad.upc (Java)
160	23	27	111	722	RandomAccess.upc (Java)

The code was compiled with an experimental version of the IBM XL UPC compiler. In our submission we focus mostly on elegance, and did not concentrate too much on optimizing performance (except for the benchmark verification, as noted in each benchmark description). The performance results are almost exclusively based on compiler and runtime optimizations.

From a productivity perspective, even for a user who is not familiar with the UPC language, the implementation is almost trivial. The benchmarks pretty much follow the description in the specification document. For parallelization, only three features of the UPC language were used: shared arrays, parallel loops (`upc_forall`) and barriers. It took us about half an hour to code each benchmark.

2 Environment

We used three different BlueGene installations for our runs. 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) called BG/X, located at IBM TJ Watson. We also used this machine for the single processor runs.

Production runs were done on the BG/W machine, a 20 rack machine located at TJ Watson. Production runs for 32 racks were done at Lawrence Livermore National Labs, using the 64 rack BlueGene/L machine.

Each node has 512 MB of local memory and 4 MB of L3 cache. This gives, a memory footprint of 512 GB for one rack, and 32 TB for 64 racks [1].

We will now discuss the implementation of each benchmark.

3 Benchmark Discussion

3.1 Random Access

We used the simplest possible algorithm for RandomAccess, both to keep source code simple and to guarantee zero errors.

The main loop in RandomAccess resolves to a number of read-modify-write (RMW) operations to remote locations across the machine. Since BlueGene is a distributed memory machine, each RMW operation translates to a network packet; hence, in the current form of the UPC RandomAccess code, performance is bounded by network packet latency. It performs somewhat better than the naive MPI RandomAccess implementation, simply because the UPC runtime has less overhead than MPI; also, we are using a newer generation of the BlueGene messaging library which delivers better performance.

Because each BlueGene process is running on a single thread, there are no write hazards in the system. Hence, zero errors. Verification is done by running the update sequence twice and checking that we get back the original values in the array. Just to make sure that the update sequence is not compiled into a "no-op" we commented out one of the updates and as expected the benchmark failed verification.

The main table in RandomAccess has to be at least half the memory size. A node of BlueGene has 512 MBytes of memory; hence a BlueGene rack (1024 nodes) has 512 GB.

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

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

An N value of 35 will lead to a table size of 256 GBytes, or 1/2 of a rack. For 2, 4, 8, 16, 32 and 64 racks, N has to have the value of 36, 37, 38, 39, 40 and 41 respectively.

We use a loop iteration count of 2^{N+2} (as suggested). This yields a runtime of about 300 seconds (complete with verification) for the array sizes as described above. The benchmark doesn't quite scale as the machine size grows, so total runtime also increases slightly on larger runs.

3.2 EP Stream Triad

In the EP version of the STREAM triad, all the computation is done locally. We obtained this effect in UPC by using the affinity clause of the `upc_forall` loop.

The memory requirements of STREAM are dictated by the 3 main arrays: the size of these arrays has to be more than a quarter of the main memory and not fit in the cache. A node of BlueGene has 512MBytes of memory and 4MB of L3 cache, shared by the two cores. A BlueGene rack (1024 nodes) has 512GB of memory and 4GB of L3 cache.

We chose to be conservative and selected the arrays to fill half the memory of the machine, and the sizes are presented 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 operation, in which each node operates on local data exclusively, the verification step involves communication across the machine.

4 Performance Results

Discussion

Tables 2 and 3 show the absolute and scaling performance of our benchmarks as measured on up to 32 racks of BlueGene. During production runs we discovered that the UPC runtime does not handle more than 65,536 processors; hence no 64 rack runs in the submission.

For RandomAccess (Table 2) 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 3) is embarrassingly parallel, and there is no scaling drop. In the table we left out the intermediate results because they contribute no information.

Machine	Processors	Performance (GUPS)	Memory TBytes		efficiency (%)
			used	total	
BG/X (1 CPU)	1	5.4E-4	0.000128	0.000512	100
BG/X (2 CPU)	2	7.8E-4	0.000256	0.000512	72
BG/X (4 CPU)	4	1.3E-3	0.000512	0.001	61
BG/X (1 node card)	64	0.02	0.008192	0.016	61
BG/W (1 rack)	2048	0.56	0.250000	0.500	51
BG/W (2 racks)	4096	1.11	0.500000	1.000	50
BG/W (4 racks)	8912	1.70	1.000000	2.000	38
BG/W (8 racks)	16384	3.36	2.000000	4.000	38
BG/W (16 racks)	32768	6.10	4.000000	8.000	34
BG/L (32 racks)	65536	11.54	8.000000	16.000	33

Table 2: Random Access performance results.

Machine	Processors	Performance (GB/s)	Memory TBytes	efficiency (%)
			used	
BG/X (1 CPU)	1	0.73	0.000128	100
BG/X (2 CPU)	2	1.46	0.000256	100
BG/X (4 CPU)	4	2.92	0.000512	100
BG/X (1 node card)	64	46.72	0.008192	100
BG/L (32 racks)	65536	47827.00	8.000000	100

Table 3: STREAM Triad performance results.

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 this challenging project. In particular, we would like to thank Roch Archambault, Anthony Bolmarcich, Jose Castanos, Sid Chatterjee, John Gunnels, Manish Gupta, Roland Koo, Larry Lindsay and Fred Mintzer and Tom Spelce (LLNL) for helping at different stages of this project and with preparing and running our programs on BG/L.

References

- [1] BlueGene *The IBM Journal of Research and Development*, 49(2/3), 2005

Appendix A: Random Access

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

typedef unsigned long long u64Int;
typedef long long s64Int;

#ifndef N
#define N (21)
#endif
#define TableSize (1ULL<<N)
#define NUPLATE (4ULL * TableSize)
#define POLY 0x00000000000000007ULL
#define PERIOD 1317624576693539401LL

shared u64Int *Table;
shared unsigned verifyfailed;

double mysecond();
u64Int starts (s64Int);
void RandomAccessUpdate();
void RandomAccessVerify();

/* ***** */
/* main program */
/* ***** */
int main()
{
    u64Int i;
    Table = (shared u64Int *)upc_all_alloc(TableSize,sizeof(u64Int));
    assert(Table != 0);
    double time = 0;
    double GUPs = 0;
    verifyfailed = 0;

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

```

/* ----- */
/* begin timed portion */
/* ----- */
upc_barrier;
time = mysecond();
upc_forall (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 (verifyfailed == 0) printf ("Verification: SUCCESS\n");
    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);
}
upc_free (Table);
return 0;
}

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

void RandomAccessUpdate()
{
    u64Int i, ran = starts(NUPDATE/THREADS * MYTHREAD);
    upc_forall (i = 0; i < NUPDATE; i++; i)
    {
        ran = (ran << 1) ^ (((s64Int) ran < 0) ? POLY : 0);
        Table[ran & (TableSize-1)] ^= ran;
    }
}

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

```

```

void RandomAccessVerify()
{
    u64Int i;
    upc_forall (i = 0; i < TableSize; i++; i)
        if (Table[i] != i)
            { verifyfailed++; printf ("Failed at index=%lld: %lld\n", i, Table[i]); return; }
}

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

u64Int starts(s64Int n)
{
    /* s64Int i, j; */
    int i, j;
    u64Int m2[64];
    u64Int 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) ^ ((s64Int) temp < 0 ? POLY : 0);
        temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
    }

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

    ran = 0x2;
    while (i > 0)
    {
        temp = 0;
        for (j=0; j<64; j++)
            if ((ran >> j) & 1)
                temp ^= m2[j];
        ran = temp;
        i -= 1;
        if ((n >> i) & 1)
            ran = (ran << 1) ^ ((s64Int) ran < 0 ? POLY : 0);
    }

    return ran;
}

```

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

Appendix B: 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 u64Int;

/*
 * 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;
shared double *b;
shared double *c;
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 = FLT_MAX, maxTime = 0.0, avgTime = 0.0;
    double rateGBs = 0.0;
    const u64Int VectorSize = N;
    u64Int i;
    unsigned k;

    // -----
```

```

// allocate the arrays
// -----
a = (shared double *) upc_all_alloc(VectorSize,sizeof(double));
b = (shared double *) upc_all_alloc(VectorSize,sizeof(double));
c = (shared double *) upc_all_alloc(VectorSize,sizeof(double));
errorCode = 0;

upc_barrier;

if(a == NULL || b == NULL || c == NULL) {
    if (c != NULL) upc_free(c);
    if (b != NULL) upc_free(b);
    if (a != NULL) upc_free(a);
    fprintf(stderr, "Failed to allocate memory (%d).\n", VectorSize );
    return 1;
}

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

upc_free(a);
upc_free(b);
upc_free(c);

return 0;
}

/* ***** */
/* Verification function */
/* ***** */
int checkTriad()
{
    unsigned k;
    double val;
    u64Int index;
    const u64Int VectorSize = N;

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

```

```

for(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, 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);
}

```