**RC25067 (W1010-100) October 25, 2010**
**Computer Science**

# IBM Research Report

## HPCC 2010 Class II Submission: UPC and CoArray Fortran

**George Almási[1], Barnaby Dalton[2], Lawrence L. Hu[2], Albert Sidelnik[3], Ilie Gabriel Tanase[1], Ettore Tiotto[2], Xing Xue[2]**

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

[2]IBM Software Group
Toronto, Canada

[3]University of Illinois at Urbana-Champaign

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

**LIMITED DISTRIBUTION NOTICE**: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

# 1 Submission Description

We implemented all four of the Class 2 HPC Challenge benchmarks Global HPL (HPL), Global FFT (FFT), Global Random Access (RA) and EP STREAM Triad (EP), and an extra benchmark (k-means clustering [8, 9]) in Unified Parallel C [7]. We also implemented FFT, RA, EP and k-means in CoArray Fortran (CAF) [10]. The code for this implementation is derived directly from the HPC Challenge Official Specification Document:

`http://www.hpcchallenge.org/class2spec.pdf`.

Our submission attempts to balance scalability and performance with the elegance of written code. By leveraging the benchmark specification, the benchmarks were expressed as cleanly as we could while sacrificing as little performance as possible. Our UPC and CAF implementations are discussed in terms of expressivity, productivity and performance trade offs.

Our submission differs from last year's in the following details:

- We implemented a new benchmark to compute k-means clustering. While the code is relatively simple, the memory access pattern stresses the compiler and the machine in a novel way that complements existing benchmarks.
- We have implemented four of the five benchmarks - FFT, RandomAccess, stream and k-means in CoArray Fortran . The CoArray Fortran frontend has been recently integrated to the IBM XL family of compilers. We also discuss implementation status and challenges for the HPL benchmark.
- We ran the benchmarks and evaluated their performance on the IBM Blue Gene/P, Power 5, and the novel Power 7 systems recently introduced by IBM.

This paper discusses the benchmarks, our implementation approaches and the performance we obtained. We include relevant segments of the code here; the complete source code and all result logs are contained in the full submission package.

The rest of this document is structured as follows. Section 2 describes the compilers we used in this submission. Section 3 describes the individual benchmarks, and how we implemented them. Having taken the committee's criticism from last year to heart, we will emphasize the productivity aspect of each benchmark implementation. Section 4 discusses the performance evaluation of the benchmarks. The hardware architectures used are presented in Section 4.1 while individual benchmarks are discussed starting with Section 4.2. We conclude in Section 5.

# 2 Our compilers

## 2.1 The UPC compiler

Unified Parallel C (UPC) [7] is an explicit parallel extension to the standard C programming language which allows the concurrent execution of a program by multiple threads. The UPC language conforms to the Single-Program Multiple-Data (SPMD) programming model with a Partitioned Global Address Space (PGAS) memory model.

The IBM XL Unified Parallel C compiler prototype is built on the well-known strength of the IBM XL compiler family. As a member of the XL compiler family, the UPC compiler prototype provides extensive diagnostic and compilation-time syntax checking of Unified Parallel C language constructs.

The XL UPC compiler prototype is a fully optimizing compiler, as such we have significantly extended the traditional loop optimizer infrastructure to incorporate new optimizations specifically designed to exploit the locality information available in UPC programs. A number of new UPC specific optimizations have been implemented: shared accesses privatization, shared-object access coalescing, `upc_forall` loop optimizations, remote shared-object updates, and other parallel loop optimizations.

An alpha version of the XL UPC compiler is currently available for download from [11].

## 2.2 The CoArray Fortran compiler

CoArray Fortran (CAF) is an extension to Fortran being introduced in the F2008 standard. It extends Fortran programs into the Single-Program Multiple-Data (SPMD) model with a Partitioned Global Address Space (PGAS), similar to what UPC does to C.

The IBM CoArray Fortran compiler is built on top of the existing XL Fortran compiler stack and integrated with the PGAS runtime also used by XL UPC. It aims to eventually be a fully compliant F2008 implementation.

A modest portion of the language is already implemented. Most sorely lacking when we were writing these benchmarks were allocatable coarrays and coarrays of derived types. But most typical uses of coarrays are stable with the implementation we used.

Only two CAF-specific performance optimizations have been implemented to achieve our performance results. The first was privatizing implicit coarray accesses; i.e., dereferences or assignments that do not specify a coindex explicitly with square brackets. The second was remote update: replacing a read, modify, write sequence to the same remote address with a message to the remote image to perform the update itself (atomically).

# 3 Benchmarks

In this section we discuss the concept and implementation details of our benchmarks. We deal with language differences and the tension between performance and productivity aspects.

## 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`. We use a distributed tiled data structure in order to facilitate parallelism and readability of the code.

### 3.1.1 HPL: Implementation considerations

Our 2009 UPC submission included a UPC language extension, "multiple blocking factors", that was supposed to help programmers express tiled data structures in UPC. The UPC language committee voted down our proposal, and encouraged us to make making tiled arrays into a UPC library, rather than an extension. This year's submission therefore includes the tiled array library with a UPC port.

The main purpose of the tiled array library is to perform the distributed index calculation for multi-dimensional tiles laid out using a block-cyclic array distribution. This alleviates programmer effort and decreased readability in the needed (i.e modulo-arithmetic) steps to compute which tiles in a distributed array are local or remote to the given thread. The tiled array library's main interesting feature is its reliance on one-sided communication. This makes it natural for both CoArray Fortran and UPC, but rather awkward to use with MPI. Figure 1 illustrates some of the idioms implemented by the library. The tiled array library provides routines for array allocation/deallocation, traversing a tiled array in a loop, and retrieving a pointer to the local data in the tiled array. Note that the library does not perform any communication; if a user attempts to read from a remote portion of the array, the library will return NULL. On the other hand, if a user attempts to read data that is local to that thread, the library will return a pointer to the local data for the given index.

Unfortunately, due to time constraints and the relatively late availability of the CoArray Fortran compiler, we have not been able to complete the HPL code. Work has not stopped, and we *may* (or may not) have a working implementation of HPL in CoArray Fortran by Supercomputing.

### 3.1.2 HPL: Verification

Verification works as follows: we initialize the LU matrix using random numbers. The last column in each row is set to the sum of all elements in the row. This ensures that the solution of the linear equation is a vector of 1.0 values. We compute the error as the average deviation of the actual obtained values from 1.0. This scheme has the additional

| Initialization | Loop with affinity |
|---|---|
| `UPCTiledArray_t A;`<br>`int Adims[] = { M, N}, blks[] = { BF, BF };`<br>`int thrds[] = { tx, ty } , comms[];`<br>`tilelib_upcalloc (&A, sizeof(double),`<br>`2, Adims, blks, thrds, comms);` | `TILELIB_FOR(i, A, 0, 0, M, BF) {`<br>`TILELIB_FOR(j, A, 1, 0, M, BF) {`<br>`/* A[i,j] is local here */`<br>`double *a = tilelib_vgetaddr(A,2,i,j);`<br>`a[...] = ... ;`<br>`}`<br>`}` |
| remote Get | |
| `double a_kk[BF*BF];`<br>`shared void * A_kk =tilelib_vgetraddr(A,2,k,k);`<br>`upc_memget(a_kk, A_kk, BF*BF*sizeof(double));` | |

Figure 1: Tiled array library features

useful property that it is invariant during execution of the HPL algorithm, allowing us to monitor correctness during execution instead of waiting for the algorithm to end.

#### 3.1.3 HPL: Productivity vs. performance: split-phase memory operations

One of the lesser known high performance features of UPC is the availability of split-phase memory operations. These primitives were pioneered by Berkeley UPC [12] and implemented by IBM UPC.

We found the existence of split-phase memory operations a mixed blessing. The HPL benchmark needed 2 weeks of intensive debugging because of an optimization involving a broadcast overlapped with a split-phase GET operation (file `hpl_panelfactorize.c`, lines 87–95 - we left the broken code for the reader's enjoyment). The source of the GET operation overlapped with the target of the broadcast, which caused a data race.

The data race only showed up in Power5 runs, causing occasional wrong answers. We *never* saw this error on Blue Gene/P, causing us to (wrongly) suspect the runtime for a long time. In addition, the savings caused by the overlapped data transfers are negligible. We conclude that in this case the net effect of the high-performance feature was overall negative: no performance gains, and weeks of wasted effort.

### 3.2 Global FFT

Global FFT performs a Discrete Fourier Transform on a one-dimensional array of complex values. The [source, destination, work] array(s) in the benchmark are sized to fill half of the total memory of the machine. The arrays are evenly distributed across the memory of the participating processors.

The cluster implementation of FFT calls for a sequence of local DFT transformations followed by global and local transpositions across the nodes. The local DFT transformations are implemented by calls to ESSL, but the transformations are implemented by our code.

#### 3.2.1 FFT: Implementation

We implemented local DFTs by calling the ESSL `dcft` function. We spent some effort on the transpositions. The FFT algorithm requires the global FFT matrix to be transposed. However, the matrix is not *tiled* in memory; it is blocked by processors and distributed in a normal linearized fashion, as required by the ESSL calls. Figure 2 shows the UPC and CoArray Fortran declarations for the working arrays.

In order to transpose the arrays as declared above, two strategies present themselves. First, transposition can be executed line by line. This tends to generate a large number of short network transactions. The second strategy involves first coalescing the buffers into pieces that can be exchanged over the network with a single call to `alltoall`.

We implemented both strategies in the UPC code, but only the coalesced strategy in CoArray Fortran . We found that on Blue Gene, the coalesced strategy always wins (because it allows us to invoke the optimized `alltoall`

| UPC declaration | CoArray Fortran declaration |
|---|---|
| `#define BF ((N2/(unsigned long long)THREADS))`<br>`typedef shared [BF] complex_t ComplexArray_t [N2];` | `parameter (N=(2**M),bsize=N/PROCS)`<br>`double complex, save :: A (N,bsize)[*]` |

Figure 2: FFT array declaration in UPC and CoArray Fortran

```
for (i=MYTHREAD; i<NUPDATE; i+=THREADS) {
  ran = (ran<<1)^(((int64_1)ran<0)?POLY:0);
  Table[ran%TSize] = Table[ran%TSize] ^ ran;
}
```

```
do i=1,updates_per_image
  a     = stream_next(a)
  image = iand(ishft(a, image_shift), image_mask)
  idx   = iand(a, index_mask)
  T(idx)[image]=ieor(T(idx)[image],a)
end do
```

Figure 3: UPC and CoArray Fortran versions of RandomAccess

implementation). Small configurations of Power clusters prefer the non-coalesced approach; however, as the size increases, coalescing wins out here too.

The CoArray Fortran compiler would not allow us to write an efficient implementation of the non-coalesced strategy, which involves many split-phase PUT operations to be posted simultaneously. The current state of the compiler prevents it from generating non-blocking PUT calls internally, and of course the CoArray Fortran language does not define split-phase operators.

#### 3.2.2 FFT: Verification

In 2009, our strategy consisted of a naive reverse FFT, followed by a comparison with the original random input. This strategy badly backfired on us: since the reverse FFT was not scalable, we did not check large runs. Thus at least one bug slipped in, and it caused both performance and correctness problems.

This year we removed the reverse FFT based verification process. Instead, we generated several pure sine waves as input, and produced a graph of the output (which is expected to be a pair of Dirac spikes at the appropriate places). We found that any problems with transpositions caused *visible* perturbations in the result. Thus, while the method may not be appropriate for automated testing, it is a great tool for finding implementation bugs.

The code to generate the plots is included with the submission.

### 3.3 Random Access

The core of the benchmark is a loop that updates a random array element in a random remote processor with a random value. This generates large amounts of network traffic. Hence the Random Access benchmark measures two things: (a) overheads and latency introduced by hardware and software in the network stack and (b) the network capacity itself.

#### 3.3.1 RA: Implementation

The kernel in the Random Access benchmark is throttled by the update operation: $T[i(a)] = T[i(a)]\ xor\ a$. The cost of the `XOR` operation is negligible, but the read and write of $T[i(a)]$ potentially introduce expensive network traffic. The remote update optimization replaces the read, update, write sequences with a message sent to the remote process to perform the update there atomically. This reduces two round trips to the remote process with just one, and decouples the sender from the receiver (i.e. the sender is not obliged for the update round trip to complete).

The Random Access kernel in UPC (Figure 3) is comprised of just 3 lines; one line to distribute work; one line to calculate the next random value and finally a line of code for the remote update operation itself.

The kernel in CoArray Fortran is very similar. Fortran's notation for bit operations isn't quite as compact as UPC's, but the code is almost a direct translation. The biggest difference is that unlike in UPC, the user must be aware of the segmentation of address space and be able to compute indices and coindices separately.

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

```
a(:) = b(:) + alpha * c(:)
```

Figure 4: UPC and CoArray Fortran versions of Stream Triad

#### 3.3.2 RA: Verification

Verification of the RA benchmark is performed by executing all updates twice. Correct execution should yield array values that are the same as the original - although the benchmark description allows small (1%) deviations from this. On the machines we ran on for this submission, all the updates are exact and therefore we always see 0% error rates; however, on newer machines like PERCS, there exists specialized not-quite-reliable hardware for executing remote updates; we have seen occasional dropped packets causing very small (orders of magnitude smaller than 1%) error rates.

#### 3.3.3 RA: Productivity

We eschewed complicated forwarding implementations of the algorithm. All next generation IBM HPC hardware is designed to deliver outstanding performance with short remote updates.

### 3.4 EP Stream Triad

The Stream Triad benchmark is designed to measure the memory bandwidth of a machine. It performs a combination of vector multiplication by a constant, and a sum on two source and one destination vectors.

#### 3.4.1 UPC Implementation

Figure 4 shows the code for the UPC and CoArray Fortran versions of the stream code.

The UPC code has a work distribution loop. All array references in the upc_forall loop above are actually local. Our previous work on shared array privatization allowed the compiler to recognize that the loop does not contain any communication and privatize the 3 array accesses in the stream kernel. The technique involved strip-mining the forall loop based on the computed affinities of the indexed shared variables.

Unfortunately this resulted in loop indices containing integer modulo and division computations. The proliferation of modulo arithmetic in UPC is caused by the block-cyclic array distribution. The overhead introduced by these computation is significant, but more importantly the complicated array index obfuscates the physical access pattern and prevents further downstream loop optimizations.

This year we have implemented a new optimization scheme that permits the compiler to recognize that consecutive array accesses performed by each thread have a physical distance in memory equal to the size of the array element (stride one access pattern). We accomplish this by remapping the `upc_forall` loop's iteration space from the UPC index space to the physical array distribution. The new loop has a normalized iteration space: the lower loop bound is zero and the loop increment at each iteration is the constant one. Local accesses are rewritten in terms of the much simplified induction variable and the modulo and division index arithmetic simply disappears.

The new scheme we have adopted has several advantages:

- It avoids the expensive modulo and division operation required to privatize array in our prior implementation,
- It exposes to real physical access pattern (stride one) to subsequent loop optimizations,
- It allows the traditional loop optimizer to prefetch the arrays into the cache (stream prefetching),
- On the Power7 architecture permits SIMDization of the vector operations in the stream kernel, that is automatic exploitation of the VSX unit available in the hardware.

In conclusion, the UPC code requires non-trivial compiler analysis to get good performance.

### 3.4.2 CoArray Fortran Implementation

By contrast, the CoArray Fortran code is a single line (Figure 4, right panel). All the coarray references are local, so the compiler privatizes them. It makes one function call outside of the implicit loop to obtain their local address. This happens very early in the optimization process, so the loop, to the rest of the optimizer looks like standard scalar code. In particular, the compiler found a stream prefetching opportunity – a Power7 specific optimization which warms up the cache with the data that is going to be loaded, before the loads occur.

This is an important feature of the CoArray Fortran compiler. Loops without communication treat coarray accesses like normal arrays with the benefit of all of the optimizers transformations for non-coarray code.

### 3.4.3 Verification

Given known initial values for the vectors the end result of the operation can be predicted. This known value is checked against the actual obtained values in the vector.

## 3.5 K-means clustering

The rules have changed this year and we have an opportunity to add a new kernel to the list. While many of our colleagues consider non-traditional, non-SPMD algorithms like UTS and graph breadth-first search, we considered a relatively simple problem that nevertheless caused us considerable difficulty: an implementation of Lloyd's algorithm [8].

The problem, in a nutshell, is to cluster a set of $N$ points $\{x_1, x_2, ...x_N\}$ in $D$-dimensional Euclidean space into $K$ sets $\{S_1, S_2, ...S_k\}$ so as to minimize the sum of Euclidean distances from the points to the clusters' centers $\mu_j, j \in 1..k$, i.e. calculate:

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2$$

### 3.5.1 k-means: Implementation

Lloyd's algorithm is an iterative technique. Each iteration involves two phases; the first, or *classification* phase assigns each of the $N$ points from the original set to one of $K$ centroids based on proximity. The second phase *relocates* every centroid to be the gravicenter of the points assigned to it. The two phases alternate until the desired precision is achieved.

A *naive* implementation of the algorithm is fairly computationally intensive, as the classification phase involves the calculation of the Euclidean distance between every pair of points and centroids, resulting in $O(N \times K \times D)$ complexity. There are complicated tree-based algorithms in existence that cut down computation to a fraction of the above number by eliminating redundant distance calculations. However, as far as we are aware, a three-level nested loop computing pairs of Euclidean distances exists in some form in every variant of optimized code:

```
 1  do p=1, N
 2    kmin=-1; dmin=\inf;
 3    do k=1, K
 4      d0 = 0
 5      do d=1, D
 6        d0 = d0 + (points[p][d]-clusters[k][d])^2
 7      end do
 8      if (d0 < dmin) { dmin = d0; kmin = k; }
 9    end do
10    nearest[p] = kmin;
11  end do
```

Figure 5: K-means clustering: pseudo-code for classification phase

It is this loop that has attracted our attention. We tried many compilers on several architectures including Power, Intel, Blue Gene, cell based systems and CUDA based systems. The naive loop (implemented as above) got *uniformly poor* performance results on every architecture.

| Benchmark | UPC l.o.c. | CAF l.o.c |
|---|---|---|
| HPL | 1607 | N/A |
| RandomAccess | 165 | 134 |
| FFT | 369 | 332 |
| Stream | 186 | 58 |
| k-means | 1139 | 361 |
| utilities | 324 | 351 |

Figure 6: Lines of code summary for all our benchmarks

#### 3.5.2 k-means: Classification Phase Loop Analysis

In terms of computation to bandwidth ratio goes, the loop in Figure 5 generally corresponds to a BLAS-2 primitive. It is thus limited by the available memory bandwidth in a system. Line 6 in the loop requires one memory load for every 3 floating point operations, assuming that the compiler is smart enough to unroll and fuse either the p loop or the k loop and therefore eliminate one of the two memory loads from the innermost loop (unrolling the d loop is counterproductive because D values are small, limiting the gain). Changing the data layout, loop order and unrolling factors profoundly affects performance.

The second performance limiter in this loop is the if statement just outside the innermost loop (line 8). Unrolling the k loop causes a proliferation of these if statements, which then proceed to cause a series of branch misspredictions on machines that do not support some form of predicated execution.

#### 3.5.3 k-means: Parallelism

The main loop is very computationally intensive. The typical approach is to distribute the point set (which is large) and replicate the centroid values (which are relatively fewer). The classification phase thus becomes embarrassingly parallel, while the distributed gravicenter calculation in the second phase can be implemented using reduction-style collectives. Because the computation is concentrated in the classifier kernel, the algorithm has very good strong scaling properties.

#### 3.5.4 k-means: Our implementations

In this submission we describe the UPC and CoArray Fortran based implementations (although we have in the past experimented with cell and CUDA based implementations too [9]).

Both our implementations roughly correspond to the loop described in Figure 5. Since peak bandwidth on Blue Gene/P cannot be achieved except by using "double hummer" vector loads, we created a specialized version of the loop that uses XLC compiler intrinsics to express vector loads.

#### 3.5.5 Why k-means?

We consider k-means a good replacement for the stream benchmark. Like stream, the kernel stresses the machine's peak bandwidth. Like stream, the code itself is almost trivial. However, unlike the stream benchmark, the resulting access pattern is not trivial and compilers have to work at figuring out the best way to bring the best machine performance out. The kernel is highly relevant in areas including (but not limited to) high performance computing applications.

### 3.6 Lines-of-code table

The table in Figure 6 lists the lines of code for our 5 benchmarks, plus a common directory with utilities like the Mersenne Twister random number generator, and timing utilities. These are reused across the benchmarks, so we did not want to have them counted several times. Note the exceptionally large number for k-means: we counted all three variants of k-means. The l.o.c. numbers seem to correlate well across the two languages, even though different teams wrote the benchmarks.

# 4 Performance

In this section we describe the way we ran our experiments, and the performance we obtained.

## 4.1 Hardware Setup

For our experiments we used systems based on Power5, Power7 and Blue Gene/P nodes.

- P7 cluster: This machine is composed of 32 HV32 CECs (or nodes). Each CEC has 32 cores, 234 GBytes of memory, 4 gigabit ethernet adapters, and 1 Infiniband adapter. During the test the cluster is configured as 1 LPAR (logical partition) per CEC and booted into SMT-4 mode. The LPARs are installed with IBM's HPC software stack: RHEL6 RC1, LAPI 4.1, SCI 2.0, PE 5.3, LoadL 4.2, and ESSL 5.1.
- P5 cluster: since the P7 system is in high demand and hard to schedule for extended runs, we substituted many of our scaling measurements with the NCSA BluePrint system. BluePrint consists of 120 Power5+ nodes, each having 16 cores and equipped with 64 GBytes of memory. The system is connected by IBM's High Performance Switch.
- WatsonShaheen is a 4 rack Blue Gene/P system sporting 16384 processors running at 850 MHz and equipped with 1 GByte of memory on each processor.

## 4.2 Performance measurements

### 4.2.1 HPL: scaling and performance

The HPL numbers we measured this year reflect the performance of the tiled array library. We measured HPL on Blue Gene/P and on the NCSA Power5 BluePrint cluster. Normally we attempt to load the machines to as close to the full memory as we can, but on the Power5 cluster this would have meant run times on the order of days, and an abuse of privilege on our part. Hence the Power5 nodes were sized only a bit above 50% memory load.

Figure 7 shows completely flat scaling to arbitrary machine size on both architectures. Performance on Blue Gene/P is identical to last year's measured performance, even though the implementation has changed radically. We continue to lose 10-10% of the peak performance to load imbalance in the panel factorization code; this is a limitation of our implementation (which continues to be around 2,000 lines of code, even with the tiled array library added).

The figure shows that while we spend 80+% of execution time in computation, we still only achieve 58% of peak performance. This is due to ESSL itself only achieving 80% of peak performance.

### 4.2.2 FFT: a comparison of transposition algorithms

Figure 8 shows the performance of global FFT. The left panel shows performance of two variants of UPC code on Blue Gene/P; the right panel shows both UPC and CoArray Fortran performance on the Power5 cluster.

- UPC numbers are better than in last year's submission, largely because we found and eliminated several performance bugs (and even correctness bugs) in our submission code.
- In Section 3 we discussed the difference between coalesced transposes and line-by-line exchanges. While the coalesced transposes win on Blue Gene/P, they are counterproductive on the Power5 machine. This is due to the different balance of memory bandwidth, machine size and interconnect bandwidth.
- We only had one CoArray Fortran implementation, coalesced transposes. However, our implementation is suboptimal since we were not able to write a blocked local transpose routine before submission deadline. Thus performance in the CoArray Fortran case suffers substantially. We hope to correct this by the final presentation time.

Blue Gene/P performance:

| Name | NC | 4NC | MP | 1R | 2R | 4R |
|---|---|---|---|---|---|---|
| processors | 128 | 512 | 2048 | 4096 | 8192 | 16384 |
| % peak | 56% | 58% | 58% | 58% | 58% | 57% |

Power5 cluster performance:

| processors | 16 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|
| % peak | 55% | 58% | 57% | 55% | 55% |

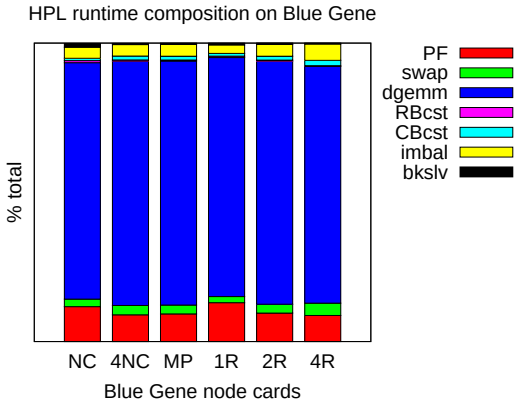


Figure 7: UPC HPL performance graphs; achieved percentage of peak (left) and composition of execution time (right). In the figure on the right side, "PF" stands for panel factorization; RBcst and CBcst are times spent distributing blocks for the rank-k updates; "imbal" is the load imbalance during the updates; and "bksolv" is the back-solver. Computation ("dgemm") dominates the execution time.

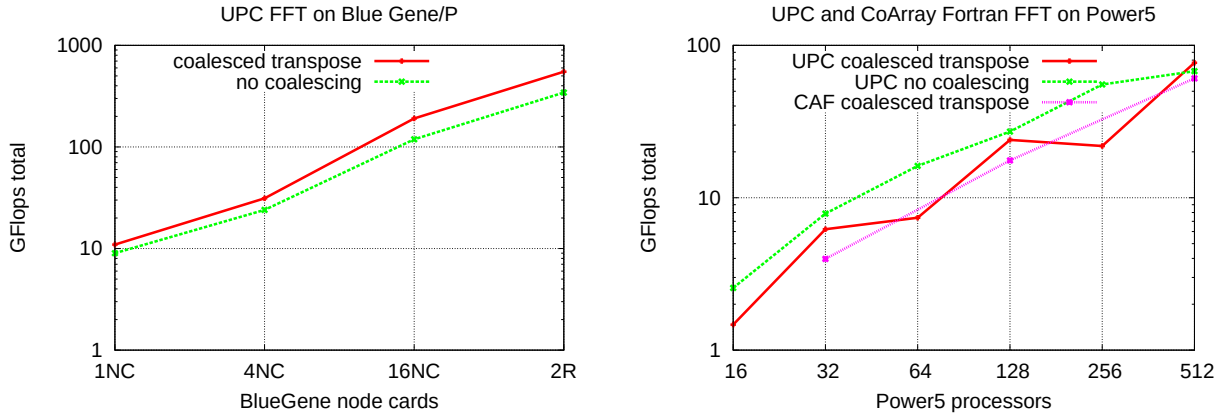


Figure 8: Global FFT weak scaling numbers: Blue Gene/P WatsonShaheen and Power5 BluePrint

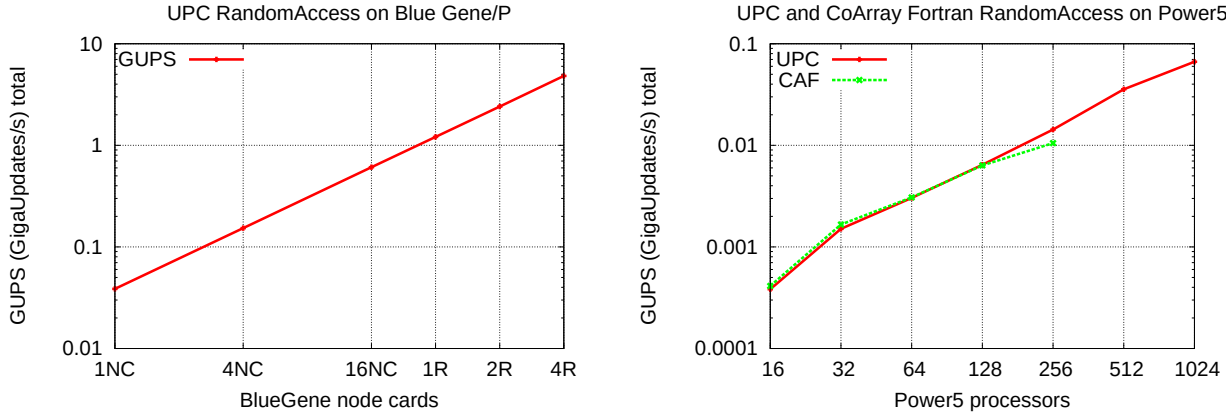


Figure 9: RandomAccess benchmark performance: Blue Gene/P WatsonShaheen and Power5 BluePrint

EP Stream Triad on Blue Gene/P

| CPUs | 1 | 4 | 16 | 64 | 128 |
|---|---|---|---|---|---|
| naive | 1.1 | 4.3 | 17.2 | 79.1 | 158.2 |
| optimized | 2.8 | 12.1 | 48.3 | 193.1 | 360.1 |
| peak | 3.4 | 13.6 | 54.4 | 217.2 | 434.4 |

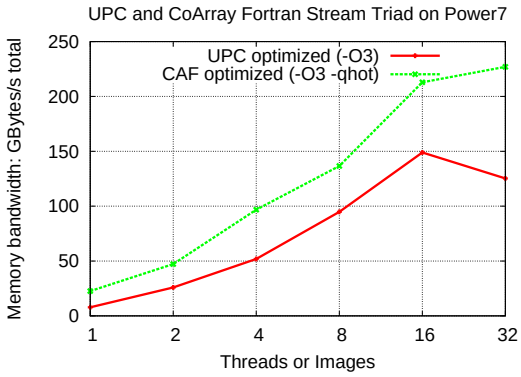


Figure 10: All UPC and CoArray Fortran Stream Triad measurements

#### 4.2.3 RandomAccess performance

Figure 9 shows the performance of the RandomAccess benchmark.

- In earlier years our Blue Gene submissions tended to not have linear behavior, because the application was reaching the network cross-section limit. The WatsonShaheen machine is however too small to reach that limit; RandomAccess performance is limited by the overhead of network interaction. This explains the linear scaling behavior of the machine.
- UPC and CoArray Fortran numbers look very similar because the same compiler opportunities are being exploited (remote update optimization), and the runtime/network interaction is done through shared code.

#### 4.2.4 Stream Triad: scaling and performance

We measured Stream on Blue Gene/P and on a Power7 cluster. The Blue Gene/P numbers did not benefit from the improvement in compiler technology; however, we wrote a hand-optimized version of UPC stream for Blue Gene in order to ascertain what the achievable peak is. The left panel of Figure 10 contains bandwidth numbers, both normal and hand-optimized, running on up to one node card (128 processors) of Blue Gene/P. The numbers suggest that at least the hand-vectorized code is close to ideal performance.

By contrast the Power7 numbers (right panel in Figure 10) did benefit from compiler improvements. For Power7, we also were able to run the CoArray Fortran program; the disparity in performance shows that the UPC compiler can be further improved to reach the same performance as CoArray Fortran . The CoArray Fortran compiler is not hampered by any of the obstacles described in Section 3. Our Power7 measurements were done on both the scaling cluster and standalone Power7 nodes; the results currently included in this section come from a standalone node.

#### 4.2.5 k-means: scaling and performance

Figure 11 shows the performance of the k-means benchmark. We measured the performance for several versions of code. Since k-means is almost embarrassingly parallel we did not attempt to scale it very far; we consider the algorithm a good candidate for *strong scaling* analysis, but have not explored in that direction yet.

On Blue Gene/P we tested two UPC variants of the k-means kernel. The "simple" version of the code has the three loops - N, K and D - but the K loop is unrolled $4\times$ and fused into the innermost D loop. The more complicated version of the loop uses XL compiler intrinsics to force the generation of Blue Gene/P vector instructions including "double hummer" loads and stores.

In terms of compute power, the innermost loop requires 3 floating point operations. The Blue Gene/P can execute 4 floating point operations per cycle; based on floating point power alone the innermost loop could be executed in a single cycle.

| System | Benchmark | loop time | Perf. |
|---|---|---|---|
| BG/P | UPC naive (-O3) | 15.5 ns | 0.2 |
| BG/P | UPC vectorized (-O3) | 6.3 ns | 0.48 |
| P7 | UPC naive (noopt) | 3.20 ns | 0.94 |
| P7 | UPC vectorized. (-O3) | 0.69 ns | 4.36 |
| P7 | CAF naive (noopt) | 9.55 ns | 0.31 |
| P7 | CAF naive(-O3 -qhot) | 1.19 ns | 2.51 |

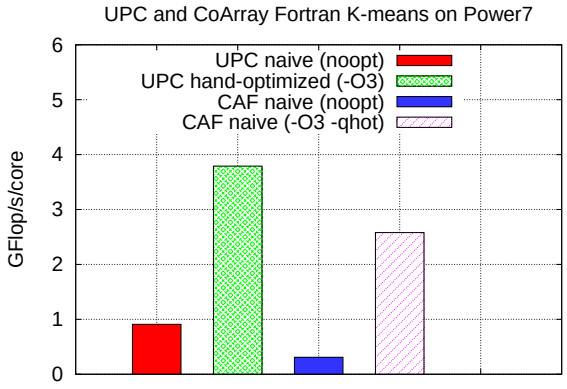


Figure 11: UPC and Coarray Fortran k-means runs

In terms of memory bandwidth, a properly unrolled innermost loop requires just a bit more than one double precision floating point load for every iteration. Since the Blue Gene/P processor's maximum bandwidth is around 4 Bytes/cycle (or 3.4 GBytes/s), this would allow it to complete the innermost loop in 2 cycles. However, the nature of the innermost loop causes the memory to be fetched in a non-linear order, which limits the performance to about 2 Bytes/cycle. Our best hand-vectorized code comes close to executing the innermost loop in 4 cycles.

Unfortunately the many comparisons and branches just outside the innermost loop cause enough performance degradation that the apparent loop time (measured as total time divided by innermost loop iterations) never drops below 5 cycles.

On Power7 machines we tested both the CoArray Fortran and UPC codes. Here again the hand-optimized code performed best. Surprisingly the naive CoArray Fortran implementation did fairly well even compared to the hand-optimized UPC code, which again underscores the fact that the optimization problem in CoArray Fortran is much simpler than in UPC.

# 5 Conclusion

We have discussed in this paper the implementation and performance of five different benchmarks using two modern parallel programming languages, UPC and CoArray Fortran and three different hardware architectures. The results show the applications achieve very good performance in terms of both per CPU utilization and scaling across large systems. We will continue our work on the compiler, the run time system and finding new ways to express parallel application using both UPC and CoArray Fortran . The results of these ongoing efforts will be available for the final presentation.

# 6 Acknowledgments

Work on this paper was done under the aegis of the DARPA PERCS program (Agreement #HR0011-07-9-0002).

Runs on the NCSA BluePrint system were sponsored by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation. We are personally grateful to Mike Showerman and Bill Gropp for letting us run on their system.

We would also like to thank Fred Mintzer and Dave Singer, in charge of the IBM WatsonShaheen system, for their patience when dealing with our machine requests.

# 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 Hoeflinger, George Almasi, Basilio B. Fraguela, 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] Bonachea, Dan. Proposal for Extending the UPC Memory Copy Library Functions. *Lawrence Berkeley National Lab Tech Report LBNL-56495 v2.0*, March 2007.

[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] A. Gara et al. Overview of the Blue Gene/L system architecture. *The IBM Journal of Research and Development*, 49(2/3), 2005

[6] The HPL Algorithm. `http://www.netlib.org/benchmark/hpl/algorithm.html`.

[7] UPC Consortium. *UPC Language Specification, V1.2*, May 2005.

[8] k-means on Wikipedia: http://en.wikipedia.org/wiki/K-means_clustering

[9] k-means at apgas 2009 http://research.ihost.com/apgas09/apgas09program.html

[10] R. Numrich and J. Reid, Co-array Fortran for parallel programming. SIGPLAN Fortran Forum, Vol. 17(2), pp. 1–31, 1998

[11] The IBM xlUPC compiler download website: `http://www.alphaworks.ibm.com/tech/upccompiler`

[12] Berkeley UPC download website: http://upc.lbl.gov/download/