RC24364 (W0710-010) October 2, 2007
Computer Science

IBM Research Report

Performance without Pain = Productivity: Data Layout and Collective Communication in UPC

Rajesh Nishtala
Computer Science Division
University of California at Berkeley

Gheorghe Almási, Calin Cascaval
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

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

Performance without Pain = Productivity

Data Layout and Collective Communication in UPC

Rajesh Nishtala[†] George Almási[‡] Călin Caşcaval[‡]

[†] Computer Science Division
University of California at Berkeley
rajeshn@eecs.berkeley.edu

[‡] IBM T.J. Watson Research Center
Yorktown Heights, NY
{gheorghe,cascaval}@us.ibm.com

Abstract

The next generations of supercomputers are projected to have hundreds of thousands of processors. However, as the numbers of processors grow, the scalability of applications will be the dominant challenge. This forces us to reexamine some of our fundamental ways that we approach the design and use of parallel languages and runtime systems.

In this paper we show how the globally shared arrays in a popular Partitioned Global Address Space (PGAS) language, Unified Parallel C (UPC), can be combined with a new collective interface to improve both performance and scalability. This interface allows subsets, or teams, of threads to perform a collective together. As opposed to MPI's communicators, our interface allows set of threads to be placed in teams instantly rather than explicitly constructing communicators, thus allowing for a more dynamic team construction and manipulation.

We motivate our ideas with several important applications such as Dense Matrix Multiplication, Cholesky factorization and multidimensional Fourier transforms using the new collectives interface. We describe how the three aforementioned applications can be succinctly written in UPC thereby aiding productivity. We also show how such an interface allows for scalability by running on up to 16,384 processors on the BlueGene/L. In a few lines of UPC code, our dense matrix multiply routine achieves 28.8 TFlop/s. We analyze our performance results through models and show that the machine resources rather than the interfaces themselves limit the performance.

Keywords Parallel programming, PGAS, UPC, Collective communication, Programming productivity, Blue Gene

1. Introduction

As the demand for computational power continues to increase for both scientific and commercial applications, machines exhibiting large scale parallelism are becoming ubiquitous. Leading the way, the IBM Blue Gene architecture already provides more than a hundred thousand processors in its large configuration [22].

One of the biggest challenges facing programmers for these machines is application scalability. This affects not only algorithm design, but also the design parallel languages and runtime systems. A new class of languages, called Partitioned Global Address Space (PGAS) languages, has recently emerged to aid in the performance and scalability of these applications. Rooted in traditional shared memory programming models, these languages expose a global address space that is logically partitioned across the processors. The design of these languages recognizes the fact that memory access times in modern machines in non-uniform. Compared to OpenMP [32] which assumes a shared memory system or MPI [28] that is designed for distributed memory systems, the PGAS languages are better positioned to provide a unique programming model for modern machines.

In this paper we demonstrate how applications written in Unified Parallel C (UPC) [36], one of the most popular PGAS languages, can perform on par with hand-coded applications. We make minor extensions to the language to allow for data layouts more suitable to the linear algebra domain, and develop a new collective communication library that takes advantage of the data affinity and one-sided communication expressed in UPC. While we demonstrate these aspects in the context of UPC, these optimizations are equally applicable to other PGAS languages, such as Titanium [42] and Co-Array Fortran [30].

The multidimensional blocking directive [9] is a syntactical extension to the declaration of shared data that allows finer control over data distribution. This extension enables us to directly call high performance linear algebra libraries such as [19, 12] that use contiguous blocks of memory to represent matrices. Significant effort has been spent implementing serial versions of such libraries, and a productive parallel language must support efficiently the use of native libraries.

Most large scale machines have hardware support for collective communication and optimized MPI libraries that take advantage of this support. However, MPI libraries are typically designed to be thread-centric, that is, the user is forced to think about which threads or processors it must communicate with. To exploit the collective communication hardware primitives in PGAS languages, we propose a new, data centric, collective communication interface that takes advantage of the shared data semantics expressed in a PGAS program. The threads participating in our collective operations are dynamically determined by data affinity, thus paving the way for modern, dynamic and adaptive applications such as Adaptive Mesh Refinement (AMR) [11] to take advantage of optimized collective communication libraries.

To summarize, our paper makes the following contributions:

- a new, data-centric collective communication interface design that takes advantage of shared data semantics, instant teams, and one-sided communication semantics to provide the user with high-performance collective communication
- propose simple annotations to enable data distributions and layouts that allow the programmer to exploit high-performance serial libraries, such as those developed for linear algebra;
- demonstrate productivity and performance for three application kernels: parallel dense matrix multiplication, 3D Fast Fourier Transform (FFT) and Cholesky factorization on a large scale Blue Gene/L configuration – with a few lines of UPC code we show linear scalability and performance comparable to the best hand tuned kernels developed for this machine.

The rest of the paper is organized as follows: in Section 2 we discuss limiting factors for the performance and scalability of applications on large scale machines; in Section 3 we describes our new collectives interface and in Section 4 we discuss productivity aspects focusing on the three kernel applications; we present performance results in Section 5, discuss related work in Section 6 and conclude in Section 7.

2. Performance Considerations

High performance versions of the sequential linear algebra subroutines exist for most architectures: Atlas [40], ESSL [19], the Intel Math Kernel Library [26], FFTW [21] and SPIRAL [33]. However, for large scale parallel machines the landscape is completely different: application programmers have to handcode parallel kernels, distribute data and load balance the computation for each workload, incurring significant effort and cost. For example, individually optimized versions of HPL and FFT for these architectures run to many lines of code, encompass multiple man-years of effort, and are far from ubiquitous.

One of the key questions that we are addressing in this paper is "Can a high productivity language be used to express the inherently complicated set of requirements posed by a high performance parallel algorithm?" To answer the question, we shall go through a simple exercise: write a parallel version of a simple algorithm, such as matrix multiplication (MM), for a large scale machine and strive to achieve performance on par with the best hand-optimized codes, without sacrificing readability and programmability. We choose UPC as the high productivity language.

High performance parallel implementations of MM are typically parallelized and blocked at multiple levels. Parallelism is employed at instruction level to take advantage of ILP and at thread/CPU level to take advantage of multiple functional units. Blocking is used to take advantage of locality at register, cache, memory and network level. A number of parallel matrix multiply algorithms exist [14, 38, 16]. Our implementation is essentially a blocked version of the SUMMA algorithm.

The UPC language constructs provide easy parallelism through the `upc_forall` construct and data blocking. However, upon closer examination we note that UPC does not support data blocking in more than one dimension, requiring the user to manually linearize two-dimensional matrix indices. To alleviate this problem, we proposed UPC syntax additions in [9] to enable multidimensional tiled arrays in UPC. Tiled arrays are declared as follows:

```
shared [b0][b1]...[bn] <type> A[d0][d1] ... [dn];
```

As many blocking factors can be added as there are array dimensions. The blocking factors are not required to divide array dimensions. Tiled UPC arrays enable us to directly call the serial BLAS optimized version of the kernel and thus the resulting matrix multiply code is very simply as shown in Figure 1. This code is compact and intuitive. It relies on a sequential BLAS implementation for performance, but takes explicit care of parallelism (by means of the `upc_forall` loops) and communication aspects (by using the `upc_memget` builtins). Communication is aggregated, which makes for better efficiency.

```
for (kk=0; kk<P; kk+=B) {
  double a_local[B*B], b_local[B*B];
  upc_forall (int ii=0; ii<M; ii+=B; continue)
    upc_forall (int jj=0; jj<N; jj+=B; &A[ii][jj]) {
      upc_memget (a_local, &A[ii][kk], sizeof(double)*B*B);
      upc_memget (b_local, &B[jj][kk], sizeof(double)*B*B);
      dgemm ("T", "N", &n, &m, &p, &alpha, b_local, &B,
              a_local, &B, &beta, (void *)&C[ii][jj], &B);
    }
}
```

Figure 1. UPC code for parallel matrix multiplication

However, the code snippet in Figure 1 also suffers from unpredictable performance and has scalability issues. In [9] we demonstrated good scaling up to about 16 UPC threads, beyond which efficiency decayed rapidly. Subsequent examination revealed two major causes for this decay.

- **Processor layout:** While the matrices are tiled, the programmer has no real control over the distribution of tiles among the processing elements, making a mapping to a high performance network topology next to impossible.
- **Communication pattern:** Communication in Figure 1 is point-to-point, resulting in communication imbalance and waste of bandwidth.

We propose the following techniques to address these problems: First, we allow the programmer to specify a Cartesian processor distribution for a UPC array. This roughly corresponds to Cartesian topologies in MPI: an ability to denote threads with a tuple $< t_0, t_1, ...t_n >$ instead of a single number $t, 0 \leq t < THREADS$. This has been done by other languages, such as HPF[23] and ZPL[15]. We propose a syntax similar to HPF, in which processor mappings are named and shared arrays are mapped to these distributions. The system verifies at runtime whether the distributions are legal and ignores them if they do not match the current running configuration.

```
#pragma processors distribution_name(X, Y)
shared [B1][B2] (distribution_name) int A[N1][N2];
```

Second, we replace point-to-point communication primitives with UPC collectives that are both easy and intuitive to program and can take advantage of the cartesian topologies. We discuss these issues in the next section.

3. Collective Communication

Collective communication operations (or collectives) are abstractions that encapsulate common data movement patterns that most parallel programming models provide. This section describes how our new interface can succinctly express common communication patterns across shared arrays found in PGAS languages. The main attributes of these new collective interface are the following: data centric communication, subset of processors participating in collectives (teams), and the fact that the teams are dynamically constructed based on affinity.

Since UPC is a data parallel language, our primary goal is to make the collectives *data centric* rather than the thread centric model employed by many other parallel programming models. UPC provides distributed arrays as described in Section 2. These arrays create natural partitions of the processors in each of the

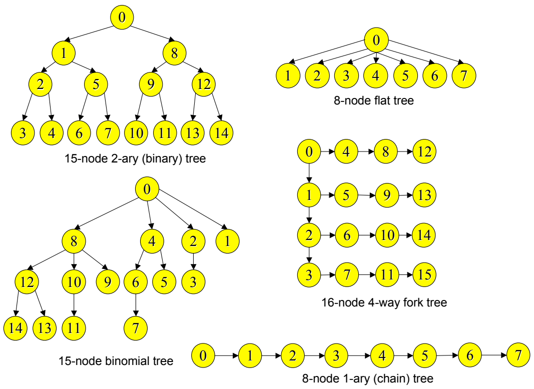

Figure 2. A few example trees used in collectives

dimensions of the multi-blocked array. We call these partitions of the processors *teams*. The closest equivalent to teams are the MPI[28] *communicators*.

3.1 Proposed Collective Model

In MPI the user is required to explicitly specify the team members and handle his own packing of the data into contiguous blocks so that the collectives can handle the transfers. The current UPC collective interface lacks support for teams and imposes very strict thread centric data layout rules while also requiring the user to handle the packing and unpacking. Our proposed extensions would allow the users to specify the blocks of data involved and let the underlying runtime system handle the problem of mapping data blocks to threads and packing the data into contiguous blocks. Since the runtime system already has full knowledge of how the shared arrays are mapped onto the processors, the overhead of obtaining this information is relatively small.

For efficiency reasons we require that all threads that have affinity to one of the active blocks of data to make a call into the collective, similar to the current MPI and UPC collective models. This allows us to build a scalable implementation of the communication schedule by letting the collective build a tree over all threads participating in the collective. In a broadcast, for example, the root thread can then send data to a small set of children who then forward the data down the tree. This type of communication schedule for broadcast increases the available bandwidth and decreases the latency since more of the network resources are being used in parallel. Figure 2 shows an example of various tree topologies rooted at thread 0. Our experience showed that the fork trees mapped well to the BlueGene network while other topologies are likely to do better on different networks. If a thread that does not have affinity to any of the the data involved the call is treated like a no-op. An alternative implementation and interface would require only one thread to handle the data movement for all the threads involved in the collective, without the need for any of the other threads to participate in the collective. However, this forces the implementations to either (1) always use flat trees, which severely hinders scalability at large processor counts, or (2) have an auxiliary thread on each node that is not part of the runtime that handles the collective communication responsibilities for that node. The overheads for a such a thread on some platforms will be prohibitively expensive due to the overheads of context switching, thread scheduling and memory coherence issues.

3.2 An Example Interface

As mentioned above, the main goal of this work is to show that the data centric collectives alongside the shared arrays in UPC can simultaneously provide productivity as well as performance. We are less concerned with the exact formal specification of the collectives in the language. While we are going to work with the UPC language consortium to propose these collectives, we first want to demonstrate their usefulness, and consider syntax to be outside the scope of the paper.

We use a Matlab [27] style interval notation to specify blocks of data in each dimension that will be used in the collective. The application experience in using these collectives motivated the specification of block indices in the interval rather than the array indices themselves. However this decision is not fundamental to the interface. Also our interface adheres to the current UPC collective synchronization specification, since they are intended be an extension of the current UPC language specification.

To motivate the interface we will discuss one example in each of the two collective categories: (1) one-to-many (e.g., broadcast) and (2) many-to-many (e.g., exchange[1]). Section 4 goes into further detail on how these operations can be incorporated into real applications.

- **One-To-Many**
  In the first category of collectives, one root block contains the data to be disseminated to other blocks of the shared array. Common collectives in this category include `broadcast()` and `scatter()`. Typical scalable implementations of these operations construct a tree over the threads rooted on the thread that owns the original data. In the case of a broadcast the root processor sends the data to a small subset of the processors and these other processors then forward the data down the tree. In our interface (as well as the current UPC collective specification), the user specifies a shared pointer rather than explicitly specifying the root thread.

  In addition, we allow the user to specify a list of intervals to the destination which dictate which blocks the broadcast data will be stored into. The number of intervals is dictated by the number of dimensions of the shared array. The proposed prototype for this type of collective is:

```
upc_stride_broadcast(shared void* dst<intervals>,
                     shared void* src,
                     size_t len, int sync_flags);
```

  The example in Figure 3(a) declares a two-dimensional destination array. The `src` array broadcast the data into every other row and column of the `dst` matrix.

- **Many-To-Many**
  In the next important category of collective operations, every output block involves data from *all* the input blocks. The input blocks are likely to be distributed across many processors. Scalable implementations of these collectives carefuly tune the communication schedule to avoid creating hotspots in the network, however the performance is often limited by the bisection bandwidth of the network. Many methods[13] also exist for performing the communication in $O(N\ log\ N)$ rounds rather than $O(N^2)$ rounds.

  A popular example of a collective in this category is `exchange()`. This collective breaks each block in the input array into k pieces of len bytes each. It is assumed that there are k blocks specified in the each of the source and destination intervals. It then takes the i^{th} slot from block j and places it into the

[1] In MPI parlance this operation is MPI_Alltoall

```
shared [] int src[4];
shared [4][4] int dst[200][200];

upc_stride_broadcast(dst<0:2:49,0:2:49>,
src, sizeof(int)*4, 0);
```

(a)

```
shared [10] int src[100][100];
shared [10] int dst[100][100];

upc_stride_exchange(dst<0,:>, src<:,0>,
sizeof(int), 0);
```

(b)

Figure 3. Strided Collective Examples. (a) A code-snippet to perform a broadcast `src` to every other row and column of the `dst` matrix (b) A code-snippet to exchange dat from the first column of `src` into the first row of `dst`

j^{th} slot in block i on the destination. The following prototype illustrate our proposed interface:

```
upc_stride_exchange(shared void* dst<intervals>,
                    shared void* src<intervals>,
                    size_t len, int sync_flags);
```

Figure 3(b) shows an example of an exchange operation. In the example all the blocks in the first column of the source matrix are exchanged into the first row of the destination matrix.

Notice that neither the definition nor the examples of collectives interface require the user to specify the identity or number of threads involved in the communication. The threads invovled are implicitly defined by specifying which blocks of data the collective is to be run across. Since there is no explicit mention of how many blocks a particular thread owns, it is up to the implementation to infer this information and make the correct decisions on how to correctly pack the data. Allowing the runtime to make such decisions rather allows for much greater performance portabiltity.

However, since we do require all the threads with affinity to any part of the memory being communicated call the collective we provide a simple utility function that can query whether the calling thread has affinity to *any* part of the data. Since this information is already stored inside the runtime system such query functions will be fast.

```
int upc_haveaffinity(shared void* arr<intervals>);
```

The function will return 1 if the calling thread has affinity to any of the data in the specified interval or 0 otherwise.

3.3 Hiding the Tuning Process

We are also witness to a large variety of processors and the interconnection networks. The wide variety of machines makes the prospect of tuning communication schedules at the application level infeasible. By using a collective interface and a runtime system that can handle efficient packing and unpacking of the data, these tuning issues are left in the hands of network and runtime system designers who often have access to much lower level network features to implement the operations. Our implementation, for example, on the BlueGene/L leveraged many features in the BlueGene messaging layer [4] (such as Active Messages) that have not been exposed to the UPC programmer. Creating this abstraction ensures performance portability since the application designer need not worry about the intricacies of tuning the communication schedules.

4. Productivity

Thus far we have motivated the use of the global shared arrays and a new collective interface that takes advantage of these arrays. In this section we show how three common and important computational kernels can be written very succinctly in UPC with our proposed additions. In Section 5 we demonstrate that the codes presented in this section scale to thousands of processors.

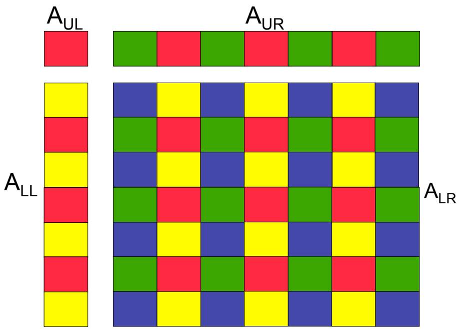

Figure 6. One step of the dense matrix factorization recursion

4.1 Dense Matrix Multiplication and Dense Cholesky Factorization

One of the most commonly used computational kernel in large scale parallel applications is dense matrix multiplication. In this kernel we perform the operation $C = A \times B$ where A, B, and C are dense matrices of sizes $M \times P$, $P \times N$, and $M \times N$, respectively. In order to effectively parallelize the problem, each of these matrices must be partitioned across the processors.

In addition to being an important kernel in itself, this operation is considered the rate limiting step in other dense factorization methods such as the Dense LU decomposition ($A = LU$) or the Dense Cholesky factorization ($A = U^TU$). As shown in Figure 6, the popular implementations of the parallel factorization methods of these operations[16, 29] break the matrix into 4 quadrants: a small upper left corner (A_{UL}), a tall skinny lower left corner (A_{LL}), a short and wide upper right corner (A_{UR}), and a large lower right corner (A_{LR}). The methods perform serial computation on the upper right corner and update the lower left and upper right quadrants of the matrix. Then a large parallel outer product of these two quadrants is performed to update the lower right corner. The algorithm continues by recursively factorizing the lower right quadrant. Hence the matrix elements in the lower right corner tend to be more heavily used and updated compared to the other parts. A purely blocked layout would induce a poor load balance since the most heavily used elements will be concentrated amongst a few processors. In order to alleviate this problem, a checkerboard layout[25] of processors is used so that the load is more evenly distributed across the processors. However, such a checkerboard layout dramatically increases the complexity of an implementation since the user is responsible for managing the mapping of these blocks to the processors. With our proposed extensions, the runtime system handles this distribution. What remains to be shown, is

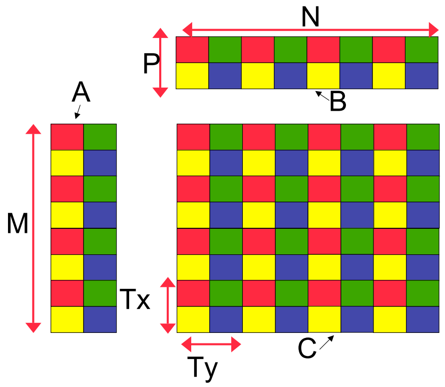

Figure 4. Matrix Multiply Diagram

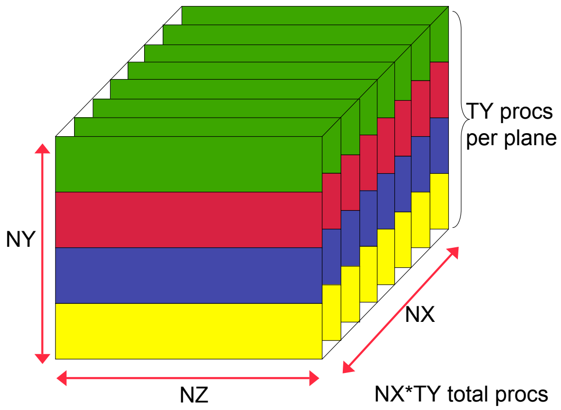

Figure 5. 3D FFT Diagram

```
1. shared [b][b] double A[M][P], B[P][N], C[M][N];
2. shared [b][b] double scratchA[b*Tx][b*Ty], scratchB[b*Tx][b*Ty];
3.  int myrow=MYTHREAD/Ty; int mycol=MYTHREAD%Ty;
4.  for(k=0; k<P; k+=b) {
5.   for(i=0; i<M; i+=Tx*b) {
6.    upc_stride_broadcast(scratchA<myrow,:>, &A[i+myrow*b][k], sizeof(double)*b*b, 0);
7.      for(j=0; j<N; j+=Ty*b) {
8.          upc_stride_broadcast(scratchB<:,mycol>, &B[k][j+mycol*b], sizeof(double)*b*b, 0);
9.           /* matmult*/
10.          dgemm((double*) &C[i+myrow*b][j+myrow*b],
11.                (double*) &scratchA[myrow*b][mycol*b],
12.                (double*) &scratchB[myrow*b][mycol*b], b, b, b);
13.      }
14.   }
15. }
```

Figure 7. UPC Code for Dense Matrix Multiply

how the collectives interface described in the previous section can be applied to this problem.

Figure 4 shows an example of the outer product used in most factorization methods. The two dense matrices A and B are multiplied together to get C. In the figure the pieces of the matrix are color coded by the processor that owns the piece of the matrix. We can compute a particular block, $C[i][j]$ by performing the operation: $C[i][j] = C[i][j] + A[i][k] \times B[k][j]$ for each of the k blocks in the P dimension. Notice that for all blocks in a given row i we need only broadcast the elements of $A[i][k]$ once and store into a temporary array. The subset of the processors that own row i has size $O(\sqrt{N})$, where N is the total number of processors. Next we need to perform a column broadcast of $B[k][j]$ into a separate scratch array. Again notice that column broadcasts occur over a set of $O(\sqrt{N})$ processors in the column dimension. By using calls to the collectives interface, the runtime system has much finer control on how the communication is coordinated compared to the use of many uncoordinated gets. This enables the algorithm to be scaled to large processor counts.

The UPC code for matrix multiplication is shown in Figure 7 while the Cholesky factorization example can be found in Figure 11 in Appendix A. Line 1 of Figure 7 declares the dense matrices with the specified blocksizes. The matrix is partitioned according to the mapping specified in Section 4. Notice that with one simple declaration that UPC offers, the entire matrix is load balanced in the optimum checkerboard pattern. Such a task in MPI is much more cumbersome since the matrix block to processor mapping has to be controlled by the application writer rather than the runtime system. We then allocate a set of scratch arrays in Line 2 that will be used for intermediate results. We iterate through the blocks of the matrix as one would do in a standard blocked implementation of the kernel. In addition, we replace the upc_memget()s in the original code (Figure 1) with broadcast collectives. Notice all the broadcasts in one dimension occur simultaneously and each processor is only responsible for specifying the portion of the data that it owns. As we will see in the Section 5, with the call to an optimized collective rather than using many upc_memgets()s the code in Figure 7 scales much better without significant changes in the complexity.

4.2 Three Dimensional Fourier Transform

Unlike the Dense Matrix Multiplication algorithm (and to a lesser extent, Dense Cholesky factorization), which is bound by the total amount of computation, a large parallel three dimensional Fast Fourier Transform routine is typically bound by the interconnect bandwidth. Given a rectangular prism, the three-dimenional FFT performs an FFT in each of the dimensions of the grid. Thus each point in the domain is involved in 3 different FFTs. When this problem is mapped across a two-dimensional processor grid, only one of these dimensions can be computed without communication. The FFT is also used in many areas of science such as molecular dynamics, computational fluid dynamics, image processing, signal processing, nanoscience, astrophysics, etc. While we focus our analysis on a 3D FFT, Agarwal et al.[2] show how any lower dimension FFT (say 1D) can be transformed into a 3D FFT. Our techniques can equally be applied to these lower dimension FFTs.

Figure 5 shows the mapping of a 3-dimensional domain onto a 2 dimensional processor grid. In this example, each processor owns $\frac{NY}{TY}$ rows of NZ complex elements each. Thus each of the NX planes is broken up into TY pieces and our 2D processor grid is $NX \times TY$. We first perform the FFT in the Z direction, which is completely local and requires no communication. In order to perform the FFTs in the Y direction each of the planes need to be rotated. Thus the group of TY threads that own each plane perform an exchange amongst each other. Once the data is in place we can easily perform the appropriate set of local FFTs. Finally we need to perform an exchange in the X dimension. Thus all the processors that own a common row across all the planes (e.g. all the processors that own the green rows) will perform an exchange. Finally we compute the last round of FFTs. Figure 12 in Appendix A shows the UPC code to implement these operations. The calls to `fft()` are calls to high-performance FFT libraries such as ESSL or FFTW which provide the appropriate interface and the required serial performance.

4.3 Observations

We use these benchmarks as a case study to explore the effectiveness of the interface and examine how three of the major hurdles to efficient and scalable parallel programming are addressed.

- **Data distribution and load balancing** In our examples, the user specifies high level properties of how the data should be layed our across processors. For example in the case of the matrix multiply and Cholesky factorization, the user is responsible for specifying the granularity of the checkerboard. In the case of the FFT, the user specifies the TY and NX dimensions to dictate how many processors are involved in each of the exchange rounds. However, notice that once the data distribution directives are given, access to the arrays is straight forward. Mapping the data distribution and array indices to the processors is left to the runtime.
- **Constructing an efficient and scalable communication schedule between the processors** After distributing the problem across the processors it is important that these processors work together and communicate as efficiently as possible. Therefore we wrote the three benchmarks with collectives rather than manually controlling the communication. This allows us to let the runtime layer to handle the communication more efficiently by using network features, such as Active Messages, that were unavailable to the UPC programmer. By passing the responsibility of tuning the communication schedule to the runtime layer through a clean interface, the user absolves himself from having to worry about the painstaking task of tuning communication. The wide body of literature on the subject of collective tuning for a variety of architectures indicates that the process is a non-trivial problem and very dependent on the specifics of the interconnect.
- **Efficient serial computational performance** Once the data has effectively been distributed and communicated the last piece that remains is to perform the serial computation. Serial tuning for many of the popular serial computational kernels have been well studied and serial libraries such as ESSL, FFTW, ATLAS, and OSKI[39] have been well tuned (either by hand or automatically) on many architectures. Any parallel programming language must allow for easy ways to leverage this work to realize optimal serial performance. In our implementations, the data movement is handled in UPC while the computation is handled through optimized serial libraries.

Software libraries, such as PETSc [7], alleviate some of the pain of data distribution and coordinating communication by providing an extensive API to handle these operations. However, introducing these features at the language level allows for more expressivity than a library can provide. In order to minimize the complexity of the interface a library writer must try to keep the interface very simple by making an educated guess about which data layouts to support and which data layouts to omit. However, when these data layouts are incorporated into a language, a simple grammar can lead to a much more rich set of data layouts that are infeasible to efficiently provide at the library level.

By allowing the user to only specify high level language directives regarding the data distribution, letting the runtime handle the details of the communication and allowing the user to use preexisting libraries we can dramatically reduce the number of lines required to program common and important kernels. We would like to show by comparison the equivalent programs written in MPI or in UPC without our extensions. However, it will be impossible to fit them in the alloted space.

5. Evaluation

We validated the proposed UPC language extensions and collectives on the IBM Blue Gene supercomputer located at the IBM TJ Watson Research Center. We implemented the codes discussed in Section 4 and ran scaling experiments from 32 processors to 16384 processors.

Compiler: we used the the IBM UPC compiler for our work. The compiler supports a number of SMP, distributed and hybrid architectures including Blue Gene/L. The compiler includes a runtime, which translates UPC remote memory accesses to messages using the Blue Gene Distributed Computing Messaging Framework (DCMF) API.

The IBM UPC compiler's front-end does not (yet) support the language extensions described in this paper. However, an experimental version of our UPC runtime supports all array layouts, data distribution primitives and collective APIs described here. We hand-coded our benchmarks to work with this experimental runtime, trying to stay as faithful as we could to code that an extended compiler frontend would have generated.

Sequential performance: For sequential performance we used the ESSL library provided by Blue Gene/L without any changes or tuning. We clocked the performance of ESSL's generic `dgemm` routine between 1.8 and 2.1 GFlops; smaller matrices resulted in somewhat lower sequential performance. For sequential FFT we measured about 300 MFlops/node, also using ESSL.

Virtual Node Mode: For the compute bound algorithms (Cholesky factorization and matrix multiplication) we booted the Blue Gene system in virtual node mode. Thus the largest configuration we used was 16 racks with 512 MBytes of memory per node.

For parallel 3D FFT, which – on the Blue Gene platform – is communication bound, we chose coprocessor mode. This allowed us to double the amount of per-processor memory, resulting in longer messages and, consequently, in lower relative messaging overheads. Also, since half as many processors use the same network in coprocessor mode, bisection bandwidth is double on a per-processor basis.

Optimized collectives: The UPC collectives we propose do not map exactly on MPI collectives. Thus we were unable to use the optimized MPI collective suite from the Blue Gene MPI library: we implemented the broadcast and exchange routines ourselves. MPI collective implementations on Blue Gene are notoriously difficult and time consuming [5]. To expedite the measurement process we

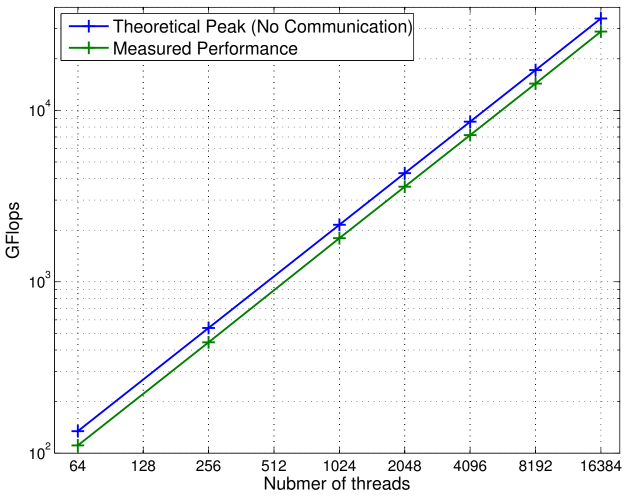

Figure 8. Matrix Multiplication

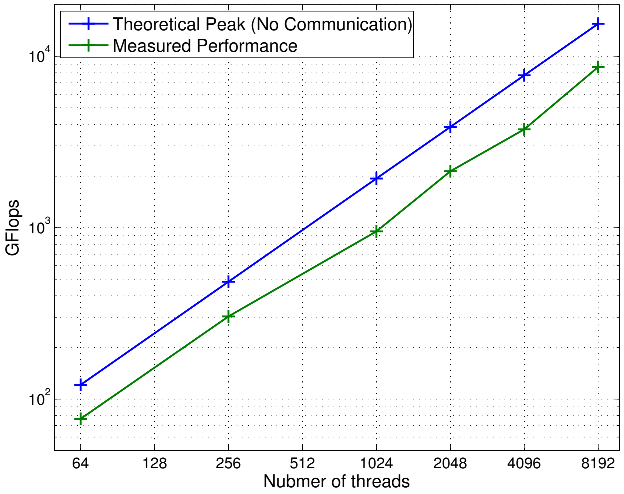

Figure 9. Cholesky Factorization

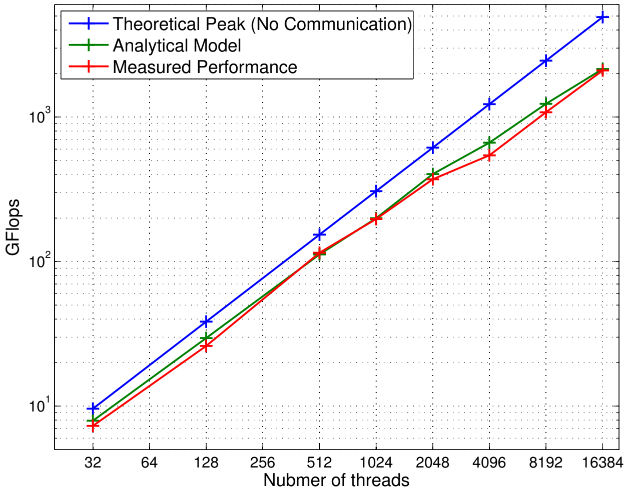

Figure 10. 3D FFT

settled for somewhat lower collective performance, as long as it was enough to make the point of our paper.

Blue Gene mapping: In order to further improve the performance of the UPC collectives we took great care to map the Cartesian processor grid into the Blue Gene/L's 3D torus network.

Weak scaling: We ran all three of our algorithms in weak scaling mode. We designed our experiments for a constant per-processor memory load, adjusting the global problem size as the total number of processors (and hence total available memory) increased.

Scaling results. Figures 8, 10 and 9 show the scaling results we obtained. On each graph we show the computed peak as well as the actual measured performance.

Matrix multiplication: since the matrix multiply and Cholesky algorithms are compute bound, we computed peak performance as the product of the sequential matrix multiply performance and the number of processors. For matrix multiplication we used the best measured sequential performance, 2.1 GFlops, as the baseline. The blocking factor for these measurements was held constant at $B = 500$, while the problem size was always chosen to fill up all the usable memory (approx. 240 MBytes) in each processor.

We mapped the two-dimensional matrices onto the 4-dimensional Blue Gene torus network (we consider communication between Blue Gene co-processors on the same node to be the 4th dimension). We achieved this by linearizing pairs of torus dimensions into single virtual dimensions and mapping the matrix on these. Table 1 shows the way we performed this mapping.

Figure 8 shows that the actual measured performance closely mirrors the theoretical peak, but is consistently 15% lower for every machine size. The discrepancy is due to communication overhead. With a fixed blocking factor of B the compute/communicate ratio of the algorithm is constant: B^2 doubles communicated for every $2 \times B^3$ floating-point operations. Therefore, the amount of performance loss with respect to the theoretical peak is a function of the achieved broadcast bandwidth. On Blue Gene this loss can be reduced by deploying a better broadcast algorithm (in our estimation we are a factor of 3 away from what MPI's own broadcast can deliver). The gap between theoretical and measured performance can also be manipulated by varying the blocking factor.

Cholesky factorization: Figure 9 shows a scenario similar to the matrix multiply algorithm, but with two important differences. First, the discrepancy between theoretical and measured performance is larger. This is due to the larger relative communication overhead. Unlike matrix multiplication, where block matrices are always broadcast along rows or columns of the Cartesian $\{Tx, Ty\}$ processor layout, the Cholesky factorization code causes block matrices to be transmitted across processor rows/columns, further diminishing broadcast performance.

The second noteworthy fact about Figure 9 is that performance is less predictable. This is due to load balance issues. While the processor tile we chose is rectangular, the algorithm itself is triangular: this causes some of the processors in the rectangular distribution to be idle at times when other processors are performing matrix updates, leading to overall performance degradation. Choosing a suitable blocking factor is a more complex issue for the Cholesky factorization algorithm. Too small a blocking factor leads to inadequate sequential performance of the library routines we are calling; too large a blockig factor destroys the load balance.

Note that an obscure bug in the UPC runtime caused us not to have publishable 16 rack numbers for matrix multiplication and 8 and 16 rack numbers for Cholesky factorization. We will try to correct this lack by the time of the final paper.

3D FFT: Figure 10 shows the computed and measured performance of the FFT algorithm. An accurate performance model of FFT is somewhat more difficult to obtain than in the case of matrix multiply or Cholesky factorization. Since FFT is communication bound, the computed peak performance has to take into account both computation and communication.

3D FFT has three computation phases and two inter-node communication phases. We model the computation phases using the usual $O(n \times log(n))$ FFT formula. Communication is modeled based on the processor mapping in Figure 5. Every processor exchanges the entirety of its memory core with its neighbors in each communication phase, and half of all messages in this exchange pass through the bisection bandwidth of the network. Given the network's topology in the table, and a base bandwidth of 154 MBytes/s of a Blue Gene torus link, we can estimate bisection bandwidths for each communication phase, and therefore arrive at a reasonably accurate total execution time for 3D FFT.

The ratio of available compute power and bisection bandwidth changes on the Blue Gene machine as the machine itself grows. A further complication is caused by the specifics of the Blue Gene/L installed at IBM TJ Watson, which causes bisection bandwidth to increase unevenly as the machine is scaled up (e.g. the 4192 node machine we ran on had the same bisection bandwidth as the 2048 node machine). Therefore the FFT scaling curve is not as smooth as the matrix multiply curve. This limitation is inherent to the algorithm we chose and to the machine we ran it on. Figure 10 shows that the system's actual behavior follows our theoretical model reasonably accurately on up to 16,384 processors.

Interpreting the results: The measurements shown here are proof that the language extensions proposed in this paper can lead to competitive performance and scaling as well as compact representations for the codes shown in this section. Better collective infrastructure could potentially lead to even better results. It remains to be shown that the results we obtained here can be generalized to other codes.

CPUs	torus				UPC array mapping	
	x	y	z	t	dim. order	distribution
64	4	4	4	2	XT, YZ	8 x 16
1k	8	8	8	2	XT, YZ	16 x 64
2k	8	8	16	2	XT, YZ	16 x 128
4k	8	16	16	2	XT, YZ	16 x 256
8k	8	32	16	2	XT, YZ	16 x 512

Table 1. Mapping two-dimensional tiled arrays onto the 4 dimensional Blue Gene torus network.

6. Related Work

Due to its complexity, there have been many projects over the years that have aimed to improved the productivity of parallel programmers. One of the major directions has been to provide important and computational tasks as sets of libraries that run over MPI[28]. Popular examples include SCLAPACK [20] and PBLAS[16]. As mentioned earlier large software engineering efforts such as PETSc provide a common framework that encompass many popular tools. In practice these distributions have been successful.

In order to provide a richer abstraction and expression of the higher level semantics there have been many language efforts to improve productivity. Popular examples of these languages include UPC, Co-Array Fortran[31], Titanium[24], ZPL[15], HPF[23], Chapel[1], Fortress[3], X10[41], and many others.

While each of the languages has their own corresponding performance and productivity studies, we highlight those that rely on UPC since they are more directly related. El-Ghazawi et al. [18] demonstrate the potential of UPC as a viable programming language and show their potential performance advantages. Bell et al. [10] show how the performance advantages of one-sided communication models and overlap can be applied to improve performance of bandwidth limited problems such as 3DFFTs. Barton et al. [8] further demonstrate how the shared memory programming model found in UPC is a good fit for large distributed memory machines. Coarfa et al. [17] evaluate the effectiveness of these global address space languages and highlight their limitations.

In addition to the language level issues there have been many previous projects that have addressed tuning building optimum communication schedules for collectives. Some are focused on tuning collectives for a particular machine[34, 5] and others are focused on tuning the collectives so that they work well on a wide variety of machines[13, 6]. From the body of literature it is clear that the tuning space is indeed large[37]. Seidel et al. have also proposed an alternative model for collectives in UPC [35].

7. Conclusions

Programmer productivity is becoming ever more important as the scale of machines continues to grow and parallel architectures becomes more common place. The most important challenge facing the programmers of these machines is application scalability, because the way the applications scale determines how effectively the machines are used. To address this challenge, we need to look at both how parallel languages allow the user to express important problems and how well parallel language implementations map onto machines.

In this paper we analyzed how a popular parallel programming language, UPC, handles this task. We started by implementing from scratch three kernels used in a number of scientific applications: matrix multiplication, Cholesky Factorization, and 3D FFT. We identified three major areas which affect parallel performance: data distribution and load balancing, scalable communication, and efficient serial performance. For each of these we propose minimal extensions to the UPC language to make more effective use of existing libraries, such as serial, optimized high performance libraries, and optimized collective communication libraries. We demonstrate the effectiveness of these extensions by running the benchmarks on 32 to 16384 processors on Blue Gene/L. Our results prove that codes do not have to be thousands of lines to achieve best class performance. Our 15 line matrix multiplication kernel obtains 28.8 TFlops on 16K processors (63% of peak, 84% of serial ESSL), and Cholesky at 25 lines obtains 8.6 TFlops on 8K processors (37% of peak and 56% of serial performance). The 17 lines 3D FFT obtains 2.196 TFlops on 16K processors, on par with the best published Blue Gene/L number of 2.178 TFlops for 1D FFT on 64K processors.

References

[1] The cascade high productivity language. *hips*, 00:52–60, 2004.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d fft. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0α edition, Sept. 2006.

[4] G. Almási, C. Archer, J. G. C. nos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. SteinmacherBurow, W. Gropp, and B. Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, March/May 2005. Available at http://www.research.ibm.com/journal/rd49-23.html.

[5] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of mpi collective communication on bluegene/l systems. In *ICS*

'05: Proceedings of the 19th annual international conference on Supercomputing, pages 253–262, New York, NY, USA, 2005. ACM Press.

[6] P. Balaji, D. Buntinas, S. Balay, B. Smith, R. Thakur, and W. Gropp. Nonuniformly communicating noncontiguous data: A case study with petsc and mpi. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[7] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[8] C. Barton, C. Caşcaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterjee, and J. N. Amaral. Shared memory programming for large scale machines. In *Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, 2006.

[9] C. Barton, C. Cascaval, G. Almasi, R. Garg, and J. N. Amaral. Multidimensional blocking in UPC. Technical Report RC24305, IBM, July 2007.

[10] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *The 20th Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[11] M. J. Berger. *Adaptive mesh refinement for hyperbolic partial differential equations*. PhD thesis, 1982.

[12] BLAS Home Page. http://www.netlib.org/blas/.

[13] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. W. y. Efficient algorithms for all-to-all communications in multiport message-passing systems. 1997.

[14] L. E. Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montanat State University, 1969.

[15] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.

[16] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, pages 107–114, London, UK, 1996. Springer-Verlag.

[17] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM Press.

[18] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[19] ESSL User Guide. http://www-03.ibm.com/systems/p/software/essl.html.

[20] L. S. B. et al. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, page 15 (electronic), Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[21] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[22] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-burow, T. Takken, and P. Vranas. Overview of the BlueGene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[23] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[24] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[25] HPL Algorithm description. http://www.netlib.org/benchmark/hpl/algorithm.html.

[26] Intel Math Kernel Library Reference Manual. http://www.intel.com/software/products/mkl/techtopics/mklman52.pdf.

[27] T. MathWorks. Using matlab, 1997.

[28] Message Passing Interface. http://www.mpi-forum.org/docs/docs.html.

[29] J. C. Nash. *"The Cholesky Decomposition." In Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, chapter 7, pages 84–93. Bristol, England: Adam Hilger, 2nd edition, 1990.

[30] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[31] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[32] OpenMP. Simple, portable, scalable SMP programming. http://www.openmp.org/, 2000.

[33] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[34] Y. Qian and A. Afsahi. Efficient rdma-based multi-port collectives on multi-rail qsnetii clusters. In *The 6th Workshop on Communication Architecture for Clusters (CAC 2006), In Proceedin gs of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[35] Z. Ryne and S. Seidel. A specification of the extensions to the collective operations of unified parallel c. Technical Report Technical Report 05-08, Michigan Technological University, Department of Computer Science, 2005.

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

[37] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Performance modeling for self adapting collective communications for mpi. In *LACSI Symposium*, 2001.

[38] R. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *TR-95-13, Department of Computer Sciences, University of Texas*, 1995.

[39] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[40] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[41] The X10 programming language. http://x10.sourceforge.net, 2004.

[42] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.

A. Example UPC Code

```
shared [B][B] double A[M][M];
shared [B][B] double scratchA[TX*B][TY*B];
shared [B][B] double scratchB[TX*B][TY*B];
int i, j, k;
int myrow = MYTHREAD/TY; int mycol = MYTHREAD%TY;

for(k=0; k<M; k+=B) {
  /* dense cholesky on upper left block */
  if(upc_threadof(&A[k][k]) == MYTHREAD)
    dpotrf("Upper Triangular", &A[k][k], B, B);

   /* triangular solve across top row of matrix */
  int proc_row = upc_threadof(&A[k][k])/TY;
  upc_stride_broadcast(scratchA<proc_row, :>, &A[k][k], sizeof(double)*B*B, 0);
   for(j=k+B; j<M; j+=B) {
    if(upc_threadof(&A[k][j]) == MYTHREAD)
       dtrsm(&A[k][j], &scratchA[myrow*B][mycol*B], &A[k][j]);

   /*update (outer product on upper triangular part)*/
   for(i=k+B; i<M; i+=TX*B) {
    for(ti=i; ti<i+(TX*B); ti++)
         upc_stride_broadcast(scratchA<(ti/B)%TX,:>, &A[k][ti],  sizeof(double)*B*B, 0);
    for(j=i; j<M; j+=TY*B) {
       for(tj=j; tj<j+(TY*B); tj++)
           upc_stride_broadcast(scratchB<:,(tj/B)%TY>, &A[k][tj], sizeof(double)*B*B, 0);
        dgemm(&A[i][j],  &scratchA[myrow*B][mycol*B], &scratchB[myrow*B][mycol*B], B, B, B);
     }
 }
```

Figure 11. UPC Code for Dense Cholesky Factorization

```
void fft(complex_t *out, complex_t *in, int len, int howmany,
         int instride, int indist, int outstride, int outdist);
void main(int argc, char **argv) {
 shared [1][NY/TY][] complex_t A[NX][NY][NZ], B[NX][NY][NZ];
 int myplane = MYTHREAD/TY; int myrow = MYTHREAD%TY;
 complex_t *myA = (complex_t*) &A[myplane][myrow*(NY/TY)][0];
 complex_t *myB = (complex_t*) &B[myplane][myrow*(NY/TY)][0];
 initialize_input_array(A);
 upc_barrier;
 fft(myB, myA, NZ, NY/TY, 1, NZ, NY/TY, 1);
 upc_stride_exchange(A<myplane,:,0>, B<myplane,:,0>, sizeof(complex_t)*(NY*NZ)/(TY*TY), 0);
 local_transpose(myB, myA, NX, NY, NZ, Ty);
 fft(myA, myB, NY, NZ/TY, 1, NY, NZ/TY, 1);
 upc_stride_exchange(B<:,myrow,0>, A<:,myrow,0>, sizeof(complex_t)*(NZ/NX)*(NY/TY), 0);
 fft(myA, myB, NX, (NZ/NX)*(NY/TY), (NZ/NX)*(NY/TY), 1, 1, NX);
 upc_barrier;
}
```

Figure 12. UPC Code for Parallel 3D FFT