UPC Implementation of the Sparse Triangular Solve and NAS FT

Christian Bell Rajesh Nishtala
May 2004: CS267 Final Project
Computer Science Division
University of California, Berkeley
{csbell,rajeshn}@cs.berkeley.edu

1 Introduction

One of the main trends in modern high performance computing is the use of clusters of SMPs instead of the large monolithic machines to solve scientific codes. Clusters have recently received a lot of traction in the community through the availability of low-cost high performance interconnects. These interconnects allow not only high performance, but good scalability as well. One of the main drawbacks to the use of clusters is their difficulty to program because of the distributed memory model. The Message Passing Interface (MPI) is the current standard for programming for these massively parallel systems. However, the two-sided nature of MPI severely restricts the programmability of many applications as it constrains programmers to adopt a bulk synchronous programming approach. Partitioned Global Address Space (PGAS) languages address this issue by simulating one large memory space over distributed memory. One of the newer languages in this space is called Unified Parallel C [6](UPC). We focus on using Berkeley's implementation of UPC [5] to analyze the strengths and weaknesses of the languages in writing parallel applications.

This paper focuses on the analysis and development of two programs written in UPC: the NAS FT benchmark[3] and the Sparse Triangular Solve (SpTS) kernel. These two programs radically differ in their communication and computation characteristics. The NAS FT benchmark is considered a regular problem since its access patterns and operations are very predictable and all communication operates synchronously across participating threads. The SpTS kernel, however, greatly depends on the input parameters, such as the nonzero pattern of the matrix. The access patterns are also very irregular, which limits the scalability of the kernel and poses significant parallelizing challenges. In addition to using the one-sided paradigm offered by UPC, the two programs also leverage the new non blocking point to point operations that were recently proposed as an extension to the UPC language [4].

The organization of the rest of the paper is as follows. In Section 2 we present a detailed description of the UPC language and how its language features may be of interest to both FT and SpTS. We present a novel design for the SpTS in Section 3 that leverages some of these features. In Section 4 we analyze the NAS FT benchmark. We compare the two benchmarks and suggest language constructs that would improve the performance or programmability of these applications in Section 5. We conclude with related and future work in Section 6.

2 UPC as a target language

UPC (Unified Parallel C) is a parallel extension of the C programming language aimed at supporting high performance scientific applications. The language adopts the SPMD programming model, such that every thread runs the same program but keeps its own private local data. In addition to each thread's private address space, UPC provides a shared memory area to facilitate implicit communication amongst threads. Programmers can create shared objects through the use of the `shared` type qualifier or the dynamic shared memory allocation library functions. While a private object may only be accessed by its owner thread, all threads can read or write objects in the shared address space. Because the shared memory space is logically divided among all threads, from a single thread's perspective the shared space can be further divided into a local shared memory and a remote one. Data located in a thread's local portion of the shared space are said to have "affinity" with the thread, and compilers can utilize affinity information to exploit data locality in applications and help reduce communication overhead.

UPC gives users direct control over shared data placement through distributed arrays. When creating a shared array, programmers specify a block size in addition to the dimension and element type, and the system uses this value to distribute the array elements block by block in a round-robin fashion over all threads. For example, a declaration of `shared [2] int ar[10]` tells the compiler to allocate the first two elements of `ar` on thread 0, the next two on thread 1, and so on. If the block size is omitted the value defaults to one (cyclic layout), while a layout of `[]` or `[0]` indicates indefinite block size, i.e., that the entire array should be allocated on a single thread. A pointer-to-shared thus needs three logical fields to fully represent the address of a shared object: `address, thread_id,` and `phase`. The `thread_id` indicates the thread that the target has affinity to, the `address` field stores some representation of the object's "local" address on the thread, while the `phase` field gives the offset of the target within the current block. Other notable UPC features include a `upc_forall` parallel loop, block transfer library functions, synchronization constructs, and flexible language-level control of the memory consistency model; consult the UPC language specification for more details [6].

2.1 UPC shared data placement and distribution

From the various possible distributed array representations, we opted not to make use of the most advanced features such as blocked and/or cyclic distribution of arrays. For one, the larger NAS classes for FT lead to many millions of complex elements to solve, that far exceed the largest supported phase of 2^{20} elements. Moreover, we did not want to incur the extra overhead of blocked shared pointers when accessing elements of the shared array with no interesting gain in programmability. In both of their core execution stages, both FT and SpTS do not benefit in programmability from a richer data distribution as both make exclusive use of the local portion of their shared UPC heap. In terms of communication, FT is limited to a core all-to-all step which exchanges portions of the 3D FFT cube while SpTS already indexes into thread-local distributed data structures.

At initialization, both UPC programs proceed to building a directory of per-thread structures, a C struct with thread local information such as pointers to locally allocated shared memory as well as indexing and miscellaneous metadata. According to the application, information in this structure can be used to statically partition a problem or to initialize enough indexing structures to allow threads to agree on a mechanism to dynamically partition a problem. This approach allows asymmetric allocation of data and initialization structures across threads and relaxes the requirement that each allocated object be symmetric across the entire job. This particular point should be contrasted to the Cray *shmem* distributed programming model which assumes SIMD program execution and enforces symmetric data placement of objects across processing elements. While UPC can still lean on this useful feature through the global portion of its shared heap, irregular applications definitely benefit from non-collective allocation of data structures satisfied by the local portion of its shared heap. Although FT is a regular application and our initial approach to SpTS is irregular, we were encouraged by the idea of knowing where each thread allocates its data and the indexing data it has been assigned for both debugging and upward compatibility for future revisions to the code (especially in SpTS).

Once each per-thread structure is distributed to all participating threads through a global exchange operation, information located in the directory allows other threads to read and write into any thread's shared heap in a synchronization-free manner. This process allows true one-sided operation, regardless of the semantics and data access patterns of the application. The distinction is important, especially in the context of relying on the term *one-sided* as a particular language or library feature and is further explored in subsection 4.1.

In summary, none of the benchmarks make use of the most expressive UPC data placement constructs and make exclusive use of phase less pointers in the critical path.

3 Sparse Triangular Solve

We next consider the solution of sparse lower triangular system $Lx = b$ for a single dense vector x, given a sparse lower triangular sparse matrix L and the dense vector b. We will refer to x as the solution vector and b as the *right hand side* (or RHS) vector. These sparse matrices frequently arise in sparse Cholesksy and LU factorization.

Our solution leverages the UPC language by creating a producer/consumer relationship between the different threads in order to solve the solution. This relationship allows the threads to act more asynchronously than in the other MPI implementations.

```
for r=1:N,
  temp = b(r);
  for c=1:r-1
      temp = temp - L(r,c)*x(c);
  end
  x(r) = temp/L(r,r);
end
```

Figure 1: **Dense Triangular Solve Code**. This code segment shows the inner loops needed to solve for x given L and b

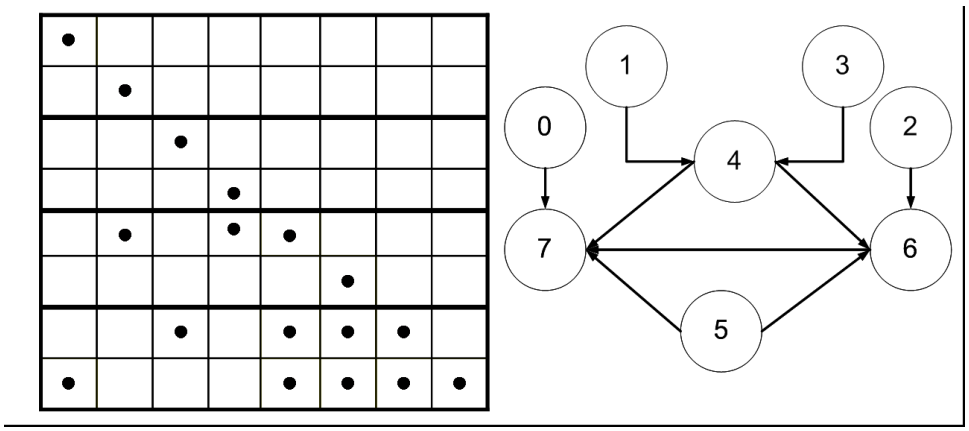

Figure 2: **Example Sparse Matrix and Dependence Graph**. On the left (a) is an example sparse 8x8 sparse matrix. On the right (b) is the corresponding dependency graph.

3.1 Problem Formulation

Figure 1 shows the serial code needed to solve a dense NxN lower triangular matrix. The extension to the upper triangular case is straight forward. For simplification, the code is written in Matlab notation.

In order to calculate the value for x_r, all the values of x above it must have been solved. This is what we call a *dependency* for x_r. Since L is a sparse matrix, x_r only depends on the columns in row r in which T is non zero. Thus the number of dependencies per row is only known at run time because it changes from matrix to matrix.

3.2 Data Structures

In this section we describe the data structures that were used to enable a fast implementation of the produce/consumer relationship described above. We create a dependence tree that is based on the relationship of the elements in the solution vector described above. The dependence tree is defined as follows: there is an edge from vertex i to vertex j if the result of x_i influences the value of x_j. A simple way to check this condition is to look at row j in the matrix L. For each column i with a non zero in row j, we draw an edge from vertex i to vertex j. Figure 2b shows the resulting graph from Figure 2a.

With this framework in mind, we describe how this tree can be constructed in parallel, independently of how the matrix is laid out amongst the different threads. The initial implementation of our results leveraged a blocked layout. In this layout each thread gets N/τ contiguous rows where N is the dimension of the matrix while τ is the total number of threads involved in the computation. Leveraging an existing code base by Vuduc [17], each of the threads loads their rows in a compressed sparse row (CSR) format [13].

Each of the threads thus has a sub-matrix of the original and will be responsible for producing the corresponding parts solution vector x for every row it owns. In addition, they will be responsible for singling the other threads with when this data is available. However in order to solve a row, all the dependencies for that row must have been satisfied. As an optimization, each thread only signals other threads that are waiting on a dependency rather than doing a broadcast. To enable this, each thread must advertise its dependencies to everyone so that they know who to signal. This operation would be greatly aided with the

presence of an all-to-all construct. To make this process easier, each thread converts its portion of the matrix from the CSR format to a compressed sparse column format (CSC). The CSR format is useful for extracting the elements along the diagonal as well as counting the number of non zeros per row, which is needed to aid in termination of the algorithm. The CSC format is useful for receiving signals and processing dependencies. We thus get all the information that is easy to extract out of the CSR format and then perform the rest of the calculation with the CSC format. In order to save memory, only one copy of the matrix is stored (except during the actual conversion).

Each thread must analyze its sub matrix and build a dependency subgraph. Because of our static mapping functions, every thread knows what rows the other threads will own. Once the sub graph is built at each of threads, they can then analyze all the dependencies to find out which thread will eventually satisfy that dependency[1]. The thread then allocates enough space in shared memory to which it has affinity in order to receive these dependencies. We will describe the consistency issues associated with this later in the section. Each thread then multicasts its external dependencies to all the other threads that will be able to solve its dependencies giving them information about what values of x_j that it needs along with the location in which to place the results. Each of the threads also creates a mapping between the locations along with the column index that corresponds to that dependency. This enables the O(1) lookup while limiting the extra space that needs to be allocated.

Each thread then creates R linked lists, where $R = N/\tau$, that holds information on who needs to be signaled with the information that row r has finished. It then fills out this list with the dependency lists that it got from all the other threads. Once this list has been created the actual solve can commence. Again because of the R linked lists are stored in an array we have an O(1) lookup time to find out who to signal once row r is done.

3.3 The Solve

Once the data structures have been built the solve portion is relatively straight forward. Since we know the external dependency list, we simply poll on these lists until other threads have written to it. Once we get a signal indicating a particular value of x_j is ready, we iterate over all the rows that depend on the value x_j. For each row, we update a temporary version of the solution vector with the value from this point. If we notice that the number of dependencies for that specific row has r gone to 0, then the value of x_r is valid. The thread then indexes into the array of linked lists to find out which threads to signal the completion event. For each thread in the list it writes the value of x_r into the appropriate location for the remote threads. The remote address calculation is again an O(1) step because of the advertisement phase of the setup. Since the other threads are polling on these lists for the values, they will automatically see the value and continue with its calculation. If no values are ready, then it will continue to poll. We also keep a counter indicating how many rows are left that need to be solved. Once this row counter drops to 0 the thread can simply just wait for the others to finish.

By induction we can prove that this solution terminates. We only present a high level sketch of the proof here. If we have a lower triangular matrix, then row 0 will always terminate in the first round since there are, by definition, no dependencies. Thus if we assume row j terminates then we can show that row $j+1$ terminates. If row j finishes then all the dependencies for row $j+1$ must have been solved and therefore row $j+1$ will also terminate because L is lower triangular.

3.4 Signaling

In the previous section we mentioned the notion of signaling to indicate when the value is ready. One of the traditional problems in setting up systems to use a produce/consumer paradigm is the overhead associated with synchronizing access to shared data structures. This problem is mitigated by the fact that only one thread will ever perform the write into a particular memory address, and once this write happens it will never change, i.e. all subsequent operations will be reads. Polling for dependencies can either be based on a pull model or a push model. In the former, producers would post the value that is ready locally and expect dependent threads to pull out the resulting data value. This approach implies that all threads must poll on remote data values, which seriously stresses the scalability of the network for large thread configurations and also increases the latency between discovery of the new value and when it can be used. We opted for a push

[1]In order to avoid the difficulties in creating a local and global dependency list, a thread will include itself in the signaling list if it can satisfy dependencies for its own rows. This happens if a thread owns both rows i and j and there is a dependence between x_i and x_j

model, where threads know of other thread's dependencies and put the resulting value at a location where the dependent thread can read it. For RDMA-based networks, this allows polling processors to spin on an in-cache value and essentially wait for the resulting "put" to invalidate the cached contents. This leads us to the following useful property. We first initialize our dependency vector on which we poll. If the values in that vector change from this initializer, then we know that a valid write has occurred and thus we can safely read it. The initializer that we chose to use was Not A Number (NaN). According to the IEEE 754 floating point standard [11], the NaN can be used not only to signal incorrect calculations but also as an initializer. We can thus use the *isnan* functions to enable the polling operations. Since there is only one value that we set, we can be guaranteed that the operation is atomic. Thus we do not need any slow synchronization mechanisms to ensure correctness.

3.5 Experiences with Triangular Solve and UPC

Our implementation differs in the key aspect that it relies exclusively on one-sided communications enabled by the global address space paradigm in UPC. All the other implementations in MPI require some form of handshaking in the critical path, and this process is essential to solving the problem. The fundamental problem with this approach is that each thread not only has to keep track of where its state in the calculation but the states of all the others as well. However in the one-sided model, the thread only needs to keep track of its location in the calculation and make sure it provides for others when dependencies are found. As such, individual threads are a lot less tied to the performance of the other threads, however the thread that has the most computation is still the bottleneck.

In addition to the simpler one-sided model, we are able to leverage the non blocking operations in UPC since the dependencies can be signaled with *fire and forget* like semantics.

4 NAS FT

The next benchmark we analyze is the NAS FT benchmark [3]. This program calculates a specific partial differential equation (PDE) using a forward and inverse FT. The main bottleneck to parallel performance is the 3 dimensional Fourier Transform. In this section we only analyze the aspects of the benchmark that affect the parallelism. We refer the reader to the full description of the benchmark for a more detailed explanation.

$$F_{q,r,s}(u) = \sum_{l=0}^{N_z-1} \sum_{k=0}^{N_y-1} \sum_{j=0}^{N_x-1} u_{j,k,l} e^{\frac{-2\pi ijq}{N_x}} e^{\frac{-2\pi ikr}{N_y}} e^{\frac{-2\pi ils}{N_z}} \quad (1)$$

Equation (1) shows the mathematical operations needed to perform the forward Fourier transform on the function u. The inverse transform is identical except the signs on the exponents are reversed. One of the important things to notice about this operation is that the output at a particular point in the output domain is influenced by *all* the other points in the input domain. The use of the Fast Fourier Transform significantly reduces the computation overhead, however the problem still remains that all the data must be transposed so that every thread can account for the influence of all the other data points. One of the ways to do this is to perform an in place 1D FFT on each of the rows in the N_x direction. Then perform in place 1D FFTs in each of the columns of the N_y dimension. Lastly one would want to perform the 1D FFT in each of the vectors in the N_z direction. This order of operations would mimic the ordering that the naive loops perform, however it would be very slow.

In order to parallelize the problem, we break the domain up into slabs. We define a slab to be N_z/τ planes. Each of these planes is $N_x \times N_y$ in dimension. Each of the threads computes a 2D FFT on each of the planes of the slab. Once we finish the computation of a plane it is split up into $\frac{N_x \times N_y}{\tau}$ pieces, where τ is the number of threads involved in the computation. Each of these pieces then gets sent to the appropriate thread. This step is our main *all-to-all*. As an optimization, each thread starts the all-to-all communication step at the thread immediately succeeding it in rank. This allows a balanced distribution of communication. Figure 3 shows a graphical representation of the operation that needs to be performed. In addition, the all-to-all communication step is done using non-blocking communication. This allows the computation of the 2D FFTs to be overlapped with the communication. After the 2D and communication step has been performed each of the threads then performs the required 1D FFTs and then re-does the all-to-all step with out the computation described above to put the values back in their original location to make it appear that the FFT was done in-place.

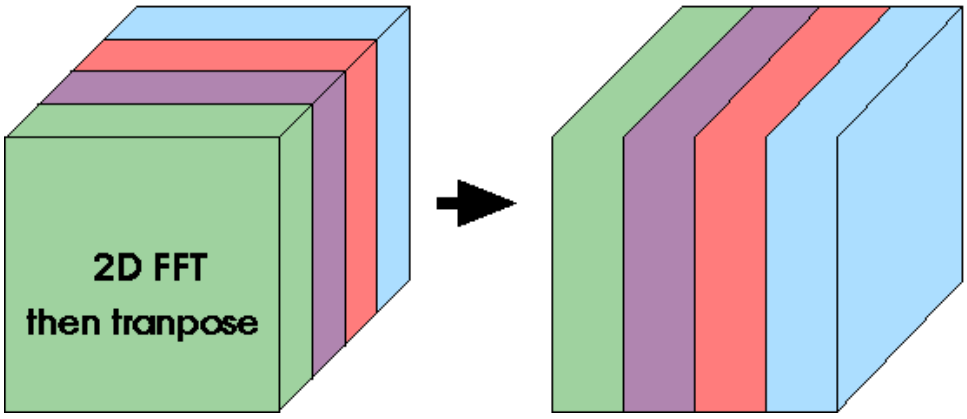

Figure 3: **Transpose Operation for 3D FFT**. This picture shows how the 3D prism needs to be rotated after the 2D FFTs across in order to perform the final 1D FFT. This rotation is the largest component of communication in this algorithm

Thus during the computation of the 2D FFTs there are three simultaneous operations at each thread. The first is the actual 2D FFT for the ith plane. The second is the distribution of the $(i-1)$th plane. Lastly each of the threads is receiving sections from computed planes from the other threads. If the network is RDMA capable, the third component has zero cost and is a result of using zero-copy remote DMA puts in the third component. If cost of computing a 2D FFT is larger than that of communicating the sections of each plane, the communication cost is completely hidden behind the 2D FFT. Once the 2D FFT and communication is done, the 1D FFT is computed and the data is sent back to where the 2D data came from, which effectively concludes a 3D FFT.

Once the 3D FFT has been communicated and performed in place, the remainder of the solution can be computed on the planes having affinity to each thread. The performance analysis in Section 5 shows the effect and net improvement of using non-blocking operations during the 2D Communication phase.

4.1 UPC compared to MPI

In UPC, contrary to MPI (and even MPI-2 one-sided), threads do not need to enter specific synchronization phases when issuing one sided operations, as phases are enforced exclusively by the application programmer to maintain data dependencies in the forms of barriers or finer methods of synchronization. For example, our FT integrates two examples of practical truly one-sided operations. The first is the interleaving of 2D FFT plane calculations with the communication of portions of the previously computed plane to other threads can be done non-collectively and does not require preposting of communication buffers. Furthermore, in the last 1D FFT, each thread locally transposes and computes its 1D FFT before retransposing the data and subsequently communicates the new data to the originating threads. This last communication step is completed after each thread completes the 1D FFT and causes each thread to do the inverse of the communication step completed after the first 2D FFT – each thread uses a truly one-sided get operation to recover the result of the 1D FFT from each thread. Although there is no computation integrated in this last communication step, each three proceeds into its FFT verification step independently and as such, is free from further synchronizing with any other thread until the 2D FFT of the next iteration is completed. In both of these cases, threads enter one-sided operation phases as dictated by the application and not language or library semantics, as in MPI-2 one-sided.

5 Performance

5.1 Sparse Triangular Solve Performance

We bench-marked our kernel on three different clusters(Table 1) and six different sparse matrices(Table 2).

The first factor that we will look at is how well our implementation of the triangular solve scales. Figure 4 shows that our implementation gets good parallel speedups. These speedups are consistent with those presented in [9]. The speedups also show that the performance only grows logarithmically with the number

	Alvarez	Lemieux	Seaborg
Processor Type	Pentium III Xeon	Compaq Alpha Server ES45	IBM Power 3
Processor Clock Rate	866 MHz	1Ghz	375MHz
Number of Processors per Node	2	4	16
Physical Memory Per Node	1 GB	4 GB	16-64GB
Interconnect Type	Myrinet	Quadrics	LAPI
Location	NERSC	PSC	NERSC

Table 1: This table shows a summary of the clusters used in the experiments.

Matrix No.	Name	Dimension	Number of Nonzeros
1	bmw7st_1-amd-L.rua	141347	3612924
2	br1r2008-amd-L.rua	60322	8874303
3	lhr71c-amd-L.rua	70304	3110198
4	pwtk-amd-L.rua	217918	43335946
5	shipsec1-amd-L.rua	140874	52189480
6	tire44k-amd-L.rua	44700	14832799

Table 2: This table shows a summary of the matrices used in the experiments. The "spy" plots of these matrices are available in the appendix of this paper.

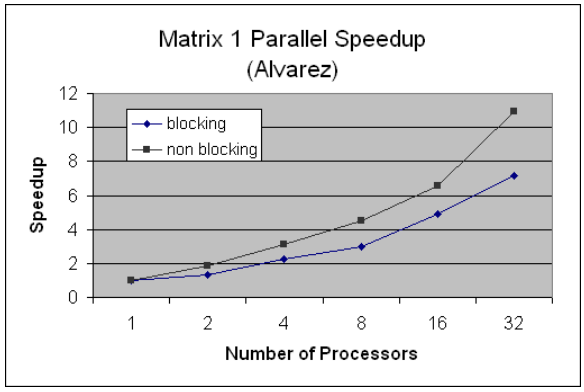

Figure 4: **Matrix 1 Parallel Speedup**. This plot shows the parallel speedup for matrix 1 on the Alvarez cluster. It also compares the performance of using blocking communications versus non blocking communications.

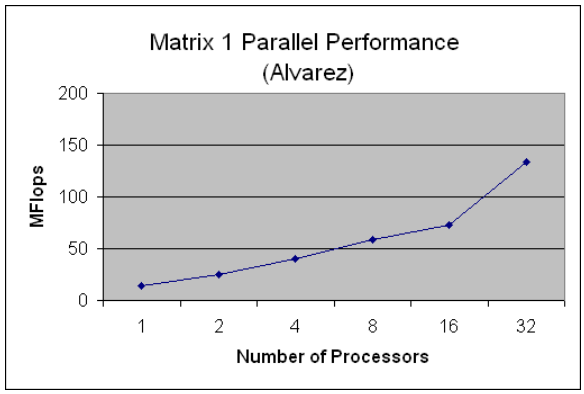

Figure 5: **Matrix 1 Parallel Performance**.

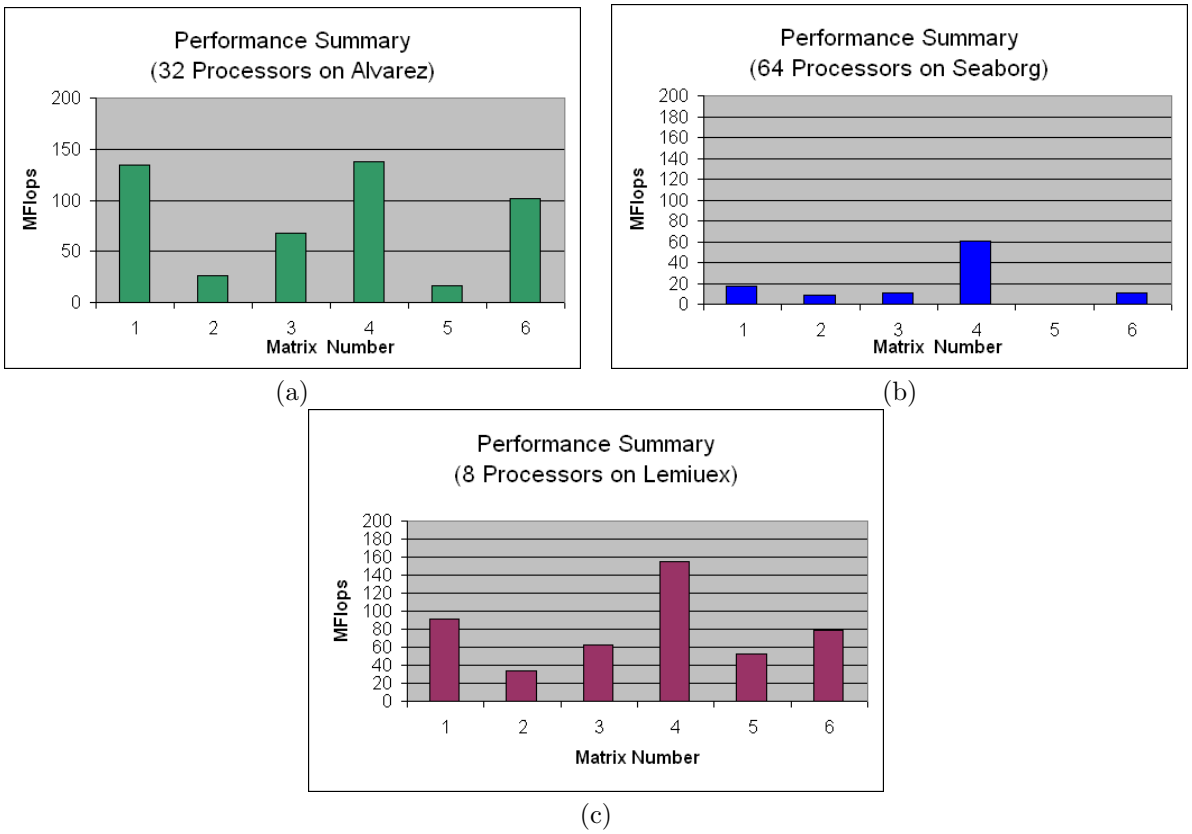

Figure 6: **Performance Summary for SpTS**.
This table shows the performance of the matrix suite in Table 2 on the clusters specified in Table 1.

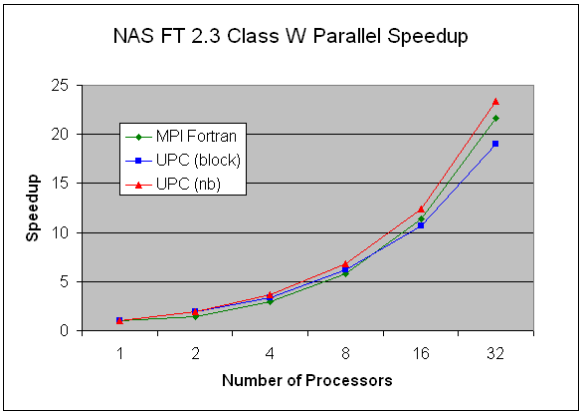

Figure 7: **NAS FT Parallel Speedup**. This plot shows the parallel speedup for NAS FT Benchmark on the Alvarez cluster. It also compares the performance of using blocking communications versus non blocking communications. In addition we also compare our implementation with MPI.

of processors. The extra overhead in communication is preventing linear speedups since the amount of messages grows super-linearly. In addition we notice that the nonblocking operations on the Alvarez cluster help. The blocking operations still have some overhead because a two sided synchronization is still needed to post the signals. However in the nonblocking case, this synchronization can be done lazily and thus there is more overlap between other parts of the computation. The nonblocking performance improvements motivate the *fire and forget* model.

From Figure 5 we see that the absolute performance is rather low. One of the fundamental issues with our implementation is that we incur the overhead of many small messages when calculating the solution. Therefore the computation to communication ratio is very poor. One of the ways to address the problem is to modify the implementation to pack the signals until there are enough to amortize the cost of a sending a message with the calculation on satisfied dependencies. This solution could however lead to deadlock if not carefully implemented.

From Figure 6a shows the performance of the different matrices on the Alvarez platform. As the plot shows matrices 2 and 5 have very poor performance compared to the others. This signifies that the dependency graphs have more edges than the other matrices. Thus we hypothesize that matrices 1 and 4 have the least complicated dependency structures allowing them to reach a higher performance than the other matrices.

Figure 6b and Figure 6c show the same story, however the relative differences in performance is different than that of the Alvarez cluster[2]. On the Seaborg cluster the performance is noticeably lower since there is no support for RDMA operations. Therefore every network communication has to involve the remote processor, therefore slowing the processors down. The cases in which the network provides RDMA support (thus allowing for effective use of the non-blocking operations) the performance is a lot better. Without the RDMA support, the *fire and forget* model performs very poorly.

The results here are only loosely compared to existing MPI codes because extracting the triangular solve from existing libraries, such as MUMPS, BlockSolve, and SuperLU would have been an unfair comparison. The triangular solve step is tightly coupled with the rest of the application. In order to get a fair comparison we would have had to rewrite the kernel in MPI, which is beyond the scope of this work.

5.2 NAS FT Performance Results

First we show that the FT benchmark itself scales well. Figure 5.2 shows the parallel speedup of the class W benchmark. The smaller benchmark was needed so that it could be run on a uniprocessor in a reasonable amount of time. The near linear speedups suggest that the computation to communication ratio is well balanced for this application, a lot more so than the SpTS. In addition we see that the difference between the various implementations of the benchmark scale with each roughly with each other. This implies that the algorithm itself, rather than the implementations is well balanced.

Figure 5.2 shows the parallel performance of the various classes of the benchmark. The performance

[2]Matrix 5 was not run on the Seaborg cluster due to resource limitations

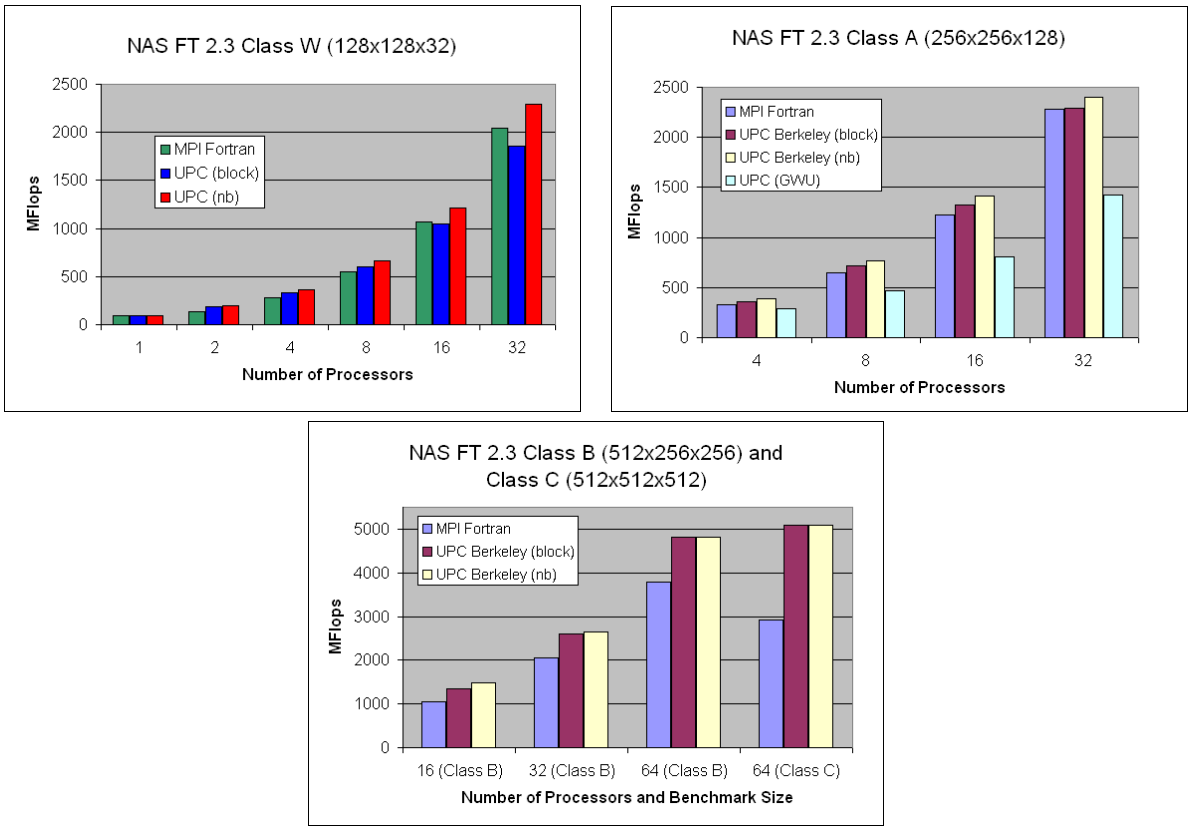

Figure 8: **NAS FT Parallel Performance**. These plots show the parallel performance of different classes of the benchmark.

of the UPC implementation is significantly higher due to the one-sided operations and the use of FFTW [7]. We notice that the blocking version of our code is actually very similar to the MPI version since the all-to-all cost is not hidden. Even though the network operations are blocking, they do not interrupt the remote processor to perform the communication since they can be performed through RDMA. Whereas in the two-sided MPI model, the remote processor gets interrupted every time a new message is ready. Thus the MPI performance limitation is the rate at which this synchronization can occur.The UPC implementation's main limitation is the rate at which a processor can inject messages into the network. Another key difference is that we leverage the use of FFTW to perform the actual FFT operations. However we do not foresee this to be a significant factor since the computation of the FFTs is not the primary bottleneck, the all-to-all is.

As the problem size and the number of processors increase, the UPC versions do much better. Again the reason for this is the RDMA ability of the network allows the remote processor to be uninterrupted during the message transfer and thus requiring little synchronization amongst the various threads. Our target system, alvarez, runs Myrinet/GM which requires memory regions to be explicitly pinned by the communications interface (in this case GASNet), before RDMA operations can be entirely issued as one-sided operations. Since the original Fortran MPI benchmark starts the FT bechmark with an untimed FFT for reasons it attributes to variable startup costs, we were actually able to do the same but in return allow all the pages on remote nodes to be pinned. Since we know that MPI does not use the same pinning strategy and cannot issue purely one-sided operations, the UPC compiler has a definite advantage in all of the FT communication steps. Additionally, the MPI version requires a lot more synchronization as the number of processors is increased.

Further analysis of the Fortran MPI version revealed that it actually does only one all-to-all communication and relaxes the assumption that the 3D FFT must be done in place. The stricter assumption requires a second all-to-all to make every transform appears as it was done in place, which the MPI version does not have. Thus even though the MPI version does one less round of communication, the UPC version still outperforms it. Future work will modify the existing framework to analyze the performance when the in-place assumption is relaxed. We would expect the UPC versions to gain significant performance improvements since most of the communication cost can be hidden.

6 Related and Future Work

6.1 Sparse Triangular Solve

There has been a lot of work in the area of analyzing SpTS for distributed memory systems, however this is the first work that we are aware of that has written an implementation in UPC. Vuduc et al.[16] did an extensive survey of the problem for the serial performance case. One of the most interesting findings from this work was that in most matrices that arise from the LU factorization, the lower right corner is dense. It was found that if they use a dense algorithm in the lower triangle then they gain significant performance improvements. As future work this algorithm would do the same. The sparse part would get solved using the algorithm described here. Santos analyzed the theoretical performance of this kernel in [14] by using the popular LogP framework. Joshi et al. [9] have created created a tree based approach to solving the SpTS on a 2 dimensional grid of processors. The dependency graph work in Section 3 was mainly motivated from this work. Yang and Gerasoulis [18] have done related work on how to schedule parallel tasks on an unbounded number of processors which is also directly related to this work.Works by [12, 2, 8, 9] have also created other implementations of SpTS that motivated our work here. One of the main future goals of the work on SpTS is to create a more optimal algorithm that has better communication to computation ratios. Such algorithms would involve better packing of the signals in order to amortize the cost of communication. In addition, since the algorithm is being writing in UPC, eventual integration into applications such as SuperLU [10] would be nice. Creating a set of UPC based kernels to insert into other applications would an ambitious and long term goal.

6.2 NAS FT

Since the NAS Parallel Benchmarks is a common set of benchmarks they have been the focus of many studies in order to analyze their performance. Vetter and Mueller [15] characterized the communication patterns of other similar large benchmarks while Abandah [1] has done some work characterizing the other NAS parallel benchmarks in a shared memory context. This work is loosely related to the work presented here.

The future work on the NAS FT benchmark is to modify the benchmark to match the semantics of the MPI implementation, not just correctness. In addition future work would also try to lift the limit of N_z processors when performing the benchmark. This would allow higher degrees of parallelism for the new Peta Flop clusters. In addition future work would also leverage the collective communications that are currently in the processor being added to the UPC language. Integrating the 3D FFT into an actual application level code would also be useful in showing the performance effects of the non-blocking operations.

7 Conclusion

This paper has covered the implementation of two new programs in UPC. They have radically different communication and computation characteristics. However in both cases we see that the non blocking communications showed a good performance improvement, since it allows the operations to be truly one sided. In addition we found that the PGAS model that the UPC language provides can lead to some novel programming ideas that are not necessarily available in MPI because of its limitation to two-sided operations. We have seen that our applications scale just as well as the existing MPI applications. However with better tuning our applications can achieve the same performance as MPI if not better (as shown with the NAS FT benchmark).

One of the main advantages of programming in UPC was that the code is a lot shorter and more expressive. Since programming for a shared memory system is well understood, it is easy to transition to using UPC over MPI. However, as is the case with any language, there are still many correctness and performance bugs that need to be addressed.

8 Acknowledgments

We would like to thank Richard Vuduc for providing his support and providing the matrices in our benchmark suite. We would also like to thank Parry Husbands for providing assistance in the NAS FT benchmark. In addition, our discussions with Jason Riedy proved to be extremely valuable in influencing our final design. Alas we can not forget the Coca Cola Corporation for keeping us awake during the long grueling nights.

References

[1] Gheith A. Abandah. Characterizing shared-memory applications: A case study of the NAS parallel benchmarks. January 17 1997.

[2] Fernando L. Alvarado, Alex Pothen, and Robert Schreiber. Highly parallel sparse triangular solution. In Alan George, John R. Gilbert, and Joseph W.H. Liu, editors, *Graph theory and sparse matrix computation*, pages 159–190. Springer-Verlag, New York, 1993.

[3] D. Bailey and P. Frederickson. A 3-d fast-fourier transform partial differential equation benchmark. In *The NAS Parallel Benchmarks Version 2.3*.

[4] Dan Bonachea. Proposal for extending the upc memory copy library functions and supporting extensions to gasnet. 2004.

[5] The Berkeley UPC Compiler.

[6] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC langauge specification (version 1.1.1). 2003.

[7] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[8] Michael T. Heath and Padma Raghavan. The performance of parallel sparse triangular solution. In S. Schreiber, M. T. Heath, and A. Ranade, editors, *Proceedings of the IMA Workshop for Algorithms for Parallel Processing*, volume 105, pages 289–306. Springer-Verlag, 1998.

[9] Mahesh V. Joshi, , Anshul Gupta George Karypis, and Vipin Kumar. A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems. In *Proceedings of the 4th International Conference on High Performance Computing*, December 1997.

[10] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, June 2003.

[11] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[12] Edward Rothberg. Alternatives for solving sparse triangular systems on distributed-memory multiprocessors. *Parallel Computing*, 21(7):1121–1136.

[13] Yousef Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[14] Eunice E. Santos. Solving triangular linear systems in parallel using substitution. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 553–560, October 1995.

[15] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures, 2002.

[16] Richard Vuduc, Shoaib Kamil, Jen Hsu, Rajesh Nishtala, James W. Demmel, and Katherine A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[17] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

[18] Tao Yang and Apostolos Gerasoulis. A fast static scheduling algorithm for dags on an unbounded number of processors. In *Proceedings of Supercomputing '91, IEEE/ACM*, Albuquerque, NM, November 1991.