

Architectural Probes for Measuring Communication Overlap Potential

Rajesh Nishtala

rajeshn@cs.berkeley.edu

Computer Science Division, University of California at Berkeley

Abstract

Conventional wisdom suggests that the most efficient use of modern computing clusters employs techniques that separate communication and computation into distinct phases. This allows the communication to remote processors to be coalesced into fewer but larger messages allowing the network to minimize the cost of message initiation as well as exploiting higher network bandwidth due to the larger message sizes. However this technique limits the cluster from realizing its full power since either the processing power or the network power is fully utilized at any one point while the other component is idle. As the processor counts and the cost of the network grow it will become increasingly important to consider algorithms that are able to fully utilize both components simultaneously. In this work we analyze one such optimization that overlaps communication and computation so that both components can be used simultaneously, thereby increasing overall application performance. Our analysis, through the use of benchmarks derived from Parallel Fourier Transforms and Parallel Sparse Matrix Vector Multiplication, explores when the technique of overlapping communication and computation is successful and what conditions aid and hinder its viability. To gain further intuition into the problem we present a series of performance models to provide more insights into the effectiveness of communication/computation overlap.

Contents

1	Introduction	3
2	Fourier Transform	3
3	Parallel SPMV	4
4	Differences between the Algorithms	4
4.1	Computation Patterns	5
4.2	Communication Patterns	5
5	Communication Benchmarks	5
5.1	Parallel FFT Derived Communication Benchmark	5
5.2	Parallel SPMV Derived Communication Benchmark	7
6	Performance Results	8
6.1	Timing Methodology	9
6.2	Pure Communication	9
6.3	Constant Number of Operations	10
6.4	Communication Pattern 1: Small number of large messages	13
6.5	Communication Pattern 2: Large number of small messages	15
6.6	Overlap and the Effects on Computation Performance	15
6.7	Overall Results	16
7	FT Performance Models	17
7.1	Preliminaries	17
7.2	Model Parameters	17
7.3	The Three Cases	19
7.4	Combining the Three Cases	19
7.5	Measuring Overhead and Gap	20
7.6	Model Verification: Many, Small Messages	20
7.7	Model Verification: Few, Large Messages	20
8	Parallel SPMV Performance Models	22
8.1	The Model	22
8.2	Model Verification: Opteron/Infiniband	22
8.3	Model Verification: Power5/Federation	24
9	Related Work	24
10	Conclusion	24

1 Introduction

Since the Berkeley NOW [12] and Beowulf [19] projects were first introduced, using clusters to solve large scale parallel problems has been gaining momentum amongst the scientists and engineers that rely on large simulations. In the span of ten years clusters have gone from being a novelty to become a main stream tool as evidenced by their rise in popularity and power on the Top 500 list [2]. However, increasingly the cost of these large parallel machines is not in the processors that perform the computation, but the network that interconnects the processors. This work primarily focuses on how to efficiently use this network by leveraging functionality that allows for efficiently overlapping communication and computation to hide the communication costs and thereby increasing performance.

Conventional wisdom says that the bulk-synchronous communication model is the most efficient way to program to modern clusters. In this model the communication and computation are separated out into distinct phases allowing the data destined for a remote processor to be packed into one phase. This allows for fewer, larger messages and thus the network is able to amortize the costs of the latency and message startup overhead required to transfer messages. The larger message sizes also allow the network to deliver higher bandwidth. One major drawback of this style of communication, however, is that during the during the computation phase the network is kept idle and during the communication phase the processors' arithmetic units are idle.

Overlapping communication and computation allows both the network and the processors to be busy simultaneously computing and transferring data. However this requires either the programmer or the compiler to explicitly specify the lack of data dependencies to expose what can be overlapped. In addition, because the communication and computation are not separated, the message count potentially increases and the message size potentially decreases since the benefits of packing can no longer be used. Many modern networks have Remote Direct Memory Access (RDMA) [6] which allows the network card to copy data directly into the memory without interrupting the processor, truly allowing overlap of communication and computation. However this capability adds a contender for memory bus bandwidth (i.e. the network card) and has interesting implications for the computation as the later sections will show.

Our previous work [7] has shown that combining the communication and computation, thus surprisingly sending more, smaller messages, allows bandwidth-limited problems to see significant performance improvements. This work further explores this phenomenon and when and how communication/computation overlap works well. RDMA based networks inherently use a one-sided approach for transferring messages since the network card is directly able to

transfer data into a remote processor's memory. We thus use a language, UPC, that is capable of using this one-sided model [26]. However the observations and the techniques can be extended into programs written in both one- and two-sided MPI [22].

We show the effectiveness of communication and computation overlap using benchmarks that are derived from two popular parallel algorithms: a multidimensional Fourier Transform and Parallel Sparse Matrix Vector Multiplication. These two algorithms have dramatically different communication and computation requirements that affect the effectiveness of communication / computation overlap. The rest of the paper is organized as follows: Sections 2 and 3 further explain the Fourier Transform and Parallel Sparse Matrix Vector Multiplication while Section 5 describes the benchmarks that we use to analyze the effectiveness of overlap. Section 6 describes detailed experimentation and performance results of when overlapping communication and computation is useful and why. Finally, Sections 7 and 8 further analyze the performance by constructing performance models in the LogGP framework [4] and verify their accuracy.

2 Fourier Transform

A large multi-dimensional Fourier Transform (FT) is widely considered to be the cornerstone in many large scale scientific applications such as computational fluid dynamics and molecular dynamics. The family of transforms that contains the Fourier Transform depend on every element of the input array interacting with every other element. For multi-dimensional FFTs, a one-dimensional FFT must be performed across each of the different dimensions. However, when the multi-dimensional domain is partitioned across processors at least one of these dimensions must be broken across multiple processors. Thus the FFT of a three dimensional cube with a 1-D partitioning (i.e. all the dimensions are larger than the number of processors) allows two out of the three dimensions to be contiguous while the third dimension is non-contiguous. Typical parallel implementations thus perform the series of steps shown in Algorithm 1 to perform a three dimensional FFT using a 1-D partitioning. Without loss of generality we assume that the X (rows) and Y (columns) dimensions are local while the Z dimension is broken across the processors.

The operation described by line 3 of Algorithm 1 is the critical performance bottleneck for parallel FFT performance. It places large strains on the network while keeping most of the computational power of the processors idle. In our previous work we analyzed how we can ease this bottleneck through effective use communication/computation overlap and one-sided communication. In this work we further explore when these techniques of overlap are effective and

Algorithm 1 Typical Parallel 3-Dimensional FFT using a 1-D layout

- 1: perform FFTs on the Y dimension
 - 2: perform the FFTs on the X dimension
 - 3: perform a global transpose of the domain so that the third dimension is contiguous
 - 4: perform the FFTs on the Z dimension
-

what network qualities facilitate this operation. Section 5.1 further analyzes the benchmark. The techniques focus on the three-dimensional Fourier Transform in the NAS Parallel Benchmark Suite [5] however, as described by Agarwal *et. al* [3], a large one-dimensional FFT can also be recast as a multi-dimensional FFT.

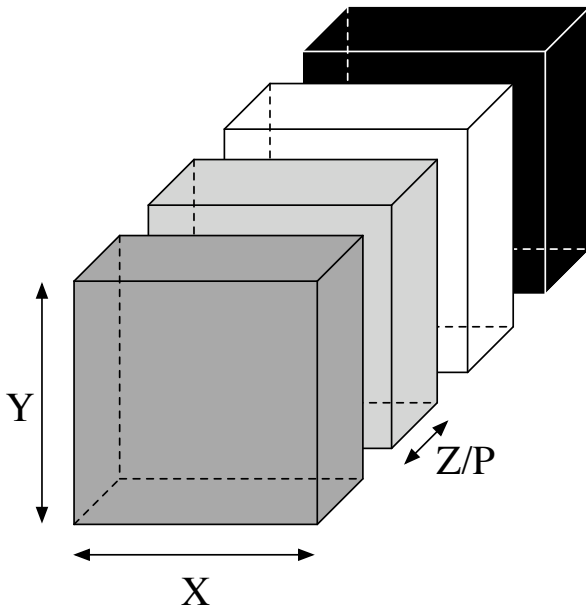


Figure 1. A sample 3 dimensional array partitioned across P=4 processors

3 Parallel SPMV

A second common parallel algorithm is a parallel Sparse Matrix Vector Product (SPMV). This kernel multiplies a sparse matrix A by a dense vector x and adds the result into another dense vector y as follows:

$$y = y + A \times x$$

To simplify notation, we will use the term “source vector” to refer to x and the term “destination vector” to refer to y .

Our previous work [23, 29] analyzed the serial performance issues associated with this operation. The main observation is that SPMV has a very poor flops per memory reference ratio, unlike the FFTs. For every element of the sparse matrix that is loaded, only two operations are performed (a floating point multiply and a floating point add). This poor ratio strains the memory system and has important implications for parallelism which will be explored later in this work. Each nonzero also requires extra storage for meta-data (indices) as well further increasing the memory requirements.

Algorithm 2 Simple Parallel Algorithm for SPMV

- 1: **for** each row r of the matrix A that this processor owns **do**
 - 2: **for** each nonzero n in row r **do**
 - 3: let c denote the column in which the n^{th} nonzero resides in the matrix
 - 4: **if** $x(c)$ is on located on a remote processor **then**
 - 5: get $x(c)$ from the remote processor where it is stored
 - 6: **end if**
 - 7: $y[r] = y[r] + A(r, c) * x(c);$
 - 8: **end for**
 - 9: **end for**
-

Algorithm 2 shows a simple parallel nested loop for performing this operation. Because we are interested in modeling SPMV performance we choose a synthetic matrix and a straight-forward parallel decomposition. In order to parallelize SPMV [17], we assume that the matrix is split evenly across the processors such that each processor gets roughly the same number of rows of the matrix. In addition, the source and destination vectors are also divided evenly amongst all the processors. Thus a processor could potentially have to fetch (or be sent) the remote source vector elements before the computation that uses those remote source vector elements can start. The matrix can also be broken up by nonzeros so that every processor gets an equivalent amount of nonzeros; however the techniques we demonstrate here are orthogonal to the issue of parallel partitioning and load balancing of the sparse matrix. Thus we chose the simpler approach and evenly divide the number of rows. Even though this might not seem like a realistic approach to the problem, it does reveal some interesting insights into communication/computation overlap. Section 5.2 further analyzes the benchmark that is used.

4 Differences between the Algorithms

At first glance it might seem as if the two algorithms have nothing in common and a comparison between them is not relevant, however both the algorithms stress the memory

systems and the communication hardware in different ways. These differences provide interesting insights into potential gains of overlapping communication and computation. In this section we explore where these differences arise and their effects on overall performance.

4.1 Computation Patterns

The first major difference between the two algorithms are the computation patterns. It is not uncommon for the uniprocessor versions of FFTs to achieve over 70% of peak performance on small problem sizes and around 25% of peak on larger problems [1]. However, SPMV performance typically only reaches at most 40% of peak on small problem sizes and can drop well below 10% of peak performance for larger ones [29]. There are many causes for the noticeable performance difference however one of the interesting causes is their differing demands on the memory system. The memory traffic to multiply a sparse matrix with a vector can be characterized as follows: every element of the matrix used exactly once, the accesses to the source vector are unpredictable (and sometimes erratic), and there are only two floating point operations per load of the matrix element¹. These factors lead to an aggressive use of the memory system which makes the memory system performance the limiting factor in overall performance and thus leading to low percentages of peak performance. These factors pose large hurdles in optimizing SPMV. High quality implementations of the FFT algorithms, such as FFTW [18], differ in the following ways: the memory access pattern is only dependent on the size of the problem, blocking can leverage locality to further reduce the memory traffic, and there are many more operations that can be performed for every element loaded. These differences, which will be demonstrated in later sections, can lead to better parallel performance results.

4.2 Communication Patterns

Another major difference between the two algorithms is the communication patterns they induce. Typical multi-dimensional FFTs are broken into distinct phases: 1) FFTs on the local and contiguous dimensions followed by 2) a global exchange so that the remaining FFTs in the other dimensions can be computed. Our previous work [7] has shown that the first phases of the local computation and the global exchange can be overlapped to realize significant performance improvements. The key observation is that the communication for the global exchange can be initiated as soon as the local computation on that piece of the

¹If a CSR [29] data structure is used to store the sparse matrix, loading a matrix element results in one load for the column index and one load for the actual value.

input array is completed and thus leading to the overlap described in the previous sections. Thus all the processors perform one-sided *put* operations to every other processor and then use a barrier to confirm that all the other threads have finished their communication. Also notice that the local computation can proceed independently of the communication events since the data for computation is already local. The lack of algorithmic interaction between the communication and computation allows for easy decoupling of the two phases. However in the SPMV algorithms, the source vector for the local computation could be located in memory with affinity to a remote thread and the computation cannot proceed until the communication events to get the remote elements are finished. The computation and communication can be overlapped through optimizations such as pipelining and prefetching the remote source vector elements before they are needed, but the dependency between the two phases poses an extra hurdle for the potential overlap.

5 Communication Benchmarks

As described in the previous sections, a multi-dimensional Fourier Transform and parallel SPMV have drastically different computation and communication requirements. In order to simplify the analysis and the potential for overlap we present two different benchmarks that model the communication and computation requirements of these kernels.

5.1 Parallel FFT Derived Communication Benchmark

In this section we design a communication benchmark that further explores the potential for overlap in multi-dimensional FFTs. Since our main goal is to overlap communication and computation we also incorporate the FFT before the global exchange (line 2 of Algorithm 1) into the benchmark.

We assume that each processor is allocated an $N \times R$ array where each $N/\text{THREADS}$ chunk of rows is to be sent to a different processor (a 3-D domain can be linearized in two out of the three dimensions to recast the problem as we have presented it). The benchmark has every processor perform an FFT on each row of length R before it can be communicated. Thus this benchmark corresponds to the middle portion of the full 3-D FFT after the FFTs in one of the dimensions is done. We ignore lines 1 and 4 from Algorithm 1 since they require no communication and thus outside the scope of this work.

The traditional bulk-synchronous approach performs all the FFTs and then sends all the data in one communication phase as represented in Algorithm 3. In this algorithm, each processor sends THREADS (where THREADS represents the number of parallel threads of control) messages

Algorithm 3 FT Communication Benchmark: Exchange (E) Version

```

1: BARRIER
2: for  $i = 0 ; i < N ; i = i + 1$  do
3:   run FFT of length R on row  $i$ 
4: end for
5: for  $t = 1 ; t \leq \text{THREADS} ; t = t + 1$  do
6:    $p = (t + \text{myid}) \bmod (\text{THREADS})$ 
7:   initiate put for rows  $[p * \frac{N}{\text{THREADS}}, (p+1) * \frac{N}{\text{THREADS}})$ 
     to processor  $p$ 
8: end for
9: for  $t = 1 ; t \leq \text{THREADS} ; t = t + 1$  do
10:   $p = (t + \text{myid}) \bmod (\text{THREADS})$ 
11:  wait for the put to processor  $p$  to finish
12: end for
13: BARRIER
  
```

of $\frac{RN}{\text{THREADS}}$ double-precision complex words. Algorithm 3 overlaps (or pipelines) communication with communication but the computation is done before any communication starts.

Algorithm 4 FT Communication Benchmark: Overlap Slabs (S) Version

```

1: BARRIER
2: for  $t = 1 ; t \leq \text{THREADS} ; t = t + 1$  do
3:    $p = (t + \text{myid}) \bmod (\text{THREADS})$ 
4:   for  $i = p * \frac{N}{\text{THREADS}} ; i < (p+1) * \frac{N}{\text{THREADS}} ; i = i + 1$ 
     do
5:     run FFT of length R on row  $i$ 
6:   end for
7:   initiate put for rows  $[p * \frac{N}{\text{THREADS}}, (p+1) * \frac{N}{\text{THREADS}})$ 
     to processor  $p$ 
8: end for
9: for  $t = 1 ; t \leq \text{THREADS} ; t = t + 1$  do
10:   $p = (t + \text{myid}) \bmod (\text{THREADS})$ 
11:  wait for the put to processor  $p$  to finish
12: end for
13: BARRIER
  
```

Algorithm 4 folds the communication phase from Algorithm 3 into the computation loop. The number and size of the messages is unchanged, but the message injection rate has changed. We define the *message injection* rate as follows: if the network is infinitely fast, the injection rate is the inverse of the time between successive network requests. The message injection rate as defined here relies purely on the rate at which the processor can finish computation without giving regards to the network traffic.

A more aggressive method to overlap the computation and communication, as motivated by our previous work, is based on the observation that once the FFT of a particular row is performed it can be sent immediately. We define

Algorithm 5 FT Communication Benchmark: Overlap Pencils (P) Version

```

1: BARRIER
2: for  $i = 0 ; i < \frac{N}{\text{THREADS}} ; i = i + 1$  do
3:   for  $t = 1 ; t \leq \text{THREADS} ; t = t + 1$  do
4:      $p = (t + \text{myid}) \bmod (\text{THREADS})$ 
5:      $x = p * \frac{N}{\text{THREADS}} + i$ 
6:     run FFT of length R on row  $x$ 
7:     initiate put for row  $x$  to processor  $p$ 
8:   end for
9: end for
10: for  $i = 0 ; i < N ; i = i + 1$  do
11:  wait for the put of row  $i$  to finish
12: end for
13: BARRIER
  
```

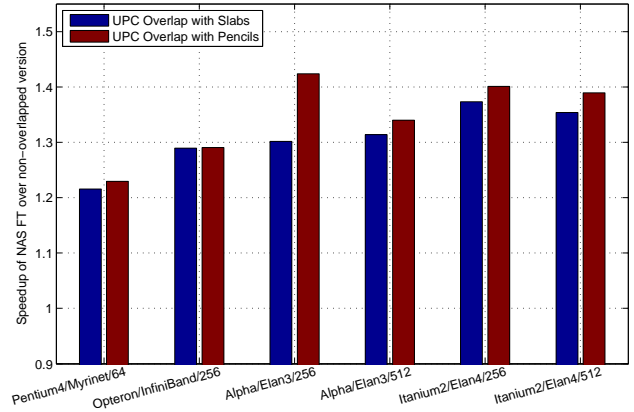


Figure 2. Performance Data from our previous work [7] motivating the three versions of the Parallel-FFT derived algorithms

a *pencil* to be a row. Algorithm 5 presents an outline for this version. The significant difference between this more aggressive algorithm and the previous ones is that the communication initiation follows directly after the FFT. However, this new algorithm sends N messages of R double-precision complex words as opposed to THREADS messages of $\frac{RN}{\text{THREADS}}$ double-precision complex words for Algorithms 3 and 4.

Figure 2 shows the speedup achieved through leveraging overlap in the NAS FT Parallel Benchmark [5]. In all the cases the largest class of the benchmark that would fit in the machine was run. The x-axis shows the processor, network, and the processor count combinations while the y-axis shows the speedup over the unoverlapped case. All the implementations are written in a one-sided UPC model which outperformed the default MPI/Fortran implementa-

y0	A(0,0)	A(0,1)	A(0,2)	A(0,3)	x0
y1	A(1,0)	A(1,1)	A(1,2)	A(1,3)	x1
y2	A(2,0)	A(2,1)	A(2,2)	A(2,3)	x2
y3	A(3,0)	A(3,1)	A(3,2)	A(3,3)	x3

Figure 3. Sample sparse matrix and corresponding vectors partitioned across 4 processors

tions by at least 20%. As the figure shows all the clusters performed the best with the most aggressive overlap pattern thus motivating the three different versions of the benchmark.

Between Algorithm 4 and Algorithm 5 there is a continuum of algorithms that can be used in the form of sending X messages of $\frac{RN}{X}$ double-precision complex words. In this work we chose to analyze the endpoints of this continuum since they provide a more drastic contrast in the implications of overlap.

5.2 Parallel SPMV Derived Communication Benchmark

In our setup of the parallel SPMV, the $N \times N$ sparse matrix, the source, and destination vectors are partitioned evenly across the processors (Figure 3 shows an example using 4 processors). Each processor only owns the source vector component corresponding to the diagonal block. The source vector for the computation is located on remote threads and it can not proceed until the communication events are finished. For example, processor 0 will need to get source vector piece x_1 before the computation on block $A_{0,1}$ can commence.

The remote source vector elements can be gathered in many different ways. Typical implementations analyze which remote source vector elements are needed and appropriately fetch these elements by having the remote processors pack all the elements into one message. Since our main focus is on benchmarking communication/computation interaction we simplify the benchmark by using a matrix in which the nonzeros are uniformly distributed throughout the matrix. If the nonzeros of the matrix are uniformly distributed throughout the matrix and the matrix is sufficiently

dense then, with high probability, the entire source vector will be needed. This benchmark assumes that condition and fetches the entire vector rather than the individual elements as needed. This allows the analysis to more easily control the amount of work that each processor has and thus give a deeper understanding of the impact of overlapping communication and computation. This artificial matrix provides interesting insights into the most difficult case of overlap of communication and computation in SPMV; the case when each processor has to communicate a large amount of data with every other processor. Understanding this case provides an understanding of the lower bound in performance of Parallel SPMV. Better ordered matrices will perform much better since their communication requirements will be a lot less intensive. Analogous to the Parallel Fourier

Algorithm 6 Parallel SPMV Communication Benchmark: Bulk-Gather (B) Version

```

1: BARRIER
2:  $p = myid + 1$ 
3: for  $i = 1 ; i < THREADS ; i = i + 1$  do
4:   initiate get for source vector from processor  $p$ 
5:    $p = (p + 1) \bmod THREADS$ 
6: end for
7:  $p = myid + 1$ 
8: for  $i = 1 ; i < THREADS ; i = i + 1$  do
9:   wait for get for source vector from processor  $p$  to finish
10:   $p = (p + 1) \bmod THREADS$ 
11: end for
12: for  $t = 0 ; t < THREADS ; t = t + 1$  do
13:   $\phi = (t + myid) \bmod THREADS$ 
14:  perform SPMV on block  $myid, \phi$ 
15: end for
16: BARRIER

```

Transform, we begin the presentation of the algorithms with a version that does not perform any communication / computation overlap and only leverages communication/communication overlap. Algorithm 6 presents an outline of the algorithm. Notice that all the communication is done before any computation is started.

As before, we can apply techniques to overlap the communication and computation. The observation is that the remote source vector elements are not needed until the SPMV on that piece of the matrix is performed. In addition the diagonal blocks on all the processors are guaranteed not to incur communication. Thus an operation could initiate the one-sided `get` operations for a few of the remote source vector pieces before starting the local computation with the anticipation that once the local computation finishes the `get` operations will finish and the communication costs are hidden. Any communications events not done in this prefetch

phase are made by pipelining the communication and the computation. Algorithm 7 represents an outline of this approach.

Algorithm 7 Parallel SPMV Communication Benchmark: Gather (G) Version

```

1: BARRIER
2:  $p = myid + 1$ 
3:  $count = 0$ 
4: for  $i = 1 ; i < \eta ; i = i + 1$  do
5:   initiate get for source vector from processor  $p$ 
6:    $p = (p + 1) \bmod \text{THREADS}$ 
7:    $count = count + 1$ 
8: end for
9: for  $t = 0 ; t < \text{THREADS} ; t = t + 1$  do
10:   $\phi = (t + myid) \bmod \text{THREADS}$ 
11:  if  $count < \text{THREADS} - 1$  then
12:    initiate get for source vector from processor  $p$ 
13:     $p = (p + 1) \bmod \text{THREADS}$ 
14:     $count = count + 1$ 
15:  end if
16:  if  $t > 0$  then
17:    wait for get from processor  $\phi$  to finish
18:  end if
19:  perform SPMV on block  $myid, \phi$ 
20: end for
21: BARRIER

```

One of the crucial parameters to understand is η , the number of the source vector blocks we prefetch ahead. Notice that in this implementation we always overlap at least one get. When η is one less than the number of processors then we initiate all the gets and have a bulk-synchronous style communication approach while a small η indicates a finer overlap between communication and computation.

A major drawback with `get` operations are that they incur two-round trip latencies: one for the request and the other for the reply. To reduce this affect we can also build an analogue communication pattern out of `put` operations, however one-sided puts lack the semantic power to guarantee that the message has been delivered; it can only guarantee that message has been sent. One way around this is to use a series of two `put` operations, the first is the payload while the second is a flag indicating that the transfer is finished. In order to assure the guarantee of delivery the second flag can not be sent until there is a guarantee from the remote side that the data has arrived. Naively sending the flag information piggy-backed with the data does not guarantee correctness since the data and flags can appear out of order on the remote end triggering a false message arrival indication. Another way around this is to use Active Messages [27] in which the data itself carries information on which flag to signal thus incurring only one message, which

we call a signaling put. This method, however, requires the processor to be interrupted to run the Active Message handler after the entire data transfer is finished. A signaling put implementation of the benchmark is represented in Algorithm 8.

Algorithm 8 Parallel SPMV Communication Benchmark: Scatter (S) Version

```

1: BARRIER
2:  $p = myid - 1$ 
3:  $count = 0$ 
4: for  $i = 1 ; i < \eta ; i = i + 1$  do
5:   initiate signaling put for source vector to processor  $p$ 
6:   if  $p == 0$  then
7:      $p = \text{THREADS} - 1$ 
8:   else
9:      $p = p - 1$ 
10:  end if
11:   $count = count + 1$ 
12: end for
13: for  $t = 0 ; t < \text{THREADS} ; t = t + 1$  do
14:   $\phi = (t + myid) \bmod \text{THREADS}$ 
15:  if  $count < \text{THREADS} - 1$  then
16:    initiate signaling put for source vector to processor
17:     $p$ 
18:    if  $p == 0$  then
19:       $p = \text{THREADS} - 1$ 
20:    else
21:       $p = p - 1$ 
22:    end if
23:     $count = count + 1$ 
24:  end if
25:  if  $t > 0$  then
26:    wait for remote processor  $\phi$  to signal that its put
27:    has finished
28:  end if
29:  perform SPMV on block  $myid, \phi$ 
30: end for
31: BARRIER

```

6 Performance Results

The algorithms in Section 5 have shown different implementations of the benchmarks. In this section we present a performance analysis on two different clusters of the different algorithms. The first, named “Jacquard”, is an Opteron cluster using the Infiniband interconnect with two processors per node while the second, named “Bassi”, is a Power5 cluster using the Federation interconnect with eight processors per node. Full details of these clusters are available in the appendix. These machines differ in many ways but the key differences that we will explore are the processor

speeds (the peak performance of the Power 5 is about twice as fast that of the Opteron) and the network characteristics. The Infiniband cluster has RDMA (Remote Direct Memory Access) [6] built into network while on the Federation cluster, the RDMA support has not yet been enabled. We will use these important differences to explore the impacts of overlap within the framework of these the algorithms.

6.1 Timing Methodology

The following sections present data when communication and computation are overlapped. However timing such a phenomenon can be subtle since it is hard to explain with one serial timer when communication and computation are overlapped. Our implementations of all the benchmarks rely on the nonblocking communication primitives built into the Berkeley UPC compiler [8]. Every communication event is broken into two distinct steps: an *initialization* step and a *wait* step. The computation, however, is unbroken and considered a black box for the sake of these experiments. When communication and computation are overlapped we expect that the computation time to stay constant while the “visible” communication time decreases.

For example, let’s say a communication initiation takes 1 unit of time while it takes an additional 5 units of time for that communication operation to finish. Now assume that a computation takes 2 units of time to execute. So if we initialize the communication, run the computation, and wait for the communication to finish we overlap the communication and the computation. The total time is measured as 1 unit for initialization, 2 units for computation, and $5 - 2 = 3$ units of time that we have left waiting for the communication to finish, which implies the entire sequence takes 6 units of time. If the computation takes longer than 5 units of time in our example, then the total time spent waiting for communication to finish is 0 since the communication event finished while the computation was still running. We thus defined the “visible” communication as the time during which the computation does *not* hide the communication. Thus, the wait time is a direct correlation of the communication that could not be overlapped behind computation. Ideally the wait time would be 0 indicating that communication was perfectly overlapped with the computation.

6.2 Pure Communication

In the first experiment Figures 4 and 5 show the performance when the total communication volume is kept constant while varying the number and size of the messages. In all the bars Algorithms 3 (E) and 4 (S) communicate the same volume of data using 32 messages (since our experiments are conducted across 32 processors) while Algorithm 5 (P) sends 8k messages of 512 complex elements each on the

left side of the figures. On the right side *Pencils* sends the same volume of data through 64 messages of 64k elements each. In these experiments the computation in lines 3, 5, and 6 of Algorithms 3, 4, and 5, respectively are not executed. Thus *Exchange* and *Slabs* reduce to identical algorithms and we expect them to yield identical performance results. The x-axis in Figures 4 and 5 shows the number of double-precision complex elements per row (R) and the number of rows (N), while $N \times R$ yields the total number of double-precision complex elements. The y-axis shows the time and each bar is broken into three distinct pieces. The section titled “Comm Init” shows the amount of time spent in line 7 of Algorithms 3- 5 (timers are started before and stopped after line 7). “Comm Sync” shows the time spent in line 11 for all three algorithms (i.e. timers for this section are started before line 11 and stopped after line 11), while “Barrier” represents the total time spent executing line 13 (i.e. timers are started and stopped before and after line 13 respectively). The “Barrier” time includes the time waiting for other processors to finish as a result of load imbalance as well as the time to perform the global barrier. The labels on the bars “E”, “S”, and “P” refer to Algorithms 3, 4, and 5, respectively. The time presented in each of the segments is the mean of the individual times collected across all the processors, however it was found that the variance in the data for all the experiments was low.

From Figures 4 and 5 we can deduce the following observations:

- The total time required for *Exchange* (Algorithm 3) and *Slabs* (Algorithm 4) is roughly constant for all configurations for a given network, as expected, since the total communication volume is constant and we the benchmarks reduce to bandwidth intensive communication benchmarks.
- On the Opteron/Infiniband cluster *Pencils* (Algorithm 5) performs worse with a large number of messages, as expected, however the performance degradation is not extremely severe; *Pencils* only takes 10% more time at its worst. On the Power5/Federation cluster, like the Opteron/Infiniband cluster, *Pencils* performs worse but this time the degradation is much more severe at about 40%.
- There is a large drop in the execution time of *Pencils* as the number of messages drops. The higher initiation time (up to 3.3x) of *Pencils* on the Power5-Federation cluster quantifies the higher per message cost since this machine has no RDMA.
- At high message counts, on the left, the time for *Pencils* is dominated by the communication initiation time while this initiation time is negligible on the right with low message counts.

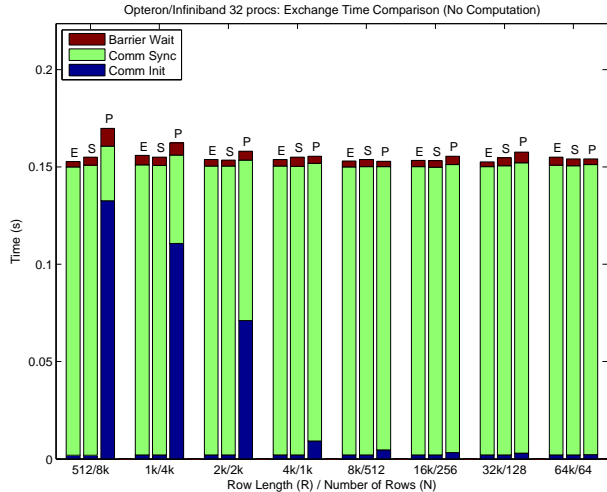


Figure 4. Opteron/Infiniband Parallel FT Communication Comparison (No Computation)

- Finally, comparing across the two graphs shows that the Opteron/Infiniband cluster is able to perform this operation in about 60% of the time it takes the Power5/Federation cluster.

SPMV Gather and *SPMV Scatter* (Algorithms 7 and 8) present a second style of overlap algorithms, those in which the computation depends on previous communication events. Figures 6 and 7 show the impact of running *SPMV Gather* and *SPMV Scatter* on both platforms when the computation step is skipped. The x-axis represents the prefetch depth, η , while the y-axis represents time. Each color in the stacked bar represents a different portion of the runtime. The “Prefetch Comm Init” represents the time spent in lines 4 (4) through 8 (12) in *SPMV Gather* (*SPMV Scatter*) while “Pipeline Comm Init” represents the time spent in lines 11 (15) through 15 (23). The time spent waiting for communication to finish, lines 16 (24) - 18 (26) is shown in the bar marked “Comm Sync.” Finally, the time spent at the final barrier in line 21 (29) is represented by the bar marked “Barrier.”

Figures 6 and 7 thus reduce to presenting the effect of varying the number of outstanding messages in the exchange. The following observations can be drawn these figures:

- On the both clusters, the performance of *SPMV Gather* has a lot less variability in performance than *SPMV Scatter*.
- On the Opteron/Infiniband cluster, *SPMV Scatter* has much better performance when there are many outstanding messages (around $\eta = 20$) while *SPMV Gather* has the best performance when the number of outstanding messages is small (around $\eta = 4$).

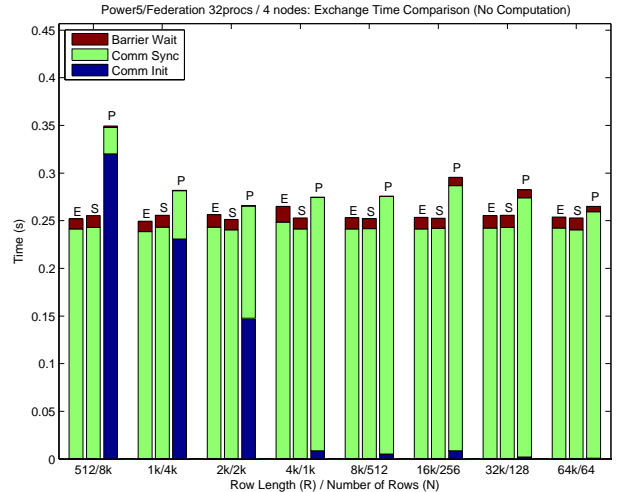


Figure 5. Power5/Federation Parallel FT Communication Comparison (No Computation)

- On the Power5/Infiniband cluster, *SPMV Gather* performs the best when $\eta = 1$ while *SPMV Scatter* has the lowest execution time when $\eta = 16$.
- Like the previous set of plots, the performance the Opteron/Infiniband cluster is able to perform the network operations in about 65% of the time that it takes the Power5/Federation cluster.

This data implies that *SPMV Scatter* is more susceptible to load imbalance when the number of outstanding messages is kept low compared to *SPMV Gather*. If one processor slows down, then the impact of *SPMV Scatter* on the other processors is a lot more pronounced since they can not initiate their signaling puts until previous communication events have finished causing a chain reaction that slows the entire system down.

Even though there is no computation at this stage of the experimentation, the results do reveal some interesting insights. Most notable is that while sending more, smaller messages hurts performance the performance degradation on the Opteron/Infiniband cluster is not as pronounced as one would expect. Thus the RDMA is also helping the algorithms overlap communication with communication. This will have important implications as the later sections will show.

6.3 Constant Number of Operations

The data presented so far do not motivate our algorithms that send more, smaller messages because when only communication is involved, the traditional idea of sending fewer, larger messages yields the best performance; however the experimentation does not end here. In this section we add

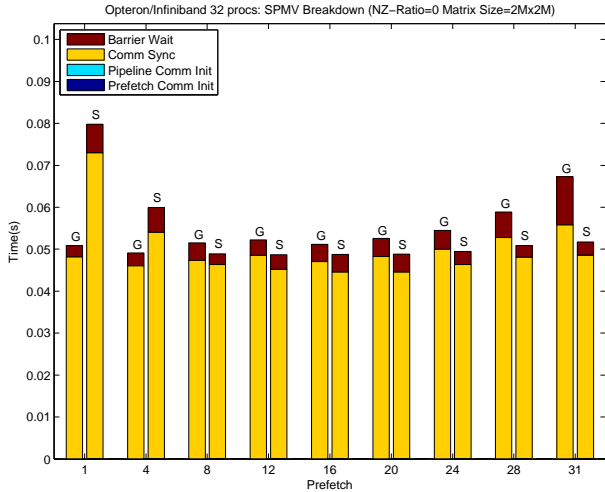


Figure 6. Opteron/Infiniband SPMV Gather and SPMV Scatter Comparison (No Computation)

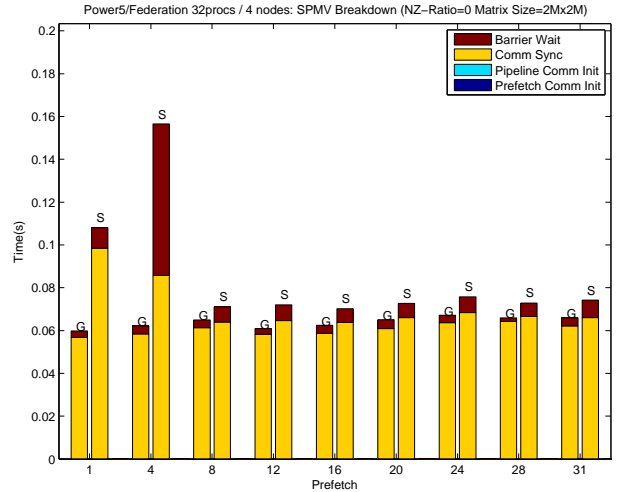


Figure 7. Power5/Federation SPMV Gather and SPMV Scatter Comparison (No Computation)

computation to all of the algorithms and show how the impact of communication *and* communication affect the overall performance of the different algorithms.

To introduce computation into the picture we rerun the experiments from Figures 4 and 5 for *Exchange*, *Slabs*, and *Pencils* but add a constant amount of computation in every step. Thus, Figures 8 and 9 show how the performance of the algorithms is affected when the computation is added and the communication is overlapped behind this computation. In order to keep the total number of flops (floating point operations) constant, the FFT is performed on R' (where $R' \leq R$) of the elements in every row such that the total number of flops is constant across the various combinations of R and N . The timers to measure the local computation are started before and stopped after lines 3, 5, and 6 of *Exchange*, *Slabs*, and *Pencils* respectively. It should be noted, however, that keeping the flops constant did not necessarily guarantee that the total execution time was constant since larger values of R' incurred more memory traffic and thus more memory system affects.

Even though this benchmark differs from the our motivating FFT example, it does show the impact of overlap by adding a constant amount of computation to the previous communication benchmark. However only performing the FFTs across all R elements for every row varies *both* the communication pattern *and* the computation pattern. Thus to draw conclusions about the ability to overlap we present data in which the total amount of computation (i.e. the number of flops) is fixed across all the different runs.

The following observations can be drawn from Figures 8 and 9:

- Unlike the case with no computation, *Slabs* and *Pencils* perform better than *Exchange*, their bulk-synch-

ronous counterpart. On the Opteron/Infiniband cluster the communication time is almost completely hidden by using this method of overlap. *Slabs* and *Pencils* have roughly the same performance on the Opteron/Infiniband cluster since they both effectively hide almost all the communication and thus it is difficult to discern the differences between them.

- On the Power5/Federation cluster *Slabs* is almost always the winner while *Pencils* still beats *Exchange*. This indicates that overlapping computation and communication is a good idea, however too many messages still hurt the performance leading one to prefer a slabs style communication pattern for this cluster.
- Even though the Power5 consistently performs the computation faster, the total time to solution when R is small and N is large is about 20% quicker on the Opteron cluster simply because of the ability to hide the communication costs.

From the figures and the observations we see that *Slabs* is almost always the winner on the Power5/Federation cluster while *Slabs* and *Pencils* result in the same performance on the Opteron/Infiniband cluster. Thus overlapping communication and computation is a good idea for this benchmark, however the large granularity of overlap is much more critical for performance on the Power5/Federation cluster.

Section 4 outlined the computational differences between the two kernels leading us to hypothesize that the kernel's impact on the memory system would affect the impact of the overlap. In addition, the semantic differences in the communication requirements, such as waiting for communication on a segment to finish before that computation on that segment can start, also pose hurdles for the Parallel

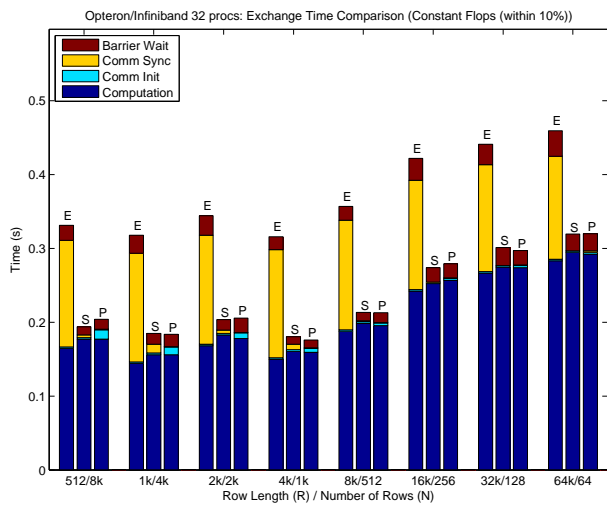


Figure 8. Opteron/Infiniband Algorithm Comparison (Constant Number of flops)

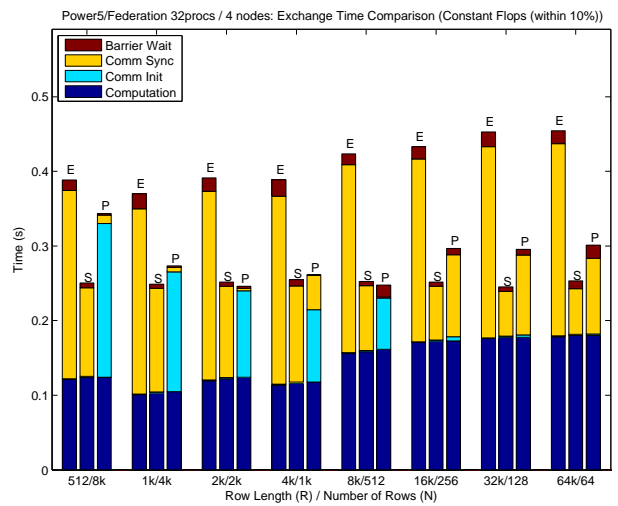


Figure 9. Power5/Federation Algorithm Comparison (Constant Number of flops)

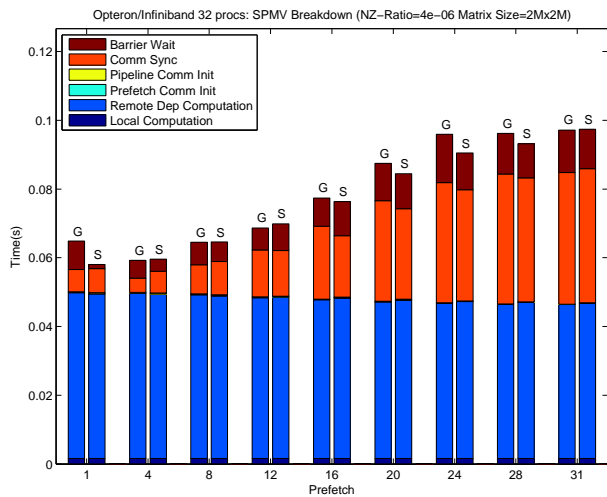


Figure 10. Opteron/Infiniband SPMV Gather and SPMV Scatter Comparison (Constant Computation)

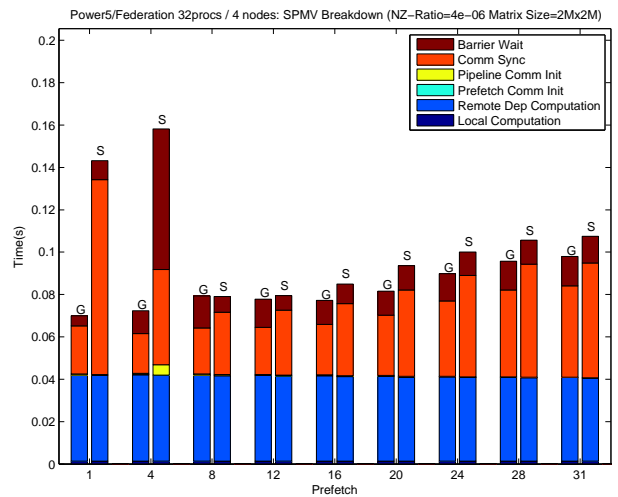


Figure 11. Power5/Federation SPMV Gather and SPMV Scatter Comparison (Constant Computation)

SPMV that the multi-dimensional FFT did not have to contend with.

Figures 10 and 11 explore this space by keeping the amount of flops constant while varying the prefetch depth, η . Since the matrix is not changed across the runs, the variation is not as dramatic as the FFT case. A nonzero ratio (the ratio of the number of nonzeros to the square of the matrix dimension) of 4×10^{-6} is a good representation of the effects however the trends that we extract are not unique to this ratio. The “Local Computation” time represents the amount of time spent computing the diagonal block (i.e. the SPMV on the piece that does not require communication). The “Remote Dep Computation” bar represents the amount of time spent doing the SPMV on the off-diagonal blocks (i.e. the SPMV on the pieces where communication is required to get the source vector). The breakdowns for the other timers are the same as Figures 6 and 7. From Figures 10 and 11 we can draw the following conclusions:

- Since this is a memory intensive benchmark, even though the peak performance of the Power 5 is about twice as fast as the Opteron, the computation is only running in about 80% of the time it takes the Opteron to perform the same operations.
- On both the clusters, the *SPMV Gather* does the best with a low prefetch depth. In addition the variation between the various low values of η is small for both the clusters indicating that a prefetch of 1, while not a perfect choice, would lead to a nearly optimal communication pattern.
- Comparing these figures to Figures 6 and 7 show that the total execution time has not dramatically changed, even though more computation is being done. The communication time is being hidden behind the computation time, demonstrating that the overlap is helping.
- We also notice that *SPMV Scatter* achieves the same performance as *SPMV Gather* on the Opteron clusters for low prefetch depths and beats the gather-based approach for higher values of η . On this cluster, choosing a prefetch of 1 yields for either algorithm leads close to optimal performance.
- On the Power5/Federation cluster, however, the *SPMV Gather* handily beats *SPMV Scatter* for low prefetch depths, and unlike the Opteron Infiniband cluster, tracks the gather-based approach at *high* prefetch depths. We also notice that if *SPMV Scatter* is employed, then careful analysis or search must be done to pick the optimum prefetch.

6.4 Communication Pattern 1: Small number of large messages

In the previous section, it was found that the introduction of computation into the benchmarks affected the overall runtime of the different algorithms. The different algorithms also experienced different speedups or slowdowns depending on how much communication was overlapped with the fixed amount of computation.

To further explore these differences, the communication pattern is kept constant while varying the computation. Since different communication patterns induce different behaviors with respect to the ability to overlap, we first explore the simpler pattern where every processor wants to send exactly one message to every other processor. We also increase the length of each row to 64k complex elements, thus $N = 32$ and $R = 64k$. Thus every processor, in this example, sends a small number of large messages, representative of the right-most group of bars in Figures 4, 5, 8, and 9.

Figures 12 and 13 show the impact of varying computation with a fixed communication pattern. The computation is varied by performing the FFT on a fraction of the row. The x-axis shows the fraction of flops that are performed, when compared to performing the FFT on all $R = 64k$ elements in all the rows. The y-axis shows the time in seconds and the breakdowns are the same as previous figures. From these figures the following observations can be drawn:

- When there is no computation, the algorithms have nearly identical performance. This is expected since the algorithms perform the exact same sequence of operations.
- *Slabs* and *Pencils* exhibit the same performance regardless of the computational load since with $N = 32$ they both reduce to identical algorithms
- The performance for *Exchange* shows the total runtime if the two phases were separated and thus the difference in runtimes between *Exchange* and either *Slabs* or *Pencils* shows the performance gains through overlap.
- On the Opteron/Infiniband cluster the performance of *Slabs* and *Pencils* is constant until the computation time is larger than the time it takes to perform the communication. This indicates that the total time is determined by the time to perform the communication when the computation is low and that the time is being hidden effectively behind the communication.
- When there are a small number of large messages, the Power5/Federation cluster benefits from overlap even though the network does not have RDMA capabilities enabled.

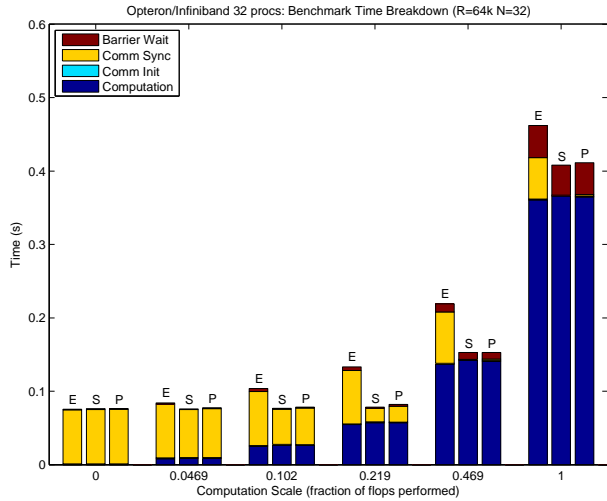


Figure 12. Opteron/Infiniband *Exchange - Pencils* Comparison (Small Number of Large Messages)

- The “Computation” bars for both clusters and all three algorithms show that there is little variance in local computation time across the algorithms. Thus a comparison between the case that has no overlap, *Exchange*, and the cases that do, *Slabs* and *Pencils*, shows that the local computation is not affected by the aggressive overlap. Thus implying that the memory system is able to sustain both the network card as well as the processor.

The SPMV communication pattern also falls into the category of a small number of large messages and thus we present the data for these experiments in Figures 14 and 15. In these figures the column “B” denotes the performance of *SPMV Bulk-Gather* (Algorithm 6), the version that has no communication / computation overlap and relies on message pipelining and communication / communication overlap. The computational load was varied by changing the number of nonzeros in the matrix. The previous section demonstrated that a prefetch of $\eta = 1$ had the best performance amongst both the algorithms and so we keep the prefetch constant at 1 for this experiment. From these figures we can draw the following observations:

- On the Opteron/Infiniband cluster both *SPMV Gather* and *SPMV Scatter* effectively use communication / computation overlap to significantly outperform *SPMV Bulk-Gather*.
- On the Power5/Federation cluster *SPMV Gather* also effectively uses overlap to outperform *SPMV Bulk-Gather* however there are no benefits to using *SPMV Scatter*. The cost of the active messages thus cause

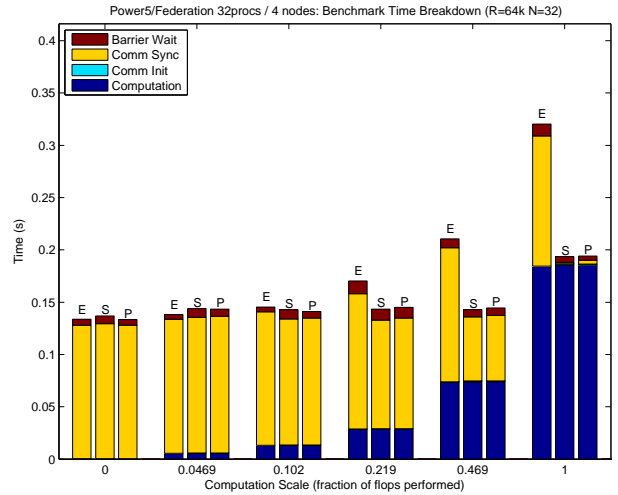


Figure 13. Power5/Federation *Exchange - Pencils* Comparison (Small Number of Large Messages)

the communication to slow down enough where the overlap no longer helps.

- On the Power5/Federation cluster when the nonzero ratio is 1×10^{-4} *SPMV Gather* slightly outperforms *SPMV Bulk-Gather* even though it hides most of the communication. The seemingly inconsistent is a result of *SPMV Gather* spending more time in the computation than *SPMV Bulk-Gather* indicating that the aggressive overlap slowed the local computation performance down.
- Unlike the FFT-based benchmarks, the total runtime steadily increases as the computational load increases even for low computational loads. This is probably due to the nature of the communication pattern and having to wait for previous communication to finish before new computation can start.
- On the Opteron/Infiniband cluster, the communication accounts for a small fraction of the total runtime while the Power5/Federation cluster spends more time in communication.
- *SPMV Gather* is generally faster than *SPMV Scatter* however this is probably due our choice of $\eta = 1$. A prefetch of $\eta = 4$ would minimize this effect.
- For low computational intensities, the Opteron/Infiniband cluster is on par with the Power5/Federation cluster even though it is a slower processor because of the time the Power5/Federation cluster has to spend in network operations. However as the computation becomes the dominant factor, the Power5/Federation cluster wins.

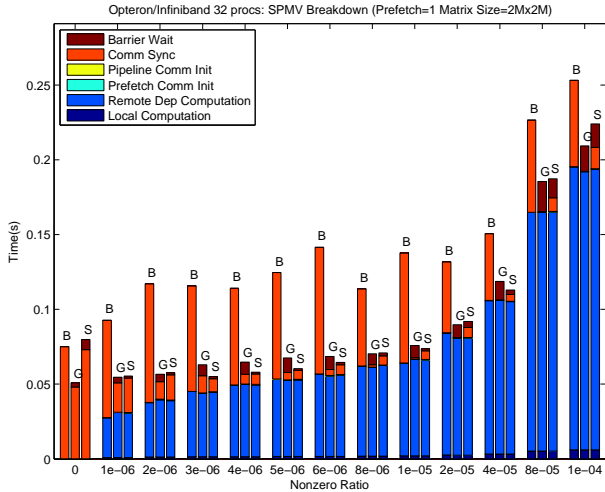


Figure 14. Opteron/Infiniband SPMV Gather and SPMV Scatter Comparison (Small Number of Large Messages)

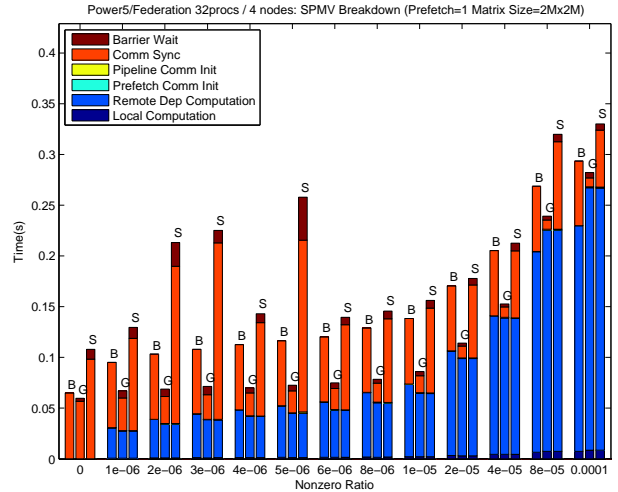


Figure 15. Power5/Federation SPMV Gather and SPMV Scatter Comparison (Small Number of Large Messages)

6.5 Communication Pattern 2: Large number of small messages

Our thesis thus far has been that overlapping communication and computation is useful when one wishes to increase the performance by diffusing the communication events throughout the computation. With the previous message count and size, the communication events were spread apart since the FFTs were relatively large. However, to further motivate the aggressive use of overlap, we present data when the message sizes are small ($R = 512$) and when the message count is high ($N = 8192$). The message injection rate, the message size, and the number of messages make this communication pattern very different than the previous pattern. This problem size is a good representative of the main transpose step in the NAS FT Class C benchmark across 32 processors. Since our implementation of the parallel SPMV benchmark does not lend itself easily to this messaging pattern, we only analyze the Parallel-FFT derived benchmark.

Figures 16 and 17 show the performance when a large number of small messages are sent. Again the computation is varied as mentioned in the previous section. We observe the following from these plots:

- When there is no computation, the *Pencils* performs the worst since the higher number of messages incur more overhead.
- However, as the computational load increases *Slabs* and *Pencils* beat *Exchange*, which indicates that the overlap is working well even when the message count is high.
- Since the performance of *Slabs* and *Pencils* is roughly

constant as the computation increases, it shows that both clusters are able to successfully overlap communication and computation.

- The difference in execution time between *Slabs* and *Pencils* is a lot higher on the Power5/Federation because of the network performance. Most of the time in both cases is being spent initiating communication events.
- The Opteron/Infiniband cluster consistently outperforms the Power5/Federation cluster for this message count/size combination. Thus the slower network sufficiently hinders the faster Power 5 processor enough where the slower Opteron cluster can execute the benchmark quicker even though the local computation is quicker on the Power 5.
- Again comparing the local computation across the algorithms for both the clusters shows that the aggressive overlap does not affect local computation when the FFTs are used for the computation.

6.6 Overlap and the Effects on Computation Performance

In the previous sections, it was found that the local computation performance for the parallel-FFT derived benchmark was not affected by the aggressive overlap of communication and computation. However in Section 4 we discussed how the Sparse Matrix Vector Multiply, SPMV, was a very memory intensive kernel. Thus we argue that the aggressive overlap, which leads to both the network card

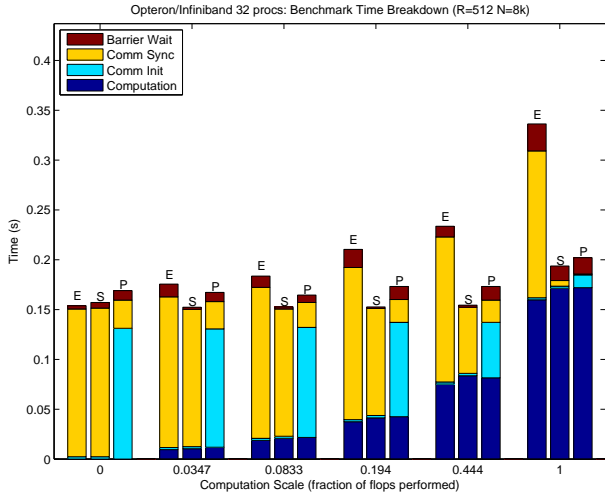


Figure 16. Opteron/Infiniband Exchange - Pencils Comparison (Large Number of Small Messages)

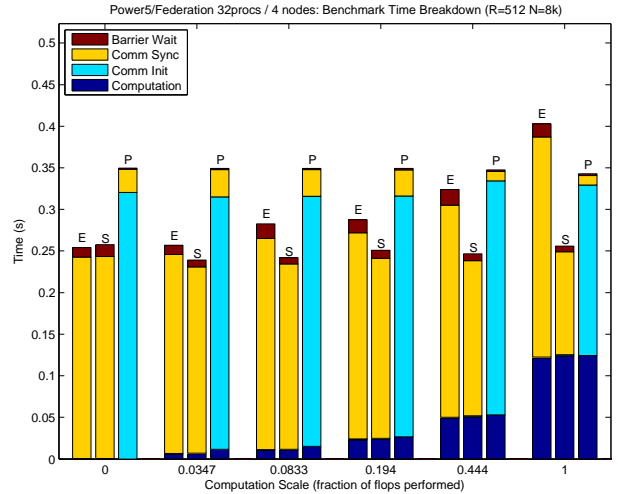


Figure 17. Power5/Federation Exchange - Pencils Comparison (Large Number of Small Messages)

and the CPU placing requests on the memory bus would slow down the computational parts of the SPMV benchmark. Our analysis of Figure 15 showed experimental evidence of this effect.

Figures 18 and 19 show the effects of the aggressive overlap on local computation for the SPMV-derived benchmark. The bar titled “Algorithm 4” shows the time spent performing SPMVs for *SPMV Gather* while “Algorithm 4 (no comm)” shows the time spent performing SPMVs for the same algorithm, however this time all the communication has been turned off. Similarly the same data is shown for *SPMV Scatter*. Thus there will be no remote communication events to interrupt the computation and thus the memory system need not worry about servicing network requests. From these figures we can make the following observations:

- On the Opteron/Infiniband cluster the variations that don’t perform any communication run over 10% faster than the versions that do require communication in some cases indicating that the communication has an impact on local computation performance as predicted.
- On the Power5/Federation cluster the variations in performance are hardly noticeable at around 2-3%. Since we do not use the RDMA capabilities of this cluster, the previously mentioned memory bandwidth contention problem does not come up thus showing that computation does not slow down.
- Because we are using 8 processors per node in the case of the Power 5/ Federation cluster the per processor memory bandwidth is lower and thus affects the

SPMV performance. Our previous work [28] more deeply explores this phenomenon.

- There is also a high degree of variability in the difference between enabling and disabling the communication on the Opteron/Infiniband cluster while the variability is a lot less pronounced in the Power5/Federation cluster. One cause of this variability is the inherently unpredictable nature of how the processor and memory system are affected when the network hardware has to deal with communication events.

6.7 Overall Results

The experiments performed during this section all exposed different results however there were a few interesting and common results that can be gleaned from all the experiments. The most important is that as the computation costs grow the overlap allowed the “visible” communication time to dramatically decrease, indicating that we were successfully overlapping communication and computation. The experiments also showed that the slower Opteron/Infiniband cluster was able to perform calculations that required a lot of communication and computation in less time than the Power5/Federation cluster due to the ability to effectively manage overlap. In addition, it was found that the Opteron/Infiniband cluster is able to more easily deal with more smaller messages thus indicating that algorithms requiring finer-grained overlap of communication and computation would benefit more from an RDMA based cluster than one without. Finally, overlapping communication and computation helped performance regardless of the RDMA capabilities of the network. Both clusters saw improve-

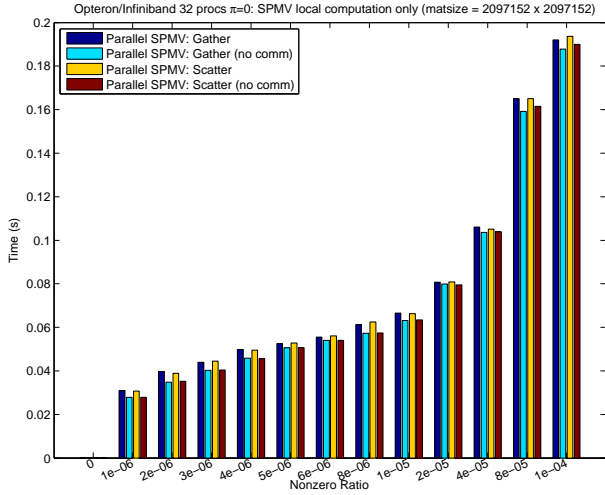


Figure 18. Opteron/Infiniband SPMV Gather and SPMV Scatter Local Computation Comparison

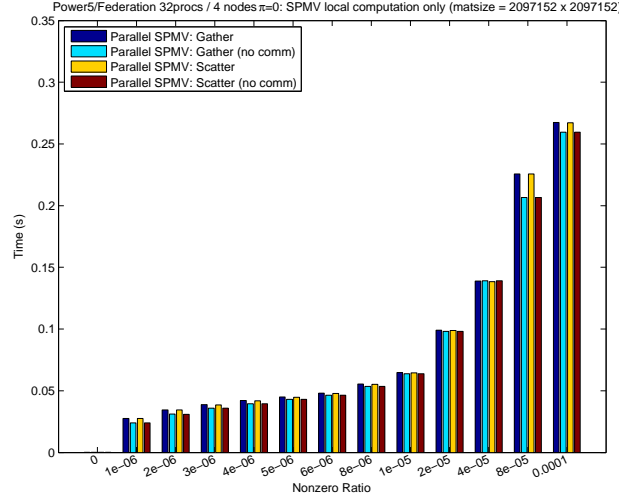


Figure 19. Power5/Federation SPMV Gather and SPMV Scatter Local Computation Comparison

ments in the performance through the algorithms that used overlap.

7 FT Performance Models

In order to better understand the benchmark presented in Algorithm 5 and understand the factors that influence effectiveness of the overlap we present a set of performance models in the LogGP [4] framework. The primary goal of the models is to understand how long a particular processor spends in computation and communication of its $N \times R$ array.

7.1 Preliminaries

Our hypothesis thus far has been that the overlap of communication and computation provides performance advantages because the communication time or the computation time can be hidden. We assume the following when constructing the models:

- The computation can be overlapped with the network communication.
- There is an overhead cost associated with initiating a network message that can not be overlapped.
- The time to copy the data into the network, i.e. the G term in the LogGP model, can be overlapped with computation because of the advanced hardware (RDMA) in the network card.
- There is a queue in the network that handles transmissions and it takes a nonzero time to add something to this queue.

- The time needed to transfer the entire $N \times R$ array is much greater than the time needed to initiate the corresponding communication events.

7.2 Model Parameters

- **L: Latency.** We assume that the latency is zero since the latency is small compared to the bandwidth terms and thus does not play a critical role.
- **o: Overhead.** In our context we define the overhead as the amount of time that the processor is busy initiating communication events.
- **g: Gap.** We define the gap to be the amount of time that the network is busy processing a communication initiation (i.e. time spent adding a communication event to the queue). This time can be overlapped with computation, however the message initiation has to wait until the network has finished queuing the previous message. Queuing a message does not mean that the message has been sent, it merely means that it is ready to be sent. Section 7.5 goes into more detail about how o and g are measured.
- **G: Inverse bandwidth.** We measure this directly through a bi-directional flood bandwidth test built into GASNet.
- **P: The number of processors.** This is an input parameter during runtime. For all our experiments and verification we use 32 processors.

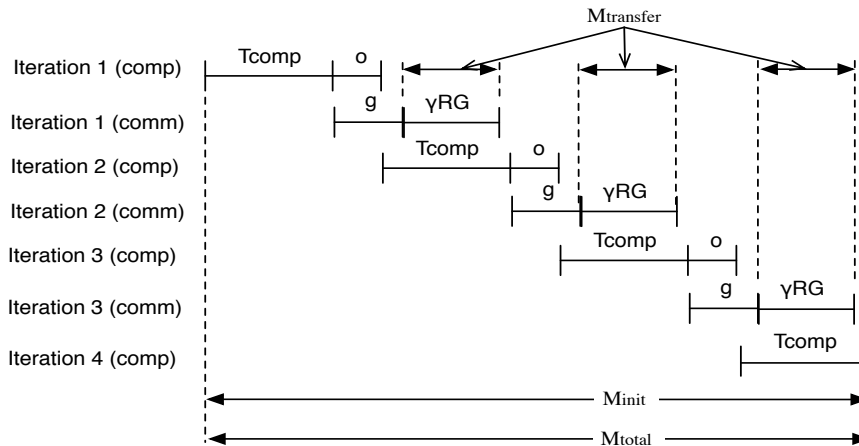


Figure 20. FT Model Case 1 ($N = P = 4$): Computation Time > Communication Transfer Time

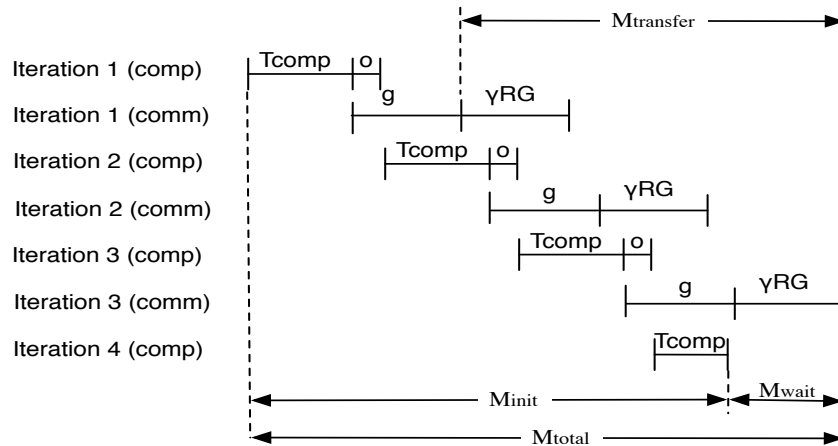


Figure 21. FT Model Case 2 ($N = P = 4$): Communication Transfer Time > Computation Time > Communication Initiation Time

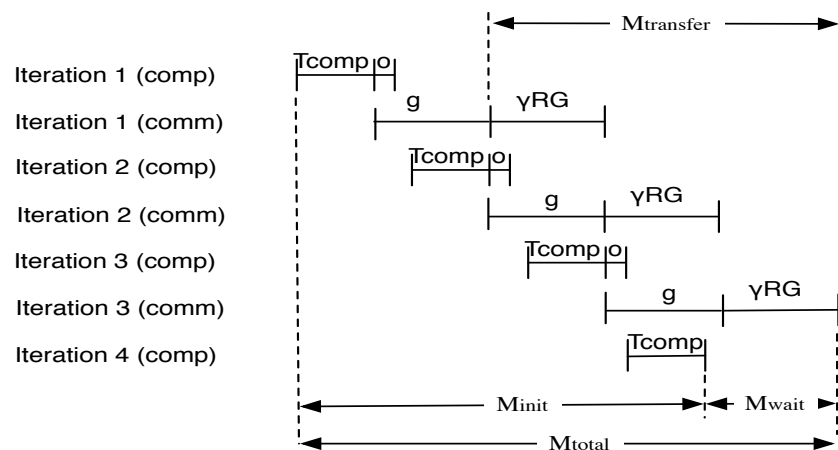


Figure 22. FT Model Case 3 ($N = P = 4$): Communication Initiation Time > Computation Time

7.3 The Three Cases

We present three flavors of the model, one where the computation is the dominant factor and the other two where the communication is the dominant factor. Since the models themselves are very similar they can be combined into a final model that will be used as the basis for understanding the performance.

- **Computation Time > Message Transfer Time**

The first performance model we show is where the computation ends up being the dominant factor rather than communication.

Figure 20 shows a Gantt chart for a single processor as it computes the different iterations of the computation and sends the data out ($N = P = 4$). We focus our analysis on *Pencils* since it is the more difficult case to model because of the amount of fine-grained communication/computation interaction. The total time in the initiation phase (i.e., the compute/put pipeline in lines 2-9 of *Pencils*) can be expressed as follows:

$$M_{init}^{comp} = [(T_{comp}^R + o) \times (P - 1) + T_{comp}^R] \times \frac{N}{P} \quad (1)$$

In our benchmark we have set up the communication such that the last operation will always be destined to the local processor and therefore does not incur any communication costs since it is purely a local memory copy. Since the computation is expensive, the network card will be able to process the next message transfer request as soon as the computation is done (i.e., $T_{comp}^R + o > g$).

Every processor has to communicate with $P - 1$ other processors and thus the total time to transfer the domain is:

$$M_{transfer} = [\gamma \times (P - 1) \times R \times G] \times \frac{N}{P} \quad (2)$$

We use γ to represent the number of bytes per double-precision complex word.

Since the computation is the dominant term, as shown in Figure 20, all the communication can be overlapped and thus the total time can easily be written as:

$$M_{total}^{comp} = M_{init}^{comp} \quad (3)$$

- **Message Transfer Time > Computation Time > Communication Initiation**

In the second case we create a set of models where the computation time is less than the message transfer time. However the computation time is still greater

than the time to inject messages into the network, as shown by Figure 21. Thus the total time to run the initiation phase is unchanged and can be expressed using Equation 1. As shown by Figure 21, when the data transfers become the dominant term then the total time is dominated by $M_{transfer}$. However, the transfer of the first message does not begin until the first computation is finished and thus this time needs to be included. The total can time can thus be expressed as:

$$M_{total}^{comm} = T_{comp}^R + \max(o, g) + M_{transfer} \quad (4)$$

- **Communication Initiation Time > Computation Time**

In the case where the communication initiation is the dominant factor we present a slightly different model. Figure 22 shows a Gantt chart for this case. The time for each component have been exaggerated to illustrate the concept and are thus not to scale. Unlike the previous case, here the message transfer initiation (and therefore the processor) has to wait until the network is free to accept another message before it can return and thus the time needed to push messages into the network queue becomes the bottleneck (i.e. $T_{comp}^R + o < g$). Notice that the gap term from one communication event is used as a measure of how long the next communication event has to wait before it can get started. Thus the last computation event does not need to wait for the network to become idle since there will be no subsequent communication initiations. From the chart we can construct an expression for the total initiation phase (i.e., lines 2-9 of *Pencils*) as follows:

$$M_{init}^{comm} = [T_{comp}^R + \max(o, g) \times (P - 2) + T_{comp}^R] \times \frac{N}{P} \quad (5)$$

The time taken to transfer the domain is still unchanged and can still be expressed as $M_{transfer}$ above. Since the message transfer time is again the overall limiting factor the total runtime can be expressed using Equation 4.

7.4 Combining the Three Cases

In order to combine the initiation models (Equations 1 and 5) we notice they are nearly identical and a simple $\max(T_{comp}^R + o, g)$ accounts for both cases. The total initiation time can thus be expressed as follows:

$$M_{init} = [(2 \times T_{comp}^R + o) + \max(g, T_{comp}^R + o) \times (P - 2)] \times \frac{N}{P} \quad (6)$$

By taking the maximum of Equations 3 and 4, we can get an approximation for the total running time as follows:

$$M_{total} = \max(M_{total}^{comp}, M_{total}^{comm}) \quad (7)$$

If M_{total}^{comp} is the larger of the two, then it means that the data transfer was completely overlapped with the computation / communication initiation pipeline. The sync operations (lines 10-12 of *Pencils*) will just perform routine housekeeping to reap expired handles (which, for simplicity, is modelled as instantaneous) and thus the wait time will be 0.

However, if M_{total}^{comm} is the larger of the two, it means that after all the computation and communication initiation time we have to wait for the transfers to finish before the benchmark completes thus the time spent in lines 10-12 of *Pencils* is actually the time for waiting for the network to finish its data transfers. Taking these two cases into account the total times spent in lines 10-12 of *Pencils* can be expressed as follows:

$$M_{wait} = M_{total} - M_{init} \quad (8)$$

Next we also notice that M_{init} takes into account both the time taken to perform the computation *and* initiate the communication events. Thus to separate the two phases out we subtract the total computation time from the total computation/communication initiation phase to yield the time the processor is busy waiting to put messages on to the network. Since the main focus of this paper is understanding network performance we use the computation time from measured data since building an analytic model for the local computation is outside the scope of this work. This is expressed as follows

$$M_{comm_init} = M_{init} - T_{comp}^R \times N \quad (9)$$

Thus akin to our previous presentations of the plots we have 3 different pieces to the execution time, the time spent in computation ($T_{comp}^R \times N$), initiating communication events (M_{comm_init}), and waiting for communication to finish (M_{wait}). We now verify the accuracy of the models and show that in practice they do a good job of predicting the various components of the benchmark.

7.5 Measuring Overhead and Gap

To measure the overhead we use the *Pencils* algorithm and measure the time taken in initiating communication events (timers started before and stopped after line 7 of *Pencils*). In addition we assume that we perform the computation on the full row (computation scale is 1) and therefore assume that this computation is larger than the communication initiation time (i.e., $T_{comp} + o > g$). Therefore the time spent in communication initiation can be an approximation of the

overhead. We thus take the total time spent in communication initiation and divide it by N to yield an average measure of per message overhead.

To measure the gap term we analyze the benchmark when there is no computation and use the communication initiation time as an approximation of the $\max(o, g)$. In addition all our experiments have shown that this value is *larger* than the o term measured from the above mentioned methodology. Thus we can deduce the following: since the value is larger, it must be a measurement of g . Again taking the total communication initiation time and dividing it by N yields an approximation for the average gap per message.

7.6 Model Verification: Many, Small Messages

Figure 23 shows the result of applying the model to the performance results presented in the previous section on the Opteron/Infiniband cluster. As before the x-axis shows the different computational scales while the y-axis shows the time in seconds. Each of the pieces shows a breakdown of where the time is spent in reality and the model. The actual data is shown by the columns marked “A” while the model is shown in the columns marked “M.” As shown, the simple models presented above do a fairly accurate job of predicting the performance of the different aspects of the benchmark and thus the overlap is proceeding as expected and modeled. One interesting factor from these sets of plots is that the communication initiation term drops significantly implying that at low computational scale (when the benchmark is network performance bound) the gap becomes the dominant factor but as we ease the bottleneck the impact of the gap term reduces and overhead plays a bigger role.

Figure 24 shows the same analysis for the Power5/Federation cluster. Again we see the same trends however the bandwidth noticeably too small since according to the model no time is needed waiting for communication to finish, however the measured data shows that there is some time waiting for these events. Since the Power5/Federation cluster does use the RDMA support (and since our models have assumed its existence) the difference in communication wait time is an approximation of the error due to this assumption. The communication initiation time, on the other hand, is an accurate measure of the experimental data.

7.7 Model Verification: Few, Large Messages

Figures 25 and 26 shows the same benchmark verification for a different communication pattern. Unlike the previous case, since we are sending a few, larger messages here, the total runtime is dominated either by the message transfer time or the computation time. Due to the low message count, the gap and overhead terms are negligible and thus the communication initiation time is also negligible.

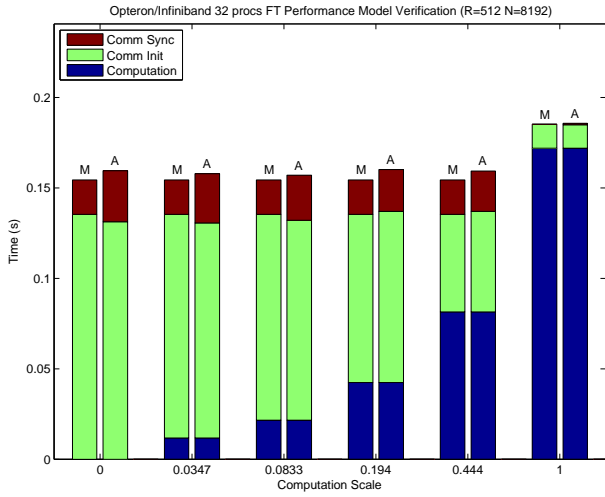


Figure 23. Opteron/Infiniband model verification: R=512, N=8192

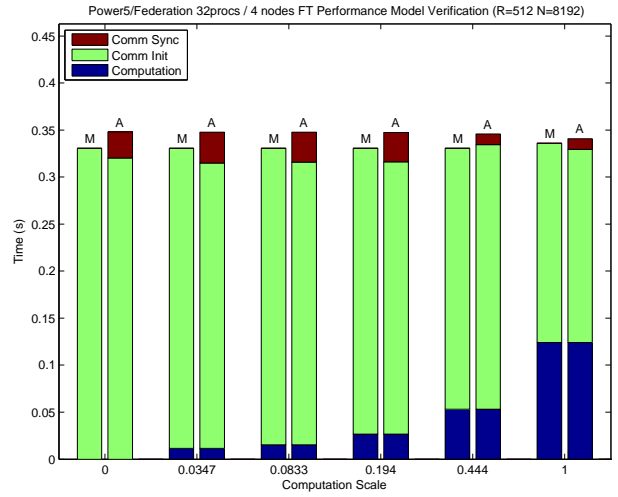


Figure 24. Power5/Federation Model verification: R=512, N=8192

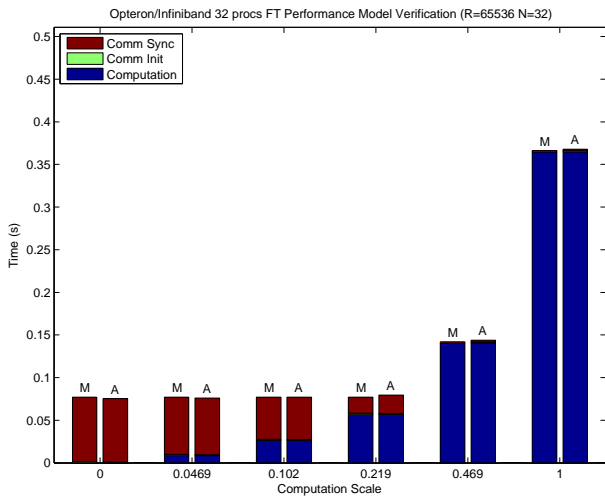


Figure 25. Opteron/Infiniband model verification: R=64k, N=32

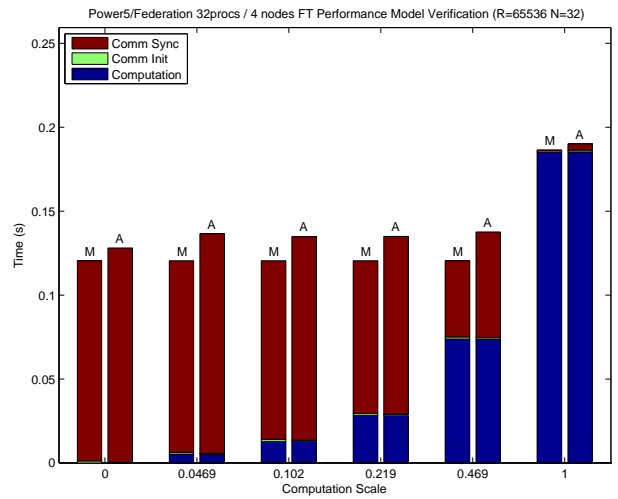


Figure 26. Power5/Federation model verification: R=64k, N=32

On both platforms, the bandwidth term does a good job of predicting the total runtime when communication is the dominant factor. On the Power5/Federation cluster the models under-predict the wait time. Again this is probably due to the model’s assumption of RDMA. Future work will examine effect this further.

8 Parallel SPMV Performance Models

In order to further understand the performance of the parallel SPMV we present of performance models in the LogGP framework.

8.1 The Model

Akin to the model presented in Section 7 we present a model that explores the case in which the most overlap of communication and computation occurs: when $\eta = 0$ and we alternate between issuing communication operations and performing computation. Recall that the SPMV algorithms require the communication operations to finish before the computation on those corresponding pieces can start. Fortunately, the computation on the first block can progress without any communication. As before the models can be broken into two different cases: (1) the computation time is greater than the communication time as represented by the Gantt chart in Figure 27 and (2) the communication time is greater than the computation time as shown by the Gantt chart in Figure 28. When the computation is the dominant term the computation iterations merely have to wait for the previous computation cycle to finish before. However when the communication is the dominant term, the computation must wait for the previous communication even to finish before the next cycle can start as shown in the figures.

Unlike the models in Section 7, the communication operations in this case are `gets` which incur two round trip latencies, one for the request and a second for the reply. Since we assume the network is capable of RDMA operations we only incur one overhead to initiate the `get` operation. There is no need to account for the overhead to transfer the message into the initiator’s memory since the network card can transfer the data directly without interrupting the CPU. Like the previous models, the time taken to transfer the message is accounted for by the inverse bandwidth term, G . For simplicity assume that we have an $N \times N$ sparse matrix distributed evenly by rows across P processors. In all our models $T_{comp}^{N/P}$ represents the time to perform the SPMV on one of the blocks of our matrix.

From the Gantt charts above we can write out a closed form solution for both the cases and then combine them into one case that we will call our model for parallel SPMV. In the first case when the computation is the dominant term all the communication, except the overhead costs to initiate the

`gets`, is overlapped behind the computation. Thus the total time for this can be expressed as follows:

$$M_{comp} = (o + T_{comp}^{N/P}) \times (P - 1) + T_{comp}^{N/P} \quad (10)$$

When communication is the dominant term, the performance can be written as follows:

$$M_{comm} = (o + 2 \times L + \gamma \times \frac{N}{P} \times G) + \quad (11)$$

$$(max(o, g) + 2 \times L + \gamma \times \frac{N}{P} \times G) \times (P - 2) + T_{comp}^{N/P} \quad (12)$$

Here γ refers to the number of bytes in a double-precision word. The maximum of the two models yields the total time that we expect the benchmark to take which is shown as follows:

$$M = max(M_{comp}, M_{comm}) \quad (13)$$

8.2 Model Verification: Opteron/Infiniband

As evidenced in Section 7.7, when there are a few, large messages (as is the case with our implementation of the parallel SPMV) the $L, o,$ and g terms are negligible since the total time is dominated by the transfer time. Thus our figures only show the *Computation* time and the *Exposed Communication* time. Initiation time is not separated out since it is miniscule compared to the other terms and does not reveal any new insights.

Figure 29 shows the application of the model to the Opteron/Infiniband cluster. The x-axis shows the non-zero ratio while the y-axis shows the time in seconds. For each computation scale there are three bars marked “M”, “E”, and “A.” The bars marked “A” show the actual data that was collected from the benchmark. The bars marked “M” (analytic model) and “E” (empirical model) show the application of the models. The overhead and gap terms are calculated as described in Section 7.1 while the latency term is taken from a latency benchmark in the GASNet test suite. The main difference between “E” and “M” is the methodology used to calculate the bandwidth term, G . In the bars marked “M” the bandwidth is measured through a GASNet bidirectional bandwidth benchmark using two processors. However it was found that for some cases the measured bandwidth was too optimistic to make any useful predictions. In order to get a more realistic measure of bandwidth when all 32 processors were issuing communication operations, a second bandwidth term is derived from the case when no computation is done. When no computation is done the benchmark degenerates into 32 processors simultaneously performing gather operations and thus places much more arduous demands on the network hardware. The bandwidth extracted from this instance of the benchmark is used to generate the

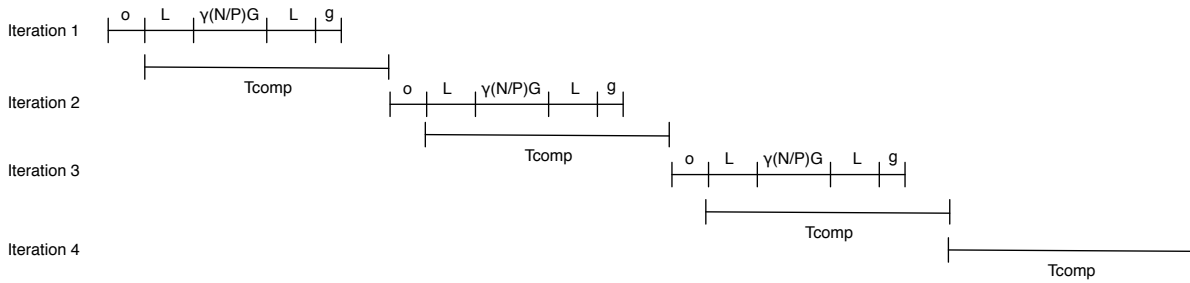


Figure 27. SPMV Model Case 1: Computation Time > Communication Time

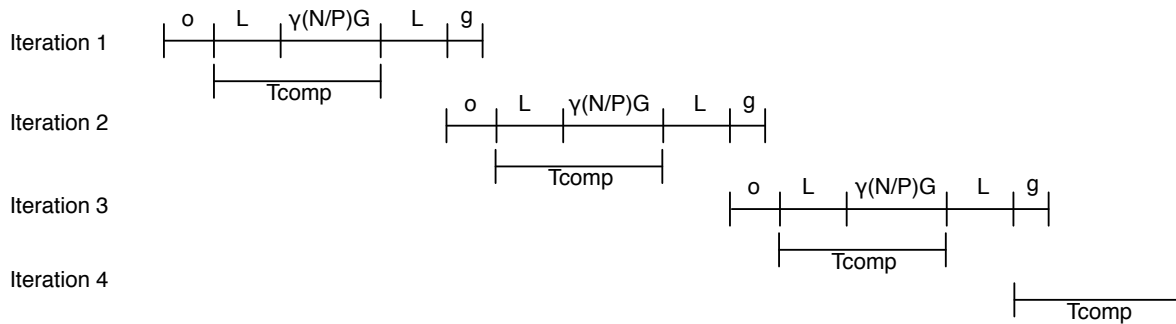


Figure 28. SPMV Model Case 2: Communication Time > Computation Time

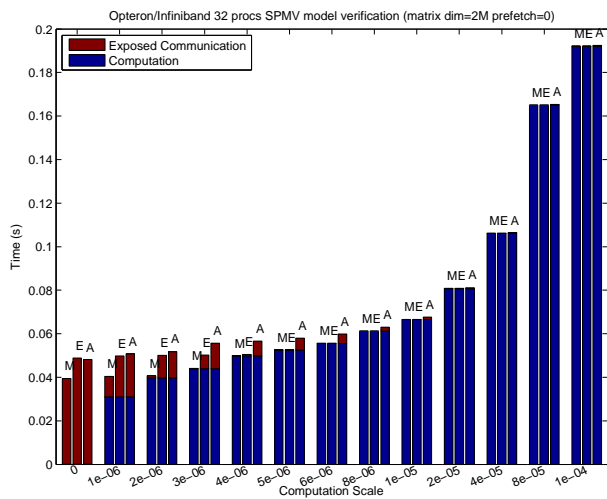


Figure 29. Opteron/Infiniband SPMV Model Verification Prefetch $\eta = 0$

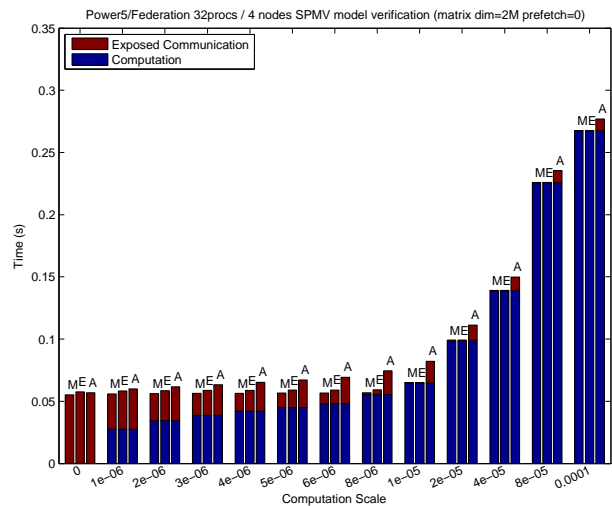


Figure 30. Power5/Federation SPMV Model Verification Prefetch $\eta = 0$

empirical model and the bars “E.” Like the previous models the computation time is taken as a given. To yield the time waiting for communication to finish, the computation time is subtracted from the modeled execution time. In some cases, when the computation is the dominant factor, the exposed communication is 0 indicating that all the communication is overlapped.

As Figure 29 shows the models do a relatively good job of predicting the actual performance, however at low computational scales, the exposed communication time is not predicted accurately. As computation becomes the more dominant factor the models accurately predict that the communication costs approach 0. These trends indicate that there are communication costs associated that are still not accounted for when the communication and computation time are. Future work will explore these discrepancies.

8.3 Model Verification: Power5/Federation

Figure 30 shows the application of the model to the Power5/Federation cluster. On this cluster, the bidirectional bandwidth test does a good job of predicting the actual bandwidth and thus the empirical model is not needed but is shown for completeness. Unlike the Opteron/Infiniband cluster, the models do a good job of predicting the performance at low computational scales but as the computation increases for a fixed communication volume, the models predict that all the communication will be hidden. However the actual data shows that this is not the case. There is still exposed communication indicating that the computation and communication are not fully being overlapped, as expected, since the RDMA capabilities are not being used. Again, future work will further explore these effects.

9 Related Work

Danalis et. al [13] have described techniques similar to our own to explicitly spread the communication across the computation to achieve better performance. However their main focus is on applications written for a two-sided model. Their techniques are similar to our *Slabs* and *Pencils* algorithms however they are target a two-sided communication model. This work extends the findings in [7] and leverages the performance of one-sided communication. Danalis et. al also show the advantages of using RDMA and communication-computation overlap and show how utilizing a lower level communication library can result in better performance. Their work focuses on compiler techniques to automate the process while this work focuses more on the effects of the granularity of overlap and the optimum communication/computation schedule. Our work demonstrates the effectiveness of these techniques on a variety of cluster interconnects and shows that this approach

scales to large processor counts and large problem sizes, further extending and validating their findings.

Previous work by Quinn [24] has shown that communication/computation overlap can, at best, achieve a factor of two speedup while in practice the speedup is a lot less. The models presented in that work are very similar to our work here, however we explore these results through actual experimentation and validate the models through experimental data. Our models also leverage the LogGP framework to fully express when and why communication / computation overlap is useful. De Cerio et al. [14] also present similar a performance analysis as our own, however their algorithms target hypercubes and thus the performance models do not target general clusters. Liu and Abdelrahman [21] have also discussed techniques of overlapping communication and computation on networks of workstations. Our work primarily analyzes modern networks that allow the overlap through efficient use of the network hardware.

Analyzing and optimizing the all-to-all communication pattern needed for a large parallel Fourier transform has been the subject of many papers. The analyses have ranged from modeling the performance on small commodity clusters [10, 16] through using highly specialized networks [11, 15, 25].

Iancu et. al [20] consider the benefits of breaking large messages into smaller ones to automate overlap, although their message segmentation is done implicitly by the compiler and therefore subject to the limitations of static analysis. This work is based on explicit overlap where the programmer directly expresses the data dependencies.

10 Conclusion

In this paper we have shown that overlapping communication and computation is often a good idea because the costs of communication can potentially be hidden behind the computation. However, we have also shown that, in order for this to be successful, the computation costs need to outweigh the communication costs, as expected. It is possible to hide the cost of the *fastest* component (either computation or communication) behind the other. In addition, we have also shown that the communication dependencies also affect the effectiveness of overlap. Parallel FT’s communication pattern has more flexibility in this regard than Parallel SPMV.

We have also drawn other conclusions as a result of these experiments. One result shows that overlapping communication and computation has the potential of slowing down the local computation as a result of memory bandwidth contention. Thus algorithms that heavily rely on the memory system for good performance will suffer due to this surprising result. We have also shown that networks capable of RDMA allow a much more fine-grained overlap of

communication and computation and realize the best performance results when the algorithms dictate the transfer of many small messages. However gains from overlapping communication and computation are not merely limited to the networks that support RDMA as shown with the results from the Power5/Federation cluster. We have also verified these conclusions by constructing performance models that accurately predict the communication performance of our application derived parallel benchmarks.

As the importance (and cost) of the network hardware within clusters grow, effective utilization of these network resources gains importance. Separating communication and computation into two distinct phases does not allow the cluster to use all of the available power to its full potential; either the communication subsystems or the computation subsystems are stressed while the other is idle. In addition, separating the phases into two distinct steps tends to create bottlenecks in the network, a problem that will get more acute as the number of processors within a cluster grows. These bottlenecks necessitate the design of algorithms that can efficiently overlap communication and computation so that applications can realize the full computational power of these new machines.

References

- [1] FFTW's FFT benchmark results. <http://www.fftw.org/benchfft/>.
- [2] Top500 List: List of top 500 supercomputers. <http://www.top500.org/>.
- [3] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d fft. In *SC*, pages 34–40, 1994.
- [4] A. Alexandrov, M. F. Ionescu, K. E. Schauer, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *J. of Parallel and Distributed Computing*, 44(1):71–79, 1997.
- [5] D. H. Bailey, et al. The NAS Parallel Benchmarks. *The Int'l J. of Supercomputer Applications*, 5(3):63–73, Fall 1991.
- [6] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *Communication Architecture for Clusters (CAC03)*, Nice, France, 2002.
- [7] 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.
- [8] The Berkeley UPC Compiler, 2002. <http://upc.lbl.gov>.
- [9] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.
- [10] S. Chalasani and P. Ramanathan. Parallel FFT on ATM-based networks of workstations. In *Proc. of the 6th Int'l Symposium on High Performance Distributed Computing (HPDC)*, 1997.
- [11] C. Y. Chu. Comparison of two-dimensional FFT methods on the hypercube. In *Proc. of Hypercube concurrent computers and applications*, pages 1430–1437, 1988.
- [12] D. E. Culler, A. C. Arpaci-Dusseau, R. Arpaci-Dusseau, B. N. Chun, S. S. Lumetta, A. M. Mainwaring, R. P. Martin, C. O. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Parallel Processing Symposium*, Kobe, Japan, 1997.
- [13] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *Supercomputing 2005*, November 2005.
- [14] L. D. de Cerio, M. Valero-Garcia, and A. Gonzalez. Overlapping communication and computation in hypercubes. In *Euro-Par, Vol. I*, pages 253–257, 1996.
- [15] L. Díaz, M. Valero-García, and A. González. A method for exploiting communication/computation overlap in hypercubes. *Parallel Computing*, 24(2):221–245, 1998.
- [16] P. Dmitruk, et al. Scalable parallel FFT for spectral simulations on a beowulf cluster. *Parallel Computing*, 2001.
- [17] S. Filippone and M. Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, Dec. 2000.
- [18] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.
- [19] W. Gropp, E. Lusk, and T. Sterling, editors. *Beowulf Cluster Computing with Linux*. MIT Press, 2nd edition, 2003.
- [20] C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks. In *Proc. High Performance Computing for Computational Science (VECPAR)*, 2004.
- [21] G. Liu and T. S. Abdelrahman. Computation-communication overlap on network-of-workstation multiprocessors. In *Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.
- [22] MPI-2: a message-passing interface standard. *Int'l J. of High Performance Computing Applications*, 12:1–299, 1998.
- [23] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick.

When cache blocking sparse matrix vector multiply works and why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.

- [24] M. J. Quinn and P. J. Hatcher. On the utility of communication-computation overlap in data-parallel programs. *Journal of Parallel and Distributed Computing*, 33(2):197–204, 1996.
- [25] P. Swartztrauber and S. Hammond. A comparison of optimal FFTs on torus and hypercube multicomputers. *Parallel Computing*, 2001.
- [26] UPC language specifications, v1.2. Technical Report LBNL-59208, Berkeley National Lab, 2005.
- [27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.
- [28] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.
- [29] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

Appendix: Platforms used in measurement

System	Processor	Network	Software	Location
Opteron/ InfiniBand	Dual 2.2 GHz Opteron (320 nodes 4GB/node)	Mellanox Cougar Infini- Band 4x HCA	Linux 2.6.5, Mellanox VAPI, MVAPICH 0.9.4, Pathscale CC/F77 2.2	NERSC Jacquard
Power5/ Federation	8-way 1.9 GHz Power5 64-bit (111 nodes 32GB/node)	One 2-link High Per- formance Federation Switch adapter	IBM AIX Version 5.2 ML 5, IBM LAPI, IBM C/C++ Enter- prise Edition V 7.0	NERSC Bassi

All platforms used Berkeley UPC v2.2.2 [8], with the appropriate GASNet v1.6 [9] native conduit.