

Algorithmic-Based Fault Tolerance for Matrix Multiplication on Amazon EC2

Grey Ballard, Erin Carson, Nick Knight
CS 262a Fall 2009

December 21, 2009

1 Introduction

Cloud computing presents a unique alternative to traditional computing approaches for many users and applications. The goals of this project were to assess the viability of the cloud for scientific computing applications, and to explore fault tolerance as a mechanism for maintaining high performance in this variable and unpredictable environment. Most previous attempts to run scientific computations on commercial clouds such as Amazon EC2 achieved poor performance, orders of magnitude slower than dedicated scientific clusters. Our experiments confirmed this observation, which we used to motivate the application of recent work in algorithmic-based fault tolerance to scientific codes. In this project, we focused on matrix multiplication, an operation ubiquitous throughout scientific computing algorithms.

Following the example of [BDDL08], we implemented a modified version of the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [GW97] using C and MPI (Message Passing Interface), which could simulate processor faults, coordinate the recovery of lost data, and complete its operation with a correct answer. We saw orders of magnitude degradation in the performance of SUMMA (with and without fault tolerance and faults) on EC2 versus a NERSC supercomputer, but we still believe that with the right development, the cloud can be an useful tool for scientific computing. In this paper we discuss our experiences with EC2, the performance experiments we conducted, and our implementation and experimentation with our fault tolerant matrix multiplication algorithm.

2 Amazon’s Elastic Compute Cloud

Amazon’s Elastic Compute Cloud (EC2) is a cloud-computing service that sells compute time on a per-hour, per-node (hereafter: instance) basis. Customers create and upload their own virtual machine images and spawn EC2 instances running these images. In our experiments, we used “small” instances, which provided a 32-bit platform, 1.7 GB of memory, and one virtual core with one “EC2 Compute Unit”¹. The main attraction of EC2 is its elasticity - the ability to quickly scale up (or down) the number of running instances as demand requires - which offers efficiency (in terms of time, effort, and money) over having to manage the actual hardware.

EC2 is primarily used for web services, which can benefit from EC2’s elasticity. We considered the possibility of using this computational resource to perform scientific computation. Scientific computing applications are CPU- and memory-intensive and typically run on dedicated computing clusters, such as distributed-memory supercomputers and networks-of-workstations (NOWs). These clusters are mostly static, so EC2’s elasticity is irrelevant. Overall, EC2 is a very different environment, with looser performance guarantees, longer and more variable communication times, and no control over the underlying hardware - in fact, running within a virtual machine, you cannot necessarily determine the actual hardware. The variability

¹“provides the equivalent CPU capacity of a 1.0 – 1.2 GHz 2007 Opteron or 2007 Xeon processor. This is also the equivalent to an early-2006 1.7 GHz Xeon processor” [Ama09]

in performance is briefly discussed below in Section 3.1 and Figure 1 and the variability in communication is discussed in Section 3.3 and Figure 5.

In our project, we built our own Amazon Machine Image (AMI) running Linux. With the help of some online tutorials and example code, we developed a set of Python scripts to launch multiple instances of this image and configure the instances to run MPI jobs as a cluster. This involved distributing MPI machine files with internal addresses and configuring ssh to accept MPI communications from these addresses. We installed Intel’s Math Kernel Library (MKL) [Int09], a mathematical software library providing an optimized BLAS (Basic Linear Algebra Subprograms) for Intel platforms. Although Amazon was careful not to specify that we were guaranteed an Intel architecture, `/proc/cpuinfo` consistently reported we were running on an Intel Xeon E5430 processor, and so we feel that our choice of MKL was appropriate (see Section 3.1 for performance results). We used MKL’s implementations of OpenMPI, ScaLAPACK (including PBLAS and BLACS), and BLAS for most of our experiments. This platform as described was representative of one used in scientific computing.

3 Performance Experiments

3.1 Sequential Matrix Multiplication

We performed experiments running sequential matrix multiplication algorithms to confirm the validity of Amazon’s performance guarantees. We obtained performance results for running a sequential DGEMM (level-3 BLAS General Matrix Multiply) algorithm on a single instance, as well as results comparing the relative performances of MKL, ATLAS (Automatically Tuned Linear Algebra Software) [Wha05], and the reference BLAS from the Netlib Repository.

For our first sequential DGEMM experiment, we emulated the local rank- k update operation that the blocked SUMMA algorithm performs at every step (see Section 4.2 for our experiments with the SUMMA algorithm), using MKL only. We held the block (local matrix) dimension constant at 1000, while varying the SUMMA panel size k from 100 to 1000. Our results demonstrated that for most panel sizes we maintained a flop rate of approximately 3.6 Gflops/second (billions of floating-point arithmetic operations per second). During one run of the experiment, however, we obtained significantly lower performance rates, around 1.6 Gflops/second. Results comparing this run with a run on a different node are shown in Figure 1. These data reflected the performance variability of the cloud, where an instance could perform significantly worse than other instances, despite running the same (sequential) code and having the same number of EC2 Compute Units. A single node with poor performance could negatively affect the runtime of a parallel algorithm, especially in the case where data dependencies and communication necessitate a higher degree of synchronization between nodes (cf. SUMMA algorithm, Section 4.2). Later in this paper, we will use this variability in sequential performance to help motivate the need for fault tolerance on the cloud.

Our next sequential experiment compared the relative performances of matrix multiplication routines in MKL, ATLAS, and the reference BLAS for various matrix sizes. Both MKL and ATLAS were tuned for the specific architecture at install time: both attempted to detect the hardware (looking at, e.g. `/proc/cpuinfo`) and loaded preconfigured BLAS libraries based on this information. ATLAS also performed an extensive parameter search to automatically tune the final BLAS routines, while MKL presumably was already tuned. Given that the installers only had access to the information available to the virtual machine (which may not have accurately represented the true hardware), it was unclear how well the tuning process performed. Furthermore, if this instance was subsequently run on a (physical) node with different hardware, then we could not expect the same performance as on the (original) architecture for which it was tuned.

In our case, we observed speedups for both ATLAS and MKL over the reference BLAS. Our results, shown in Figure 2, indicated that MKL outperformed both ATLAS and the reference BLAS, achieving around 3.8 Gflops/second. Considering that our one EC2 Compute Unit provided (‘the equivalent of’) a 1.0 – 1.2 GHz 2007 Intel Opteron, and taking into account the theoretical $4\times$ speedup we would have expected on such an Intel architecture ($2\times$ due to 2-way SIMDization, $2\times$ due to independent floating-point multiplication and addition pipelines), we noted our proximity to this peak performance of 4.0 – 4.8 Gflops/second.

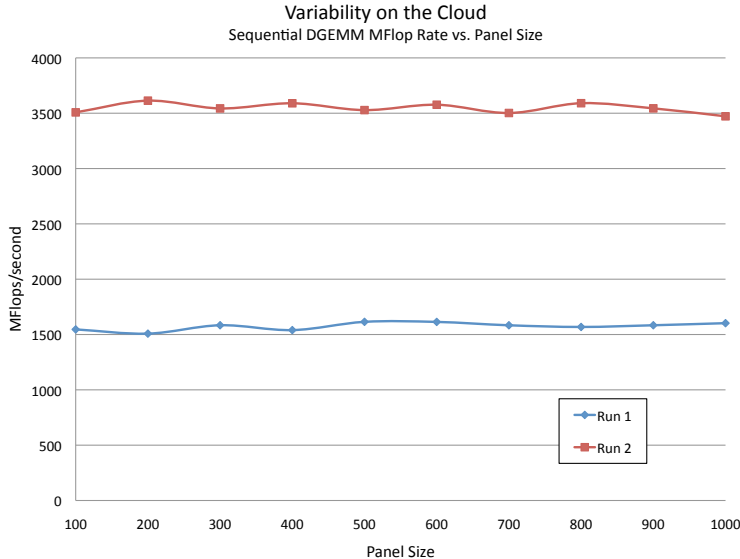


Figure 1: Performance comparison of MKL’s DGEMM on two different nodes at different times (Run 1 was ‘anomalous’, while Run 2 exemplified the typical performance).

3.2 Parallel Matrix Multiplication

We also performed experiments running parallel matrix multiplication algorithms, namely the SUMMA algorithm. A detailed discussion of this algorithm follows in Section 4.2. We used MKL to perform the local DGEMM operations, like those discussed above.

Operating on the same set of instances, we performed three runs with a constant global matrix size and panel size, as shown in Figure 3. This experiment primarily served to demonstrate the variability in parallel runtimes, although the underlying cause of this variability was more complicated than in the sequential case. It was likely a combination of sequential variability, like in Figure 1, and communication variability, as discussed below in Section 3.3. The slight decrease in runtimes might have indicated some (strong) scaling over this processor range, although this behavior was not consistent over larger processor ranges.

We also performed a weak scaling experiment on the SUMMA algorithm, shown in Figure 4, with a (local) block size of 500 and a panel size of 50. We witnessed a decline in performance (per node) up to about 36 processors, and roughly constant per-node performance (weak scaling) afterward. The initial decline indicated the communication cost of additional messages (more words moved overall) was impeding the ability of each processor to do a fixed amount of work, locally. The ‘flattening’ after $p = 36$ indicated that the additional work each processor contributed to the overall computation balanced the overhead of their presence.

3.3 Communication Benchmarks

Following [BDDL08], we hoped to model the performance of the SUMMA algorithm (see Section 4.2) using estimates for the time required to perform one flop, the (time) overhead of sending a message between two processors (latency cost), and the per-word communication time (bandwidth cost). The model assumed a constant α seconds was required to send any message, regardless of size, and another constant β determined the per-word cost of a message; thus a message of size n words required $\alpha + \beta n$ seconds to be sent from one processor to another. Given the size of the matrix, the size of the panel, the number of processors, and these hardware constants, one could have estimated the total time required to complete the SUMMA algorithm in this model. These hardware constants were known for the Jacquard supercomputer (at NERSC, Lawrence Berkeley Laboratory), and the model in [BDDL08] predicted the measured performance very accurately.

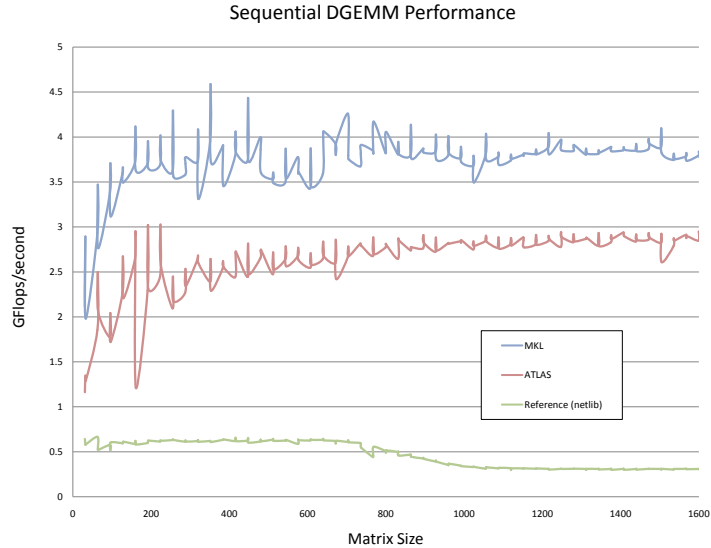


Figure 2: Performance comparison of three implementations of sequential matrix multiplication (DGEMM).

Since these hardware constants were not available on EC2, we sought to measure them. We found that the sequential computational performance of about 3.8 Gflops/sec, as seen above in Section 3.1, was achieved for sufficiently large matrix and panel sizes, but the results of our communication experiments were not as clear.

We first measured the bandwidth and latency costs separately, but did not find consistent values. For simplicity, since the SUMMA algorithm passed messages of only one size ², we decided to measure the cost of sending a message of a certain size. Our benchmark ran on an even number of processors by pairing up processors, passing a message of a set size size back and forth (‘ping-ponging’) within the pairs about 100 times, and computing an average, minimum, and maximum time for each pair. Figure 5 shows the data for messages with sizes ranging from 1 word to over 10 kilowords (words are 8-byte `double s`).

As shown in the graph, the cost per message for small messages was less than 5 milliseconds, while for larger messages (larger than about 2800 words), the cost was around 10 milliseconds, on average. The time to send a message did not change linearly with the size of the message, as our model predicted. This indicated that α and β could not be both approximated as constants with respect to message size. Furthermore, the data showed high variability - about a factor of four for all message sizes. The nonlinear behavior and high variability prevented us from accurately predicting performance using the linear latency/bandwidth model, although we conclude that the average times were all an order of magnitude slower than the results from [BDDL08] on Jacquard.

4 ABFT SUMMA

4.1 Algorithmic-Based Fault Tolerance

In a distributed system, it is desirable that an algorithm continue to make progress in the event that a subset of the processing units becomes unresponsive (a fault). In the case of EC2, a fault might manifest itself as an instance which becomes unresponsive to MPI communication, where ‘unresponsive’ could vary in definition from slow to completely offline. In more physical terms, a slow instance may be one hosted in a data center in a distant location, and a dead instance might be one who has been taken offline for maintenance. Given

²The fault tolerant version of the algorithm additionally passed messages of the size of a single integer to communicate fault information.

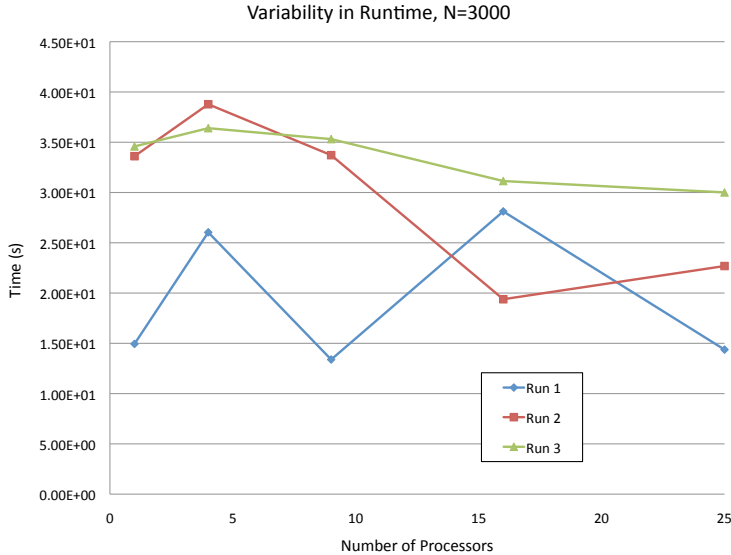


Figure 3: Variability on the cloud, SUMMA with constant global matrix size and panel size.

that the computational scientist has no control over their EC2 cluster’s hardware, their algorithms must be flexible enough to deal with this variability.

The traditional approach to fault-tolerance is the checkpoint/restart method, where a recoverable snapshot of the state of the computation is stored at certain points in time. This checkpoint data may be placed in nonvolatile storage, or the fault-tolerance scheme may be implemented in a diskless fashion, where the checkpoint data is distributed across the cluster nodes. However, these approaches do not scale - that is, since the probability of faults increases as the number of processors increases, the recovery time per processor should decrease accordingly - which is not the case.

In algorithmic-based fault tolerance (ABFT), the data needed to recover the state of the computation is calculated simultaneously, and in the same manner, as the algorithmic data. This approach opens the possibility of fault-tolerance that scales. [BDDL08] presented an implementation of ABFT PDGEMM, using SUMMA on a distributed-memory supercomputer, demonstrating this scaling.

We built upon this previous work by extending the ideas behind ABFT in HPC to cloud computing. As interest in cloud computing as a platform for scientific computing builds, it is necessary to evaluate whether the cloud can serve as an appropriate alternative to dedicated clusters. Previous experiments, including ours described above, have shown that performance on the cloud is inherently variable, and thus a faulty or slow processor (a straggler) could negatively affect performance, causing poor scaling relative to a cluster such as Jacquard. We argue that the extension of ABFT to include accounting for stragglers is an obvious addition for cloud computing. The experiments performed here serve as a proof of concept, demonstrating that ABFT for matrix multiplication on the cloud (accounting for stragglers) scales at a relatively similar rate to that on a scientific cluster.

4.2 SUMMA

SUMMA [GW97] is a parallel matrix-matrix multiplication algorithm that constructs the product $C = \alpha A \cdot B + \beta C$ through a sequence of rank- k updates. The underlying matrices are stored in a 2D block layout, and panels of A and B are passed along rows and columns, respectively, of the processor grid in ring broadcasts. These ring broadcasts can be implemented as a pipeline, where each processor receives a ‘panel’ of matrix A from its left neighbor, sends a panel of A to its right neighbor, receives a panel of B from its top neighbor and sends a panel of B to its bottom neighbor, at almost every step of the algorithm. This optimization reduces the communication cost from $O(n \log p)$ messages to $O(n + \sqrt{p})$ messages, where p is

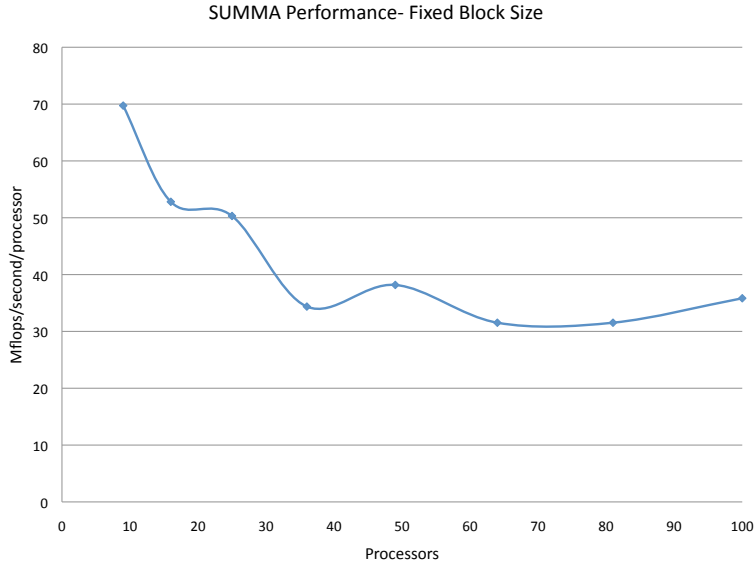


Figure 4: SUMMA weak scaling experiment, with local block size of 500 and panel size of 50.

the number of processors and n is the matrix dimension of A and B . We have assumed square matrices, a square processor grid, and rank-1 updates. In our SUMMA implementation, we use a larger panel size to take advantage of level-3 BLAS operations. This also decreases the total number of messages, while increasing the size of each message - so the total number of words moved remains the same.

4.3 Our Implementation

We implemented a FT-SUMMA (fault tolerant SUMMA) algorithm that simulated and handled faults. In our implementation, a “fault” was a processor that stopped normal execution of SUMMA, cleared its local memory (its blocks of A , B , and C , and its workspaces), and then resumed execution. In order to emulate FT-MPI (see Section 6.1), we dedicated one process in the MPI job as the “controller” which coordinated detection and recovery of a fault. This process simply listened for a message of a fault from all other processes, and, in the case of a fault, it notified all processes of the fault and of the identity of the faulty process. All other MPI processes ran the SUMMA algorithm with the following modifications:

1. one predetermined process at a predetermined step would “fail,” send a message to the controller, and wait for a message to recover and resume,
2. all processors “listened” for a message to recover from the controller each time they sent and/or received a message within the ring broadcast,
3. if necessary, processors executed recovery code to recompute lost data of A , B , or C matrices.

Because this was a pipelined algorithm, there were two types of recovery. When the faulty process lost its local data, it was restored by subtracting the sum of all other local matrices in the same processor row or column from the computed checksum in the last processor of that row or column. Since the input matrices A and B never changed, this recovery was straightforward: even if two processes were not at the same step of the algorithm, their local blocks of the input matrices were consistent. Thus, the input matrix data could be restored immediately upon receipt of recovery messages from the controller.

In the case of the C matrix, the recovery was more complicated. After the rank- k update at each step of SUMMA, the checksums of the output matrix were again consistent, maintained with no other work or communication by the design of the fault tolerant version of the algorithm. However, since the algorithm was pipelined, processors in a given row or column were not in lock-step. Thus, in order to recover C during the

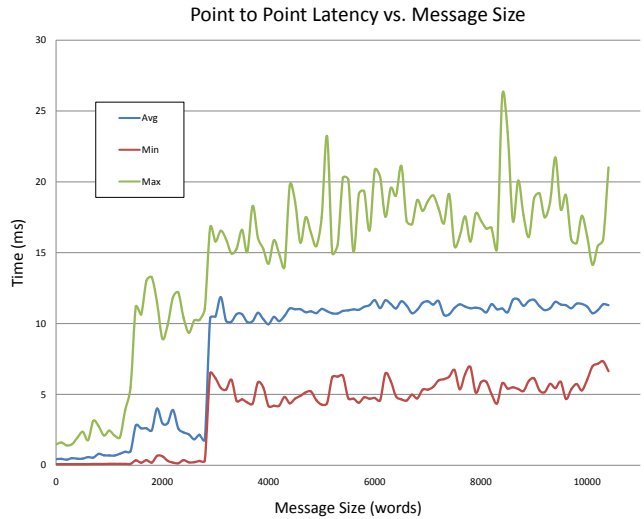


Figure 5: Time required to send messages of different sizes between pairs of processors; this experiment was run with 50 pairs of processors, communicating concurrently.

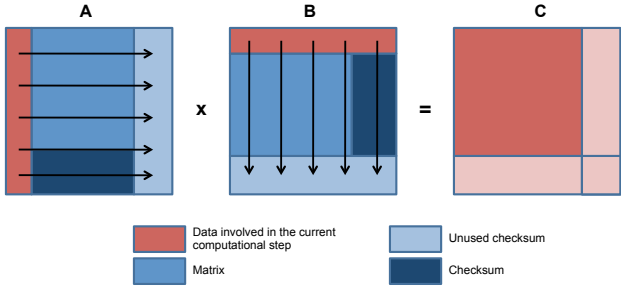


Figure 6: Modified matrix multiplication to allow fault-tolerance without checkpointing. Only the columns of A and the rows of B need checksums.

course of the algorithm, some processes would need to wait for others to catch up. When a consistent state was reached (that is, when the processes in the same row or column as the faulty one all reached the same step), C could be recovered and the algorithm could be resumed. Note that this was effectively equivalent to emptying the pipeline and refilling it in the middle of the run (for performance implications, see Section 4.4).

For recovery, two values had to be known to all processes. First, each process needed to know the ID of the faulty process in order to decide whether or not to participate in the recovery phase (only those in the same row or column as the faulty process were needed). In our implementation, when the controller sent out notifications to recover, it included the ID of the faulty process. Second, each process involved in the recovery needed to know at which step to stop and recover C . We implemented this as an `MPI_Allreduce`; the latest step among all processes was determined so that recovery could happen as soon as possible without any process having to backtrack and then redo work.

Note that we could have reduced the overhead of emptying and refilling the pipeline by allowing the entire matrix multiplication operation to finish before the C matrix was recovered. After the fault, all updates made by the faulty process to its local block of C would have been garbage, but we could have recovered the correct values from the other processes at the end of the algorithm when we achieved the (final) consistent state (thereby emptying the pipeline only once). However, delaying the recovery in this manner would not have worked in certain (unlikely) cases. Since we could recover information via process rows or columns, we

could recover any three faulty processes. See Figure 7 for an example of four process failures that cannot be recovered at the end of the algorithm. As our implementation recovered C during the algorithm, it could recover from arbitrarily many failures, as long as they did not occur simultaneously or during each other’s recovery phases. In general, ABFT can be used to recover from simultaneous faults and faults that occur during each others recovery phases, so long as they do not fall into a configuration like Figure 7.

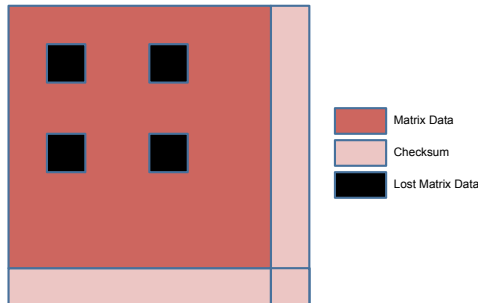


Figure 7: A configuration of four faults where C needed to be recovered during the course of the algorithm, as opposed to at the end. In the case of an ABFT implementation that supports simultaneous faults, this configuration is irrecoverable if the faults occur simultaneously or during each other’s recovery phases.

4.4 Results

After extensive testing to ensure the correct functionality of our algorithm under a variety of situations, we ran experiments to determine the overhead in handling a fault. In one run, our FT-SUMMA algorithm had no simulated faults. In the second run, we simulated a fault at a predetermined node and step within the algorithm. We compared the performance with the standard SUMMA algorithm. Figure 8 gives a graphical view of the performance overheads of our fault-tolerant implementation in the cases of no fault and one fault, relative to the standard algorithm.

Although we did not undertake thorough error analysis, we noted that after recovering from a fault, the matrix multiplication algorithm was no longer backwards stable in a componentwise sense. If an element of a block lost in a fault was much smaller in magnitude than the other elements summed to obtain its checksum, then a large relative error could be committed in the recovery of this element. However, the error was still bounded in a normwise sense. Typically, the result of a matrix multiplication would be then passed to some other linear algebra routine, and the algorithms in LAPACK (including solving a linear system) are provided only with normwise error bounds, so we argue that the loss of componentwise stability should not be an issue for most users.

The main performance overhead incurred by the fault-tolerant algorithm (even in the absence of faults) was the dedication of some processors to maintaining the checksum data. That is, given p^2 processors to perform a matrix multiplication, choosing to run the fault-tolerant algorithm implied that that user could expect performance no better than running the standard algorithm with $(p - 1)^2$ processors. In addition to this sacrifice in performance, the ring broadcasts needed to include one extra message passed (a widening of the pipe). As mentioned in Section 4.1, this relative overhead decreased as the number of processors increased (this characteristic distinguishes ABFT from diskless checkpointing methods). In Figure 8, this difference is represented by the widths of the pipes (fatter pipe implies shorter length).

We also expected some overhead in the computation of the checksums of A , B , and C before the algorithm began. Computing checksums required little extra arithmetic (a lower order term), but also required reductions over rows and columns of processors (the SUMMA algorithm relies only on nearest-neighbor communication). We included the cost of computing the checksums in our experiments, but as pointed out in [BDDL08], assuming the matrix multiplication was part of a larger fault-tolerant numerical algorithm, the checksums would have already been computed before the matrix multiplication subroutine is called. This overhead is shown at the top of the pipes in Figure 8.

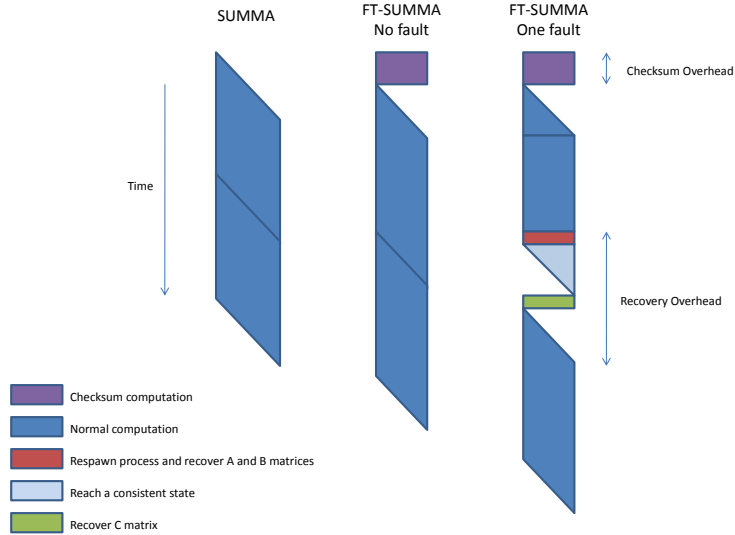


Figure 8: Performance overheads of fault-tolerant algorithms relative to standard SUMMA.

In comparing the performance between FT-SUMMA with no faults and FT-SUMMA with one fault, we expected a fairly constant gap in performance. Recovery after a fault required the controller to send notifications to all processors, the processors in the fault column performed three reductions to recover the lost data, and the pipeline needed to be filled and emptied (in order to reach a consistent state for C recovery). In the case of multiple faults, we expected the gap between the performance of FT-SUMMA with f faults and FT-SUMMA with no faults to be f times the gap of FT-SUMMA with one fault. This is labeled as “Recovery Overhead” in Figure 8.

Our measured performance is given in Figure 9. We fixed a local block size of 500 and used a panel size of 50 for all fault tolerance experiments. For a fair comparison, for standard SUMMA, we assigned smaller local blocks to the same number of processors so that the global matrix sizes of the multiplication were the same for each of the three runs.

Our goal was to reproduce the comparison given in [BDDL08]. While the absolute performance is orders-of-magnitude slower from that achieved on Jacquard, the relative behavior is similar. In the case of FT-SUMMA, both with no fault and one fault, we saw slightly better-than-weak scalability (the overhead decreased as the number of processors increased). The difference between no faults and one fault was small and nearly constant for all numbers of processors. The performance of standard SUMMA was slightly unexpected. We did not expect standard SUMMA to run an order of magnitude faster than the fault tolerant versions. As discussed in Section 3.2, the algorithm should have been weakly scalable, but we did not achieve this on EC2 (per node performance was not constant for fixed local block size). We expected the standard SUMMA to outperform the fault-tolerant versions slightly, and for that gap to decrease as the number of processors increase (this behavior is modeled and measured in [BDDL08]). We gathered data for these experiments on different cluster instances, and we believe this is the main reason for the unexpected performance. One frustration of measuring performance on the cloud was the lack of repeatability, especially for algorithms requiring inter-instance communication. The only other explanation for this gap was the cost of computing the checksums before performing the matrix multiplication, but we did not believe that could account for an order-of-magnitude difference.

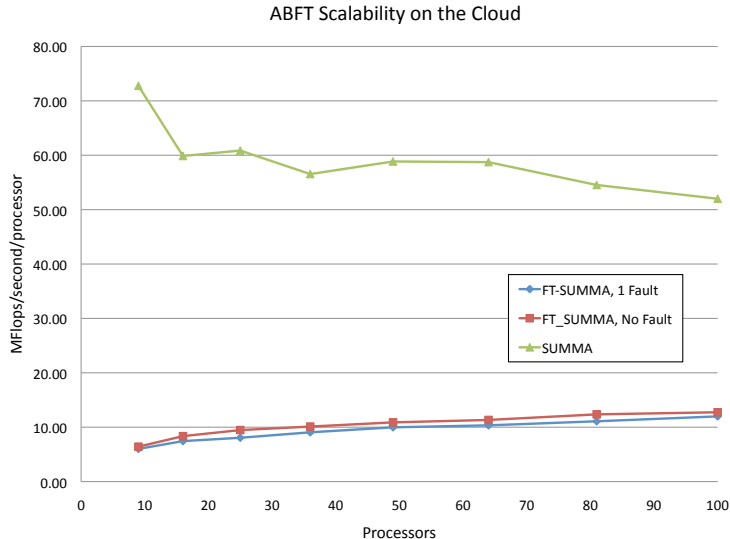


Figure 9: Weak scalability experiment for standard and fault tolerant SUMMA. The global matrix size was given by 500 times the number of processors in a column storing true matrix data.

5 Discussion

Our work aims to answer the question, ‘is cloud computing a suitable platform for scientific computing?’ Our results demonstrated that scientific computing applications run on EC2 achieved at least an order-of-magnitude worse performance than the same application run on a dedicated scientific cluster, such as Jacquard [BDDL08]. These findings are similar to those reported in [Wal08], which notes a significant gap between performance on EC2 versus an NCSA cluster. We believe that these seemingly pessimistic results are not due to the nature of cloud computing in general, but can instead be attributed to the service offering constraints of EC2.

Amazons EC2 is built primarily for business purposes and web-based applications. The key feature that makes EC2 attractive in this sense is its elasticity. Many of those using Amazons EC2 only rent nodes when their web server fails to handle peak demand, at which time work can spill over into the cloud. For databases and web servers, the variability and performance in bandwidth and latency currently offered by EC2 is sufficient. For scientific computing purposes, however, commercial clouds have not yet reached maturity. As scientific computing on the cloud has failed to gain mass attention thus far, Amazon has no motive to extend and adapt their service offerings.

Of course, viable alternatives to commercial clouds are emerging. Scientific compute clouds are being installed at many universities, including University of Chicago’s Nimbus and the University of Florida cloud [KF08]. Although some scientific applications have been run on these clouds [MTF08], their use is still considered experimental.

Our work leads us to a similar conclusion as Walker [Wal08]: although cloud computing remains an attractive idea for scientists who do not own or have access to a dedicated cluster, the infrastructure and services currently offered by cloud computing are not mature enough to serve as a practical alternative. In order for scientific cloud computing to reach its full potential, either science clouds must develop and separate from “business-oriented” clouds, or commercial clouds, such as EC2, must expand their services, specifically in the realm of high-performance networks.

Until further development efforts materialize, our fault tolerant algorithms, which consider “stragglers” to be faults, can help improve performance and reduce variability for scientific applications. If particular nodes in a simulated MPI cluster are performing below par compared to the rest of the cluster, our algorithm will handle reassigning the data to a new node in the middle of a computation. Our algorithm also, of course,

handles the case where a node times out or becomes unavailable, which would otherwise cause the entire computation to fail. Tuning to find optimal processor layouts, discussed in the next section, can further work to increase performance by optimizing communication between adjacent nodes.

6 Future Work

The results and insight gained thus far serves as a starting point for further exploration of cloud computing and the feasibility of its use for scientific computing. Work to be done in the near future includes incorporating libraries to enable our code to recover from real faults, benchmarking and comparing results for instances with higher compute power and larger memory, and exploring how tuning can be used to optimize communication within the cluster.

6.1 Fault Tolerant MPI

Fault Tolerant MPI (FT-MPI) [Bos03] is an MPI implementation, built on the HARNESS runtime system, which provides process-level fault tolerance for MPI applications. In non-FT-MPI implementations, the failure of one node will cause the failure of all nodes. FT-MPI addresses this issue, and is able to survive the failure of $n - 1$ processes in an n -process job. Although the FT-MPI API provides the ability to respawn the failed processes, the application must handle recovering lost data. In the future, we will adapt our fault tolerant algorithm to make use of the FT-MPI library, so that recovery from actual failures (instead of just simulated ones) is possible.

6.2 Larger Instances

Although Amazon offers many instance types, we restricted our study to “small” instances for consistency and simplicity. It was advised [McC09] that the “large” instance type typically performs better for MPI communications due to better I/O performance. We also note that our local physical memory was capped at 1.7 GB, which limited our local matrix dimension to about $N = 7000$. Increasing our local memory at each node by switching to a ‘larger’ instance would allow us larger local matrix blocks, that is, we could multiply larger matrices with a given number of processors and the same number of messages (although they would have more words per message).

For a given matrix size, inter-node communication can be minimized by using much larger local matrix blocks, stored both in-core and on disk. The I/O latency from disk to RAM to cache is certainly much less than the (approximately) 5 ms latency between EC2 nodes we observed. We could further minimize communication by using instances with multiple cores at each node. This hybrid architecture would allow more local work for a constant global communication. Further exploration of the performance implications of using various instance types offered by Amazon is needed.

6.3 Autotuning

Given the performance variability between pairs of nodes, it makes sense to have adjacent nodes in the 2D torus-wrap layout assigned to the node pairs which communicate the fastest (likely, located in closer physical proximity). Our implementation currently uses the default MPI behavior and assigns nodes to ranks in a communicator based on ordering in the machine file. If we instead organized our communicators to take the relative inter-node response times into account, we expect better overall communication, and thus better performance. This tuning would likely occur only once, before a computation and/or during cluster boot-up and configuration.³

6.4 ABFT-BLAS

Our work only concerns matrix-matrix multiplication. ABFT should be extended to the rest of the PBLAS. From this building block, ScaLAPACK could be reconstructed to make use of ABFT. It is also possible to

³Of course, if a fault occurred and a new node was substituted, the tuning might become suboptimal.

introduce ABFT at a higher level we refer to an example of ABFT LU decomposition [LP88] although LU (LAPACK DGETRF) uses BLAS operations like DGEMM (in the blocked right-looking variant), which could themselves be ABFT, as in our exposition.

References

- [Ama09] Amazon.com. Amazon Web Services documentation, 2009. <http://aws.amazon.com/documentation>.
- [BDDL08] George Bosilca, Remi Delmas, Jack J. Dongarra, and Julien Langou. Algorithmic based fault tolerance applied to high performance computing. Technical Report 205, LAPACK Working Note, June 2008.
- [Bos03] George Bosilca. Harness fault tolerant mpi, 2003. <http://icl.cs.utk.edu/harness>.
- [GW97] Robert A. Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.
- [Int09] Intel. Intel math kernel library reference manual, 2009. <http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf>.
- [KF08] K. Keahey and T. Freeman. Science Clouds: Early experiences in cloud computing for scientific applications. *Cloud Computing and Its Applications*, 2008.
- [LP88] Franklin T. Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5(2):172 – 184, 1988.
- [McC09] Alex McClure. Personal communication, 2009.
- [MTF08] Andrea Matsunaga, Mauricio Tsugawa, and Jose Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. *In Proceedings of the 4th IEEE International Conference on e-Science*, 2008.
- [Wal08] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *login: The Magazine of USENIX and SAGE*, 33(5):18–23, 2008.
- [Wha05] Clint Whaley. Atlas homepage. <http://math-atlas.sourceforge.net/>, 2005.