Highly Latency Tolerant Gaussian Elimination

Toshio Endo
University of Tokyo
endo@logos.ic.i.u-tokyo.ac.jp

Kenjiro Taura
University of Tokyo/PRESTO, JST*
tau@logos.ic.i.u-tokyo.ac.jp

Abstract

Large latencies over WAN will remain an obstacle to running communication intensive parallel applications on Grid environments. This paper takes one of such applications, Gaussian elimination of dense matrices and describes a parallel algorithm that is highly tolerant to latencies. The key technique is a pivoting strategy called batched pivoting*, which requires much less frequent synchronizations than other methods. Although it is one of relaxed pivoting methods that may select other pivots than the 'best' ones, we show that it achieves good numerical accuracy. Through experiments with random matrices of the sizes of 64 to 49,152, batched pivoting achieves comparable numerical accuracy to that of partial pivoting. We also evaluate parallel execution speed of our implementation and show that it is much more tolerant to latencies than partial pivoting.*

1 Introduction

Grid technologies [7] enable users to integrate widely distributed computing resources for running CPU intensive applications on them. This approach has been successful especially for applications that involve little interactions among subtasks [1, 2].

Seeing this success, it is attractive to attempt to run communication intensive applications such as matrix operations and PDE solvers on Grid environments, while they have been mainly run on supercomputers or clusters. This approach is supported by recent development and experiments of programming tools designed for Grid [4, 6, 11, 14]. However, there are still several obstacles for such applications to achieve sufficient performance: lower bandwidth of wide area network (WAN), volatility of computing resources, larger communication latency over distributed sites, and so on. While shortage of bandwidth and volatility will be solved in the near future by innovation of network architectures and programming tools for Grid, *large latencies* will remain problematic. Typically the latencies over WAN are the order of ten milliseconds and sometimes exceed 100 milliseconds, while those of supercomputers are several microseconds. In order to exploit Grid environments for communication intensive applications, we require proper design and implementation of algorithms that tolerate large latencies.

This paper takes one application, parallel Gaussian elimination of dense matrices for solving linear equations. It is one of applications that require frequent interactions among computing nodes, and is used as a popular benchmark for parallel computers[3]. We have previously described an implementation of Gaussian elimination that is tolerant to volatility of resources and large latencies[5]. The implementation was, however, numerically unstable because it was not equipped with any pivoting. When partial pivoting is implemented for improving numerical accuracy, we have noticed that tolerance for latencies are severely degraded because pivoting introduces tight data dependencies. With the order of ten milliseconds latencies, the critical path caused by partial pivoting is so long that even ideal pipelining techniques may fail to hide it.

This paper presents a latency tolerant Gaussian elimination algorithm by introducing more relaxed pivoting method, named *batched pivoting*. The basic idea of batched pivoting is to reduce frequency of synchronizations for pivot selection; each node independently selects candidates for pivots of several contiguous steps and then the candidates are gathered. This method largely reduces the critical path length, thus it makes the algorithm latency tolerant. In addition to tolerance to latencies, this paper evaluates numerical accuracy. And we show that batched pivoting achieves both tolerance to latencies and good accuracy.

Section 2 describes a traditional Gaussian elimination algorithm with partial pivoting. Section 3 mentions existing pivoting methods and their problems.

*Japan Science and Technology Agency

Section 4 proposes our batched pivoting and Section 5 describes the details of our implementation. Section 6 evaluates the implementation both in tolerance to latencies and numerical accuracy. We discuss applicability of our method in Section 7 and conclude in Section 8.

2 Gaussian Elimination with Partial Pivoting

Gaussian elimination with partial pivoting is a traditional method for solving dense linear equations. For an $(n \times n)$ matrix A, it solves $Ax = b$ with $(2/3)n^3 + O(n^2)$ floating point operations. We describe the algorithm briefly and show that its performance heavily suffers from large latencies.

2.1 Algorithm

Figure 1 shows an outline of the serial algorithm. Each step of the outermost loop consists of pivoting phase, row exchange phase and update phase. In pivoting phase of the kth step, we examine the kth column and find an element called the *pivot*, whose absolute value is largest. Then we exchange the row that includes the pivot and the kth row. In update phase, we update the lower right $(n-k-1) \times (n-k-1)$ submatrix by using elements in the kth column and new kth row.

Selecting a large element as a pivot is important for numerical accuracy. In update phase, elements are updated by using the value $a_{ik}a_{kj}/a_{kk}$, where a_{kk} is the pivot. If a_{kk} is closer to zero, the results of update phase get larger, thus numerical accuracy is degraded.

```
for (k = 0; k < n; k++) {
    /* pivoting */
    finds pivot element a_pk in the kth column
    /* row exchange */
    exchanges the p th row and the kth row
    /* update */
    for (i = k + 1; i < n; i++) {
        for (j = k + 1; j < n; j++) {
            a_ij = a_ij − a_ik a_kj / a_kk
        } } }
```

Figure 1. An outline of Gaussian elimination with partial pivoting.

2.2 Problems with Large Latencies

In parallel implementations, such as High performance Linpack (HPL)[12], a large number of optimization techniques have been invented. Blocking technique, which aggregates update phases of several steps, reduces frequency of communication and cache misses. Pipelining technique, which invokes pivoting phase before update phases of preceding steps are completed, effectively hide costs for pivoting phase when latencies are small.

On Grid environments, however, we may fail to hide pivoting phase even with these techniques, because the critical path is too long. In a typical data mapping, called two dimensional block cyclic mapping, each column is partitioned among several computing nodes. Thus each pivot selection requires synchronization among the nodes, whose costs are determined by latencies. To make matters worse, pivoting phases of different steps cannot overlap or be batched, since each pivoting phase depends on the results of all preceding pivoting phases.

In summary, the length of the critical path caused by pivoting phase is at least $O(nl)$, where n is the size of the matrix and l latency. This may dominate the whole execution time, if latencies are so large. Figure 2 shows a rough estimation of the limit of achievable speed with latencies of 0.1 ms, 1 ms, and so on [1]. According to the graph, with latencies of 10 milliseconds, the speed never exceeds 66.7TFlops even when matrices are of sizes of 10^6. Considering that IBM Bluegene/L supercomputer has achieved 136.8TFlops with $n = 1,277,951$ (cf. Top500 list in June 2005), we see that we cannot achieve similar performance to supercomputers on Grid, even with very large number of nodes.

A naive method to reduce the critical path length is to abandon two dimensional block cyclic mapping; instead, we map each column onto a single node. Thus pivoting phase at each step can be conducted locally. This method is, however, undesirable because it heavily increases communication; the total amount of communication is $O(n^2p)$ where p is the number of nodes, while it is $O(n^2\sqrt{p})$ in two dimensional block cyclic mapping. Although we could map each column onto a single cluster, where synchronization costs are low, it is inapplicable if nodes are placed distantly from each other as in desktop Grid.

[1] It is an optimistic estimation that simply compares nl and $(2/3)n^3/s$, where s is the total CPU speed in Flops. The limit would be worse if we take factors into account such as effects of reduction algorithm

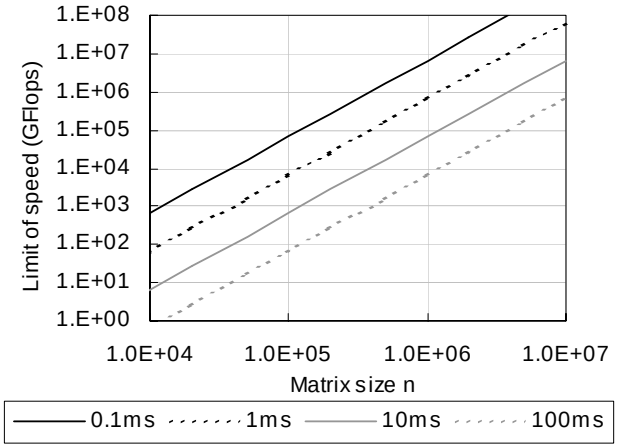

Figure 2. An estimation of limit of achievable speed in partial pivoting method.

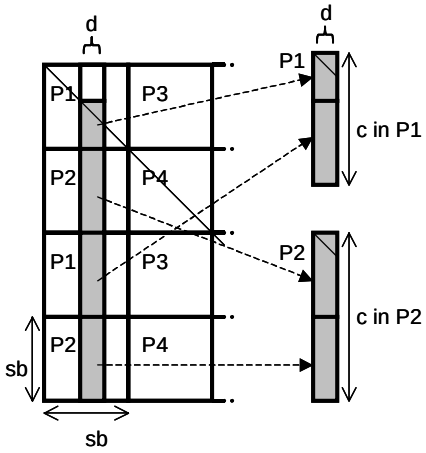

Figure 3. An illustration of batched pivoting. Each of nodes that owns part of grayed columns (P1 and P2) performs Gaussian elimination locally by using its own submatrix.

3 Related Work

Several pivoting methods other than partial pivoting have been proposed in various contexts. This section describes some of them and shows that they have problems either in tolerance to latencies or in numerical accuracy.

In *threshold pivoting*[15, 10], the largest value in absolute may not be selected as a pivot. An element a_{pk} can be selected if it satisfies $|a_{pk}| \geq \tau \max_{i \geq k} |a_{ik}|$, where $0 \leq \tau \leq 1$ is a predefined parameter. Since there may exist several elements that satisfy this condition, threshold pivoting can select one of them arbitrarily; it usually selects one so that the amount of communication for row exchange is reduced. However, this method is not latency tolerant because it still requires synchronization at each step to obtain $\max |a_{ik}|$.

Unlike methods described so far, *pairwise pivoting* [13] does not select a single pivot for each step. Instead, it repeatedly takes adjacent two rows and eliminates one of them. This method achieves good tolerance to latencies, since each row can start an elimination step immediately after it and its neighbors finish the previous step. On the other hand, it is inferior in numerical accuracy as shown in Section 6.

In the next section, we will describe a pivoting method that achieves both tolerance to latencies and numerical accuracy.

4 Our Pivoting Method

For the purpose of making Gaussian elimination highly tolerant to large latencies, we present an alternative pivoting method named *batched pivoting*. This method avoids frequent synchronizations by a simple idea; we aggregate pivoting phases of several contiguous steps. We hereafter let d the number of pivots selected in a batch. Each node locally selects candidates for pivots of d contiguous steps and then the candidates are gathered. While batched pivoting is more tolerant to latencies, it is not as stringent as partial pivoting; it sometimes selects inferior pivots. However, as we will show in Section 6, batched pivoting achieves comparable accuracy to partial pivoting.

Here we discuss pivot selection of $[k, k+d)$th steps. The d pivots are selected from $[k, k+d)$th columns, which are grayed in Figure 3 (Part upper than the kth row is not grayed because it is not used for pivot selection). In this case, nodes P1 and P2 are involved in the selection, since grayed submatrix is partitioned among these nodes. We hereafter call them *sharing nodes*. Then each sharing node conceptually owns a submatrix that is a subset of grayed part, as in the right part of the figure. We let c the number of the local submatrix (c may be different by nodes).

The following describes how we determine pivots of $[k, k+d)$th steps.

1. Each sharing node determines a set of d *candidates*

for pivots as follows. The node locally performs Gaussian elimination with partial pivoting by using its $(c \times d)$ submatrix. This computation is a speculative one; the original matrix should not be overwritten. So each node duplicates this submatrix in a temporal buffer and performs the local Gaussian elimination there. During the computation, each node finds d pivots. The node records them as candidates for pivots.

2. By using the set of candidates, each sharing node calculates a *score* for the set, which is defined as $\min_{k \le k' < k+d} |p_{k'}|$, where $p_{k'}$ is a candidate for pivot at the k'th step.

3. After all sharing nodes determine their sets of candidates and their scores, we gather them. Then we select one of candidate sets that has the largest score; all d candidates included in the selected set are adopted as *final pivots*, and they are sent to all sharing nodes. Now we have obtained d final pivots.

Batched pivoting reduces frequency of synchronizations because they occur at every d steps, not at each step. Thus the critical path length is reduced to $O(nl/d)$ and tolerance to latencies is improved. Note that, however, worse pivots may be selected than in partial pivoting. Batched pivoting constrains the selection of pivots in such a way that pivots of d consecutive columns must be taken from rows owned by a single node. In partial pivoting, each pivot is selected independently from the whole column. In spite of this difference, batched pivoting suppresses degradation of numerical accuracy; it can avoid selecting bad pivots with a strong probability, because a set of candidates that includes a pivot closer to zero has a worse score and thus rejected at the final decision.

Another difference is that batched pivoting requires additional computation costs for the local Gaussian elimination. However, the amount of additional cost is $O(dn^2)$, which is much smaller than the original cost $(2/3)n^3 + O(n^2)$ as long as $d \ll n$ holds.

Requirement about data mapping. To succeed in selecting d pivots from $[k, k+d)$th columns, data mapping needs to satisfy the following condition: for each $i \ge k$, contiguous elements $a_{ik}, a_{i,k+1}, \cdots a_{i,k+d-1}$ should be owned by a single node. Under this condition, part of grayed submatrix owned by each sharing node composes a submatrix. If the matrix is distributed per block whose size s_b is larger than or equal to d, as in Figure 3, this condition is satisfied.

5 Implementation

We have implemented a sequential version and two parallel versions of Gaussian elimination with batched pivoting. One of parallel versions is written with MPI and the other uses the Phoenix message passing library[14].

The MPI version is based on High performance Linpack (HPL), which originally uses partial pivoting. We have modified the code for pivoting to implement our batched pivoting. Like HPL, it uses two dimensional block cyclic mapping; thus the number of nodes must be fixed during the computation. This version and sequential one are evaluated in Section 6, since the MPI version is faster than the current Phoenix version.

The Phoenix version is written from scratch by using the Phoenix library. Unlike the MPI version, it can support dynamic changes of the number of computing nodes[5]. With this feature, this version will be more suitable for actual Grid environments, though the implementation has not been refined yet. Supporting dynamic environments is enabled by the following load balancing method. Nodes periodically sends the number of its own matrix blocks to each other, and we move blocks between a pair of nodes if it improves load balance. Thus new nodes that initially have no matrix data can join the running computation. The Phoenix library makes it easier to write parallel programs that support dynamic changes of data mapping.

6 Experimental Results

This section evaluates our Gaussian elimination algorithm by using two criteria: tolerance to latencies and numerical accuracy. Parallel experiments have been conducted on a 190-node Linux cluster connected via Gigabit Ethernet. Each node is equipped with dual Xeon processors, whose clock speeds are 2.4 or 2.8 GHz. We have run a single computing process for each node. Latencies among cluster nodes are 55 to 75 microseconds. In parallel experiments, we have used the MPI version described in Section 5. It has been linked with mpich 1.2.6 library for communication and high performance BLAS library by Kazushige Goto[8] as a linear algebra kernel. The experiments are done on a single cluster, not on actual Grid environments. To evaluate tolerance to latencies, we insert large artificial latencies by modifying message passing APIs.

6.1 Tolerance to Latencies

Figure 4 shows parallel performance of our implementation on a cluster without inserting latencies. The

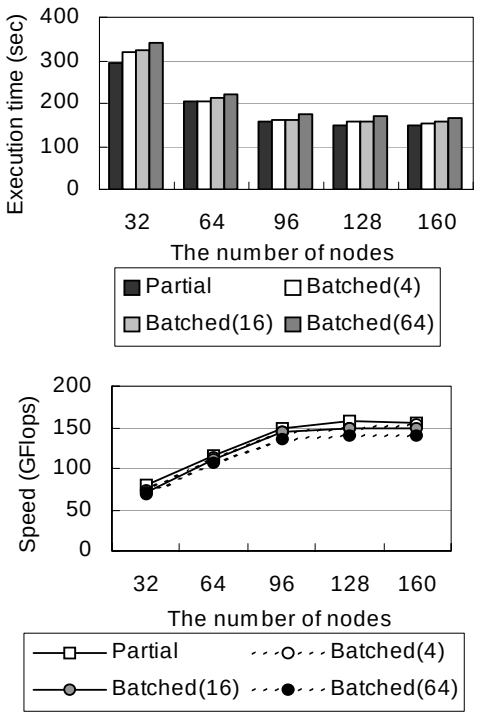

Figure 4. Parallel performance on a cluster($n = 32,768$, s_b=256). The upper graph shows execution times and the lower shows speeds in GFlops. The sizes of process grids are $4 \times 8, 8 \times 8, 8 \times 12, 8 \times 16$ and 8×20.

matrix size n is 32,768 and the block size s_b is 256. The number of nodes is 64. 'Batched(d)' denotes our implementation with batched pivoting, where d is the number of pivots computed at once. 'Partial' denotes original HPL that uses partial pivoting.

We see that batched pivoting achieves similar scalability to partial pivoting, while they suffer from some overhead, which increases with larger d. This overhead is due to additional computation introduced by the local Gaussian elimination; it is 7.5 to 15 % when d is 64.

Figure 5 shows performance when we insert artificial latencies on each message send. We have inserted identical latencies for all pairs of nodes in each experiment. The matrix size, the block size and the number of nodes are same as above.

We see that the speed of partial pivoting heavily decreases with large latencies; when we insert 10 milliseconds latencies, it gets 6.0 times slower. On the other hand, batched pivoting is much more tolerant to large latencies; the decrease of speed with 10 milliseconds latencies is only 22% when d is 64. In all cases when we insert latencies, batched pivoting is faster than partial pivoting. This result shows that effects of tolerance to latencies dominate costs for additional computation.

6.2 Numerical Accuracy

We evaluate numerical accuracy of various pivoting methods through numerical experiments with random matrices. We have conducted two sets of experiments: sequential experiments with small matrices and parallel experiments with large matrices. In both experiments, we evaluate numerical accuracy by using a normalized residual:

$$||A\tilde{x} - b||_\infty / (||A||_\infty ||\tilde{x}||_\infty n\epsilon)$$

where $\tilde{x}$ is the computed solution of the equation and ϵ is the machine precision 2^{-53}. This is one of the three residuals used in HPL residual check [2] .

In sequential experiments, we use random matrices whose elements distribute uniformly on $[-1, 1]$. The sizes of matrices are 128 to 2048.

Care should be taken because batched pivoting in sequential execution behaves identically to partial pivoting. To observe the difference of these methods, we modified the implementation of batched pivoting as follows. We divide the matrix into blocks, and then treat them as nodes; we let each block provide its candidate

[2] Although the web page [12] says the corresponding residual is $||A\tilde{x} - b||_\infty / (||A||_\infty ||\tilde{x}||_\infty \epsilon)$ that excludes n, above residual is actually used, according to the source code(`HPL_pdtest.c`).

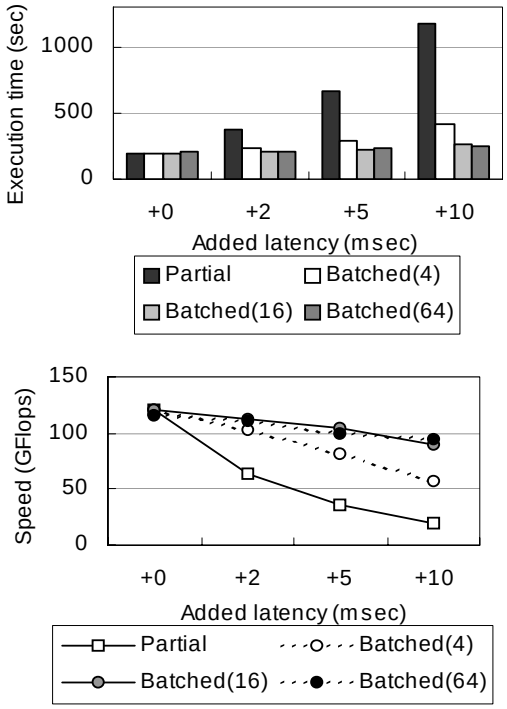

Figure 5. Parallel performance with emulated large latencies ($n = 32,768$, s_b=256). The number of nodes is $64 = 8 \times 8$. The upper graph shows execution times and the lower shows speeds in GFlops.

set for pivots. In sequential experiments, the block size s_b is 64.

Figure 6 shows the results. We have conducted experiments with 100 random matrices for each condition, and the graph shows the average normalized residuals among them. The graph includes the results of partial pivoting, threshold pivoting, pairwise pivoting and our batched pivoting. It also includes results of Gaussian elimination without any pivoting ('No' in the legend), which is obviously numerically unstable. The number in parenthesis for 'Batched()' denotes d, while that for 'Threshold()' is the threshold parameter τ described in Section 3.

As expected, partial pivoting shows the best accuracy among those methods. We see that the normalized residuals go down as the matrix size n increases, which means that the growth of $||A\tilde{x} - b||_\infty$ is milder than $O(n)$. A similar tendency is also observed in all cases of batched pivoting and threshold pivoting. Although their residuals are larger than that of partial pivoting, we see that the difference is rather small; difference in residual between 'Batched(4)' and 'Partial' is 1.09–1.55 times. When d is larger, we see the residuals increase; here we observe a tradeoff between tolerance to latencies and accuracy. On the other hand, pairwise pivoting shows qualitatively different results. Although its average residual is at a similar level to other methods when n is 128, it rises as n increases and is 120 times worse than partial pivoting when n is 2048. Considering that partial pivoting and threshold pivoting are not latency tolerant, our batched pivoting is a good compromise between numerical accuracy and tolerance to large latencies.

Figure 7 shows the results of parallel experiments by using the MPI version described before. We have used the random matrices generated by HPL. The matrix sizes are 8192 to 49,152 and the block size is 256. The graph shows similar results to the previous one; batched pivoting achieves comparable accuracy to partial pivoting. On the other hand, we see that the effects of using different d are smaller. We consider this is due to experimental conditions. First, the block size is larger than in sequential experiments. Secondly, in parallel experiments, each node that possesses several blocks provides candidates. Thus batched pivoting has more chances to select better candidates for pivots than in sequential experiments, where each block provides candidates.

7 Discussion

Average accuracy. We have observed that partial, threshold and batched pivoting display good accu-

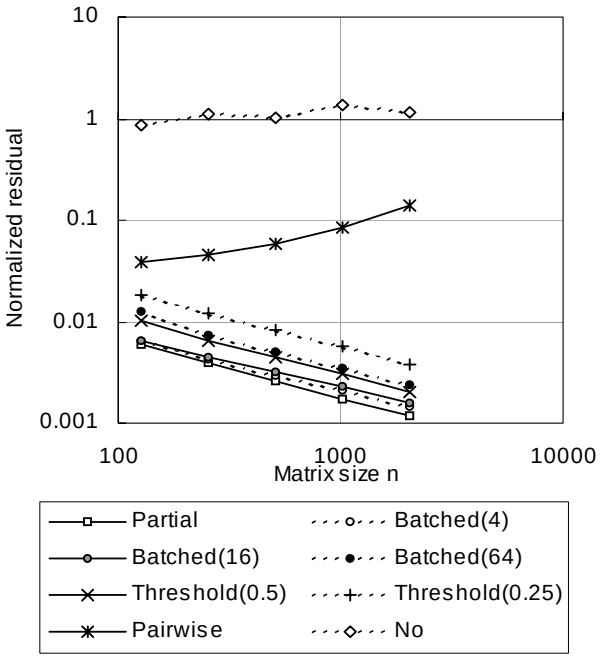

Figure 6. Average normalized residuals for small matrices.

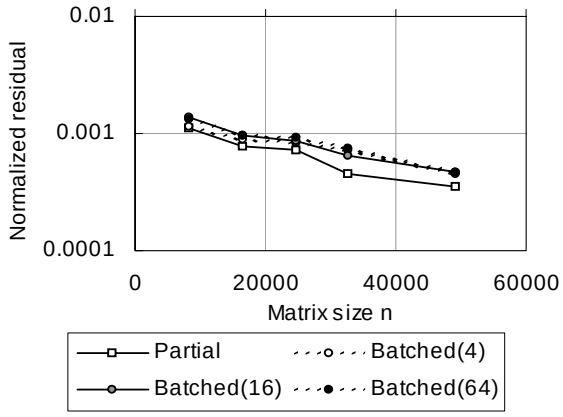

Figure 7. Normalized residuals for large matrices.

racy, while pairwise pivoting is much worse. Trefethen and Screiber [15] have given an explanation for this. To obtain good stability on average, following conditions should be satisfied: (1) that $|a_{ik}a_{kj}/a_{kk}|$ is sufficiently small, and (2) that the correction matrix introduced in each update phase is of rank 1. While the first condition is satisfied by all pivoting methods we have described, pairwise pivoting breaks the second. It is satisfied by partial, threshold and batched pivoting, because a single pivot row is used to update the whole $(n-k-1)\times(n-k-1)$ submatrix at each step. Here the correction matrix has elements $(-a_{ik}a_{kj}/a_{kk})$, thus its rank is 1. On the other hand, pairwise pivoting breaks the condition because each row may be eliminated by different pivot row. We consider batched pivoting achieves good average accuracy because it is one of methods that satisfy both conditions.

Cases when the current algorithm fails.
Through experiments using random matrices, we have shown that batched pivoting works well. There are, however, matrices where current batched pivoting fails, while partial pivoting successes. In general, when all elements examined in pivoting phase are zero, we cannot select a pivot and the computation cannot proceed any more. This problem arises with our current algorithm when we use random permutation matrices, in which non-zero elements are located sparsely and distributedly (Although they are sparse matrices, there are dense matrices that have similar property). In such cases, all nodes that share columns may fail to select candidates for pivots. This situation occurs if all $(c \times d)$ submatrices that compose d columns have a rank lower than d. This problem will be avoided by improving the algorithm, such as adjusting the number of pivots adopted in the final decision according to the result of the local Gaussian elimination.

Message driven objects. This paper has taken one of communication intensive applications, Gaussian elimination, and described techniques for tolerating WAN latencies. For this goal, an approach with message driven objects has been described [9, 5]; the whole data structure is divided into many message driven objects, which are invoked by asynchronous messages. This approach is generally useful to make algorithms latency tolerant. However, this paradigm alone does not help when the execution time of the algorithm is dominated

by a large critical path sensitive to message latencies, as was the case in Gaussian elimination with partial pivoting. In such cases, we need to develop a new algorithm for tolerating latencies.

8 Conclusion

This paper has presented a parallel algorithm of dense Gaussian elimination that is highly tolerant to large latencies. The key technique for latency tolerance is batched pivoting, which requires much less frequent synchronizations than traditional partial pivoting. Although batched pivoting suffers from the increase of computation costs and the degradation of numerical accuracy, we have shown that those impacts are small through experiments. Among all pivoting methods we have examined, batched pivoting is the only method that achieves both latency tolerance and accuracy.

As future work, we are planning to implement a strategy to support matrices with that the current algorithm fails. We will also conduct experiments on actual Grid environments. In addition to experiments, we will analyze numerical accuracy of batched pivoting in a method similar to Trefethen's average case analysis. We would also like to investigate and improve tolerance to latencies of other communication intensive applications to enlarge the range of Grid applications.

Acknowledgement

This work is partially supported by "Precursory Research for Embryonic Science and Technology" of Japan Science and Technology Corporation and "Grants-in-Aid for Scientific Research" of Japan Society for the Promotion of Science.

References

[1] Folding@home. `http://folding.stanford.edu`.

[2] SETI@home. `http://setiathome.ssl.berkeley.edu`.

[3] TOP500 supercomputer sites. `http://www.top500.org/`.

[4] G. Allen, T. Dramlitsch, I. Foster, N. T. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Proceedings of ACM/IEEE conference on High Performance Networking and Computing (SC)*, 2001.

[5] Toshio Endo, Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. High performance LU factorization for non-dedicated clusters. In *Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004.

[6] I. Foster, J.Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24(12):1735–1749, 1998.

[7] Ian Foster and Carl Kesselman. *The Grid: A Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[8] Kazushige Goto. High-performance BLAS. `http://www.cs.utexas.edu/users/flame/goto/`.

[9] Gregory A. Koenig and Laxmikant V. Kale. Using message-driven objects to mask latency in grid computing applications. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2005.

[10] Joel Malard. Threshold pivoting for dense LU factorization on distributed memory multiprocessors. In *Proceedings of ACM/IEEE conference on Supercomputing (SC)*, pages 600–607, 1991.

[11] Motohiko Matsuda, Tomohiro Kudoh, and Yutaka Ishikawa. Evaluation of mpi implementations on grid-connected clusters using and emulated wan environment. In *Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

[12] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`.

[13] Danny C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, c-34(3):274–278, 1985.

[14] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 216–229, 2003.

[15] Lloyd N. Trefethen and Robert S. Schreiber. Average case stability of Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 11(3):335–360, 1990.