# High Performance LU Factorization for Non-dedicated Clusters

Toshio Endo
*PRESTO, JST*[*]
endo@logos.t.u-tokyo.ac.jp

Kenji Kaneda
*University of Tokyo/PRESTO, JST*
kaneda@is.s.u-tokyo.ac.jp

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

Akinori Yonezawa
*University of Tokyo*
yonezawa@is.s.u-tokyo.ac.jp

## Abstract

*This paper describes an implementation of parallel LU factorization. The focus is to achieve high performance on non-dedicated clusters, where the number of available computing resources may be arbitrary and even dynamically changing. We accommodate joining/leaving processes by describing the algorithm in the Phoenix programming model. We achieve high performance in this setting by a combination of techniques including a latency tolerant communication and data partitioning that achieves both load balance and small communication volume for arbitrary and dynamically changing number of processors. We observed 130 GFlops with 128 processes on a 70-node dual 2.4GHz Xeon cluster, at matrix size = 46,080. This performance is comparable to that of the High Performance Linpack (HPL). When cluster nodes are loaded by background processes, our implementation surpasses HPL.*

## 1 Introduction

Today's common practice in managing heavily shared cluster resources is space sharing with batch schedulers. This is especially true for traditional high performance computing (HPC) applications such as matrix operations and PDE solvers, because they often need dedicated resources for predictable load-balancing and synchronization latencies. On the other hand, so-called "high-throughput computing" [6, 3, 8] have a strength that they can take advantage of non-dedicated resources such as desktops and interactively shared, unmanaged clusters.

As far as hardware performance is concerned, powerful desktops/laptops and ever increasing local-/campus-/wide-area bandwidths are almost "ready" for HPC applications. Sooner or later, we will have a pool of shared and decentralized resources connected by links powerful enough for traditional HPCs. We cannot exploit such resources for them for free, however. A combination of proper algorithm design, programming model, and middleware is necessary to make it real.

This paper takes one HPC application, LU factorization, and studies an algorithm that achieves robust performance in shared environments and therefore can take advantage of non-dedicated resources. We chose this problem because it has been a popular benchmark for dedicated parallel computers and there is a common basis for absolute performance comparison. Standard algorithms for LU perform static load-balancing, static data-partitioning, extensive communication, and frequent barrier synchronization, so we face every difficulty in running it efficiently in non-dedicated environments. We tackle this problem by a combination of a proper, latency-tolerant and adaptive algorithm design and middleware we developed called Phoenix [11]. The algorithm is asynchronous to combat with scheduling skews. It also performs dynamic load balancing and data partitioning, while carefully maintaining required communication volume. Finally, it allows newly available processes to join the ongoing computation. On a 70-node dual Xeon 2.4GHz cluster connected via Gigabit Ethernet, we achieved 130 GFlops with 128 processes at matrix size = 46,080. This is comparable to HPL[10] in our environment. When some nodes are loaded, our algorithm is more robust than HPL and surpasses it. We also confirmed it can take advantage of dynamically joining resources; when it started with a small number of nodes and then acquired additional resources at runtime, it reached a similar peek speed with the case where all nodes participate from the beginning and assume processes are static.

The rest of the paper is organized as follows. Section 2 describes the Phoenix programming model briefly. In Section 3, we mention characteristics of LU factorization and a traditional data distribution method. We describe our implementation of LU on the Phoenix model in Section 4. Section 5 evaluates its performance

[*]Japan Science and Technology Agency

and compares it with that of HPL. We mention related work in Section 6 and conclude in Section 7.

## 2 Phoenix Model

The Phoenix model, which we used to describe our LU algorithm, is a small but important extension to a regular message passing model. The extension is to describe message passing algorithms using dynamic processes. As in usual message passing models, application processes can communicate by sending messages to a desired destination. Unlike in traditional models, however, the name of the destination is not permanently bound to a physical computing resource. Instead, Phoenix provides another name space, called the *virtual node space*. Usually, the virtual node space is a contiguous range of integers. An application specifies a virtual node as the destination of each message. In typical applications, the number of virtual nodes is much larger than that of processes. In our LU implementation, we associate each matrix block with a single virtual node. Applications can specify the mapping between virtual nodes and processes arbitrarily. When a process performs a Phoenix API call `ph_send(`$v$`)`, where $v$ is a virtual node name, the runtime system delivers the message to the process that is associated with $v$ at the time. The receiver process receives the message by `ph_recv()`(blocking) and `ph_try_recv()` (non-blocking). Phoenix allows applications to change the mapping between virtual nodes and processes dynamically. After a process $p$ calls `ph_assume(`$s$`)`, where $s$ is a set of virtual nodes, all messages whose destination are in $s$ are delivered to $p$. After $p$'s calling `ph_release(`$s$`)`, messages destined for $s$ are not delivered to $p$ any more. For details of the Phoenix API, see [11]. An important note is that Phoenix is not similar to the data-parallel or the SIMD model, where the programmer specifies the behavior of each "virtual processor" and the system automatically schedules multiple virtual processors mapped on a physical process. Instead, Phoenix is still an MPI-like message passing model, where the programmer writes the behavior of each (physical) process, and a receiving API call (i.e., `ph_recv()`) receives *all* messages destined for that physical process. The only difference is that Phoenix programmers specify message destinations not by physical process names, but by virtual node names whose corresponding physical processes are known only at runtime.

In summary, we can write most parts of parallel programs independently from the number of physical processes. Suppose we map an element of an array $A(i)$ to virtual node $i$. When computation of $A(n)$ depends on the result of $A(m)$, all a programmer needs to do is to send the value $A(m)$ to the virtual node $n$, without knowing which process actually possesses $A(n)$.

```
1:  for ($k = 0$; $k < B$; $k$++) {
2:      $A_{k,k}$ = factorize($A_{k,k}$);
3:      for ($i = k + 1$; $i < B$; $i$++)
4:          $A_{i,k}$ = update_L($A_{i,k}$, $A_{k,k}$);
5:      for ($j = j + 1$; $j < B$; $j$++)
6:          $A_{k,j}$ = update_U($A_{k,j}$, $A_{k,k}$);
7:      for ($i = k + 1$; $i < B$; $i$++) {
8:          for ($j = j + 1$; $j < B$; $j$++) {
9:              $A_{i,j} = A_{i,j} - A_{i,k} \times A_{k,j}$;
10:         } } }
```

Figure 1: An outline of sequential LU factorization

## 3 LU Factorization

Figure 1 shows an outline of sequential LU factorization. We assume $N \times N$ dense matrix is divided into $SB \times SB$ blocks. In the figure, $A_{i,j}$ represents the $(i, j)$th block rather than a single element. We let $B$ to be the number of blocks $N/SB$. The $k$th iteration of the outermost loop consists of the following operations. First, we factorize the diagonal block $A_{k,k}$ (line 2). Then we update the lower blocks in column $k$ by using the value of $A_{k,k}$ (line 4). Similarly, blocks in row $k$ are updated (line 6). Finally, we update the trailing sub-matrix, which contains blocks $A_{i,j}$, where $k+1 \leq i, j < B$ (line 9). Block $A_{i,j}$ requires the results of $A_{k,j}$ and $A_{i,k}$ for its computation.

### 3.1 Discussion of Parallel Performance

Now, let us consider the communication cost of parallel LU. When the computation of a block $b$ is finished, it should be multicast to all processes that contain blocks of the same row or column with $b$. Thus the communication cost is proportional to the number of processes that cover a single row or column. We can see it is important to reduce the number of such processes.

We also need to take care of the load balance. As the outermost loop is advanced, the size of the sub-matrix to be updated shrinks. Since blocks in the right-lower are updated more times than blocks in the upper-left, such 'heavy' blocks should be scattered across many processes to avoid load imbalance.

To achieve both low communication cost and load balance, many traditional parallel implementations have adopted *two-dimensional block cyclic distribution*. This is also adopted in HPL, which uses LU factorization to solve linear equations system. All $P$ processes conceptually construct a two-dimensional $P_x \times P_y$ grid. Each block is assigned cyclically along both dimensions of the grid. By using a cyclic assignment, we can scatter heavy blocks on all processes. This mapping also limits the number of processes that cover a single row or column. The total communication cost is $O(N^2(P_x+P_y))$, which gets smaller as $P_x$ and $P_y$ are closer.

The problem of this mapping is that it requires the size of the process grid to be fixed statically. In the next section, we describe an implementation that supports arbitrary and even dynamically changing number of processes.

## 4 Our Implementation

### 4.1 Overview

This section presents our implementation of parallel LU that supports dynamically changing number of processes. Our implementation is based on the Phoenix model. We introduce a virtual node space whose size is fixed during the whole computation. Each virtual node is associated with a block of the matrix, as described in Section 4.2. At the beginning of the computation, we start a fixed number of 'initial processes'. The virtual node space is distributed among them. Each process invokes the computation of blocks in a data-driven fashion, rather than computes them synchronously. We expect this method makes our implementation be resilient to unexpected delays caused by background processes. To schedule the computation of blocks, each process maintains a prioritized task queue. The details are shown in Section 4.4. During the computation, our implementation allows new processes to join dynamically, whenever you want to do so. A newly added process steals some virtual nodes from existing processes and then starts its computation.

### 4.2 Mapping Data onto Virtual Nodes

In our implementation, we construct a $B \times B$ two-dimensional grid of virtual nodes regardless of the number of physical processes. Then we map a single block to each virtual node (As described in Section 2, each virtual node actually has a name of an integer; we use the two dimensional name in the following discussion). Here we focus on the load imbalance introduced by the inequality among the blocks. In the traditional implementation, the cyclic distribution alleviates this imbalance. However, we cannot adopt it simply, because it uses the number of processes as the length of the cycle. Instead, we hash the position of blocks on the virtual node space as shown in Figure 2. We assign block $A_{i,j}$ to virtual node $(h(i), h(j))$, where $h$ is a random permutation of $[0, B)$. The permutation $h$ needs to be static and agreed among all processes. By doing this, we expect 'heavy' blocks are scattered on many processes when the virtual node space is divided.

Note that blocks in a single row (or column) on the original matrix are still settled in the same row (or column) on the virtual node space. Hence the discussion on the communication cost in the previous section also applies to the virtual node space. In the next section,

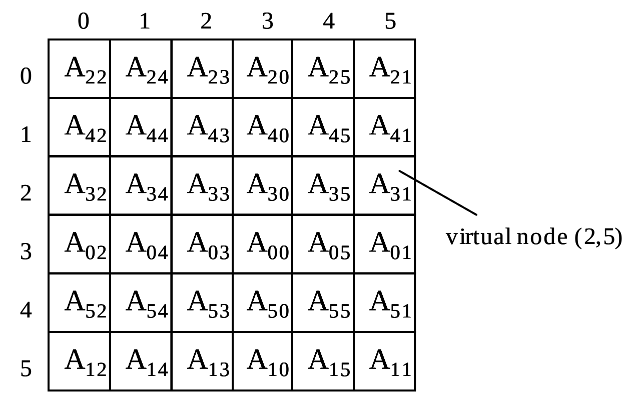
0 1 2 3 4 5
0 A22 A24 A23 A20 A25 A21
1 A42 A44 A43 A40 A45 A41
2 A32 A34 A33 A30 A35 A31
3 A02 A04 A03 A00 A05 A01
4 A52 A54 A53 A50 A55 A51
5 A12 A14 A13 A10 A15 A11
virtual node (2,5)

Figure 2: An example of mapping between blocks and virtual nodes, when $h(0) = 3$, $h(1) = 5$, $h(2) = 0$, $h(3) = 2$, $h(4) = 1$ and $h(5) = 4$.

we pay attention to how we divide the space in order to reduce the communication cost.

### 4.3 Distributing Virtual Nodes

We describe the distribution of a two-dimensional virtual node space among initial processes. We allow the number of initial processes, say $P$, to be arbitrary, while all of them must know $P$. Our forcus is to make the shape of the partial space of each process be as close to a regular square as possible. Such a division tends to reduce the communication cost, because it reduces the number of processes that cover a single row or column. This is achieved by using the recursive bisection technique, described by Crandall et al. [7] in the context of heterogeneous clusters.

Figure 3 shows an example of the division among nine processes. First, we divide $P$ processes into two groups, whose sizes are $P_1 = \lfloor P/2 \rfloor$ and $P_2 = P - \lfloor P/2 \rfloor$, respectively. In the figure, we divide nine processes into four and five. Then we divide the longer edge of the rectangle in a ratio of $P_1 : P_2$. If the rectangle is a regular square, the direction to be cut is arbitrary. Next, we associate two new rectangles to the process groups respectively and repeat the division recursively.

### 4.4 Behavior of Processes

Figures 4 and 5 describe the behavior of each initial process. The behavior of dynamically added processes is similar except that they do not initialize local data in the beginning, but steal data from other processes.

Each process maintains its assigned virtual node set $L$ and its local blocks $A_{i,j}$. For data-driven execution, a process maintains the iteration number, named $K_{i,j}$, for each local block. $T$ is a prioritized task queue that

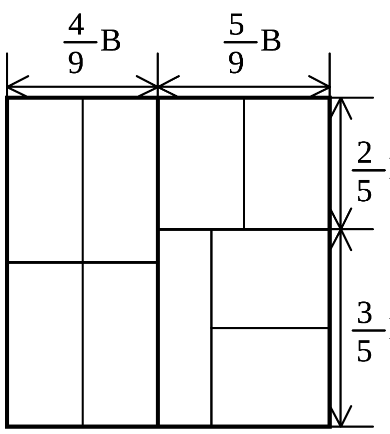

Figure 3: Dividing a $B \times B$ virtual node space among 9 initial processes.

holds all local runnable tasks. $R$ is an array of buffers for received data from other processes; computation result of $A_{i,j}$ is stored at $R_{i,j}$ in receiver processes.

At the beginning, initial processes set up their local data in the function `initialize`. Since the task graph of LU starts at $A_{0,0}$, a process that possesses $A_{0,0}$ throws the initial task into $T$.

Then each process starts `mainloop`, where it repeatedly examines its task queue $T$ and message queue by using `ph_try_recv` API. If there is a runnable task in $T$, it is executed in `run_task`. The result of the task is sent to processes that depend on its result in `mcast` function. Our multicast method is described below. When a process receives a result of $A_{m,n}$, it is stored at $R_{m,n}$ in `handle_msg`. Then all blocks that depend on $A_{m,n}$ are examined whether they are runnable now. If a runnable block is found, it is thrown into $T$. By repeating these tasks, all processes cooperatively proceed in the whole computation.

**multicast** Here we discuss how we should multicast the contents of blocks. Suppose we attempt to send a message $M$ to all members in a virtual node set $D$. It is too expensive to send distinct messages to each member. Instead, we use the following method described in `mcast` function. The sender chooses one virtual node $d$ from $D$ and sends a pair $(M, D)$ to $d$. When a process that possesses $d$ receives the message, it forwards $(M, D \setminus L)$ to one of $D \setminus L$. This method limits the number of total messages to the number of involved processes, rather than $|D|$.

**garbage collection of** $R$ Although omitted from the pseudo code, we remove elements in the receiving buffer $R$ when they are no longer required.

**task scheduling** Through the experiments, we have found that when the task queue is a simple FIFO, the algorithm does not have a desired resiliency to background processes. To improve this, we execute tasks that reside on the critical path earlier. For this purpose, we implement task queues as prioritized queues. We assign a number $Pr(t)$ that represents priority to each task. The smaller $Pr(t)$ stands for higher priority. We let $t$ to be the $k$-th updating task of a block $A_{i,j}$. We define the priority of it as $Pr(t) = min(min(i,j), k + S)$, where $S$ is a given constant number. Our intention is to suppress the skews of iteration among blocks. The iteration numbers of blocks in a single process tends to be within a range of $S$. Thus we call $S$ 'target skew'. Although we can tolerate larger delay by using larger $S$, the memory consumption for receiving buffer $R$ grows. We will show the effects of $S$ in Section 5.

## 4.5 Dynamically Joining Processes

We allow new processes to join the running computation dynamically. Unlike initial processes, each new process needs to know neither the number of running processes, nor the number of processes to be added. Instead, we assume it knows the size of the whole virtual node space. Now let us suppose a new process $t$ attempts to join the running computation. As $t$ has no virtual node, it needs to obtain some nodes from running processes. For this purpose, $t$ (called the thief) sends a `steal` message to a randomly chosen destination in the virtual node space by using `ph_send`. When a process $v$ (called the victim) receives the message, it divides its local virtual nodes into two groups. Then $v$ abandons one of the two by `ph_release` and sends them with associated matrix data to the thief $t$. When $t$ receives them, it starts the computation. How should $v$ divide its virtual nodes? If the virtual nodes compose a single rectangle, the situation is similar to the case for initial processes; we divide the longer edge of it. If a victim has several rectangles, we divide them into two groups arbitrary.

Contrarily, a running process may attempt to leave the computation. In this case, it gives all its virtual nodes to one of other processes and then quits.

After the number of processes has been changed, the amount of the local virtual nodes may be imbalanced among processes. To alleviate it, we introduce dynamic load balancing function by making a simple extension to the `steal` messages. We let each running process, say $t$, send a `steal` message to a random destination periodically. It attaches an additional information $S_t$, the number of $t$'s local virtual nodes, to the message. The receiver or the victim $v$ compares the number of its local virtual nodes $S_v$ to $S_t$. If $v$ founds $S_v$ is sufficiently larger (currently if $S_v > 2S_t$), it attempts to

```
initialize() {
    determine initial $L$;
    ph_assume($L$);
    for all $(p, q)$ in $L$ {
        $i := h^{-1}(p)$; $j := h^{-1}(q)$;
        set up $A_{i,j}$;
        $K_{i,j} := 0$;
    }
    for all $(i, j)$ in $[0, B) \times [0, B)$
        $R_{i,j} :=$ NULL;
    $T :=$ empty;
    if $((h(0), h(0)) \in$ L) // if I have $A_{0,0}$
        enqueue($T, \langle 0, 0, 0 \rangle$);
    mainloop();
}

mainloop() {
    while (true) {
        if ($T$ is not empty) run_task(dequeue($T$));
        $M :=$ ph_try_recv();
        if ($M \neq$ NULL) handle_msg($M$);
    }}

handle_msg($\langle$block, $m, n, A, D\rangle$) {
    $R_{m,n} := A$; // store received $A_{m,n}$ locally
    for all $(p, q)$ in $(D \cap L)$ {
        // for all local blocks affected by $A_{m,n}$
        $i := h^{-1}(p)$; $j := h^{-1}(q)$;
        if (is_runnable($i, j$))
            enqueue($T, \langle i, j, K_{i,j}\rangle$);
    }
    mcast($\langle$block, $m, n, A, D \setminus L\rangle$); // forward msg
}

is_runnable($i, j$) {
    $k := K_{i,j}$;
    if ($i = k$ and $j = k$) return TRUE;
    else if ($j = k$ and $R_{k,k} \neq$ NULL)
        return TRUE;
    else if ($i = k$ and $R_{k,k} \neq$ NULL)
        return TRUE;
    else if ($R_{i,k} \neq$ NULL and $R_{k,j} \neq$ NULL)
        return TRUE;
    return FALSE;
}

mcast($\langle$block, $i, j, A, D\rangle$) {
    // send msg to all virtual nodes in $D$
    if ($D \neq \emptyset$) {
        $d :=$ any member of $D$;
        ph_send($d$, $\langle$block, $i, j, A, D\rangle$);
    } }
```

Figure 4: An outline of our asynchronous parallel LU factorization

```
run_task($\langle i, j, k\rangle$) {
    if ($i = k$ and $j = k$) comp_diag($k$);
    else if ($j = k$) comp_L($i, k$);
    else if ($i = k$) comp_U($j, k$);
    else comp_trail($i, j, k$); }

comp_diag($k$) {
    $A_{k,k} :=$ factorize($A_{k,k}$);
    // send result to row $k$
    $D := \{(h(k), h(j)) | k + 1 \leq \forall j < B\}$;
    mcast($\langle$block,$k, k, A_{k,k}, D\rangle$);
    // send result to column $k$
    $D := \{(h(i), h(k)) | k + 1 \leq \forall i < B\}$;
    mcast($\langle$block,$k, k, A_{k,k}, D\rangle$);
    $K_{k,k} := k + 1$; }

comp_L($i, k$) {
    $A_{i,k} :=$ update_L($A_{i,k}, R_{k,k}$)
    // send result to row $i$
    $D := \{(h(i), h(j)) | k + 1 \leq \forall j < B\}$;
    mcast($\langle$block,$i, k, A_{i,k}, D\rangle$);
    $K_{i,k} := k + 1$; }

comp_U is similar to comp_L. omitted

comp_trail($i, j, k$) {
    $A_{i,j} := A_{i,j} - R_{i,k} \times R_{k,j}$;
    $K_{i,j} := k + 1$; }
```

Figure 5: An outline of our asynchronous parallel LU factorization (cont.)

balance between $v$ and $t$ by giving $(S_v - S_t)/2$ virtual nodes to $t$. We expect the amount of virtual nodes are balanced among all processes after repeating these transactions sufficiently many times.

## 5 Experimental Results

### 5.1 Experimental Environments

We evaluate the performance of our LU implementation. Our implementation is written in C language on top of the Phoenix library [11].

We also show the performance of the HPL. Note that comparing HPL and our LU may not be fair, because HPL supports row pivoting, while our implementation does not support. However, the computation complexity is approximately common between them (Both perform $(2/3)N^3 + O(N^2)$ floating operations), hence we believe this comparison is useful for a rough evaluation. We ran HPL on the mpich MPI library [2].

As a basic block for linear algebra calculation, both programs use a BLAS library that is automatically generated by the ATLAS optimizer [1]. In the experiments, we let the block size $SB$ to be 240, since we have observed the `dgemm` matrix multiply function in the generated BLAS library shows good performance on this size.

Experiments are done on a 70-node IBM BladeCenter cluster system connected via Gigabit Ethernet. The central switch is Extreme Summit7i. Each Linux 2.4.19 node is equipped with two 2.4GHz Intel Xeon processors and 2GBytes memory.

### 5.2 Sequential Performance

We have run our LU and HPL on a single process at the matrix size $N = 7680$. The sequential speed of ours is 2.19 GFlops, while that of HPL is 2.85 GFlops. We suppose this difference comes from the costs paid for each block in ours. For example, ours apply the `dgemm` function for each block independently, while HPL updates several blocks by a single `dgemm` call because it adopts synchronous style. We are planning to design a method that supports both asynchronism and reducing the costs.

### 5.3 Parallel Performance

Figure 6 shows the parallel performance of our LU and HPL at $N = 30720$ and $N = 46080$. The number of processes is fixed during each run. When we use 96 or 128 processes, we assign them to nodes in the cluster in a round-robin style. The graphs include the speeds with several values of target skew $S$: 0, 5 and 'Inf' (infinity).

In all cases, we can see both programs scale well. Ours achieves 130 GFlops ($N = 46080, S = 5$) on 128

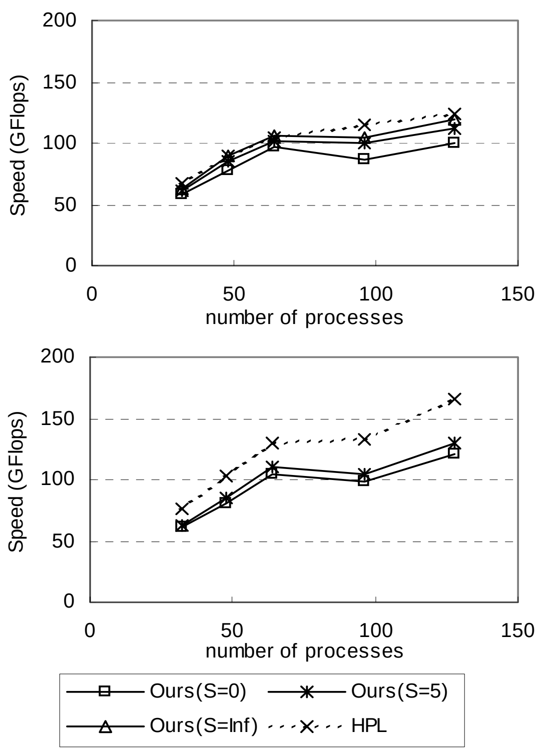

Figure 6: Speeds of our LU and HPL on 32 – 128 fixed processes. The matrix size $N$ is 30720 in above graph and 46080 in below. The sizes of the process grids in HPL are $4 \times 8$, $6 \times 8$, $8 \times 8$, $8 \times 12$ and $8 \times 16$.

processes. Compared with the sequential performance, the speedup is 55.2–59.4 times. While we can improve the speed by using larger $S$, it expands the memory consumption. '$S = Inf$' did not work at $N = 46080$ because of memory shortage. '$S = 5$' seems to be a good compromise between speed and memory consumption.

The absolute speeds of HPL surpass those of ours($S = 5$) by 12% at $N = 30720$ and 27% at $N = 46080$ on 128 processes. We consider this comes from the difference of the sequential computation speed.

### 5.4 Resiliency to Background Processes

Next, we evaluate the performance with the existence of background processes. While our LU or HPL is running, we ran processes that consume processors on randomly chosen nodes. The number of the background processes is four per chosen node. They are moved to other random nodes every ten seconds. Figure 7 shows the performance of our LU and HPL with the background load. The number of computation processes is

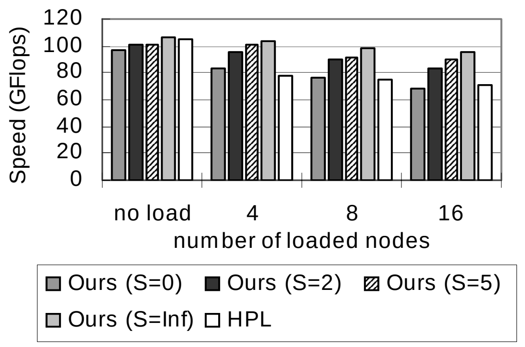

Figure 7: The performance with the existence of background processes. The number of computation processes is 64.

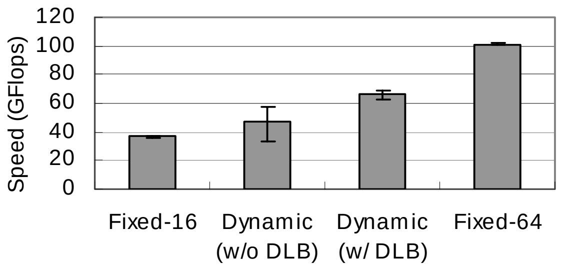

Figure 8: The speed of our LU with dynamically joining processes. The graph includes the average, best, worst speeds among five runs. In 'Dynamic', 16 processes run first and then 48 processes join. In 'Fixed-$P$', we always use $P$ processes.

64 and $N = 30720$. We see the performance of HPL and '$S = 0$' heavily suffers from background processes. When sixteen nodes are loaded, they slow down by 31% and 29% respectively. With larger target skew $S$, we can make our LU much more resilient; it slows down only by 10% at '$S = 5$' and '$S = Inf$'. From these results, we can see that we need both the data-driven execution method and a proper task scheduling for resiliency.

## 5.5 Dynamically Joining Processes

Figure 8 shows performance of our LU when new processes are added at runtime. In the 'Dynamic' bars, we start the computation with 16 initial processes and immediately add 48 new processes. Then all 64 processes work until the end. In 'Dynamic (w/o DLB)', we turn off the dynamic load balancing described in Section 4.5; after each new process succeeds in its first

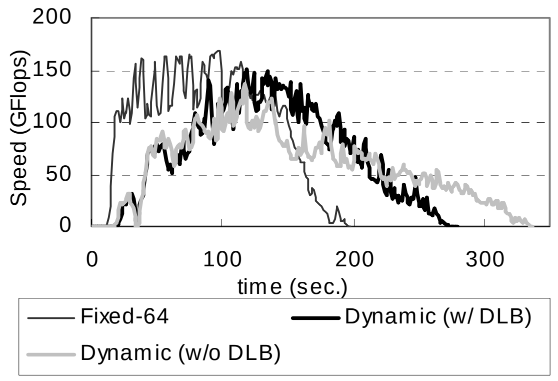

Figure 9: The transition of the speed with dynamically joining processes.

stealing, no more load balancing is performed for that process. 'Fixed-P' uses $P$ fixed processes during the whole computation. In all cases, the matrix size $N$ is 30720. Each bar represents the average speed among five runs, with the best and the worst speeds indicated with error bars.

It is clear from the graph that the dynamic load balancing is necessary to take advantage of dynamically joining processes. Without it, even adding 48 processes does not raise any improvement in the worst case. With dynamic load balancing, the speed rises to 63–69 GFlops, which is 1.73 to 1.88 times faster than Fixed-16.

Figure 9 shows the transition of the speed during the computation. It includes two dynamic cases and Fixed-64. We can see that, both dynamic cases reach a peek speed close to that of Fixed-64. This suggests that if the problem gets larger and hence the computation time longer, completion times of Dynamic and Fixed-64 will get closer. The difference between 'w/o DLB' and 'w/ DLB' appears in later stages of the computation. With DLB, the speed descends more rapidly than in 'w/o DLB', hence the total computation time is reduced. Because this descent gets more rapid as the load is balanced more fairly, there seems to be room for further improvement in 'w/ DLB' by introducing more aggressive load balancing.

# 6 Related Work

Some researchers have been working on parallel numerical algorithms that are suitable for modern cluster architectures. Beaumont et al. [5] constructed a algebra library for heterogeneous clusters. They extend a two dimensional block-cyclic distribution so that blocks are assigned to each process according to the processor speed. Unlike our method, they construct a 2D process grid. This seems inefficient when the number of processes is prime. Our data distribution is similar to that of Crandall et al. [7]. By recursively dividing a 2D matrix, they efficiently harness all computing resources of arbitrary numbers. They also take the processor speed into account.

Both approaches above deal with only static environments; they assume the number and the speed of machines are constant while programs are running. Matsubara et al. [9] have proposed an extension to MPI to support dynamic changing number of processes and loads by other processes. Currently, they use only one-dimensional distribution, which causes large communication overhead. Their framework assumes that all processes in user programs reach 'scheduling points' simultaneously. This assumption seems to make it difficult to write programs in an asynchronous style like ours.

## 7 Conclusion

This paper has described an implementation of parallel LU factorization. Our implementation performs well in non-dedicated clusters, since it achieves a good performance with processes of arbitrary and even dynamic changing number. By using the recursive bisection and the random permutation of the two dimensional space, we have succeeded in describing LU in a style independent from the number of processes. By invoking the computation of each block in a data-driven fashion and scheduling tasks properly, our algorithm is much more resilient to background processes than regular synchronous algorithms. Experiments show that our implementation achieves 130GFlops on 128 processes at matrix size = 46,080.

According to the Top500 list[4] just announced, a cluster of a similar hardware configuration to ours achieved about a twice performance of ours with a similar matrix size (210GFlops at size = 25,000 to 30,000, which is half of their best at size = 90,000 to 130,000). Since we do not know the details of their algorithm, we cannot conclude what is the source of the difference as of writing. We can hopefully incorporate any sequential optimization we are missing into ours, if any. One of the common challenges in Grid and large-cluster programming is how to build programs that "adapt to" the environment. Ideas we used are all common ones. Asynchrony masks skews and latencies and dynamic load/data partitioning relieves impact of disturbing loads. A proper programming model support made it possible to describe general message passing algorithms with dynamic processes. We hope our study shows proper combination of these ideas are effective even for HPC applications, which traditionally aim at squeezing performance almost solely in dedicated environments.

## References

[1] Automatically tuned linear algrbra software (ATLAS). `http://math-atlas.sourceforge.net`.

[2] MPICH - a portable MPI implementation. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[3] Portable batch system. `http://www.openpbs.org/`.

[4] TOP500 supoercomputer sites. `http://www.top500.org/`.

[5] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Trans. on Computers*, 50(10):1052–1070, 2001.

[6] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proc. of HPC ASIA*, pages 283–289, 2000.

[7] Phyllis E. Crandall and Michael J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *Proc. of HPDC*, pages 42–49, 1993.

[8] M. Litzkow, M.Livny, and M. Mutka. Condor - a hunter for idle workstations. In *Proc. of ICDCS*, pages 104–111, 1988.

[9] Masazumi Matsubara, Kazuhiro Suzuki, and Akira Katsuno. Dynamic load balancing in HPC applications for autonomous computing. *IPSJ Trans. on ACS*, 44(SIG 11 (ACS 3)):89–100, 2003. (in Japanese).

[10] 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/`.

[11] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In *Proc. of PPoPP*, pages 216–229, 2003.