Parallel Gaussian Elimination

Shule Li

Method

Gaussian elimination is a well-known method in solving linear equation systems. Given the following equation system characterized by matrix *a* and right hand side vector *b* (*a* and *b* combine to form so called "augmented matrix"),

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1k} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2k} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & b_k \end{array}\right] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix}.$$

one can manipulate the augmented matrix so that the sub-matrix consisted by elements of a is upper-triangular, as follows:

$$\left[\begin{array}{cccc|c} a'_{11} & a'_{12} & \cdots & a'_{1k} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2k} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{kk} & b'_k \end{array}\right].$$

To achieve this, one only need to look at the sub-columns below the diagonal elements. For instance, in order to eliminate the sub-column below diagonal element [*n*][*n*], we substitute each row *j* (*j* > *n*) by *a* [*j*][*i*] – *a* [*j*] [*n*] * *a* [*n*][*i*] / *a* [*n*][*n*]. The right hand side vector is changed accordingly: *b* [*j*] – *a* [*j*] [*n*] * *b* [*n*][*i*] / *a* [*n*][*n*], where *i* runs from 1 to k . The solution to the obtained system remains the same under this linear operation. Starting from element [1][1], we eliminate all the element below [1][1]. We then move on to element [2][2] etc, till the final diagonal element [*k*][*k*].

One exception is the case when a certain diagonal element is zero. In this case, the linear operation on the subsequent rows yield infinite. When *a* [*n*][*n*] is zero, we simply swap this row with row *l* below it such that *a* [*l*][*n*] is non-zero. Of course, when swapping happens, the right hand side vector *b* [*n*] needs to be swapped with *b* [*l*] as well. If one cannot find a non-zero element below [*n*][*n*], it indicates that the determinant of matrix *a* is zero, i.e. *a* cannot be inverted. In this case, the system considered does not have a unique solution.

Once the matrix is upper-triangular, *x*[*k*] can be readily obtained. Then since *x*[*k-1*] only depends on *x*[*k*], we can obtain *x*[*k-1*]. In this fashion, we work our way upward till the first element of *x* is obtained.

In terms of coding, we initialize the matrix *a* and right hand side vector *b* by assigning random numbers ranging from 0 to 1 to their elements. Using Gauss elimination method, we can obtain a solution vector *x.* We then verify that all the elements of the vector subtraction *ax-b* (where *a* and *b* are the original matrix and right hand side vector, not the converted ones) are zero (compared to a small tolerance number). We run the code on a Linux machines as well as a SunOS machine with matrix size (hereafter *nsize*) varying from 128 to 2048, thread number ranging from 1 to 32.

Sequential Code

The code includes 4 parts: (1) initialization of *a* and *b*; (2) swapping operations on augmented matrix; (3) elimination of lower-triangular rows; (4) verifying the result. The time spent on each of these section are measured for *nsize* ranging from *128 to 2048*. The result is plotted in the set of figures below.

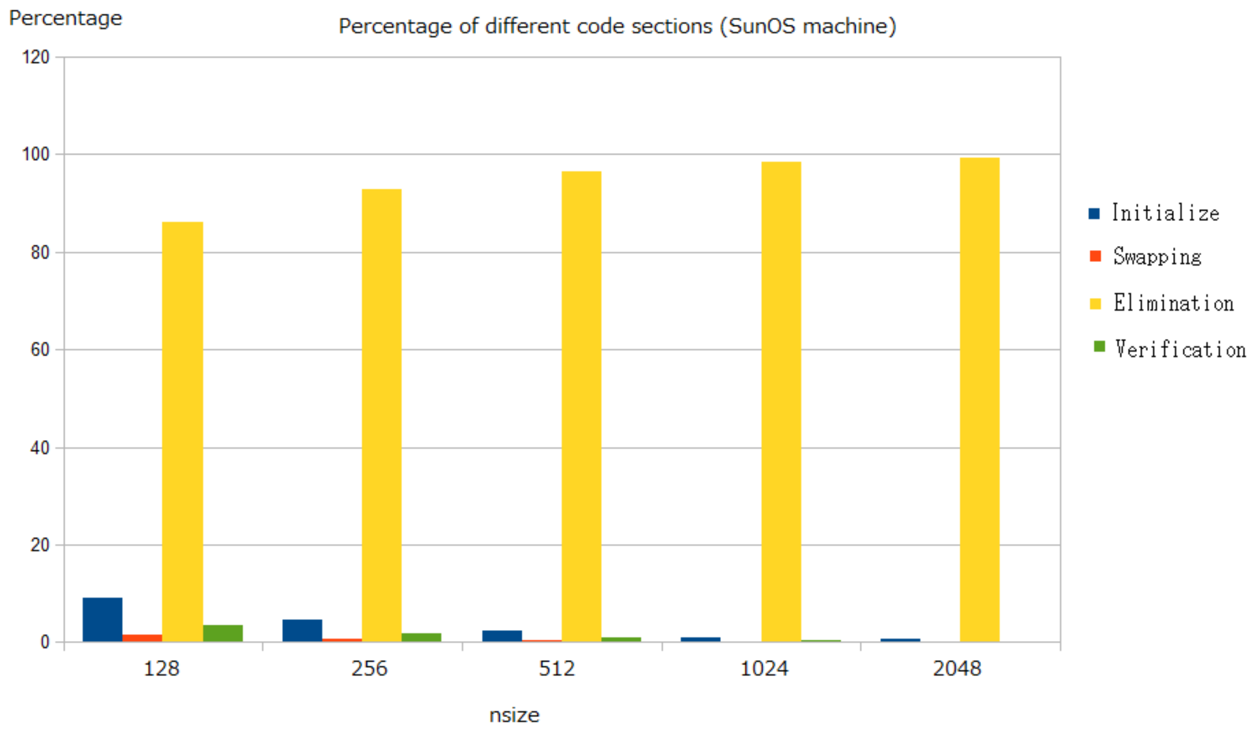


Fig.1

From the above figure, we can see that most of computation time is spent on the lower-triangular elements. The swapping has small cost because most of the time, the code does not need to swap the rows in the demonstrated experiments where random number initialization was used for the matrix *a* and vector *b*. If *a* has a big amount of diagonal elements that are zero, the time needed for swapping (indicated by the orange bar) could increase significantly. This profiling result suggests that to avoid overhead, it probably is the best to parallelize the elimination executions. In the actual code, we use three different parallelization strategies: Method 1 with vertical decomposition, Method 2 with horizontal decomposition. Both of the methods parallelize both the swapping and the elimination sections of the code. In Method 3, we only parallelize the elimination section based on vertical decomposition. From the performance results, we can demonstrate that Method 3 is the most efficient approach of the three strategies considered.

Parallelization Strategy

The swapping section and the elimination section are parallelized separately. For the swapping section, we first look at each column *k* to determine whether it needs a swapping. This is done by checking if *a* [*k*][*k*] is zero. This results in a loop that has to executed in a sequential order, since swapping operations are not commutable. If it is determined that the k-th row needs a swapping with the j-th row, each element of the k-th row needs to be swapped with those in the j-th row. Since these elements has no dependencies on one another, this part of the code can be parallelized: multiple threads can work on swapping a portion of the k-th row with the j-th row. At the end of the swapping operation, a barrier is placed to synchronize the progress.

The elimination section can be parallelized in different ways. Here, we use two different ways to compare the results. Notice that when eliminating the elements below *a* [*k*][*k*], only the sub-matrix *a* [*i'*][*j'*] where $k+1 \leq i' \leq nsize$, $k \leq j' \leq nsize$ (as shown in the diagram below), we can decompose the sub-matrix in two different ways.

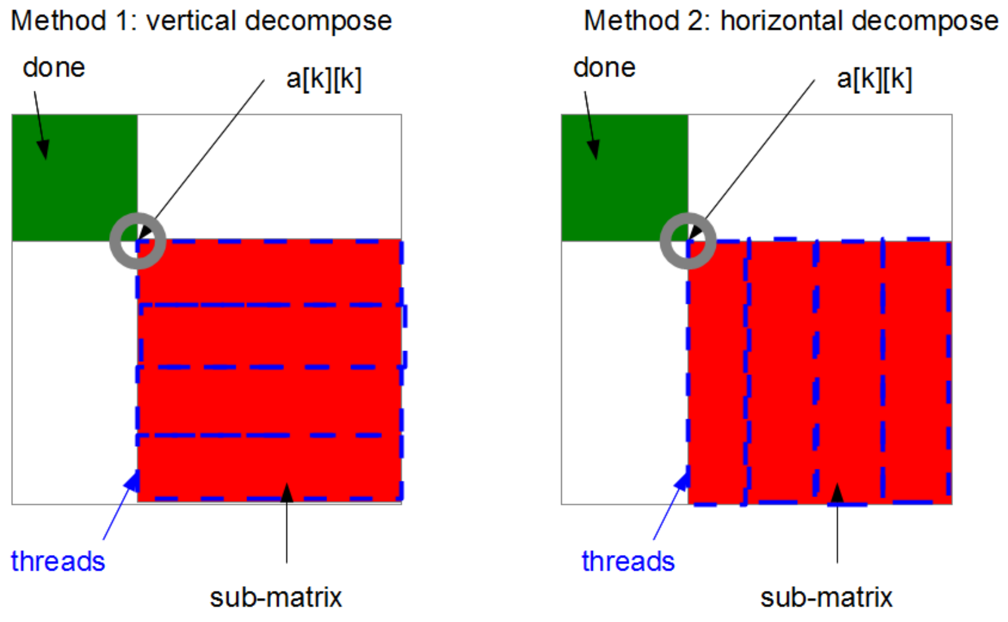


Fig.2

In a horizontal decomposition (Method 1), as shown in the diagram, each thread works on a certain number of rows of *a* [*i'*][*j'*]. Since there is no dependencies between the linear operations on each row, these thread can work concurrently. After everyone is done, the result is synchronized, and the program moves on to the next diagonal element. Alternatively, each thread can work on a certain number of columns of *a* [*i'*][*j'*] (method 2). In this decomposition, thread 0 first needs to broadcast the pivot value for a certain row to all the other threads. Then all the other threads can work on their portion of this row. A barrier is placed at the end of the operation to synchronize the row. After everyone finished, the program moves on to the next row, etc. Both of the two methods divide the sub-matrix *a* [*i'*][*j'*] into *n* parts where *n* is the number of threads. One can choose to always divide the entire matrix into *n* parts, but since we only work with a portion of the matrix at a time, many threads would end up idling during the computation. For Method 2, there are roughly *nsize^2* synchronizations needed instead of *nsize* in the case of Method 1. If the overhead of each waiting is non trivial, Method 2 can get much slower comparing with Method 1, especially when the number of threads is large. Since we have already shown that there is only a tiny portion of the computation time spent on swapping, it is probably beneficial to not parallelize the swapping phase to minimize the overhead introduced by creating threads. This results in Method 3, where only the elimination phase is parallelized by vertical decomposition. Another caveat is that when the sub-matrix has a number of rows or columns smaller than the number of threads, we only create as many threads as the remaining rows/columns to avoid multiple threads getting assigned with the same row/column.

As for the correctness, we implemented verification function where the vector subtraction *ax-b* (where *a* and *b* are the original matrix and right hand side vector, not the converted ones) is compared with a small tolerance. To verify that the code works with any input, *a* and *b* are initialized using random numbers. Every runs listed below have been verified to have the correct solution. To test the correctness of the code, simply put the verification flag to 1. Once executed, it will produce a message “solution verified” or “wrong solution” depending on the computed result.

Environment

The experiment is conducted on 8 processor cycle machines: Intel(R) Xeon(TM) CPU operates at 1200MHz, cache size 2048 KB, address sizes 36 bits physical, 48 bits virtual, Linux OS, and on the niagara1, a Sun T1000 multiprocessors with 8 4-way multithreaded cores, the 32 virtual processors are sparcv9, operates at 1000 MHz, running SunOS.

Compiler options on Linux machines:

```
CC = gcc
CFLAGS = -g -O3 -D_REENTRANT
LFLAGS = -lm -lpthread
```

Compiler options on SunOS:

```
CC = gcc
CFLAGS = -Wa,-xarch=v8plus -O3
LFLAGS = -lm -lpthread
```

Run options:

-s value of *nsize*
-p number of threads
-v 0 for Method 1, 1 for Method 2, 2 for Method 3.

Input Parameters

nsize = 128, 256, 512, 1024, 2048
number of threads = 1 to 8 on Linux machines, 1, 2, 4, 8, 16, 32 on SunOS machines
solution verification is done for all of the runs presented

Performance analysis 1: parallel speed up

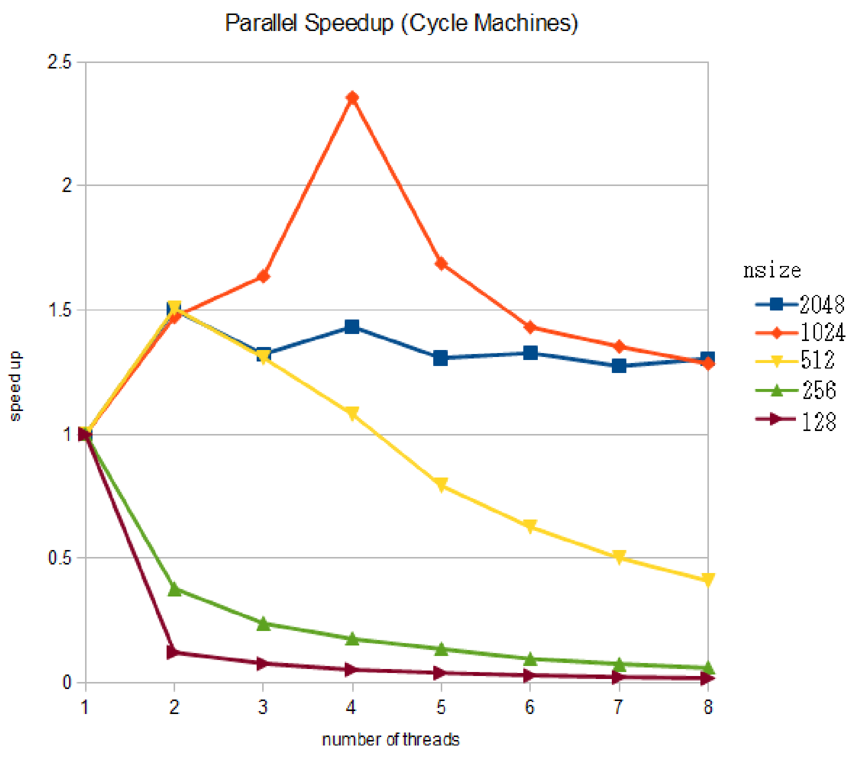
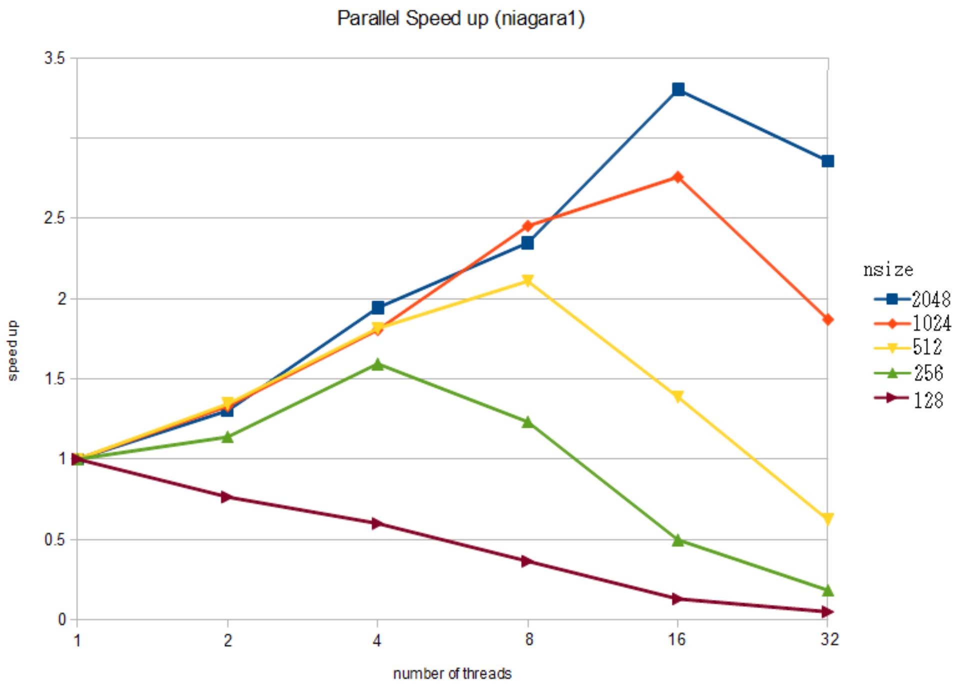


Fig.3
Fig.4



Performance analysis 2: Method 1 (vertical decomposition) vs Method 2 (horizontal decomposition)

Here we compare the vertical decomposition method and the horizontal

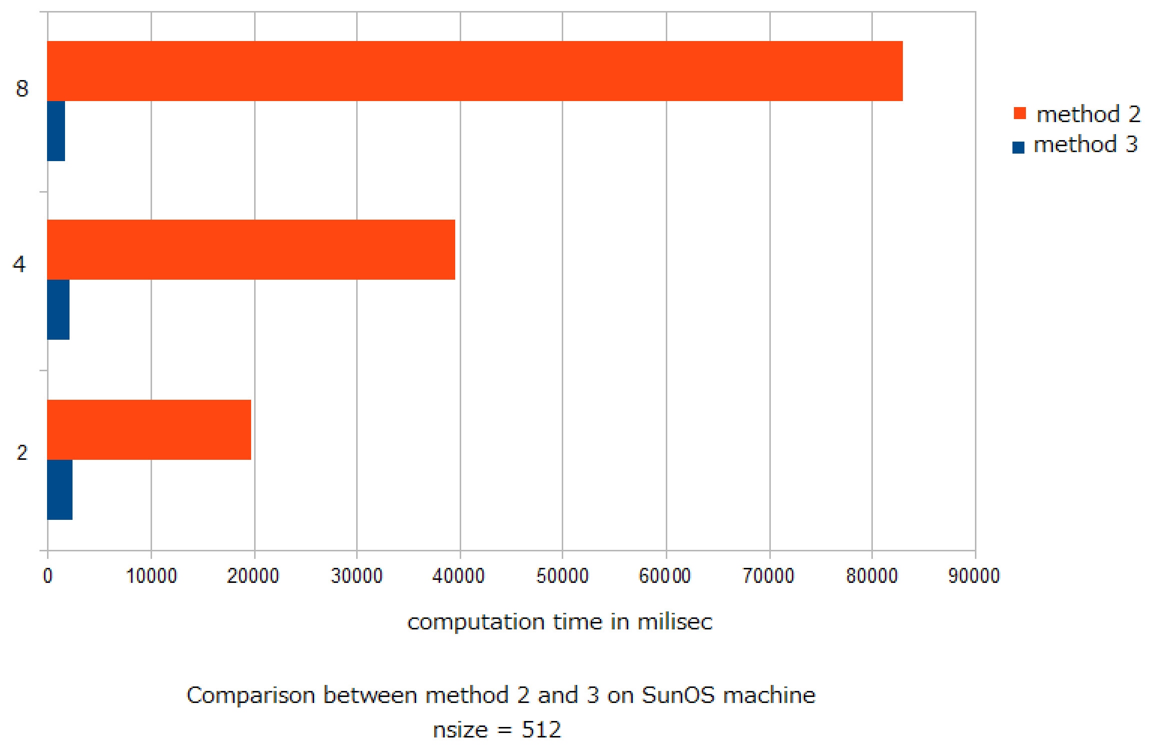


Fig.5

decomposition method introduced in the Strategies section. The above figure shows time spent on the niagara1 machine when solving the *nsize = 512* problem on a range of number of processors. We observe that the horizontal decomposition is about 2 orders of magnitudes slower on 8 threads. This is not surprising since this decomposition strategy requires *nsize^2* waits on each thread instead of *nsize* as in the case of vertical decomposition. Also we observe the computation time spent increases as the number of threads increases. Again, since the time spent on waiting relies on the number of threads, this result is expected. In general, we prefer the vertical decomposition as in Method 1.

Performance analysis 3: Parallel profile for Method 1

In this measurement we show the profile of costs on different sections of the code when Method 1 is implemented. The following figures show these costs on the SunOS as well as the Linux machines. These results suggest that the time spent on swapping increases with the number of threads. The first reason is that since the swapping tested in this code does not actually happen for most of the rows. Therefore the benefits we gain from parallelizing this section of code is minimal, and most of the cost on parallel swapping comes from the overhead of blocking. The second reason is that the swapping has true dependence between different rows. We are forced to use horizontal decomposition of the sub-matrix, which is not an efficient approach as demonstrated in the previous section.

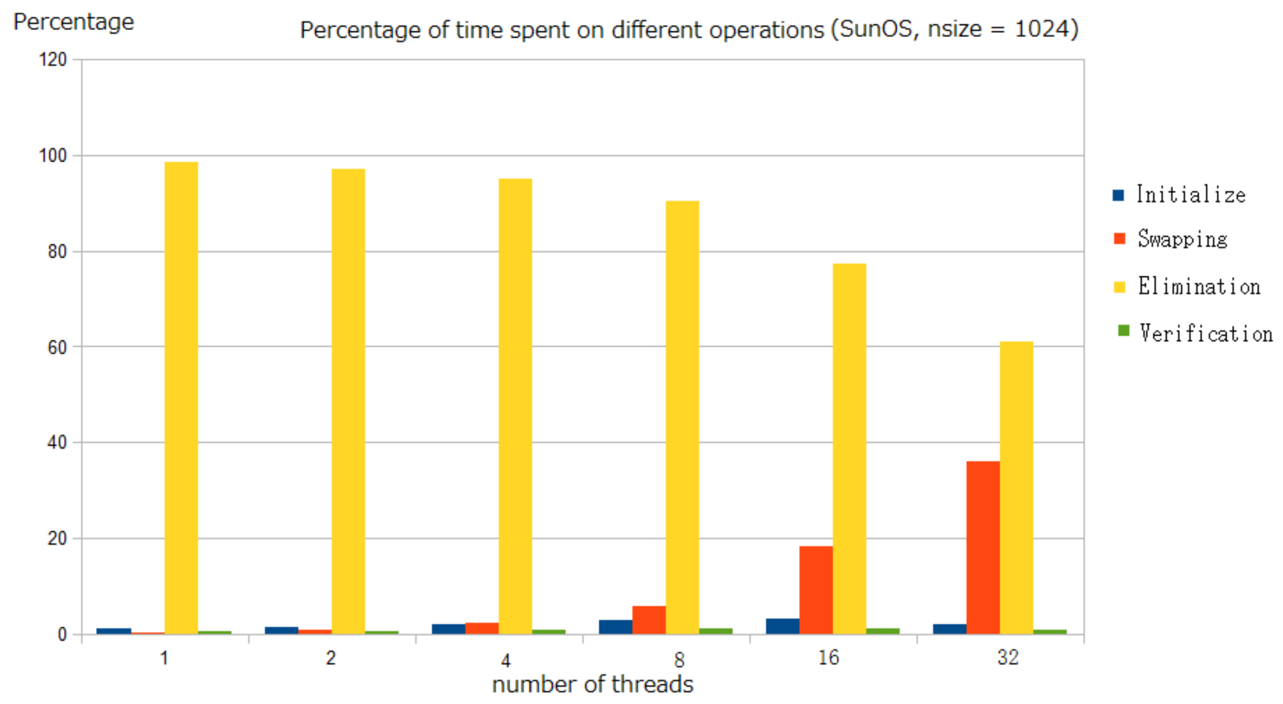


Fig.6

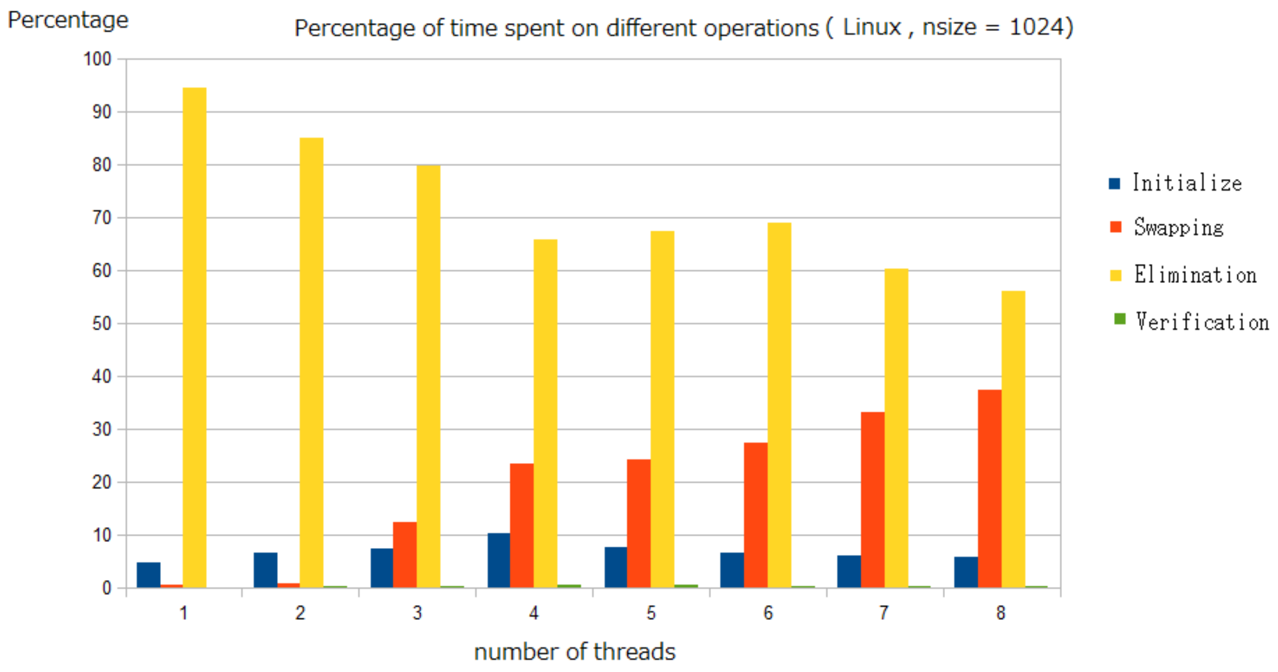


Fig.7

Performance analysis 3: Performance enhancement for Method 1

In this section we benchmark the parallel performance on both the SunOS and Linux machines for Method 3, where we remove the parallelism in the swapping section in Method 1. The elimination section is paralleled by vertical decomposition, as in Method 1.

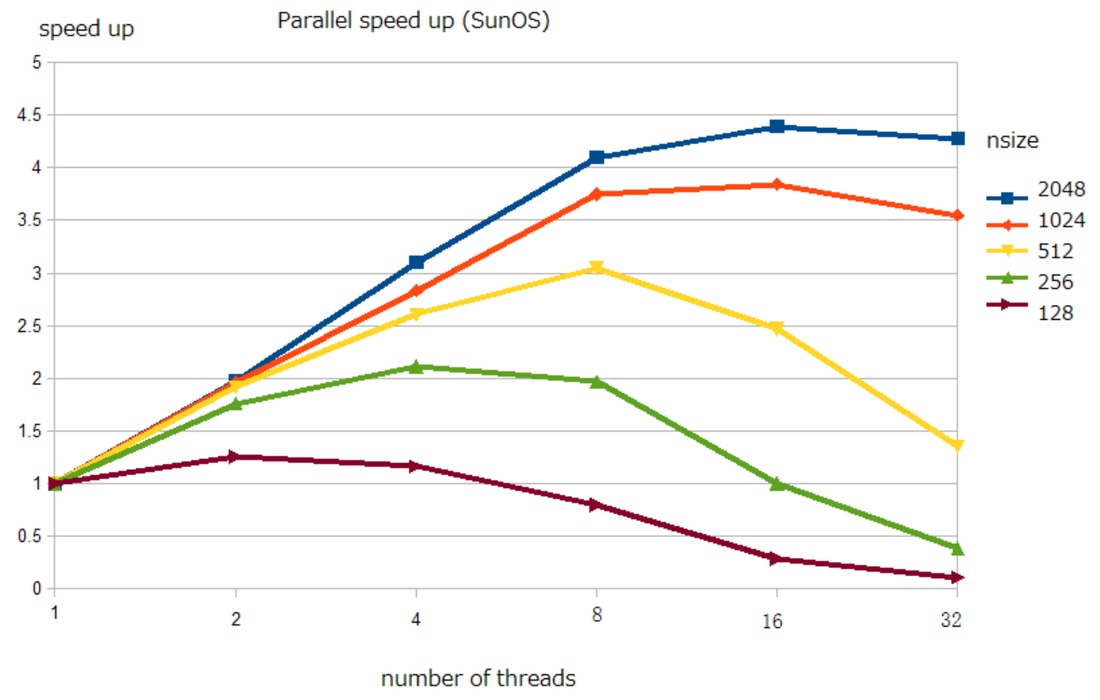


Fig.8

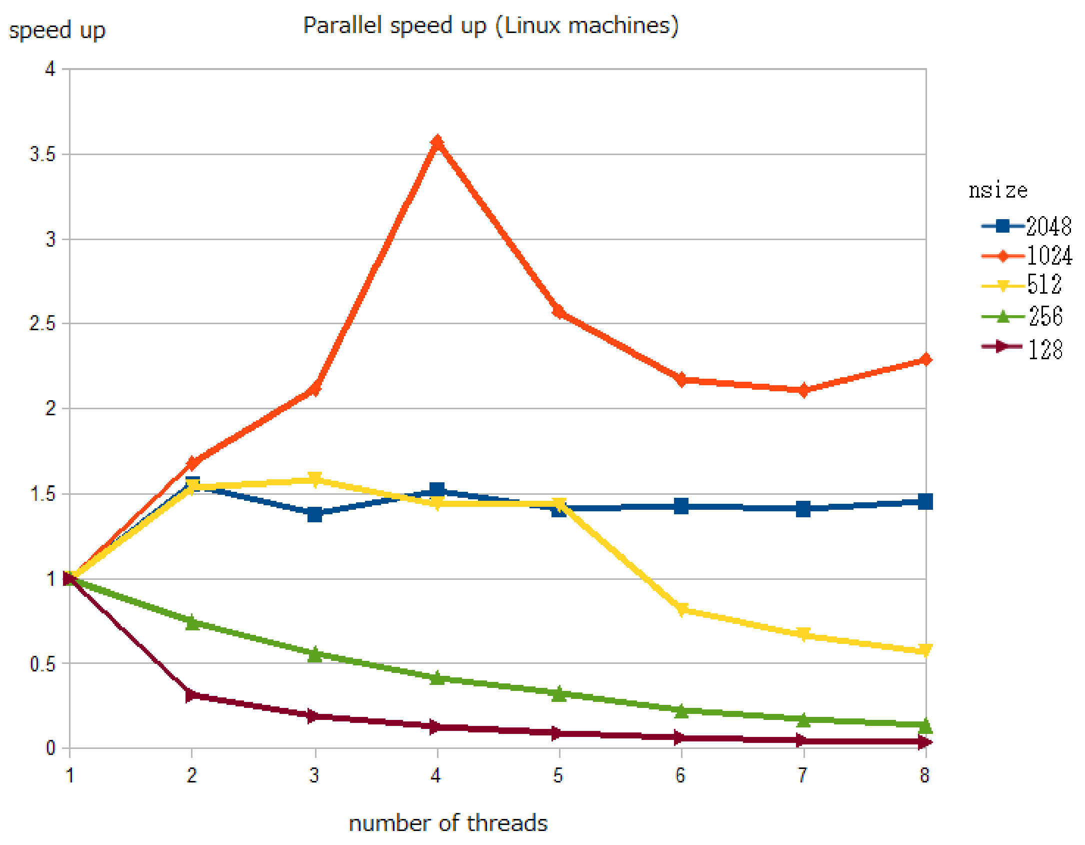


Fig.9

Analysis and Summary

In this summary we present three parallel strategies: Method 1 with horizontal decomposition on swapping and vertical decomposition on elimination; Method 2 with horizontal decomposition on both the swapping and elimination; Method 3 with only vertical decomposed on elimination. We divide the program into different sections and find out the time spent on each parts.

As for the decomposition techniques themselves, the horizontal decomposition requires waits on each iteration of the sub-matrix considered because each threads can only work on a portion of a given row. This results in *nsize^2* waits which can be costly if the matrix is large. On the other hand, the vertical decomposition only requires one synchronization per sub-matrix. However, it seems the swapping cannot be done in a vertical decomposition. The result of swapping a certain row depends on the previous swapping operations. For instance, if we try to swap multiple rows simultaneously on different threads, thread 0 may attempt to swap row 100 with row i while thread 1 may attempt to swap row 10 with row 100 at the same time (true dependency). Since the time spent doing swapping is small compared to elimination, it is beneficial to not do the horizontal decomposition on swapping at all. The elimination phase has concurrency that each row manipulation can be executed independently, and each element of a row can be manipulated independently. Therefore both the horizontal decomposition and vertical decomposition works. By looking at the breakdown of computation time, we see that the cost on doing parallel swapping is close to O(p), where p is the number of threads. With 16 or 32 threads on the SunOS machines, this cost greatly reduce the performance for Method 1, as we see a significant drop in speedup for $p > 8$ for the SunOS machines. This reduction is mitigated by implementing Method 3, as indicated in Fig.8. Although the Linux machines run the program faster than the due to faster processors in sequential execution, they do not exhibit good scalability as indicated in Fig.9. They also have heavier dependency on the input parameters probably due to architecture: it shows better speedup when there are four threads. The SunOS on the other hand offers cleaner results. Another alternative to Method 2 is to broadcast the pivot values at the start of the elimination so that threads do not have to wait for the current sub-matrix iteration to complete to start on the next elimination. This can remove the *nsize* waits per iteration, but will introduce an additional memory cost of *sizeof(double)*nsize*. In conclusion, Method 3 demonstrates the best parallel efficiency in the experiment, while Method 1 could be efficient when a lot of swappings are happening. Method 2 can be improved if the *nsize^2* waiting cost is reduced. Lastly we note that in our program, the number of tasks is limited by the size of the sub-matrix n': if $n' < p$, we only create n' tasks to prevent multiple threads writing to the same row.

We could hybrid the two decomposition methods: each thread can work on a certain number of rows of the given sub-matrix and while working, each thread can create second level threads so that each second level thread works on a portion of the rows. If the thread number is large enough so that each thread works on 2 or 3 rows but with a large number of columns, then it is possibly beneficial to divide the work on each row into multiple threads.