A Parallel LLL Algorithm

Yixian Luo
Department of Computing and Software,
McMaster University,
1280 Main St. West
Hamilton, Ontario, L8S 4K1, Canada.

Sanzheng Qiao
Department of Computing and Software,
McMaster University,
1280 Main St. West
Hamilton, Ontario, L8S 4K1, Canada.
qiao@mcmaster.ca

ABSTRACT

The LLL algorithm is a well-known and widely used lattice basis reduction algorithm. In many applications, its speed is critical. Parallel computing can improve speed. However, the original LLL is sequential in nature. In this paper, we present a multi-threading LLL algorithm based on a recently improved version: an LLL algorithm with delayed size reduction.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and vector implementations; G.1.2 [**Numerical Analysis**]: Approximations—*Least squares approximation*; G.1.6 [**Numerical Analysis**]: Optimization—*Least squares methods*

General Terms

Algorithms

Keywords

Algorithms, parallel computing, multi-threading, lattice basis reduction, LLL algorithm

Conference topics: Parallel computing, algorithms.

1. INTRODUCTION

The LLL algorithm, introduced by Lenstra, Lenstra, and Lovász [3] in 1982, is a method for reducing lattice bases. It has received a lot of attention as an effective numerical tool for preconditioning the integer least squares problem. In 2008, Luk and Tracy [5] presented a matrix version of the LLL algorithm. In this paper, we first present the original LLL algorithm in Section 2 and the LLL algorithm with delayed size-reduction in Section 3, then propose a parallel LLL algorithm in Section 4 and its Pthread implementation in Section 5.

C^3S^2E-11 2011, May 16-18, Montreal [QC, CANADA]
Editors: Abran, Desai, Mudur
Copyright 2011 ACM 978-1-4503-0626-3/11/05 $10.00.

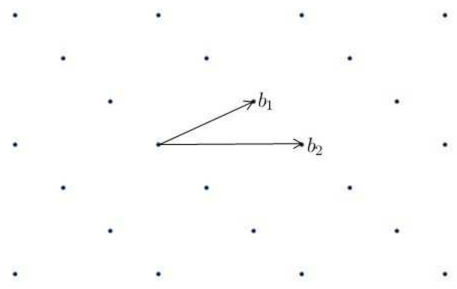


Figure 1: The lattice points generated by the column vectors b_1 and b_2 of B (1).

1.1 Bases for Lattices

Let n be a positive integer, a lattice is a subset of the n-dimensional real vector space $\Re^n$, defined by

$$L = \{Bz\},$$

where z are all integer n-vectors and B is an m-by-n ($m \geq n$) matrix with real entries and of full column rank, called lattice generator matrix.

Let $B = [b_1, b_2, ..., b_n]$, then $b_1, b_2, ..., b_n$ are linearly independent columns. They form a basis for L. For example, the matrix

$$B = \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix} \tag{1}$$

generates the lattice points in Figure 1.

1.2 Reduced Bases

A lattice may have more than one basis. For example, the matrix

$$C = \begin{bmatrix} c_1 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix} \tag{2}$$

also generates the lattice in Figure 1. Figure 2 depicts the columns of B and C, and the same lattice as in Figure 1.

Since a lattice L may have more than one basis, some of the bases are more desirable than others. We can expect that short basis vectors are close to orthogonal. As shown in Figure 2, c_1 and c_2 are shorter basis vectors than b_1 and b_2 and they are "more orthogonal" than b_1 and b_2. Short bases are called reduced bases.

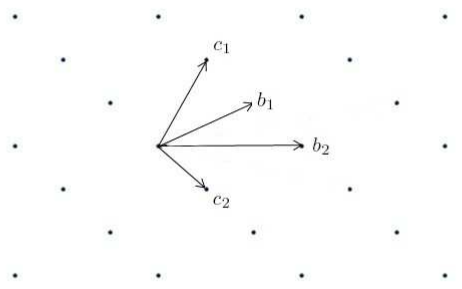


Figure 2: The lattice and the columns of B and C. (2).

1.3 The Gram-Schmidt Process

The Gram-Schmidt process is a method for constructing an orthogonal (or orthonormal) basis for any subspace of $\Re^n$, given a set of linearly independent vectors. It iteratively constructs vectors orthogonal to all of the vectors that have already been constructed [2]. In other words, it triangularizes a matrix using orthogonal transformations.

Since basis vectors are linearly independent, we can apply the Gram-Schmidt process to them. Now we present this process. Let $b_1, b_2, ..., b_n$ form a basis for a lattice L and

$$\begin{aligned} b_1^* &= b_1, \\ b_2^* &= b_2 - \frac{b_2^T b_1^*}{(b_1^*)^T b_1^*} b_1^*, \\ b_3^* &= b_3 - \frac{b_3^T b_2^*}{(b_2^*)^T b_2^*} b_2^* - \frac{b_3^T b_1^*}{(b_1^*)^T b_1^*} b_1^*, \\ &\vdots \\ b_n^* &= b_n - \frac{b_n^T b_{n-1}^*}{(b_{n-1}^*)^T b_{n-1}^*} b_2^* - \cdots - \frac{b_n^T b_1^*}{(b_1^*)^T b_1^*} b_1^*, \end{aligned}$$

then $b_1^*, b_2^*, ..., b_n^*$ are orthogonal.

1.4 Unimodular Matrix

Two different bases for a same lattice are related by an integer matrix whose inverse is also an integer matrix. For example, the matrix B in (1) and the matrix C in (2) are related by

$$C = BM, \quad \text{where} \quad M = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}.$$

Note that $\det(M) = 1$. Specifically, we have the following definition.

Definition 1 (Unimodular matrix). *A nonsingular integer matrix M is called unimodular if* $\det(M) = \pm 1$.

2. THE LLL ALGORITHM

The LLL algorithm first applies the Gram-Schmidt process to an m-by-n ($m \geq n$) lattice generator matrix B:

$$\begin{aligned} B &= \begin{bmatrix} b_1 & ... & b_n \end{bmatrix} \\ &= \begin{bmatrix} b_1^* & ... & b_n^* \end{bmatrix} \begin{bmatrix} 1 & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{b_1^*}{\|b_1^*\|_2} & \cdots & \frac{b_n^*}{\|b_n^*\|_2} \end{bmatrix} \operatorname{diag}(\|b_i^*\|_2) \begin{bmatrix} 1 & \cdots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \\ &:= QD^{1/2}U, \end{aligned} \tag{3}$$

where Q has orthonormal columns, $D = \operatorname{diag}(d_i)$ with $d_i = \|b_i^*\|_2^2$, and $U = [u_{i,j}]$ is upper triangular with a unit diagonal.

In terms of the decomposition (3), we have the following definition of a size-reduced basis.

Definition 2 (Size-reduced). *A basis $\{b_1, b_2, ..., b_n\}$ for a lattice is called size-reduced if the U in the decomposition (3) satisfies*

$$|u_{i,j}| \leq \frac{1}{2}, \quad \textit{for} \quad 1 \leq i < j \leq n$$

In [3], a notion of reduced basis, stronger than size-reduced, is defined as follows

Definition 3 (LLL-reduced). *A basis $\{b_1, b_2, ..., b_n\}$ for a lattice is called LLL-reduced if the U and D in the decomposition (3) satisfy the two conditions:*

$$|u_{i,j}| \leq \frac{1}{2}, \quad \textit{for} \quad 1 \leq i < j \leq n \tag{4}$$

and

$$d_i + u_{i-1,i}^2 d_{i-1} \geq \omega d_{i-1}, \quad \textit{for} \quad 2 \leq i \leq n, \tag{5}$$

where $1/4 < \omega < 1$.

The condition (4) ensures that an LLL-reduced basis is size-reduced. The condition (5) requires that d_i be loosely ordered.

After the Gram-Schmidt process, the LLL algorithm computes a new basis that satisfies the above two conditions (4) and (5).

If $|u_{i,j}| > 1/2$ for some $j > i$, a procedure called Reduce(i, j) is applied to ensure condition (4).

Procedure 1 (Reduce(i, j)). *Given a lattice generator matrix B, upper triangular U in the decomposition (3), and a unimodular matrix M, form an n-by-n unimodular matrix $M_{ij} = I_n - \gamma e_i e_j^T$ where $\gamma = \lceil u_{i,j} \rfloor$ is an integer closest to $u_{i,j}$ and e_i is the ith unit vector. Apply M_{ij} to U, B and M:*

$$U \leftarrow UM_{ij}, \quad B \leftarrow BM_{ij}, \quad \textit{and} \quad M \leftarrow MM_{ij}.$$

If $|u_{i,j}| > 1/2$ for some $j > i$, then in the updated U, $|u_{i,j}| \leq 1/2$.

If the condition (5) does not hold for some $2 \leq i \leq n$, another procedure called SwapRestore(i) is applied.

Procedure 2 (SwapRestore(i)). *Given a lattice generator matrix B, diagonal D and upper triangular U in the* *decomposition (3), and a unimodular matrix M, let $\mu = u_{i-1,i}$, compute $\hat{d}_{i-1} = d_i + \mu^2 d_{i-1}$ and $\xi = \mu d_{i-1}/\hat{d}_{i-1}$, and the new*

$$d_i \leftarrow d_{i-1} d_i / \hat{d}_{i-1} \text{ and } d_{i-1} \leftarrow \hat{d}_{i-1},$$

then form

$$X_i = \begin{bmatrix} \mu & 1-\mu\xi \\ 1 & -\xi \end{bmatrix}.$$

Swap the columns i and $i-1$ of U, B and M, then update U:

$$\begin{aligned} U &\leftarrow \begin{bmatrix} I_{i-2} & & & \\ & \xi & 1-\xi\mu & \\ & 1 & -\mu & \\ & & & I_{n-i} \end{bmatrix} U \\ &= \operatorname{diag}(I_{i-2}, X_i^{-1}, I_{n-i}) U. \end{aligned}$$

It can be verified that the new D and U satisfy the condition (5).

Now, we have the LLL algorithm.

ALGORITHM 1 (LLL ALGORITHM). *Given an m-by-n, where $m \geq n$, lattice generator matrix B, compute the D and U in the decomposition (3) of B using the Gram-Schmidt method;*

```
set $M \leftarrow I$;
$k \leftarrow 2$;
while $k \leq n$
    if $|u_{k-1,k}| > 1/2$
        Reduce($k-1, k$);
    endif
    if $d_k < (\omega - u_{k-1,k}^2) d_{k-1}$
        SwapRestore($k$);
        $k \leftarrow \max(k-1, 2)$;
    else
        for $i = k-2$ down to 1
            if $|u_{i,k}| > 1/2$
                Reduce($i, k$);
            endif
        endfor
        $k \leftarrow k+1$;
    endif
endwhile
```

As we can see, the original LLL algorithm is very sequential.

3. THE LLL ALGORITHM WITH DELAYED SIZE-REDUCTION

Recently, Zhang, Wei, and Qiao [7] proposed a modified LLL algorithm, which can save a significant amount of operations and also provides a basis for a parallel implementation. We begin with an example to illustrate the idea. Let $\omega = 3/4$ and a lattice basis matrix

$$B = \begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix}. \tag{6}$$

We trace the LLL algorithm but only give the values of D and U in every step according to the decomposition (3).

$$\begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix} \xrightarrow{\text{Gram-Schmidt (S1)}}$$

$$D = \operatorname{diag}\left(3, \tfrac{14}{3}, \tfrac{9}{14}\right),\ U = \begin{bmatrix} 1 & \frac{1}{3} & \frac{14}{3} \\ 0 & 1 & \frac{13}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \text{ do nothing (S2)}}$$

$$\xrightarrow{k=3 \text{ Reduce(2,3) (S3)}}$$

$$D = \operatorname{diag}\left(3, \tfrac{14}{3}, \tfrac{9}{14}\right),\ U = \begin{bmatrix} 1 & \frac{1}{3} & \frac{13}{3} \\ 0 & 1 & -\frac{1}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{\text{SwapRestore(3) (S4)}}$$

$$D = \operatorname{diag}\left(3, \tfrac{2}{3}, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & \frac{13}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \quad \text{Reduce(1,2) (S5)}}$$

$$D = \operatorname{diag}\left(3, \tfrac{2}{3}, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & \frac{1}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{\text{SwapRestore(2) (S6)}}$$

$$D = \operatorname{diag}\left(1, 2, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \quad \text{Reduce(1,2) (S7)}}$$

$$D = \operatorname{diag}\left(1, 2, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=3 \quad \text{do nothing (S8)}} \text{end.}$$

After (S4), the algorithm found that $|u_{1,2}| = 13/3 > 1/2$, so we had step (S5) to reduce $u_{1,2}$. Then step (S6) was applied because $d_2 < (3/4 - u_{1,2}^2) d_1$. After (S6), we found that $|u_{1,2}| = 1$, which is larger than $1/2$, so Reduce$(1,2)$ was applied again. If we did not reduce $u_{1,2}$ in step (S5), instead swapped d_2 and d_1 first, we would just need to apply Reduce$(1,2)$ once. This means that some size reductions can be delayed until the condition (5) is satisfied first. This method is called the LLL algorithm with delayed size-reduction [7].

Before presenting the new algorithm, we introduce the following procedure, where $\gamma = \lceil u_{k-1,k} \rfloor$ is an integer closest to $u_{k-1,k}$.

PROCEDURE 3 (REDUCESWAPRESTORE(i, γ)). *Given a lattice generator matrix B, diagonal D and upper triangular U in the decomposition (3), and a unimodular matrix M, let $\mu = u_{i-1,i}$, compute $\hat{d}_{i-1} = d_i + (\mu - \gamma)^2 d_{i-1}$ and $\xi = \frac{(\mu-\gamma) d_{i-1}}{\hat{d}_{i-1}}$, and the new*

$$d_i \leftarrow \frac{d_{i-1} d_i}{\hat{d}_{i-1}} \text{ and } d_{i-1} \leftarrow \hat{d}_{i-1},$$

then form

$$\begin{aligned} \widehat{X}_i &= \begin{bmatrix} \mu - \gamma & 1 - \mu\xi + \gamma\xi \\ 1 & -\xi \end{bmatrix} \\ &= \begin{bmatrix} 1 & -\gamma \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mu & 1-\mu\xi \\ 1 & -\xi \end{bmatrix}. \end{aligned}$$

Let

$$P = \begin{bmatrix} 1 & -\gamma \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and $\Pi_i = \mathrm{diag}([I_{i-2}\ P\ I_{n-i}])$*, apply* Π_i *to* U*,* B *and* M*:*

$$U \leftarrow U\Pi_i, \quad B \leftarrow B\Pi_i, \quad M \leftarrow M\Pi_i.$$

Then apply $\widehat{X}_i^{-1}$ *to* U*:*

$$\begin{aligned} U &\leftarrow \begin{bmatrix} I_{i-2} & & & \\ & \xi & 1-\xi\mu+\gamma\xi & \\ & 1 & \gamma-\mu & \\ & & & I_{n-i} \end{bmatrix} U \\ &= \mathrm{diag}(I_{i-2}, \widehat{X}_i^{-1}, I_{n-i})U. \end{aligned}$$

We can see that the above procedure is an integration of Reduce$(i-1, i)$ and SwapRestore(i).

Now, we present the LLL algorithm with delayed size-reduction.

Algorithm 2. *(LLL Algorithm with delayed size-reduction) Given an m-by-n,* $m \geq n$*, lattice generator matrix* B*, compute* D *and* U *in the decomposition (3) of* B *using the Gram-Schmidt method;*

```
1   set $M \leftarrow I$;
2   $k \leftarrow 2$;
3   while $k \leq n$
4       $\gamma = \lceil u_{k-1,k} \rfloor$;
5       if $d_k < (\omega - (u_{k-1,k} - \gamma)^2) d_{k-1}$
6           ReduceSwapRestore($k$,$\gamma$);
7           $k \leftarrow \max(k-1, 2)$;
8       else
9           $k \leftarrow k+1$;
10      endif
11  endwhile
12  for $k \leftarrow 2 : n$
13      for $i = k-1$ down to 1
14          if $|u_{i,k}| > 1/2$
15              Reduce($i, k$);
16          endif
17      endfor
18  endfor
```

The above algorithm consists of two parts. The first part enforces the condition (5) and the second part does the size reduction.

Now we trace the LLL algorithm with delayed size-reduction using the same matrix B in (6):

$$\begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix} \xrightarrow{\text{Gram-Schmidt (P1)}}$$

$$D = \mathrm{diag}\left(3, \tfrac{14}{3}, \tfrac{9}{14}\right),\ U = \begin{bmatrix} 1 & \frac{1}{3} & \frac{14}{3} \\ 0 & 1 & \frac{13}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \text{ do nothing (P2)}}$$

$$\xrightarrow{k=3 \text{ ReduceSwapRestore}(3,1)\ \text{(P3)}}$$

$$D = \mathrm{diag}\left(3, \tfrac{2}{3}, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & \frac{13}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \text{ ReduceSwapRestore}(2,4)\ \text{(P4)}}$$

$$D = \mathrm{diag}\left(1, 2, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \text{ do nothing (P5)}}$$

$$\xrightarrow{k=3 \text{ do nothing (P6)}} \text{endwhile}$$

$$\xrightarrow{\text{Reduce}(1,2)\ \text{(P7)}} D = \mathrm{diag}\left(1, 2, \tfrac{9}{2}\right),\ U = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

endfor $\rightarrow$ end.

As expected, in this process, the procedure Reduce$(1,2)$ was applied only once. Typically, the LLL algorithm with delayed size-reduction runs twice as fast as the original LLL algorithm.

4. A PARALLEL LLL ALGORITHM

For a parallel implementation of Algorithm 2, we call the first part of the algorithm, including the lines from 3 to 11, the Swap part, and the second part, the lines from 12 to 18, the Reduce part.

In the Swap part, we apply the odd-even ordering technique. We first execute ReduceSwapRestore(k, γ) for all even k such that the condition $d_k < (\omega - (u_{k-1,k} - \gamma)^2)d_{k-1}$ is true, then for all odd k such that the condition is true. The while-loop terminates when the condition is false for all k. Thus we design two for-loops, one for even k and one for odd k. While the two for-loops must be executed sequentially, the iterations in each for-loop can be executed in parallel. For example, the first for-loop, when $k = 2$, if the condition $d_2 < (\omega - (u_{1,2} - \gamma)^2)d_1$ is true, ReduceSwapRestore$(2, \gamma)$ permutes the columns 1 and 2 and modifies the rows 1 and 2. For $k = 4$, if the condition is true, ReduceSwapRestore$(4, \gamma)$ permutes the columns 3 and 4 and modifies the rows 3 and 4. Therefore, they can be executed in parallel, because there is no data conflict. Note that the permutation in ReduceSwapRestore$(4, \gamma)$ and the row operation in ReduceSwapRestore$(2, \gamma)$ can be executed in any order without affecting the result at the end of the loop, although they share variables $u_{1,3}$, $u_{1,4}$, $u_{2,3}$, and $u_{2,4}$. Similarly, the second for-loop for odd k can be executed in parallel.

In the Reduce part, the off diagonal elements $u_{i,j}$ are reduced. In order to modify $u_{i,j}$ and $u_{p,q}$ in parallel without data conflict, the indices i, j, p, q must be distinct, because reducing $u_{i,j}$ modifies the jth column using the ith column. Consider two elements $u_{i,j}$ and $u_{p,q}$ on a same antidiagonal, their indices satisfy $i + j = p + q$. Also, two different entries on a same antidiagonal must be in different rows and columns, that is, $i \neq p$ and $j \neq q$. Moreover, in the Reduced part, we modify the entries in the upper triangular part, implying that $i < j$ and $p < q$. Thus i, j, p, q are distinct and can be modified in parallel without data conflict. As shown in Figure 3, the elements on a same antidiagonal can be reduced in parallel. There are $2n - 3$ antidiagonals.

In summary, we have the following parallel algorithm.

Algorithm 3 (A Parallel LLL Algorithm). *Given an m-by-n,* $m \geq n$*, lattice generator matrix* B*, compute the* D *and* U *in the decomposition (3) of* B *using the Gram-Schmidt method;*

```
1   set $M \leftarrow I$;
2   $f \leftarrow$ false;
3   while $f \neq$ true
4       $f \leftarrow$ true;
5       for $k \leftarrow 2 : +2 : n$ (run in parallel)
6           $\gamma = \lceil u_{k-1,k} \rfloor$;
7           if $d_k < (\omega - (u_{k-1,k} - \gamma)^2) d_{k-1}$
8               $f \leftarrow$ false;
```

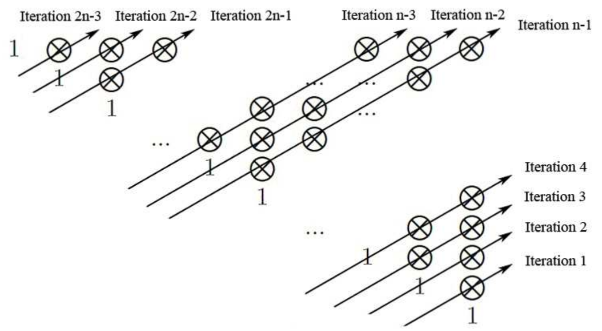


Figure 3: Reordering the procedure $Reduce(i, j)$. For legibility, we use $\otimes$ to represent $u_{i,j}$ in the relative position.

```
9            ReduceSwapRestore($k$,$\gamma$);
10        endif
11    endfor
12    for $k \leftarrow 3 : +2 : n$ (run in parallel)
13        $\gamma = \lceil u_{k-1,jk} \rfloor$;
14        if $d_k < (\omega - (u_{k-1,k} - \gamma)^2) d_{k-1}$
15            $f \leftarrow$ false;
16            ReduceSwapRestore($k$,$\gamma$);
17        endif
18    endfor
19 endwhile
20 for $k \leftarrow 2n - 3 : 1$
21    if $k \leq n - 1$
22        $h = 1$;
23    else
24        $h = k - n + 2$;
25    endif
26    for $i \leftarrow h : (k + 3)/2$ (run in parallel)
27        $j = k + 2 - i$;
28        if $|u_{i,j}| > 1/2$
29            Reduce($i, j$);
30        endif
31    endfor
32 endfor
```

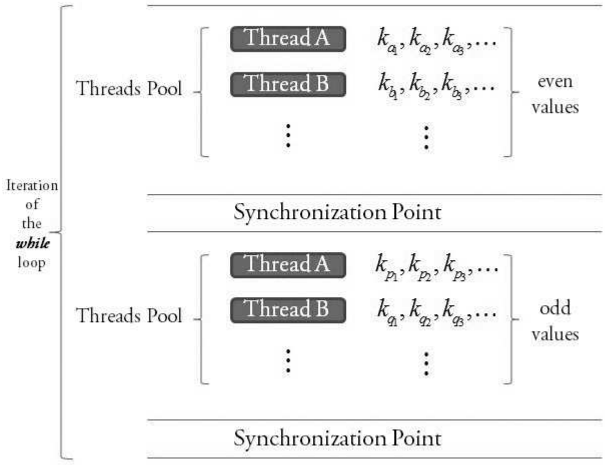
Note that the Reduce part in the algorithm is executed in the order shown in Figure 3.

5. IMPLEMENTATION OF THE PARALLEL LLL ALGORITHM WITH PTHREADS

Both threads and processes can provide parallel program execution, but a thread can be created with much less operating system overhead than a process. Managing threads requires fewer system resources than managing processes.

Pthreads is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. The primary motivation for considering the use of Pthreads on an SMP architecture is to achieve high performance [1].

The Pthreads library aims to be expressive as well as portable and provides a fairly comprehensive set of features to create, terminate, and synchronize threads and to prevent different threads from trying to modify the same values at the same time. It includes synchronization mechanisms mutexes, locks, condition variables, and semaphores.



Iteration of the *while* loop
Threads Pool
Thread A
$k_{a_1}, k_{a_2}, k_{a_3}, \ldots$
Thread B
$k_{b_1}, k_{b_2}, k_{b_3}, \ldots$
even values
Synchronization Point
Threads Pool
Thread A
$k_{p_1}, k_{p_2}, k_{p_3}, \ldots$
Thread B
$k_{q_1}, k_{q_2}, k_{q_3}, \ldots$
odd values
Synchronization Point

Figure 4: Implementation of the *Swap* part by using a threads pool.

There are no set rules for threading a program, but there are some models [6], such as the Boss/Worker Model, the Peer Model, and the Pipeline Model, that define how a threaded application delegates its work to its threads and how these threads communicate with each other.

Now we present our thread implement of the parallel Algorithm 3. As described in Section 4, the parallel algorithm consists of two parts: the Swap part and the Reduce part.

5.1 Implementation of the Swap Part

To implement the Swap part, we create a fixed number of threads, called Threads Pool method, as shown in Figure 4. The iterations of each of these two for-loops are divided into several groups, each of which contains a fixed number of the values of k. Each thread is assigned a group. So in each iteration of the while-loop, these threads are running in parallel. At the end of the first for-loop, we should make a synchronization point to ensure that all threads finish their work before starting the second for-loop. Otherwise, a thread may enter the second for-loop while some threads are still in the first for-loop, causing data conflict. Similarly, a synchronization point should be added at the end of the second for-loop.

Now, we address two issues, common in parallel programming.

- *Overhead*: The implementation has a few synchronization points. The there are two synchronization points in each iteration of the while-loop. When one thread holds the lock, other threads have to wait, introducing overhead. In addition, threads intercommunicate when the program enters into the synchronization part in order for each thread to know states of the others, introducing more overhead.

- *Load imbalance*: We know the major cost in this part is the procedure ReduceSwapRestore. When we group the values of k, there may be a load imbalance problem. For example, assuming $n = 20$ and we have four

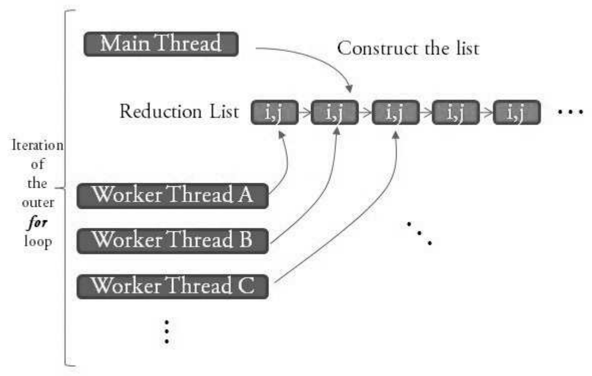


Figure 5: Implementation of the *Reduce* part by using a reduction list.

threads A, B, C, and D, so the values of k assigned to these threads for the first for-loop may be A:(2, 10, 18), B:(4, 12, 20), C:(6, 14), and D:(8, 16). If in an iteration of the while-loop only in positions 2, 10 and 18 ReduceSwapRestore is called, then the work load of thread A is much heavier than the other threads B, C, and D, providing no benefit while requiring synchronization overhead. Since the work distribution is unpredictable, there is no fixed solution for this problem.

5.2 Implementation of the Reduce Part

We can also apply the Threads Pool method to a parallel implementation of the Reduce part. After the threads check and reduce the entries on the same antidiagonal, we also need to add a synchronization point at the end. However, we have found that sometimes the Reduce procedure may not be called on an antidiagonal, especially when the antidiagonal is short. On the other hand, the major cost is the Reduce procedure, the cost of checking the condition and the simple assignment is minor, less than the cost of the synchronization between threads. So there is no need to use several threads on short antidiagonals.

This gives rise to another method, called Reduction List method, as shown in Figure 5. The main thread checks the conditions of all entries on an antidiagonal, records those entries which need to be reduced, and adds them into a list, but without calling the Reduce procedure. After checking all the entries on the current antidiagonal, if there are entries on the list, the worker threads remove one entry at a time from the list and reduce these entries until the list becomes empty. If there is no such entry, the program moves to the next antidiagonal without using any worker threads. In addition to reducing the synchronization cost, this method also balances thread workload by assigning the worker threads almost the equal number of entries that need to be reduced.

5.3 Testing

We tested the algorithm by using three different implementations as shown in Table 1, where serial denotes the sequential Algorithm 2, TP represents that both the Swap and Reduce parts are implemented by using the Threads Pool method, and RL means that the Swap part is implemented by using the Threads Pool method, and the Reduce part is implemented by using the Reduction List method.

Implementation Type	Swap Part	Reduce Part
serial	Sequential	Sequential
TP (2,3,4)	Thread Pool	Thread Pool
RL (2,3,4)	Thread Pool	Reduction List

Table 1: Implementations types. The numbers (2,3,4) represent the number of threads used. Since the test platform has a 4-core cpu, we use at most four threads.

Figures 6 to 15 show the results of 100 random matrices, 10 matrices for each dimension, by using the seven kinds of implementations in Table 1.

From Figures 6 to 9 we can see that no parallel implementation can outperform the serial implementation when dimension is smaller than (50,50). However, Figure 10 begins to show that the thread-pool implementation by using 2 threads (TP2) is better in some cases.

Figure 11 shows that the thread-pool implementation using 3 or 4 threads is a bad choice, but no clear winner for others.

From Figures 12 to 15we notice that the reduction list implementation using 3 or 4 threads (TR3 or TR4) is always the most efficient, showing that the reduction list implementation is a good choice for matrices of size larger than (100,100).

6. REFERENCES

[1] Blaise Barney. POSIX threads programming. Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/

[2] G.H. Golub and C.F. Van Loan. *Matrix Computations, Third Edition*. The Johns Hopkins University Press, Baltimore, MD, 1996.

[3] A.K. Lenstra, H.W. Lenstra, Jr. and L. Lovász. Factorizing polynomials with rational coefficients. *Mathematicsche Annalen*, **261**, 1982, 515–534.

[4] Franklin T. Luk, Sanzheng Qiao, Wen Zhang. A Lattice Basis Reduction Algorithm. *Institute for Computational Mathematics Technical Report 10-04*. Hong Kong Baptist University, Kowloon, Hong Kong, China, 2010.

[5] Franklin T. Luk and Daniel M. Tracy. An improved LLL algorithm. *Linear Algebra and its Applications*, **428**(2-3), 2008, 441–452.

[6] Bradford Nichols, Dick Buttlar, and Jackie Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc. 1996.

[7] Wen Zhang, Yimin Wei, and Sanzheng Qiao. LLL algorithm with delayed size reduction. Manuscript, 2010. Personal Communication.

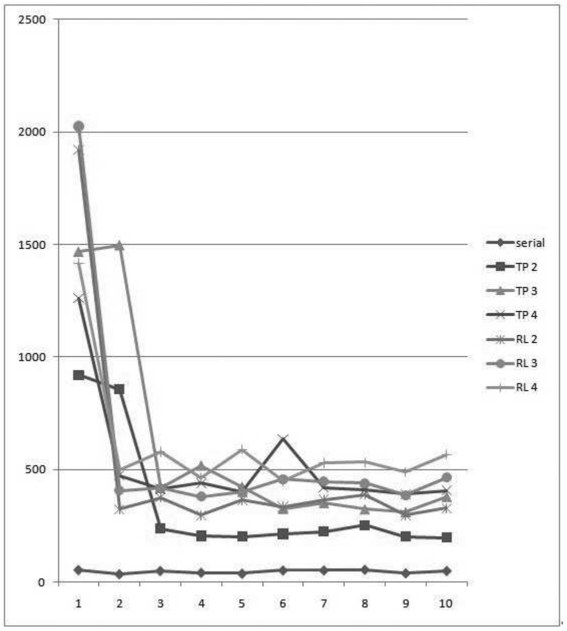


Figure 6: Costs of ten $10-by-10$ matrices

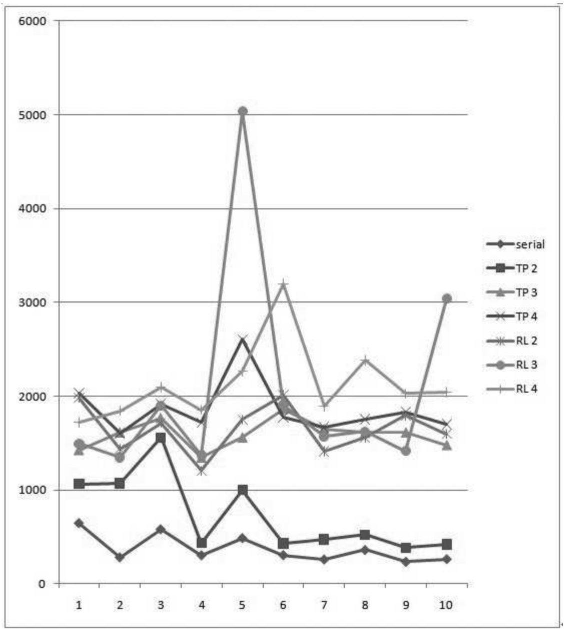


Figure 8: Costs of ten $30-by-30$ matrices

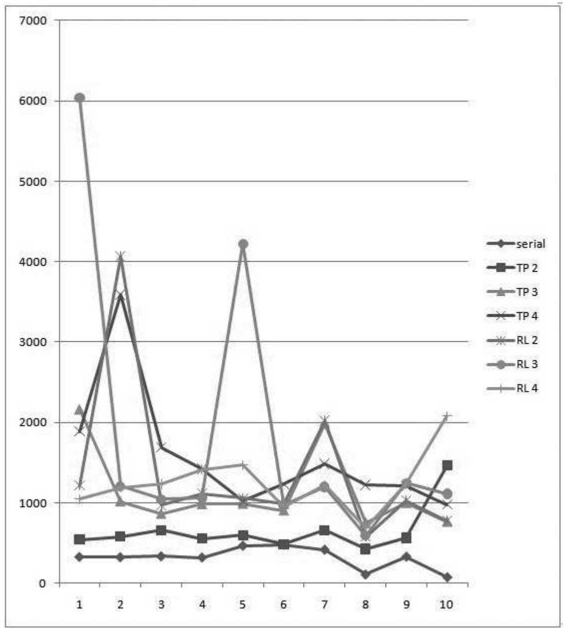


Figure 7: Costs of ten $20-by-20$ matrices

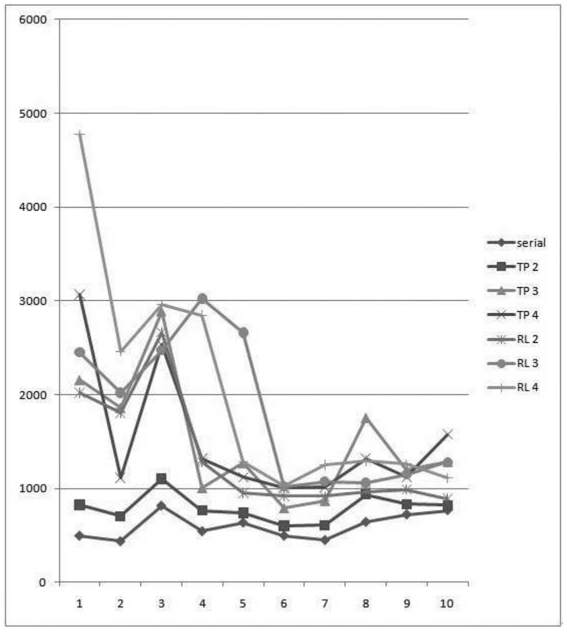


Figure 9: Costs of ten $40-by-40$ matrices

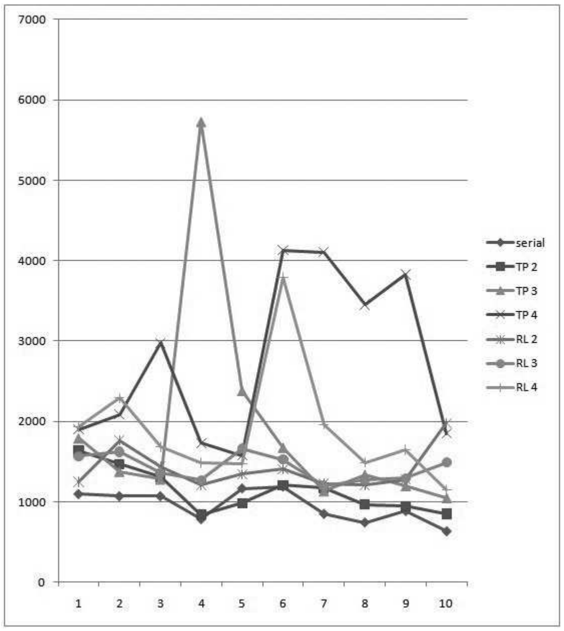


Figure 10: Costs of ten $50 - by - 50$ matrices

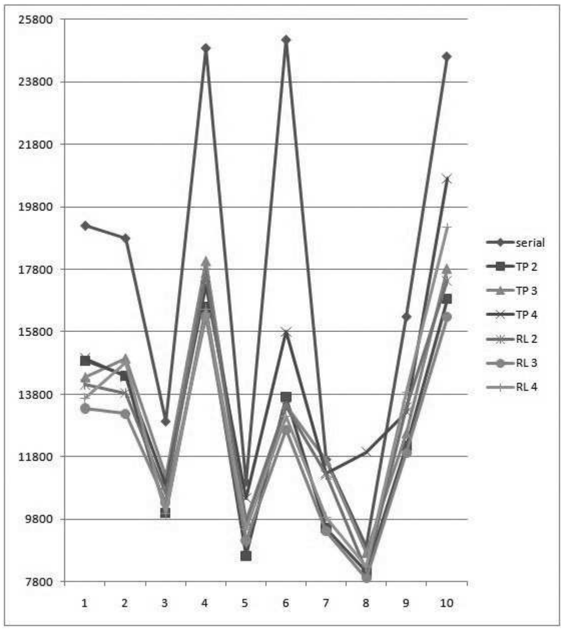


Figure 12: Costs of ten $200 - by - 200$ matrices

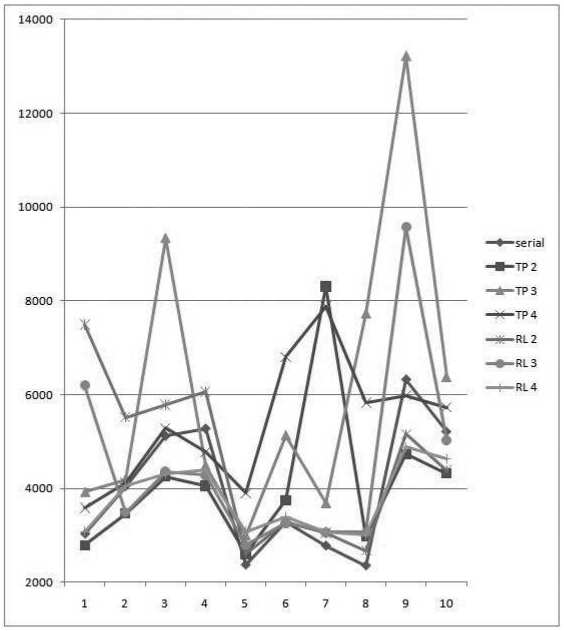


Figure 11: Costs of ten $100 - by - 100$ matrices

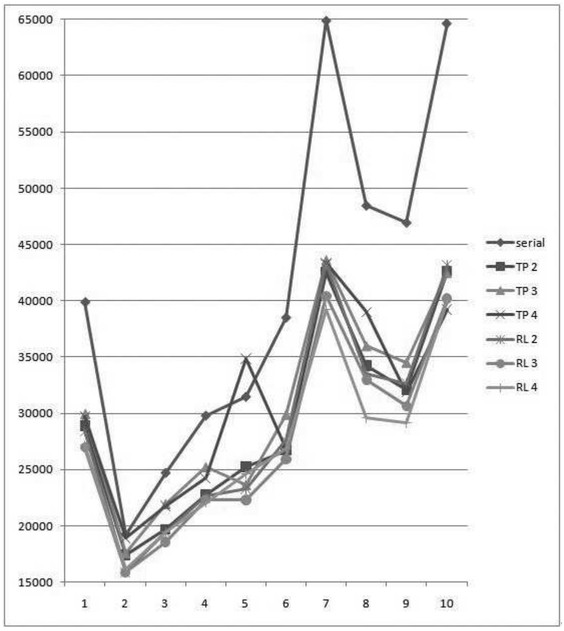


Figure 13: Costs of ten $300 - by - 300$ matrices

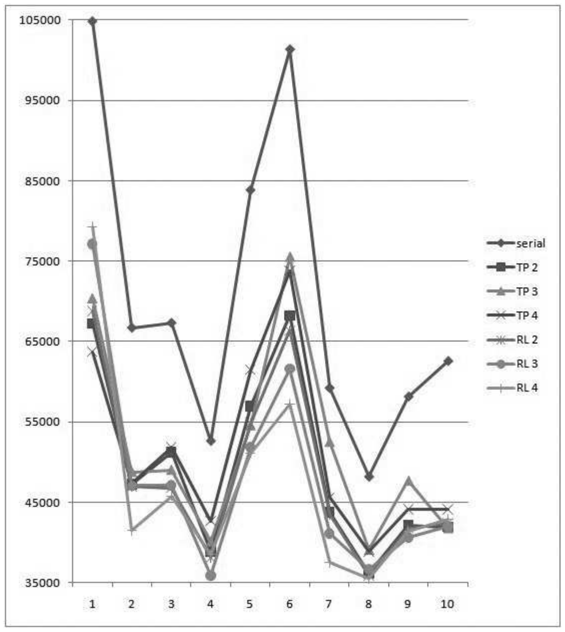


Figure 14: Costs of ten $400-by-400$ matrices

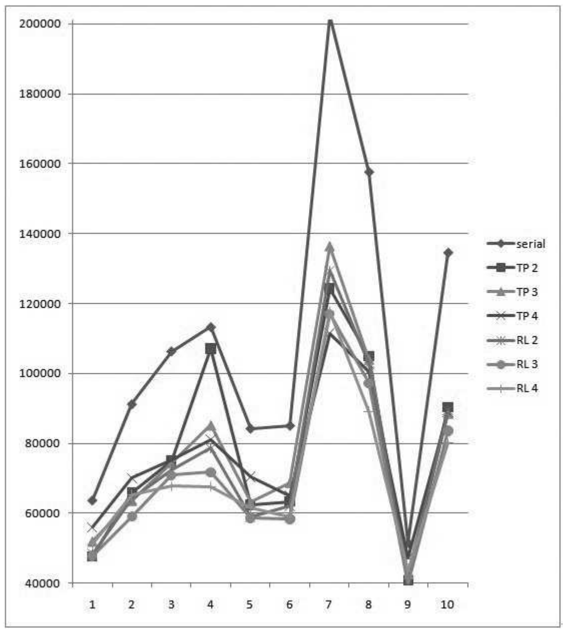


Figure 15: Costs of ten $500-by-500$ matrices