A Parallel LLL Algorithm

Yixian Luo and Sanzheng Qiao

Technical Report No. CAS-10-03-SQ

Department of Computing and Software, McMaster University

Hamilton, Ontario, L8S 4K1, Canada.

A Parallel LLL Algorithm

Yixian Luo[1] and Sanzheng Qiao[2]
[1,2]Department of Computing and Software, McMaster University
Hamilton, Ontario, L8S 4K1, Canada.
[1]luoy26@mcmaster.ca
[2]qiao@mcmaster.ca

Abstract

The LLL algorithm is a well-know and widely used lattice basis reduction algorithm. In many applications, its speed is of essential. However, it is very difficult to parallelize the original LLL algorithm. We present a multi-threading LLL algorithm based on a recent improved version: an LLL algorithm with delayed size reduction.

1 Introduction

The LLL algorithm, introduced by Lenstra, Lenstra, and L.Lovasz [3] in 1982, is used to reduce a lattice basis. It has received a lot of attention as an effective numerical tool for preconditioning an integer least squares problem. In 2008, Franklin T. Luk and Daniel M. Tracy [5] presented a matrix version of the LLL algorithm. Here we will first present the original LLL algorithm and the LLL algorithm with delayed size-reduction, then propose a parallel LLL algorithm and implement it by using Pthread library.

1.1 Bases for Lattices

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

$$L = \{Bz\}$$

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 $B = [b_1, b_2, ..., b_n]$; $b_1, b_2, ..., b_n$ are linearly independent columns and span L, they form a basis for L.

For example, the matrix B below generates the lattice points in Figure 1.

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

1.2 Reduced Bases

A lattice may have more than one basis. For example, the matrix C below also generates the lattice in Figure 1. Figure 2 depicts the columns of B and C, and the same lattice as in Figure 1.

$$C = \begin{bmatrix} C_1 & C_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix} \tag{2}$$

Since a lattice L may have lots of bases, some of the bases are better than others. In some situations, those whose lengths are short are the ones we desire. These short bases are called reduced.

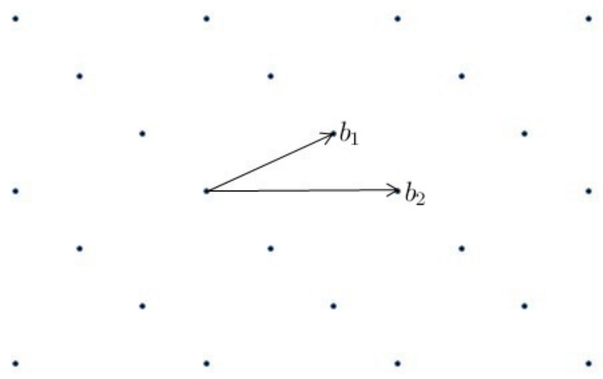

Figure 1: The lattice points generated by the column vectors b_1 and b_2 of B.

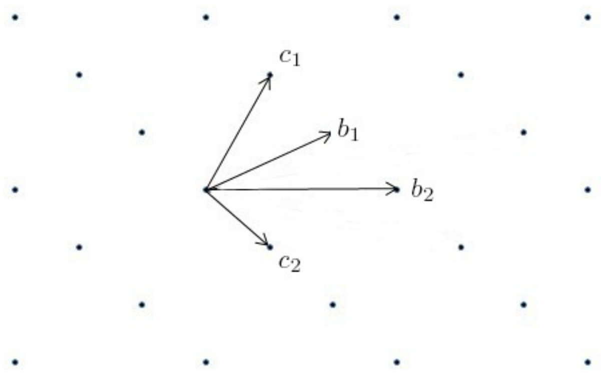

Figure 2: The lattice and the columns of B and C.

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 the components of subsequent vectors orthogonal to all of the vectors that have already been constructed [1].

Since basis vectors are linearly independent, we can find an orthogonal basis for a lattice using the Gram-Schmidt Process. Now we present this process, let $[b_1,b_2, \dots ,b_n]$ be a basis for a lattice L and define the following:

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

then $[b_1^*,b_2^*, \dots ,b_n^*]$ forms an orthogonal basis for L.

1.4 Unimodular Matrix

Two different bases for a lattice can be 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

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

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

Definition 1 (Unimodular) A nonsingular integer matrix M is called unimodular if $det(M) = \pm 1$ [2].

2 The LLL Algorithm

In the LLL algorithm, firstly, given a lattice generator m-by-$n(m \geq n)$ matrix B, it can be decomposed according to the Gram-Schmidt process as

$$\begin{aligned}
B &= \begin{bmatrix} b_1 & b_2 & \dots & b_{n-1} & b_n \end{bmatrix} \\
&= \begin{bmatrix} b_1^* & b_2^* & \dots & b_{n-1}^* & b_n^* \end{bmatrix} \begin{bmatrix} 1 & u_{1,2} & \dots & u_{1,n-1} & u_{1,n} \\ 0 & 1 & \dots & u_{2,n-1} & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & u_{n-1,n} \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \frac{b_1^*}{\|b_1^*\|_2} & \dots & \frac{b_n^*}{\|b_n^*\|_2} \end{bmatrix} \begin{bmatrix} \|b_1^*\|_2 & & \\ & \ddots & \\ & & \|b_n^*\|_2 \end{bmatrix} \begin{bmatrix} 1 & \dots & u_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{bmatrix}
\end{aligned}$$

that is,

$$B = QD^{1/2}U \tag{3}$$

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

In terms of the decomposition (3), we have the following definitions [3]:

Definition 2 (size-reduced) *A basis $B = [b_1,b_2, \dots ,b_n]$ for a lattice is called size-reduced if U in the decomposition (3) satisfies:*

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

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

$$|u_{i,j}| \leq \frac{1}{2},\ for\ 1 \leq i < j \leq n\ (sized - reduced) \tag{5}$$

and

$$d_i + u_{i-1,i}^2 d_{i-1} \geq \omega d_{i-1},\ for\ 2 \leq i \leq n \tag{6}$$

where $\frac{1}{4} < \omega < 1$.

The LLL algorithm iterates a sequence of steps to get a resulting basis satisfying the above two conditions.

If $|u_{i,j}| > \frac{1}{2}$ for some $j > i$, a procedure called *Reduce*(i, j) [4] is applied to ensure condition (5).

Reduce(i,j) *Define an elementary unimodular transformation $M_{ij} \in M^{n\times n}$ by $M_{ij} = I_n - \gamma e_i e_j^T$ where $\gamma = \lceil u_{i,j} \rfloor$ is the closest integer to $u_{i,j}$ and e_i is the ith unit vector.*

M_{ij} is an integer unimodular transformation, which is used to make sure that $u_{i,j}$ is sufficiently small.

Apply M_{ij} to U, B and M:

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

Note that $BM_{ij} = QD^{1/2}UM_{ij}$. Let $C = BM_{ij}$, since M_{ij} is an unimodular matrix, C is a new basis in which $|u_{i,j}| \leq \frac{1}{2}$.

Moreover, in the iteration of the algorithm, if the condition (6) does not hold for some $2 \leq i \leq n$, another procedure called *SwapRestore*(i) [4] is applied to enforce this condition.

SwapRestore(i) *Let $\mu = u_{i-1,i}$, compute $\hat{d}_{i-1} = d_i + \mu^2 d_{i-1}$, $d_i \leftarrow \frac{d_{i-1}d_i}{\hat{d}_{i-1}}$, $\xi = \frac{\mu d_{i-1}}{\hat{d}_{i-1}}$, $d_{i-1} = \hat{d}_{i-1}$, $u_{i-1,i} = \xi$, then define a transformation in the $(i-1, i)$ plane where $2 \leq i \leq n$:*

$X_i = [\begin{smallmatrix} \mu & 1-\mu\xi \\ 1 & -\xi \end{smallmatrix}]$

Swap the columns i and $i-1$ of U, B and M, then apply X_i^{-1} to U:

$$U \leftarrow \begin{bmatrix} I_{i-2} & & & \\ & \xi & 1-\xi\mu & \\ & 1 & -\mu & \\ & & & I_{n-i} \end{bmatrix} U$$

After swap, b^*_{i-1} is replaced by $b^*_i + u_{i-1,i}b^*_{i-1}$, so the new $\hat{D}[i-1] = \hat{d}_{i-1} = \left\|\hat{b}^*_{i-1}\right\|_2^2 < \omega\left\|b^*_{i-1}\right\|_2^2 = \omega d_{i-1} = \omega D[i-1] = \omega(\hat{d}_i + \hat{u}^2_{i-1,i}\hat{d}_{i-1}) = \omega\left\|\hat{b}^*_i + \hat{u}^2_{i-1,i}\hat{b}^*_{i-1}\right\|_2^2$, that is $\omega\hat{d}_{i-1} < \frac{1}{\omega}(\hat{d}_{i-1}) < \hat{d}_i + \hat{u}^2_{i-1,i}\hat{d}_{i-1}$[1] which satisfies the condition (6).

When we have these two procedures, we can see the LLL algorithm [4] as:

LLL Algorithm *Given a lattice generator $m-by-n$ ($m \geq n$) 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.     if $|u_{k-1,k}| > 1/2$
 5.         Reduce($k-1, k$);
 6.     endif
 7.     if $d_k < (\omega - u^2_{k-1,k})d_{k-1}$
 8.         SwapRestore($k$);
 9.         $k \leftarrow max(k-1, 2)$;
10.     else
11.         for $i = k-2$ down to 1
12.             if $|u_{i,k}| > 1/2$
13.                 Reduce($i, k$);
14.             endif
15.         endfor
16.         $k \leftarrow k+1$;
17.     endif
18. endwhile
```

3 The LLL Algorithm with Delayed Size-Reduction

Wen Zhang [6] presented a modified LLL algorithm, which can save significant amounts of unnecessary operations when compared with the original LLL algorithm.

Let's see an example first. Given $\omega = \frac{3}{4}$ and a lattice basis matrix

$$B = \begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix} \tag{7}$$

We present the process of the LLL algorithm here but just give the values of D and U of every step according to the decomposition (3) .

[1]We use "$\wedge$" above symbols to represent the new values of the relative variables after swap.

$$\begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

$$\xrightarrow{Gram-Schmidt} \qquad (S1)$$

$$DU = \begin{bmatrix} 3 & & \\ & \frac{14}{3} & \\ & & \frac{9}{14} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{3} & \frac{14}{3} \\ 0 & 1 & \frac{13}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=2 \quad do\, nothing} \qquad (S2)$$

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

$$DU = \begin{bmatrix} 3 & & \\ & \frac{14}{3} & \\ & & \frac{9}{14} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{3} & \frac{14}{3} \\ 0 & 1 & -\frac{1}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$DU = \begin{bmatrix} 3 & & \\ & \frac{14}{3} & \\ & & \frac{9}{14} \end{bmatrix} \begin{bmatrix} 1 & \frac{13}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$DU = \begin{bmatrix} 3 & & \\ & \frac{2}{3} & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$DU = \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$DU = \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\xrightarrow{k=3 \quad do\, nothing} \qquad (S8)$$

End

Note that after the step $(S4)$, the algorithm found that $|u_{1,2}| = \frac{13}{3} > \frac{1}{2}$, so we had the step $(S5)$ to reduce $u_{1,2}$. Then the step $(S6)$ was applied because $d_2 < (\frac{3}{4} - u_{1,2}^2)d_1$. After the step $(S6)$, we found that $|u_{1,2}| = 1$, which is bigger than $\frac{1}{2}$, so $Reduce(1,2)$ was applied again. However, if we did not reduce $u_{1,2}$ in the step $(S5)$, instead swapped d_2 and d_1 first, we would just need the step $Reduce(1,2)$ once. This means that size reduction can be delayed until the condition (6) is satisfied first. This method is called the LLL algorithm with delayed size-reduction [6].

Before presenting the new algorithm, we introduce the procedure $ReduceSwapRestore(i, \gamma)$, in which $\gamma = \lceil u_{k-1,k} \rfloor$ is an integer that is closest to $u_{k-1,k}$.

ReduceSwapRestore(i,γ) *Let $\mu = u_{i-1,i}$, compute $\hat{d}_{i-1} = d_i + (\mu - \gamma)^2 d_{i-1}$, $d_i \leftarrow \frac{d_{i-1}d_i}{\hat{d}_{i-1}}$, $\xi = \frac{(\mu-\gamma)d_{i-1}}{\hat{d}_{i-1}}$, $d_{i-1} = \hat{d}_{i-1}$, $u_{i-1,i} = \xi$, then define a transformation in the $(i-1, i)$-plane, where $2 \leq i \leq n$:*

$$X_i' = \begin{bmatrix} \mu - \gamma & 1 - \mu\xi + \gamma\xi \\ 1 & -\xi \end{bmatrix} = \begin{bmatrix} 1 & -\gamma \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mu & 1 - \mu\xi \\ 1 & -\xi \end{bmatrix}$$

Let $P = \begin{bmatrix} 1 & -\gamma \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, combination of reduction and permutation $\Pi_i = diag([\ I_{i-2} \quad P \quad I_{n-i}\ ])$, apply Π_i to U, B and M:

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

Then apply $X_i^{'-1}$ to U:

$$U \leftarrow \begin{bmatrix} I_{i-2} & & & \\ & \xi & 1 - \xi\mu + \gamma\xi & \\ & 1 & \gamma - \mu & \\ & & & I_{n-i} \end{bmatrix} U$$

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

LLL Algorithm with delayed size-reduction *Given a generator m-by-n $(m \geq n)$ 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
```

Now let's see the process of the LLL algorithm with delayed size-reduction applied to the same matrix B in (7):

$$\begin{bmatrix} 1 & -1 & 3 \\ 1 & 0 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

$\xrightarrow{\underline{Gram-Schmidt}}$ $(P1)$

$----Begin-While-Loop----$

$$DU = \begin{bmatrix} 3 & & \\ & \frac{14}{3} & \\ & & \frac{9}{14} \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{3} & \frac{14}{3} \\ 0 & 1 & \frac{13}{14} \\ 0 & 0 & 1 \end{bmatrix}$$

$\xrightarrow{\underline{k=2 \quad do\,nothing}}$ $(P2)$

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

$$DU = \begin{bmatrix} 3 & & \\ & \frac{2}{3} & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & \frac{13}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$DU = \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$\xrightarrow{\underline{k=2 \quad do\,nothing}}$ $(P5)$

$\xrightarrow{\underline{k=3 \quad do\,nothing}}$ $(P6)$

$----End-While-Loop----$

$----Begin-For-Loop----$

$Reduce(1,2)$ $(P7)$

$$DU = \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$----End-For-Loop----$

End

As we expected, in this process, the procedure $Reduce(1,2)$ was applied only once.

4 The LLL Algorithm with Delayed Size-Reduction and Odd-Even Ordering

In order to parallelize the LLL algorithm with delayed size-reduction, let's reorder the sequence of the executions of the two procedures: $ReduceSwapRestore(i,\gamma)$ in the $while$ loop and $Reduce(i,k)$ in the for loop.

In the *while* loop, we first execute *ReduceSwapRestore*(i, γ) for all d_k such that k is even and $d_k < (\omega - (u_{k-1,k} - \gamma)^2)d_{k-1}$, then execute *ReduceSwapRestore*(i, γ) for d_k such that k is odd and $d_k < (\omega - (u_{k-1,k} - \gamma)^2)d_{k-1}$. If there is no any d_k such that $d_k < (\omega - (u_{k-1,k} - \gamma)^2)d_{k-1}$, this subprocess will end, otherwise, we execute this subprocess again.

When the *while* loop finishes we enter into the for loop. We will reduce $u_{i,j}$ if $|u_{i,j}| > 1/2$ as in the LLL algorithm with delayed size-reduction but in a different order: first check the superdiagonal from $u_{1,2}$ to $u_{n-1,n}$, then check the subsequent diagonal from $u_{1,3}$ to $u_{n-2,n}$. Repeat this process until the final diagonal which has only one element $u_{1,n}$. For every diagonal, first check the even elements and reduce $u_{i,j}$ where $|u_{i,j}| > 1/2$, then check the odd elements and reduce $u_{i,j}$ where $|u_{i,j}| > 1/2$. For example, given a matrix

$$U = \begin{matrix} 1 & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} \\ 0 & 1 & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} \\ 0 & 0 & 1 & u_{3,4} & u_{3,5} & u_{3,6} & u_{3,7} & u_{3,8} \\ 0 & 0 & 0 & 1 & u_{4,5} & u_{4,6} & u_{4,7} & u_{4,8} \\ 0 & 0 & 0 & 0 & 1 & u_{5,6} & u_{5,7} & u_{5,8} \\ 0 & 0 & 0 & 0 & 0 & 1 & u_{6,7} & u_{6,8} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & u_{7,8} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{matrix},$$

we begin at the diagonal from $u_{1,2}$ to $u_{7,8}$, firstly, we check $u_{2,3}$, $u_{4,5}$ and $u_{6,7}$. If any of them is bigger than 1/2, the procedure *Reduce*(i, j) will be applied. Then, we will check $u_{1,2}$, $u_{3,4}$, $u_{5,6}$ and $u_{7,8}$. Also, if there is any element bigger than 1/2, the procedure *Reduce*(i, j) will be applied. After finishing checking the diagonal from $u_{1,2}$ to $u_{7,8}$, we will go to the next diagonal from $u_{1,3}$ to $u_{6,8}$ and apply the same rule again.

Now we can present the new algorithm, called the LLL algorithm with delayed size-reduction and odd-even ordering.

LLL algorithm with delayed size-reduction and odd-even ordering *Given a generator m-by-n$(m \geq n)$ matrix B, compute 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 $k \leftarrow 2 : +2 : n$
6.          $\gamma = \lceil u_{k-1,k} \rfloor$;
7.          if $d_k < (\omega - (u_{k-1,k} - \gamma)^2)d_{k-1}$
8.              $f \leftarrow false$
9.              ReduceSwapRestore(k,$\gamma$);
10.         endif
11.     endfor
12.     for $k \leftarrow 3 : +2 : n$
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 $k \leftarrow 2 : n$
21.     for $i \leftarrow 1 : +2 : n$ and $j \leftarrow k : +2 : n$
22.         if $|u_{i,j}| > 1/2$
23.             Reduce($i, j$);
24.         endif
25.     endfor
26.     for $i \leftarrow 2 : +2 : n$ and $j \leftarrow k + 1 : +2 : n$
27.         if $|u_{i,j}| > 1/2$
28.             Reduce($i, j$);
29.         endif
30.     endfor
31. endfor
```

Where to parallelize? For the parallel implementation of the above algorithm, let's first divide this algorithm into two parts: *Swap* part (from line 3 to line 19) and *Reduce* part (from line 20 to the end), disregarding the first line and the second line because it is not necessary to parallelize these two simple initialization statements. For simplicity, we will use "*Swap* part" and "*Reduce* part" to denote the corresponding part in later sections.

The *Swap* part consists of a *while* routine with two *for* routines within it. As designed, the two *for* routines should be executed in sequential order. However, for each *for* routine, its iterations can run in parallel because they update entries that are independent of each other and therefore do not conflict. For example, assuming we have a basis B that is big enough. In the first *for* routine, when $k = 2$, it checks whether d_2 is smaller than $(\omega - (u_{1,2} - \gamma)^2)d_1$ where $\gamma = \lceil u_{1,2} \rfloor$, and swaps d_1 and d_2 if so. But when $k = 4$, the routine checks whether d_4 is smaller than $(\omega - (u_{3,4} - \gamma)^2)d_3$ where $\gamma = \lceil u_{3,4} \rfloor$ and swaps d_3 and d_4 if so. Therefore, the variables used when $k = 2$ do not conflict with the variables used when $k = 4$. Since the values of k are all even in the first *for* routine, it can be concluded that these iterations can run in parallel.

The same conclusion can be applied to the second *for* routine where all the values of k are odd. Now we write the parallel pseudocode of the *Swap* part as

```
while true

    parallelize the first for routine
    parallelize the second for routine
```

The *Reduce* part consists of an outer for routine with two inner for routines. As in the *Swap* part, the two inner for routines should also be executed in sequential order. However, this part cannot be parallelized, let's see what would happen.

When the value of the outer for routine's k equals 2 , the first inner for routine will check all values of $u_{i,i+1}$ where i is odd and $i \leq n-1$. For example, when $i=1$, the routine checks if $|u_{1,2}|$ is bigger than 1/2. If so, *Reduce(*$1,2$*)* will be called. As showed in section 2, in order to reduce $u_{1,2}$, the procedure *Reduce(i, j)* uses the values of the first column in matrix U to update the values of the second column. Then when $i=3$, if *Reduce(*$3,4$*)* is called, to reduce $u_{3,4}$, the procedure *Reduce(i, j)* will use the values of the $3th$ column in U to update the values of the $4th$ column. So the variables used when $i=1$ do not disturb the variables used when $i=3$, or when i equals any other odd number. This allows checking or reducing the elements in U in parallel when $k=2$. The same analysis can be applied to the second inner for routine.

However, when k equals 3, some problems emerge. Take the first inner for routine as an example, when $i=1$, the routine checks $|u_{1,3}|$ and calls *Reduce(*$1,3$*)* if $|u_{1,3}|$ is bigger than 1/2. To reduce $u_{1,3}$, the procedure *Reduce(i, j)* uses the values of the first column of U to update the values of the $3th$ column. But when $i=3$, if *Reduce(*$3,5$*)* is called, the procedure *Reduce(i, j)* will use the values of the $3th$ column in U to update the values of the $5th$ column. If this for routine is being parallelized, *Reduce(*$3,5$*)* and *Reduce(*$1,3$*)* may be executed in parallel, which will generate wrong results because *Reduce(*$1,3$*)* is updating the $3th$ column while *Reduce(*$3,5$*)* is using the values of the $3th$ column whose values may have been updated or not.

To solve this problem, we need to change our algorithm when k equals 3. We can use three inner for routines, the i indices of which begin from 1, 2, 3 to $n-2$, $n-1$, n respectively, and change the increment of the indices i to be 3. This will make it possible for our algorithm to be parallelized when k equals 3, eliminating the problem described above. But this does not guarantee to solve the similar problem when k equals other value. For example, assuming the matrix U is big enough, when k equals 6, *Reduce(*$6,11$*)* and *Reduce(*$1,6$*)* will generate the same problem, so we need to change our algorithm again, which makes the process of parallelizing too complex.

Therefore, in order to parallelize the LLL algorithm, we should offer a new structure.

5 A Parallel LLL Algorithm

Since the *Swap* part can be parallelized very well, we only need to consider the *Reduce* part.

From section 4, we can conclude that, for any two elements $u_{i,j}$ and $u_{p,q}$ in U, the reason why the executions of reducing them may conflict with each other is that p may equal j or q may equals i, which will cause a problem that the column used to update the new column may be the column being updated.

We notice that, in figure 3, for any two different elements $u_{i,j}$ and $u_{p,q}$ in U on the same anti-diagonal line, firstly, $i+j$ and $p+q$ always equal a constant m, so $i-p=q-j$; second, they are not on the same column or row, so $i \neq p$ and $j \neq q$; third, they are above the diagonal line, so $i < j$ and $p < q$. Therefore, $i \neq j \neq p \neq q$, and the problem we mentioned before will never happen if we simultaneously reduce the elements on the same anti-diagonal line. That is, for every anti-diagonal line in U, reducing one element does not conflict with reducing any other element on the same line. We call that these elements are independent for reducing. For example, if $u_{1,4}$ and $u_{2,3}$ are needed to be reduced, it is no matter which one is reduced first because they are on the same anti-diagonal line. Now we can parallelize the elements on the same anti-diagonal line. The sequential version of this algorithm is as follows.

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

```
1. set $M \leftarrow I$

2. $k \leftarrow 2$;
```

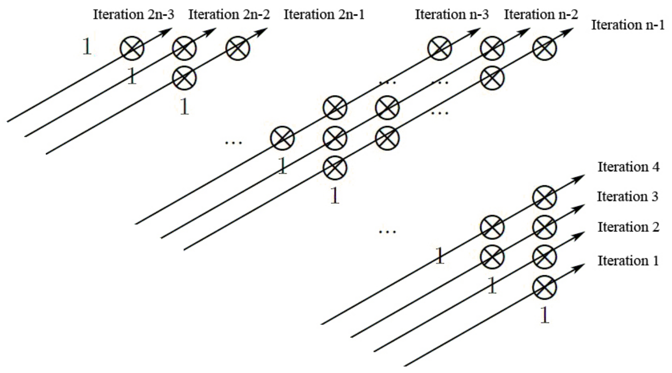

Figure 3: Reordering the procedure *Reduce*(i, j). For legibility, we use $\otimes$ to represent $u_{i,j}$ in the relative position.

```
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 $k \leftarrow 2n-3 : 1$
13.     if $k \leq n-1$
14.         $h = 1$;
15.     else
16.         $h = k - n + 2$;
17.     endif
18.     for $i \leftarrow h : (k+3)/2$
19.         $j = k + 2 - i$;
20.         if $|u_{i,j}| > 1/2$
21.             Reduce($i, j$);
22.         endif
23.     endfor
24. endfor
```

As in figure 3, there are $2n-3$ anti-diagonal lines, so we need $2n-3$ iterations. We will execute the algorithm from iteration 1 to iteration $2n-3$. In a serial implementation, we will begin with the rightmost element on the current anti-diagonal line for every iteration. Let k be the loop control variable, when $k \leq n-1$, all the rightmost elements $u_{i,j}$ in every iteration are on the last column, so $i = k-n+2$. But when $n-1 < k \leq 2n-3$, all the rightmost elements $u_{i,j}$ are on the first row, so $i = 1$. For any $u_{i,j}$ in iteration k, if we know i, we can compute j because $i+j$ always equals $k+2$.

Note that this algorithm must be executed exactly as the order from iteration 1 to iteration $2n-3$ as we showed in figure 3. It may generate a wrong result if it is executed in the reverse order or others.

However, this method generates a different reduction basis compared to the LLL Algorithm with Delayed Size-Reduction because the total number of executing the procedure *Reduce(i,j)* in this method may be more than or less than the latter, which depends on specific matrix.

How to parallelize? Now we can show the pseudocode of parallelizing the *reduce* part (line 12 to line 24) in this algorithm as

```
for k from 2n-3 to 1

    compute the range of the elements in iteration k
    simultaneously check these elements on the current anti-diagonal
    line and reduce them if their abstract values are bigger than 1/2
```

6 Implementation of the Parallel LLL Algorithm with Phreads

6.1 Threads Versus Processes

Both threads and processes can provide parallel program execution, but there are some differences. As in figure 4, processes contain information about program resources and program execution state, including [9]:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory
- Program instructions
- Registers
- Stack
- Heap
- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

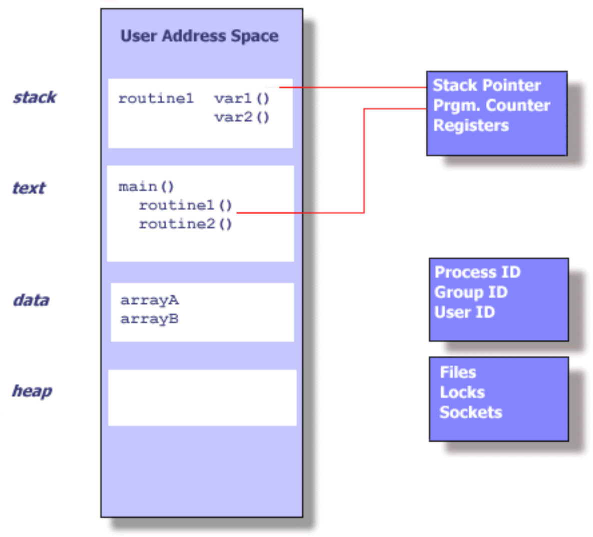

Figure 4: UNIX Process

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code. From the figure 5 , we can see this independent flow of control is accomplished because a thread maintains its own [9]:

- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.

It can be expensive to create a new process. First, the entire process should be copied. Second, the context-switching mechanism will start if the process creation triggers process rescheduling activity. Also, the operating system kernel may be called introducing the cost of interprocess communication and synchronization of shared data.

However, creating a new thread takes less time and memory because there is no need to replicate an entire process, and part of the work of creation can be done in user space rather than operating system kernel. In addition, synchronization can stay within the user space by using a variable.

Therefore, when compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

6.2 Introduction of Pthreads

Pthreads is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. The "P" in Pthreads comes from POSIX (Portable Operating System Interface), the family of IEEE operating system interface standards in which Pthreads is defined (POSIX Section 1003.1c to be exact) [8]. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library.

The primary motivation for considering the use of Pthreads on an SMP architecture is to achieve optimum performance. In particular, if an application is using MPI for on-node communications,

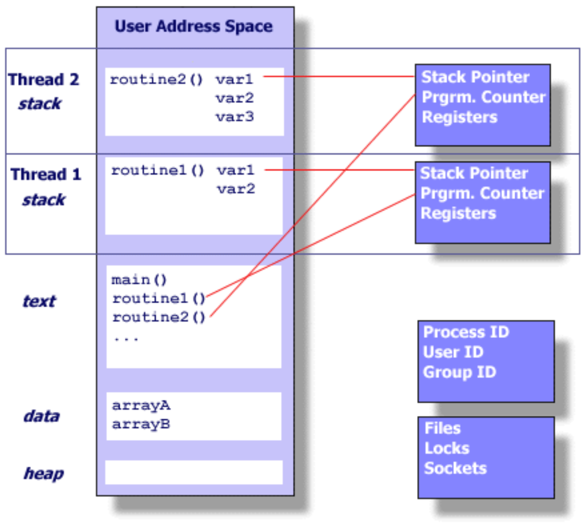

Figure 5: THREADS WITHIN A UNIX PROCESS

there is a potential that performance could be greatly improved by using Pthreads for on-node data transfer instead. For example, MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process). For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It becomes more of a cache-to-CPU or memory-to-CPU bandwidth (worst case) situation. These speeds are much higher [9].

The Pthreads library aims to be expressive as well as portable, and it 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 mutexes, locks, condition variables, and semaphores.

Pthreads realizes the shared-memory programming model via a collection of routines for creating, managing and coordinating a collection of threads. In such shared-memory model (Figure 6), all threads have access to the same global resources, which should be synchronized by programmers. Each thread also has its own local resources.

In order to take advantage of Pthreads, a program should be able to be organized into discrete, independent tasks which can execute concurrently. There are some characteristics which can be well suited for Pthreads:

- *Subtasks*: A task can be executed by multiple subtasks simultaneously. For example, Matrix multiplication takes two two-dimensional input arrays of data and computes a third, which has the characteristic of repeating multiplication operations over and over again on subsets of these arrays. We can improve the performance by simultaneously executing these operations.

- *Overlapping I/O*: If some tasks represent a long I/O operation that may block for waiting for an I/O system call to complete, allowing CPU-intensive tasks to continue independently may have performance advantages.

- *Asychronous events*: If some tasks subject to asynchronous events are in some unknown state of completion, it may be more efficient to allow other tasks to proceed.

- *Real-time Scheduling*: One task may have high priority than another, but if they should be executed whenever possible, we may need some scheduling priorities and policies to run them independently.

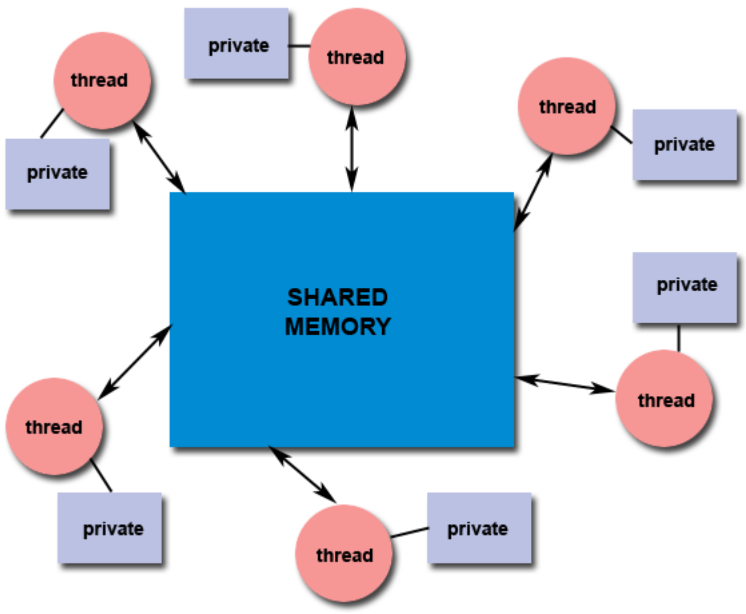

Figure 6: A shared-memory model

There are no set rules for threading a program, but there are some models that define how a threaded application delegates its work to its threads and how these threads communicate with each other [9]:

- *Boss/Worker Model*: The boss thread creates each worker thread, assigns it tasks, and, if necessary, waits for it to finish. Figure 7 depicts the boss/worker model.
- *Peer Model*: In the peer model, depicted in Figure 8, all threads work concurrently on their tasks without a specific leader. One thread create all the other peer threads when the program starts. However, unlike the boss thread in the boss/worker model, this thread subsequently acts as just another peer thread that processes requests, or suspends itself waiting for the other peers to finish.
- *Pipeline Model*: The pipeline model assumes: a long stream of input, a series of suboperations through which every unit of input must be processed and each processing stage can handle a different unit of input at a time. As illustrated in Figure 9, a single thread receives input for the entire program, always passing it to the thread that handles the first stage of processing. Similarly a single thread at the end of the pipeline produces all final output for the program. Each thread in between performs its own stage of processing on the input it received from the thread that performed the previous stage, and passes its output to the thread performing the next.

6.3 Designing the Parallel Implementation

Now let's add threads to our algorithm. As we described in section 5, in the parallel algorithm, there are two parts: *Swap* part and *Reduce* part, that need to be parallelized. For each part, there are independent tasks within them that can run in parallel. Let's first see the *swap* part.

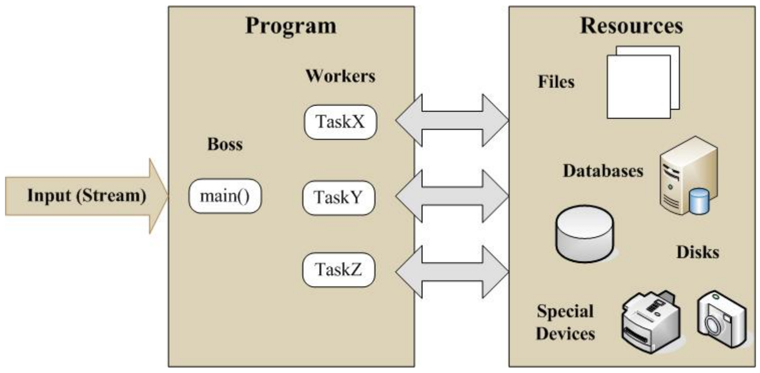

Figure 7: The Boss/Worker Model

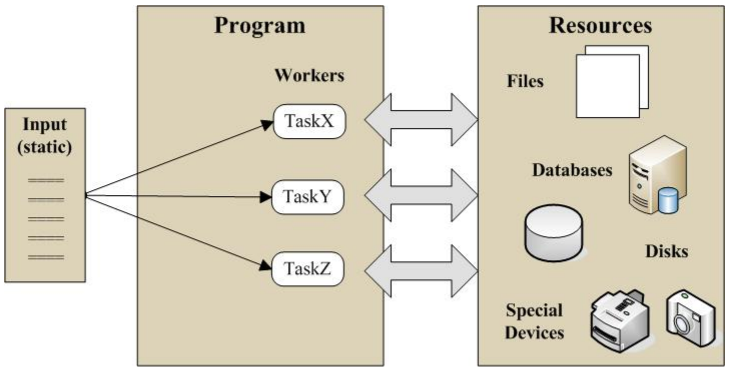

Figure 8: The Peer Model

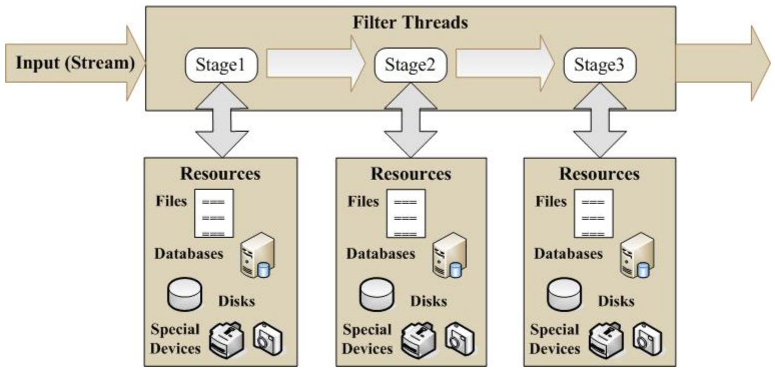

Figure 9: The Pipeline Model

6.3.1 Implementation of the *Swap* Part

Section 5 gives the serial version of the parallel algorithm, where the *Swap* part can be programmed as

```
int r, k;
bool finish = false;
while(!finish){
    finish = true;
    for(k=2;k<=n;k=k+2){
        r = closedInt(U[k-1][k]);
        if( D[k] < (w - (U[k-1][k] - r)*(U[k-1][k] - r)) * D[k-1] ){
            finish = false;
            ReduceSwapRestore(k,B,U,M,D,m,n,r);
        }
    }
    for(k=3;k<=n;k=k+2){
        r = closedInt(U[k-1][k]);
        if( D[k] < (w - (U[k-1][k] - r)*(U[k-1][k] - r)) * D[k-1] ){
            finish = false;
            ReduceSwapRestore(k,B,U,M,D,m,n,r);
        }
    }
}
```

U and M are both two-dimensional double arrays which represent $n \times n$ matrices U and M in the algorithm, respectively. B is a two-dimensional array representing the $m \times n$ matrix B. D is an array used to store the values of D where $D = diag(d_i)$. m is the number of the rows of the computed basis, whereas n is the number of the columns. The function closedInt returns an integer nearest the argument, which is a double value. Procedure *ReduceSwapRestore(k,γ)* is realized by the function ReduceSwapRestore.

A parallel implementation by using peer model As we mentioned, for every iteration in the *while* routine, we can parallelize the first *for* routine, then the second. We'll use peer model for each *for* routine, where we create a peer thread for each individual iteration. A main thread will also exist - not so much as a peer thread but as a setup and cleanup thread. It performs all of the setup tasks for the program, creates the peer threads, and waits for them to complete. In the serial version of our algorithm, the *for* routine made a procedure call to check the condition and invoke ReduceSwapRestore if the condition is satisfied. For using the create function pthread_create in Pthreads library, here we need another function subSwap that wraps the codes within the *for* routine in order for each worker thread to execute. However, there is one complication - the ReduceSwapRestore routine, as used in the serial version, has many arguments, but the pthread_create function lets threads start only in routines that are passed a single argument, so we bundle everything the *for* routine wants to pass to its peer threads into a single structure. We call this structure the work_swap_t, and it contains fields for all of the arguments passed to the subSwap. Our *for* routine passes each peer thread a pointer to a work_swap_t structure as the last argument in the pthread_create call.

```
...
struct work_swap_t {
    int k;
    int m;
    int n;
    basis *B;
    double **U;
    double **M;
    double *D;
    double w;
} ;


pthread_mutex_t swap_count_mutex = PTHREAD_MUTEX_INITALIZER;
static bool finish = false;


void *subSwap(void* w_swap) {
    work_swap_t* work = (work_swap_t*)w_swap;
    int r = closedInt((work->U)[work->k-1][work->k]);
    if ((work->D)[work->k] < (work->w - ((work->U)[work->k-1][work->k] - r)
    *((work->U)[work->k-1][work->k] - r)) * (work->D)[work->k-1]){

        ReduceSwapRestore(work->k,work->B,work->U,work->M,work->D,
        work->m,work->n,r);
        pthread_mutex_lock(&swap_count_mutex);
        finish = false;
        pthread_mutex_unlock(&swap_count_mutex);
    }
}
...
void parallelLLL(const int m, const int n, basis *B, double w) {
    ...
    while(!finish){
        finish = true;
        for(k = 2;k <= n;k = k + 2){
            w_swap = new work_swap_t;
            w_swap->k = k;
            w_swap->m = m;
            w_swap->n = n;
            w_swap->B = B;
            w_swap->U = U;
            w_swap->M = M;
            w_swap->D = D;
            w_swap->w = w;
            pthread_create(&(swap_thread[k-2]),NULL,subSwap,(void*)w_swap);
        }
        for(k = 2;k <= n;k = k + 2){
            pthread_join(swap_thread[k-2],NULL);
        }
        for(k = 3;k <= n;k = k + 2){
            w_swap = new work_swap_t;
            w_swap->k = k;
            w_swap->m = m;
            w_swap->n = n;
            w_swap->B = B;
            w_swap->U = U;
            w_swap->M = M;
            w_swap->D = D;
            w_swap->w = w;
            pthread_create(&(swap_thread[k-2]),NULL,subSwap,(void*)w_swap);
        }
        for(k = 3;k <= n;k = k + 2){
            pthread_join(swap_thread[k-2],NULL);
        }
    }
    ...
}
...
```

`parallelLLL` is the main routine executing the parallel LLL algorithm, in which we omit other codes except the necessary structures and variables for the *Swap* part. Using the `work_swap_t` structure lets the `parallelLLL` routine bundle various pieces of information into a single `pthread_create` argument, but the thread's start routine `subSwap` must accept only a single argument whose type is `void*`, so we should cast `work_swap_t*` to `void*` in the `pthread_create` statement, then cast it back to `work_swap_t*` at the beginning of the `subSwap` routine. Since threads may write the boolean variable `finish` at the same time, we use a lock `swap_count_mutex` to avoid race conditions. For every *while* loop, we use `pthread_join` to join threads that have been created after every parallel *for* routine.

Testing the peer-model implementation In order to test our implementation, we will compare efficiencies between the serial version and the parallel version of the parallel algorithm, by using two tested programs between which the only different part is that the *Swap* part in the parallel version is parallelized by using Pthread library. We'll use several test groups and just record cost spent on this part. In every test group, we adopt the same randomized matrix, which is generated by a specific method in the program, for the two programs,

The test platform is

- Module: Dell Inspiron 580s (i-3 4-core cpu)
- Operating System: Ubuntu 10.04
- Development Environment: g++

Let $\omega = 0.75$. Table 1 shows the results. There are seven tested groups. The dimensions of the randomized matrices are listed in the first row. "*Swap* Times" represents the number of times that `ReduceSwapRestore` has been called. S denotes the serial version and PM denotes the parallel version with peer model. "Time cost" represents the cost spent on the *Swap* part in each test group. The unit of time is microsecond.

From table 1 we can see that the costs spent on the parallel implementation are 10-20 times more than those spent on the serial version, except the (100,100) group, but even in this group the former is still more than the latter. It is clearly that this is not a viable parallel implementation. The reason is that, our machine can run at most four threads at the same time and we use too

Dimensions	10,10	20,20	50,50	100,100	200,200	300,300	400,400
Swap Times(S)	29	17	93	74	118	110	121
Swap Times(PM)	29	17	93	74	118	110	121
Time Cost(S)	115	72	995	23185	1869	2711	4163
Time Cost(PM)	1498	3408	12310	24177	28406	66299	78751

Table 1: Comparison of the *swap* time cost between the serial implementation and the parallel implementation by using peer model.

many threads to execute each iteration in the for routine so that the most cost in the parallel implementation is spent on creating and joining these threads. For example, in (100,100) group, we need create 98 threads in each *while* iteration.

Therefore, considering the limitation of the platform and the overhead of creating and joining procedures, we should redesign our implementation.

A better parallel implementation by using a thread pool The platform has a 4-core CPU, since one core may be used for the operating system processes, the efficient number of work threads at the same time may be 3 or 4. We'll consider using a thread pool, where the boss thread creates a fixed number of worker threads up front. Like their boss, these worker threads survive for the duration of the program. The for routine will be separated to several parts, and each worker thread takes one part. These threads are created only once at the beginning and reused then, whereas in the previous implementation we keep joining and creating in order for synchronization among threads. We'll adopt condition variable combined with mutex to synchronize threads. Below is our new implementation.

```
...
struct work_swap_t {

    int offset;
    int m;
    int n;
    basis *B;
    double **U;
    double **M;
    double *D;
    double w;

} ;


const int NUM_THREADS = 4;
static int joinCount = 1;
static bool finish = false;
static bool subfinish[NUM_THREADS];
pthread_mutex_t swap_count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t swap_loop_end = PTHREAD_COND_INITIALIZER;


void *subSwap(void* w_swap) {

    work_swap_t* work = (work_swap_t*)w_swap;
    int r, k;


    while(!finish) {
```

```
    subfinish[work->offset] = true;
    for(k = 2 + 2 * (work->offset);k <= work->n;k = k + 2 * (NUM_THREADS)){
        r = closedInt((work->U)[k-1][k]);
        if((work->D)[k] < (work->w - ((work->U)[k-1][k] - r)*
        ((work->U)[k-1][k] - r))*(work->D)[k-1] ){
          subfinish[work->offset] = false;
          ReduceSwapRestore(k,work->B,work->U,work->M,work->D,
          work->m,work->n,r);
        }
    }

    pthread_mutex_lock(&swap_count_mutex);
    if(joinCount < NUM_THREADS){
        joinCount++;
        pthread_cond_wait(&swap_loop_end, &swap_count_mutex);
    } else {
        joinCount = 1;
        pthread_cond_broadcast(&swap_loop_end);
    }
    pthread_mutex_unlock(&swap_count_mutex);

    for(k = 3 + 2 * (work->offset);k <= work->n;k = k + 2 * (NUM_THREADS)){
        r = closedInt((work->U)[k-1][k]);
        if( (work->D)[k]<(work->w - ((work->U)[k-1][k] - r)
        *((work->U)[k-1][k] - r))*(work->D)[k-1]){
          subfinish[work->offset] = false;
          ReduceSwapRestore(k,work->B,work->U,work->M,work->D,
          work->m,work->n,r);
        }
    }

    pthread_mutex_lock(&swap_count_mutex);
    if(joinCount < NUM_THREADS){
        joinCount++;
        pthread_cond_wait(&swap_loop_end, &swap_count_mutex);
    } else {
        finish = true;
        for(int i = 0;i < NUM_THREADS; i++){
          if(subfinish[i] == false) {
            finish = false;
            break;
          }
        }
        joinCount = 1;
        pthread_cond_broadcast(&swap_loop_end);
    }
    pthread_mutex_unlock(&swap_count_mutex);
}
```

```
}
...
void parallelLLL(const int m, const int n, basis *A, double w){

    ...
    pthread_t swap_thread[NUM_THREADS];
    work_swap_t *w_swap[NUM_THREADS];
    for(int i = 0;i < NUM_THREADS;i++) {
        w_swap[i] = new work_swap_t
        w_swap[i]->offset = i;
        w_swap[i]->m = m;
        w_swap[i]->n = n;
        w_swap[i]->B = B;
        w_swap[i]->U = U;
        w_swap[i]->M = M;
        w_swap[i]->D = D;
        w_swap[i]->w = w;
        pthread_create(&swap_thread[i], NULL, subSwap, (void*)w_swap[i]);
    }
    for(int i = 0;i < NUM_THREADS;i++){
        pthread_join(swap_thread[i],NULL);
    }
    ...
}
...
```

NUM_THREADS indicates the number of worker threads that we will use to run the *Swap* part in parallel. The variable k in struct work_swap_t has been deleted but a new variable offset has been added, which is used to compute the start position of variable k of the two *for* routines. Each thread handles a different group of the values of k. For example, assume the value of n is 20, so the values of k in the first *for* routine will be: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. And the second: 3, 5, 7, 9, 11, 13, 15, 17, 19. If NUM_THREADS equals 4, then we divide these values into 4 parts for each *for* routine. The first *for* routine:

- 2, 10, 18
- 4, 12, 20
- 6, 14
- 8, 16

and the second:

- 3, 11, 19
- 5, 13
- 7, 15
- 9, 17

Now each thread handles one part. In order to synchronize running threads after they all finish executing the first *for* routine, we use the condition variable swap_loop_end. We make the global variable joinCount a shared resource that these threads increment and create the mutex swap_count_mutex (in global scope) to protect it. We use this condition variable swap_loop_end to represent an event - the joinCount variable's reaching a defined threshold value, NUM_THREADS, which indicates that all threads have finished their work, so it's time to handle the second *for* routine. If joinCount is smaller than NUM_THREADS, it means that there may be one or more threads that still have not finished executing the first *for* routine, so the current thread will wait other threads to finish their work by using pthread_cond_wait. If joinCount has reached its threshold value NUM_THREADS, subSwap will call pthread_cond_broadcast to notify the threads

Dimensions	10,10	20,20	50,50	100,100	200,200	300,300	400,400	500,500
$Swap$Times(S)	22	69	64	149	69	186	238	96
$Swap$Times(TP)	22	69	64	149	69	186	238	96
Time Cost(S)	54	352	891	1216	1119	4491	7899	4222
Time Cost(TP, 2 threads)	218	582	407	1111	1006	3344	5552	7059
Time Cost(TP, 3 threads)	269	837	362	975	865	3426	5175	3021
Time Cost(TP, 4 threads)	254	1003	467	1064	1018	3576	4982	2642

Table 2: Comparison of the *Swap* costs between the serial implementation and the parallel implementation by using a thread pool.

that's waiting for this particular event. Since we will recount `joinCount`, it should be set to 1 again. However, the synchronization part after the second for routine has some differences.

We use the variable `finish` to determine whether the *while* loop should be ended, and if `ReduceSwapRestore` is called, the variable is set to `false`, which tells us that we should execute the *while* loop again. Since there are several threads which may write value to this variable simultaneously and cause race condition, we use a boolean array `subfinish` to store such value for each thread. When a thread finishes executing the second for routine, and if it is the last thread that checks the `joinCount` variable (if `joinCount = NUM_THREADS`), it will first check the values in the array `subfinish` to determine the value of the global variable `finish`. If all values in the array `subfinish` are `true`, which means that no `ReduceSwapRestore` has been called in the current iteration of the *while* loop, `finish` will be set to be `true` and all threads will stop working. Otherwise, all threads will enter the next iteration of the *while* loop. Here we can also use the lock `swap_count_mutex` to protect the global variable `finish` when it is being changed without using `subfinish`, but it will increment times of synchronization and may depress efficiency.

The `parallelLLL` routine creates threads at the beginning and retracts then at the end, so the threads are created and destroyed only once, which reduces overheads used in Pthreads library.

Testing the thread-pool implementation As before, we will test this program on the same platform and use the same rules and ω, but different matrices. TP denotes the thread-pool implementation. Table 2 shows the results.

From table 2 we can see increasing efficiency when dimension is $(50, 50)$ or larger. However, it is not like what we expect, such as, obtaining a speed which is 3 times faster than the serial implementation when using 3 threads. There are two primary reasons.

- *Overhead*: The algorithm has too many synchronization points. The *while* loop may iterate many times, and for each iteration, there are two synchronization points. Every synchronization part has some logic codes to execute. When one thread obtain the lock `swap_count_mutex`, other threads should wait, so the cost of all blocks amount to a huge value, which the serial version does not have. In addition, these threads should intercommunicate when the program enters the synchronization part in order for each thread to know states of the others, this also add much overhead.
- *Structure of Decomposition*: We separate the values of `k`, as we mentioned before, but there is a problem. For example, assuming $n = 20$ and we have four threads A, B, C, D, so the values of k assigned to these threads for the first for routine may be (A: 2, 10, 18), (B: 4, 12, 20), (C: 6, 14), (D: 8, 16). In an iteration of the *while* loop, if only in positions 2, 10 and 18 `ReduceSwapRestore` is called, threads B, C and D will just have a little work to do, so the program will not benefit from these parallel structure, and cost of thread A determines the total cost. Although in the serial version the routine also call `ReduceSwapRestore` at positions 2, 10 and 18, it may take too much time on synchronization in the parallel version, which will cause that the parallel implementation spends more time than the serial one. We can eliminate this asymmetry decomposition problem if we have enough cores, which means, if we have enough threads that can run in parallel, each thread will just have at most one

value of k to handle. However, more threads add more overhead of intercommunication when synchronizing.

Analysis of the thread-pool implementation We can analyze the thread-pool implementation from an average point of view. Let's first define some variables:

- $N_{iteration}$: the total number of for iterations in the *Swap* part. For example, if $n = 20$, then in each iteration of the *while* loop, the first for routine iterates from 2 to 20, and the second iterates from 3 to 19, so the number of for iterations in current *while* iteration is 19. If there are N_0 *while* iterations, then $N_{iteration} = 19N_0$.
- N_{swap}: the total number of the invoked times of the procedure `ReduceSwapRestore`.
- N_{syn}: the total number of the synchronization points in the implementation. Note that in the above program we have a synchronization point after each for routine.
- t_{swap}: the average cost of executing the procedure `ReduceSwapRestore`, plus the statement of assigning value to the variable `subfinish`.

```
subfinish[work->offset] = false;
ReduceSwapRestore(k,work->B,work->U,work->M,work->D,work->m,work->n,r);
```

- t_{check}: the average time cost of executing each for iteration which contains the average time of computing `r`, plus the average time used to check the condition of `if` statement, but without the statements within it.

```
r = closedInt((work->U)[k-1][k]);
if( (work->D)[k] < (work->w - ((work->U)[k-1][k] - r)
*((work->U)[k-1][k] - r)) * (work->D)[k-1] ){
    //Omit these statements
}
```

- t_{syn}: the average cost for a synchronization point. Note that the threads cannot get the lock `swap_count_mutex` at the same time, so we compute t_{syn} from the time when the last thread finishes the for routine to the time point when the last thread unlocks `swap_count_mutex`. Figure 10 shows an example. The green "*for*" block represents the process for the thread to execute the current iteration of the for routine. The yellow "*mutex*" block means the process for the thread to possess of the lock. t_{syn} begins at the end of the thread C's "*for*" block, which is the last thread that finishes the for block, and ends at the end of the thread B's mutex "*block*".

```
pthread_mutex_lock(&swap_count_mutex);
if(joinCount < NUM_THREADS){
    joinCount++;
    pthread_cond_wait(&swap_loop_end, &swap_count_mutex);
} else {
    joinCount = 1;
    pthread_cond_broadcast(&swap_loop_end);
}
pthread_mutex_unlock(&swap_count_mutex);
```

- $P(M)$, the probability for a for iteration to call `ReduceSwapRestore` in a matrix M. $P(M) = \frac{N_{swap}}{N_{iteration}}$.

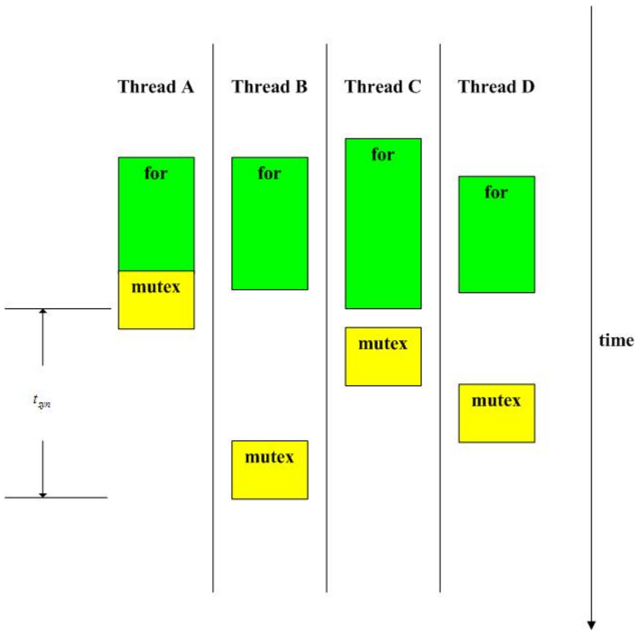

Figure 10: An example of computing t_{syn}

Id	N_{swap}	N_{syn}	$N_{iteration}$	$P(M)$	C_{syn}
1	37	16	72	0.514	4.5
2	15	8	36	0.417	4.5
3	32	18	81	0.395	4.5
4	32	12	54	0.593	4.5
5	36	20	90	0.4	4.5
6	32	14	63	0.508	4.5
7	33	14	63	0.524	4.5
8	44	16	72	0.611	4.5
9	33	16	72	0.458	4.5
10	29	14	63	0.46	4.5
Average				0.488	4.5

Table 3: Analysis of the thread-pool implementation by using ten $10 - by - 10$ matrices.

- C_{syn}, the average number of for iterations before a synchronization point. $C_{syn} = \frac{N_{iteration}}{N_{syn}} = \frac{n-1}{2}$ where n is the dimension of U.

Given ten $10 - by - 10$ matrices, we compute the values of these variables as in table 3.

From table 3 we know, for an $10 - by - 10$ matrix, there are 4.5 iterations for each for routine, and the probability for each iteration to call `ReduceSwapRestore` is 0.488. So in one $while$ iteration there may have $2 \approx 0.488 \times 4.5$ iterations which will call `ReduceSwapRestore` for each for routine, namely, before a synchronization point. Now we can compare the costs between the serial implementation and the new parallel implementation, approximately. For simplicity, assume we have 3 threads and $C_{syn} = 5$, figure 11 shows the possible processes of these two implementations. Every green block represents an iteration. In iterations 2 and 4, `ReduceSwapRestore` is called, so the cost should be $t_{check} + t_{swap}$. Other iterations just check the if condition, so the cost is t_{check}.

Therefore, for each for routine, the total cost of the serial implementation t_S and the total cost of the parallel implementation t_{TP} are

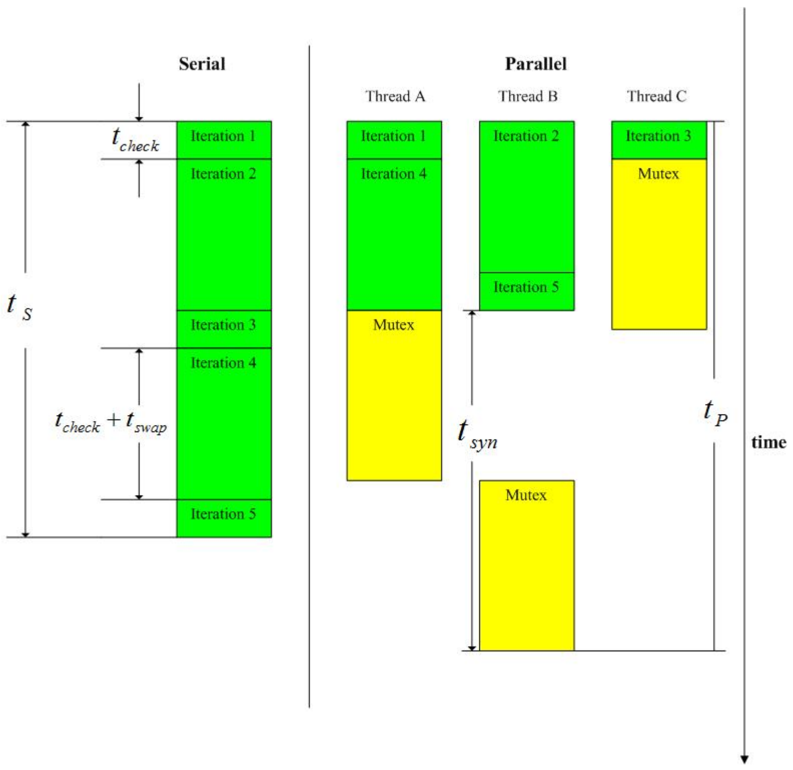

Figure 11: The for routine processes of the serial implementation and the new parallel implementation.

$t_S = 5t_{check} + 2t_{swap}$
$t_{TP} = 2t_{check} + t_{swap} + t_{syn}$
$t_S - t_{TP} = 3t_{check} + t_{swap} - t_{syn}$

- If $3t_{check} + t_{swap} \leq t_{syn}$, then $t_S \leq t_{TP}$, the program cannot benefits from the parallel implementation.
- If $3t_{check} + t_{swap} > t_{syn}$, then $t_S > t_{TP}$, the parallel implementation can improve the efficiency.

The above example is an ideal situation. In fact, `ReduceSwapRestore` may happen in iterations 1 and 4, which will make the parallel implementation slow. We'll compute $t_S - t_{TP}$ for more matrices of different dimensions. Assuming there are four threads for the thread-pool implementation, and we have an ideal situation like before. Table 4 show the results of 9 groups of different matrices, where each group has 10 matrices of the same dimension. $P(M)$ and C_{syn} are average values.

When the dimension (m, n) increments, t_{check} and t_{syn} may not change because they do not relate to n, but the coefficient of t_{check} becomes larger. In addition, t_{swap} increments because it correlates with (m, n). So when (m, n) increases, $t_S - t_{TP}$ increments, so the parallel implementation can improve efficiency when (m, n) reaches a large enough point. We can observe that in table 2 such point may be $(50, 50)$. Table 4 tells us possible $t_S - t_{TP}$ values which can be used to determine when the program can benefit if we already know t_{check}, t_{syn} and t_{swap}. However, as we said, the results of table 4 is an ideal situation. If the structure of decomposition is uneven, $t_S - t_{TP}$ may be very little, or less than 0, where the parallel implementation cannot gain too much advantage, or may even take more time.

Dimension	$P(M)$	C_{syn}	$P(M) \times C_{syn}$	t_S	t_{TP}	$t_S - t_{TP}$
20,20	0.326	9.5	3	$10t_{check} + 3t_{swap}$	$2t_{check} + t_{swap} + t_{syn}$	$8t_{check} + 2t_{swap} - t_{syn}$
30,30	0.206	14.5	3	$15t_{check} + 3t_{swap}$	$4t_{check} + t_{swap} + t_{syn}$	$11t_{check} + 2t_{swap} - t_{syn}$
40,40	0.21	19.5	4	$20t_{check} + 4t_{swap}$	$5t_{check} + t_{swap} + t_{syn}$	$15t_{check} + 3t_{swap} - t_{syn}$
50,50	0.136	24.5	3	$25t_{check} + 3t_{swap}$	$7t_{check} + t_{swap} + t_{syn}$	$18t_{check} + 2t_{swap} - t_{syn}$
100,100	0.11	49.5	5	$50t_{check} + 5t_{swap}$	$13t_{check} + 2t_{swap} + t_{syn}$	$37t_{check} + 3t_{swap} - t_{syn}$
200,200	0.055	99.5	5	$100t_{check} + 5t_{swap}$	$25t_{check} + 2t_{swap} + t_{syn}$	$75t_{check} + 3t_{swap} - t_{syn}$
300,300	0.04	149.5	6	$150t_{check} + 6t_{swap}$	$38t_{check} + 2t_{swap} + t_{syn}$	$112t_{check} + 4t_{swap} - t_{syn}$
400,400	0.028	199.5	6	$200t_{check} + 6t_{swap}$	$50t_{check} + 2t_{swap} + t_{syn}$	$150t_{check} + 4t_{swap} - t_{syn}$
500,500	0.022	249.5	6	$250t_{check} + 6t_{swap}$	$63t_{check} + 2t_{swap} + t_{syn}$	$187t_{check} + 4t_{swap} - t_{syn}$

Table 4: $t_S - t_{TP}$ of 9 groups of matrices

6.3.2 Implementation of the *Reduce* Part

Let's first see the serial implementation of the *Reduce* part.

```
int i,j,start;
for(k = 2*n-3;k >= 1;k--){
    if(k <= n-1){
        start = 1;
    }
    else {
        start = k-n+2;
    }
    for(i = start; i < (k+3)/2; i++){
        j = k+2-i;
        if(absd(U[i][j]) > 0.5){
            Reduce(i, j, n, B, U, M);
        }
    }
}
```

B, U and M are the same symbols as we explained in previous section. *Reduce*(i, j) is realized by function Reduce. Function absd returns the abstract value of $U[i][j]$.

A parallel implementation with a thread pool We first designed the *Swap* part according to the peer model for mutithreaded programs. One thread creates other threads on demand. When each thread completes its iteration, it exits. It can slow our program clearly because we don't reuse idle threads to handle new iteration. Rather, we create and destroy a thread for each iteration. Consequently, our program spends a lot of time in the Pthreads library. We addressed these performance snags by redesigning it to use a thread pool. Also, we'll use this design technique to implement the *Reduce* part.

```
const int NUM_THREADS = 4;
static int joinCount = 1;
pthread_mutex_t reduce_count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t reduce_loop_end = PTHREAD_COND_INITIALIZER;
```

```
struct work_reduce_t {
    int offset;
    int n;
    basis *B;
    double **U;
    double **M;
};


void *subReduce(void *w_reduce) {
    work_reduce_t *work = (work_reduce_t*)w_reduce;
    int k,i,j,start;
    reduceCount[work->offset] = 0;
    for(k = 2 * (work->n) - 3 ; k >= 1; k-- ){
        if( k <= (work->n)-1){
            start = 1;
        }
        else{
            start = k - (work->n) + 2;
        }
        for(i = start + (work->offset);i < (k+3)/2;i += NUM_THREADS){
            j = k + 2 - i;
            if(absd((work->U)[i][j])>0.5){

              Reduce(i, j, work->n, work->B, work->U, work->M);
        }
    }

    pthread_mutex_lock(&reduce_count_mutex);
    if(joinCount < NUM_THREADS){
        joinCount++;
        pthread_cond_wait(&reduce_loop_end, &reduce_count_mutex);
    }else{
        joinCount = 1;
        pthread_cond_broadcast(&reduce_loop_end);
    }
    pthread_mutex_unlock(&reduce_count_mutex);
    }
}
...
void parallelLLL(const int m, const int n, basis *A, double w){
    ...
    pthread_t reduce_t[NUM_THREADS];
    work_reduce_t *w_reduce[NUM_THREADS];
    for(int i = 0;i < NUM_THREADS;i++){
        w_reduce[i] = new work_reduce_t;
        w_reduce[i]->offset = i;
        w_reduce[i]->n = n;
```

Dimensions	10,10	20,20	50,50	100,100	200,200	300,300	400,400	500,500
Reduce Times(S)	33	121	486	655	2220	3062	6023	2717
Reduce Times(TP)	33	121	486	655	2220	3062	6023	2717
Time Cost(S)	55	327	1095	3026	19201	39883	104778	63777
Time Cost(TP, 2 threads)	920	541	1633	2786	14880	28908	67252	47737
Time Cost(TP, 3 threads)	1467	2164	1791	3917	14346	29961	70382	52004
Time Cost(TP, 4 threads)	1261	1886	1899	3581	14949	29693	63720	56016

Table 5: Comparison of the *Reduce* cost between the serial implementation and the parallel implementation using a thread pool.

```
            w_reduce[i]->B = B;
            w_reduce[i]->U = U;
            w_reduce[i]->M = M;
            pthread_create(&reduce_t[i], NULL, subReduce, (void*)w_reduce[i]);
        }
        for(int i = 0;i < NUM_THREADS;i++){
            pthread_join(reduce_t[i],NULL);
        }
        ...
    }
    ...
```

The symbols `B`, `U`, `M`, `n` and `offset` in structure `work_reduce_t` are the same as we defined in structure `work_swap_t`, but here we don't need `m`, `D`, `w` anymore. The lock `reduce_count_mutex` is used for synchronization. The condition variable `reduce_loop_end` represents the event that the variable `jointCount` reaches `NUM_THREADS`. Each thread calls an individual `subReduce` which handles a different part of the values of `i` in the inner for routine assigned by `parallelLLL` of the boss thread.

Testing the thread-pool implementation Given some matrices of different dimensions, table 5 shows its efficiency compared to the serial version. "*Reduce* Times" represents the number of times that `Reduce` has been executed. S is the serial version and TP is the parallel version. "Time cost" represents cost spent on the *Reduce* part in each test. The unit of time is microsecond.

Like the similar implementation for the *Swap* part, the two problems: *Overhead* and *Structure of Decomposition* affect the efficiency of this implementation. Note that in Table 2, we can gain efficiency increase when dimension just reaches $(50, 50)$, but in table 5, it's $(100, 100)$. To see why, let's compute $t_S - t_P$ as we did in table 4 under the same situation, but first we need to redefine some variables.

- $N_{iteration}$: the total number of iterations of the inner for routine in the *Reduce* part.
- N_{reduce}: the total number of the invoked times of the procedure `Reduce`.
- N_{syn}: the total number of the synchronization points in the parallel implementation.
- t_{reduce}: the average cost of executing the procedure `Reduce`.
- t_{check}: the average cost of executing each inner for iteration which contains the average time of computing `start`, plus the average cost used to check the condition of `if` statement, but without the statements within it.
- t_{syn}: the average cost for a synchronization point. We use the same calculation rule as in figure 10.

Dimension	$P(M)$	C_{syn}	$P(M) \times C_{syn}$	t_S	t_{TP}	$t_S - t_{TP}$
20,20	0.545	5.135	3	$5t_{check} + 3t_{reduce}$	$2t_{check} + t_{reduce} + t_{syn}$	$3t_{check} + 2t_{reduce} - t_{syn}$
30,30	0.444	7.632	3	$8t_{check} + 3t_{reduce}$	$2t_{check} + t_{reduce} + t_{syn}$	$6t_{check} + 2t_{reduce} - t_{syn}$
40,40	0.366	10.13	4	$10t_{check} + 4t_{reduce}$	$3t_{check} + t_{reduce} + t_{syn}$	$7t_{check} + 3t_{reduce} - t_{syn}$
50,50	0.34	12.629	4	$13t_{check} + 4t_{reduce}$	$5t_{check} + t_{reduce} + t_{syn}$	$8t_{check} + 3t_{reduce} - t_{syn}$
100,100	0.169	25.127	4	$25t_{check} + 4t_{reduce}$	$7t_{check} + t_{reduce} + t_{syn}$	$18t_{check} + 3t_{reduce} - t_{syn}$
200,200	0.076	50.126	4	$50t_{check} + 4t_{reduce}$	$13t_{check} + t_{reduce} + t_{syn}$	$37t_{check} + 3t_{reduce} - t_{syn}$
300,300	0.066	75.126	5	$75t_{check} + 5t_{reduce}$	$19t_{check} + 2t_{reduce} + t_{syn}$	$56t_{check} + 3t_{reduce} - t_{syn}$
400,400	0.056	100.125	6	$100t_{check} + 6t_{reduce}$	$25t_{check} + 2t_{reduce} + t_{syn}$	$75t_{check} + 4t_{reduce} - t_{syn}$
500,500	0.039	125.125	5	$125t_{check} + 5t_{reduce}$	$32t_{check} + 2t_{reduce} + t_{syn}$	$93t_{check} + 3t_{reduce} - t_{syn}$

Table 6: $t_S - t_{TP}$ of 9 groups of matrices

Dimension\\$t_S - t_{TP}$	the *Swap* part	the *Reduce* part
20,20	$8t_{check} + 2t_{swap} - t_{syn}$	$3t_{check} + 2t_{reduce} - t_{syn}$
30,30	$11t_{check} + 2t_{swap} - t_{syn}$	$6t_{check} + 2t_{reduce} - t_{syn}$
40,40	$15t_{check} + 3t_{swap} - t_{syn}$	$7t_{check} + 3t_{reduce} - t_{syn}$
50,50	$18t_{check} + 2t_{swap} - t_{syn}$	$8t_{check} + 3t_{reduce} - t_{syn}$
100,100	$37t_{check} + 3t_{swap} - t_{syn}$	$18t_{check} + 3t_{reduce} - t_{syn}$
200,200	$75t_{check} + 3t_{swap} - t_{syn}$	$37t_{check} + 3t_{reduce} - t_{syn}$
300,300	$112t_{check} + 4t_{swap} - t_{syn}$	$56t_{check} + 3t_{reduce} - t_{syn}$
400,400	$150t_{check} + 4t_{swap} - t_{syn}$	$75t_{check} + 4t_{reduce} - t_{syn}$
500,500	$187t_{check} + 4t_{swap} - t_{syn}$	$93t_{check} + 3t_{reduce} - t_{syn}$

Table 7: Comparison of $t_S - t_{TP}$ between the parallel implementations with a thread pool on the *Swap* part and the *Reduce* part

- $P(M)$, the probability for an inner *for* iteration to call `Reduce` in a matrix M. $P(M) = \frac{N_{reduce}}{N_{iteration}}$.
- C_{syn}, the average number of inner *for* iterations before a synchronization point. $C_{syn} = \frac{N_{iteration}}{N_{syn}}$.

Table 6 shows $t_S - t_{TP}$ for 90 matrices of different dimensions using 4 threads. Comparisons between table 4 and table 6 are showed in table 7.

In table 7, assuming t_{syn} of the *Swap* part and t_{syn} of the *Reduce* part are almost the same, two t_{check} are both $O(1)$. For the same dimension matrix, on the one hand, the coefficient of t_{check} in the *Swap* part is almost two times more than that in the *Reduce* part. On the other hand, t_{swap} is more than t_{reduce} because `ReduceSwapRestore` executes more codes than *Reduce*. Therefore, $t_S - t_{TP}$ in the *Swap* part will reach 0 prior to $t_S - t_{TP}$ in the *Reduce* part, which explains what we found between table 2 and table 5.

A parallel implementation with reduction list In the thread-pool implementation of the *Swap* part, it always needs to call `ReduceSwapRestore` at some positions of k in each iteration of the *while* loop, because if not, it means *finish* is true and the program will end. However, in the implementation of the *Reduce* part, there is no need to call `Reduce` in some iterations of the outer *for* loop. In such an iteration, the inner *for* routine will only compute the variable `j` and check the condition of `if`, but never call `Reduce`. This give rise to another parallel implementation.

In this new implementation, we'll use a boss thread to compute the positions that need to be reduced in each iteration of the outer *for* loop. When it finds such a position, the thread will first store it on a linked list without calling `Reduce`, then check another position. After the thread stores all positions that need to be reduced in current iteration, it will assign them to other worker threads to reduce one by one. If there is no such position, the thread will enter into the next iteration of the outer *for* loop without waking up other worker threads. This method may increase efficiency from three aspects:

- Since there is no need to call other worker threads when there is no position to be reduced in an iteration, the synchronization part at the end can be omitted.
- The positions assigned to other worker threads are all needed to be reduced, so each thread will get an even part of work, which eliminates the problem of *Structure of Decomposition.*
- Although the boss thread checks all positions serially, since t_{check}, even after plusing the expenses of operations for storing positions, is very small, the benefits from the above two aspects may be much more than those saved from assigning check procedures to several worker threads.

We'll use a structure to record position information and a linked list to link these positions.

```
const int NUM_THREADS = 4;
static int joinCount = 1;
bool end = false;


struct work_reduce_t {
    int id;
    int n;
    basis *B;
    double **U;
    double **M;
};


struct node {
    int i;
    int j;
    node *next;
};
node* linklist = NULL;
node* head = NULL;
node* current = NULL;


pthread_mutex_t reduce_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t list_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t list_not_empty = PTHREAD_COND_INITIALIZER;


void *subReduce(void *w_reduce) {
    work_reduce_t *work = (work_reduce_t*)w_reduce;
    int i, j;
    bool run = false;
    while(!end){
        pthread_mutex_lock(&reduce_mutex);
        if(current == NULL){
           if(joinCount < NUM_THREADS){
             joinCount++;
             pthread_cond_wait(&list_not_empty, &reduce_mutex);
           }
           else {
             joinCount = 1;
             pthread_cond_signal(&list_empty);
             pthread_cond_wait(&list_not_empty, &reduce_mutex);
           }
        }
        else {
           run = true;
           i = current->i;
           j = current->j;
           current = current->next;
        }
        pthread_mutex_unlock(&reduce_mutex);
        if(run){
           run = false;
           Reduce(i,j,work->n, work->B, work->U, work->M);
        }
    }
}


void parallelLLL(const int m, const int n, basis *A, double w){
    ...
    pthread_mutex_lock(&reduce_mutex);
    int i,j,start;
    linklist = new node;
    head = linklist;
    pthread_t reduce_t[NUM_THREADS];
    work_reduce_t *w_reduce[NUM_THREADS];
    for(int i = 0; i < NUM_THREADS; i++){
        w_reduce[i] = new work_reduce_t;
        w_reduce[i]->id = i;
        w_reduce[i]->n = n;
        w_reduce[i]->B = B;
        w_reduce[i]->U = U;
        w_reduce[i]->M = M;
        pthread_create(&reduce_t[i], NULL, subReduce, (void*)w_reduce[i]);
    }
    for(k = 2 * n - 3;k >= 1;k--){
        if(k <= n-1){
           start = 1;
        }
        else {
           start = k - n + 2;
        }
        for(i = start;i < (k+3)/2;i++){
           j = k + 2 - i;
           if(absd(U[i][j])>0.5){
             linklist->next = new node;
             linklist = linklist->next;
             linklist->i = i;
             linklist->j = j;
             linklist->next = NULL;
             if (current == NULL) {
               current = linklist;
             }
           }
        }
        if(current != NULL) {
           pthread_cond_broadcast(&list_not_empty);
           pthread_cond_wait(&list_empty, &reduce_mutex);
        }
    }
    end = true;
    pthread_cond_broadcast(&list_not_empty);
    pthread_mutex_unlock(&reduce_mutex);

    for (int i = 0;i < NUM_THREADS;i++){
        pthread_join(reduce_t[i],NULL);
    }

    node* previous;
    while (head != NULL){
        previous = head;
        head = head->next;
        delete previous;
    }
    ...
}
```

node is a structure used to store i and j of a position of U that needed to be reduced. The pointer linklist is used to construct a list and head points to the first node of the list, which will be used to delete this list at the end. current is a pointer that points to the head of the positions that have not been reduced. The list work as figure 12 shows. When the program begins, it creates the first node without any value of i or j so that the program can delete this list from this node later. When it finds a position which is the first one, it creates a node and link it to the tail of the list and set current and linklist to point to this node. If the program finds another position, it will create another node, add it to the tail of the list and set linklist to this new node. When the program finds all positions in current iteration of the outer *for* routine, it may look like what the figure shows, where linklist points to the last node whose next node is set to be NULL. Then the program will enter the next iteration and add new nodes.

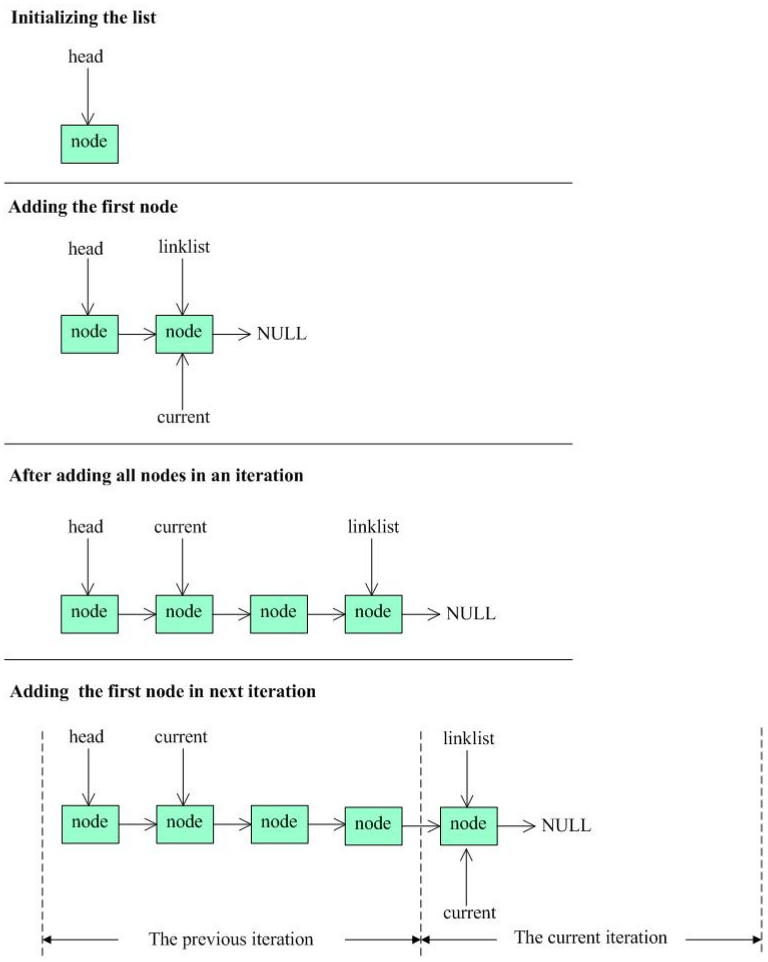

Figure 12: Procedures of adding nodes in a list

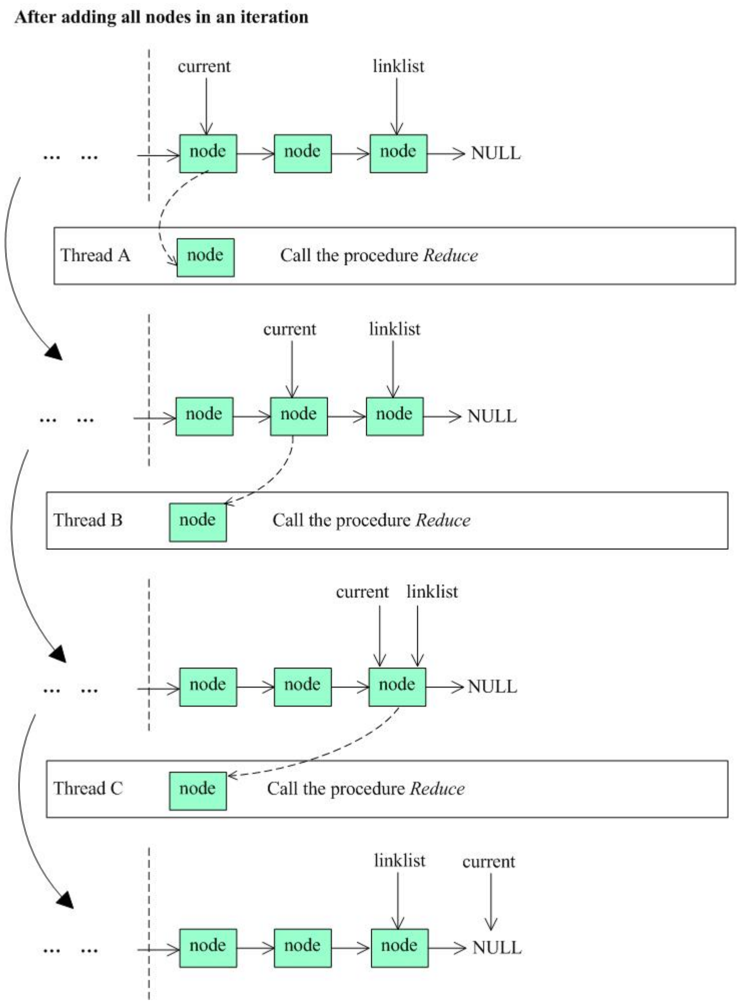
After adding all nodes in an iteration
current
linklist
node
NULL
Thread A
Call the procedure *Reduce*
Thread B
Thread C

Figure 13: Procedures of worker threads on a list

The condition variable `list_empty` represents an event - the list is empty or all nodes on the list have been reduced. `list_not_empty` represents that new node(s) has(have) been added to the list. After each iteration of the outer *for* loop, if the program finds some positions, that is, `current!=NULL`, the boss thread will wake up other threads, then sleep and wait for the worker threads to reduce these positions on the list. If there is no such position, namely, `current=NULL`, the program will enter the next iteration directly, without any synchronization. When the worker threads are waken up, they will work as figure 13 shows. Each thread selects a node from the list at a time and calls `Reduce`. If a thread is holding the list, other threads are forbidden to possess it in order to avoid race conditions caused by writing a value to `current` simultaneously. When a thread fetches a node, firstly, it stores its values of `i` and `j` to the local variable `i` and `j`, and set `current` to point to the next node. Then, after it releases the lock `reduce_mutex` so that other threads can fetch nodes from the list, it calls `Reduce`. If a thread finds `current` to be `NULL`, which means all nodes have been reduced, it will signal our `list_empty` condition variable to wake up the boss thread and wait for another cycle.

u

Testing the reduction-list implementation To compare efficiencies between the reduction-list implementation and the thread-pool implementation, we'll test their performances by using

Dimensions	10,10	20,20	50,50	100,100	200,200	300,300	400,400	500,500
Reduce Times(S)	33	121	486	655	2220	3062	6023	2717
Reduce Times(TP/RL)	33	121	486	655	2220	3062	6023	2717
Time Cost(S)	55	327	1095	3026	19201	39883	104778	63777
Cost(TP, 2 threads)	920	541	1633	2786	14880	28908	67252	47737
Cost(TP, 3 threads)	1467	2164	1791	3917	14346	29961	70382	52004
Cost(TP, 4 threads)	1261	1886	1899	3581	14949	29693	63720	56016
Cost(RL, 2 threads)	1920	1214	1248	7490	14120	28429	68781	48329
Cost(RL, 3 threads)	2025	6035	1569	6195	13343	26960	77149	47819
Cost(RL, 4 threads)	1417	1043	1930	3079	13673	27169	79306	51302

Table 8: Comparison of the *Reduce* cost among the serial implementation, the implementation with a thread pool and the implementation with reduced list.

the same matrices used in table 5. Table 8 shows the results. “RL” means the reduction-list implementation, and “TP” represents the thread-pool implementation.

From table 8 we know that it is not possible to say that one parallel implementation is better than another. The reason is that the efficiencies of them depend on the specific matrix and the running platform. However, we can decide which one will be used in different situations on our platform. Before that, we need more tests.

Figure 14 to figure 21 shows the results of 100 random matrices, 10 matrices for each dimension, by using 7 kinds of implementations. The meanings of the symbols used in each figure are

- serial: the serial implementation
- TP 2: the thread-pool implementation by using 2 threads.
- TP 3: the thread-pool implementation by using 3 threads.
- TP 4: the thread-pool implementation by using 4 threads.
- RL 2: the reduction-list implementation by using 2 threads.
- RL 3: the reduction-list implementation by using 3 threads.
- RL 4: the reduction-list implementation by using 4 threads.

From figure 14 to figure 17 we know that no parallel implementation can run faster than the serial implementation when dimension is smaller than (50,50). But from figure 18 we already find in some matrices, the thread-pool implementation by using 2 threads is better.

Figure 19 shows that the thread-pool implementation by using 3 or 4 threads are bad choices, but others cannot be distinguished better or worse.

From figure 20 to 23 we notice the reduction-list implementation by using 3 or 4 threads are always the most efficient implementations, which proves that the reduction-list implementation is a good choice when the dimension of the computed basis is larger than (100,100).

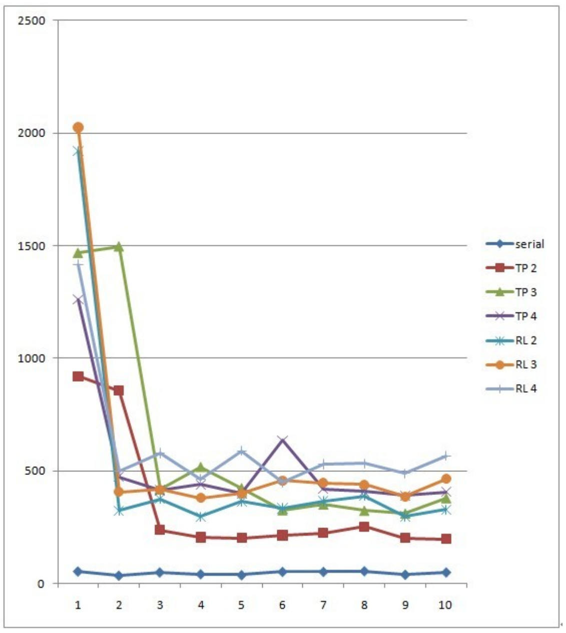

Figure 14: Costs of ten $10 - by - 10$ matrices

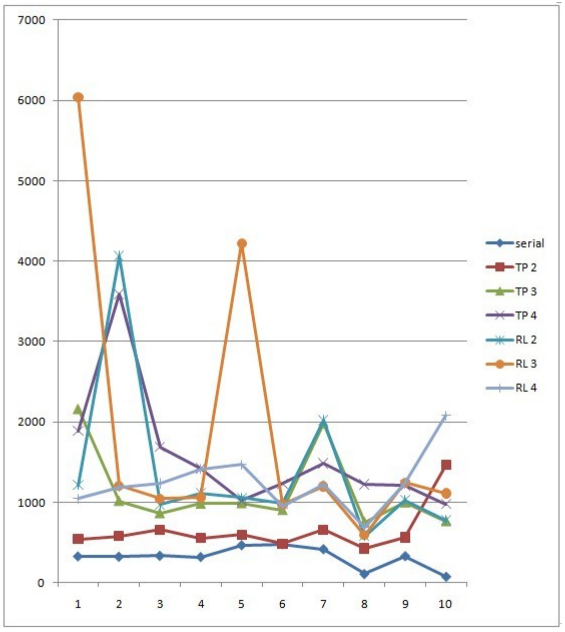

Figure 15: Costs of ten $20 - by - 20$ matrices

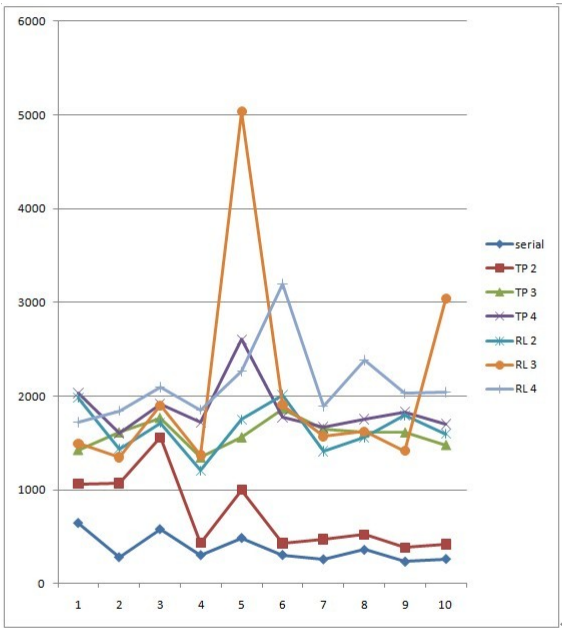

Figure 16: Costs of ten $30 - by - 30$ matrices

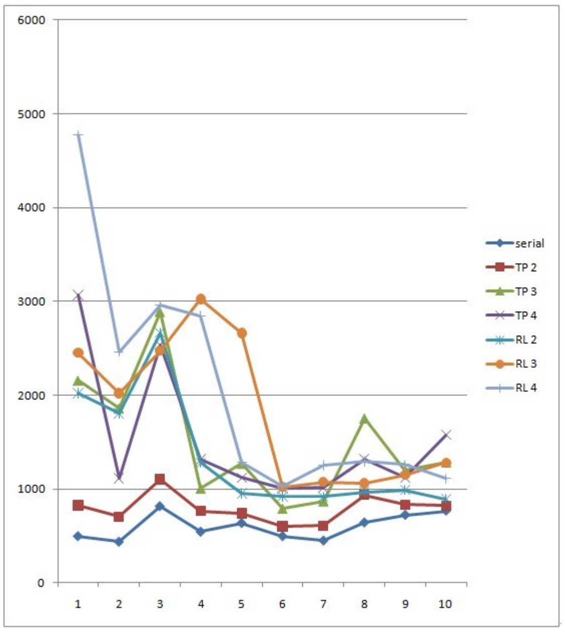

Figure 17: Costs of ten $40 - by - 40$ matrices

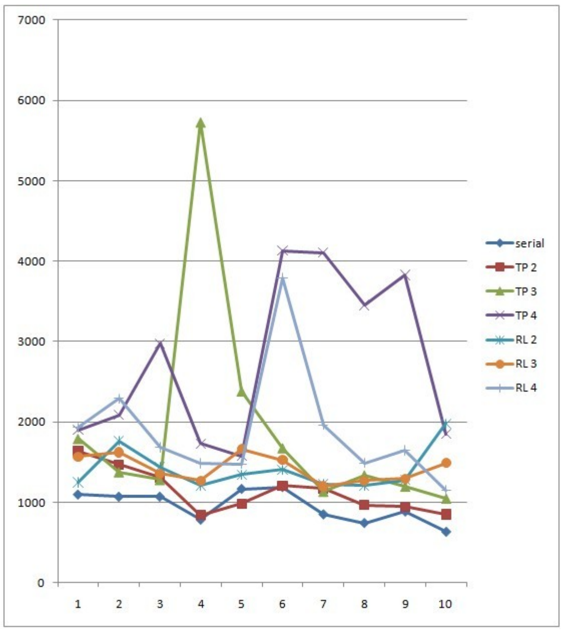

Figure 18: Costs of ten $50 - by - 50$ matrices

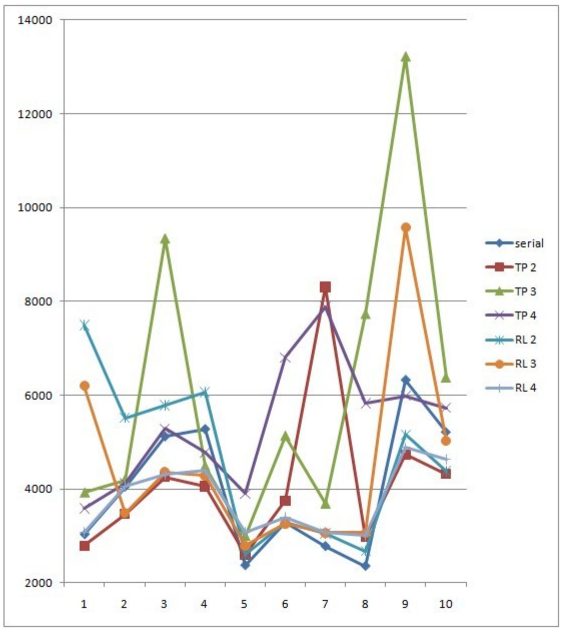

Figure 19: Costs of ten $100 - by - 100$ matrices

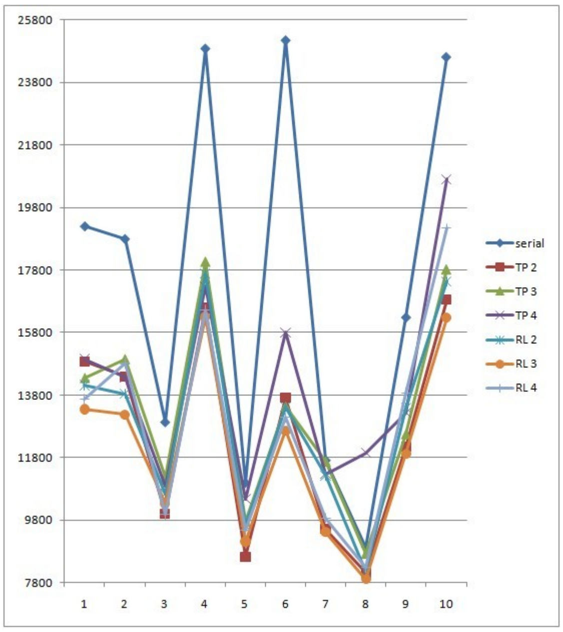

Figure 20: Costs of ten $200 - by - 200$ matrices

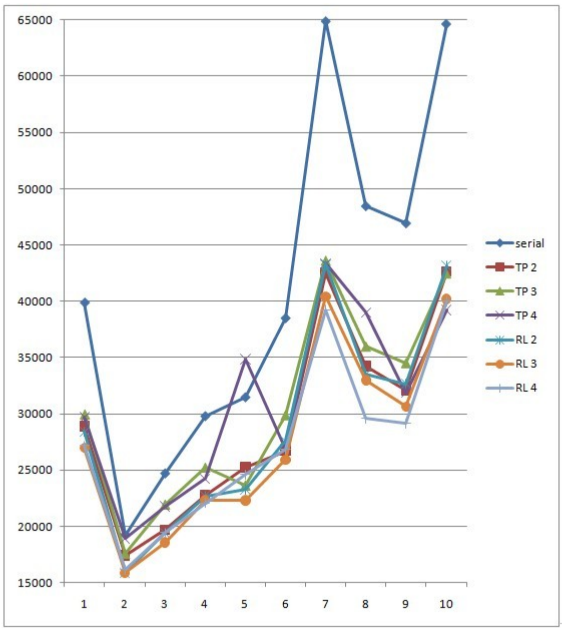

Figure 21: Costs of ten $300 - by - 300$ matrices

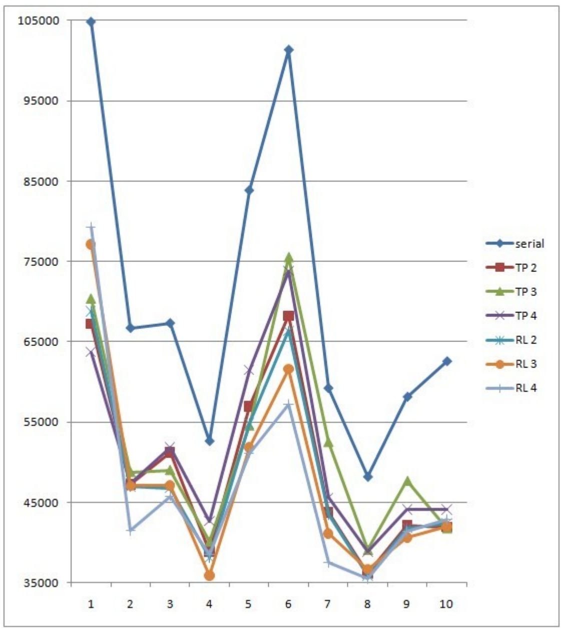

Figure 22: Costs of ten $400 - by - 400$ matrices

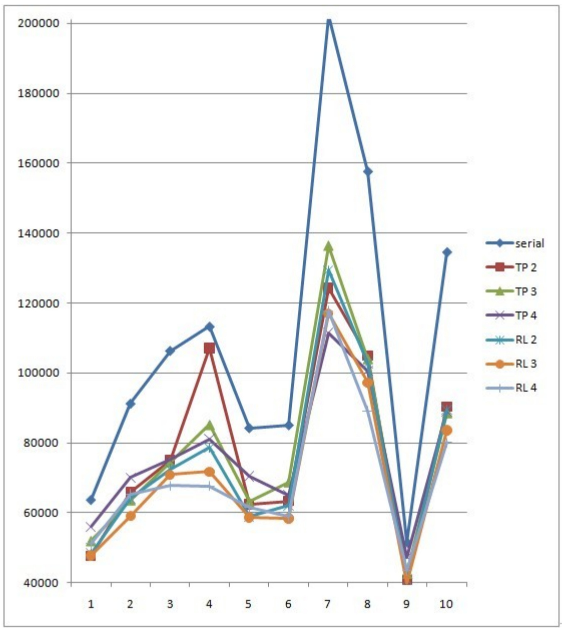

Figure 23: Costs of ten $500 - by - 500$ matrices

Appendix Code listings

For simplicity, we just give the parallel implementation with Pthread, where the "Swap" part is implemented by using a thread pool, and the "Reduce" part is implemented by using a reduction list. Also, we give a simple test file.

```
/*
*   vector.h
*
*   Class vector : data structure of the vector in linear algebra,
*   which can be represented by an n-tuple of real numbers.
*       V = [Vi] = (v1, v2, . . . . . , vn)
*   The class defines the necessary input and ouput functions and
*   arithmetic operations on vectors:
*       - Multiplication of vectors
*       - Scalar Multiplication
*       - Addition
*       - Subtraction
*   Also, the class defines other useful functions for vector computing.
*/

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <stdexcept>

using namespace std;

#ifndef VECTOR_H
#define VECTOR_H

class vector{

    //output operator
    friend ostream& operator<<(ostream&, const vector&);

    //Multiplication of vectors
    friend double operator*(const vector&, const vector&);

    //Scalar Multiplication
    friend vector operator*(double, const vector&);
    friend vector operator*(const vector&, double);

    //Addition
    friend vector operator+(const vector&, const vector&);

    //Subtraction
    friend vector operator-(const vector&, const vector&);

    //The number of the elements in the vector
    size_t length;

    //Point to an array that stores elements of the vector
    double *val;

public:

    //Default constructor
    vector();

    //Constructor by length
    vector(const size_t);

    //Copy constructor
    vector(const vector&);

    //Overloaded assignment operator =
    vector& operator=(const vector&);

    //Overloaded subscript operator []
    double& operator[](const size_t);

    //Overloaded subscript operator []
    const double& operator[](const size_t) const;

    //Input function
    void input();

    //Computer two vectors, verify whether they are independent
    bool isIndependent(const vector&);

    //Give the number of elements in this vector
    size_t size(){
        return length;
    }

    //destructor
    ~vector();
};

#endif




/*
*   vector.cpp
*
*   Implementation of the class vector.
*
*/

#include "vector.h"
```

```
/*
*   Function: default constructor
*   ---------------------------------------------------
*   The default vector has no any element in it.
*
*/
vector::vector(){
    length = 0;
    val = 0;
};

/*
*   Function: constructor using an length
*   ---------------------------------------------------
*   The values of the vector will be 0.
*
*/
vector::vector(const size_t l):length(l){
    val = new double[l];
    for(size_t i = 0;i != length;++i){
        *(val+i) = 0;
    }
}

/*
*   Function: copy constructor
*   ---------------------------------------------------
*   If the lengths of two vectors are not the same, an exception will be thrown.
*
*/
vector::vector(const vector& vec):length(vec.length){
    try{
        if(length != vec.length)
            throw logic_error("Error: Not the same length \
in copy constructor!");
        else{
            if(length != 0){
                val = new double[length];
                for(size_t i = 0;i != length;++i){
                    *(val + i) = *(vec.val + i);
                }
            }
        }
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
}

/*
*   Function: assignment operator =
*   ---------------------------------------------------
*   Assignment operator of two vectors. As in copy constructor, If the lengths
*   of them are not the same, an exception will be thrown.
*
*/
vector& vector::operator=(const vector& vec){
    try{
        if(length != vec.length)
            throw logic_error("Error: Not the same length in operator = !");
        else{
            if(length != 0){
                for(size_t i = 0;i != length;++i){
                    *(val + i) = *(vec.val + i);
                }
            }
        }
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return *this;
}

/*
*   Function: subscript operator =
*   ---------------------------------------------------
*   Get the ith element using the subscript of the vector (l-value)
*
*/
double& vector::operator[](const size_t i){
    try{
        if(i <= length && i > 0)
            return *(val + i - 1);
        else
            throw logic_error("Error:Wrong index!");
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return *val;
}

/*
*   Function: subscript operator =
*   ---------------------------------------------------
*   Get the ith element using the subscript of the vector (r-value)
*
*/
const double& vector::operator[](const size_t i) const{
    try{
        if(i <= length && i > 0)
            return *(val + i - 1);
        else
            throw logic_error("Error:Wrong index!");
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return *val;
}

/*
*   Function: input
*   ----------------------------------------------------
*   Initialize the vector or change the values in this vector.
*
*/
void vector::input(){
    double d;
    for(size_t i = 0;i < length;i++){
        cout<<"Input element "<<i + 1<<" : ";
        cin>>d;
        *(val + i) = d;
    }
    cout<<endl;
}

/*
*   Function: isIndependent
*   ----------------------------------------------------
*   Compute two vectors, verify whether they are independent. In linear
*   algebra, a family of vectors is linearly independent if none of them
*   can be written as a linear combination of finitely many other vectors
*   in the collection. Two vectors P and Q are dependent when the determinant
*   of the matrix
*                           | P*P P*Q |
*                           | Q*P Q*Q |
*   equals to 0, where * is inner product of vectors.
*
*/
bool vector::isIndependent(const vector & v){
    double d = ((*this) * (*this)) * (v*v) - ((*this) * v) * (v * (*this));
    if(d)
        return true;
    return false;
}

/*
*   Function: destructor
*   ----------------------------------------------------
*   Since we have a double pointer val that points to an array that stores
*   elements of the vector, we need to free these memories if we don't use
*   this vector anymore.
*
*/
vector::~vector(){
    delete [] val;
}
```

```
/*
*   Function: output operator <<
*   -----------------------------------------------------
*   Output the elements of the vector.
*
*/
ostream& operator<<(ostream& os, const vector &b){
    for(size_t i = 0;i != b.length;i++){
        os<<*(b.val + i)<<endl;
    }
    return os;
}

/*
*   Function: operator *
*   -----------------------------------------------------
*   Representing inner product of two vectors. If the lengths of them are
*   not the same, an exception will be thrown.
*
*/
double operator*(const vector& bl, const vector& br){
    double sum = 0;
    try{
        if(bl.length != br.length)
            throw logic_error("Error: not the same length in operator * !");
        else{
            for(size_t i = 0;i != bl.length;i++){
                sum += (*(bl.val + i)) * (*(br.val + i));
            }
        }
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return sum;
}

/*
*   Function: operator *
*   -----------------------------------------------------
*   Representing the scalar multiplication on a vector, which is the product
*   of a vector and a numerical value, the numerical value is on the left.
*
*/
vector operator*(double d, const vector& br){
    vector b(br.length);
    for(size_t i = 0;i != br.length;i++){
        *(b.val + i) = d * (*(br.val + i));
    }
    return b;
}
/*
```

```
*   Function: operator *
*   ------------------------------------------------------
*   Also represent the scalar multiplication on a vector, but the numerical
*   value is on the right.
*
*/
vector operator*(const vector& bl, double d){
    vector b(bl.length);
    for(size_t i = 0;i != bl.length;i++){
        *(b.val + i) = d * (*(bl.val + i));
    }
    return b;
}

/*
*   Function: operator +
*   ------------------------------------------------------
*   Addition between two vectors. If the lengths of them are not the same,
*   an exception will be thrown.
*
*/
vector operator+(const vector& bl, const vector& br){
    vector b(bl.length);
    try{
        if(bl.length != br.length)
            throw logic_error("Error: not the same length in operator + !");
        else
        {
            for(size_t i = 0;i != bl.length;i++){
                *(b.val + i) = (*(bl.val + i)) + (*(br.val + i));
            }
        }
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return b;
}

/*
*   Function: operator -
*   ------------------------------------------------------
*   Substraction between two vectors. If the lengths of them are not the same,
*   an exception will be thrown.
*
*/
vector operator-(const vector& bl, const vector& br){
    vector b(bl.length);
    try{
        if(bl.length != br.length)
            throw logic_error("Error: not the same length in operator - !");
        else
        {
            for(size_t i = 0;i != bl.length;i++){
                *(b.val + i) = (*(bl.val + i)) - (*(br.val + i));
            }
        }
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return b;
}




/*
*    basis.h
*
*    Class basis : data structure of the basis for lattice.
*    In linear algebra, a basis is a set of linearly independent vectors that,
*    in a linear combination, can represent every vector in a given vector
*    space or free module, or more simply put form a "coordinate system".
*    The class defines the necessary input and ouput functions, subscript
*    operator and file operations.
*
*/

#include "vector.h"

#ifndef BASIS_H
#define BASIS_H

class basis {

    //Overloaded operator <<
    friend ostream& operator<<(ostream&, const basis&);

    //The number of coloumns in the basis.
    size_t n;

    //The number of rows in the basis.
    size_t m;

    //Pointer to a 2-dimensioned array which stores the values of the basis.
    vector **b;

    //The lower bound of the range in which a random number will be generated.
    static const int lowest = -10;

    //The higher bound of the range in which a random number will be generated.
    static const int highest = 10;

public:

    //Default constructor.
    basis():n(0),m(0){
        b = 0;
    }

    //Constructor using dimension
    basis(const size_t, const size_t);

    //Copy constructor.
    basis(const basis&);

    //Interface to return the number of coloumns: n.
    size_t sizeN(){
        return n;
    }

    //Interface to return the number of rows: m.
    size_t sizeM(){
        return m;
    }

    //Input function
    void input();

    //Overloaded subscript operator []
    vector &operator[] (const size_t);
    const vector &operator[] (const size_t) const;

    //Generate an random basis whose vectors are independent
    void randomBasis();

    //Write the basis to a fixed file
    bool writeToFile();

    //Initialize the basis from a fixed file
    bool readFromFile();

    //Write the basis to a given file
    bool writeToFile(string);

    //Initialize the basis from a given file
    bool readFromFile(string);

    //Destructor
    ~basis(){
        delete [] b;
     }
};

#endif



/*
```

```
*    basis.cpp
*
*    Implementation of basis class
*
*/

#include "basis.h"
#include "vector.cpp"
#include <cstdlib>
#include <ctime>

/*
*    Function: default constructor
*   ---------------------------------------------------
*    Constructor using the dimension of the basis, that is, the number of
*    vectors that this basis contains, and the number of elements in these
*    vectors each.These values in the basis are not initialized.
*
*/
basis::basis(const size_t ml, const size_t nl){
    m = ml;
    n = nl;
    try{
        if(m > 0 && n > 0){
            b = new vector*[n];
            for(size_t i = 0;i != n;i++){
                b[i] = new vector(m);
            }
        }
        else
            throw logic_error("Error,constructing with wrong sizes in \
basis(const size_t n, const size_t m)!");
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
}

/*
*    Function: operator []
*   ---------------------------------------------------
*    Overloaded subscript operator [], give the ith vector in the basis (l-value)
*
*/
vector& basis::operator[] (const size_t i){
    try{
        if(i <= n && i>0)
            return *(b[i-1]);
        else
            throw logic_error("Error: wrong index in basis::operator[]");
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return *(b[0]);
}

/*
*    Function: operator []
*   -------------------------------------------------------
*    Overloaded subscript operator [], give the ith vector in the basis (r-value)
*
*/
const vector& basis::operator[] (const size_t i) const{
    try{
        if(i <= n && i > 0)
            return *(b[i-1]);
        else
            throw logic_error("Error: wrong index in basis::operator[]");
    }
    catch(logic_error l){
        cout<<l.what()<<endl;
    }
    return *(b[0]);
}

/*
*    Function: operator <<
*   -------------------------------------------------------
*    Overloaded operator <<, output the values in the basis as the form of a
*    matrix.
*
*/
ostream& operator<<(ostream& os, const basis& ba){
    for(size_t i = 1;i != ba.m + 1;i++){
        cout<<"| ";
        for(size_t j = 1;j != ba.n + 1;j++){
            cout<<ba[j][i]<<" ";
        }
        cout<<"|"<<endl;
    }
    return os;
}

/*
*    Function: randomBasis
*   -------------------------------------------------------
*    Generate an random basis whose vectors are independent, and the values
*    are in the range of "lowest" to "highest".
*
*/
void basis::randomBasis(){
    srand((unsigned)time(0));
    int range = (highest - lowest) + 1;
    for(size_t i = 0;i < n;i++){
        for(size_t j = 1;j <= m;j++){
            (*(b[i]))[j] = lowest + int(range * (rand() / (RAND_MAX + 1.0)));
        }
        if(i > 0){
            for(size_t k = 0;k < i;k++){
                //Test if *b[i] and *b[k] are independent.If not, i = i - 1.
                if( !((*(b[i])).isIndependent((*(b[k])))) ){
                    i--;//Rerandom the vector *b[i];
                    break;
                }
            }
        }
    }
}

/*
*    Function: input
*    ----------------------------------------------------
*    Initialize the basis or change the values in this basis.
*
*/
void basis::input(){
    double d;
    for(size_t i = 0;i < n;i++){
        cout << "Input the "<<i + 1<<"th vector of the basis:"<<endl;
        for(size_t j = 1;j <= m;j++){
            cout<<"Element "<<j<<" : ";
            cin>>d;
            (*(b[i]))[j] = d;
        }
        cout<<endl;
    }
}

/*
*    Function: writeToFile
*    ----------------------------------------------------
*    Write the basis to a fixed file "Basis.txt". The first line contains the
*    dimension (m,n), which enables the program to allocate appropriate
*    memory first when it reads the basis from the file, then add the elements
*    to the current basis.
*
*/
bool basis::writeToFile(){
    try{
        ofstream outfile("Basis.txt");
        if(outfile){
            outfile<<m<<" "<<n<<endl;
            for(size_t i = 0;i < n;i++){
                for(size_t j = 1;j <= m;j++){
                    outfile<<(*b[i])[j]<<" ";
                }
                outfile<<endl;
            }
            outfile.close();
            cout<<"Successfully save -- Basis.txt!"<<endl;
            return true;
        }
        else{
            cout<<"No such file!"<<endl;
        }
    }
    catch(exception e){
        cout<<e.what()<<endl;
    }
    return false;
}

/*
*    Function: readFromFile
*   ----------------------------------------------------
*    Initialize the basis from a fixed file "Basis.txt". The program reads the
*    first line to get dimension in order to check if the dimension of the
*    current basis matches the dimension of the basis in the file, if so, then
*    it will assign these elements to the current basis, otherwise, throw an
*    exception.
*
*/
bool basis::readFromFile(){
    try{
        ifstream infile("Basis.txt");
        if(infile){
            string line;
            getline(infile,line,'\n');
            istringstream stream(line);
            int m1,n1;
            stream>>m1>>n1;
            if(m1!=m||n1!=n){
                cout<<"Failure! The sizes of the basis in the file and this \
basis are not compatible!"<<endl;
                cout<<"The basis in the file is \
                      a "<<m1<<"-by-"<<n1<<"basis."<<endl;
                return false;
            }
            else{
                size_t i = 0;
                while(getline(infile,line,'\n')){
                    stream.clear();
                    stream.str(line);
                    for(size_t j = 1;j <= m;j++){
                        stream>>(*b[i])[j];
                    }
                    i++;
                }
                cout<<"Successfully read -- Basis.txt"<<endl;
                return true;
            }
            infile.close();
        }
        else{
            cout<<"No such file!"<<endl;
        }
    }
    catch(exception e){
        cout<<e.what()<<endl;
    }
    return false;
}

/*
*    Function: writeToFile
*   -------------------------------------------------------
*    Write the basis to a given file.
*
*/
bool basis::writeToFile(string file){
    try{
        ofstream outfile(file.c_str(),ios::out);
        if(outfile){
            outfile<<m<<" "<<n<<endl;
            for(size_t i = 0;i < n;i++){
                for(size_t j = 1;j <= m;j++){
                    outfile<<(*b[i])[j]<<" ";
                }
                outfile<<endl;
            }
            outfile.close();
            cout<<"Successfully save -- "<<file<<endl;
            return true;
        }
        else{
            cout<<"No such file!"<<endl;
        }
    }
    catch(exception e){
        cout<<e.what()<<endl;
    }
    return false;
}

/*
*    Function: readFromFile
*   -------------------------------------------------------
*    Read the basis from a given file.
*
*/
bool basis::readFromFile(string file){
    try{
        ifstream infile(file.c_str());
        if(infile){
            string line;
            getline(infile,line,'\n');
            istringstream stream(line);
            int m1,n1;
            stream>>m1>>n1;
            if(m1 != m || n1 != n){
                cout<<"Failure! The sizes of the basis in the file and this \
basis are not compatible!"<<endl;
                cout<<"The basis in the file \
                      is a "<<m1<<"-by-"<<n1<<" basis."<<endl;
                return false;
            }
            else{
                size_t i = 0;
                while(getline(infile,line,'\n')){
                    stream.clear();
                    stream.str(line);
                    for(size_t j = 1;j <= m;j++){
                        stream>>(*b[i])[j];
                    }
                    i++;
                }
                cout<<"Successfully read -- "<<file<<endl;
                return true;
            }
            infile.close();
        }
        else{
            cout<<"No such file!"<<endl;
        }
    }
    catch(exception e){
        cout<<e.what()<<endl;
    }
    return false;
}




/*
*    LLL.cpp
*
*    The file defines the several necessary functions used in the LLL algorithm.
*
*/

#include "basis.cpp"

/*
*    Function: absd
*   ---------------------------------------------------------
*    Calculate the abstract value of a double value. In the algorithm, the
*    argument will be an element of the matrix U.
```

```
*
*/
float absd(double d){
    return (d >= 0)?d:-d;
}

/*
*    Function: closedInt
*   ----------------------------------------------------
*    Calculate an integer nearest a value.
*
*/
int closedInt(double d){

    int i = (int)(absd(d)+0.5);
    if(d >= 0)
        return i;
    else
        return -i;
}

/*
*    Function: maxValue
*   ----------------------------------------------------
*    Calculate the bigger one of two values.
*
*/
int maxValue(int a,int b){
    return a > b?a:b;
}

/*
*    Function: outputMatrix
*   ----------------------------------------------------
*    Output the values in a n-by-n matrix. In the algorithm, there are two
*    such matrices: U and M. The matrix U, computed from Gram-Schmidt method,
*    is upper triangular with a unit diagonal. M is an integer unimodular
*    transformation matrix, which is computed from the procedure Reduce(i,j).
*
*/
void outputMatrix(double **matrix,int n){

    cout<<endl;
    for(int i = 1;i <= n;i++){
        cout<<"| ";
        for(int j  = 1;j <= n;j++)
            cout<<matrix[i][j]<<" ";
        cout<<"|"<<endl;
    }
    cout<<endl;
}

/*
```

```
*    Function: outputArray
*   ---------------------------------------------------
*    Output the values in array. We use this function to output the values of
*    D, which represents the matrix D computed from the Gram-Schmidt method.
*
*/
void outputArray(double *ar,int n){

    for(int i = 1;i < =n;i++){
        cout <<"D"<<i<<" : "<< ar[i] << " | ";
    }
    cout<<endl;
}

/*
*    Function: initializeDU
*   ---------------------------------------------------
*    Compute the D and U in the decomposition B = Q(D^1/2)U using the
*    Gram-Schmidt method.
*
*/
void initializeDU(double *D,double **U,basis *Q,basis *B,int n){

    for(int i = 1;i <= n;++i){
        (*Q)[i] = (*B)[i];
        for(int j = 1;j < i;++j){
            U[j][i] = ((*B)[i] * (*Q)[j]) / D[j];
            (*Q)[i] = (*Q)[i] - U[j][i] * (*Q)[j];
        }
        U[i][i] = 1;//Unit diagonal
        for(int k = i + 1;k <= n;k++){//Initialize U[i][j] = 0 when i>j
            U[k][i] = 0;
        }
        D[i] = (*Q)[i]*(*Q)[i];
    }
}

/*
*    Function: initializeMatrix
*   ---------------------------------------------------
*    Initialize a matrix to be an identity matrix. In the algorithm, we use this
*    function to initialize the matrix M.
*
*/
void initializeMatrix(double **matrix,int n){

    for(int q = 1;q <= n;q++){
        for(int p = 1;p <= n;p++){
            if(q != p){
                matrix[q][p] = 0;
            }
            else{
                matrix[q][p] = 1;
            }
        }
    }
}

/*
*    Function: Reduce
*    -----------------------------------------------------
*    Procedure Reduce(i,j) used in the LLL algorithm
*
*/
void Reduce(int i,int j,int n,basis *B,double **U,double **M){

    int r = closedInt(U[i][j]);// r is an integer nearest U[i][j]

    /*
    * B <- B * M[i][j] :
    *    Apply M[i][j] to B. M[i][j] = I[n] - r * e[i] * (e[j]~T)
    *    e[i] is the ith unit vector
    */
    (*B)[j] = (*B)[j] - r * (*B)[i];

    //U <- U * M[i][j] : Apply M[i][j] to U
    for(int k = 1;k <= i-1;++k){
        U[k][j] = U[k][j] - r * U[k][i];
    }
    U[i][j] = U[i][j] - r;

    //Update M
    for(int q = 1;q <= n;q++){
        M[q][j] = M[q][i] * (-r) + M[q][j];
    }
}

/*
*    Function: SwapRestore
*    -----------------------------------------------------
*    Procedure SwapRestore(i) used in the LLL algorithm
*
*/
void SwapRestore(int i,basis *B,double **U,double **Z,double *D,int m,int n){

    //Update D
    double d = D[i] + U[i-1][i]*U[i-1][i]*D[i-1];
    D[i] = (D[i-1]*D[i])/d;
    double x = (U[i-1][i]*D[i-1])/d;
    D[i-1] = d;

    //Swap the columns i-1 and i of B
    vector tempB(m);
    tempB = (*B)[i];
    (*B)[i] = (*B)[i-1];
    (*B)[i-1] = tempB;
    //Update M, swap the columns i-1 and i of M
    double m;
    for(int q = 1;q <= n;q++){
        m = M[q][i-1];
        M[q][i-1] = M[q][i];
        M[q][i] = m;
    }

    //Swap the columns i-1 and i of U
    double tempU;
    for(int j = 1;j<=i-2;j++){
        tempU = U[j][i-1];
        U[j][i-1] = U[j][i];
        U[j][i] = tempU;
    }

    //Apply X[i]~(-1) to U
    double u = U[i-1][i];
    for(int k = i+1;k<=n;k++){
        double u1 = U[i-1][k];
        double u2 = U[i][k];
        U[i-1][k] = u1 * x + (1 - x * u) * u2;
        U[i][k] = u1 - u*u2;
    }
    U[i-1][i] = x;
}

/*
*    Function:  ReduceSwapRestore
*    ---------------------------------------------------
*    ReduceSwapRestore(i,r) in the LLL algorithm with delayed size-reduction.
*
*/
void ReduceSwapRestore(int i,basis *B,double **U,double **Z,double *D,int m,int n,int r){

    //Update D
    double d = D[i] + (U[i-1][i]-r)*(U[i-1][i]-r)*D[i-1];
    D[i] = (D[i-1]*D[i])/d;
    double x = ((U[i-1][i]-r)*D[i-1])/d;
    D[i-1] = d;

    //Update the columns i-1 and i of B
    vector tempB(m);
    tempB = (*B)[i-1];
    (*B)[i-1] = (*B)[i] - r * ((*B)[i-1]);
    (*B)[i] = tempB;

    //Update the columns i-1 and i of M
    double m;
    for(int q = 1;q <= n;q++){
        m = M[q][i-1];
        M[q][i-1] = M[q][i] - r * M[q][i-1];
```

```
            M[q][i] = m;
        }

        //Update the columns i-1 and i of U
        double tempU;
        for(int j = 1;j <= i - 2;j++){
            tempU = U[j][i-1];
            U[j][i-1] = U[j][i] - r * U[j][i-1];
            U[j][i] = tempU;
        }

        //Apply X[i]~(-1) to U
        double u = U[i-1][i];
        for(int k = i + 1;k <= n;k++){
            double u1 = U[i-1][k];
            double u2 = U[i][k];
            U[i-1][k] = u1 * x + (1 - x * u + r * x) * u2;
            U[i][k] = u1 + (r- u) * u2;
        }
        U[i-1][i] = x;
    }

    /*
    *    Function:  verifyConditions
    *   -------------------------------------------------------
    *    Verify whether a basis satisfies the conditions defined by the LLL algorithm.
    *
    */
    void verifyConditions(basis *a, basis* b,int m,int n,double w,double **z){

        //Another Q, used in verification process
        basis *q = new basis(m,n);

        //Another D, used in verification process
        double *d = new double[n+1];

        //Another U, used in verification process
        double **u = new double*[n+1];
        for(int i = 0;i <= n;++i){
            u[i] = new double[n+1];
        }

        /*
        * According to the result basis B , initialize the values of d, u and q
        * by Gram-Schmdit process.
        */
        initializeDU(d,u,q,b,n);

        cout<<endl;
        cout<<"--------------------  Verify Conditions  -------------------"<<endl;
        cout<<endl;

        /*
```

```
* Output U to justify the size-reduced condition.If it satisfies this
* condtion, then all values in upper triangular should be bigger than
* or equal to 0.5.
*/
cout<<"Verify Size Reduced"<<endl;
cout<<"-------------------"<<endl;
cout<<endl;
cout<<"The values in U are:"<<endl;
outputMatrix(u,n);
bool sr = true;
for(int j = 1;j <= n;j++){
    for(int i = 1;i < j;i++){
        if(u[i][j] > 0.5){
            sr = false;
            cout << "Mistake : U["<<i<<"]["<<j<<"] > 0.5"<<endl;
        }
    }
}
if(sr){
    cout << "Yes, all |U[i][j]|<=0.5, the size \
            reduced condition is satisfied!"<<endl;
}
else{
    cout << "Sorry, the size reduced condition is not satisfied!"<<endl;
}
cout<<endl;
cout<<"-------------------"<<endl;
cout<<endl;

/*
* For k = 2,3,...,n, verify the loose increasing condition.If it satisfies
* this condition:
* When k = i, then (d[i] + d[i-1] * U[i-1][i] * U[i-1][i]) >= w * d[i-1],
* it outputs "Yes", otherwise, "Not".
*/
cout<<"Verify Loose Increasing"<<endl;
cout<<"-----------------------"<<endl;
bool sv = true;
for(int k = 2;k < =n;k++){
    cout<< "i = "<<k<<": ";
    if( d[k] < (w - u[k-1][k] * u[k-1][k]) * d[k-1] ){
        sv = false;
        cout<<"(d[i]+d[i-1]*U[i-1][i]*U[i-1][i]) < w*d[i-1]. NOT"<<endl;
    }
    else
    {
        cout<<"(d[i]+d[i-1]*U[i-1][i]*U[i-1][i]) >= w*d[i-1]. YES"<<endl;
    }
}
if(sv){
    cout << "Yes, all (d[i] + d[i-1]*U[i-1][i]*U[i-1][i]) >= w*d[i-1], \
            this condition is satisfied!"<<endl;
}
```

```
    else{
        cout << "Sorry, the condition is not satisfied!"<<endl;
    }
    cout<<"-----------------------"<<endl;
    cout<<endl;

    /*
    * Verify if the result basis equals the product of the original basis A
    *  and Z.
    */
    cout<<"Verify if B = AM"<<endl;
    cout<<"-----------------------------------"<<endl;
    cout<<"Now A is:"<<endl;
    cout<<*a<<endl;
    cout<<"Now M is:"<<endl;
    outputMatrix(m,n);
    cout<<"So AM is:"<<endl;
    bool baz = true;
    double sum;
    for(int i = 1;i <= n;i++){
        cout<<"| ";
        for(int j = 1;j <= m;j++){
            sum = 0;
            for(int k = 1;k <= n;k++){
                sum = sum + (*a)[k][i] * z[k][j];
            }
            cout<<sum<<" ";
            if(sum != (*b)[j][i]){
                cout << "Error ";
                baz = false;
            }
        }
        cout<<"|"<<endl;
    }
    cout<<endl;
    if(baz){
        cout<<"Congratulations! B = AM."<<endl;
    }
    else{
        cout<<"Sorry! B != AM."<<endl;
        cout<<"The result basis B is:"<<endl;
        cout<<*b<<endl;
    }
    cout<<"-----------------------------------"<<endl;
    cout<<endl;
    cout<<"-------------------------------------------------------------"<<endl;
    cout<<endl;

    delete q;
    delete [] d;
    delete [] u;
}
```

```
/*
*    parallelLLLRL.cpp
*
*    Implementation of the parallel LLL algorithm with Pthread.
*    The "Swap" part is implemented by using a thread pool.
*    The "Reduce" part is implemented by using a reduction list.
*    We use the same number of threads in these two parts.
*
*/

#include <pthread.h>
#include <sys/time.h>
#include <sys/wait.h>
#include "LLL.cpp"

//Number of threads
const int NUM_THREADS = 4;

//Record the number of threads that enter the waiting state
static int joinCount = 1;

/*
* Used to record the times that the procedure ReduceSwapRestore has been
* called in each thread.
*/
int swapCount[NUM_THREADS];

//Used to record the times that the procedure Reduce has been called
int reduceCount = 0;

/* ----------- Pthread Varible definitions for the "Swap" part ------------- */
/* ------------------------------------------------------------------------- */

//The mutex lock for the swap procedures
pthread_mutex_t swap_count_mutex = PTHREAD_MUTEX_INITIALIZER;

/*
* Condition variable used by some thread to notify other threads of the ending
* of the current iteration of the while loop in the "Swap" part.
*/
pthread_cond_t swap_loop_end = PTHREAD_COND_INITIALIZER;

//This structure is used to pass parameters to swap worker threads
struct work_swap_t {
    int offset;
    int m;
    int n;
    basis *B;
    double **U;
    double **Z;
    double *D;
    double w;
};

/*
* The global variable finish is used to determine whether the while loop should
* be ended. If the procedure ReduceSwapRestore is called in an iteration of the
* while loop, the variable is set to be false. It is set to be global beacuse
* all worker threads need to check the variable.
*/
static bool finish = false;

/*
* In order to avoid the race condition that the workers write a value to the
* global variable finish, we use an individual variable subfinish for each
* thread instead.
*/
static bool subfinish[NUM_THREADS];

/* -------------------------------------------------------------------------- */
/* -------------------------------------------------------------------------- */


/* ----------- Pthread Varible definitions for the "Reduce" part ----------- */
/* -------------------------------------------------------------------------- */

//The mutex lock for the reduce procedures
pthread_mutex_t reduce_mutex = PTHREAD_MUTEX_INITIALIZER;

/*
* Condition variable used by the reducing thread to notify the main thread of
* the ending of reducing process.
*/
pthread_cond_t list_empty = PTHREAD_COND_INITIALIZER;

/*
* Condition variable used by the main thread to broadcast
* the ending of computing the list to reducing threads.
*/
pthread_cond_t list_not_empty = PTHREAD_COND_INITIALIZER;

/*
* When the main thread ends, the varable is set to be true
* to notify other reducing threads of the ending.
*/
bool end = false;

//This structure is used to pass parameters to reducing worker threads
struct work_reduce_t {
    int id;
    int n;
    basis *B;
    double **U;
    double **M;
};
```

```
/*
* This structure is used to record the positions that need
* to be reduced in the matrix U.
*/
struct node {
    int i;
    int j;
    node *next;
};

//linklist is used to construct the list
node* linklist = NULL;

/*
* head points to the first node of the list, which will be used to delete this
* list at the end.
*/
node* head = NULL;

//A pointer that points to the head of the positions that have not been reduced
node* current = NULL;

/* ------------------------------------------------------------------------- */
/* ------------------------------------------------------------------------- */


/*
*    Function: subSwap
*    -----------------------------------------------------
*    This function is executed by each swap worker thread. Each thread handles
*    a different group of the values of k in the two for routines.
*
*/
void *subSwap(void* w_swap) {

    //Cast void* to work_swap_t*
    work_swap_t* work = (work_swap_t*)w_swap;

    int r, k;

    //Initialize the count variable for the current thread.
    swapCount[work->offset] = 0;

    while(!finish) {

        //At the beginning, ReduceSwapRestore has not been called.
        subfinish[work->offset] = true;

        //The first for routine
        for(k = 2 + 2 * (work->offset);k <= work->n;k = k + 2 * (NUM_THREADS)){
            r = closedInt((work->U)[k-1][k]);
            if((work->D)[k] < (work->w - ((work->U)[k-1][k] - r) *
                ((work->U)[k-1][k] - r)) * (work->D)[k-1] ){
                subfinish[work->offset] = false;
                ReduceSwapRestore(k,work->B,work->U,work->Z,work->D,work->m,
                work->n,r);
                //Increment the count variable by 1.
                swapCount[work->offset]++;
            }
        }

        /*
        * If there is one or more threads that still have not finished executing
        * the first for routine, the current thread will wait other threads to
        * finish their work, otherwise it will notify other threads of the
        * ending of the first for routine.
        */
        pthread_mutex_lock(&swap_count_mutex);
        if(joinCount < NUM_THREADS){
            joinCount++;
            pthread_cond_wait(&swap_loop_end, &swap_count_mutex);
        }
        else{
            joinCount = 1;
            pthread_cond_broadcast(&swap_loop_end);
        }
        pthread_mutex_unlock(&swap_count_mutex);
        //The second for routine
        for(k = 3 + 2 * (work->offset);k <= work->n;k = k + 2 * (NUM_THREADS)){
            r = closedInt((work->U)[k-1][k]);
            if((work->D)[k] < (work->w - ((work->U)[k-1][k] - r) *
                ((work->U)[k-1][k] - r)) * (work->D)[k-1] ){
                subfinish[work->offset] = false;
                ReduceSwapRestore(k,work->B,work->U,work->Z,work->D,work->m,
                work->n,r);
                swapCount[work->offset]++;
            }
        }

        /*
        * If there is one or more threads that still have not finished
        * executing the second for routine, the current thread will wait other
        * threads to finish their work, otherwise it will check if
        * ReduceSwapRestore has been called in this while iteration by checking
        * the values of the variable subfinish in all threads.Also, in will
        * notify other threads of the ending of the second for routine.
        */
        pthread_mutex_lock(&swap_count_mutex);
        if(joinCount < NUM_THREADS){
            joinCount++;
            pthread_cond_wait(&swap_loop_end, &swap_count_mutex);
        }
        else{
            finish = true;
            for(int i = 0;i < NUM_THREADS; i++){
```

```
                if(subfinish[i] == false) {
                    finish = false;
                    break;
                }
            }
            joinCount = 1;
            pthread_cond_broadcast(&swap_loop_end);
        }
        pthread_mutex_unlock(&swap_count_mutex);
    }
}

/*
*    Function: subReduce
*    ----------------------------------------------------
*    This function is executed by each reducing worker thread. It checks the
*    list calculated by the main thread and reduce the elements in the list
*    if it is not empty. When some threads find that the list is empty, they
*    enter the waiting state. Before entering the waiting state, the last
*    reduce thread notifies the main thread they finish the reducing work.
*
*/
void *subReduce(void *w_reduce) {

    //Cast void* to work_reduce_t*
    work_reduce_t *work = (work_reduce_t*)w_reduce;

    int i, j;
    bool run = false;

    while(!end){

        /*
        * When the main thread releases the reduce lock, the worker threads
        * on the waiting state will obtain this lock orderly
        */
        pthread_mutex_lock(&reduce_mutex);

        /*
        * If the thread finds all nodes have been reduced, it will signal our
        * list_empty condition variable to wake up the boss thread, otherwise,
        * it will set run to be true and fetch the current node to get the
        * values of i and j, then relase the reduce lock and let other worker
        * threads to check the list. After it gets the position information and
        * release the lock, it will then reduce this position.
        */
        if(current == NULL){
            if(joinCount < NUM_THREADS){
                joinCount++;
                pthread_cond_wait(&list_not_empty, &reduce_mutex);
            }
            else {
                joinCount = 1;
                pthread_cond_signal(&list_empty);
                pthread_cond_wait(&list_not_empty, &reduce_mutex);
            }
        }
        else {
            run = true;
            i = current->i;
            j = current->j;
            current = current->next;
        }
        pthread_mutex_unlock(&reduce_mutex);
        if(run){
            run = false;
            Reduce(i,j,work->n, work->B, work->U, work->M);
        }
    }
}

/*
*    Function: parallelLLLRL
*    -------------------------------------------------------
*    Main process to calculate the parallel algorithm. In this function,
*    the "Reduce" part is implemented by using a reduction list.
*
*/
void parallelLLLRL(const int m, const int n, basis *A, double w){

    //B is used to substitute A so that the original basis can be saved
    basis *B = new basis(m,n);
    for(int i = 1;i <= n;i++){
        (*B)[i] = (*A)[i];
    }

    /*
    * Q is used to save the result of the orthogonal basis computed by
    * Gram-Schmdit process.
    */
    basis *Q = new basis(m,n);

    //D = diag(d1,d2,...,dn) is the positive diagonal.D[i] = Q[i]*Q[i].
    double *D = new double[n+1];

    /*
    * U is an array n+1 elements, each element consists
    * of n+1 doubles, which equals 2-dimensional array.
    * U is upper triangular with unit diagonal. The first
    * row and the first coloumn will not be used, they are
    * used for reader to identify the index of U.
    */
    double **U = new double*[n+1];
    for(int i = 0;i <= n;i++){
        U[i] = new double[n+1];
    }
```

```
/*
* M is an array n+1 elements, each element consists
* of n+1 doubles, which equals 2-dimensional array.
* The first row and the first coloumn will not be
* used, they are used for reader to identify the
* index of M.
*/
double **M = new double*[n+1];
for(int i = 0;i <= n;i++){
    M[i] = new double[n+1];
}

/*
* According to the basis B, initialize the values of D, U and Q by
* Gram-Schmdit process.
*/
initializeDU(D,U,Q,B,n);

//It initializes M to identity matrix.
initializeMatrix(M,n);

int r;
int k;

/*
* timeval variables are used to record the time points in the algorithm
* tv1: the time point when the "Swap" part begins.
* tvm: the time point when the "Swap" part ends, and at the same time,
*      the "Reduce" part begins.
* tv2: the time point when the "Reduce" part ends.
*/
timeval tv1,tv2,tvm;

//Record the beginning time point of the "Swap" part
gettimeofday(&tv1,NULL);

/* -------------------- The "Swap" part ------------------------ */

//Initialize the worker threads for the "Swap" part
pthread_t swap_thread[NUM_THREADS];

//Each swap worker thread needs a distinct work_swap_t structure
work_swap_t *w_swap[NUM_THREADS];

//Compute the work_swap_t for each thread and create each worker thread
for(int i = 0;i < NUM_THREADS;i++) {
    w_swap[i] = new work_swap_t;
    w_swap[i]->offset = i;
    w_swap[i]->m = m;
    w_swap[i]->n = n;
    w_swap[i]->B = B;
    w_swap[i]->U = U;
    w_swap[i]->Z = Z;
    w_swap[i]->D = D;
    w_swap[i]->w = w;
    pthread_create(&swap_thread[i], NULL, subSwap, (void*)w_swap[i]);
}

//Join all worker threads for the "Swap" part.
for(int i = 0;i < NUM_THREADS;i++){
    pthread_join(swap_thread[i],NULL);
}

//Record the ending time point of the "Swap" part
gettimeofday(&tvm,NULL);

/* -------------------- The "Reduce" part ---------------------- */

//The main thread obtains the reduce lock at first
pthread_mutex_lock(&reduce_mutex);

int i,j,start;

//Initialize the list
linklist = new node;
head = linklist;

//Initialize the worker threads for the "Reduce" part
pthread_t reduce_t[NUM_THREADS];

//Each reducing worker thread needs a distinct work_reduce_t structure
work_reduce_t *w_reduce[NUM_THREADS];

//Compute the work_reduce_t for each thread and create each worker thread
for(int i = 0; i < NUM_THREADS; i++){
    w_reduce[i] = new work_reduce_t;
    w_reduce[i] -> id = i;
    w_reduce[i] -> n = n;
    w_reduce[i] -> B = B;
    w_reduce[i] -> U = U;
    w_reduce[i] -> M = M;
    pthread_create(&reduce_t[i], NULL, subReduce, (void*)w_reduce[i]);
}

for(k = 2 * n-3; k >= 1; k--){
    if( k <= n - 1 ){
        start = 1;
    }
    else{
        start = k - n + 2;
    }
    for(i = start;i < (k + 3)/2;i++){

        //i + j always equals k + 2
        j = k + 2 - i;
        /*
        * If the program finds a position which needs to be reduced, it
        * will create a node and add this node to the tail of the list.
        * If this new node is the first node added into the list, then
        * current will be set to point to it.
        */
        if(absd(U[i][j])>0.5){
            linklist -> next = new node;
            linklist = linklist -> next;
            linklist -> i = i;
            linklist -> j = j;
            linklist -> next = NULL;
            if (current == NULL) {
                current = linklist;
            }

            /*
            * Here, we know the position should be reduced, so we increment
            * reduceCount by 1.
            */
            reduceCount++;
        }
    }

    /*
    * If there is(are) a(some) node added into the list in the current
    * iteration, the boss thread will wake up other threads, then sleep
    * and wait for the worker threads to reduce these positions on the
    * list.
    */
    if(current != NULL) {
        pthread_cond_broadcast(&list_not_empty);
        pthread_cond_wait(&list_empty, &reduce_mutex);
    }
}
end = true;
pthread_cond_broadcast(&list_not_empty);

//The main thread releases the reduce lock at the end
pthread_mutex_unlock(&reduce_mutex);

//Join the worker threads for the "Reduce" part
for (int i = 0;i < NUM_THREADS;i++){
    pthread_join(reduce_t[i],NULL);
}

//Free all memories that have been used for the list
node* previous;
while (head != NULL){
    previous = head;
    head = head->next;
    delete previous;
```

```
}

//Record the ending time point of the "Reduce" part
gettimeofday(&tv2,NULL);

//Output the result basis when the dimension is small
if(m <= 20 && n <= 20){
    cout<<"The reduced basis is:"<<endl;
    cout<<*B<<endl;
}
else{
    cout<<"The result is too large to display!"<<endl;
}

double sumSwapCount = 0;
for(int i = 0;i < NUM_THREADS;i++){
    sumSwapCount = sumSwapCount + swapCount[i];
}

//Output the costs and counts of the "Swap" part and the "Reduce" part
cout<<"Swap "<<sumSwapCount<<" times, Reduce "<<reduceCount<<" times."<<endl;
cout<<"It takes \
      "<<(tv2.tv_usec-tv1.tv_usec)+1000000*(tv2.tv_sec-tv1.tv_sec)<<"\
      microsecond(s) to reduce this basis!"<<endl;
cout<<"Swap cost:\
      "<<(tvm.tv_usec-tv1.tv_usec)+1000000*(tvm.tv_sec-tv1.tv_sec)<<"\
      microsecond(s)!"<<endl;
cout<<"Reduce cost:\
      "<<(tv2.tv_usec-tvm.tv_usec)+1000000*(tv2.tv_sec-tvm.tv_sec)<<"\
      microsecond(s)!"<<endl;

//Verify the result basis
char ch;
cout<<endl;
cout<<"Do you want to verify the reuslt?[Y/N]"<<endl;
cin>>ch;
while (!(ch=='Y'||ch=='y'||ch=='N'||ch=='n')){
    cout<<"Please input Y or N:"<<endl;
    cin>>ch;
}
if(ch=='Y'||ch=='y'){
    //Verify if the result satisfies the conditions of the LLL algorithm
    verifyConditions(A,B,m,n,w,M);
}

//Free all memories used for the matrices in the algorithm
delete Q;
for(int i=0;i<=n;++i){
    delete [] M[i];
}
delete [] M;
for(int i=0;i<=n;++i){
    delete [] U[i];
    }
    delete [] U;
    delete [] D;
}




/*
*    test.cpp
*
*    A simple test file for the parallel algorithm.
*
*/
void main(){
    cout<<"************************************************************"<<endl;
    cout<<"*                                                          *"<<endl;
    cout<<"*           This program is used to implement              *"<<endl;
    cout<<"*              the parallel LLL algorithm                  *"<<endl;
    cout<<"*                                                          *"<<endl;
    cout<<"************************************************************"<<endl;
    cout<<endl;

    int n,m;
    double w;

    while(true){
        cout<<"Please input the number of vectors:"<<endl;
        cin>>n;
        cout<<"Please input the number of elements in each vector:"<<endl;
        cin>>m;
        cout<<"Please input the parameter:"<<endl;
        cin>>w;
        try{
            if(w<=0.25||w>=1)
                throw logic_error("Error,constructing with wrong w in \
LLL(const size_t,const size_t,double)!");
            else if(m>0&&n>0&&m>=n){
                break;
            }
            else if(m<n)
                throw logic_error("Error,can't execute the LLL algroithm \
when m<n!");
            else
                throw logic_error("Error,constructing with wrong sizes in\
LLL(const size_t,const size_t,double)!");
        }
        catch(logic_error l){
            cout<<endl;
            cout<<l.what()<<endl;
            cout<<"Please reinput these values again"<<endl;
            cout<<endl;
        }
    }
    basis *A = new basis(m,n);

    char ch;

    cout<<"Which method do you want to generate the matrix, random, input,\
or read from a file? [R/I/F]"<<endl;
    cin >> ch;

    while(!(ch == 'R' || ch == 'r' || ch == 'I' || ch == 'i' || ch == 'F' ||
        ch == 'f')){
        cout<<"Please input R, I or F:"<<endl;
        cin>>ch;
    }

    bool file = true, rf = false;

    if(ch == 'R' || ch == 'r'){//Initialize a random generate matrix.
        (*A).randomBasis();
    }
    else if(ch == 'I'|| ch == 'i'){//Initialize the generate matrix by hand.
        bool isInde;
        while(true){
            (*A).input();
            isInde = true;
            for(int i = 1;i<n;i++){
                for(int k=i+1;k<=n;k++){
                    //Test if *b[i] and *b[k] are independent.
                    if( !((*A)[i].isIndependent((*A)[k])) ){
                        isInde = false;
                    }
                }
            }
            if(isInde){
                break;
            }
            else{
                cout<<"Error:these vectors in this matrix are not linear\
independent, please input the a right basis:"<<endl;
                cout<<endl;
            }
        }
    }
    else if(ch == 'F' || ch == 'f'){
        file = (*A).readFromFile();
        rf = true;
    }

    if(file && m<= 20 && n <= 20){
        cout<<endl;
        cout<<"The generate matrix is:"<<endl;
        cout<<(*A)<<endl;
        cout<<endl;
    }
    else if(file&&(m>20||n>20)){
        cout<<"The generate matrix is too large to display!"<<endl;
    }

    if(!rf){
        cout<<"Save the basis for using in the next time?"<<endl;
        cin>>ch;
        if(ch == 'Y' || ch = ='y'){
            (*A).writeToFile();
        }
    }

    if(file){
        //Execute the parallel LLL algorithm
        parallelLLLRL(m,n,A,w);
    }

    delete A;
}
```

References

[1] David Poole. The Gram-Schmidt Process and the QR Factorization. *LINEAR ALGEBRA: A MODERN INTRODUCTION*, 2003, 375-376

[2] Franklin T. Luk, Sanzheng Qiao, Wen Zhang. *A Lattice Basis Reduction Algorithm.* 2010.

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

[4] Wen Zhang, Yimin Wei, Sanzheng Qiao. *The LLL Algorithm Using Fast Givens.* 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] G.H. Golub and C.F. Van Loan. *Matrix Computations, Third Edition.* The Johns Hopkins University Press, Baltimore, MD, 1996.

[7] Wen Zhang. *LLL algorithm with delayed size reduction.* 2010. Personal Communication.

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

[9] Blaise Barney. *POSIX Threads Programming.* Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/pthreads/