Parallelizing Principal Component Analysis Algorithm_

CSE4001: PARALLEL AND DISTRIBUTED COMPUTING

REVIEW -3

Submitted By:
Group-23
Harshith Chukka(18BCE0941)

Mrinmay Date (18BIS0147)

Henil Jayesh Thakor(18BCE0934)

In partial fulfillment for the award of the degree of
B.Tech
in
Computer Science and Engineering
Under the guidance
of
Prof. Deebak B D
Associate Professor Grade 1, SCOPE

Fall Semester 2020

Table of Content

Objective

To parallelize Principal Component Analysis (PCA) Algorithm and compare its performance with the serial code.

1. Problem addressed

Finding Eigenvalues and Eigenvectors, in-cases where covariance matrix is vary large.

2. Prior Research

- **Techniques and its related challenges**

While using PCA algorithm, we need to calculate eigenvalues and eigenvectors. In cases where covariance matrix is very large, we can't simply solve the polynomial equations. This is where parallelization will help as it will reduce the time taken for computation. To calculate eigenvalues through a C program, we need QR algorithm. While parallelizing PCA algorithm in this project, we will calculate eigenvalues and eigenvectors using QR algorithm which in turn uses QR Factorization. QR factorization is found using Gram Schmidt. The end goal of this project is to compare the performance of the serial and parallel code using performance parameters.

3. Significance

Principal Component Analysis (PCA) is a tool for reducing the dimensionality of such datasets, enhancing interpretability but minimising the loss of information at the same time. By generating new uncorrelated variables that increase variance successively, it does so. The discovery of such new variables, the key components, reduces the problem of own value / eigenvector to solve, and the new variables are described by the dataset at hand, not a priori, making PCA an adaptive technique for data analysis. It is also adaptive in another way, as variations of the approach have been generated that are adapted to various types and structures of data.

4. Introduction, Literature Review and Methodology:

I. Introduction

Principal Component Analysis(PCA) uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of **linearly uncorrelated variables** called principal components. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components.

II. Literature Review

i. Abdi, H. and Williams, L. J. (2010), Principal component analysis. WIREs Comp Stat, 2: 433- 459. doi:10.1002/wics.101

Principal component analysis (PCA) is a multivariate technique that analyzes a data table in which observations are described by several inter-correlated quantitative dependent variables. Its goal is to extract the important information from the table, to represent it as a set of new orthogonal variables called principal components, and to display the pattern of similarity of the observations and of the variables as points in maps. The quality of the PCA model can be evaluated using cross-validation techniques such as the bootstrap and the jackknife. PCA can be generalized as correspondence analysis (CA) in order to handle qualitative variables and as multiple factor analysis (MFA) in order to handle heterogeneous sets of variables. Mathematically, PCA depends upon the eigen- decomposition of positive semi-definite matrices and upon the singular value decomposition (SVD) of rectangular matrices. The paper uses statistical and graphical analysis for data analysis using PCA. It helped us understand the working and implementation of PCA on large data.

ii. Hess, Aaron S. and John R. Hess. “Principal component analysis.” Transfusion 58 7 (2018): 1580-1582.

PCA is an old statistical technique for identifying major relationships in complex data. PCA consists of creating artificial variables (“components”) optimized to maximize how much variation is explained in a data set. It is widely used in exploratory data analysis and predictive modeling. Its uses can be as simple as

deconvoluting spectra to determine the concentrations of the individual chemicals in a mixture or as complex as trying to define the elements of a perfect IQ test. The paper uses PCA in various applications to help analyze experiment results. It helped us understand the application of PCA in practical experiments.

iii. Ian T. Jolliffe and Jorge Cadima, "Principal component analysis: a review and recent developments"

In this article, the working of the PCA has been discussed and the significance of the PCA is also explained. It is described how large datasets are represented with low dimensionality. It has been mentioned that there are many ways to adapt PCA inorder to achieve our desired result. For ex: Functional principal component analysis for chemical spectroscopy, Simplified Principal Components, Robust Principal Component Analysis and Symbolic data principal component analysis for more complex data structures.

III. Methodology

1. We use OpenMp parallel loops to convert the algorithm code to parallel.

2. For PCA, first we find the correlation matrix of the given data set using statistical tools like mean and covariance. The data will be given in a mxn matrix. From that we will obtain a nxn correlation matrix.

3. Then to find eigen values, we will use QR Algorithm.

4. QR algorithm includes QR Factorization, iterated multiple times to get values that are closer to the actual eigen values.

5. For QR factorization we first use Gram-Schmidt Process which converts the nxn correlation matrix into n unit vectors.

6. Using QR factorization, we get the equation A=RQ. We have A, the correlation matrix. We get Q from Gram-Schmidt process. We find R which is an upper triangular matrix, where the diagonal elements represent the eigen values.

7. We repeat the process to get eigenvalues that are closer to the actual calculation.

a. To Calculate Correlation Matrix

The covariance of two variables (x and y) can be represented as cov(x,y). If E[x] is the expected value or mean of a sample 'x', then cov(x,y) can be represented in the following way:

$$Cov(x,y) = E(xy) - E(x)E(y)$$

$$Var(X) = \Sigma (Xi-X)^2/N = \Sigma x_i^2/N$$

Correlation Matrix:

$$V = \begin{bmatrix} \Sigma x_1^2/N & \Sigma x_1 x_2/N & \dots & \Sigma x_1 x_c/N \\ \Sigma x_2 x_1/N & \Sigma x_2^2/N & \dots & \Sigma x_2 x_c/N \\ \dots & \dots & \dots & \dots \\ \Sigma x_c x_1/N & \Sigma x_c x_2/N & \dots & \Sigma x_c^2/N \end{bmatrix}$$

b. Gram-Schimdt Process To Find Unit Vectors

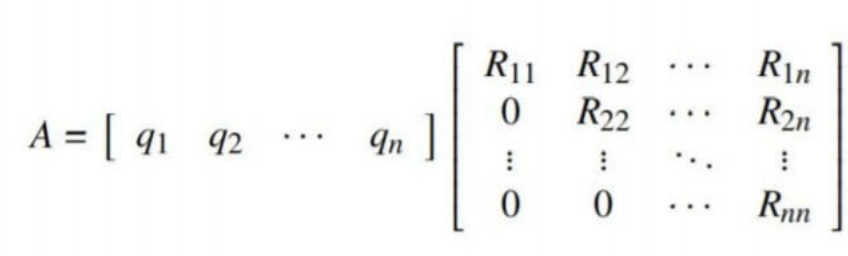
V is an inner product space and {v1,v2,...,vn} is a set of linearly independent vectors in V, then we can construct a set of orthonormal vectors {e1,e2,...,en} of V where:

$$e_1 = \frac{v_1}{\|v_1\|}, \quad e_j = \frac{v_j - \left[< v_j, e_1 > e_1 + < v_j, e_2 > e_2 + \dots + < v_j, e_{j-1} > e_{j-1} >\right]}{\left\|v_j - \left[< v_j, e_1 > e_1 + < v_j, e_2 > e_2 + \dots + < v_j, e_{j-1} > e_{j-1} >\right]\right\|}, j = 1, 2, \dots, n$$

e1, e2, e3, ..., en are **unit vectors.**

c. QR Factorization To Find R

If A ∈ R m×n has linearly independent columns then it can be factored as A=QR



$$A = \begin{bmatrix} q_1 & q_2 & \cdots & q_n \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1n} \\ 0 & R_{22} & \cdots & R_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{nn} \end{bmatrix}$$

Where q1,q2,q3,….,qn are orthonormal unit vectors found using Gram-Schmidt. We have to find R, which is an upper triangular matrix.

$Q^T Q = I$

$R = Q^T A$

$A = RQ$

d. QR Algorithm To Find Eigenvalues

Using A=RQ, we repeat Gram-Schmidt and QR factorization till we get approx. value of eigenvalues.

We get a diagonal matrix, where diagonal elements are eigenvalues of the data and the remaining elements are equal to zero or tend to be equal to zero(depending on the number of iterations)

5. Contributions

Harshith Chukka: [1] Abdi, H. and Williams, L. J. (2010), Principal component analysis. WIREs Comp Stat, 2: 433- 459. doi:10.1002/wics.101, proposed methodology.

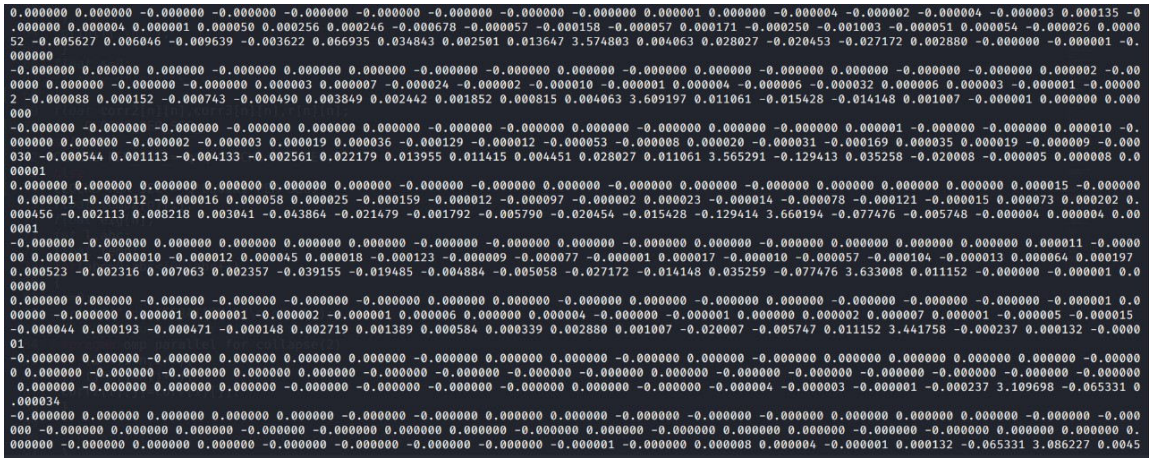
[3] Ian T. Jolliffe and Jorge Cadima, “Principal component analysis: a review and recent developments", 2016

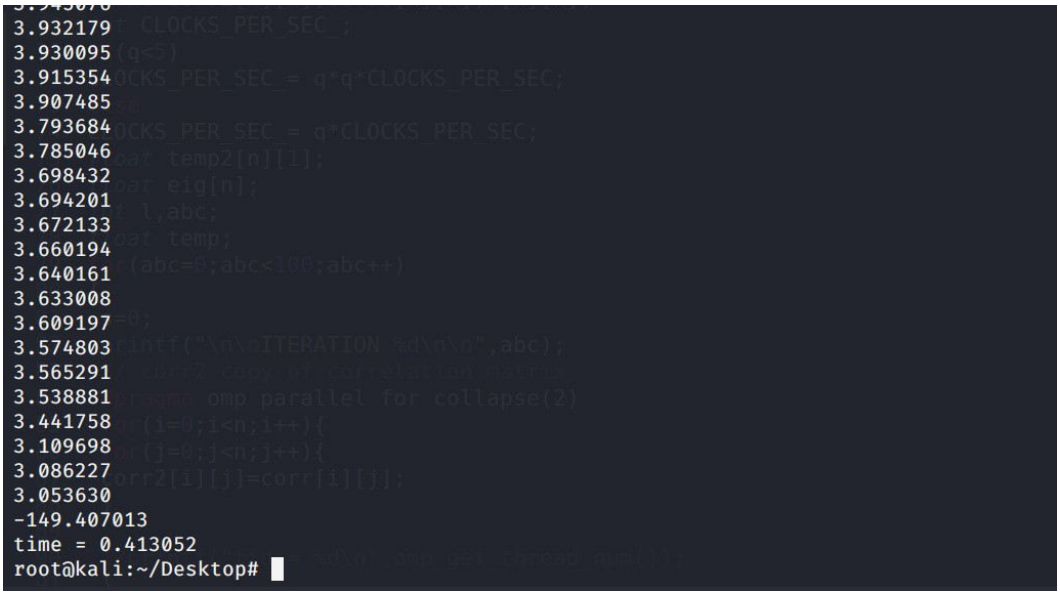
Henil Jayesh Thakor: [2] Hess, Aaron S. and John R. Hess. “Principal component analysis.” Transfusion 58 7 (2018): 1580-1582, proposed methodology.

[3] Ian T. Jolliffe and Jorge Cadima, “Principal component analysis: a review and recent developments", 2016

6. Results and Discussion

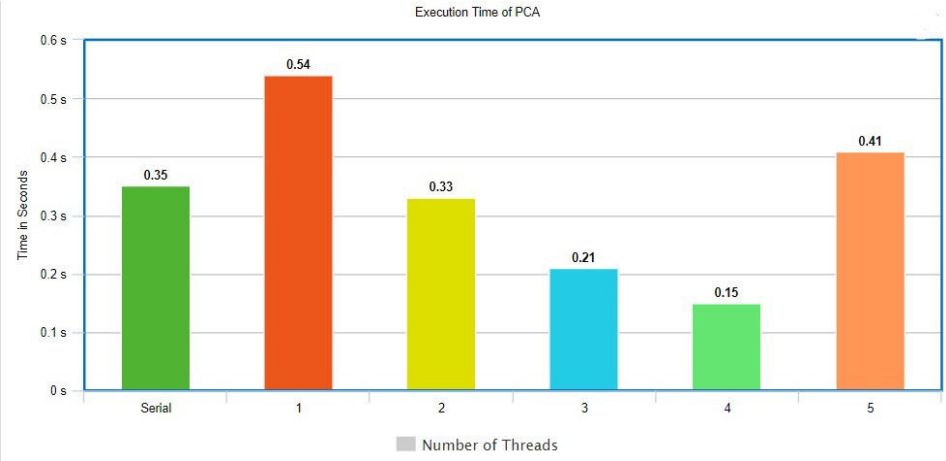
I. Output





II. Result

Serial					
Time in Seconds	0.35				
Parallel					
No. of Threads	1	2	3	4	5
Time in Seconds	0.54	0.33	0.21	0.15	0.41
Speedup	0.64	1.06	1.66	2.33	0.85
Efficiency	0.64	0.53	0.55	0.58	0.17



III. Conclusion

From the following graphs and performance measures we can see that that Amdahl's Law is followed for the the parallel program we have written. The speedup initially increases then decreases after a maximum theoretical speed is achieved.

7. Further research:

We can further try to implement this modified algorithm completely in some sort of application in-order to record practical output

8. References

1. Abdi, H. and Williams, L. J. (2010), **Principal component analysis. WIREs Comp Stat, 2: 433- 459. doi:10.1002/wics.101**

2. Hess, Aaron S. and John R. Hess. "**Principal component analysis.**" Transfusion 58 7 (2018): 1580-1582

3. Ian T. Jolliffe and Jorge Cadima, "**Principal component analysis: a review and recent developments**", 2016

9. Appendix

- Sequential Code

```
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<stdlib.h>
void main(){
 int m=1000,n=500;
 int data[m][n];
 //int data[10][3]={{7,4,3},{4,1,8},{6,3,5},{8,6,1},{8,5,7},{7,2,9},
//{5,3,3},{9,5,8},{7,4,5},{8,2,2}};
 /*
 correlation matrix
 (E[xy] - E[x]E[y])/S(x)*S(y)
 */
 int i,j,k;
 clock_t start,stop;
 float corr[n][n],exy=0,ex=0,ey=0,sx=0,sy=0;
 for(i=0;i<m;i++){
 for(j=0;j<n;j++){
 data[i][j]=rand();
 }
 }
 start=clock();
 for (i=0;i<n;i++){
 for(j=i;j<n;j++){
// printf("(%d,%d) ",i,j);
 exy=0,ex=0,ey=0,sx=0,sy=0;
 //correlation between i and j column
 if (i==j){
 corr[i][j]=1;
 }
 else{
 for (k=0;k<m;k++){
 exy+=data[k][i]*data[k][j];
 ex+=data[k][i];
 ey+=data[k][j];
 }
```

```
exy/=m;
ex/=m;
ey/=m;
for(k=0;k<m;k++){
sx+=(data[k][i]-ex)*(data[k][i]-ex);
sy+=(data[k][j]-ey)*(data[k][j]-ey);
}
sx = sqrt(sx/m);
sy = sqrt(sy/m);
corr[i][j] = (exy-(ex*ey))/(sx*sy);
corr[j][i] = (exy-(ex*ey))/(sx*sy);
}
}
}
// correlation matrix created
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("(%d,%d)%.3f ",i,j,corr[i][j]);
}
printf("\n");
}
float s=0;
// copy of matrix corr
float corr2[n][n],corr3[n][n],r[n][n];
float temp2[n][1],eig[n];
int l,abc;
float temp;
for(abc=0;abc<100;abc++)
{
 s=0;
 printf("\n\nITERATION %d\n\n",abc);
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 corr2[i][j]=corr[i][j];
 }
 }
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 temp2[j][0]=0;
 }
```

```
if(i==0){
for(j=0;j<n;j++){
s+=corr2[j][0]*corr2[j][0];
}
s = sqrt(s);
for(j=0;j<n;j++){
corr2[j][i]=corr2[j][i]/s;
}
/*
for(j=0;j<n;j++){
for(k=0;k<n;k++){
printf("%f ",corr2[j][k]);
}
printf("\n");
}
*/
}
else{
s=0;
for(k=0;k<i;k++){
temp=0;
//dot product
for(l=0;l<n;l++){
temp += corr2[l][k]*corr2[l][i];
}
// printf("tmep =%f\n",temp);
for(l=0;l<n;l++){
temp2[l][0]+=(corr2[l][k]*temp);
// printf("%f\n",temp2[l][0]);
}
temp=0;
}
for(k=0;k<n;k++){
corr2[k][i]-=temp2[k][0];
s+=(corr2[k][i]*corr2[k][i]);
}
s=sqrt(s);
for(k=0;k<n;k++){
corr2[k][i]/=s;
}
```

```
}
}
/*
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",corr2[i][j]);
}

printf("\n");
}
*/
for(i=0;i<n;i++){
for(j=0;j<n;j++){
corr3[i][j]=corr2[j][i];
}
}
/*
printf("\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",corr3[i][j]);
}
printf("\n");
}
*/
for(i=0;i<n;i++){
for(j=0;j<n;j++){
r[i][j]=0;
for(k=0;k<n;k++){
r[i][j]+=corr3[i][k]*corr[k][j];
}

}
}
/*
printf("\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",r[i][j]);
}
```

```
 printf("\n");
 }
*/
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 corr[i][j]=0;
 for(k=0;k<n;k++){
 corr[i][j]+=r[i][k]*corr2[k][j];
 }
 }
 }
/*
 printf("\n");
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){

 printf("%f ",corr[i][j]);
 }
 printf("\n");
 }
 */
}
 for(i=0;i<n;i++){
 eig[i]=corr[i][i];
 }
 for(i=0;i<n;i++){
 for (j=i;j<n;j++){
 if(eig[j]>eig[i]){
 temp=eig[i];
 eig[i]=eig[j];
 eig[j]=temp;
 }
 }
 }
stop=clock();
 printf("\n");
 for(i=0;i<n;i++){

 for(j=0;j<n;j++){
 printf("%f ",corr[i][j]);
```

```
 }
 printf("\n");
 }
 for(i=0;i<n;i++){
 printf("%f\n",eig[i]);
 }
printf("time = %lf\n",(double)(stop-start)/CLOCKS_PER_SEC);
}
```

- Parallel Code

```
#include<stdio.h>
#include<math.h>
#include<omp.h>
#include<time.h>
#include <stdlib.h>
void main(){
 int m=1000,n=500;
 int data[m][n];
 //int data[10][3]={{7,4,3},{4,1,8},{6,3,5},{8,6,1},{8,5,7},{7,2,9},
//{5,3,3},{9,5,8},{7,4,5},{8,2,2}};
 /*
 correlation matrix
 (E[xy] - E[x]E[y])/S(x)*S(y)
 */
 omp_set_num_threads(5);
 int i,j,k;
 float corr[n][n],exy=0,ex=0,ey=0,sx=0,sy=0;
 for(i=0;i<m;i++){
 for(j=0;j<n;j++){
 data[i][j]=rand();
 }
 }
clock_t start,stop;
```

```
start = clock();
#pragma omp parallel for reduction(+:exy,ex,ey,sx,sy) private(i,j,k)
 for (i=0;i<n;i++){
 for(j=i;j<n;j++){
// printf("(%d,%d) ",i,j);
 exy=0,ex=0,ey=0,sx=0,sy=0;
 //correlation between i and j column
 if (i==j){
 corr[i][j]=1;
 }
 else{
 //#pragma omp parallel for reduction(+:exy,ex,ey)
 for (k=0;k<m;k++){
 exy+=data[k][i]*data[k][j];
 ex+=data[k][i];
 ey+=data[k][j];
 }
 exy/=m;
 ex/=m;
 ey/=m;
 //#pragma omp parallel for reduction(+:sx,sy)
 for(k=0;k<m;k++){
 sx+=(data[k][i]-ex)*(data[k][i]-ex);
 sy+=(data[k][j]-ey)*(data[k][j]-ey);
 }
 sx = sqrt(sx/m);
 sy = sqrt(sy/m);
 corr[i][j] = (exy-(ex*ey))/(sx*sy);
 corr[j][i] = (exy-(ex*ey))/(sx*sy);
 }
 }
 }
 // correlation matrix created
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 printf("(%d,%d)%.3f ",i,j,corr[i][j]);
 }
 printf("\n" );
 }
float s=0;
```

```
// copy of matrix corr
float corr2[n][n],corr3[n][n],r[n][n];
float temp2[n][1];
float eig[n];
int l,abc;
float temp;
for(abc=0;abc<100;abc++)
{
 s=0;
 printf("\n\nITERATION %d\n\n",abc);
 // corr2 copy of correlation matrix
 #pragma omp parallel for collapse(2)
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 corr2[i][j]=corr[i][j];
 }
 //printf("tid = %d\n",omp_get_thread_num());
 }
 for(i=0;i<n;i++){
 #pragma omp parallel for
 for(j=0;j<n;j++){
 temp2[j][0]=0;
 }
 if(i==0){
 #pragma omp parallel for reduction(+:s)
 for(j=0;j<n;j++){
 s+=corr2[j][0]*corr2[j][0];
 }
 s = sqrt(s);
 #pragma omp parallel for
 for(j=0;j<n;j++){
 corr2[j][i]=corr2[j][i]/s;
 }
 /*
 for(j=0;j<n;j++){
 for(k=0;k<n;k++){
 printf("%f ",corr2[j][k]);
 }
 printf("\n");
 }
```

```
*/
}
else{
s=0;
// #pragma omp parallel for
for(k=0;k<i;k++){
temp=0;
//dot product
// #pragma omp critical
{
#pragma omp parallel for reduction(+:temp)
for(l=0;l<n;l++){
temp += corr2[l][k]*corr2[l][i];
}
//printf("temp =%f\n",temp);
#pragma omp parallel for reduction(+:temp2)
for(l=0;l<n;l++){
temp2[l][0]+=(corr2[l][k]*temp);
// printf("%f\n",temp2[l][0]);
}
temp=0;
}
}
#pragma omp parallel for reduction(+:s)
for(k=0;k<n;k++){
corr2[k][i]-=temp2[k][0];
s+=(corr2[k][i]*corr2[k][i]);
}
s=sqrt(s);
#pragma omp parallel for
for(k=0;k<n;k++){
corr2[k][i]/=s;
}
}
}
/*
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",corr2[i][j]);
}
```

```
printf("\n");
}
*/
#pragma omp parallel for collapse(2)
for(i=0;i<n;i++){
for(j=0;j<n;j++){
corr3[i][j]=corr2[j][i];
}
}
//printf("\n");
/*
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",corr3[i][j]);
}
printf("\n");
}
*/
#pragma omp parallel for private(i,j)
for(i=0;i<n;i++){
for(j=0;j<n;j++){
r[i][j]=0;
}
}
#pragma omp parallel for private(i,j,k)
for(i=0;i<n;i++){
for(j=0;j<n;j++){
for(k=0;k<n;k++){
r[i][j]+=corr3[i][k]*corr[k][j];
}
}
}
/*
printf("\n");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
printf("%f ",r[i][j]);
}
printf("\n");
}
```

```
*/
 #pragma omp parallel for private(i,j,k)
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 corr[i][j]=0;
 for(k=0;k<n;k++){
 corr[i][j]+=r[i][k]*corr2[k][j];
 }
 }
 }
 /*
 printf("\n");
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 printf("%f ",corr[i][j]);
 }
 printf("\n");
 }
*/
 }
#pragma omp parallel for
 for(i=0;i<n;i++){
 eig[i]=corr[i][i];
 }
 for(i=0;i<n-1;i++){
 for (j=i+1;j<n;j++){
 if(eig[j]>eig[i]){
 temp=eig[i];
 eig[i]=eig[j];
 eig[j]=temp;
 }
 }
 }
stop=clock();
 printf("\n");
 for(i=0;i<n;i++){
 for(j=0;j<n;j++){
 printf("%f ",corr[i][j]);
 }
 printf("\n");
```

```
 }
 for(i=0;i<n;i++){
 printf("%f\n",eig[i]);
 }
printf("time = %lf\n",(double)(stop-start)/CLOCKS_PER_SEC);
}
```