Parallel Computation
Of
Eigenvalues

Pushpendra

1639812

Abstract

We examine two algorithms for computing the eigenvalues and eigenvectors of real matrices. We will begin with the Jacobi method and then take up the QR method. Both these program we will first run in sequential way followed by OpenMp and then MPI. To access the program performance we are using PAPI.

Introduction

Eigenvalues are a special set of scalars associated with a linear system of equations that are sometimes also known as characteristic roots, characteristic values.

Eigenvalues and eigenvectors play an important part in the applications of linear algebra and equally important in physics and engineering. The naive method of finding the eigenvalues of a matrix involves finding the roots of the characteristic polynomial of the matrix. For very large sized matrices, however, this method is not feasible, and the eigenvalues must be obtained by other means. Fortunately, there exist several other techniques for finding eigenvalues and eigenvectors of a matrix, some of which fall under the realm of iterative methods. These methods work by repeatedly refining approximations to the eigenvectors or eigenvalues, and can be terminated whenever the approximations reach a suitable degree of accuracy. We will be using Jacobi method as iterative one.

The other approach is the decomposition one and we will be using QR Housholder method for it. QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix A into a product A = QR of an orthogonal matrix Q and an upper triangular matrix R.

1) Jacobi method:
   Jacobi eigenvalue algorithm is an iterative method for calculation of the eigenvalues and eigenvectors of a symmetric matrix. The algorithm is stable, but quite slow

2) Jacobi Serial method:

```
itr = 0;
 do
 {
  // Perform Jacobi iteration
  for (row = 0; row < N; row++)
  {
   dot = 0.0;
   for (col = 0; col < N; col++)
   {
    if (row != col)
     dot += A[row + col*N] * x[col];
   }
   xtmp[row] = (b[row] - dot) / A[row + row*N];
  }

  // Swap pointers
  ptrtmp = x;
  x    = xtmp;
  xtmp  = ptrtmp;

  // Check for convergence
  sqdiff = 0.0;
  for (row = 0; row < N; row++)
  {
   diff = xtmp[row] - x[row];
   sqdiff += diff * diff;
  }

  itr++;
 } while ((itr < MAX_ITERATIONS) && (Distance(A, xtmp, N) >= 1.0E-24));
```

Program reading:

Matrix size	Time sec	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.038610	6620	399	782	394

512	0.150865	7013	497	892	480
1024	0.594827	10014	808	12876	778
2048	3.399269	11919	1346	16236	1307
4096	33.134816	16743	8713	18379	8655
8192	135.427007	28752	13963	31254	20021

3) Jacobi OpenMp

```
Iteration = 0;
        do {

                for (irow = 0; irow < n_size; irow++)
                        X_Old[irow] = X_New[irow];

                // Start open mp
                #pragma omp parallel for private(icol) shared(Bloc_X,X_New,X_Old)
                for (irow = 0; irow < n_size; irow++) {
                        Bloc_X[irow] = Input_B[irow];

                        for (icol = 0; icol < irow; icol++) {
                                Bloc_X[irow] -= X_Old[icol] * Matrix_A[irow][icol];
                        }
                        for (icol = irow + 1; icol < n_size; icol++) {
                                Bloc_X[irow] -= X_Old[icol] * Matrix_A[irow][icol];
                        }
                        Bloc_X[irow] = Bloc_X[irow] / Matrix_A[irow][irow];
                }

                for (irow = 0; irow < n_size; irow++)
                        X_New[irow] = Bloc_X[irow];
                Iteration++;
        } while ((Iteration < MAX_ITERATIONS) && (Distance(X_Old, X_New, n_size) >= 1.0E-24));
```

Program reading:

#Cores=2

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.021325				
512	0.101224				
1024	0.322742				
2048	2.166234				

4096	20.172208				
8192	70.264004				

#Cores=4

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.012672				
512	0.060882				
1024	0.182273				
2048	1.844217				
4096	12.091532				
8192	40.112662				

#Cores=8

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.008331				
512	0.035122				
1024	0.114203				
2048	0.977432				
4096	8.124416				
8192	22.661437				

#Cores=16

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.005784				
512	0.020698				
1024	0.072507				
2048	0.644861				
4096	5.072609				
8192	13.300742				

4) QR method:

Let A be a real m × n matrix (m > n) with rank(A) = n. It is well known that A may be decomposed into the product A = QR where Q is (m×n) orthogonal ($Q^TQ = I_n$) and R is (n×n) upper triangular. Let A0:=A. At the k-th step (starting with k = 0), we compute the QR decomposition $A_k=Q_kR_k$ where Q_k is an orthogonal matrix (i.e., $Q^T = Q–1$) and R_k is an upper triangular matrix. We then form $A_k+1 = R_kQ_k$. Note that

$$A_{k+1} = R_kQ_k = Q_k^{-1}Q_kR_kQ_k = Q_k^{-1}A_kQ_k = Q_k^{\mathsf{T}}A_kQ_k,$$

so all the Ak are similar and hence they have the same eigenvalues. The algorithm is numerically stable because it proceeds by orthogonal similarity transforms.

Time Complexity:

$2mn^2–2n^3/3$

5) QR Serial

```
for(i = 0; i < lines; i++){

   double x = 0;
   vec = new double[lines-i];
   for(j = i; j < lines; j++){
     vec[j-i] = -matrixA[j][i];
     x += vec[j-i]*vec[j-i];
   }
   x = sqrt(x);

   if(vec[0] > 0) x = -x;
   vec[0] = vec[0] + x;
   x = 0;
   for(j = 0; j < lines-i; j++){
     x += vec[j]*vec[j];
   }
   x = sqrt(x);

   if(x > 0){
     //normalizovat vec
     for(j = 0; j < lines-i; j++){
       vec[j] /= x;
     }
```

```
    p = create_matrix(lines-i,lines-i, false);
    for(k = 0; k < lines-i; k++){
      for(l = 0; l < lines-i; l++){
        if(k == l) p[k][k] = 1 - 2*vec[k]*vec[l];
        else p[k][l] = -2*vec[k]*vec[l];
      }
    }


    for(k = i; k < lines; k++){
      for(l = i; l < lines; l++){
        double tm = 0;
        for(m = i; m < lines; m++){
          tm += p[k-i][m-i]*matrixA[m][l];
        }
        mat[k][l] = tm;
      }
    }
    for(k = i; k < lines; k++){
      for(l = i; l < lines; l++){
        matrixA[k][l] = mat[k][l];
      }
    }

    //Q
    for(k = 0; k < lines; k++){
      for(l = i; l < lines; l++){
        double tm = 0;
        for(m = i; m < lines; m++){
          tm += matrixQ[k][m]*p[m-i][l-i];
        }
        mat[k][l] = tm;
      }
    }
    for(k = 0; k < lines; k++){
      for(l = i; l < lines; l++){
        matrixQ[k][l] = mat[k][l];
      }
    }
   }
}
```

Program reading:

Matrix size	Time sec	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.03213889		172		16246
512	0.07375		1707		162199
1024	0.12253		1495099		1332781
2048	0.26332				
4096	0.56421				
8192	1.26326				

6) QR OpenMp

```
for(i = 0; i < lines; i++){
   double x = 0;
   vec = malloc((lines-i)*sizeof(double));
   for(j = i; j < lines; j++){
     vec[j-i] = -matrixA[j][i];
     x += vec[j-i]*vec[j-i];
   }
   x = sqrt(x);

   if(vec[0] > 0) x = -x;
   vec[0] = vec[0] + x;
   x = 0;
   for(j = 0; j < lines-i; j++){
     x += vec[j]*vec[j];
   }
   x = sqrt(x);

   if(x > 0){
     //normalizovat vec
     for(j = 0; j < lines-i; j++){
       vec[j] /= x;
     }

     //sestavit matici P
     p = create_matrix(lines-i,lines-i, false);
     for(k = 0; k < lines-i; k++){
       #pragma omp parallel for
       for(l = 0; l < lines-i; l++){
         if(k == l) p[k][k] = 1 - 2*vec[k]*vec[l];
         else p[k][l] = -2*vec[k]*vec[l];
       }
```

```
  }

  //nasobeni matic (paralelizace)
  //R
  double tm;
  for(k = i; k < lines; k++){
   #pragma omp parallel for private(tm,m)
   for(l = i; l < lines; l++){
    tm = 0;
    for(m = i; m < lines; m++){
     tm += p[k-i][m-i]*matrixA[m][l];
    }
    mat[k][l] = tm;
   }
  }
  for(k = i; k < lines; k++){
   #pragma omp parallel for
   for(l = i; l < lines; l++){
    matrixA[k][l] = mat[k][l];
   }
  }

  //Q
  for(k = 0; k < lines; k++){
   #pragma omp parallel for private(tm,m)
   for(l = i; l < lines; l++){
    tm = 0;
    for(m = i; m < lines; m++){
     tm += matrixQ[k][m]*p[m-i][l-i];
    }
    mat[k][l] = tm;
   }
  }
  for(k = 0; k < lines; k++){
   #pragma omp parallel for
   for(l = i; l < lines; l++){
    matrixQ[k][l] = mat[k][l];
   }
  }
  if(verbose) printf("Matrix Q in %d round.\n",i);
  for(k = 0; k < lines; k++){
   for(j = 0; j < lines; j++){
    if(verbose) printf("%f ",matrixQ[k][j]);
   }
   if(verbose) printf("\n");
  }
 }
 }for(i = 0; i < lines; i++){
```

```
double x = 0;
vec = malloc((lines-i)*sizeof(double));
for(j = i; j < lines; j++){
  vec[j-i] = -matrixA[j][i];
  x += vec[j-i]*vec[j-i];
}
x = sqrt(x);

if(vec[0] > 0) x = -x;
vec[0] = vec[0] + x;
x = 0;
for(j = 0; j < lines-i; j++){
  x += vec[j]*vec[j];
}
x = sqrt(x);

if(x > 0){
  //normalizovat vec
  for(j = 0; j < lines-i; j++){
    vec[j] /= x;
  }

  //sestavit matici P
  p = create_matrix(lines-i,lines-i, false);
  for(k = 0; k < lines-i; k++){
    #pragma omp parallel for
    for(l = 0; l < lines-i; l++){
      if(k == l) p[k][k] = 1 - 2*vec[k]*vec[l];
      else p[k][l] = -2*vec[k]*vec[l];
    }
  }

  //nasobeni matic (paralelizace)
  //R
  double tm;
  for(k = i; k < lines; k++){
    #pragma omp parallel for private(tm,m)
    for(l = i; l < lines; l++){
      tm = 0;
      for(m = i; m < lines; m++){
        tm += p[k-i][m-i]*matrixA[m][l];
      }
      mat[k][l] = tm;
    }
  }
  for(k = i; k < lines; k++){
    #pragma omp parallel for
    for(l = i; l < lines; l++){
```

```
      matrixA[k][l] = mat[k][l];
    }
   }

   //Q
   for(k = 0; k < lines; k++){
    #pragma omp parallel for private(tm,m)
    for(l = i; l < lines; l++){
     tm = 0;
     for(m = i; m < lines; m++){
      tm += matrixQ[k][m]*p[m-i][l-i];
     }
     mat[k][l] = tm;
    }
   }
   for(k = 0; k < lines; k++){
    #pragma omp parallel for
    for(l = i; l < lines; l++){
     matrixQ[k][l] = mat[k][l];
    }
   }
   if(verbose) printf("Matrix Q in %d round.\n",i);
   for(k = 0; k < lines; k++){
    for(j = 0; j < lines; j++){
     if(verbose) printf("%f ",matrixQ[k][j]);
    }
    if(verbose) printf("\n");
   }
  }
  }
```

Program reading:

#Cores = 4

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000602				
512	0.002229				
1024	0.008206				
2048	0.038583				
4096	0.342789				

8192	1.789551				

#Cores = 8

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000432				
512	0.001243				
1024	0.004687				
2048	0.022156				
4096	0.291324				
8192	1.351893				

#Cores = 16

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.00286				
512	0.000644				
1024	0.002846				
2048	0.012732				
4096	0.180769				
8192	1.004682				

7) QR MPI

```
MPI_Bcast((void *)&lines, 1, MPI_INT, 0, MPI_COMM_WORLD);
 matrixA = create_matrix(lines,lines, verbose);
 matrixQ = create_matrix(lines,lines, verbose);
 if(rank == 0){
  for(i = 0; i < lines; i++){
   matrixQ[i][i] = 1;
  }
  //loading of matrix from file to array
  for(i = 0; i < lines; i++){
   for(j = 0; j < lines; j++){
    fscanf(file,"%f,",&matrixA[i][j]);
   }
  }

  for(i = 0; i < lines; i++){
   for(j = 0; j < lines; j++){
    if(verbose) printf("%f ",matrixA[i][j]);
   }
   if(verbose) printf("| %f \n",matrixA[i][lines]);
  }

  //start time
  double start=MPI_Wtime();
        if (PAPI_start_counters(event, NUM_EVENTS) != PAPI_OK) {
   printf("PAPI_start_counters - FAILED\n");
  }
 }

 int k,l,m,tmpLines,tmpLines2;


 for(i = 0; i < lines; i++){
  tmpLines = (lines-i)/size;
  if(rank == (size-1) && size > 1) tmpLines += (lines-i)%size;
  tmpLines2 = (lines)/size;
  if(rank == (size-1) && size > 1) tmpLines2 += (lines)%size;

  if(i > 0 && i < lines-size){
   delete [] mat[0];
  }
```

```
mat = create_matrix(tmpLines,lines-i, false);

MPI_Bcast((void *)&matrixA[0][0], lines*lines, MPI_double, 0, MPI_COMM_WORLD);
MPI_Bcast((void *)&matrixQ[0][0], lines*lines, MPI_double, 0, MPI_COMM_WORLD);

int piece = (lines-i)/size;
int radek = lines-i;
send_counts[0] = (piece)*(radek);
displs[0] = 0;
if(size > 1){
  for(k = 1; k < size-1; k++){
    send_counts[k] = piece*(radek);
    displs[k] = displs[k-1] + piece*(radek);
  }
  displs[size-1] = displs[size-2] + piece*(radek);
  send_counts[size-1] = (((lines-i)*(radek)) - displs[size-1]);
}

piece = (lines)/size;
radek = lines-i;
send_counts2[0] = (piece)*(radek);
displs2[0] = 0;
if(size > 1){
  for(k = 1; k < size-1; k++){
    send_counts2[k] = piece*(radek);
    displs2[k] = displs2[k-1] + piece*(radek);
  }
  displs2[size-1] = displs2[size-2] + piece*(radek);
  send_counts2[size-1] = (((lines)*(radek)) - displs2[size-1]);
}

double x = 0;
if(i > 0 && i < lines-size){
  delete [] vec;
  vec = NULL;
}
vec = new double[lines-i];
if(rank == 0){
  for(j = 0; j < lines-i; j++){
    vec[j] = -matrixA[j+i][i];
    x += vec[j]*vec[j];
  }

  x = sqrt(x);


  if(vec[0] > 0) x = -x;
  vec[0] = vec[0] + x;
```

```
    x = 0;
    for(j = 0; j < lines-i; j++){
      x += vec[j]*vec[j];
    }
    x = sqrt(x);
   }
   MPI_Bcast((void *)&x, 1, MPI_double, 0, MPI_COMM_WORLD);
   MPI_Bcast((void *)&vec[0], lines-i, MPI_double, 0, MPI_COMM_WORLD);
   if(x > 0){
    if(rank == 0){

      for(j = 0; j < lines-i; j++){
        vec[j] /= x;
      }
    }

    MPI_Bcast((void *)&vec[0], lines-i, MPI_double, 0, MPI_COMM_WORLD);

    if(i > 0 && i < lines-size){
      delete [] p[0];
      p = NULL;
    }
    p = create_matrix(lines-i,lines-i, false);

    MPI_Scatterv(&p[0][0],send_counts,displs,MPI_double,&mat[0][0],tmpLines*(lines-
i),MPI_double,0,MPI_COMM_WORLD);
    for(k = 0; k < send_counts[rank]/(lines-i); k++){
      for(l = 0; l < lines-i; l++){
        if((k+(displs[rank]/(lines-i))) == l) mat[k][l] = 1 - 2*vec[k+displs[rank]/(lines-i)]*vec[l];
        else mat[k][l] = -2*vec[k+displs[rank]/(lines-i)]*vec[l];
      }
    }

MPI_Gatherv(&mat[0][0],send_counts[rank],MPI_double,&p[0][0],send_counts,displs,MPI_double,0,
MPI_COMM_WORLD);
    MPI_Bcast((void *)&p[0][0], (lines-i)*(lines-i), MPI_double, 0, MPI_COMM_WORLD);


    for(k = 0; k < send_counts[rank]/(lines-i); k++){
      for(l = 0; l < lines-i; l++){
        double tm = 0;
        for(m = i; m < lines; m++){
          tm += p[k+displs[rank]/(lines-i)][m-i]*matrixA[m][l+i];
        }
        mat[k][l] = tm;
      }
    }
    if(i > 0 && i < lines-size){
```

```
      delete [] matTmp[0];
      matTmp = NULL;
    }
    matTmp = create_matrix(lines-i,lines-i, false);

MPI_Gatherv(&mat[0][0],send_counts[rank],MPI_double,&matTmp[0][0],send_counts,displs,MPI_do
uble,0,MPI_COMM_WORLD);
    if(rank == 0){
      for(k = i; k < lines; k++){
        for(l = i; l < lines; l++){
          matrixA[k][l] = matTmp[k-i][l-i];
        }
      }
    }

    //Q
    if(i > 0 && i < lines-size){
      delete [] mat[0];
      mat = NULL;
    }
    mat = create_matrix(tmpLines2,lines-i, false);
    for(k = 0; k < send_counts2[rank]/(lines-i); k++){
      for(l = 0; l < lines-i; l++){
        double tm = 0;
        for(m = i; m < lines; m++){
          tm += matrixQ[k+displs2[rank]/(lines-i)][m]*p[m-i][l];
        }
        mat[k][l] = tm;
      }
    }

    if(i > 0 && i < lines-size){
      delete [] matTmp[0];
      matTmp = NULL;
    }
    matTmp = create_matrix(lines,lines-i, false);

MPI_Gatherv(&mat[0][0],send_counts2[rank],MPI_double,&matTmp[0][0],send_counts2,displs2,MPI
_double,0,MPI_COMM_WORLD);
    if(rank == 0){
      for(k = 0; k < lines; k++){
        for(l = i; l < lines; l++){
          matrixQ[k][l] = matTmp[k][l-i];
        }
      }
    }
  }
 }
```

Program reading:

#Processes=4

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000532				
512	0.002048				
1024	0.007806				
2048	0.032347				
4096	0.265632				
8192	1.553823				

#Processes=8

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000386				
512	0.001864				
1024	0.004261				
2048	0.018663				
4096	0.187834				
8192	1.009871				

#Processes=16

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000216				
512	0.001234				
1024	0.002488				
2048	0.011882	`			
4096	0.126427				
8192	0.798223				

#Processes=32

Matrix size	Time	L1 Data Cache misses (PAPI_L1_DCM)	L1 Instruction Cache misses (PAPI_L1_ICM)	L2 Data Cache misses (PAPI_L2_DCM)	L2 Instruction Cache misses (PAPI_L2_ICM)
256	0.000201				
512	0.000866				
1024	0.001277				
2048	0.00846				
4096	0.08441				
8192	0.49324				

Analysis of QR and Jacobi

The primary advantage of the Jacobi method over the symmetric QR algorithm is its parallelism. As each Jacobi update consists of a row rotation that affects only rows p and q, and a column rotation that effects only columns p and q, up to n/2 Jacobi updates can be performed in parallel. Therefore, a sweep can be efficiently implemented by performing n−1 series of n/2 parallel updates in which each row i is paired with a different row j.

The Jacobi iteration: Advantages of the Jacobi iteration

(i) Fully diagonalizes a symmetric matrix in one iteration
(ii) Stable.

Disadvantages of the Jacobi iteration

(i) Works only for symmetric matrices
(ii) Slow

QR: Advantages of the QR-iteration:

(i) Gives eigenvalues for general matrices and also eigenvectors for symmetric ones.
(ii) Gives everything in one iteration
(iii) Faster than Jacobi, if one has a good QR-decomposition algorithm
(iv) Stable