Homework #2

1. Overview

The work to be done is to parallelize and run the sequential scientific program written in Fortran and to speed it up as high as possible. The program is to be run on the Cray T3E parallel computer system at Supercomputer Center.

First, we have analyzed the sequential program systematically to find which portion of the code is critically time-consuming. We have also examined the feasibility of parallelization at the part where most of execution time is consumed. Then, we have parallelized the sequential code and measured the performance improvement including speedup.

The program consists of many 'DO' loops and the computations are performed on the three-dimensional arrays. Our parallelization and performance improvement are carried out primarily for the 'DO' loops and array operations.

2. Analysis of the Sequential Program

2.1 Program Structure

First, we have examined the hierarchy of the sequential program. The program is written in Fortan, in which most of computations are floating-point operations performed on three-dimensional arrays.

The source files for the program are as follows:

```plaintext
mypara.f      Major variables and parameters are defined.
mysfd.f      Main and subroutines for large eddy simulation
```

The program hierarchy is as follows in depth of three:

```plaintext
Main       /* main program */
INITIA     /* initialization */
READVF     /* data read */
```
The computation-intensive parts of the program consist of many 'DO' loops and array operations. Since most of array operations are carried out on the regular data structures of three-dimensional arrays, the 'DO' loops may be easily parallelized if there exists no unsafe data dependency.

2.2 Bottleneck analysis

We have measured the execution time of the whole sequential program and the first-level subroutines of the program by using the 'rtc()' function call in the Cray T3E parallel computer. The subroutine SAVEVF is not measured because it only saves velocity and pressure fields for future runs and may not be parallelized. The results measured are as follows:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>12.013829966666666</td>
</tr>
<tr>
<td>INITIA</td>
<td>9.1815622222222222E-3</td>
</tr>
<tr>
<td>CMPNUT</td>
<td>Not executed (default)</td>
</tr>
<tr>
<td>CMPVEL</td>
<td>1.812672888888888E-2</td>
</tr>
<tr>
<td>CMPBUO</td>
<td>1.14773244444444436E-2</td>
</tr>
<tr>
<td>SLVPOI</td>
<td>1.6486098133333334</td>
</tr>
<tr>
<td>UPDATE</td>
<td>8.3573088888888863E-3</td>
</tr>
</tbody>
</table>

Time unit is second for all the execution time. All ten steps are iterated in the main program in our run, i.e. the above subroutines are called ten times during a run except that the subroutine CMPVEL is called (executed) three times in each step.

3. Code Parallelization

Because the subroutine SLVPOI is the dominant part of execution time, we have analyzed it in detail.
The execution time of SLVPOI is 100 to 200 times longer than that of other subroutines. Then, we have measured the two second-level subroutines of SLVPOI in depth. The results measured are as follows:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPS</td>
<td>3.820451111111108E-3</td>
</tr>
<tr>
<td>GAUSEI</td>
<td>1.61221607555555555</td>
</tr>
</tbody>
</table>

As shown in the data, the subroutine GAUSEI is the dominant part of SLVPOI. That is, most of computation time is spent in the subroutine GAUSEI. The execution time of GAUSEI is approximately 200 times longer than that of the other subroutine CMPS. So it is more preferable that we develop the parallel code for the subroutine GAUSEI.

Subsequently, we have analyzed the 'DO' loops in depth. There exist two 'DO' loops in the subroutine GAUSEI: the loop labeled 500 and the loop labeled 200. Also we have measured the execution time of the two 'DO' loops. The results measured are as follows:

<table>
<thead>
<tr>
<th>Loop</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop 500</td>
<td>8.04333555555555551E-3</td>
</tr>
<tr>
<td>Loop 200</td>
<td>4.05133999999999963E-3</td>
</tr>
</tbody>
</table>

Each 'DO' loop is executed many times repeatedly until the computation results are converged, in worst case of more than 200 times in our run.

Each performs floating-point operations for three-dimensional arrays. And the operations require both adjacent data in the array spatially and previous data in the array temporally. Therefore, the communications for boundary data exchange are necessary before the computation proceeds to next time step so that the 'DO' loops are correctly parallelized. Consequently, we have used MPI_SEND and MPI_RECV to exchange the calculated data before proceeding to next time step.

The data decomposition is a critical issue in our parallel code. Memory address is assigned by array-oriented allocation in Fortran. That is, \( A[i,j,k] \) and \( A[i+1,j,k] \) are located in adjacent memory addresses. In the two 'DO' loops, Z dimension corresponds to the index \( i \) of \( A[i,j,k] \) notation and the value of Z is given as 128 in the include file 'mypara.f'. So, we have decomposed the three-dimensional array data on the basis of Z dimension. After computation at each time step, all participating processes exchange their boundary results each other to proceed to next time step. Then, every process starts the computation repeatedly until the computation result is converged.

Figure 1 shows the data decomposition of a three-dimensional array which is applied to our parallelization work.
The parallel code of the loop labeled 500 is as follows:

```fortran
... 400 DO 500 K=1,NZ
   400 DO 500 K=ioffsetNZ,(ioffsetNZ+iaveNZ-1)
   DO 500 I=1,NX
      PHI(I,1,K) = (AX*(PHI(I+1,1,K)+PHI(I-1,1,K))
      &            +AY*PHI(I,2,K)
      &            +AZ*(PHI(I,1,K+1)+PHI(I,1,K-1))
      &            -FARRAY(I,1,K)
      &            -FARRAY(I,1,K)
      &            -APYB*OMEGA+PHI(I,1,K)*0.0E0-OMEGA)
   DO 510 J=2,NY-1
      PHI(I,J,K) = (AX*(PHI(I+1,J,K)+PHI(I-1,J,K))
      &            +AY*(PHI(I,J+1,K)+PHI(I,J-1,K))
      &            +AZ*(PHI(I,J,K+1)+PHI(I,J,K-1))
      &            -FARRAY(I,J,K)
      &            -AP*OMEGA+PHI(I,J,K)*0.0E0-OMEGA)
   510 CONTINUE
   PHI(I,NY,K) = (AX*(PHI(I+1,NY,K)+PHI(I-1,NY,K))
      &            +AY*PHI(I,NY-1,K)
      &            +AZ*(PHI(I,NY,K+1)+PHI(I,NY,K-1))
      &            -FARRAY(I,NY,K))
      &            -APYB*OMEGA+PHI(I,NY,K)*0.0E0-OMEGA)
500 CONTINUE

CP Data exchange
   IF (itaskid.EQ.((itaskid/2)*2)) THEN
      IF (itaskid.NE.0) THEN
         CALL MPI_SEND(PHI(0,0,ioffsetNZ),
         &      (NX+2)*(NY+2),MPI_REAL,(itaskid-1),101,
         &      MPI_COMM_WORLD,ierrmsg);
         ENDIF
         CALL MPI_SEND(PHI(0,0,(ioffsetNZ+iaveNZ-1)),
         &      (NX+2)*(NY+2),MPI_REAL,(itaskid+1),102,
         &      MPI_COMM_WORLD,ierrmsg);
      ELSE
         IF (itaskid.NE.(inumtasks-1)) THEN
```

Figure 1: Data (domain) decomposition of a 3-D array
CALL MPI_RECV(PHI(0,0,(ioffsetNZ+iaveNZ)),
  (NX+2)*(NY+2),MPI_REAL,(itaskid+1),101,
  MPI_COMM_WORLD,istatus,ierrmsg);
ENDIF
CALL MPI_RECV(PHI(0,0,(ioffsetNZ-1)),
  (NX+2)*(NY+2),MPI_REAL,(itaskid-1),102,
  MPI_COMM_WORLD,istatus,ierrmsg);
ENDIF
IF (itaskid.EQ.((itaskid/2)*2)) THEN
  CALL MPI_RECV(PHI(0,0,(ioffsetNZ+iaveNZ)),
    (NX+2)*(NY+2),MPI_REAL,(itaskid+1),201,
    MPI_COMM_WORLD,istatus,ierrmsg);
  IF (itaskid.NE.0) THEN
    CALL MPI_RECV(PHI(0,0,(ioffsetNZ-1)),
      (NX+2)*(NY+2),MPI_REAL,(itaskid-1),202,
      MPI_COMM_WORLD,istatus,ierrmsg);
  ENDIF
ELSE
  CALL MPI_SEND(PHI(0,0,ioffsetNZ),
    (NX+2)*(NY+2),MPI_REAL,(itaskid-1),201,
    MPI_COMM_WORLD,ierrmsg);
  IF (itaskid.NE.(inumtasks-1)) THEN
    CALL MPI_SEND(PHI(0,0,(ioffsetNZ+iaveNZ-1)),
      (NX+2)*(NY+2),MPI_REAL,(itaskid+1),202,
      MPI_COMM_WORLD,ierrmsg);
  ENDIF
ENDIF
... 

The parallel code of the loop labeled 200 is as follows:

... 
CP  DO 200 K=1,NZ
  DO 200 K=ioffsetNZ,(ioffsetNZ+iaveNZ-1)
  DO 200 I=1,NX
    DIF(I,1,K)=PHI(I,1,K)
    & -APYB*(AX*(PHI(I+1,1,K)+PHI(I-1,1,K))
    &   +AY*PHI(I,2,K)
    &   +AZ*(PHI(I,1,K+1)+PHI(I,1,K-1))
    & -FARRAY(I,1,K))
    TDIF=TDIF+ABS(DIF(I,1,K))
    TOTAL=TOTAL+ABS(PHI(I,1,K)-PHI0)
  DO 210 J=2,NY-1
    DIF(I,J,K)=PHI(I,J,K)
    & -AP*(AX*(PHI(I+1,J,K)+PHI(I-1,J,K))
    &   +AY*(PHI(I,J+1,K)+PHI(I,J-1,K))
    &   +AZ*(PHI(I,J,K+1)+PHI(I,J,K-1))
    & -FARRAY(I,J,K))
    TDIF=TDIF+ABS(DIF(I,J,K))
    TOTAL=TOTAL+ABS(PHI(I,J,K)-PHI0)
  210 CONTINUE
  DIF(I,NY,K)=PHI(I,NY,K)
  & -APYB*(AX*(PHI(I+1,NY,K)+PHI(I-1,NY,K))
  &   +AY*PHI(I,NY-1,K)
  &   +AZ*(PHI(I,NY,K+1)+PHI(I,NY,K-1))
  & -FARRAY(I,NY,K))
  TDIF=TDIF+ABS(DIF(I,NY,K))
  TOTAL=TOTAL+ABS(PHI(I,NY,K)-PHI0)
  200 CONTINUE
CP  Data exchange
  IF (itaskid.EQ.0) THEN
    DO 5675 imoh=1,(inumtasks-1)
      CALL MPI_RECV(TDIFTMP,
        1,MPI_REAL,imoh,785,
        MPI_COMM_WORLD,istatus,ierrmsg);
      TDIF=TDIF+TDIFTMP
    5675 CONTINUE
    DO 5676 imoh=1,(inumtasks-1)
      CALL MPI_RECV(TOTALTMP,
        1,MPI_REAL,imoh,786,
        MPI_COMM_WORLD,istatus,ierrmsg);
      TOTAL=TOTAL+TOTALTMP
    5676 CONTINUE

5
DO 5677 imoh=1,(inumtasks-1)
   CALL MPI_SEND(TDIF, &
   1, MPI_REAL, imoh, 787, &
   MPI_COMM_WORLD, ierrmsg);  
5677 CONTINUE
DO 5678 imoh=1,(inumtasks-1)
   CALL MPI_SEND(TOTAL, &
   1, MPI_REAL, imoh, 788, &
   MPI_COMM_WORLD, ierrmsg); 
5678 CONTINUE
CP   CALL MPI_BARRIER(MPI_COMM_WORLD, ierrmsg);
ELSE
   CALL MPI_SEND(TDIF, &
   1, MPI_REAL, 0, 785, &
   MPI_COMM_WORLD, ierrmsg); 
   CALL MPI_SEND(TOTAL, &
   1, MPI_REAL, 0, 786, &
   MPI_COMM_WORLD, ierrmsg); 
   CALL MPI_RECV(TDIF, &
   1, MPI_REAL, 0, 787, &
   MPI_COMM_WORLD, istatus, ierrmsg); 
   CALL MPI_RECV(TOTAL, &
   1, MPI_REAL, 0, 788, &
   MPI_COMM_WORLD, istatus, ierrmsg); 
CP   CALL MPI_BARRIER(MPI_COMM_WORLD, ierrmsg);
ENDIF
... 

After the parallelization and run, we have checked the correctness of the parallelized code. We have compared the results of the parallel code with those of the sequential code, and we have confirmed the two results are the same.

4. Performance Improvement

The parallelized code of the sequential program is run on the Cray T3E parallel computer for the various number of processors. Then we have measured the execution time of each run in the master node, and compared them with each other.

The execution time measured is summarized as follows:

<table>
<thead>
<tr>
<th># PEs</th>
<th>Execution time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.0138299666666666</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>9.7844039866666659</td>
<td>1.23</td>
</tr>
<tr>
<td>4</td>
<td>6.3714660555555556</td>
<td>1.89</td>
</tr>
<tr>
<td>6</td>
<td>5.3602804866666665</td>
<td>2.24</td>
</tr>
<tr>
<td>8</td>
<td>5.1341545599999998</td>
<td>2.34</td>
</tr>
<tr>
<td>10</td>
<td>5.0126524400000001</td>
<td>2.40</td>
</tr>
</tbody>
</table>

As the number of processors are increased, the speedup is degraded gradually because some communications are used in the parallelized code for boundary data exchange of 3-dimensional arrays. Figure 2 shows the execution time with respect to the number of participating processors on the Cray T3E
parallel computer system.

![Graph showing execution time vs number of participating nodes.](image)

Figure 2: Execution time with respect to the number of processors

5. Concluding Remarks

As we can see Figure 2, the performance enhancement is not so large, for many data exchange communications of 3-dimensional arrays depress the speedup of the parallel code. Moreover, at every time step, the boundary data are exchanged repeatedly. Hence, if the number of processors is increased more and more, the linearity of speedup is rapidly decreased because the communication becomes the dominant factor of total execution time under the fixed array size. That is, the amount of inherent communication restricts the practical degree of parallelism in the given application.