Helpful Hints for Lab Sessions

*A guide to the practical work (in PDF format) can be found on the UvA Blackboard. For lab sessions you are encouraged to work independently, however during lab sessions assistances can provide you with necessary help. You can also work in groups of no more than two students. As indicated in page 25 of the "**Guide to the Practical Work**", this course involves a large amount of practical work; you are urged to stick to the schedule.*

1. **Minimum program for first lab session (First assignment):**

a) Set up MPI environment on both the OW cluster and the Lisa.

b) Run sample parallel programs (e.g., mpitestprg.c and some simple examples on the book) on both the OW cluster and the Lisa clusters.

c) Understand how to use commands like MPI_Send and MPI_Recv, etc.

d) Understand the meaning of the first assignment (Parallel Matrix Vector Product) and begin to write parts of the code (see the content of files below).

.aliases file

```
#
#       Aliases file sourced in the .cshrc file
#
alias   h               'history 23'
alias   l               'ls -F'
alias   ll              'ls -lFt'
alias   la              'ls -alFt'
alias   .               'source'
alias   j               'jobs -l'
alias   z               'suspend'
alias   mail            mailx
alias   rm              '/bin/rm -i'
alias mpirun 'mpirun -machinefile $HOME/.mpirun.machines'
```

.mpirun.machines

```
ow2     ow10
ow2     ow11
ow3     ow13
ow5     ow14 ..ow21
ow6
ow7
. . .
```

.pkgrc

```
#
#    Default packages loaded by softpkg
#
#    $Id: pkgrc,v 1.3 1997/03/07 09:29:24 gert Exp $
#

#    The DEFAULT package may differ across architectures, it contains
#    at least local utilities. It also includes the search-path components
#
#    ~/bin/sh or $HOME/bin/sh     for your own shell scripts, and
#    ~/bin/ARCH or $HOME/ARCH    where ARCH is sparc for suns, RS6000 for
#                                    ibms, i386 for unix-pc's.
#                                    Store your own binary executables here.
#                                    softpkg -D show your systems architecture.

DEFAULT

#    uncomment the next entry to have "." ($cwd) in your path

# dot

#    If a package starts with a slash (/) or a percent (%) it is not
#    considered a package name but an absolute path of a directory. Softpkg
#    will check for its existence, and if it exists include the absolute
#    path in the definition of the PATH variable.

#    So if you want to include a bin directory in a project directory which
#    is in your homedirectory, you can include something like:

# %HOME/project/bin

#    WARNING
#
#    If, for example, you want to use the Java Development kit, DO NOT
#    include /usr/local/java as an absolute path in your .pkgrc file.
#    Instead DO use the predefined package "java".
#
#    softpkg -l
#
#    shows all packages that are installed. In general, preinstalled
#    package names are preferred.
#
#    Note in the above example that %XXX are softpkg variables. Softpkg
#    will not expand $YYY shell variables. You may use the following
#    softpkg variables:
```

```
#
#   %USER    your username
#   %HOME    your homedirectory. typically /home/%USER
#   %ARCH    the systems processor architecture
#   %HOST    the systems hostname
#   %OSNAME        the name of the operating system in lowercase
#   %OSMAJOR       the major operating system revision
#   %OSMINOR       the minor operating system revision
#

#    Add your own packages below
elm
mathematica
mpich
matlab52
```

Structure of the matrix-vector product parallel program

1. Initialize MPI.
2. Compute data decomposition
3. Generate matrix elements locally using dynamic memory allocation.
4. Generate vector elements locally and collect them by using a global routine (e.g. MPI_Allgather).
5. Do the matrix vector product in parallel.
6. Collect the results and print them.
7. Terminate MPI.

2. Second lab session (First assignment):

After establishment of the MPI environment correctly during the first lab session you can work with actual program. Herewith you can find some useful sample programs so that you can have an easier start.

a) Sample version of matrix vector product "sample.c". You can easily run it in parallel on the OW cluster or Lisa.

In this sample program of matrix vector product, at the very beginning, the matrix elements are on every node rather than distributed. The vector is broadcasted to every node from processor 0 through MPI_Bcast. Then the product is done in parallel and results are collected through MPI_Send and MPI_Recv. Do spend time to understand the every part of this sample program and it will help you greatly in the further works

```
#include <stdio.h>
```

```
#include <mpi.h>
#include <math.h>
#define MAXVEC 10                  /* size of biggest vector */
#define n 10

main(int argc, char *argv[])
{
  int i, j, rank;
  int p;                           /* p is the number of processors */
  int npoints, offset;             /* npoints is the number of rows which the
                                      current node is responsible to calculate */
  float vector[n];
  float matrix[n][n];
  MPI_Status status;
  float res[n];                     /* array res is to store the results of the product*/

  /* Initialize MPI */
  MPI_Init(&argc,&argv);

  /* Get environment, i.e. the number of processors and the i.d. */
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(p > MAXVEC)
    {
     if(rank==0)
       printf("Too many processes, maximum allowed: %d/n", MAXVEC);
       MPI_Abort(MPI_COMM_WORLD,0);
    }

  /* Initialization of the elements of the vector and the matrix */
  for(i=0;i<n;i++)
          {vector[i]=0.;res[i]=0.;}
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
matrix[i][j]=i;

  /* Generate the vector on processor 0 and print out the elements of the vector and
     the matrix*/
  if(rank==0)
    {
     for(i=0;i<n;i++)
vector[i]=i;
          printf("the elements of the matrix are:\n");
     for(i=0;i<n;i++)
printf("%0.0f  ",vector[i]);
```

```
        printf("\nThe elements of matrix are:\n");
    for(i=0;i<n;i++)
      {
       for(j=0;j<n;j++)
          printf("%0.0f  ",matrix[i][j]);
          printf("\n");}
       }

/* Compute data decomposition, i.e., the values of npoints and offset*/
npoints = n/p+(n%p+p-rank-1)/p;
if(npoints>n/p)
   offset=npoints*rank;
else
  offset=n-(p-rank)*npoints;

/* Broadcast the vector from process 0 to all other processes */
MPI_Bcast(vector, n, MPI_FLOAT, 0, MPI_COMM_WORLD);

/* Do the matrix vector product in parallel */
/* Each process simply performs an inner product */
for(i=0;i<npoints;i++)
  for(j=0;j<n;j++)
  res[i]+=vector[j]*matrix[i+offset][j];

/* All processes send their result back to process 0,*/
 which puts the results in the result vector and prints them */
if(rank==0)
  {
    for(i=1;i<p;i++)
         {
       npoints = n/p+(n%p+p-i-1)/p;
       if(npoints>n/p)
  offset=npoints*i;
       else
   offset=n-(p-i)*npoints;
                  MPI_Recv(&res[offset],npoints, MPI_FLOAT, i, i,
  MPI_COMM_WORLD, &status);
          }
    printf("\n\nThe result vector is: \n\n");
    for(i=0;i<n;i++)
      printf("  %f\n",res[i]);
  }
else
  {
    MPI_Send(&res[0], npoints, MPI_FLOAT, 0, rank, MPI_COMM_WORLD);
  }
```

```
 /* Terminate MPI */
 MPI_Finalize();
}
```

What you need to do is to understand it and learn from this program so that you can complete the missing codes in the formal program "formal.c". Be aware that in the lab report if you submit "sample.c" rather than the completed "formal.c" as your final program, then you will get no score on this first assignment.

b) formal.c

This is the formal program for the first assignment. Please complete the missing code of PART A, B, C, D, and E. In many aspects you can learn from the sample program (i.e., sample.c) that we sent you. Note that PART D and PART E both consist of only single line of code.

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#define MAXVEC 10                  /* size of biggest vector */
#define n 10

main(int argc, char *argv[])
{
 int i, j,rank;
 int p,npoints,offset;              /* p is the number of processors, npoints is the
                                       number of rows which the current node is
                                       responsible to calculate */
 float *subvector;                  /* array subvector is to store the distributed
                                       elements of the vector */
 float *submatrix;                  /* array submatrix is to store the distributed
                                       elements of the matrix */
 float vector[n],newvector[2*n];
 MPI_Status status;
 float res[n];

 /* Initialize MPI */

  PART   A

 /* Get environment, i.e. the number of processors and the i.d. */

  PART   B

 for(i=0;i<n;i++)
```

```
{
   vector[i]=0.;
   newvector[2*i]=0.;
   newvector[2*i+1]=0.;
   res[i]=0.;
};

 /*Compute data decomposition, i.e., calculate the values of npoints and offset*/

   PART   C

 /*Generate matrix elements locally*/

   PART   D

 for(i=0;i<npoints;i++)
  for(j=0;j<n;j++)
    *(submatrix+i*n+j)=offset+i;

 /*Generate vector elements locally and collect them*/
 subvector = (float *)malloc((n/p+1)*sizeof(float));
 for(i=0;i<n/p+1;i++)
       *(subvector+i)=0.;
 for(i=0;i<npoints;i++)
       *(subvector+i)=(float) rank;


   PART   E                              /*using MPI_Allgather to collect all distributed
                                           elements of subvector into newvector*/

 for(i=0;i<p;i++)
{
    npoints = n/p+(n%p+p-i-1)/p;
    if(npoints>n/p)
         offset=npoints*i;
    else
         offset=n-(p-i)*npoints;
    for(j=0;j<npoints;j++)
      vector[offset+j]=newvector[i*(n/p+1)+j];
}

 /*Print the elements of the vector and the matrix*/
 if(rank==0)
{
  printf("the elements of the vector are:\n");
  for(i=0;i<n;i++)
```

```
printf("%0.0f  ",vector[i]);
                printf("\n the elements of the matrix are:\n");
   for(i=0;i<n;i++)
{
     for(j=0;j<n;j++)
printf("  %d",i);
           printf("\n");
}
}

  /* Now, finally do the matrix vector product in parallel */
  /* Each process simply performs an inner product */
  npoints = n/p+(n%p+p-rank-1)/p;
  if(npoints>n/p)
offset=npoints*rank;
  else
offset=n-(p-rank)*npoints;
  for(i=0;i<npoints;i++)
    for(j=0;j<n;j++)
      res[i]+=vector[j]*(*(submatrix+i*n+j));

  /* All processes send their result back to process 0 */
  /* which puts the results in the result vector and prints them */
  if(rank==0)
    {
      for(i=1;i<p;i++)
           {
        npoints = n/p+(n%p+p-i-1)/p;
        if(npoints>n/p)  offset=npoints*i;
        else
offset=n-(p-i)*npoints;
                MPI_Recv(&res[offset],npoints, MPI_FLOAT, i, i, MPI_COMM_WORLD,
&status);
            }
      printf("\n\nThe result vector is: \n\n");
      for(i=0;i<n;i++)
        printf("  %f\n",res[i]);
    }
  else
    {
      MPI_Send(&res[0], npoints, MPI_FLOAT, 0, rank, MPI_COMM_WORLD);
    }
  free(subvector);
  free(submatrix);

/* Terminate MPI */
```

```
  MPI_Finalize();
}
```

c) The third attached program “scsample.c” is to help you understand how to measure the scalability.

This program is to measure the scalability of the sample program (i.e.,sample.c) of matrix vector product. The wall clock time is acquired using function gettimeofday(). Please try to find all the differences between this program and sample.c. And this will help you to write a similar program for the formal program (i.e., formal.c) to test its scalability

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <sys/time.h>
#ifndef EXIT_FAILURE
#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0
#endif
#define MAXVEC 10                /* size of biggest vector */
#define n 10

main(int argc, char *argv[])
{
  int i, j, rank;
  int p;                         /* p is the number of processors */
  int npoints,offset;            /* npoints is the number of rows which the current
                                    node is responsible to calculate */
  float vector[n];
  float matrix[n][n];
  MPI_Status status;
  float res[n];                  /* array res is to store the results of the product*/

  double td;
  long sec,usec;
  struct timeval tp;

  /* Initialize MPI */
  MPI_Init(&argc,&argv);

  /* Get environment, i.e. the number of processors and the i.d. */
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(p > MAXVEC)
    {
```

```
    if(rank==0)
      printf("Too many processes, maximum allowed: %d/n", MAXVEC);
      MPI_Abort(MPI_COMM_WORLD,0);
   }

  /*use function gettimeofday() to record the first time point*/
  if(rank==0)
      {
        if(gettimeofday(&tp,NULL)==0)
          {
            /* Record the first point of time */
            sec=tp.tv_sec;
  usec=tp.tv_usec;
          }
        else
          {
            perror("Can not get Wall-Clock time!\n");
            MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
          }
      }

  /* Initialization of the elements of the vector and the matrix */
  for(i=0;i<n;i++)
{
   vector[i]=0.;
          res[i]=0.;
       }
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
matrix[i][j]=i;

  /* Generate the vector on  processor 0 and print out the elements of the vector and
     the matrix*/
  if(rank==0)
    {
     for(i=0;i<n;i++)
vector[i]=i;
          printf("the element of the matrix are:\n");
     for(i=0;i<n;i++)
printf("%0.0f  ",vector[i]);
          printf("\nThe elements of matrix are:\n");
     for(i=0;i<n;i++)
       {
        for(j=0;j<n;j++)
        printf("%0.0f  ",matrix[i][j]);
        printf("\n");
```

```
}
 }

  /* Compute data decomposition, i.e., the values of npoints and offset*/
  npoints = n/p+(n%p+p-rank-1)/p;
  if(npoints>n/p)
    offset=npoints*rank;
  else
    offset=n-(p-rank)*npoints;

  /* Broadcast the vector from process 0 to all other processes */
  MPI_Bcast(vector, n, MPI_FLOAT, 0, MPI_COMM_WORLD);

  /* Do the matrix vector product in parallel */
  /* Each process simply performs an inner product */
  for(i=0;i<npoints;i++)
    for(j=0;j<n;j++)
    res[i]+=vector[j]*matrix[i+offset][j];

  /* All processes send their result back to process 0 ,
      which puts the results in the result vector and prints them */
  if(rank==0)
    {
      for(i=1;i<p;i++)
           {
        npoints = n/p+(n%p+p-i-1)/p;
        if(npoints>n/p)
offset=npoints*i;
        else
offset=n-(p-i)*npoints;
                  MPI_Recv(&res[offset],npoints, MPI_FLOAT, i, i, MPI_COMM_WORLD,
    &status);
           }
      printf("\n\nThe result vector is: \n\n");
      for(i=0;i<n;i++)
        printf("  %f\n",res[i]);
    }
  else
    {
      MPI_Send(&res[0], npoints, MPI_FLOAT, 0, rank, MPI_COMM_WORLD);
    }

  /*use function gettimeofday() to record the second time point*/
  if(rank==0)
      {
        if(gettimeofday(&tp,NULL)==0)
```

```
            {
              sec=tp.tv_sec-sec;
    usec=tp.tv_usec-usec;
              /* Get the second point of time */
              td=sec+usec/1.0e6;
                    /* Calculate the time difference and print it out */
               printf("The execution time of the matrix vector product program using
                        %d processors is:\n %f    seconds\n",p,td);
            }
           else
            {
              perror("Can not get Wall-Clock time!\n");
              MPI_Abort(MPI_COMM_WORLD,EXIT_FAILURE);
            }
    }

  /* Terminate MPI */
  MPI_Finalize();
}
```

Follows are some technical details which may be helpful for you:

a) For those of you who still has difficulty in establishing the environment on the gene OW cluster, you can download the attached 3 files: .pkgrc, .mpirun.machines and .aliases to your root directory on gene, then log out and in again. After this, you should be able to run on at least 20 nodes in parallel.

b) As indicated in the "Guide to the Practical Work", if you want to run parallel program on the Lisa cluster, you need a batch file. A sample batch file named as "testbatch" can be found on blackboard. You can use this batch file as a template to "qsub" your MPI jobs. There you can also find "mpitestprg.c". You can compile and run this program as a test.

c) To transfer files from gene OW cluster to the Lisa cluster, you can use command "ftp lisa.sara.nl", then the system will ask for your name and password. After entering your account name and the password, you can see the ftp interface:

ftp>

then you should use command "put <filename>" to put a file on the Lisa for use "get <filename>" to get a file from the Lisa. If you want to put or get more than one files in one time, you can use "mput" or "mget". After you finish, you should use "quit" to leave the ftp interface. If you want to transfer files from the Lisa cluster to the gene OW cluster, please use command "ftp gene.science.uva.nl".

d) To change your password on the Lisa cluster, please use command "yppasswd".

e) As explanations of MPI commands in the book "Parallel scientific computing in C++ and MPI" are distributed all over the whole book, follows is an incomplete list of the locations of these stuffs:

MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Comm_size (See Page 75-76)

MPI_Send and MPI_Recv (See page 179-181)

MPI_Sendrecv and MPI_Sendrecv_replace (See page 271-273)

MPI_Allgather and MPI_Scatter (See page 376-378)

MPI_Reduce and MPI_Allreduce (See page 246-249)

MPI_Wtick and MPI_Wtime (See page 336)

(Note: You can also use MPI_Wtick and MPI_Wtime to measure the scalability)

Especially we advise you to have a careful look into those introductions and sample programs of MPI_Allgather, MPI_Send, MPI_Recv,MPI_Wtick and MPI_Wtime. What is more, when you are on the gene OW cluster or on the Lisa, you can always use command "man <mpicommandname>" to see the manual of this mpi command.

What is scalability

Scalability is defined as the study of how the performance of parallel program behaves as a function of the number of processors and as a function of the problem size. It is usually characterized by :

(1) the wall clock time that is needed to complete the job: Tp

(2) the relative speedup: Sp=T1/Tp

(3) the relative efficiency: Sp/p

You can find some introduction stuff of scalability on page 69 of the book. For the vector matrix product, we hope that you can measure the wall clock time on p=1,2,4,8 processors. And the suggested data size can be n=1000, 2000, 3000. Do not run too large size problem and occupy too many nodes on the Lisa cluster, otherwise it will be easily jammed.

3. Third lab session (Second assignment)

a) To start doing the second assignment, you should first download following files from the blackboard:

apr.h - a specific header file for string problem
makefile - you can use this file to compile
waveplot - you will need this file to process the data
wave.c - the template skeleton file

What you need to do is to complete the missing codes in "wave.c", then compile it using "Makefile" or use "mpicc" (see page 21 of the "Guide to the practical work") and run it using "mpirun" and visualize it (see page 38-39 of the "Guide to the practical work").

Obviously this problem is far from being an easy task. Additional theoretical materials and explanations will be available during the lab sessions.

b) Structure "parms" in "wave.c"

Structure "parms" is a parameter structure which stores all the necessary parameters which are passed to the program when we run it using "mpirun" (see page 38 of the Guide). You can find the complete description of this structure in "apr.h".

So the first step is that you parse the command line options using "parseargs()" in processor 0, then you have to broadcast the content of this structure from processor 0 to all the other processors.

Be aware that you can not broadcast the whole structure directly using "MPI_Bcast". So the recommended way is to broadcast each item of this structure by using "MPI_Bcast". And this means you have to use many "MPI_Bcast" sentences.

c) To visualize the vibrating string, after you run the program with "mpirun", you should type "./waveplot" to process the data first, then you go to the directory "~/tmp/wave" to visualize using "gnuplot".

d) For those of you who want to compile and run "wave.c" on the Lisa cluster, you need to copy a special "Makefile" from blackboard to compile. After running the program, you should use "perl waveplot" to process the data first, then go to the directory "~/tmp/wave" to visualize using "gnuplot".

4. Fourth lab session (Third assignment)

a) The template of the third assignment

Below you can find an almost empty template program for the third assignment. As usual, you need to complete the program, then run it and compare your simulation results with those analytical results which are stored in "opt/stud/pwrs/NEW/datafiles" on the gene OW cluster. Actually we hope that you can finish this assignment in two lab sessions and then we can quickly proceed to the iterative solvers like Jacobi iteration and write the report.

This is a template program for the time-dependent diffusion.
The space domain decomposition is done by using the strip-wise decomposition. You need to complete the missing codes (PART A,B,C,D,E,F). In many aspects you can learn from the vibrating string problem and maybe also the vector matrix product problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#ifndef EXIT_FAILURE
#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0
#endif
#define TRUE 1
#define FALSE 0
#define N 100          /*the number of grid data points in each row or column*/
#define dt 0.00001     /*the time stepsize*/
#define iter 1000      /* the number of iterations and it depends on both the
simulation time and the time stepsize*/
static double *curval;  /* data grid point values for next time step*/
static double *newval;  /*current values of grid points on the concentration field*/
static MPI_Comm ring;   /*mpi communicator for ring/line topology */
static int nvalues;     /* number of values in array curval and newval  */
static int npoints;     /* number of grid data points for local process        */
static int nprocs;      /* number of processes/mpi-tasks in spmd implementation */
static int rank;        /* rank of this mpi process in ring/line topology       */
static int left;        /* rank of left mpi process in ring/line topology       */
static int right;       /* rank of right mpi process in ring/line topology      */
static double result[N]; /*this array is to store the elements of a column which you
take from the computed concentration field*/

static void
compute(void)  /*
{
```

```
   PART   F

}


int
main(int argc, char *argv[])
{
   int dims[1], periods[1], coords[1], neighbour[1];
   int i,j,k, offset;
   MPI_Status status;

   /* initialize mpi,get the environment and create ring topology */


   PART   A


   /* compute data decomposition and dynamic memory allocation of arrays*/


   PART   B


   /*Initialization of array elements */

   PART   C


   for(i=0;i<iter;i++)   compute();


   /*Take a column from the computed concentration field and then collect the
result into processor 0 and print out*/

   PART   D

     /*Free the allocated memory and terminate MPI*/


   PART   E

   exit(EXIT_SUCCESS);
}
```

b) The scalability measurement of the "wave.c"

Here is the datasize for the scalability measurement of the "wave.c":

```
mpirun -np p scwave -s 1 -n 100000 -t 40 -d 0.5 -v 0
mpirun -np p scwave -s 1 -n 200000 -t 40 -d 0.5 -v 0
mpirun -np p scwave -s 1 -n 400000 -t 40 -d 0.5 -v 0
```

As usual, the number of processors should be 1,2,4,8 processors (and maybe also 16 if your measurements is done on the OW cluster).

Detailed theoretical explanation of the assignments will be available in the lab sessions.

5. Fifth lab session (Fourth assignment)

a) The template of the fourth assignment

Below is an almost empty template program for the fourth assignment: Jacobi iteration. Actually it will be more convenient if you use your program for the time-dependent diffusion as the template. The only great difference is that the number of iteration is not predefined anymore and you have to implement a stopping criterion in the Jacobi iteration.

This is the template program for the Jacobi iteration. You need to complete the missing codes(PART A,B,C,D,E,F,G).In many aspects you can learn from your program for the time dependent diffusion (i.e., the third assignment).

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#ifndef EXIT_FAILURE
#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0
#endif
#define TRUE 1
#define FALSE 0
#define N 20            /*the number of grid data points in each row or column*/
#define sigma 1e-6      /*the stopping tolerance*/
#define niter 4
static double *curval;   /*the estimated concentration values of grid points in the
current iteration */
static double *newval;   /*the estimated concentration values of grid points in the
next iteration */
static MPI_Comm ring;    /*mpi communicator for ring/line topology */
```

```
static int nvalues;      /* number of rows in array curval and newval  */
static int npoints;      /* number of grid data rows for local process         */
static int nprocs;       /* number of processes/mpi-tasks in spmd implementation */
static int rank;         /* rank of this mpi process in ring/line topology       */
static int left;         /* rank of left mpi process in ring/line topology       */
static int right;        /* rank of right mpi process in ring/line topology      */
static double result[N];  /*this array is to store the elements of a column which
you take from the computed concentration field*/
static double delta, maxdif;  /*delta is the local maximum difference between two
iterations, maxdif is the global maximum difference*/

static void
compute(void)
{

   PART   G

}


int
main(int argc, char *argv[])
{
   int dims[1], periods[1], coords[1], neighbour[1];
   int i,j,k, offset;
   long  iter;       /*the number of iterations*/
   MPI_Status status;


   /* initialize mpi,get the environment and create ring topology */


   PART   A


   /* compute data decomposition and dynamic memory allocation of arrays*/


   PART   B


   /*Initialization of array elements */

   PART   C
```

```
    iter=0;


   /*Implement the stopping criterion. You should use MPI_Allreduce to calculate
the global maximum difference between two iterations*/


   PART  D

   /*Print out the number of iterations. Take a column from the computed
concentration field and then collect the result into processor 0 and print out*/


   PART  E

    /*Free the allocated memory and terminate MPI*/


   PART  F

   exit(EXIT_SUCCESS);
}
```

b) About the scalability measurement of the third assignment

The suggested datasize for the scalability measurement of the third assignment(i.e., time-dependent diffusion) is N=128 and N=256 (maybe also N=512). As usual, the number of processors should be 1,2,4,8. The measurement should be done on the Lisa cluster.

c) Additional (optional) assignment

There will be an additional assignment (SOR iteration) available in about 7 days. SOR iteration method is a very efficient method and it will be very useful for your future research. This additional assignment is not compulsory, but it will give an opportunity for those who want to achieve high score(e.g., 9.0 or even 9.5).

6. Sixth lab session (Fifth assignment)

There is no empty template for the fifth assignment. It will be convenient if you use your program for the Jacobi iteration as a template.

a) Indications of Gauss-Seidel iteration

For the Gauss-Seidel iteration, we would like to give two indications:

(1) the red-black points should be judged in the global data point matrix, rather in the local matrix. This means you have to use the "offset" variable.

(2) there is a pseudo code of Gauss-Seidel iteration in the handout.

b) Additional (optional) assignments

In the attached file(Newassign.pdf) you can find two additional assignments : SOR iteration and DLA (diffusion limited growth). The first one (i.e., SOR iteration) is to help increase your score. We hope that many of you will try to accomplish it. The second one (i.e., DLA) is a very challenging problem and it is only for really ambitious students.

c) About the explanations and handout

As you may have noticed, there are explanations and handout materials available in the lab sessions. These shallow explanations and handout are only for helping you to accomplish the assignments. We hope that you will not mix them up with those nice stuffs (including those valuable notebooks) which teacher Walter Hoffmann taught in the lectures.

7. Additional Information

a) Scalability measurements of the third, fourth and fifth assignments

For time-dependent diffusion, the recommended datasize is N=128 and N=256. The timestep should be small enough (for example, 0.000001) to make the computation stable.

For Jacobi and Gauss-Seidel iterations, the recommended datasize is N=128 and N=256. The recommended tolerance is 1e-6. The measurements should be done on 1,2,4,8 processors of the Lisa Cluster.

b) The convergence behavior

For Jacobi and Gauss-Seidel iterations, you need to test the convergence behavior which means that you need to plot graphs about

(1) the number of iterations as a function of the stopping condition.
(2) the maximum (or mean) error as a function of the stopping condition.

You can find example graphs in the handout. The datasize can be N=20 or 40. If you can manage to make a comparison of the convergence behaviors of the Jacobi iteration and Gauss-Seidel iteration (maybe also SOR iteration), it will be really great.

c) Additional scores

You can get some additional scores under these situations.

(1) Good fresh ideas in programming and clever implementation (e.g., clever formula of data decomposition). Please do not forget to mention these for our notice in the lab report.

(2) Additional work. For example, some students did both the strip-wise decomposition and block-wise decomposition in the third assignment. If both the programs work correctly, this surely deserve some additional score. We also noted that some students have done part of the SOR iteration. It is really good.

(3) Working alone. Because working alone means more work than those who are working in pairs.

d) Data analysis

For those of you who do not know how to analyze the data using graphs, please ask in the lab. We will teach you how to use Mathematica to analyze the data.

e) The grading standard

Although still under discussion, the outline of the grade distribution of the lab work is as follows:

Programming and correctness (60%)
Scalability measurement (including convergence) and analysis (20%)
Lab report description and answering the rest questions (20%)