MPI – Message Passing Interface

Rutine de management pentru comunicator si grup

Topologii virtuale

MPI – Grupuri si comunicatoare

Grupuri si comunicatoare

- Un grup este o multime ordonata de procese. Fiecare proces dintr-un grup este asociat cu un rank unic. Valorile de ordine (rank) pornesc de la 0 si pana la N-1, N fiind numarul de procese dintr-un grup. In MPI, un grup este reprezentat in memoria sistemului ca un obiect, accesibil programatorului printr-un handler. Un grup este intotdeauna asociat cu un comunicator.
- Un comunicator cuprinde un grup de procese care pot comunica unul cu celalalt. Toate mesajele MPI trebuie sa specifice un comunicator. In cel mai simplu sens, un comunicator este un tag suplimentar care trebuie inclus cu apelurile MPI. La fel ca si grupurile, comunicatorii sunt reprezentati in interiorul memoriei sistemului ca obiecte accesibile programatorului prin intermediul unor handlere. De exemplu, handlerul pentru comunicatorul care cuprinde toate task-urile este MPI_COMM_WORLD.
- Din punctul de vedere al programatorului, un grup si un comunicator sunt acelasi lucru. Rutinele de grup sunt folosite in primul rand pentru a specifica procesele care trebuie folosite pentru a construi un comunicator.

Scopurile principale pentru obiecte de tip grup si comunicator

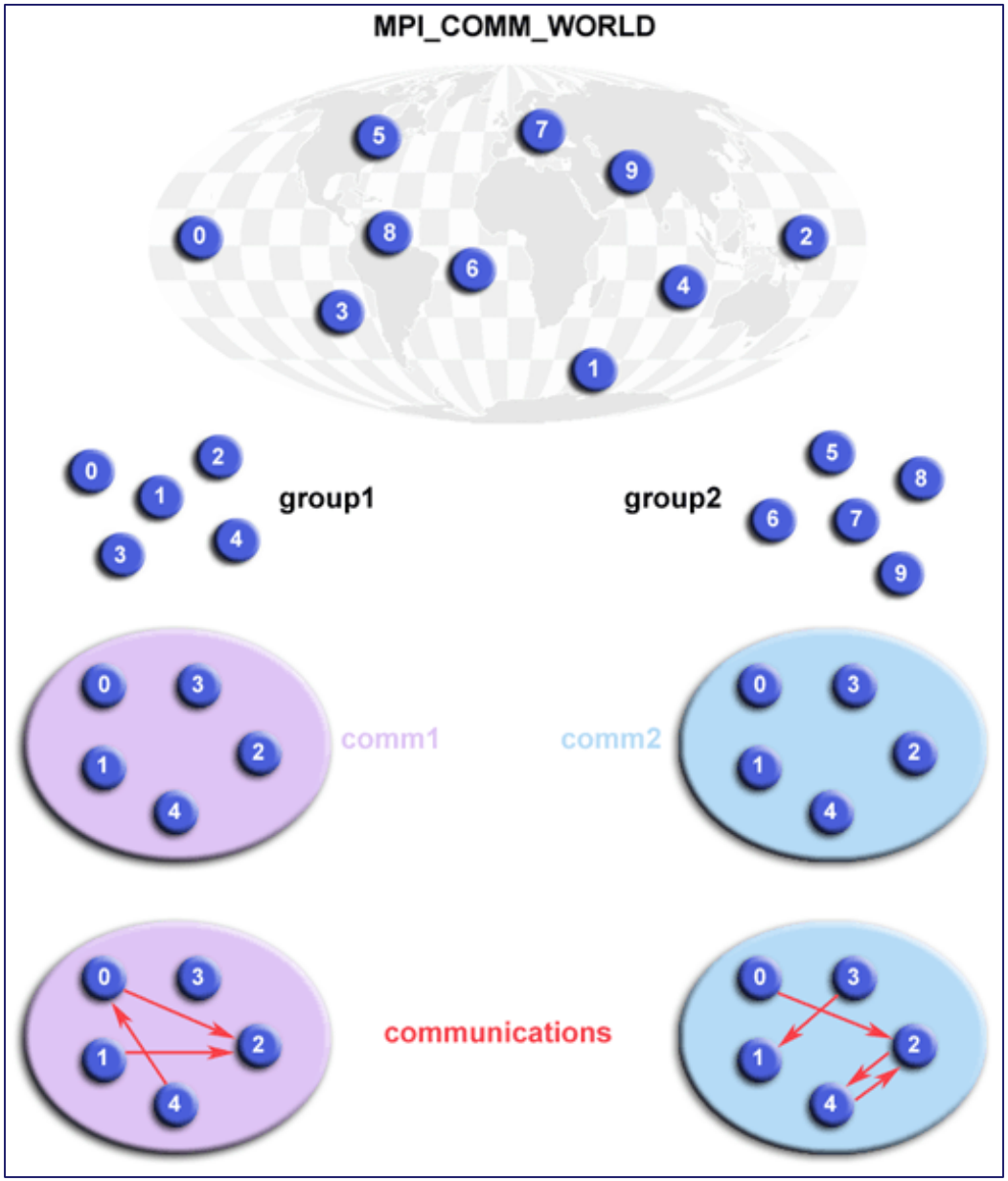
- Permit organizarea task-urilor in grupuri de task-uri;
- Permit operatiile de comunicare colectiva intre o submultime de task-uri aflate in legatura;
- Furnizeaza baza pentru implementarea topologiilor virtuale;
- Furnizeaza baza pentru comunicatia in siguranta.

MPI – Grupuri si comunicatoare

Consideratii si restrictii privind programarea

- Grupurile si comunicatoarele sunt dinamice: pot fi create si distruse in timpul executiei programului;
- Procesele pot fi in mai mult de un grup/comunicator; ele au un rank unic in fiecare grup/comunicator.
- MPI pune la dispozitie peste 40 de rutine legate de grupuri, comunicatoare si topologii virtuale.
- Utilizare tipica:
  - Extragerea unui handler de grup global din **MPI_COMM_WORLD** folosind **MPI_Comm_group**;
  - Formarea unui nou grup ca o submultime a grupului global folosind **MPI_Group_incl**;
  - Crearea unui nou comunicator pentru noul grup folosind **MPI_Group_incl**;
  - Determinarea noului ***rank*** in noul comunicator folosind **MPI_Comm_rank**;
  - Realizarea de comunicatii folosind orice rutina care transmite mesaje MPI;
  - Cand terminam, se elibereaza noul comunicator sau grup (optional) folosind **MPI_Comm_free** si **MPI_Group_free**;

MPI – Grupuri si comunicatoare



MPI – Grupuri si comunicatoare

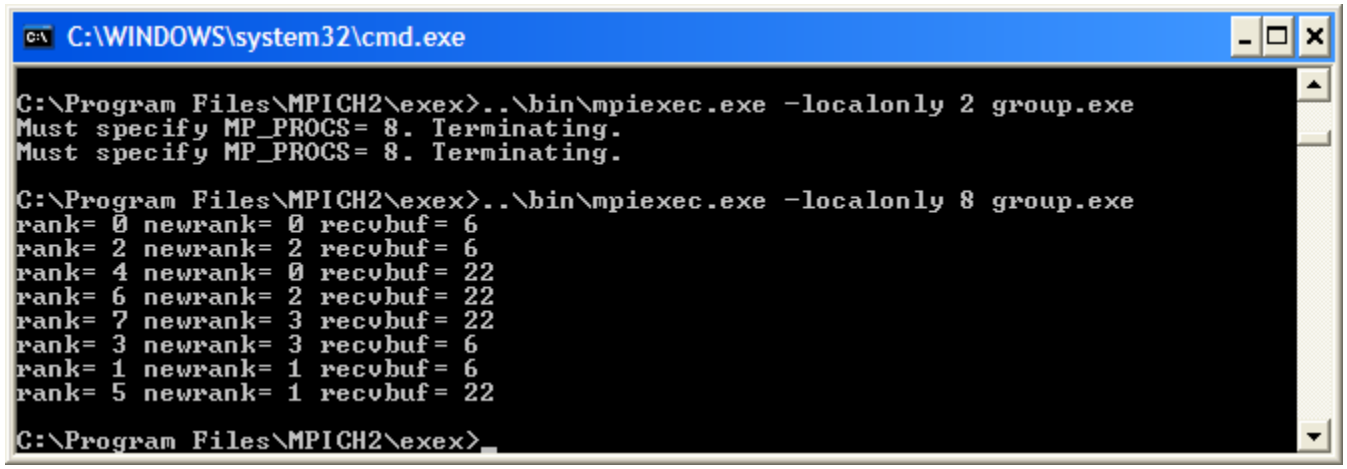
Grupuri si comunicatoare: exemplu

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

int main(argc,argv)
int argc;
char *argv[];
{
    int         rank, new_rank, sendbuf, recvbuf, numtasks;
    int         ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group   orig_group, new_group;
    MPI_Comm    new_comm;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    if (numtasks != NPROCS)
    {
        printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
        MPI_Finalize();
        exit(0);
    }
    sendbuf = rank;
    /* Extract the original group handle */
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

MPI – Grupuri si comunicatoare

Grupuri si comunicatoare: exemplu (continuare)

```
    /* Divide tasks into two distinct groups based upon rank */
    if (rank < NPROCS/2)
    {
        MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
    }
    else
    {
        MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
    }
    /* Create new new communicator and then perform collective communications */
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

    MPI_Group_rank (new_group, &new_rank);
    printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

    MPI_Finalize();
}
```



MPI – Topologii virtuale

Topologii virtuale

- O topologie virtuala descrie ordonarea proceselor MPI intr-o forma geometrica
- Cele doua mari tipuri de topologii suportate de MPI sunt Cartezian (grid) si Graph
- Topologiile MPI sunt virtuale – poate sa nu existe nici o legatura intre structura fizica a masinii paralele sau clusterului folosit si topologia proceselor
- Topologiile virtuale sunt construite folosind comunicatori MPI si grupuri
- Topologiile virtuale trebuie sa fie programate de dezvoltatorul aplicatiei

La ce folosesc topologiile virtuale

- Pot fi utile pentru aplicatii cu paternuri de comunicatie specifile – paternuri care se potrivesc cu o structura topologica MPI
- De exemplu, o topologie Carteziana poate sa fie convenabil de utilizat pentru o aplicatie care presupune comunicarea in 4 directii cu cei mai apropiati vecini pentru date structurate ca un grid
- Unele arhitecturi hardware pot impune penalitati pentru comunicarea intre noduri succesive departate;
- O implementare particulara poate optimiza structura topologica a proceselor pe baza caracteristicilor fizice ale unei anumite masini paralele.
- Structurarea proceselor intr-o topologie virtuala MPI e dependenta de implementarea MPI si poate fi cu totul ignorata.

MPI – Topologii virtuale

Topologii virtuale: exemplu – Topologie carteziana

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

MPI – Message Passing Interface

Topologii virtuale: exemplu

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP    0
#define DOWN  1
#define LEFT  2
#define RIGHT 3

int main(argc,argv)
int argc;
char *argv[];
{
    int numtasks, rank, source, dest, outbuf, i, tag=1,
        inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
        nbrs[4], dims[2]={4,4},
        periods[2]={0,0}, reorder=0, coords[2];

    MPI_Request reqs[8];
    MPI_Status stats[8];
    MPI_Comm cartcomm;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    if (numtasks == SIZE) {
        MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
        MPI_Comm_rank(cartcomm, &rank);
        MPI_Cart_coords(cartcomm, rank, 2, coords);
        MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
        MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

        printf("rank= %d coords= %d %d  neighbors(u,d,l,r)= %d %d %d %d\n",
             rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
             nbrs[RIGHT]);
```

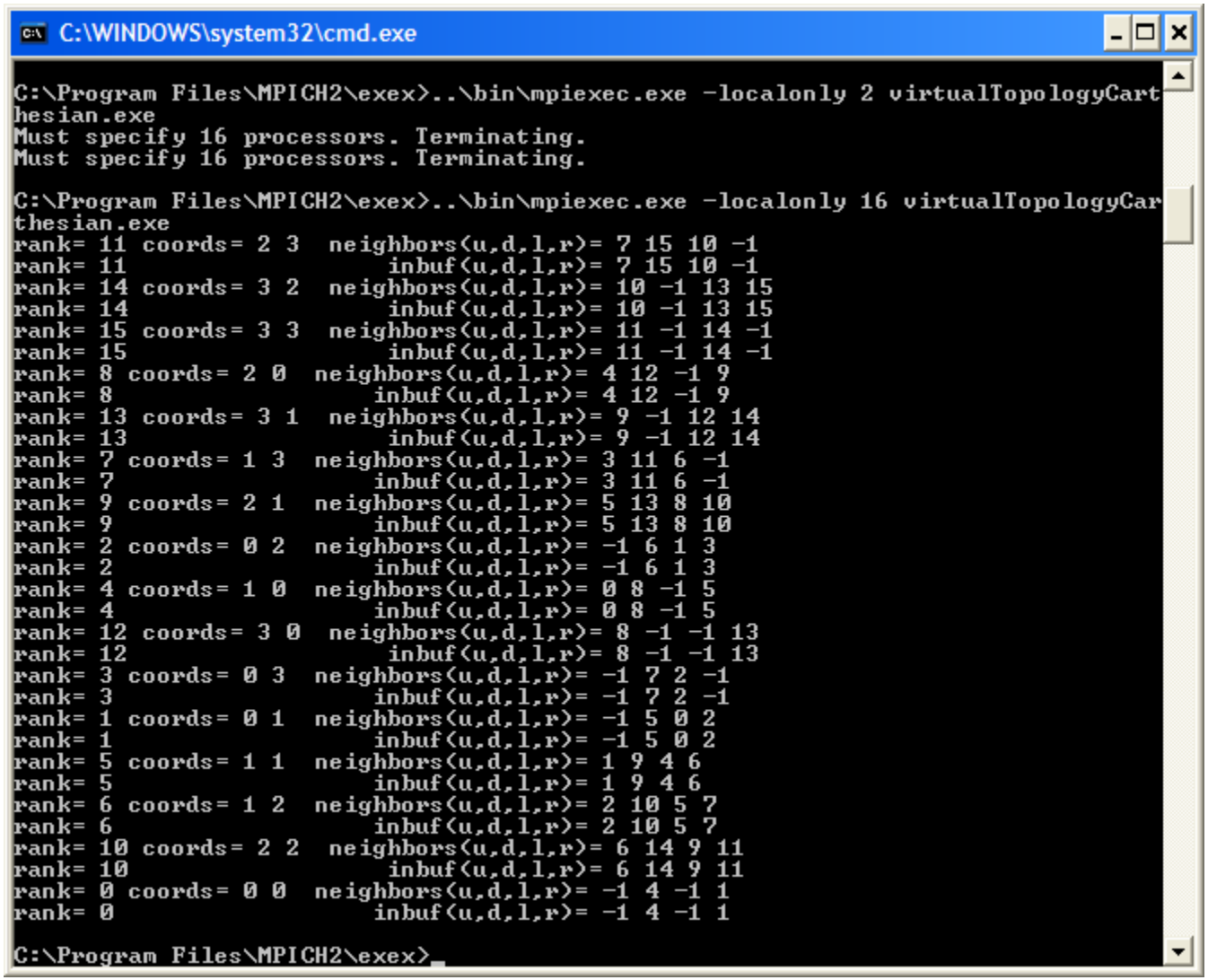
MPI – Topologii virtuale

Topologii virtuale: exemplu (continuare)

```
        outbuf = rank;

        for (i=0; i<4; i++)
        {
            dest = nbrs[i];
            source = nbrs[i];
            MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
                    MPI_COMM_WORLD, &reqs[i]);
            MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                    MPI_COMM_WORLD, &reqs[i+4]);
        }

        MPI_Waitall(8, reqs, stats);

        printf("rank= %d                    inbuf(u,d,l,r)= %d %d %d %d\n",
         rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
    }
    else
        printf("Must specify %d processors. Terminating.\n",SIZE);

    MPI_Finalize();
}
```

MPI – Topologii virtuale

Topologii virtuale: exemplu: rezultate



MPI – Message Passing Interface

Aplicatie

Rezolvarea ecuatiei caldurii intr-un domeniu rectangular, folosind o solutie paralela

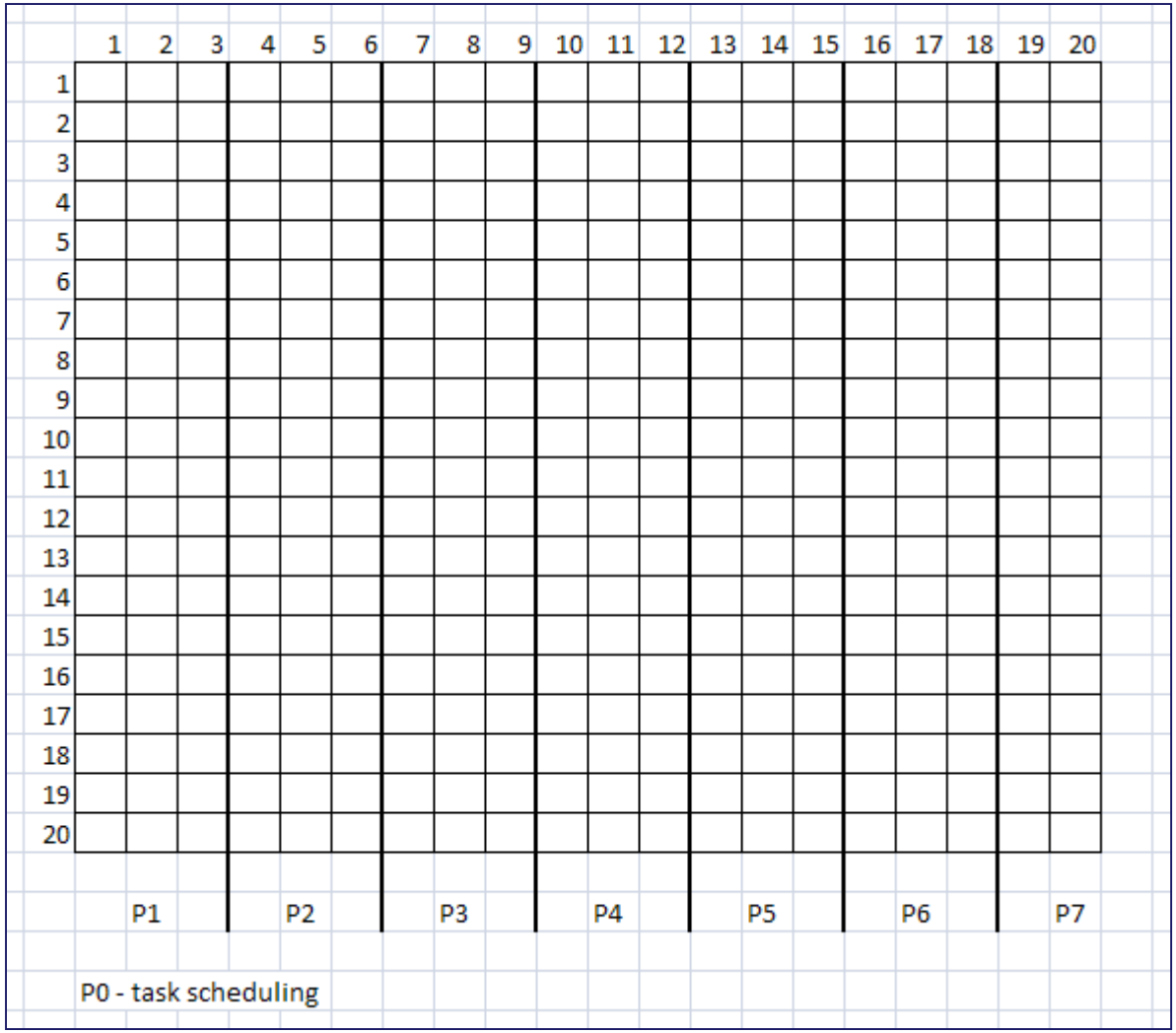
MPI – Message Passing Interface

Exemplu aplicatie pentru rezolvarea ecuatiei caldurii intr-un domeniu rectangular, folosind o solutie paralela

- Se implementeaza ca un model SPMD
- Intreaga matrice este impartita si distribuita in submatrice (dupa coloane) fiecarui proces. Fiecare proces va detine o parte din matricea totala
- Se determina dependintele de date:
  - Elementele din interior care apartin unui task sunt independente de celelelalte elemente;
  - Elementele de frontiera depind atat de elementele interioare task-ului cat si de datele unui vecin, si va fi nevoie de comunicatie pentru a colecta rezultatele de la toate taskurile
  - Procesul Master trimite informatia initiala task-urilor worker si apoi asteapta sa colecteze rezultatele de la toati lucratorii
  - Procesele Worker calculeaza solutia intr-un numar fix de pasi de timp, comunicand dupa cum e necesar cu procesele vecine.

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D



In implementarea acestei solutii pentru problema difuziei termice, se rezolva echivalentul unei probleme matriceale lineare printr-o metoda interativa. La fiecare iteratie, se aproximeaza fiecare element al matricei in functie de valorile celor 4 vecini (sus, jos, stanga, dreapta); in cazul elementelor aflate in preajma frontierei, se iau in calcul valorile de pe frontiera (care vor fi fixate); astfel, dupa mai multe iteratii, solutia va converge spre solutia ecuatiei partiale cu derivate partiale de ordinul doi care modeleaza problema difuziei termice.

In implementarea paralela a solutiei, problema (matricea) se divide in n-1 sectoare (n fiind numarul de procese) fiecare proces trimitand si primind informatia de la vecini.

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D

```
/******************************************************************************
 * FILE: mpi_heat2D.c
 * DESCRIPTIONS:
 *   HEAT2D Example - Parallelized C Version
 *   This example is based on a simplified two-dimensional heat
 *   equation domain decomposition.  The initial temperature is computed to be
 *   high in the middle of the domain and zero at the boundaries.  The
 *   boundaries are held at zero throughout the simulation.  During the
 *   time-stepping, an array containing two domains is used; these domains
 *   alternate between old data and new data.
 *
 *   In this parallelized version, the grid is decomposed by the master
 *   process and then distributed by rows to the worker processes.  At each
 *   time step, worker processes must exchange border data with neighbors,
 *   because a grid point's current temperature depends upon it's previous
 *   time step value plus the values of the neighboring grid points.  Upon
 *   completion of all time steps, the worker processes return their results
 *   to the master process.
 *
 *   Two data files are produced: an initial data set and a final data set.
 *   An X graphic of these two states displays after all calculations have
 *   completed.
 * AUTHOR: Blaise Barney - adapted from D. Turner's serial C version. Converted
 *   to MPI: George L. Gusciora (1/95)
 * LAST REVISED: 12/11/09 Blaise Barney
 ******************************************************************************/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define NXPROB      20                 /* x dimension of problem grid */
#define NYPROB      20                 /* y dimension of problem grid */
#define STEPS       500                /* number of time steps */
#define MAXWORKER   8                  /* maximum number of worker tasks */
#define MINWORKER   3                  /* minimum number of worker tasks */
#define BEGIN       1                  /* message tag */
#define LTAG        2                  /* message tag */
#define RTAG        3                  /* message tag */
#define NONE        0                  /* indicates no neighbor */
#define DONE        4                  /* message tag */
#define MASTER      0                  /* taskid of first process */

struct Parms {
  float cx;
  float cy;
} parms = {0.1, 0.1};

int main (int argc, char *argv[])
{
void inidat(), prtdat(), update();
float  u[2][NXPROB][NYPROB];        /* array for grid */
int taskid,                     /* this task's unique id */
    numworkers,                 /* number of worker processes */
    numtasks,                   /* number of tasks */
    averow,rows,offset,extra,   /* for sending rows of data */
    dest, source,               /* to - from for message send-receive */
    left,right,           /* neighbor tasks */
    msgtype,                    /* for message types */
    rc,start,end,               /* misc */
    i,ix,iy,iz,it;              /* loop variables */
MPI_Status status;
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
/* First, find out my taskid and how many tasks are running */
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   numworkers = numtasks-1;

   if (taskid == MASTER) {
      /************************ master code *******************************/
      /* Check if numworkers is within range - quit if not */
      if ((numworkers > MAXWORKER) || (numworkers < MINWORKER)) {
         printf("ERROR: the number of tasks must be between %d and %d.\n",
                MINWORKER+1,MAXWORKER+1);
         printf("Quitting...\n");
         MPI_Abort(MPI_COMM_WORLD, rc);
         exit(1);
         }
      printf ("Starting mpi_heat2D with %d worker tasks.\n", numworkers);

      /* Initialize grid */
      printf("Grid size: X= %d  Y= %d  Time steps= %d\n",NXPROB,NYPROB,STEPS);
      printf("Initializing grid and writing initial.dat file...\n");
      inidat(NXPROB, NYPROB, u);
      prtdat(NXPROB, NYPROB, u, "initial.dat");

      /* Distribute work to workers.  Must first figure out how many rows to */
      /* send and what to do with extra rows.  */
      averow = NXPROB/numworkers;
      extra = NXPROB%numworkers;
      offset = 0;
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
for (i=1; i<=numworkers; i++)
{
   rows = (i <= extra) ? averow+1 : averow;
   /* Tell each worker who its neighbors are, since they must exchange */
   /* data with each other. */
   if (i == 1)
      left = NONE;
   else
      left = i - 1;
   if (i == numworkers)
      right = NONE;
   else
      right = i + 1;
   /*  Now send startup information to each worker  */
   dest = i;
   MPI_Send(&offset, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
   MPI_Send(&left, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
   MPI_Send(&right, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
   MPI_Send(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, dest, BEGIN,
            MPI_COMM_WORLD);
   printf("Sent to task %d: rows= %d offset= %d ",dest,rows,offset);
   printf("left= %d right= %d\n",left,right);
   offset = offset + rows;
}
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
    /* Now wait for results from all worker tasks */
    for (i=1; i<=numworkers; i++)
    {
       source = i;
       msgtype = DONE;
       MPI_Recv(&offset, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD,
                &status);
       MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
       MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source,
                msgtype, MPI_COMM_WORLD, &status);
    }

    /* Write final output, call X graph and finalize MPI */
    printf("Writing final.dat file and generating graph...\n");
    prtdat(NXPROB, NYPROB, &u[0][0][0], "final.dat");
    printf("Click on MORE button to view initial/final states.\n");
    printf("Click on EXIT button to quit program.\n");

    MPI_Finalize();
}   /* End of master code */
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
/************************** workers code **********************************/
if (taskid != MASTER)
{
   /* Initialize everything - including the borders - to zero */
   for (iz=0; iz<2; iz++)
      for (ix=0; ix<NXPROB; ix++)
         for (iy=0; iy<NYPROB; iy++)
            u[iz][ix][iy] = 0.0;

   /* Receive my offset, rows, neighbors and grid partition from master */
   source = MASTER;
   msgtype = BEGIN;
   MPI_Recv(&offset, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&left, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&right, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source, msgtype,
            MPI_COMM_WORLD, &status);

   /* Determine border elements.  Need to consider first and last columns. */
   /* Obviously, row 0 can't exchange with row 0-1.  Likewise, the last */
   /* row can't exchange with last+1.  */
   start=offset;
   end=offset+rows-1;
   if (offset==0)
      start=1;
   if ((offset+rows)==NXPROB)
      end--;
   printf("task=%d  start=%d  end=%d\n",taskid,start,end);
```

MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
      /* Begin doing STEPS iterations.  Must communicate border rows with */
      /* neighbors.  If I have the first or last grid row, then I only need */
      /*  to  communicate with one neighbor  */
      printf("Task %d received work. Beginning time steps...\n",taskid);
      iz = 0;
      for (it = 1; it <= STEPS; it++)
      {
         if (left != NONE)
         {
            MPI_Send(&u[iz][offset][0], NYPROB, MPI_FLOAT, left,
                     RTAG, MPI_COMM_WORLD);
            source = left;
            msgtype = LTAG;
            MPI_Recv(&u[iz][offset-1][0], NYPROB, MPI_FLOAT, source,
                      msgtype, MPI_COMM_WORLD, &status);
         }
         if (right != NONE)
         {
            MPI_Send(&u[iz][offset+rows-1][0], NYPROB, MPI_FLOAT, right,
                      LTAG, MPI_COMM_WORLD);
            source = right;
            msgtype = RTAG;
            MPI_Recv(&u[iz][offset+rows][0], NYPROB, MPI_FLOAT, source, msgtype,
                      MPI_COMM_WORLD, &status);
         }
         /* Now call update to update the value of grid points */
         update(start,end,NYPROB,&u[iz][0][0],&u[1-iz][0][0]);
         iz = 1 - iz;
      }
      /* Finally, send my portion of final results back to master */
      MPI_Send(&offset, 1, MPI_INT, MASTER, DONE, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, MASTER, DONE, MPI_COMM_WORLD);
      MPI_Send(&u[iz][offset][0], rows*NYPROB, MPI_FLOAT, MASTER, DONE,
               MPI_COMM_WORLD);
      MPI_Finalize();
   }
}
```

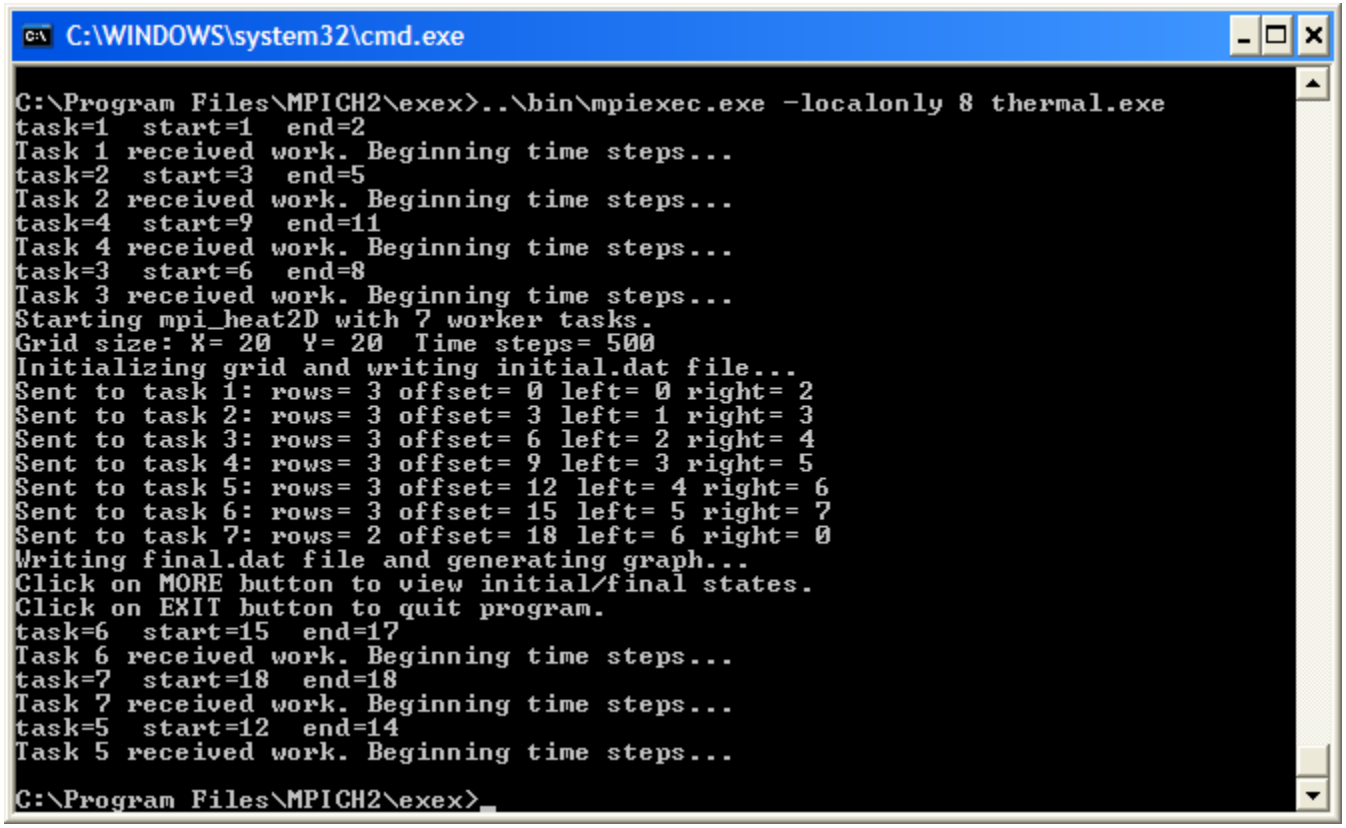
MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare)

```
/*****************************************************************************
 *  subroutine update
 *****************************************************************************/
void update(int start, int end, int ny, float *u1, float *u2)
{
   int ix, iy;
   for (ix = start; ix <= end; ix++)
      for (iy = 1; iy <= ny-2; iy++)
         *(u2+ix*ny+iy) = *(u1+ix*ny+iy)  +
                          parms.cx * (*(u1+(ix+1)*ny+iy) +
                          *(u1+(ix-1)*ny+iy) -
                          2.0 * *(u1+ix*ny+iy)) +
                          parms.cy * (*(u1+ix*ny+iy+1) +
                         *(u1+ix*ny+iy-1) -
                          2.0 * *(u1+ix*ny+iy));
}

/*****************************************************************************
 *  subroutine inidat
 *****************************************************************************/
void inidat(int nx, int ny, float *u) {
int ix, iy;

for (ix = 0; ix <= nx-1; ix++)
  for (iy = 0; iy <= ny-1; iy++)
     *(u+ix*ny+iy) = (float)(ix * (nx - ix - 1) * iy * (ny - iy - 1));
}

/*****************************************************************************
 * subroutine prtdat
 *****************************************************************************/
void prtdat(int nx, int ny, float *u1, char *fnam) {
int ix, iy;
FILE *fp;

fp = fopen(fnam, "w");
for (iy = ny-1; iy >= 0; iy--) {
  for (ix = 0; ix <= nx-1; ix++) {
    fprintf(fp, "%6.1f", *(u1+ix*ny+iy));
    if (ix != nx-1)
      fprintf(fp, " ");
    else
      fprintf(fp, "\n");
    }
  }
fclose(fp);
}
```

MPI – Message Passing Interface

<u>Exemplu aplicatie mpi_heat2D (continuare)</u>: executie aplicatie



MPI – Message Passing Interface

Exemplu aplicatie mpi_heat2D (continuare):
Comparatie distributie initiala si distributie finala a temperaturii