Lab 6e - Programmazione ibrida MPI+openMP

https://elly2023.smfi.unipr.it/mod/page/view.php?id=5405

OBIETTIVO
Programmazione ibrida mpi+openMP di Heat

Cos'è la programmazione ibrida
Nelle moderne architetture hardware ibride (multicore + multinodo + GPU) è necessario l'utilizzo contemporaneo di diversi modelli di programmazione; MPI per la comunicazione tra i nodi, mentre all'interno del nodo conviene utilizzare openMP (MPI+openMP) oppure, MPI+Cuda se è disponibile una GPU.
Nei casi più complessi MPI+openMP+CUDA. In generale questa programmazione ibrida viene denominata MPI+X.

Nella programmazione ibrida (MPI + openMP) viene generalmente attivato un solo task MPI per nodo (o per socket) il quale si occupa della comunicazione con gli altri task, mentre il calcolo all'interno del nodo (o del socket) viene parallelizzato con openMP. Questa architettura consente di sfruttare la scalabilità multimodo di MPI ma riducendo al minimo l'overhead di comunicazione.

Vedi: https://www.hpcwire.com/2014/07/16/compilers-mpix/

Attività svolte

Dopo aver creato la directory di lavoro ~/HPC2324/mpi+omp/heat, ho copiato al suo interno le diverse versioni del programma heat: "heat.c", "omp_heat.c" e "mpi_heat.c".

Dalle versioni MPI e OpenMP ho realizzato la versione ibrida mpi+openMP, dove partendo dalla versione MPI ho:

- Aggiunto la funzione "#pragma omp parallel for private (i)" alla funzione Jacobi_Iterator_CPU nel file mpi_heat.h, per distribuire su più thread le iterazioni sulle righe
- Modificato la stampa per aggiungere i nuovi dati

mpi+omp_heat.c

```
/*
HEAT - versione ibrida mpi+openMP


mpi+omp_heat.c


module load gnu openmpi
mpicc -O2 mpi+omp_heat.c -o mpi+omp_heat


mpirun mpi+omp_heat -h
mpirun mpi+omp_heat -r 24 -c 24 -s 4
*/


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```c
#include <mpi.h>

int WNX = 256;         // --- Number of discretization points along the x axis
int WNY = 256;         // --- Number of discretization points along the y axis
int MAX_ITER = 1000;  // --- Number of Jacobi iterations
int NX,NY;

#include "mpi_heat.h"



// variabili globali per MPI
int mpi_rank=0, mpi_size=0;
int prev_rank=0, next_rank=0;
int tag = 999;
MPI_Status status;
char hostname[MPI_MAX_PROCESSOR_NAME]; int namelen;

float *h_T_new;
float *h_T_old;
float *h_T_temp;
float *h_T_whole;

/********/
/* MAIN */
/********/
int main(int argc, char **argv)
{

   int iter,i,j;

   double t1, t2;

   options(argc, argv);         /* optarg management */

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
   MPI_Get_processor_name(hostname,&namelen);

   // scomposizione di dominio:
   // dividiamo le righe tra i rank MPI
   prev_rank = (mpi_rank-1+mpi_size) % mpi_size;
   next_rank = (mpi_rank+1) % mpi_size;
   NX =  WNX;  // Local NX: numero colonne per rank
   NY =  WNY/mpi_size+2;  // Local NY: numero righe   per rank

   // stampa tutti i dati della simulazione
   fprintf (stderr,"# WNX:%d, WNY:%d, MAX_ITER:%d, MPI_RANK:%d, MPI_SIZE:%d, NX:%d, NY:%d, %s \n", WNX,
WNY, MAX_ITER, mpi_rank, mpi_size, NX, NY, hostname);

   // valori temperatura corrente
   h_T_new = (float *)calloc(NX * NY, sizeof(float));
   // valori temperatura precedente
   h_T_old = (float *)calloc(NX * NY, sizeof(float));
   if (mpi_rank == 0)  h_T_whole  = (float *)calloc(WNX * WNY, sizeof(float));



   t1=MPI_Wtime();

   for(iter=0; iter<MAX_ITER; iter=iter+1)
   {

       // Init_center(h_T_old,  NX, NY);
       // Init_left(h_T_old,    NX, NY);
       if (mpi_rank==0)   Init_top(h_T_old,     NX, NY);
       // copy_rows(h_T_old, NX, NY);
```

```
       // copy_cols(h_T_old, NX, NY);

       // scambio delle righe di bordo
       // downward
       MPI_Sendrecv( &(h_T_old[NX*(NY-2)]), NX, MPI_FLOAT, next_rank, tag, //send
               &(h_T_old[NX*0]), NX, MPI_FLOAT, prev_rank, tag, //recv
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       // upward
       MPI_Sendrecv( &(h_T_old[NX*1]), NX, MPI_FLOAT, prev_rank, tag, //send
               &(h_T_old[NX*(NY-1)]), NX, MPI_FLOAT, next_rank, tag, //recv
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);


       Jacobi_Iterator_CPU(h_T_old, h_T_new, NX, NY);

       h_T_temp=h_T_new;
       h_T_new=h_T_old;
       h_T_old=h_T_temp;


   }


   t2=MPI_Wtime();

   // WNX/WNY = Number of discretization points along the x/y axis
   //if(mpi_rank == 0)   fprintf(stderr,"MPI, %d, %d, %d, %d,  %f\n", mpi_size,  WNX, WNY, MAX_ITER, t2-
t1);
   if(mpi_rank == 0)   fprintf(stderr,"MPI+OMP, %d, %d, %d, %d, %d,  %f\n", mpi_size,
omp_get_max_threads(),  WNX, WNY, MAX_ITER, t2-t1);


   MPI_Gather(&(h_T_old[NX]), NX*(NY-2), MPI_FLOAT, h_T_whole,  NX*(NY-2), MPI_FLOAT,  0,
MPI_COMM_WORLD);

   if (mpi_rank == 0) print_colormap(h_T_whole,WNX,WNY);


   free(h_T_new);
   free(h_T_old);
   if (mpi_rank == 0 ) free(h_T_whole);


   MPI_Finalize();


   return 0;
}
```

mpi_heat.h

```
/*
mpi_heat.h
*/



/*********************************/
/* JACOBI ITERATION FUNCTION - CPU */
/*********************************/
void Jacobi_Iterator_CPU(float * __restrict T, float * __restrict T_new, const int NX, const int NY)
{
int i,j;


        // --- Only update "interior" (not boundary) node points
     #pragma omp parallel for private (i)
        for(j=1; j<NY-1; j++)
            for(i=1; i<NX-1; i++) {
                float T_E = T[(i+1) + NX*j];
                float T_W = T[(i-1) + NX*j];
                float T_N = T[i + NX*(j+1)];
                float T_S = T[i + NX*(j-1)];
                T_new[NX*j + i] = 0.25*(T_E + T_W + T_N + T_S);
```

```
            }
}


/**********************************/
/* JACOBI ITERATION ITERNAL FUNCTION - CPU */
/**********************************/
void Jacobi_Iterator_CPU_internal(float * __restrict T, float * __restrict T_new, const int NX, const int
NY)
{
int i,j;


        // --- Righe dalla 2 alla penultima
        for(j=2; j<NY-2; j++)
            for(i=1; i<NX-1; i++) {
                float T_E = T[(i+1) + NX*j];
                float T_W = T[(i-1) + NX*j];
                float T_N = T[i + NX*(j+1)];
                float T_S = T[i + NX*(j-1)];
                T_new[NX*j + i] = 0.25*(T_E + T_W + T_N + T_S);
            }
}


/**********************************/
/* JACOBI ITERATION ITERNAL FUNCTION - CPU */
/**********************************/
void Jacobi_Iterator_CPU_external(float * __restrict T, float * __restrict T_new, const int NX, const int
NY)
{
int i,j;


            j=1;  // prima riga
            for(i=1; i<NX-1; i++) {
                float T_E = T[(i+1) + NX*j];
                float T_W = T[(i-1) + NX*j];
                float T_N = T[i + NX*(j+1)];
                float T_S = T[i + NX*(j-1)];
                T_new[NX*j + i] = 0.25*(T_E + T_W + T_N + T_S);
            }


            j=NY-2;  // ultima riga
            for(i=1; i<NX-1; i++) {
                float T_E = T[(i+1) + NX*j];
                float T_W = T[(i-1) + NX*j];
                float T_N = T[i + NX*(j+1)];
                float T_S = T[i + NX*(j-1)];
                T_new[NX*j + i] = 0.25*(T_E + T_W + T_N + T_S);
            }
}


/*********************************/
/* TEMPERATURE INITIALIZATION : */
/* parte centrale della griglia */
/*********************************/
void Init_center(float * __restrict h_T, const int NX, const int NY)
{
    int i,j;
    int startx=NX/2-NX/10;
    int endx=NX/2+NX/10;
    int starty=NY/2-NY/10;
    int endy=NY/2+NY/10;
//    int starty=NY/4;
//    int endy=NY-NY/4;
    for(i=startx; i<endx; i++)
        for(j=starty; j<endy; j++)
              h_T[NX*j + i] = 1.0;
}



/*********************************/
/* TEMPERATURE INITIALIZATION : */
```

```
/* bordo sinistro               */
/*******************************/
void Init_left(float * __restrict h_T, const int NX, const int NY)
{
    int i,j;
    int startx=1;
    int endx=2;
    int starty=0;
    int endy=NY-1;
    for(i=startx; i<endx; i++)
        for(j=starty; j<endy; j++)
              h_T[NX*j + i] = 1.0;
}


/*******************************/
/* TEMPERATURE INITIALIZATION : */
/* bordo alto                  */
/*******************************/
void Init_top(float * __restrict h_T, const int NX, const int NY)
{
    int i,j;
    int startx=0;
    int endx=NX-1;
    for(i=startx; i<endx; i++)
              h_T[NX + i] = 1.0;
}


/*******************************/
/* Periodic boundary conditions */
/* COPY BORDER: COLS            */
/*******************************/
void copy_cols (float * __restrict h_T, const int NX, const int NY)
{

int i;


// copy cols
  for (i = 1; i < NY-1; ++i) {
    h_T[NX*i+0]    = h_T[NX*i+NX-2];
    h_T[NX*i+NX-1] = h_T[NX*i+1];
  }
}



/*******************************/
/* Periodic boundary conditions */
/* COPY BOREDER: ROWS           */
/*******************************/
void copy_rows (float * __restrict h_T, const int NX, const int NY)
{

   memcpy(&(h_T[NX*0])      ,&(h_T[NX*(NY-2)]), NX*sizeof(float) );
   memcpy(&(h_T[NX*(NY-1)]) ,&(h_T[NX*1]),      NX*sizeof(float) );
}



/*****************************************/
/* print color map                       */
/*****************************************/

void print_colormap(float * __restrict h_T, const int NX, const int NY)
{
   int i,j;

   for (j=0; j<NY; j++){
        for (i=1; i<NX-1; i++) {
            printf("%.2f ",h_T[NX*j + i]);
        }
        printf("\n");
```

```
        }
}

/****************************************/
/* save color map  to file              */
/****************************************/

void save_colormap(float * __restrict h_T,  const int NX, const int NY)
{
   int i,j;
    FILE *fp;
    fp=fopen("colormap.dat","wb");

   for (j=0; j<NY; j++)
        for (i=0; i<NX; i++)
            fprintf(fp,"%.2f ",h_T[NX*j + i]);

   fclose (fp);
   return;
}

/****************************************/
/* Load color map  from file            */
/****************************************/

void load_colormap(float * __restrict h_T,  const int NX, const int NY)
{
   int i,j;
    FILE *fp;
    char temp[10];
    fp=fopen("colormap.dat","r+");

   for (j=0; j<NY; j++)
        for (i=0; i<NX; i++)
           { fscanf(fp, "%s ", temp ); h_T[NX*j + i]=atof(temp);}

   fclose (fp);
   return;
}

/****************************************/
/* options management                   */
/****************************************/
void options(int argc, char * argv[]) {

  int i;
   while ( (i = getopt(argc, argv, "c:r:s:h")) != -1) {
        switch (i) {
        case 'c':  WNX       = strtol(optarg, NULL, 10);  break;
        case 'r':  WNY       = strtol(optarg, NULL, 10);  break;
        case 's':  MAX_ITER = strtol(optarg, NULL, 10);  break;
        case 'h': printf ("\n%s [-c ncols] [-r nrows] [-s nsteps] [-h]\n",argv[0]); exit(1);
        default:  printf ("\n%s [-c ncols] [-r nrows] [-s nsteps] [-h]\n",argv[0]); exit(1);
        }
    }
}
```

File di output generato dallo script mpi+omp_heat.slurm

```
[martina.genovese@ui01 heat]$ more mpi+omp_heat.dat
# WNX:2048, WNY:2048, MAX_ITER:1000, MPI_RANK:0, MPI_SIZE:4, NX:2048, NY:514, wn20
# WNX:2048, WNY:2048, MAX_ITER:1000, MPI_RANK:1, MPI_SIZE:4, NX:2048, NY:514, wn20
# WNX:2048, WNY:2048, MAX_ITER:1000, MPI_RANK:3, MPI_SIZE:4, NX:2048, NY:514, wn21
# WNX:2048, WNY:2048, MAX_ITER:1000, MPI_RANK:2, MPI_SIZE:4, NX:2048, NY:514, wn21
MPI+OMP, 4, 8, 2048, 2048, 1000,  26.172231
```

File di output generato dallo script mpi_omp_heat_scaling.slurm (confronta le prestazioni delle

diverse versioni, seriali e parallele)
… (non sono riuscita ad eseguirlo perché ci mette troppo tempo a terminare) …

Infine ho eseguito il programma in Python heat_mpi_omp_scaling.py per generare il plot che compara lo scaling tra la versione MPI pura, la versione openMP e la versione ibrida con 2 nodi.
… (heat_mpi_omp_scaling.png) …