MPI Programming Assignment: Extinguishing fires

Sofia Almirante Castillo (1527718)
Pau Reig Llunell (1527816)

December 2022

1 Introduction

In order to carry out the parallelization of this program, the strategy will consist in split the matrix per rows, and distribute the smaller sections of the main surface in the different processors. Concretely, every small surface will have the same number of columns as the original matrix, but the number of rows will consist of the original number of rows divided by the number of processors chosen. In addition, we will add two extra rows at the top and bottom of every small matrix in order to update the temperature values of the surface correctly.

We have chosen this strategy since the heat propagation only depends on the adjacent cells, so with this distribution all the small structures can evolve correctly at the same time. In addition, splitting per rows instead of columns is a very intuitive decision considering the functioning of the cache memory, which is going to show a better performance if the elements are stored consecutively. It is important to emphasize that this strategy will also work when the rows of the surface are a multiple of the number of processors used.

The first modification of the code consists in the creation of the smaller structures that are going to contain parts of the original matrix initialized to 0, and also the initialization of the MPI environment and the variables needed:

```c
  float max_global_residual;
int nproc, rank;
float *miniSurface, *miniSurfaceCopy;
miniSurface = (float*)malloc(((size_t)(rows+2)*(size_t)columns/nproc)*sizeof(float));
miniSurfaceCopy = (float*)malloc(((size_t)(rows+2)*(size_t)columns/nproc)*sizeof(float));
for( i=0; i<rows/nproc + 2; i++ )
    for( j=0; j<columns; j++ ) {
      accessMat( miniSurface, i, j ) = 0.0;
      accessMat( miniSurfaceCopy, i, j ) = 0.0;
    }
```

```

 //Initialize MPI, and create in each processor a smaller matrix:
   //int nproc, rank;
   MPI_Status status;
   MPI_Request request;
   MPI_Init( &argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Once inside the time evolution loop, we first need to activate the focal points if needed. For this step, we need to convert the positions of the original surface into the new smaller structure, as follows:

```
float global_residual = 0.0f;
    float max_global_residual;
    int step;
    for( step=0; step<10; step++ )  {
      /* 4.2.1. Update heat on active focal points */

      for( i=0; i<num_focal; i++ ) {
        if ( focal[i].active != 1 ) continue;
        int x = focal[i].x;
        int y = focal[i].y;
        if ( x < 0 || x > rows-1 || y < 0 || y > columns-1 ) continue;
        for(j = 0; j < nproc; j++){
      //When updating the heat on active focal points, we want to do it only in the
    processor that contains that part of the world
            if((y >= j*rows/nproc && y < (j+1)*rows/nproc) && rank == j){
                accessMat(miniSurface, x, y) = focal[i].heat;
            }
        }
      }
```

As it happened with the previous code, now we will make use of an ancillary structure in order to update the heat values, and computing the new values making the average of the adjacent cells:

```
    /* 4.2.2. Copy values of the surface in ancillary structure (Skip borders) */
      for( i=1; i<rows/nproc-1; i++ )
        for( j=1; j<columns-1; j++ )
          accessMat( miniSurfaceCopy, i, j ) = accessMat( miniSurface, i, j );

      /* 4.2.3. Update surface values (skip borders) */
      for( i=1; i<rows/nproc-1; i++ )
        for( j=1; j<columns-1; j++ )
          accessMat( miniSurface, i, j ) = (
            accessMat( miniSurfaceCopy, i-1, j ) +
            accessMat( miniSurfaceCopy, i+1, j ) +
            accessMat( miniSurfaceCopy, i, j-1 ) +
            accessMat( miniSurfaceCopy, i, j+1 ) ) / 4;
```

An important variation appears now, as we need to communicate between processors if the value of the additional rows used to compute the heat correctly has been changed. In order to do this, every processor (but the first one) will receive the second to last row of its precedent processor, and will store it in its first one. Analogously, every processor (but the last one) will receive the second row of its next processor, and will store it in its last one.

We are using non-blocking communication for the receiving part, since the processors can send a message while waiting to receive one. After waiting for the processor to receive the desired information, it will be stored in the appropriate row.

```
 float sendrow[columns-2], recvrow[columns-2];

     if(rank < nproc-1){
       for( j=1; j<columns-1; j++ ){
         sendrow[j]=accessMat( miniSurfaceCopy, rows/nproc, j );
       }
       MPI_Irecv(&recvrow, columns-2, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD, &request);
       MPI_Send(&sendrow, columns-2, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD);
       MPI_Wait(&request, &status);
       for( j=1; j<columns-1; j++ ){
         accessMat( miniSurfaceCopy, rows/nproc+1, j ) = recvrow[j];
       }
     }

     if(rank >  0){
       for( j=1; j<columns-1; j++ ){
         sendrow[j]=accessMat( miniSurfaceCopy, 1, j );
       }
       MPI_Irecv(&recvrow, columns-2, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD, &request);
       MPI_Send(&sendrow, columns-2, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD);
       MPI_Wait(&request, &status);
       for( j=1; j<columns-1; j++ ){
         accessMat( miniSurfaceCopy, 0, j ) = recvrow[j];
       }
     }
```

In order to compute the maximum residual difference, every small structure will calculate the maximum of its part of the world, and we will calculate the maximum of all of them with MPI_Reduce. This maximum global residual will be the one that compared to the threshold will determine if the simulation has reached its end or nor.

```
global_residual = 0.0f;
      for( i=1; i<rows-1; i++ )
        for( j=1; j<columns-1; j++ )
          if ( fabs( accessMat( miniSurface, i, j ) - accessMat( miniSurfaceCopy, i, j ) ) >
     global_residual ) {
            global_residual = fabs( accessMat( miniSurface, i, j ) - accessMat(
    miniSurfaceCopy, i, j ) );
          }
          MPI_Reduce(&max_global_residual, &global_residual, 1, MPI_FLOAT, MPI_MAX, 0,
    MPI_COMM_WORLD);
    }


    /* If the global residual is lower than THRESHOLD, we have reached enough stability,
    stop simulation at the end of this iteration */
    if( num_deactivated == num_focal && max_global_residual < THRESHOLD ) flag_stability =
    1;
```

If the simulation proceeds, once the firefighters (that move in a redundant way) have reached a focal point, the heat will be reduced in a radius. This part of the code also had to be adapted to the new structure of the matrices:

```
    /* 4.4.2. Reduce heat in a circle around the team */
      int radius;
      // Influence area of fixed radius depending on type
      if ( teams[t].type == 1 ) radius = RADIUS_TYPE_1;
      else radius = RADIUS_TYPE_2_3;
      for( i=teams[t].x-radius; i<=teams[t].x+radius; i++ ) {
        for( j=teams[t].y-radius; j<=teams[t].y+radius; j++ ) {
          if ( i<1 || i>=rows-1 || j<1 || j>=columns-1 ) continue; // Out of the heated
    surface
          float dx = teams[t].x - i;
          float dy = teams[t].y - j;
          float distance = sqrtf( dx*dx + dy*dy );
          if ( distance <= radius ) {
            accessMat( miniSurface, i -rows/nproc + 1, j ) = accessMat( miniSurface, i -rows
    /nproc + 1, j ) * ( 1 - 0.25 ); // Team efficiency factor
          }
        }
      }
```

In the last place, once the simulation has reached its end, the small structures will send the information of its part of the world to the processor 0, that will gather all the information and compute the final result by the time we close the MPI environment.

```
float recvRow[columns*rows/nproc];

    if(rank >  0){
      MPI_Send(&miniSurface[columns], columns*rows/nproc, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    }

    int r;
    if(rank == 0){
      for(r = 1; r<nproc; r++){
        MPI_Recv(&recvRow, columns*rows/nproc, MPI_FLOAT, r, 0, MPI_COMM_WORLD, &status);
        for( i=0; i<columns*rows/nproc; i++ ){
          surface[i+r*columns*rows/nproc] = recvRow[i];
        }
      }
      for( i=0; i<columns*rows/nproc; i++ ){
          surface[i] = miniSurface[i];
      }
    }

  #ifdef DEBUG
    /* 4.5. DEBUG: Print the current state of the simulation at the end of each iteration */
    print_status( iter, rows, columns, surface, num_teams, teams, num_focal, focal, global_residual );
#endif // DEBUG
  }

  MPI_Finalize();
```

Even though the compilation of this modified code does not present any errors, the execution of it presents a Floating Point Exception, which makes us realize there is a deep problem in our program. Since we were not able to find this error, we were unable to analyze the performance of it.