Report for computer exercise 4 - Particle simulator in MPI

Anton Karlsson
antka388
931217-7117

1 Code overview

1.1 Discussion on partitioning

The box in which the particles was contained in was partitionion column wise among the processors, meaning that each process owned a local box with the orginal hieght and a width of `BOX_VERT_LENGTH`/p, where p is the number of processors. The reasoning behind this partitioning is that is is easy to implement, however, the downside is that the speed-up is not the best. This is because the the local boxes that is owned by a process, has an area that decreases fast as the number of processes increase.

A partitioning that might be better is to partitioning in squares, as the area will not dercease as fast. Thus the probability a particle to a different process will no decrease has fast has the column-wise partitioning. However, the implementation of the square partitioning is harder to code.

1.2 Execution

First, the process definie their deminsion of it own local partiioning using the global values of the original box deminsions along with their process id. Then each process intilies equaly many particles (given by the paramter `NUM_PART`) with random position (within the local box of the process) and a random velocity. This way, we avoid some initial communcation without losing correctness.

The main loop that was given can now be executed. Each process moves its local particles and deal with eventual collisions and wall collsions that occour. Note that any collisions that appear between particels that are not owned by the same process are ingored. Since the probability of this happening is realative small and therefore will not affect the final result noetworthily.

When the partilces has been moved, the process check wheter any particles has coordinates that indicates that it need to move to a neighboring box. The particles that is to be sent to left respective right is sent using dynamic communication.

2 Results

Code

Listing 1: main.c

```c
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <stdbool.h>
#include <mpi.h>

#include "coordinate.h"
#include "definitions.h"
#include "physics.h"

// to do:
// Need to handle nullptr when using plst_t struct.
//Feel free to change this program to facilitate parallelization.
void print_pcord(pcord_t p){

  fprintf(stderr, "x=%f y=%f vx=%f vy=%f\n",
         p.x, p.y, p.vx, p.vy);
}


float rand1(){
  return (float)( rand()/(float) RAND_MAX );
}

void init_collisions(bool *collisions, unsigned int max){
  unsigned int i;
  for(i=0;i<max;++i)
    collisions[i]=0;
}

bool in_local(pcord_t p, cord_t wall) // checks if particle p is in wall wall.
{
  return  (p.x > wall.x0 && p.x < wall.x1);
}

bool exited_left(pcord_t p, cord_t wall, int rank){
  if(rank == 0)
    return false;
  else
    return p.x < wall.x0;
}

bool exited_right(pcord_t p, cord_t wall, int rank, int num_p){
  if(rank == num_p -1)
    return false;
  else
    return p.x > wall.x1;
}

int main(int argc, char** argv){
```

```
double stime = MPI_Wtime();

unsigned int time_stamp = 0, time_max;
int send_left_count=0, send_right_count=0;
int recv_right_count=0, recv_left_count=0;
float pressure = 0, sum = 0;

//MPI stuff
int rank, num_p;
MPI_Init(&argc, &argv);
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &num_p);
MPI_Status status;
//init_mpi_pcord();


/* create a mpi type for pcord_t */
const int nitems=4;
int          blocklengths[4] = {1,1,1,1};
MPI_Datatype ptypes[4] = {MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT};
MPI_Datatype MPI_PCORD;
MPI_Aint     poffsets[4];

poffsets[0] = offsetof(pcord_t, x);
poffsets[1] = offsetof(pcord_t, y);
poffsets[2] = offsetof(pcord_t, vx);
poffsets[3] = offsetof(pcord_t, vy);
//  poffsets[4] = offsetof(pcord_t, to_remove);

MPI_Type_create_struct(nitems, blocklengths, poffsets, ptypes, &MPI_PCORD);
MPI_Type_commit(&MPI_PCORD);


// Declare send arrays used in mpi functions
//int scounts[num_p];
plst_t  *locparticles = NULL;
locparticles = (plst_t *) malloc(sizeof(plst_t));
pcord_t temp_pcord;

// parse arguments
if(argc != 2) {
  fprintf(stderr, "Usage: %s simulation_time\n", argv[0]);
  fprintf(stderr, "For example: %s 10\n", argv[0]);
  exit(-1);
}

time_max = atoi(argv[1]);

/* Initialize */
// 1. set the walls
cord_t wall;
wall.y0 = wall.x0 = 0;
wall.x1 = BOX_HORIZ_SIZE;
wall.y1 = BOX_VERT_SIZE;

cord_t locwall;
```

```
// 2. allocate particle buffer and initialize the particles
bool *collisions= NULL;
collisions = (bool*) malloc(INIT_NO_PARTICLES*num_p*sizeof(bool));

locparticles = NULL;
locparticles = (plst_t*) malloc(INIT_NO_PARTICLES*num_p*sizeof(pcord_t));
pcord_t *send_left = NULL;
send_left = (pcord_t*) malloc(INIT_NO_PARTICLES*num_p*sizeof(pcord_t));
pcord_t *send_right = NULL;
send_right = (pcord_t*) malloc(INIT_NO_PARTICLES*num_p*sizeof(pcord_t));

pcord_t *recv_left = NULL;
recv_left = (pcord_t*) malloc(INIT_NO_PARTICLES*num_p*sizeof(pcord_t));
pcord_t *recv_right = NULL;
recv_right = (pcord_t*) malloc(INIT_NO_PARTICLES*num_p*sizeof(pcord_t));

srand( time(NULL) + 1234 );


// set up walls
locwall.y0 = 0;
locwall.y1 = BOX_VERT_SIZE;
locwall.x0 = rank*BOX_HORIZ_SIZE/num_p;
if (rank == num_p-1)
  locwall.x1 = BOX_HORIZ_SIZE;
else
  locwall.x1 = locwall.x0 + BOX_HORIZ_SIZE/num_p;


float r, a;
int i;

int num_part = INIT_NO_PARTICLES;
// send init particles and designate to processes.
for(i=0; i < num_part; i++) {
  // initialize random position
  temp_pcord.x = locwall.x0 + rand1()*BOX_HORIZ_SIZE/num_p;
  temp_pcord.y = locwall.y0 + rand1()*BOX_VERT_SIZE;

  // initialize random velocity
  r = rand1()*MAX_INITIAL_VELOCITY;
  a = rand1()*2*PI;
  temp_pcord.vx = r*cos(a);
  temp_pcord.vy = r*sin(a);

  push_lst(&locparticles, temp_pcord);
}

/* // print the first elemets */
/* { */
/*   plst_t* current = (plst_t*) malloc(sizeof(plst_t)); */
/*   current = locparticles; */
/*   for (int i = 0; i<5; ++i, current=current->next){ */
/*     fprintf(stderr, "x = %f, y = %f, vx = %f, vy = %f\n", */
/*          current->val.x, current->val.y, current->val.vx, */
/*          current->val.vy); */
/*   } */
/* } */

// if everything works, this should to

// For each proces, do:
unsigned int p, pp;
int pressure_count =0;

// should reutrn plst_t * (so that a refreence could be returned.)
plst_t* current_part = NULL;
plst_t* sub_current_part = NULL;
current_part = (plst_t*) malloc(sizeof(plst_t*));
sub_current_part = (plst_t*) malloc(sizeof(plst_t*));


/* Main loop */
for (time_stamp=0; time_stamp<time_max; time_stamp++) { // for each time stamp
  init_collisions(collisions, INIT_NO_PARTICLES);
  for(p=0,current_part = locparticles;
      p<num_part;
      p++, current_part = current_part->next) { // for all particles
    //fprintf(stderr, "p = %d for processes with rank %d\n", p, rank);
    //current_part = get_part(locparticles, p);
    if(collisions[p]) continue;
    /* check for collisions */
    // If a local boundary is hit, check if collide with any particle

    for(pp=p+1, sub_current_part=current_part->next;
        pp<num_part;
        pp++, sub_current_part=sub_current_part->next) {
      if(collisions[pp]) continue;


      float t=collide(current_part->val, sub_current_part->val);
      //fprintf(stderr, "t = %1.0f\n", t);
      if(t!=-1){ // collision

        //fprintf(stderr, "collsions in rank %d\n", rank);
        collisions[p]=collisions[pp]=1;
        //fprintf(stderr, "before interact in rank %d\n", rank);
        //print_pcord(current_part->val);
        interact(&(current_part->val), &(sub_current_part->val), t);
        //fprintf(stderr, "after interact in rank %d\n", rank);
        // print_pcord(current_part->val);
        sub_current_part = NULL;
        break; // only check collision of two particles
      }
    }
  }


  MPI_Barrier(comm);
  // move particles that has not collided with another
  for(p=send_left_count=send_right_count=0,
        recv_left_count=recv_right_count=0,
        current_part = locparticles;
      p < num_part;
      ++p, current_part = current_part->next){

    if(!collisions[p]) {
      feuler(&(current_part->val), 1);
      pressure += mpi_wall_collide(&(current_part->val),
                                   locwall, rank, num_p);
      ++pressure_count;
      //fprintf(stderr, "Pressure was added from rank %d\n", rank);
    }
    if (exited_left(current_part->val, locwall, rank)) {

      *(send_left + send_left_count) = current_part->val;
      remove_from_list(&locparticles, p);
      --p; --num_part; // to compensate for the removed particle

      ++send_left_count;
      if(num_p == 1 || rank == 0)
        exit(5);
      //printf("sending %d to left\trank %d\n", send_left_count, rank);
    }
    else if (exited_right(current_part->val, locwall, rank, num_p)) {
      *(send_right + send_right_count) = current_part->val;
      remove_from_list(&locparticles, p);
      --p; --num_part; // to compensate for the removed particle

      ++send_right_count;
      if(num_p == 1 || rank == num_p - 1){
        fprintf(stderr, "The right-most process tried to send a particle to a non-
            existent neighbour to the right\n");
        exit(6);
      }
      //printf("sending %d to right\trank %d\n", send_right_count, rank);
    }
  }



  //Send particles to other boxes

  if (rank-1 >= 0) {
    MPI_Send(send_left, send_left_count, MPI_PCORD, rank-1, 0, comm);
    //if (send_left_count > 0)
      //printf("rank %d sent %d particles to left\n", rank, send_left_count);
  }

  if (rank+1 < num_p) {
    MPI_Send(send_right, send_right_count, MPI_PCORD, rank+1, 1, comm);
    //  if (send_right_count > 0)
      // printf("rank %d sent %d particles to right\n", rank, send_right_count);
  }

  // check if something was sent. Using probe to find recv count.
  if(rank-1 >= 0){
    MPI_Probe(rank-1, 1, comm, &status);
    MPI_Get_count(&status, MPI_PCORD, &recv_left_count);
    MPI_Recv(recv_left, recv_left_count, MPI_PCORD, rank-1, 1, comm,
               &status);
      num_part += recv_left_count;
    }

    if(rank+1 < num_p){
      MPI_Probe(rank+1, 0, comm, &status);
      MPI_Get_count(&status, MPI_PCORD, &recv_right_count);
      MPI_Recv(recv_right, recv_right_count, MPI_PCORD, rank+1, 0, comm,
               &status);
      num_part += recv_right_count;
    }

    // update locparticels
    // think about what happens when we tranfer particles between boxes
    multi_push(&locparticles, recv_right, recv_right_count);
    multi_push(&locparticles, recv_left, recv_left_count); //prob. need a special case
        hanfle for when recv_right/left == NULL
  }

  // Gather pressure
  int tot_pressure_count;
  MPI_Barrier(comm);
  MPI_Reduce(&pressure, &sum, 1, MPI_FLOAT, MPI_SUM, 0, comm);
  MPI_Reduce(&pressure_count, &tot_pressure_count, 1, MPI_INT, MPI_SUM, 0, comm);

  double etime = MPI_Wtime();

  if (rank == 0)
    fprintf(stderr, "total time = %g Average pressure = %f pressure count = %d\n",
            etime - stime, sum/(WALL_LENGTH*time_max), tot_pressure_count);

  // free stuff
  free(collisions);
  free(locparticles);
  free(send_left);
  free(send_right);
  free(recv_left);
  free(recv_right);
  MPI_Type_free(&MPI_PCORD);

  MPI_Finalize();
  return 0;
}
```