Thomas Georgiou
Logs

1 MPI Optimization

Over a few days, I worked on optimizing the MPI version of my program to lower the overhead of communication costs. Previously, I had interleaved communication between the various steps of my program, but now that I unified everything into one for loop, there is no place to interleave communication between the for loops.

So to minimize communication, I setup up asynchronous sends and receives. Then, I used the fact that the only reason communication is needed is for the border cells of each subdomain. So what I do is initialize the asynchronous communications, then compute the interior, majority of each subdomain which does not depend on the communication. Then, I wait for the communication to have finished, which hopefully it will have by then, and then compute the border cells. This minimizes the impact of communication, allowing the mpi version to run faster, even on ethernet networks, to a point.

```
//copy memory between nodes
//receive right
if (rank < nodes-1) {
  MPI_Irecv(domain+IDX(xright+1,0,0), SIZEY*DIRS, MPI_FLOAT,
      rank+1, 0, MPI_COMM_WORLD, &requests[0]);
}
//receive left
if (rank > 1) {
  MPI_Irecv(domain+IDX(xleft-1,0,0), SIZEY*DIRS, MPI_FLOAT,
      rank-1, 0, MPI_COMM_WORLD, &requests[1]);
}
//send right
if (rank < nodes-1) {
  MPI_Isend(domain+IDX(xright,0,0), SIZEY*DIRS, MPI_FLOAT,
      rank+1, 0, MPI_COMM_WORLD, &requests[2]);
}
//send left
if (rank > 1) {
  MPI_Isend(domain+IDX(xleft,0,0), SIZEY*DIRS, MPI_FLOAT,
      rank-1, 0, MPI_COMM_WORLD, &requests[3]);
}
```

```
#pragma omp parallel for
for (int x = xleft+1; x < xright; x++) {
  process(x);
}

//finish communication between nodes
for (int n = 0; n < 4; n++) {
  if (requests[n] != MPI_REQUEST_NULL) {
    MPI_Wait(&requests[n],&status);
  }
}
#pragma omp sections
{
  #pragma omp section
  {
    int x = xleft;
    process(x);
  }
  #pragma omp section
  {
    int x = xright;
    process(x);
  }
}
```

2 More Visualization

I worked on different ways to visualize my program so that I could better see what the fluid is doing.

The first thing I did was to implement multicolored velocity magnitudes. The way this works is that the velocity at each grid point is computed both in the x and y directions. Then, the color for that grid point is determined by the velocities. The red component is controlled by the horizontal velocity and the green component is controlled by the vertical velocity. This allows me to see in what direction the fluid at each point is moving and how fast.

```
//compute velocity field
#pragma omp parallel for
for (int x = 0; x < SIZEX; x++) {
  for (int y = 0; y < SIZEY; y++) {
    float rho = 0.0;
```

```
      float ux = 0.0;
      float uy = 0.0;
      for (int d = 0; d < DIRS; d++) {
        rho += domain[IDX(x,y,d)];
        ux  += dx[d] * domain[IDX(x,y,d)];
        uy  += dy[d] * domain[IDX(x,y,d)];
      }
      ux /= rho;
      uy /= rho;
      if (rho < 1e-3) {
        ux = 0.0;
        uy = 0.0;
      }
      vx[x][y] = ux;
      vy[x][y] = uy;
    }
  }

  glBegin(GL_QUADS);
  for (int x = 0; x < SIZEX; x++) {
    for (int y = 0; y < SIZEY; y++) {
      float ux = fabsf(vx[x][y]);
      float uy = fabsf(vy[x][y]);
      glColor3f(ux*scale,uy*scale,0);
      glVertex2i(x,y);
      glVertex2i(x+1,y);
      glVertex2i(x+1,y+1);
      glVertex2i(x,y+1);
    }
  }
```

3 Streamlines

Another visualization technique I added were streamlines. These are lines that a particle would follow if it were to pass through the fluid. Compared to particles, these show the path that would be taken at every timestep rather than having to wait and watch the particles to figure out what is happening. To implement this, I started at the rightmost side of the cylinder and at evenly spaced nitervals and then moved backwards a fixed number of times until I hit the left side. Every timed I moved backwards, I calculated the derivative $\frac{dy}{dx}$ and then moved the streamline up or down corresponding to it.

But the streamlines would look very jagged since velocities were constant

over every gridpoint, or a 2x2 pixel square onscreen. So I implemented bilinear interpolation. This allowed me to transform the discrete velocity field into a smooth, continous one, making the streamlines much smoother. To do so, I linearly interpolated in the horizontal direction at two different y values for each point. Then, I interpolated in the vertical direction using the two previously computed values.

Streamline code:

```
for (int y = 0; y < SIZEY; y+=10) {
  continue;
  glColor3f(0.1,0.8,0.1);
  glBegin(GL_LINE_STRIP);
  float x0 = SIZEX-1.5;
  float y0 = y+0.5;
  glVertex2f(x0,y0);
  for (int i = 0; i < SIZEX/1; i++) {
    float x1 = floor(x0);
    float x2 = ceil(x0);
    float y1 = floor(y0);
    float y2 = ceil(y0);
    if (x1<0 || x2 >= SIZEX || y1 < 0 || y2 >= SIZEY)
      break;
    if (isnan(x1) || isnan(x2) || isnan(x0) || isnan(y1) ||
        isnan(y2) || isnan(y2) || isnan(y0))
      break;
    float r1x = (x2-x0)/(x2-x1)*vx[(int)x1][(int)y1] + (x0-x1)
        /(x2-x1)*vx[(int)x2][(int)y1];
    float r1y = (x2-x0)/(x2-x1)*vy[(int)x1][(int)y1] + (x0-x1)
        /(x2-x1)*vy[(int)x2][(int)y1];
    float r2x = (x2-x0)/(x2-x1)*vx[(int)x1][(int)y2] + (x0-x1)
        /(x2-x1)*vx[(int)x2][(int)y2];
    float r2y = (x2-x0)/(x2-x1)*vy[(int)x1][(int)y2] + (x0-x1)
        /(x2-x1)*vy[(int)x2][(int)y2];
    float px = (y2-y0)/(y2-y1)*r1x + (y0-y1)/(y2-y1)*r2x;
    float py = (y2-y0)/(y2-y1)*r1y + (y0-y1)/(y2-y1)*r2y;
    float scalefactor = 1/(px);
    x0 -= px*scalefactor;
    if (abs(py*scalefactor) > 20)
      break;
    y0 -= py*scalefactor;
    glVertex2f(x0,y0);
  }
  glEnd();
}
```