Physics 555b Assignment #4 (version 1.0)

Martin Snajdr

Spring 2006

1 Description of the application

The purpose of this assignment is to practice "parallel thinking" by designing a simple parallel application. The task of the application is to simulate the motion of trace particles (tracers) in a given velocity field. Imagine for example motion of smoke particles in a wind. For our purpose and simpler visualization we will consider a 2D domain. The parallelization will be based on physical domain decomposition, i.e., each processor will evolve particles that currently belong to its subdomain. Of course, particles will migrate from subdomain to subdomain and also leave the domain completely. This approach is efficient as long as the particles are approximately uniformly distributed over the domain.

2 Analytic velocity field

Let the domain be a square with the left lower corner at $(-1,-1)$ with its center at $(0,0)$. Assume for now that we can evaluate the velocity field at each point of the domain, i.e., we know the functions $v_x(x,y)$ and $v_y(x,y)$.

The first task is to divide the domain equally among n processors. One way would be to simply cut the domain into n strips along some dimension. This, however, is not the most efficient way. For large n the strips will be skinny and particle migration will be typically large. Therefore it is good to maximize the volume/surface ratio, i.e., to make the subdomains as square-like as possible. The task can be formulated as follows. Find the factorization of $n = n_x n_y$ such that n_x and n_y are as close integers as possible. Then divide the domain into $n_x \times n_y$ subdomains. Assign each subdomain to a unique processor.

Use second order Runge-Kutta scheme to advance each particle position from time level n to time level $n+1$ by Δt

$$\vec{x}^* = \vec{x}^n + \vec{v}(\vec{x}^n)\Delta t \ , \tag{1}$$

$$\vec{x}^{n+1} = \frac{1}{2}\left(\vec{x}^n + \vec{x}^* + \vec{v}(\vec{x}^*)\Delta t\right) \ . \tag{2}$$

The next step is to migrate particles that a processor's subdomain into the proper processor. A simple way of doing this (although probably not the most efficient) is that each processor *broadcasts* the positions of particles that left its subdomain. Each processor then "grabs" the particles belonging to its subdomain.

Write the program by using the techniques described next. Test the scaling behaviour, i.e., plot the computational time needed to evolve Np points over fixed time as a function of number of computationla nodes (use a log-log scale). Plot it for $Np = 100, 1000, 10000, \ldots$ and as many processors as you can find (at least up to 8). Compare with the ideal scaling and comment. Note that in general the domain is not filled with particles uniformly, so that processors must handle different number of particles. Therefore even if the communication would be infinitely fast the scaling would not be ideal.

3 Useful techniques

In this section I describe some useful techniques.

3.1 Reading input parameters

The bbhutils offer an easy way for reading parameters from files. The routines are described at the beginning of the bbhutil manual located at

`http://laplace.physics.ubc.ca/Doc/rnpl/bbhutil.ps`

Especially useful is the ivec type. The ivec type allows for easy and compact notation of specifying an index vector. This is useful for output control. The next snippet of code shows how to use the variable `out_ivec` to control the output frequency. The parameter file is passed as the first command line argument.

```
int main( int argc, char *argv[]) {
  int out_ivec[100],nsteps,step;
  .
  .
  .
  if (!get_int_param(argv[1],"nsteps",&nsteps,1))
    perror("Error reading parameters");
  if (!get_ivec_param(argv[1],"out_ivec",out_ivec,100))
    perror("Error reading parameters");
  fixup_ivec(0,nsteps,0,out_ivec);
  .
  .
  .
  step = 0;
  while (step<=nsteps) {
    .
    .
    .
  if (do_ivec(step,1,out_ivec)) save_particles_(...);
    .
    .
    .
    step++;
```

```
  }
  .
  .
  .
}
```

3.2 Saving particle positions

There is a new addition to the sdf format (not yet described in the documentation) for saving "particle" data. The data can be viewed by DV by selecting the item *Particles* from the *Options* menu of the *DV local view* window. If your copy of DV does not have the item in the menu you should install the latest version (from CVS). Here is a C routine that saves a 2D particle data.

```
subroutine save_particles(fname,time,xp,yp,Np)
implicit none
integer i
integer Np           ! number of particles
real*8 time          ! time stamp
real*8 xp(Np),yp(Np) ! x and y particle positions
real*8 scratch(4*Np) ! auxiliary array
character*200 fname  ! file name
do i=1,Np
  scratch(i) = xp(i)
  scratch(Np+i) = yp(i)
  scratch(2*Np+i) = 0.d0 ! set z coordinate to 0
  scratch(3*Np+i) = 0.d0 ! can set to some value for color map
enddo
call gft_out_part_xyzf(fname,time,scratch,Np,1)
return
end
```

Do not forget to use a unique name for output from each node. The best way is to add a node number after the base name such as `particles_0.sdf`.

3.3 Initialization

The easiest way to initialize the particle positions is to randomly generate particles within a desired domain. For this purpose the C-functions `srand48(int seed)` and `drand48()` are sufficient. Do not forget to include the header file `stdlib.h`.

Here is a simple initialization routine that populates a rectangular domain defined via `domain_bbox` with `npart` particles.

```
void populate_particles(int npart, real *pposx, real *pposy,real *domain_bbox) {
  double r1,r2;
  real wx,wy;
  int i;
  wx = domain_bbox[1]-domain_bbox[0];
  wy = domain_bbox[3]-domain_bbox[2];
  for(i=0;i<npart;i++) {
    r1 = drand48();
    r2 = drand48();
    pposx[i] = domain_bbox[0] + wx*r1;
    pposy[i] = domain_bbox[2] + wy*r2;
  }
}
```

Do not forget to call the `srand48(int seed)` before calling `populate_particles()` to initialize the pseudo-random number generator.

3.4 The velocity field

A simple way of testing the code is to use a velocity field that generates rigid rotations

$$v_x(x,y) = -y\Omega \ , \tag{3}$$

$$v_y(x,y) = x\Omega \ , \tag{4}$$

where Ω is the angular velocity. If we choose $\Omega = 2\pi$ then we should complete one rotation around the $(0,0)$ point in one unit of time. Therefore all the particles should have the same positions as at the initialization time.

A nice patterns can be obtained by distorting the rigid rotation such as

$$v_x(x,y) = -y\Omega r \ , \tag{5}$$

$$v_y(x,y) = x\Omega r \ , \tag{6}$$

where $r = \sqrt{x^2+y^2}$. Sample movies can be found on the course web page.

4 Grid based velocity field

Instead of assuming a known analytic velocity field, assume that the velocity is defined only on a grid. For simplicity assume that the velocity field is time independent. Use PAMR to distribute the two velocity fields `vx` and `vy`. Since PAMR automatically divides the domain into subdomains the manual division is no longer necessary.

In order to evaluate a velocity at any given point use linear interpolation using the four vertices that surround a given point. First interpolate in the x-direction and then in the y-direction.

Use two ghost zones for buffering. Note that the Δt should be sufficiently small so that a particle from inside the domain does not migrate beyond the buffer zone in the first Euler substep (1) (so we can complete the Runge-Kutta step). Of course it can migrate out of the physical domain as before.

This application can be used to visualize fluid flows that are calculated on a computational grid (of course in that case the velocity field would be also time dependent).

Of course, the denser the grid the closer the results should be.

5 Assignment description

Write a parallel program that evolves particles in the analytic velocity field as described above. Implement at least the two fields (3)–(4) and (5)–(6). If you find some velocity field that results in a particularly nice pattern, write it down and show the results.

Test the scaling behaviour, i.e., plot the computational time needed to evolve Np points over fixed time as a function of number of computationla nodes (use a log-log scale). Plot it for $Np = 100, 1000, 10000, \ldots$ and as many processors as you can find (at least up to 8). Compare with the ideal scaling and comment. Note that in general the domain is not filled with particles uniformly, so that processors must handle different number of particles. Therefore even if the communication would be infinitely fast the scaling would not be ideal.

For testing the scaling behavior, populate (randomly) the whole domain by particles. Use the rigid rotation field (3)–(4) and complete one rotation (scale so that $t_{\text{final}} = 1$). (The resulting shape of the particle cloud should be a disc inscribed in the physical domain). Use Δt such that after completing one full rotation the relative error $\Delta r/r$ between initial and final position of each particle is less than 1%. What is the value of Δt?

The program should read the parameters from a parameter file passed as a command line argument. Make sure that all relevant parameters are read from the parameter file so that the program itself does not need to be recompiled when you want to change the number of particles etc.. Use some parameter to choose the velocity field being used.

As a bonus, implement the grid based velocity field. Test the code by initializing the velocity fields as before by using the rigid rotation field (3)–(4) and compare with the results of the first assignment.