

1 Diffusion Equation

The diffusion equation is a simple parabolic partial differential equation that can be solved numerically using a finite difference method.

$$\frac{d\phi}{dt} = \kappa \nabla^2 \phi$$

in one dimension

$$\frac{d\phi}{dt} = \kappa \frac{d^2\phi}{dx^2}$$

The simple finite difference representation of this equation is in one dimension

$$\frac{\phi_{i,j+1} - \phi_{i,j}}{k} = \kappa \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2}$$

or

$$\phi_{i,j+1} = \phi_{i,j} + \kappa \frac{k}{h^2} (\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j})$$

$$\phi_{i,j+1} = \left(1 - 2\kappa \frac{k}{h^2}\right) \phi_{i,j} + \left(\kappa \frac{k}{h^2}\right) (\phi_{i+1,j} + \phi_{i-1,j})$$

where h is the spatial grid size, k is the time-step size, and κ is a diffusion coefficient.

For this problem, let

$$\begin{aligned} x_{min} &= 0 \\ x_{max} &= 1 \\ \kappa &= 1 \\ k &= 0.2 \times h^2 \\ t_0 &= 0 \\ t_{final} &= 0.5 \end{aligned} \tag{1}$$

Also, let

$$\phi = \begin{cases} 0, & x < 0.4 \\ 1, & 0.4 \leq x \leq 0.6 \\ 0 & x > 0.6 \end{cases}$$

and print the output every 0.05 in time.

For this project

1. Write a simple serial C or Fortran code to solve the problem. Make sure to dump the output every 0.05 time units out. The program should ask the user for the total number of cells to run, then calculate h and k based on the input.
2. Modify the serial code to run on MPI using a one-dimensional spatial decomposition. You will need to communicate with neighbors on every time-step, and gather data for the output.

2 Manager/Worker Mandelbrot Set

The Mandelbrot Set is an iterative map in the complex plane that has chaotic properties. Given an initial location in the complex plane $C = x + iy$ where $i = \sqrt{-1}$, we can go through the following iterations.

$$r_0 = c$$

$$r_i = r_{i-1}^2 + c$$

remember, for complex multiplication

$$(x + iy) \times (x + iy) = x^2 - y^2 + 2ixy$$

After a given number of iterations, many points begin to diverge. Any iteration where $|r_i| > 2$ will leave the complex plane.

To make a Mandelbrot plot, you need to loop through the complex plane from $x = (-2, 2)$ and $y = (-2i, 2i)$. The pixel color at location C is associated with the number of iterations it takes to reach $C > 2$.

It is important to note that some starting points will never reach $C > 2$, so a maximum iteration number is needed in the problem (500 iterations, for example.)

For this problem, you are to make a parallel version of the Mandelbrot problem that uses the manager/worker programming pattern.

1. Write a simple C or Fortran code to calculate the Mandelbrot set. Find the number of iterations needed to exit a given point in the complex plane using the iterative mapping described above. Use a maximum cutoff for the iterations of 500 cycles. You should have the user input the number of points (n) in the x and y direction at the beginning of the code, then divide the sampling points into an $n \times n$ grid.
2. Modify the code to work under MPI using a manager/ work paradigm.
 - (a) To do this, you should designate node 0 to manage the work load.
 - (b) Each CPU will be given an initial parcel of work to do.
 - (c) Processor 0 will then listen for communications from the others using asynchronous communication.
 - (d) When any processor finishes its work block, it should send a message to processor 0 asking for another block. At the same time, it should send its results back to node 0 so they can be output to a file. It should go in a receive state to get the message from node 1 about which block to do.
 - (e) Processor 0 will assign the processor a new block, and update the list of things left to do. If there are no blocks left to be calculated, then it should tell the CPU to exit.
3. You will need to experiment with the block sizes and resolutions for a given set of CPU's.

3 N-body Assembly Line

In this problem, you will solve the gravitational potential for a set of particles using an assembly line programming paradigm.

The gravitational potential for a given particle is given by the equations

$$\phi_j = \sum_{i=1, i \neq j}^n \frac{-GM_i M_j}{R_{ij}}$$

So, to calculate the gravitational potential, you need to have a loop over all the particles like

```
for j = 1:n
  potential(j) = 0
  for i = 1:n
    if (i /= j) then
      R_ij = sqrt( (x(i) - x(j))^2 + (y(i)-y(j))^2 + (z(i)-z(j))^2)
      potential(j) = potential(j) -G * M(j) * M(i)/ Rij
    endif
  enddo
enddo
```

To do this in parallel, you will use the “assembly line” programming pattern.

Assume you are running n particles over $nproc$ processors

1. Generate random positions for $n/nproc$ particles for each of $nproc$ files. The file names should be named “particle0”, “particle1”, “particle2”, etc where the number designates the processor ID associated with the file. You can (and perhaps should) generate the particle data with a serial code. If you do the generation in parallel, be aware that you must worry about how the random number generator works on each CPU. Assume the masses for all the particles are 1, and the gravitational constant is 1 as well. For calculational ease, assume the x, y , and z coordinates range from 0 to 1.
2. Write a serial code that reads in all the data files, and calculates the potential for each particle. The code should write out the potentials into a single file, and return the minimum and maximum potential for the entire set of particles.
3. Convert the code to use MPI.

- (a) The code should read the data files locally off each CPU. This will mean that you need to copy the appropriate data file to each node.
 - (b) After calculating the local contribution of the potential from the particles that are native to the CPU, you need to rotate the particles to the next CPU. After you have received the data from the neighboring processor, update the potential sum by calculating the contribution from each of these particles.
 - (c) Repeat this shifting of particle positions until you have circulated the data through all the CPU's.
4. Copy the appropriate data file to each CPU you will use in your parallel run.
 5. Compare the serial and parallel versions of the code, and be able to present a brief overview of how the code was constructed and how it works in class.

The figures below illustrate how data will be transferred. A copy of the local positions will be send to the neighboring CPU. Potentials will be updated using the new data, then the shift will repeat.



