CS205 PA1: All About that MPI

Due: Wednesday, February 11th, 11:59 PM

Before We Dive In

Getting to Know You

If you haven't already, please reply as a comment to the "Introduction" Thread on Piazza. In a short paragraph, tell us some general information, fun facts, unique hobbies and/or projects, academic interests and why you want to take this class.

Also fill out the Class Survey if you haven't already!

Setup (and a quick question)

The majority of the problems will require benchmarking through running on the cluster. The staff is still finalizing some arrangements on the cluster, but we will post separate instructions once this settles down. In the meanwhile, you can always run things locally if you install Python and mpi4py found here. And remember, if your computer only has 2 cores, you won't see more than 2x speedup even if you run with multiple processes with `mpirun`. It'll just get hyperthreaded by your CPU. Hyperthreading would increase instruction-level parallelism but you still won't see a speedup. Why? Jot down your thoughts in `P1.pdf` which you will answer other questions in in a bit.

For now, first sign up for a SEAS account here. You won't be using this immediately as you will need the Odyssey Cluster accounts but this will be a start. Please then submit your SEAS account name on this form.

Finally, a quick disclaimer: we are releasing this problem set a bit earlier than the class has prepared for to give as much time for those who work ahead of schedule. However, the second lab held on Wednesday, February 4, will be very useful for a lot of the syntax and workflow. The pset can be completed after that lab. Until then, feel free to post on Piazza and do some searches online — MPI is very well documented and taught in many places as a core part of parallel programming.

Now, before we introduce the problems, make sure you download the distro code here

1 Warm Up: Generalizing the Inner Product [5 pts]

Recap from Lab

If you recall from lab, we wish to compute the inner-product in parallel:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{k=0}^{K-1} a_k b_k$$

We considered using both MPI `broadcast/reduce` and MPI `scatter/gather` approaches.

Did we really do that right?

Try running the code from lab with $P = 5$ processes. Show and explain results.

Generalization

Since our implementation was a bit more naive, we required that the number of tasks or bodies, n, was fully divisible by the number of processors to ensure even division, P. We now wish to generalize to be able to handle all n and P. You are allowed to build off your solution from lab. Don't forget to explain your approach.

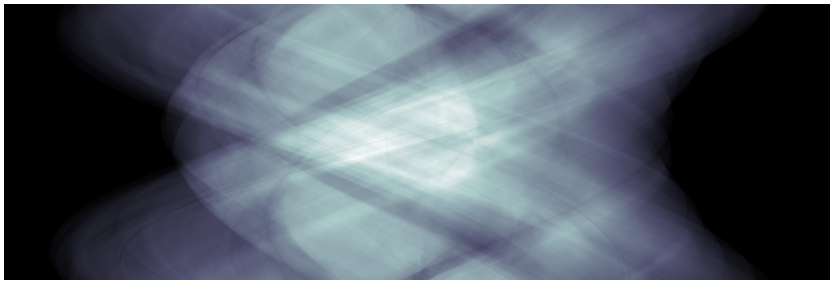
One last thing...

What other improvements could be made to the code? Explain.

Submission Requirements:

- P1.py: Generalized inner product code
- P1.pdf: Answer to question from "Setup" and P1 explanations and output

2 Application: Tomography [10 pts]



Tomography is the rendering of multiple cross-sectional images via superposition. This is the principal behind imaging technologies like CT scans. We are looking to speed up the process of rendering the tomographic projection from the data, `TomographicData.bin`, which consists of 2048 rows of 6144 double-precision (64-bit) floating point samples each. One row of data is the projection of an image at an angle ϕ, where row k corresponds to the angle $\phi = -k\pi/2048$.

The x-ray tomography machine collects data files like this as it circles around a patient's head and images a single slice of tissue at different angles. Each line of data is the attenuation curve of the slice taken at a different angle. You are tasked with creating the highest resolution image from this data as quickly as possible in parallel.

Starting with Serial

The first quick task, is to finish the serial implementation to check for understanding of the method. You have been provided a template, `P2serial.py`, where we have provided a `data_transformer` class:

```
# structure of given data_transformer class
class data_transformer:
  def __init__(self, sample_size, image_size)
  def transform(self, data, phi)
```

Remember to save your image like an x-ray image!

```
plt.imsave{filename, data, cmap='bone')
```

Putting it into Parallel: MPI Send/Recv

Write a parallel version in which the root process reads the input and uses only MPI `send` and `recv` to distribute the data and perform the computation in parallel. Show how you verify your program is correct. Plotting the image at each step is not required. You may assume the number of processes is a power of two.

Putting it into Parallel: MPI Scatter/Reduce

Instead of `send` and `recv` use only MPI `scatter` and `reduce` to write a (hopefully) simpler version. Plotting the image at each step is not required. You may assume the number of processes is a power of two.

Analysis of Results

Time the scatter/reduce version with `MPI.Wtime()` counting only the communication and computation — ignore any IO, plotting or precomputation. Compute the speedup and efficiencies of the scatter/redyuce version with `ImageSize = 512, 1024, 2048` and `P=1,2,4,8`. Tabulate these values. Discuss what you observe — do you see an increase in efficiency with larger `ImageSize`?

Submission Requirements:

- P2serial.py: Serial implementation
- P2a.pdf: Parallel Send/Recv implementation
- P2b.pdf: Parallel Scatter/Reduce implementation
- P2.pdf: Explanations/plots of outputs, times and efficiencies

3 Technique: Domain Decomposition [15 pts]

The 2D Wave Equation

We wish to model the 2D wave equation, given by:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

where u can be thought of as the wave "height" and the above equation defines its evolution in time and space. We model this problem on the domain $(x, y, t) \in [0, 1] \times [0, 1] \times [0, T]$ with the boundary conditions:

$$u(x, y, 0) = u^0(x, y) \qquad \frac{\partial}{\partial t} u(x, y, 0) = 0$$
$$\frac{\partial}{\partial x} u(0, y, t) = 0 \qquad \frac{\partial}{\partial x} u(1, y, t) = 0 \qquad \frac{\partial}{\partial y} u(x, 0, t) = 0 \qquad \frac{\partial}{\partial y} u(x, 1, t) = 0$$

Define

$$u^n_{i,j} = u[(i-1)\Delta x, (j-1)\Delta y, n\Delta t]$$

for $i = 1, \ldots, Nx$ and $j = 1, \ldots Ny$. Additionally, $\Delta x = 1/(Nx - 1)$ and $\Delta y = 1/(Ny - 1)$ are the grid spacings so that $u^0_{1,1} = u(0, 0, 0)$ and $u^0_{Nx,Ny} = u(1, 1, 0)$.

Then we can write the continuous PDE in terms of our grid of values using second-order central difference approximations.

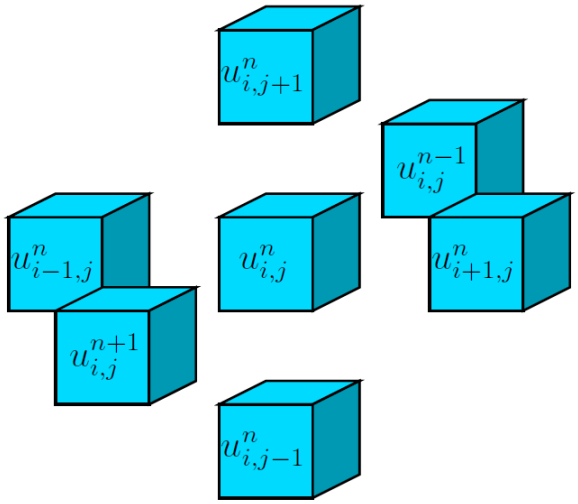
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^n_{i-1,j} - 2u^n_{i,j} + u^n_{i+1,j}}{\Delta x^2}$$
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^n_{i,j-1} - 2u^n_{i,j} + u^n_{i,j+1}}{\Delta y^2}$$
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u^{n-1}_{i,j} - 2u^n_{i,j} + u^{n+1}_{i,j}}{\Delta t^2}$$

Substituting these into the wave equation, letting $\Delta x = \Delta y$ (and $Nx = Ny$), and solving for $u^{n+1}_{i,j}$, we derive the update scheme for our grid:

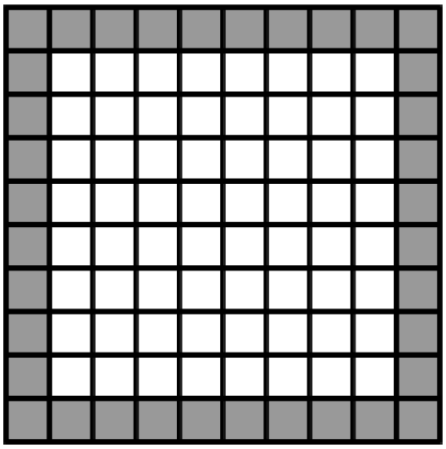
$$u^{n+1}_{i,j} = \left(2 - 4\frac{\Delta t^2}{\Delta x^2}\right) u^n_{i,j} - u^{n-1}_{i,j} + \frac{\Delta t^2}{\Delta x^2}(u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1}) \tag{1}$$

Thus, if we assume we know $u^n_{i,j}$ and $u^{n-1}_{i,j}$ for all i, j, then we can compute the next step in time, $u^{n+1}_{i,j}$,for each i, j.

We have discovered our computational stencil and defines the dependencies of the computation. In order to compute the value $u^{n+1}_{i,j}$, we need the values $u^n_{i-1,j}$, $u^n_{i+1,j}$, $u^n_{i,j-1}$, $u^n_{i,j+1}$, $u^n_{i,j}$, and $u^{n-1}_{i,j}$. This is shown in the below figure.



If we were to use this stencil to compute $u^n_{i,j}$, we would also require points that we don't represent which are outside of the problem's domain (like $u^n_{0,j}$ and $u^n_{i,Ny+1}$. Typically, we would have to test if we are currently updating the boundary to make sure we don't go off the edge of our grid. Alternatively, we can pad the domain with dummy values of those "outside" grid locations so that we don't have to modify the stencil on the boundaries, shown in the figure below. Then, we need only to force these padded grid points to have the correct value. This method is called using *ghost points*.



To determine what values the ghost points should have, we note that the central difference approximation of the boundary condition

$$0 = \frac{\partial u}{\partial x}(0, y, t) \approx \frac{u^n_{0,j} - u^n_{2,j}}{2\Delta x}$$

implies that $u^n_{0,j} = u^n_{2,j}$. Similar results hold for the others, resulting in:

$$u^n_{0,j} = u^n_{2,j} \qquad u^n_{Nx+1,i} = u^n_{Nx-1,j} \qquad u^n_{i,0} = u^n_{i,2} \qquad u^n_{i,Ny+1} = u^n_{2,Ny-1} \tag{2}$$

Therefore, after computing all the interior points for step $n + 1$, we can set the ghost values appropriately for computing step $n + 2$.

Each iteration then follows the below steps:

- Compute the interior grid points with the computational stencil for step $n+1$. Compute $u^{n+1}_{i,j}$ for all i, j, using u^n and u^{n-1} in Equation (1).
- Set the ghost points for u^{n+1} using Equation (2).
- Set $u^n \to u^{n-1}$, $u^{n+1} \to u^n$ and continue to the next step $n + 1 \to n$.

The serial version is implemented for you in `P2serial.py`.

Domain Decomposition

To parallelize this computation, we partition the domain among the processes — giving each process responsibility for a portion of the full domain. All pieces of the domain take approximately equal time thus justifying a geometric partitioning. We will use a rectangular grid decomposition with `Px` and `Py` processes in the $x-$ and $y-$directions.

Similar to the serial program, in order to compute u^{n+1} each process will require data from grid points that it is not responsible to update. If each process uses the ghost point method — it collects and uses information on its border but does not compute the update stencil at those points, we may have found ourselves a parallelization scheme. But this can be achieved differently. Illustrate, in a figure similar to the ghost point grid above, what needs to be communicated between processes at each iteration.

Implementation Notes and Tasks

Soon, you will be tasked with implementing a parallel version of the 2D wave simulation described above using the rectangular grid domain decomposition described above.

You may use any communicator strategy that you like, including:

- Using the `COMM_WORLD` throughout keeping track of the `COMM_WORLD` ranks
- Using `Split` to create your own communicator(s) each with different ranks
- Using `Cartcomm` — a communicator designed for Cartesian topologies

You may assume that `Nx`, `Ny`, `Px` and `Py` are all powers of two and may be defined as global constants. The initial conditions may be computed locally or computed on the root and distributed.

Document your choices and strategy in `P3.pdf`.

HINT: We have provided two classes in `Plotter3DCS205.py` that will help you plot 3D data. The first class, `MeshPlotter3D`, is used in the provided serial version. if this class is used on a session that is interactive, then each process can launch and update a separate plot with its local data.

Alternatively, if you use the `MeshPlotter3DParallel` class and pass in the global indices for your local date (ex: for a particular process, `u[1,1]` might represent $u_{32,32}$ so $(1,1)$ is the local index and $(32,32)$ is the global index.), then the data will be gathered and only a single plot will be produced with the global data. The script `Plotter3DCs205.py` also includes a simple example which implements `MeshPlotter3DParallel`, which can be viewed by running the plotting script itself in interactive modes:

```
$ mpirun -n (Px*Py) python Plotter3DCS205.py Px Py
```

Both `meshPlotter3D` and `MeshPlotter3DParallel` may be useful for finding errors or simply just impressing a friend — because it's cool.

Implementation: Sendrecv

Implement the discussed 2D wave simulation in parallel using MPI `sendrecv` - not MPI `send/recv` as `P2a.py`. Discuss why we aren't using MPI `send/recv`.

HINT: What's the name of the next section?

Implementation: Non-Blocking Isend/Irecv

We can use MPI `isend/irecv`. Revise your implementation to use these non-blocking communications and save as `P2b.py`.

Analysis

With some chosen (`Nx`, `Ny`), measure the time/iteration and the speedup with various (`Px`, `Py`) pairs. Discuss your results.

Recall that weak scaling is exhibited when the problem size per processor is kept constant and the efficiency does not decrease. Does this problem exhibit weak scaling? Why or why not? Explain your reasoning through both theoretical and experimental results.

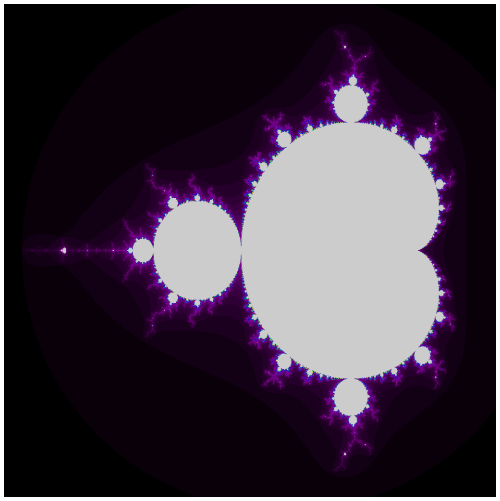
TIPS: If the amount of work per processor is kept constant, how does the amount of communication per processor scale as the number of processors is increased? What about the overhead?

Submission Requirements:

- P3a.py: `sendrecv` implementation
- P3b.py: `isend/irecv` implementation
- P3.pdf: Explanations and/or plots

4 Technique: Master/Slave [10 pts]

The Mandelbrot Set



The Mandelbrot set, visualized above, is a particular instance of a class of fractal sets which can be computed from the quadratic recurrence equation:

$$z_{n+1} = z_n^2 + C$$

Because computing the Mandelbrot set is embarrassingly parallel — no pixel of the image depends on any other pixel — we can distribute the work across many MPI processes in many ways. In this problem, we will conceptually think about load balancing this computation and implement a general strategy that works for a wide range of problems (though this doesn't always scale to extremely large problems).

A pixel of the above Mandelbrot image is located at the coordinates (x, y) can be computed with the `mandelbrot` function:

```
def mandelbrot(x,y):
  z = c = complex(x,y)
  it, maxit = 0, 511
  while abs(z) < 2 and it < maxit:
    z = z*z + c
    it += 1
  return it
```

Thinking About Load Balancing

One issue that you may have realized is that the `mandelbrot` function can require anywhere from 0 to 511 iterations to return. This means that dividing the work into equal portions

of the image might not result in even distribution. In fact, if you run `P4serial.py` and observe carefully, you can see a difference. That being said, two hypothetical students, Sally and John, propose different ways to divide up the image.

Sally implements the computation with P MPI processes by making process p compute all of the (valid) rows of $p + nP$ for $n = 0, 1, 2 \ldots$ and then using an MPI `gather` operation to collect all of the values to the root process.

John's strategy, also with P MPI processes, is to make process p copmile all of the (valid) rows $pN, pN + 1, pN + 2, \ldots, pN + (N - 1)$ where $n = \lceil height/P \rceil$ and then use an MPI `gather` operation to collect all of the values to the root process.

Which student should get the A? Why? Discuss in `P4.pdf`

Real Load Balancing: Master/Slave

In general, you may not always know how to best distribute the tasks you need to compute (unlike with the Mandelbrot set image). To effectively distribute the work in this case, you could use the master/slave model where each processor requests a job whenever it is idle and there are more tasks to complete.

The master process is responsible for giving each process a unit of work, receiving the result from any slaves and then sending slaves new units of work as needed.

A slave process is responsible for receiving a unit of work, completing it, sending the result back to the master and requesting another task until the master tells the slave to stop.

In what cases would this master/slave process not be ideal? Discuss in *P4.pdf*.

Then, implement the Mandelbrot image computation using a master/slave MPI strategy where a job is defined as computing a row of the image. Communicate as little as possible. Please ensure that your code is structured with a function for the master process and a function for the slave processes — perhaps something like:

```
def slave(comm):
  # slave do work
  return

def master(comm):
  image = np.zeros([height,width], dtype=np.uint16)
  # master's tasks
  return image
```

Extra Credit [5 pts]

Implement both John's and Sally's approaches from the first part of this problem and analyze the speedup and efficiency of all John's, Sally's and the master/slave approach against the provided serial version. Use up to 192 processes and an image size of your choice. Submit the plots and a discussion of your results.

Submission Requirements:

- P4.py: Master/Slave Mandelbrot implementation
- P4.pdf: Explanations and plots of analysis
- P4Sally.py: **(Extra)** Sally's Mandelbrot implementation
- P4John.py: **(Extra)** John's Mandelbrot implementation