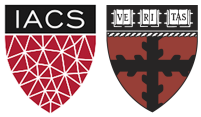
CS205, Fall 2012
Computing Foundations for Computational Science
Hanspeter Pfister, Cris Cecka



HARVARD
School of Engineering
and Applied Sciences

Homework 3 – Due October 26th, 2012 at 11:59pm

This week's homework covers more advanced MPI with domain decomposition and functional-style programming. We will also develop and execute a very simple CUDA code with Python.

Download the code for HW3 here.

or

`wget http://www.cs205.org/resources/hw3.zip`

Problem 1 - MPI Domain Decomposition **2**
- Domain Decomposition . . . 4
- Implementation . . . 4
  - sendrecv . . . 5
  - Isend/Irecv . . . 5
- Analysis . . . 5

Problem 2 - Hello CUDA **6**
- Complete the SAXPY kernel . . . 6
- Obtain a performance benchmark . . . 6
- Manipulate the block and grid size . . . 6
- Performance Model . . . 6

Problem 1 - MPI Domain Decomposition

In this problem, we wish to model the 2D wave equation:

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

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:

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

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 $u^0_{0,0} = u(0, 0, 0)$ and $u^0_{Nx,Ny} = u(1, 1, 0)$.

We can write the continuous PDE in terms of our grid of values using second-order central difference approximations:

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

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 updating scheme for our grid:

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

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.

This is called a 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 depicted in Figure .

If we were to use this stencil to compute $u^n_{i,j}$, we would also require points that we don't represent and are outside of the problem's domain, for example $u^n_{0,j}$ and $u^n_{i,Ny+1}$. Typically,

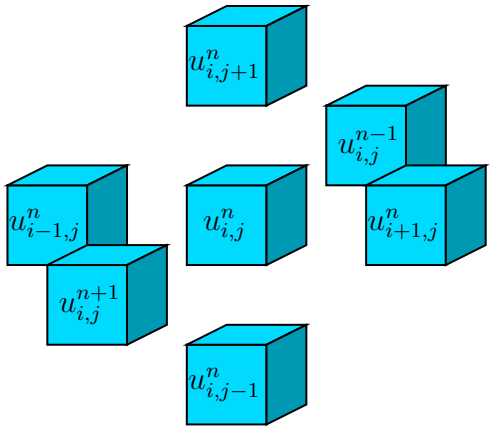


Figure 1: Spacial depiction of the computational stencil.

we would have to test if we are currently updating the boundary to make sure we don't go off the edge of the data. Alternatively, we can pad the domain for dummy values of these "outside" grid locations so that we don't have to modify the stencil on the boundaries. Then, we need only force these padded grid points to have the correct value. This is called the method of *ghost cells*.

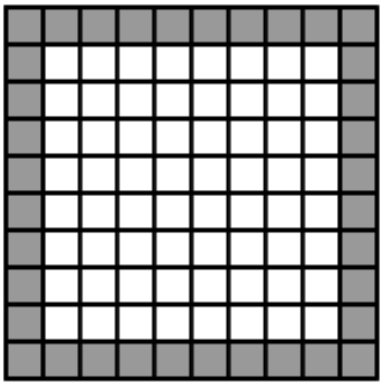


Figure 2: Ghost cells surrounding our the grid so that we can update all of the interior grid cells with the computational stencil.

To determine what values the ghost cells 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 other ghost cells:

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

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 steps

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

The serial version is implemented for your inspection in `P1_serial.py`.

Domain Decomposition

To parallelize this computation, we partition the domain among the processes, giving each process responsibility for a small piece of the domain. In this problem, we will use a rectangular grid decomposition with `Px` and `Py` processes in the x- and y-directions, respectively.

Similar to the serial program, in order to compute u^{n+1} each process will require data from grid cells that it is not responsible to update. If *each* process uses the ghost cell method – it collects and uses information on its border but does not compute the update stencil at those points – draw what needs to be communicated between processes at each iteration with an image similar to Figure 2.

Implementation

Implement a parallel version of `P1_serial.py` using the rectangular grid domain decomposition above.

You may use any communicator strategy that you like. For example:

- Using the `COMM_WORLD` throughout.
- Using `Split` to create your own communicator(s) as we discussed in class.
- Using `Cartcomm` – a specific communicator designed for Cartesian topologies like this.

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 set any way you like: Either computed locally on each process or computed on one process and distributed.

Document your choices and strategy in `P1.pdf`.

HINT: Debug with an interactive session on a compute node with up to $(\texttt{Px}, \texttt{Py}) = (4, 2)$ before going to 32 processes with the headnode.

HINT: In `Plotter3DCS205.py` we have provided two classes that will help you plot 3D data. The first class, `MeshPlotter3D`, is used in the provided serial version. If this class is used in an **interactive session on a compute node** 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 data (i.e. On some process, `u[1,1]` might represent $u_{32,32}$. Here, $(1, 1)$ 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. Both can be

used to efficiently find errors or impress a friend.

sendrecv

Make an implementation `P1A.py` that uses `sendrecv` or `Sendrecv`.

Isend/Irecv

Make an implementation `P2B.py` that uses `ISend/IRecv` or `Isend/Irecv`.

Analysis

With $(\texttt{Nx}, \texttt{Ny}) = (256, 256)$, use the headnode to measure the time/iteration and the speedup with various $(\texttt{Px}, \texttt{Py})$ pairs for $\texttt{Px} * \texttt{Py} \leq 32$. Explain the 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 with theoretical and/or experimental results.

Submission Requirements

- `P1A.py`: Completed sendrecv implementation.
- `P1B.py`: Completed Isend/Irecv implementation.
- `P1.pdf`: Explanations and/or plots.

Problem 2 - Hello CUDA

This problem is intended as a gentle introduction to CUDA programming. You will use the `pycuda` library to interact with a GPU directly from the Python language.

In this problem, you will be modifying some given sample code to perform the so-called SAXPY function (**S**ingle-precision **A**lpha **X** **P**lus **Y**) on n-dimensional vectors. The SAXPY function is written as

$$z = \alpha x + y$$

where x, y, and z are vectors and α is a scalar. Since each element of z can be computed independently from each other element, the SAXPY function is a great candidate for parallelization on the GPU.

Complete the SAXPY kernel

Fill in the body of the SAXPY kernel declared in `P2.py`. Do not change any of the other code or use any kind of loop within `saxpy_kernel`, but allow it to function for any number of threads greater than or equal to the number of elements.

Obtain a performance benchmark

Measure the total time it takes to run your kernel 1000 times. What is the average time in seconds to run your CUDA kernel? Do not include memory transfer times in your benchmarks.

Manipulate the block and grid size

The sample code is not very efficient because `saxpy_kernel` is being invoked with too many blocks and not enough threads per block. Modify the block and grid sizes so the code still runs correctly, but you make as-efficient-as-possible use of the GPU. Explain why you think your new configuration is making more efficient use of the GPU and support your claim with performance timings.

Performance Model

Approximate the bandwidth and latency of the CPU-GPU transfer by copying arrays of various sizes to and from the GPU.

Using the latency and bandwidth data and the speed of the GPU computation, write a performance model for transferring three vectors of size N, performing a SAXPY operation M times on them, and transferring one vector back to CPU.

What is the break-even point for CPU vs GPU computation? That is, how many operations M do we need in order for the GPU time to beat the time for simply computing M operations on the CPU (where no transfers are needed)?

HINT: Take a look at MPI_Bandwidth.py in http://www.cs205.org/resources/ which we used in class to determine the latency and bandwidth of Resonance's MPI network.

Submission Requirements

- `P2.py`: Final python code
- `P2.pdf`: Explanations and/or tables.