

1 Introduction & Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework04, put each programming exercise into its own subdirectory of this directory, zip the entire Homework04 directory up into the file Homework04.zip, and then submit this zip file to Homework #4. Make sure all the code and document files are included in the zip file.

This exercise set is a continuation of the matrix vector multiplication examples we did in class. We will be expanding this to distributed matrix matrix multiplication, analyze the implementation, and then use it to experiment with graph connectivity. Recall from your linear algebra class that A is a real $n \times m$ matrix if

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,m} \end{bmatrix}$$

where each $a_{i,j} \in \mathbb{R}$ for $i = 1, 2, 3, \dots, n$ and $j = 1, 2, 3, \dots, m$. In this notation, the matrix has n rows and m columns.

If A and B are both of dimension $n \times m$, then the sum is the $n \times m$ matrix C with entries given by

$$C_{i,j} = a_{i,j} + b_{i,j}$$

for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$.

If A is an $n \times m$ matrix and B is an $m \times k$ matrix, then the product AB is matrix C , where

$$C_{i,j} = \sum_{l=1}^m a_{i,l} b_{l,j}$$

for $i = 1, 2, \dots, n$, and $j = 1, 2, \dots, k$; so C is a new $n \times k$ matrix. So the (i, j) entry of C is the dot product (inner product) of the i^{th} row of A with the j^{th} column of B .

The transpose of $n \times m$ matrix A is a $m \times n$ matrix denoted as A^T and defined such that $(A^T)_{ij} = A_{ji}$. In other words, the (i, j) entry of A^T is the (j, i) entry of A .

2 Programming Exercises

1. Write a set of functions to perform the basic matrix operations of addition, subtraction, multiplication, and transpose. As we have done in class, keep the matrices as one-dimensional arrays instead of two-dimensional arrays, makes manipulation a bit easier. Also implement both serial and parallel (using MPI) versions of these functions. Test them on small matrices so you know the output is correct before working with big matrices.

- (a) For addition and subtraction split up both matrices between your available processors. Scattering and gathering is probably the best approach here. You can use our padding trick or look in to the vector forms of the scatter and gather. In either case, you are to allow the user to input the size of the matrices. As usual, for testing, populate the matrices with random doubles.
- (b) For multiplication of $C = AB$, there are a number of ways to approach this. If you follow the matrix vector multiplication example from class you could give all the processors matrix B and portions (rows) of A . The local matrix multiplication will then be the same rows of C that A had. That is, if the local matrix L had rows 3 to 7 of A then LB will have rows 3 to 7 of C .

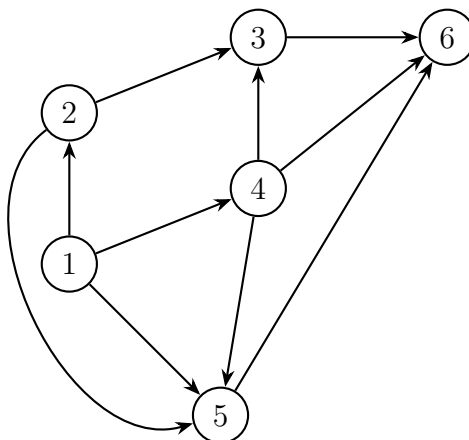
This is not the only way to break the work up. For example, say you gave processor 1 rows 3 to 7 of A (call this submatrix L) and columns 9 to 12 of B (call this submatrix M) then the product LM is defined (columns of L match rows of M), its size is 5×4 and the entries in this matrix is the portion (block) of C between rows 3 and 7 and columns 9 and 12.

Now you might think that this is a bit cumbersome since C++ is row major scattering the columns will require some manipulation and hence may degrade the performance of the application. You can create your own data type that will stride in a nice way to extract a column from a row major array. Another option is to take the transpose of B , the second matrix. So A is now $n \times m$ and B^T is $k \times m$, so now they have the same number of columns (but possibly different numbers of rows). Keep in mind that the columns of B are the rows of B^T . Now if you take rows 3 to 7 of A and rows 9 to 12 of B^T , dot product all the extracted rows of A with all the extracted rows of B^T you still get the entries of C between rows 3 and 7 and columns 9 and 12. Be careful with the bookkeeping here but using B^T will make scattering a bit easier.

There are many other layouts that can be done with this try several and in your comments discuss why you chose the method you did.

- (c) For the transpose we saw how to extract a column in a slick way using a user-defined data type. A well-formed gather on these columns back to process 0 will transpose the matrix. The only snafu to this is that with this setup the number of columns is restricted to be the number of processors. Again with a well formed set of sends and gathers, and probably a loop, you can do any size matrix.

2. Once the matrix multiplication is working well we will now use it to do an empirical connectivity experiment in graph theory. Consider the following digraph.



The adjacency matrix for this graph is

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where a 1 means that there is an edge going from the row number node to the column number node. So the 1 in the (1,2) position means that there is an edge from node 1 to node 2. Note that column 1 is all zeros meaning that there is no edge coming into node 1 and the zero row at the bottom means that there are no edges coming out of node 6. Now if we square A we get,

$$A^2 = \begin{pmatrix} 0 & 0 & 2 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This means that you can get from node 1 to node 3 in two steps (the power of A) and that there are two ways to do that, paths 1-2-3 and 1-4-3. Also note that there is no way to get from node 4 to node 5 in 2 steps.

$$A^3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This says there are 4 ways to get from node 1 to node 6 in three steps. But no other node pair are connected with three steps.

$$A^4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

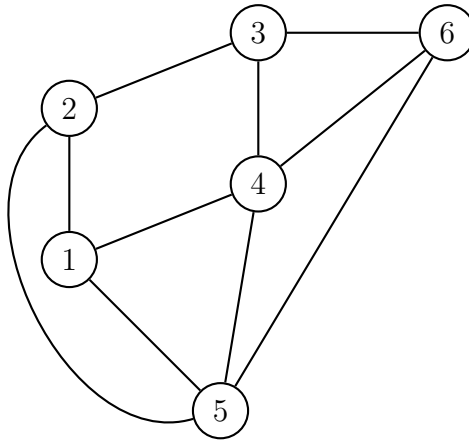
This says there is no way to get from any node to any other node in 4 steps. Since A^n , for $n > 4$, is also the zero matrix, there is no way to get from any node to any other node in more than 4 steps.

If we add all these together we get,

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 3 & 6 \\ 0 & 0 & 1 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

So any nonzero entry means that you can get from the row node to the column node and a zero means that there is no way to get from the row node to the column node. Furthermore, it tells you how many ways you can do it. For example, there are three ways to get from node 4 to node 6 (paths 4-6, 4-3-6, and 4-5-6). Also, there are no paths that connect node 5 to 4.

In the case of an undirected graph, like below, our adjacency matrix will be symmetric.



The adjacency matrix for this graph is

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Then

$$A^2 = \begin{pmatrix} 3 & 1 & 2 & 1 & 2 & 2 \\ 1 & 3 & 0 & 3 & 1 & 2 \\ 2 & 0 & 3 & 1 & 3 & 1 \\ 1 & 3 & 1 & 4 & 2 & 2 \\ 2 & 1 & 3 & 2 & 4 & 1 \\ 2 & 2 & 1 & 2 & 1 & 3 \end{pmatrix}$$

and

$$A^3 = \begin{pmatrix} 4 & 7 & 4 & 9 & 7 & 5 \\ 7 & 2 & 8 & 4 & 9 & 4 \\ 4 & 8 & 2 & 9 & 4 & 7 \\ 9 & 4 & 9 & 6 & 10 & 7 \\ 7 & 9 & 4 & 10 & 6 & 9 \\ 5 & 4 & 7 & 7 & 9 & 4 \end{pmatrix}$$

which shows us that within three moves we can get from any node to any other node. In fact, had we taken the sum of A and A^2 we would have,

$$A + A^2 = \begin{pmatrix} 3 & 2 & 2 & 2 & 3 & 2 \\ 2 & 3 & 1 & 3 & 2 & 2 \\ 2 & 1 & 3 & 2 & 3 & 2 \\ 2 & 3 & 2 & 4 & 3 & 3 \\ 3 & 2 & 3 & 3 & 4 & 2 \\ 2 & 2 & 2 & 3 & 2 & 3 \end{pmatrix}$$

which says that we can get from any node to any other node in one or two steps. With larger matrices these values can get very large very quickly. If you are not worried about how many pathways you can take from node to node, just if there is at least one, then you can replace all the non-zero numbers in the matrix by 1 and still get the same connectivity information. Specifically, if you start with an adjacency matrix A and compute A^2 , then the non-zero entries represent nodes connected by two steps. If you take A^2 and replace all the non-zero entries with 1, call that matrix B . Then AB will have the same non-zero entries as A^3 does. Hence AB and A^3 both have non-zero entries for all nodes that are connected by three moves.

In general to empirically determine if two nodes i and j are connected we would take the adjacency matrix and raise it to higher powers and look to see if the (i, j) entry is non-zero. Equivalently, take any power result and replace all the non-zero entries by 1 and then multiply by A and continue. In addition, we can keep a running total by adding the next result to an accumulation matrix like we did above. Specifically, we could let C be this matrix and each time we take another power of A add it on to the current C . In all we would have,

$$C = A + A^2 + A^3 + A^4 + \dots$$

Again the non-zero values of C can be replaced by 1 to simply represent the existence of a connection.

One question at this point is how many powers of the adjacency matrix do we need? We will take a naive approach to this question. We know that A^n represents node connections using n steps. If two nodes are connected then the path between them will, at the very most, visit all the nodes of the graph. So if there are n nodes in the graph by the time we calculate A^n we will have all the data we need. Another way to look at this is that if we get a connection of two nodes from A^m where $m > n$ then we must have visited a node in that path at least twice, hence we went through a cycle and we would have found an equivalent path without doing that cycle, and we would have found it on some power of A that is less than or equal to n .

The program we will write is going to simulate large directed and undirected graphs. Use this empirical method discussed above to determine connectivity and calculate the amount of connectivity of disconnectivity under different edge densities. Specifically the program should,

- (a) Ask the user for the number of nodes in the graph.
- (b) Ask the user if they want to use a directed or undirected graph.

- (c) Ask the user the node density to use. There are lots of ways to do this but here is one. Have the user input the density as a decimal number from 0 to 1. If there are n nodes then the adjacency matrix will be $n \times n$ and hence have a possibility of n^2 matrix entries, i.e. n^2 possible edges. So if we think about the density number as the fraction of full edges we could just take the input density times n^2 and put that many edges, at random, into the matrix. This will be executed a bit differently depending on whether or not the graph is a digraph. If it is a digraph then we would proceed as above. If it is an undirected graph then we would put half this many in the upper triangular portion and then its mirror image in the lower triangular portion. This can be done, of course, at one time, if we select the (i, j) position at random then we also put a 1 in the (j, i) position.

For example, say that we have 5000 nodes and the user selects a density of 0.2591. Then we would calculate $5000^2 \cdot 0.2591 = 6,477,500$, so we will insert 6,477,500 edges into a directed graph at random. If we are using an undirected graph we would put in 3,238,750 random edges in two locations that are mirror images of each other. This may seem like a lot but it is 25.91% of the total possible edges.

We do not want to overwrite a new edge into a position of an edge that has already in the graph, so if an entry we want to set to 1 is already 1 we will simply choose another random edge to insert. There is one special case when working with the undirected graph, if you put an edge on the diagonal (representing a loop at a vertex), there is no mirror image. Since these will be relatively few in number we will not worry about its mirror image.

- (d) Ask the user the number of trial simulations they want to do with these attributes.
- (e) Run this number of trials on the given number of nodes, graph type, and edge density. For each trial find the percentage of node pairs that are disconnected. Take the average of these disconnected percentages and output it to the console.
- (f) The program should, of course, use as much parallelism as possible. From your matrix multiplication code you will be able to calculate A^n in parallel. You also have other operations such as addition, possibly transpose, truncation to 1's, and counting that can easily be divided up to several processors.
- (g) As usual, test this on small graphs to verify the calculations are correct as well as on a serial version of the program. When you deal with very large matrices you will probably want to not do the serial version, so maybe allow the user to turn this on or off.
- (h) Run this program on various size graphs and various density settings. What do you notice about the correspondences?
- (i) Put timing around the serial and parallel versions of the program and run moderate size tests to determine if your algorithms are strongly scalable, weakly scalable, or neither.