

Assignment 3–Cannon’s Algorithm

This assignment is due at class time Wednesday, 27 October 2004 and is to provide you with experience doing matrix multiplication in MPI.

The basic assignment is to implement Cannon’s algorithm for matrix multiplication of square matrices by means of checkerboard multiplication of subblocks.

In general, doing something clever with matrices can be extremely tedious, since it can require very great care with fencepost errors and boundary conditions on subscripts. Although it’s necessary that you be able to do this sort of thing, it’s not necessarily the point of an exercise in parallel programming. The following rules are intended to let you focus on the parallel programming part and not on the general purpose get-the-subscripts-right part.

1. Your program should work using a 3×3 grid of processors. If your social security number ends in 0, 1, 2, or 3, use processors 1 through 9. If your social security number ends in 4, 5, or 6, use processors 10 through 18. If your social security number ends in 7, 8, or 9, use processors 19 through 27. This will help to balance the load.
2. Do not hard code a 3×3 grid into your code. You *may* use a defined constant of 3 for the number

of processors on eachside of a square array, but your code should use either variables or these defined constants and should not rely on inherent properties of the number 3.

3. You will be provided with sample data matrices with $3 \times 32 = 96$ rows and columns, $3 \times 128 = 384$ rows and columns, and $3 \times 512 = 1536$ rows and columns. All data entries will be double precision floating point numbers of data type **double**. Each data matrix will start with the integer number n of rows/columns and be followed by the n^2 entries of the two matrices in row major order (i.e., row 1, followed by row 2, ...).
4. You may statically allocate global matrix variables of size 1536×1536 to simplify the handling of the matrices. Again, however, your code should be generic, although you may assume for convenience that your data comes in rows/columns that are multiples of the number of processors.
5. Your main program should read the data matrix, call a function to do the multiplication using Cannon's algorithm, and then call a naive matrix multiplication and a function that will verify that the two matrices are "the same".
6. You can't ever rely on floating point numbers being "equal." You may use an epsilon of 10^{-6} as a test of equality. That is, if your Cannon algorithm and your naive algorithm product elements differ by less

than $\varepsilon = 10^{-6}$, then you may consider them to be equal.

7. I recommend that you *not* allocate space in your checking program for the entire product matrix. Dimension only enough space for a single row of the product. Compute the product row by row and check it row by row. Remember, this part of the program is only for checking, so we don't care so much whether it is all that efficient.
8. This program consists of two parts. The first part is the computation of a matrix product in blocks, pasting together the partial product values computed in each block. The second part is the passing of block values in a predefined way from one processor to another, shifting the A matrix left and the B matrix up.

I recommend that you develop code for these two parts separately. Write code that moves data in the correct way, and pass values from processor to processor so you can verify that you have the data movement correct. Separately, I suggest you write sequential code on a single processor to do the multiplication in blocks. That way, you can get the subscripting right without also having to bear the debugging burden of running on multiple processors.