Parallel Matrix Multiplication

In this assignment, you will implement distributed-memory matrix multiplication.

1 Read

In this assignment, you will design and implement a program for parallel matrix multiplication on a distributed memory multicomputer. Matrix multiplication is a common and critical operation in many domains, particularly for solving systems of equations.

1.1 Text

For more background on MPI, read:

1. Introduction to Parallel Programming, chapter 3
2. Programming Massively Parallel Processors, chapter 19

1.2 Matrix Multiplication

Consider an $m \times n$ matrix A:

$$A_{m,n} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

and an $n \times p$ matrix B:

$$B_{n,p} = \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,p-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,p-1} \end{pmatrix}.$$

The product $C = A \times B$ is an $m \times p$ matrix in which each element is computed as:

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j} \tag{1}$$

for $i = 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, p-1$. Informally, to compute the value at row i and column j of the product matrix $(C_{i,j})$, you multiply together corresponding elements of row i of A and column j of B, and add together all the results. Note:

1. The product of an $m \times n$ matrix and an $n \times p$ matrix is an $m \times p$ matrix.
2. Mathematical practice is to use one-based index values, but we're computer scientists. And in practical terms, because we will be storing these matrices in memory, the formulation here uses zero-based indices.

1.3 Example

Given $A = \begin{pmatrix} 1 & 3 & 6 & 7 \\ 4 & 6 & 3 & 9 \\ 5 & 5 & 0 & 2 \end{pmatrix}$ and $B = \begin{pmatrix} 4 & 2 \\ 6 & 3 \\ 6 & 9 \\ 0 & 9 \end{pmatrix}$, then $C = A \times B = \begin{pmatrix} 58 & 128 \\ 70 & 134 \\ 50 & 43 \end{pmatrix}$.

If you have no recent exposure to matrix multiplication, you should try to work out this computation by hand to ensure that you understand the basic algorithm.

1.4 Sequential Implementation

This section presents a simple sequential implementation of matrix multiplication. The complete listing appears in Appendix A, and the output from the sample matrices in the program is shown in Appendix B.

The `MAT_ELT` macro simplifies access to elements at a given location a matrix. The parameters are as follows.

- `mat`—pointer to the matrix itself; should be of type `(int **)`
- `cols`—number of columns in the matrix
- `i`—row to access
- `j`—column to access

Note the leading `*` in the definition of the macro. This causes it to *dereference* the element at the given row and column. In other words, the macro returns the *value* stored at location (i, j), not the *address* of that location.

Macro

```
3 #define MAT_ELT(mat, cols, i, j) *(mat + (i * cols) + j)
```

The matrix multiplication itself is a straightforward implementation of Equation (1). We pass to the `mat_mult` function the matrices (`c`, `a`, and `b`), and their dimensions: `a` is `m` × `n` and `b` is `n` × `p`. The order of the matrices is intended to be mnemonic of the operation itself: `c` = `a` × `b`.

Note the use of the `MAT_ELT` macro to access the data at the given row and column. The result is stored "in place" in the output matrix `c`.

Multiplication

```
23 void
24 mat_mult(int *c, int *a, int *b, int m, int n, int p)
25 {
26   for (int i = 0;  i < m;  i++) {
27     for (int j = 0;  j < p;  j++) {
28       for (int k = 0;  k < n;  k++) {
29         MAT_ELT(c, p, i, j) += MAT_ELT(a, n, i, k) * MAT_ELT(b, p, k, j);
30       }
31     }
32   }
33 }
```

The `main` function invokes `mat_mult`. Because we're just passing pointers to the function itself, we also have to pass along the dimensions of the three matrices. Also shown is how to invoke the `mat_print` function that prints out the contents of a matrix.

Main

```
57   mat_mult((int *)C, (int *)A, (int *)B, 3, 4, 2);
58   mat_print("A", (int *)A, 3, 4);
59   mat_print("B", (int *)B, 4, 2);
60   mat_print("C", (int *)C, 3, 2);
```

2 Set Up

Choose one of the following open-source implementations of MPI. Both are very robust and well-supported.

- MPICH
- Open MPI

Although I've used both in the past, I currently use Open MPI on my development machine.

If you use a Mac, both packages are available for easy installation using Homebrew (packages `openmpi` and `mpich`). You can install only one of them at a time.

3 Solve

The focus of this homework is to design and implement a parallel program that performs matrix multiplication on a distributed-memory multicomputer using the Message Passing Interface (MPI).

> We have emphasized how parallel programs can solve problems *faster*. Parallel computers also can solve problems that are *larger* than can be solved on a single processor

machine. Consequently, *do not* assume that the matrices are small enough to fit into the memory of each processor; you *must* partition the matrices appropriately across *all* the processors in the multicomputer.

1. (10 points) Following the PCAM design strategy, identify a partitioning of the problem. Typical partitionings for an operation on 2D matrices are horizontal or vertical stripes (1D partitioning) or blocks (2D partitioning).

   Be sure to consider the pattern of communication among processors as you select a partitioning. As you may have observed from Equation (1) or the sample implementation in Appendix A, in order to calculate the result at cell $C_{i,j}$, the processor responsible for that cell will *eventually* need access to all the values in row i of matrix A and column j of matrix B.

   What partitioning did you choose? Justify your choice.

2. (10 points) You will need to generate sample matrices of various sizes and shapes for experimentation on your algorithm and implementation.

   Write a program that generates a matrix of random integers of a given size and shape (i.e., number of rows and columns). Because both your test data generator and your matrix multiplier will read and write matrix data, create functions (e.g., `read_matrix` and `write_matrix`) in their own source file that you can compile separately and link into both executables.

   The format in which you store your matrix is up to you. One strategy would be to store one row of the matrix per line of the file, separating values by a delimiter (e.g., comma or space). You may want to use first line of the file to store the dimensions of the matrix, making your data files self-describing.

   In your write-up, document the format of your data files including the program you use to generate them.

3. (10 points) In order to verify the correctness of your parallel implementation, it's helpful to have a known-good sequential implementation.

   Write a sequential program that reads matrices in the format you designed for question 2, multiplies them together, and stores the result in the same format. You are welcome to use the program in Appendix A as a starting point. Verify that your implementation works on some toy problems.

   If you already know (or want to learn) how to use other programs that support matrix calculations (e.g., MATLAB, Octave, or Mathematica), you are welcome use them for verification. You will have to use a file format that works with your program of choice.

   Describe in your write-up how you chose to verify your calculation. If you wrote your own program (i.e., didn't use MATLAB or similar software), please include the code in your submission.

4. (35 points) Implement parallel matrix multiplication using MPI. Observe the following guidelines.

   1. Load the input matrices (A and B) from disk. The processor that loads a matrix should partition the matrix appropriately (based on your partitioning strategy) and forward the partitions to other processors.

2. *All* processors should participate in the matrix multiplication. This *includes* the processor(s) that loaded data from disk. In other words, a processor that does the loading should also be responsible for computing a partition of the result.
3. After the parallel matrix multiply, collect the results from each processor and have a single processor write the resulting matrix to disk.
4. Verify the results of each parallel computation using the software you chose for question 3.

5. For *at least three* matrices of various sizes ("modest," "large," and "ginormous"), run multiplication experiments with varying numbers of processes. If your CPU has c *physical* cores (*not* hyperthreaded cores), include runs for at least $p = 1, 2, \ldots, c$ and $2c$ processes. Measure the time t_p for each run. Be sure that t_p includes *all* overhead due to parallelization.
   (a) (10 points) Report your performance results as illustrated in Table 1.

Operand Sizes	p	t_p (s)	s	e
$a \times b$, $b \times c$	1	10.0	1.0	1.0
	2	5.0	2.0	1.0
	⋮			
$x \times y$, $y \times z$	1	8.0	1.0	1.0
	⋮			

Table 1: Format for performance report

   (b) (10 points) Following the pattern shown in class, create three plots that show $t_p \times p$, $s \times p$, and $e \times p$ for your experiments. Each of the three plots should contain three curves, one for each matrix.

6. Discuss the overall performance results for t_p, s, and e as reported for question 5.
   (a) (5 points) How did performance vary with number of processes and matrix size?
   (b) (5 points) What, if anything, surprised you about the results you obtained? Why?
   (c) (5 points) What was the most complex/difficult part of the assignment? Why?

4 Submit

Submit your solution and write-up from Section 3 to the course web site.

Question:	1	2	3	4	5	6	Total
Points:	10	10	10	35	20	15	100

A Sample Implementation

Matrix Multiply

```c
#include <stdio.h>

#define MAT_ELT(mat, cols, i, j) *(mat + (i * cols) + j)

void
mat_print(char *msge, int *a, int m, int n)
{
  printf("\n== %s ==\n%7s", msge, "");
  for (int j = 0;  j < n;  j++) {
    printf("%6d|", j);
  }
  printf("\n");

  for (int i = 0;  i < m;  i++) {
    printf("%5d|", i);
    for (int j = 0;  j < n;  j++) {
      printf("%7d", MAT_ELT(a, n, i, j));
    }
    printf("\n");
  }
}

void
mat_mult(int *c, int *a, int *b, int m, int n, int p)
{
  for (int i = 0;  i < m;  i++) {
    for (int j = 0;  j < p;  j++) {
      for (int k = 0;  k < n;  k++) {
        MAT_ELT(c, p, i, j) += MAT_ELT(a, n, i, k) * MAT_ELT(b, p, k,
        ↪  j);
      }
    }
  }
}

int
main(int argc, char **argv)
{
  int A[3][4] = {
    { 1, 3, 6, 7 },
    { 4, 6, 3, 9 },
    { 5, 5, 0, 2 }
  };
```

```
43
44    int B[4][2] = {
45      { 4, 2 },
46      { 6, 3 },
47      { 6, 9 },
48      { 0, 9 }
49    };
50
51    int C[3][2] = {
52      { 0, 0 },
53      { 0, 0 },
54      { 0, 0 }
55    };
56
57    mat_mult((int *)C, (int *)A, (int *)B, 3, 4, 2);
58    mat_print("A", (int *)A, 3, 4);
59    mat_print("B", (int *)B, 4, 2);
60    mat_print("C", (int *)C, 3, 2);
61  }
```

B Sample Output

Output from the program in Appendix A. Note that the values here correspond to the sample matrices in section 1.3.

```
== A ==
            0|     1|     2|     3|
    0|      1      3      6      7
    1|      4      6      3      9
    2|      5      5      0      2

== B ==
            0|     1|
    0|      4      2
    1|      6      3
    2|      6      9
    3|      0      9

== C ==
            0|     1|
    0|     58    128
    1|     70    134
    2|     50     43
```