Exam 1, directions and hints

As long as you do what is required, you do not have to follow these hints.

Turn in a directory that is your userid, with an MPI and an OpenMP directory that contains what you are asked to turn in for each one.

Post questions to Piazza.

The MPI problem

We will do a reduction using MPI.

Write and run an MPI program to compute the following reduction:

```
for (int i = 0; i<n; i++) {
  t = t + a[i];
}
print(t);
```

Within the same program,

1. time and run this using an MPI collective communication.
2. Time and run this using send/recv or send/irecv that forms a fan-in tree such as used by the MPI reduction.

Run each method twice and use the second timing to help reduce cache effects, i.e., the first run will bring stuff into cache and make the timing of the second run more dependent on the algorithm and less dependent on cache effects

Time the two versions using arrays of size 10000, 100000 and 1000000.

Run on one node with 4 and 16 processes.

For full credit, do not use hardcoded code for what processes are doing. You should be able to write a loop whose body performs one step of the reduction, with somewhat arithmetic determining at each iterations which processors send and receive data, and which are idle.

For this step, turn in your code and a file with the timings.

The OpenMP problem

In this part of the exam, we will implement a checkerboard matrix multiply in OpenMP. In a checkboard matrix multiply (i.e., Cannon's Algorithm), each core will compute a single square section of the C result matrix. Instead of performing computation, the program will have each core successively access the parts of the A and B matrices that are needed to perform the matrix multiply. In these instructions we will see how this works.

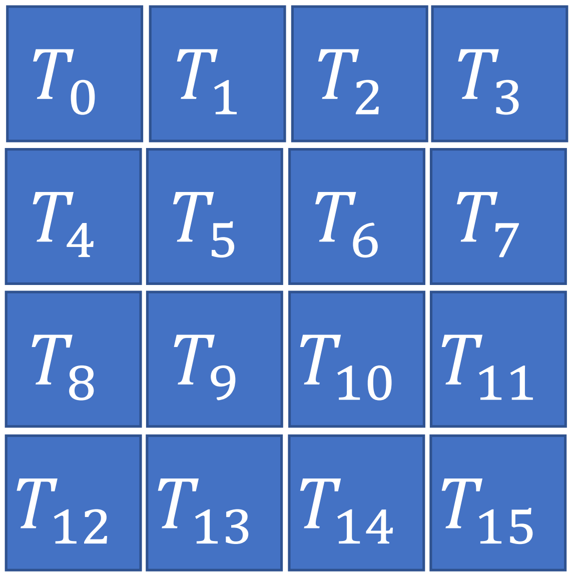
We will run on two problem sizes, 2x2 checkerboard (4 threads) and a 4x4 checkboard (16 threads). You should also perform a sequential matrix multiply.

Arrays are self-initialized, i.e., you will not read data but instead when initializing the arrays, you will assign values to them. Making these values dependent on the row and column, as is done in the code I give you, can help you to find bugs.

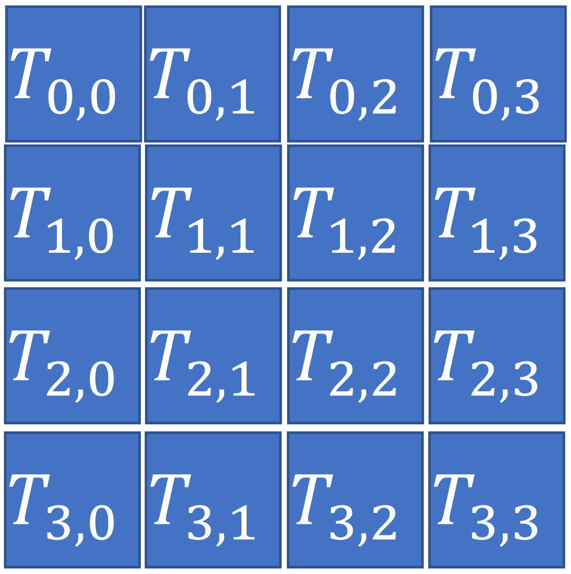
You will turn in your code, and timings for the 1, 4 and 16 thread turns. Your program should run under 1 minute on one core. You may size your data appropriately.

Performing the matrix multiply.

Below we see the A, B and C arrays, broken up into chunks. On the left we see the chunks of the array labeled with the number of the thread that *owns* the chunk, and on the right the chunks labeled with a two dimension logical grid of threads.



T_0 T_1 T_2 T_3
T_4 T_5 T_6 T_7
T_8 T_9 T_{10} T_{11}
T_{12} T_{13} T_{14} T_{15}



$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$

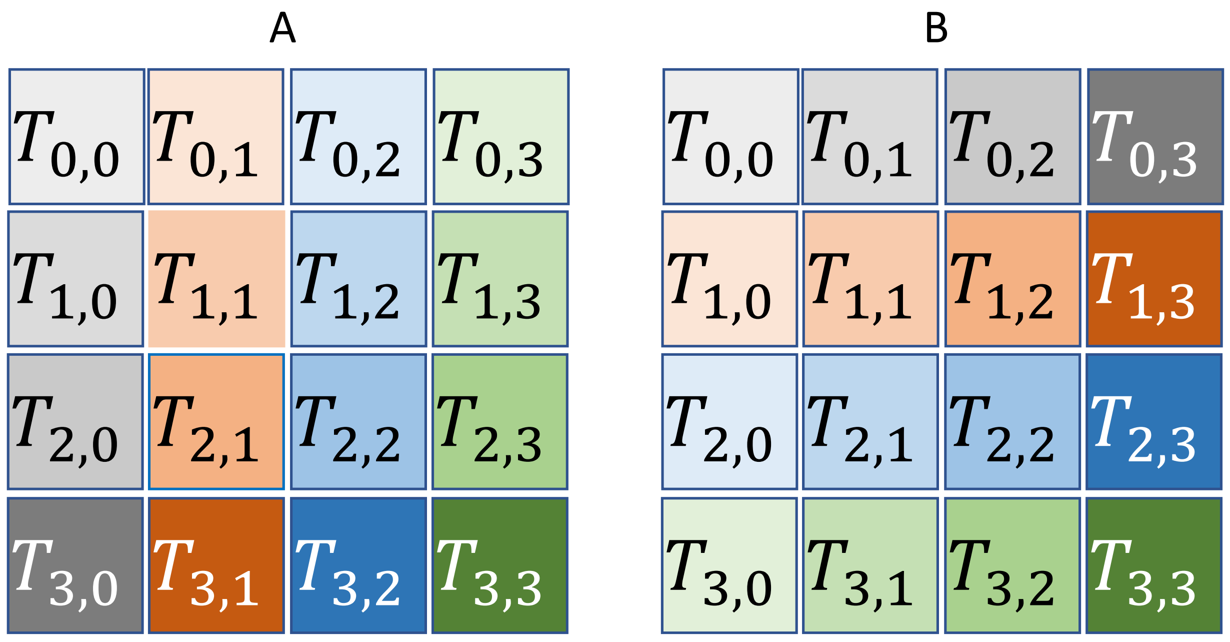
In an MPI implementation of Cannon's Algorithm, we have the following steps:

1. Perform an initial set of shifts on the A and B matrices to properly align them
2. For $\sqrt{P}$ steps, do the following
   i. Multiply the data that is local, and update the C matrix with the partial result
   ii. Shift each block of the A matrix 1 column to the left
   iii. Shift each block of the B matrix 1 column to the right
3. The C matrix contains the result of the matrix multiply

For a distributed (MPI) implementation, this algorithm has the advantage that data is only communicated across $\sqrt{P}$ processes, lowering the communication overhead and making the algorithm more efficient. For a multithreaded implementation, this algorithm has the advantage that data only is moved to $\sqrt{P}$ cores, reducing the overhead of moving the array from cache to cache. Also, by operating on smaller chunks of data a kind of tiling is performed, which also helps cache performance.

The OpenMP Cannon’s algorithm

Below we see the A, B and C arrays, broken up into chunks. On the left we see the chunks of the array labeled with the number of the thread that *owns* the chunk, and on the right the chunks labeled with a two dimension logical grid of threads. The chunk that is owned is the chunk that the thread would contain if it were an MPI process, and is the thread that it would naturally access data in a checkerboard pattern


A
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
B
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$

As with the MPI implementation, the data is not properly lined up to perform the multiply. Thus some threads own an A and a B block that they would never multiply together in a normal matrix multiply. The data can be properly aligned by shifting each A matrix to the left some number of columns, and the B matrix up some number of rows. The number of columns to shift the A matrix blocks is given by the *row* of the block, i.e., the row of the logical thread identifier. The number of rows to shift up the B blocks is given by the *column* of the B block, i.e., the column of the logical thread identifier. Doing this shift gives each thread an A and a B block that would be multiplied together at some point in the matrix multiply.

With an OpenMP program, we don’t actually shift data – it’s all in shared memory that all threads can access, and so the movement would just be wasted work. Instead, change the data that each thread initially accesses to make it access the data that it would were it an MPI process and the data was actually shifted.

The OpenMP Cannon's algorithm

On each thread, the matrix multiply will take $\sqrt{T}$ steps, where T is the number of threads. Let ARRAY_ROWS and ARRAY_COLS be the number of rows and columns in the arrays to be multiplied. To keep this simple, all arrays will be square, and the number of rows and columns will be evenly divisible by $\sqrt{T}$. Let

LOCAL_ROWS == LOCAL_COLS == ARRAY_ROWS/ $\sqrt{T}$ == ARRAY_COLS/ $\sqrt{T}$

Then, at each step, each thread will multiple a LOCAL_ROWS x LOCAL_COLS section of the A array by a similar sized section of the B array, and will compute a partial result for the same sized section of the C array.

The initial "shift" is accomplished by having offsets for the A and B array that adjusts what LOCAL_ROWS x LOCAL_COLS section of the A and B array a thread multiplies together, and another offset that adjusts what elements in the C array this computes.

Each later shift by one 1 column to the left of the A array, and 1 row up of the B array, will be done by adjusting the offsets for the A and B arrays. ~~including the offsets for the C array.~~ The row and column offsets for the C array do not need to be adjusted, since we are always updating the same LOCAL_ROWS x LOCAL_ROWS section of the C array

How I solved this:

1. Figure out the offsets for each thread that give the offset into the A, B and C arrays for what is owned
2. Adjust these offsets for the initial shift.
3. For $\sqrt{T}$ steps
   i. Do the first multiply
   ii. Do the shift by one of A and B (this can be skipped on the last iteration
4. Check your result

How to make this easier:

If you are not familiar with how this works, work this out by hand on a 2x2 or 3x3 3x3 matrix with 9 "threads".

You can do all of your functional debugging without OpenMP – simply add another loop using tid as the index going from 0..3 or 0..15. When you get the right answer, remove that loop, make the appropriate loops parallel, and run in parallel.

Instead of using omp_get_thread_num( ) you can keep a global counter, and as each thread begins executing capture the value of the counter as the threadId and increment it. When debugging with small problem sizes, a thread may finish its work and grab more work before all the threads being executing. This can lead to more than one chunk of work being executed by some tid from omp_get_thread_num( ), which will mess up your offset calculations above, which are based on the threaded.