CMSC416: Introduction to Parallel Computing

Topic: Parallel Algorithms
Date: March 7, 2024

The inspiration behind optimizing matrix multiplication is that the conventional matrix multiplication algorithm does not work for very large arrays because the size of the data is bigger than the caches on the processors.

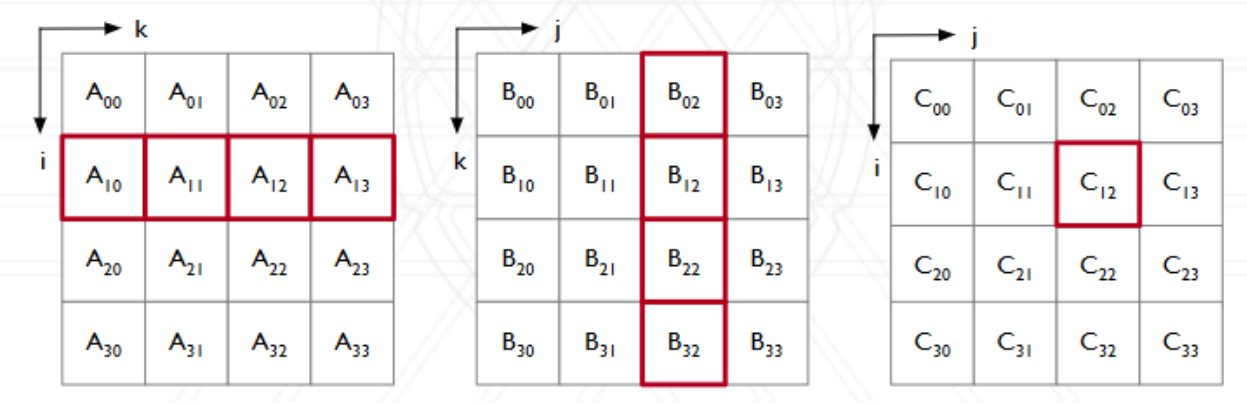
We use blocking to improve cache performance by creating smaller blocks that fit in the cache. We work in the model that there is a faster, smaller memory cache and a bigger, but slower memory available on a computer.

There is a certain number of cache lines that can be placed into the cache. Once that gets filled, old data must be replaced by the new data. This means that the next time you need that old data, that data needs to be loaded back into the cache.

WIth the notion of blocking, we are loading a block of data into the cache which, if our algorithm is implemented well, means that we have cache reuse, where we call the same data within the cache multiple times (and only have to move it once to the cache).

For the case of matrix multiplication where A*B=C, we create smaller blocks of these matrices and compute partial matrices of the product, C. From the example in the slides shown below, we create 16 blocks of each matrix and when computing a block of C, it means that we will have to do 4 block multiplications and add them up. For example,

C_12 = A_10*B_02+A_11*B_12+A_12*B22+A_13*B32



(image from slides)
Recall that in C and C++, matrices are stored in row major. This means that we can bring the rows of A that are needed, but for the B matrix, we fetch elements of B in a column. This will lead to cache misses as the naive implementation will bring in the rows of B in so most of the data loaded into the cache is not used together.

If we smartly compute the partial value of C within each block, then we reduce the wasted data in the cache and increase cache reuse. For example, if the calculate the values of C row wise first, then we will reuse the row of A when computing multiplication (we will only use a different column of B)

Recall that the naive solution for matrix multiplication involves 3 nested for loops as shown below (taken from lecture slides):

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
```

The implementation of this blocked (tile) matrix multiply has three additional nested for loops, as we are computing block multiplications where each block is a B by B sub-matrix. The algorithm looks like (taken from extinguisher slides):

Blocked (tiled) matrix multiply

```
for (ii = 0; ii < n; ii+=B) {
  for (jj = 0; jj < n; jj+=B) {
    for (kk = 0; kk < n; kk+=B) {
      for (i = ii; i < ii+B; i++) {
        for (j = jj; j < jj+B; j++) {
          for (k = kk; k < kk+B; k++) {
            C[i][j] += A[i][k]*B[k][j];
          }
        }
      }
    }
  }
}
```

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
```

So far we talked about how to compute a more efficient matrix multiplication, but only serially. We now move to parallelizing it. We have to figure out how to efficiently store data, in this case the matrices A and B in a distributed manner, and how to efficiently split up the work to compute the product C and to minimize the communication between the processes.

For our first implementation, we will have the processes hold a disjoint set of A and B, and each process will compute a separate (or disjoint) part of C. However, this will require processes to request parts of the A and B matrices that they do not initially hold.

Cannon's 2D Matrix Multiply

The first algorithm we will discuss for parallel matrix multiplication is called Cannon's 2D matrix multiply. This is an older algorithm that is 2D, in the sense that we are arranging the process in a virtual 2D grid. It is not referring to the shape of the data as they will always be two dimensional matrices. Also note that the matrices do not have to be square. The only requirement (for this algorithm and matrix multiplication in general) is that the second dimension of A must be the same as the first dimension of B.

Note: all examples in lecture consider square matrices although this can be generalized for non-square matrices as well.

We first arrange the processes in a virtual 2D grid. We then split up A and B into sub-blocks that correspond to this virtual 2D grid. Similarly, based on which sub-blocks of A and B each process received, they are responsible for computing the corresponding sub-block of C. This will require other processes to send sub-blocks to each other to compute the sub-block of C.

Consider the example given in the lecture slices where there are 16 processes:

$$C_{12} = A_{10} * B_{02} + A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32}$$

0	1	2	3
4	5	**6**	7
8	9	10	11
12	13	14	15

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Process 6 is initialized with sub-blocks A_12 and B_12. However, to compute the corresponding C sub-block, C12, it will need other A and B sub-blocks (A_10, B_02, A_11, B_22, A_13 and B_32). In fact, it cannot do any multiplications without other sub-blocks from other processes.

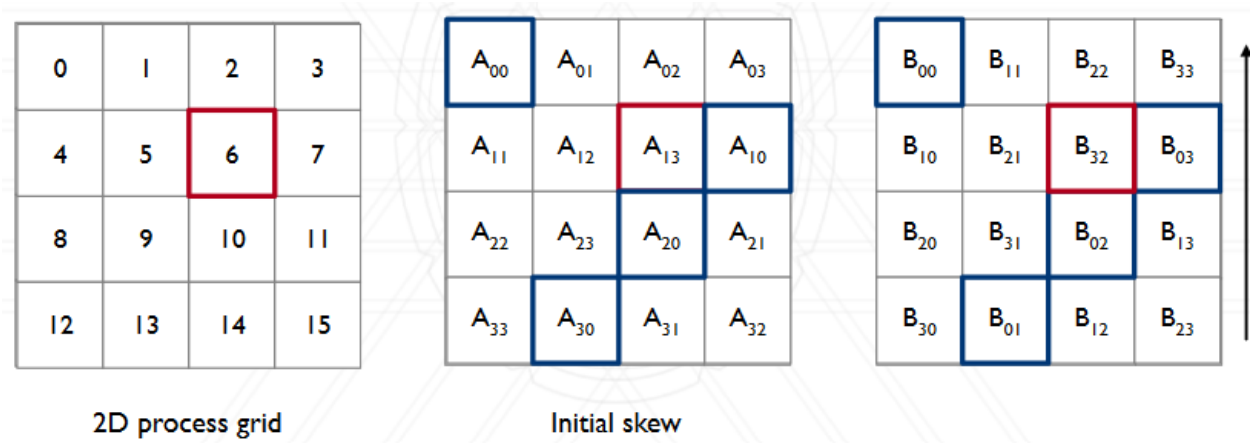
In Cannon's 2D matrix multiple, there is a structured way to send sub-blocks to other processes. To do this, each process sends its A sub-block to the adjacent row process in the virtual 2d grid to the left by i (the x-coordinate of the sub-block). For the B sub-blocks, each process sends its

B-sub-block to the process above it by j (the y-coordinate of the sub-block) Each send will do a wrap around if the process is on the topmost or leftmost edges of the virtual 2D grid.

Another way of saying this is that each A block in row i will be shifted to the left by i, and each B block in column j will be shifted to the top by j. (0-based index)

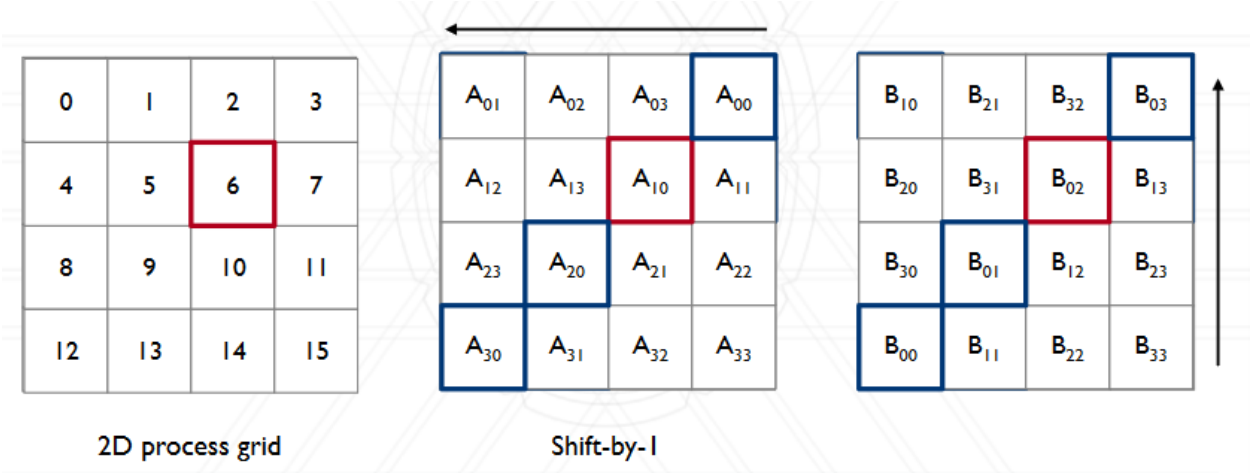
In the example in the slides, this means that process 6, which has the coordinate of (1,2) in the virtual 2D matrix, will send A_12 to process to the left by 1, which is process 5, and will send B_12 to process 2 above it (where we will wrap around), which is process 14.

After each block has been displaced once or initial skew, the data will be moved like this (from extinguisher slides):



In our example, process 6 now has A_10 and B_02. THis means that process 6 can do the first multiplication. Recall that C_12 = A_10*B_02+A_11+**B_12**+**A_12***B_22+**A_13*B_32** (the bold are the data that process 6 currently has after the initial data and first displacement).

We will then shift all blocks by 1. For A blocks, we shift all blocks by 1 to the left. For B blocks we shift up by once. In our example, below shows which blocks each process gets (image from lecture slide)



This means that each process can compute another block multiplication. For example, process 6 can now compute A_10*B_02. Process 6 now has in total the bolded blocks: C_12 = **A_10*B_02**+A_11+**B_12**+**A_12***B_22+**A_13*B_32.**

We continue to shift by one until each process is able to compute all of its block matrix multiplication. In our example, only one more shift by 1 is needed. We will need sqrt(N) transfer steps, where N is the number of processes.

The time complexity of a serial matrix multiplication of N by N matrices is O(N^3) as there are 3 nested for loops.

In the cannon's 2D matrix multiply, we are still doing the same amount total computation, but each step we move 2s^2 data (each block is s by s), and since there are sqrt(p) steps (where p is the number of processes), each process will send 2s^2*sqrt(p) data.

The downside of this algorithm is that there is a lot of communication between the processes. However, we do exhibit strong scaling since we are doing the same number of arithmetic operations total compared to the serial implementation.

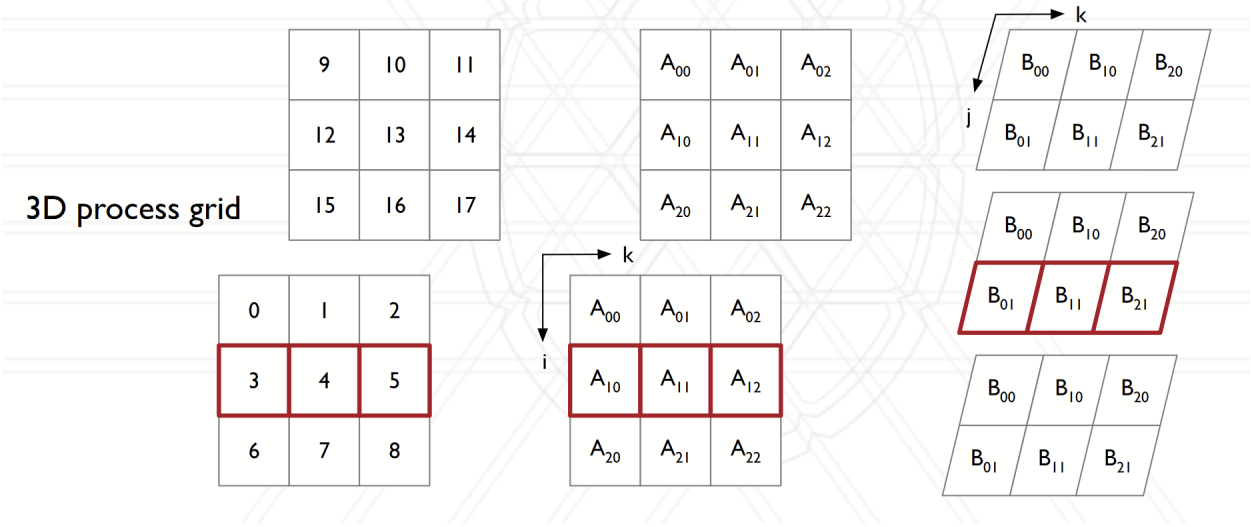
Agarwal's 3d Matrix Multiply

In this algorithm, we are trying to decrease the amount of communication needed to perform the matrix multiplication.

In this algorithm, we will arrange the processes in a 3D virtual grid (instead of the virtual 2D of the Cannon's algorithm). In addition, there will be multiple copies of A and B where each plane of processes will have a full copy. Each process will compute a sub-block of C, but it will be partial and we will need to sum of parts from multiple processes to get the result of a sub-block of C. Finally, there are only 2 phases where communication occurs, once before any computation and one after to combine the partial results of C.

We first assign the processes in a 3D virtual array. In the example from the slides as shown below, there are 18 processes which means this 3D array is 3X3X2:

- Copy A to all i-k planes and B to all j-k planes



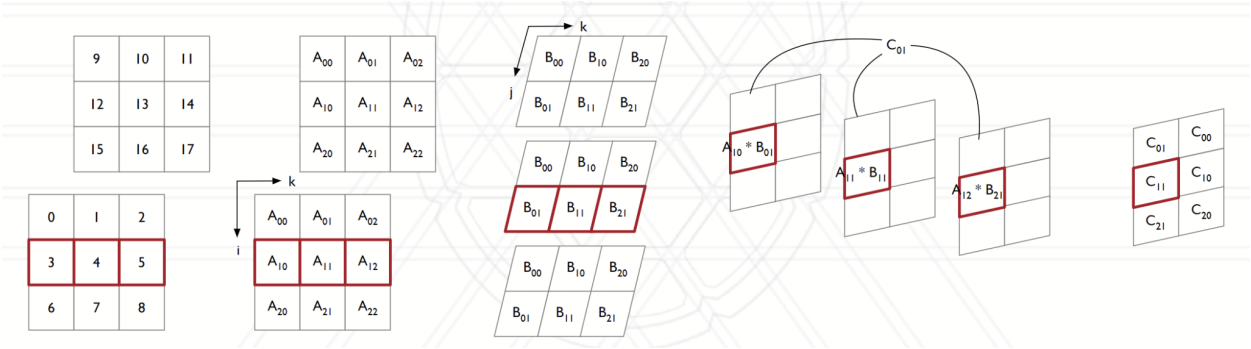
We choose a 2D plane to contain a full copy of the matrix A. In the example, the (i,k) plane contains A, which means we split A into 9 blocks. Each (i,k) slice will contain the same configuration of A. For example process 0 and 9 will have the same sub-block of A, A_00.

How we choose a different plane for matrix B. In the lecture example, we pick the (j,k) plane, which means that matrix B is split into 6 sub-blocks. Similarly, since each slice contains the same configuration, it means that each sub-block of B appears in 3 processes. For example, B_01 is on process 0, 3, and 6.

When implementing, a single plane would get the sub-blocks of a matrix and they would forward that sub-block to their corresponding process on the other slices.

In total, there are now 2 full copies of A and 3 full copies of B stored across all processes.

Now each process matrix multiplies their sub-block of A and B. This is a partial sum of C, and to get the full version of C, we do an all-reduce of the plane that does not contain the full copy of A or B. In our running example, that plane is the (i,j) plane. We all-reduce the (i,j) plane by summing up the partial sums. Below is the visualization of this from the slides.



The main differences between Cannon's and Agarwal's is that Cannon's uses less memory since there is only a combined single copy of both A and B but there are more data movements compared. In Agarwal's there are only 2 data movements. The first is a bcast across a plane (but a single process is not sending to all other processes) and the second is an all reduce from multiple processes to a single process.

Communication algorithms

There are two MPI communication functions we will examine, reduction and all-to all and their runtime. Note that these are running in parallel to be able to run efficiently.

There are two types of reduction, scalar and vector reduction. In scalar reduction, all processes perform a commutative associative operation to a single value. An example is summing a value from all processes. In a vector reduction, instead of a single value, we have an array where each process contributes to the whole vector of values.

Parallelizing Reduction

There are multiple ways to implement the reduce function. The naive way method is for all processes to send their data to the root.

The second, parallel method is to use a spanning tree. This involves organizing the processes into a k-ary tree. A k-ary tree is a tree in which there is a single root, and each node either has children or is a leaf. Each non-leaf non has at most k children.

At the start, the leaves send their data to their parent. Each non-leaf node will wait to receive all the data from their children to apply the commutative associative operation and then send to their parent. This works all the way up until the root gets all of the data and combines it.

For p processes and a k-ary tree, the number of phases is log_k(p).

Parallelizing All-to-all

Recall that in this function, each process sends a unique message to every other process. Each process has an array of data, and in the parameter of this call, it is specified how much of the array is to be sent to each process. A unique part of it will be sent to each process, including itself.
In the naive algorithm, each process sends their data pair-wise to all other processes. The runtime of this is O(n^2).

We can leverage the property that all processes have to send data to each other to make this more efficient. We will construct a virtual topology of a 2D mesh. In phase 1, every process will

send data to its row neighbors. In phase 2, every process will send data to its column neighbors. This means fewer sends are needed and each send has more meaningful data.