Homework - 3

Muhammad Talha Khan (46281174)
CS-230 Distributed Computer Systems

Matrix Multiplication

Consider the example below for multiplying two $n \times n$ matrices **A** and **B** and the result produced **C**, which is also an $n \times n$ matrix.

$$A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{21} & \cdots & a_{1(n-1)} \\ \vdots & & & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{bmatrix} B = \begin{bmatrix} b_{00} & b_{01} & \cdots & b_{0(n-1)} \\ b_{10} & b_{21} & \cdots & b_{1(n-1)} \\ \vdots & & & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & b_{(n-1)(n-1)} \end{bmatrix}$$

$$A * B = C = \begin{bmatrix} c_{00} & c_{01} & \cdots & c_{0(n-1)} \\ c_{10} & c_{21} & \cdots & c_{1(n-1)} \\ \vdots & & & \vdots \\ c_{(n-1)0} & c_{(n-1)1} & \cdots & c_{(n-1)(n-1)} \end{bmatrix}$$

The complexity of multiplying these matrices is $\mathcal{O}(n^3)$. The result to notice here is that each value in C results from the dot product of a row vector of A and a column vector of B. For example the value C_{ij} is the result of the dot product of vector A_i (i-th row of A) and B_j (j-th column of B). The the code that is used for matrix multiplication (listed below) is transformation invariant. Changing the order of the loops does not change the final results.

```
For i = 0, n − 1, do:
    For j = 0, n − 1, do:
        For k = 0, n − 1, do:
            c_{i,j} = c_{i,j} + a_{ik} × b_{k,j}
        endfor
    endfor
endfor
```

Figure 1: Matrix Multiplication Program

This assignment is divided into two parts. **Part 1** demonstrates how the matrix multiplication program maps the matrices to a linear memory and how the computation progresses over time for all the different permutations of the loop ordering. **Part 2** discusses how the matrix multiplication program progresses, for each permutation, given multiple (n) processors where unfolding and parallelization are always performed on the outermost loop.

Data to Memory Mapping

Since the memory (RAM) is a linear array. Stroring data structures such as trees, linked lists and multi-dimensional arrays require special techniques. Two of the most common techniques to map 2D arrays to memory are row-major and Column-major.

Row-Major & Column-Major

In row-major consecutive elements of a row of an array reside next to each other. In Column-major consecutive elements of a column of an array reside next to each other. The figures below help clarify this concept.

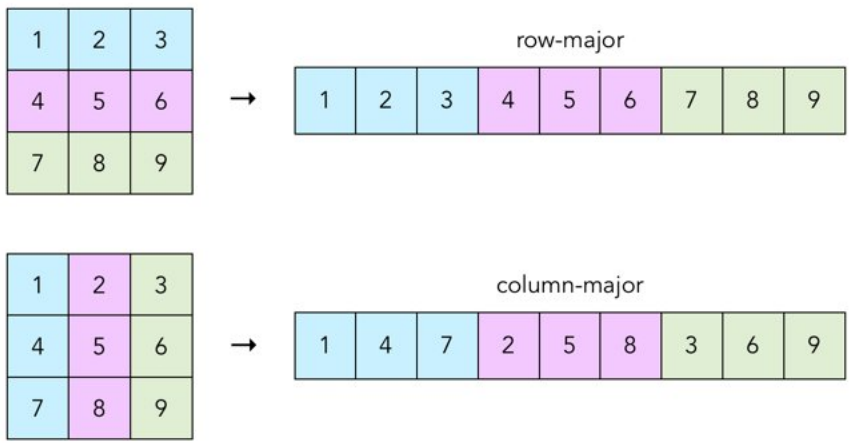

Figure 2: row & Column Major

Most of the programming languages reference 2D arrays in the row and column format respectively. E.g. if an array is named $A[0][1]$ would refer to an element in A at row 0 and column 1. Also, the indexing starts from 0 rather than 1. How $A[0][1]$ is resolved into a single number is the job of the compiler but it is pretty straightforward. For the row-major case the index for A_{ij} is found as follows.

$$Index\ in\ 1D\ Array\ (in\ memory) = (number\ of\ columns\ in\ A * i) + j \tag{1}$$

E.g. In row-major $A[2][1]$ refers to 8 in the 2D array. So in the mapped array its index can be found using equation 1. ((3 * 2) + 1) which is equal to 7. But, since arrays are 0 indexed we start counting from zero and when we reach 7 the underlying element will have the desired value. For the column-major case the index for A_{ij} is found as shown in equation 2.

$$Index\ in\ 1D\ Array\ (in\ memory) = (number\ of\ rows\ in\ A * j) + i \tag{2}$$

No matter what the size of the 2D array is it will always be stored as a 1D array in either row-major format or column major format.

Single Processor Execution

This section shows how the main arithmetic expression progresses as we try out the six different permutations. In figure 1, the code contains 3 loops executing in order **(i, j, k)**. We will also see how the execution changes when the order of these loops is changed e.g. **(i, k, j)**, **(j, i, k)**, **(j, k, i)**, **(k,i, j)**, **(k, j, i)**. Changing the order of loops does not change the final answer (invariant transformation) rather it just changes the way the computation progresses. Furthermore, we will be multiplying two matrices A and B and computing the product C. The assumption will be that A, B and C are stored in row-major format. For simplicity's sake we will be using 2×2 matrices but this idea can be extended to any dimensional matrices. The inner products are computed as follows: $C_{ij} = C_{ij} + (a_{ik} * b_{kj})$.

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} C = \begin{bmatrix} (a_{00} * b_{00} + a_{01} * b_{10}) & (a_{00} * b_{01} + a_{01} * b_{11}) \\ (a_{10} * b_{00} + a_{11} * b_{10}) & (a_{10} * b_{01} + a_{11} * b_{11}) \end{bmatrix}$$

If these were 3×3 matrices than each value in C would've contained one additional product summed to the current values.

(i, j, k)

time	1	2	3	4	5	6	7	8
i, j, k Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{01} * b_{10}$	$a_{00} * b_{01}$	$a_{01} * b_{11}$	$a_{10} * b_{00}$	$a_{11} * b_{10}$	$a_{10} * b_{01}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{00}	C_{01}	C_{01}	C_{10}	C_{10}	C_{11}	C_{11}

Each value C_{ij} is completely calculated before we move to the next value $C_{i(j+1)}$. C_{ij}'s are calculated row wise. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+2) & (3+4) \\ (5+6) & (7+8) \end{bmatrix}$$

(j, i, k)

time	1	2	3	4	5	6	7	8
j, i, k Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{01} * b_{10}$	$a_{10} * b_{00}$	$a_{11} * b_{10}$	$a_{00} * b_{01}$	$a_{01} * b_{11}$	$a_{10} * b_{01}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{00}	C_{10}	C_{10}	C_{01}	C_{01}	C_{11}	C_{11}

Each value C_{ij} is completely calculated before we move to the next value $C_{(i+1)j}$. C_{ij}'s are calculated column wise. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+2) & (5+6) \\ (3+4) & (7+8) \end{bmatrix}$$

(k, i, j)

time	1	2	3	4	5	6	7	8
k, i, j Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{00} * b_{01}$	$a_{10} * b_{00}$	$a_{10} * b_{01}$	$a_{01} * b_{10}$	$a_{01} * b_{11}$	$a_{11} * b_{10}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{01}	C_{10}	C_{11}	C_{00}	C_{01}	C_{10}	C_{11}

Each value C_{ij} is partially calculated. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+5) & (2+6) \\ (3+7) & (4+8) \end{bmatrix}$$

(i, k, j)

time	1	2	3	4	5	6	7	8
i, k, j Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{00} * b_{01}$	$a_{01} * b_{10}$	$a_{01} * b_{11}$	$a_{10} * b_{00}$	$a_{10} * b_{01}$	$a_{11} * b_{10}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{01}	C_{00}	C_{01}	C_{10}	C_{11}	C_{10}	C_{11}

Each value C_{ij} is partially calculated. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+3) & (2+4) \\ (5+7) & (6+8) \end{bmatrix}$$

(j, k, i)

time	1	2	3	4	5	6	7	8
j, k, i Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{10} * b_{00}$	$a_{01} * b_{10}$	$a_{11} * b_{10}$	$a_{00} * b_{01}$	$a_{10} * b_{01}$	$a_{01} * b_{11}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{10}	C_{00}	C_{10}	C_{01}	C_{11}	C_{01}	C_{11}

Each value C_{ij} is partially calculated. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+3) & (5+7) \\ (2+4) & (6+8) \end{bmatrix}$$

(k, j, i)

time	1	2	3	4	5	6	7	8
i, k, j Values	0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1
Value Computed	$a_{00} * b_{00}$	$a_{10} * b_{00}$	$a_{00} * b_{01}$	$a_{10} * b_{01}$	$a_{01} * b_{10}$	$a_{11} * b_{10}$	$a_{01} * b_{11}$	$a_{11} * b_{11}$
Value Computed in C	C_{00}	C_{01}	C_{10}	C_{11}	C_{00}	C_{01}	C_{10}	C_{11}

Each value C_{ij} is partially calculated. The figure below shows the order in which each value of C is computed.

$$C = \begin{bmatrix} (1+5) & (3+7) \\ (2+6) & (4+8) \end{bmatrix}$$

Multiprocessor Execution

In this section we discuss the mapping of the computation a n processor ring. We will perform unfolding and parallelization on the outermost loop of all the permutations of **(i, j, k)** mentioned in the previous section. Below are the algorithms provided for the multiplication also the mapping of the matrices A, B and C is discussed.

(i, j, k)

Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.
Each Processor P_i computes the $i-th$ row of C.

Do All $P_{i=0,n-1}$:
  For $j = i,\ do\ n\ times$:
    For $k = 0, n-1, do$:
      $C_{ij} = C_{ij} + A_{ik} * B_{kj}$
    endfor
  Each Processor i sends its current columnn of B to processor $(i+1)mod(n)$
  endfor
enddolall

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

(j, i, k)

Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.
Each Processor P_j computes the $j-th$ row of C.

Do All $P_{j=0,n-1}$:
  For $i = j,\ do\ n\ times$:
    For $k = 0, n-1, do$:
      $C_{ji} = C_{ji} + A_{jk} * B_{ki}$
    endfor
  Each Processor j sends its current column of B to processor $(j+1)mod(n)$
  endfor
enddolall

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

(k, i, j)

Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.
Each Processor P_k computes the $k-th$ row of C.

Do All $P_{k=0,n-1}$:
  For $i = k, do\ n\ times$:
    For $j = 0, n-1, do$:
      $C_{kj} = C_{kj} + A_{ki} * B_{ij}$
    endfor
  Each Processor k sends its current column of B to processor $(k+1)mod(n)$
  endfor
enddolall

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

(i, k, j)
Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.
Each Processor P_i computes the $i-th$ row of C.

```
Do All P_{i=0,n-1} :
    For k = i, do n times :
        For j = 0, n-1, do :
            C_ij = C_ij + A_ik * B_kj
        endfor
     Each Processor i sends its current
     column of B to (i+1)mod(n)
     endfor
enddolall
```

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

(j, k, i)
Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.
Each Processor P_j computes the $j-th$ row of C.

```
Do All P_{j=0,n-1} :
    For k = j, do n times :
        For i = 0, n-1, do :
            C_jk = C_jk + A_ji * B_ik
        endfor
     Each Processor j sends its current
     column of B to (j+1)mod(n)
     endfor
enddolall
```

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

(k, j, i)
Store each row of A row-major wise in each processor initially.
Store each column of B column-major wise in each processor initially.

```
Do All P_{k=0,n-1} :
    For j = 0, n-1, do :
        For i = 0, do n times :
            C_ki = C_ki + A_kj * B_ji
        endfor
     Each Processor j sends its current
     column of B to (k+1)mod(n)
     endfor
enddolall
```

Time	$\mathcal{O}(n^2)$
Space	$\mathcal{O}(n)$
Communications	$\mathcal{O}(n^2)$

Discussion
The above algorithms work for any $n \times n$ matrix multiplication case given that each processor stores the rows of A (row-major) wise and columns of B (column-major) wise. Also, some kind of intercommunication network between the processors is required. Each row of the result C_i is stored in each processor P_i.

When we consider the case where A is stored column-major wise and B is stored row-major wise then we don't need to send rows or columns of any processor to any other. For this particular mapping each processor calculates a partial value for each value of C. This means that the local memory of each processor has to store all the values of C. Hence, to obtain a final result for C we would have to sum up all the values of C across all the corresponding processors. This technique mapping requires more memory and a slight modification in the algorithm. The modification is that at the end to obtain a value C_{ij} we would have to loop over all the processors and sum all $C'_{ij}s$. This algorithm/mapping however does not seem that practical.