Parallel Algorithms

Parallel Random Access Machine (PRAM) helps to write precursor parallel algorithm without any architecture constraints. It allows parallel-algorithm designers to treat processing power as unlimited. It ignores complexity of inter-process communication. PRAM algorithms have two phases:

Phase 1: Sufficient numbers of processors are activated

Phase 2: Activated processors perform the computations in parallel

For example, binary tree reduction can be implemented using n/2 processors. Here, EREW PRAM suffices for reduction.

Parallel algorithm design strategies

The parallel programming problems may have more than one solution. The techniques for designing parallel algorithm may be given as under:

1. Divide and conquer
2. Greedy Method
3. Dynamic Programming
4. Backtracking
5. Branch & Bound
6. Linear Programming

Prefix Sum

Prefix sum is a process in which multiple elements stored in an array in such a way that each position of the array contains the summation of its previous element with its own value.

$$B[i] = \sum_{j=0}^{i} A[j]$$

The CREW PRAM algorithm is used for prefix sum calculations. It can use n/2 processors. It takes $O(\log_2 n)$ time.

For example:

A=	**3**	**5**	**4**	**1**	**9**	**6**	**8**	**7**

Here an array A is taken with eight elements if it is required to compute the prefix sum of array 'A' then the new array supposes 'B' will have the index positions as mentioned below:

B[0]=A[0]

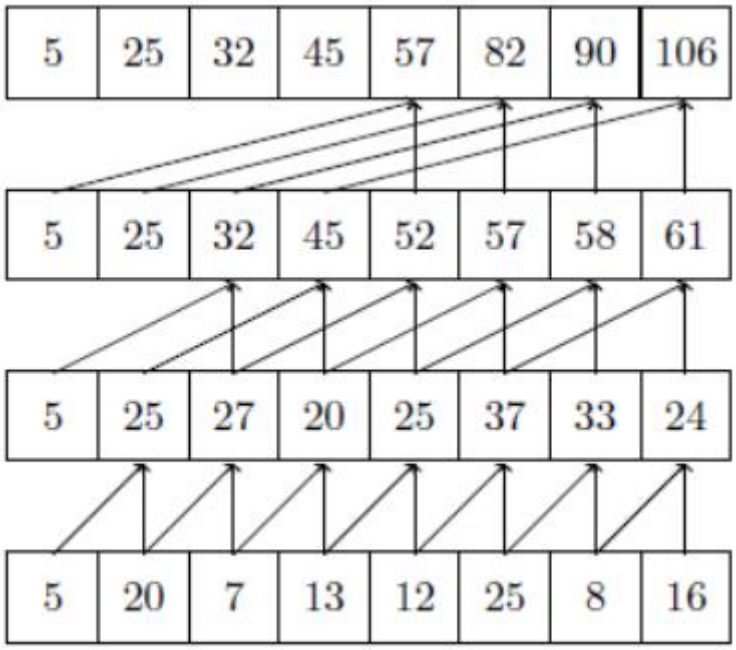
B[1]= A[0]+A[1]

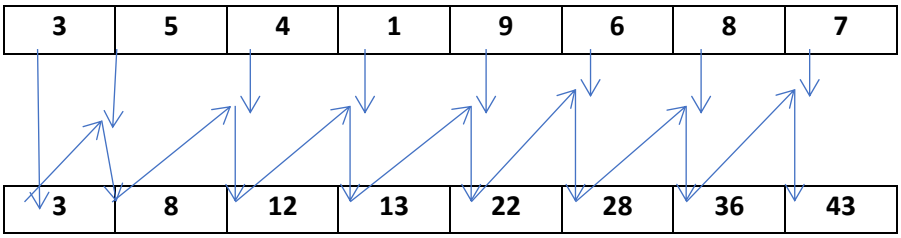
B[2]= B[1]+A[2]

Similarly, we can write

B[n] = B[n-1] + A[n]

Using mentioned above expression we can compute the prefix sum of any array, so the resultant array of ‘A’ will be given as:





It is clear from the above diagram that the last element of the computed array ‘B’ holds the sum of all elements of array ‘A’. If we need to compute this problem using sequential algorithm then it will take N - 1 steps to completely execute the algorithm. For larger problem size it will be very time taking process to compute the prefix sum using sequential approach so we can apply the parallel approach to compute the prefix sum for large input size of problem. This approach can be implemented using CREW model of parallel computing. It should be taken in proper consideration that the parallel computing algorithm must be cost optimal and faster than the sequential approach.

CREW prefix sum algorithm:

```
PREFIX.SUMS (CREW PRAM):
Initial condition: List of n ≥ 1 elements stored in A[0···(n − 1)]
Final condition: Each element A[i] contains A[0] ⊕ A[1] ⊕ ··· ⊕ A[i]
Global variables: n, A[0...(n − 1)], j
begin
  spawn (P1, P2, ..., Pn−1)
  for all Pi where 1 ≤ i ≤ n − 1 do
    for j ← 0 to ⌈log n⌉ − 1 do
      if i − 2^j ≥ 0 then
          A[i] ← A[i] + A[i − 2^j]
      endif
    endfor
  endfor
end
```

LIST-PREFIX(*L*)

FOREACH processor *i*, in parallel

$y[i] \leftarrow x[i]$

WHILE there exists an object *i* such that $next[i] \neq NIL$

FOREACH processor *i,* in parallel

IF $next[i] \neq NIL$

THEN:

$y[next[i]] \leftarrow y[i] \otimes y[next[i]]$

$next[i] \leftarrow next[next[i]]$

As it has been discussed previously that CREW approach works on concurrent read and exclusive write manner which means either multiple processors can read the specific memory location at the same time or only one processor can perform WRITE operation. Initial condition describes the problem over which computation is to be applied. Final condition describes about the output produced by the algorithm.

Example: Parallel Prefix Sum

A[1]	**A[2]**	**A[3]**	**A[4]**
3	**7**	**5**	**2**

Given:

n=4

log n => log 4 => 2

So, for d=0 to log n-1 do

0,1 (2 - 1 = 1) times

Each processor will run of 0, 1 iterations

STEP 1

for i = 1

for j = 0

if $1 - 2^0 \geq 0$

$A[1] \leftarrow A[1] + A[1 - 2^0]$

$A[1] \leftarrow A[1] + A[0]$

here A[0] does not exist

for i = 1

for j = 1

if $1 - 2^1 \geq 0$ returns false

for i = 1

for j = 2

$1 - 2^2 \geq 0$ returns false

Hence, we will get the following result:

A[1]	**A[2]**	**A[3]**	**A[4]**
3			

STEP 2

for i=2

for j=0

if $2 - 2^0 \geq 0$

$A[2] \leftarrow A[2] + [2 - 2^0]$

$A[2] \leftarrow A[2] + A[1]$

A[1]	**A[2]**	**A[3]**	**A[4]**
3	**3+7 = 10**		

STEP 3

i=3

j=0

$3- 2^0 = 2$

$A[3] \leftarrow A[3] + A[2]$

A[1]	**A[2]**	**A[3]**	**A[4]**
3	**10**	**5 + 7**	

i=3

j=1

$3-2^1=1$

$A[3] \leftarrow A[3]+A[1]$

A[1]	**A[2]**	**A[3]**	**A[4]**
3	**10**	**5 + 7 + 3 = 15**	

Since this is CREW algorithm, hence the value of A[2] = 7. Also, every processor reads the array at initial condition.

STEP 4

i=4

j=0

$4- 2^0 = 3$

$A[4] \leftarrow A[4] + A[3]$

A[1]	A[2]	A[3]	A[4]
3	**10**	**15**	**2 + 5**

i=4

j=1

A[4] ← A[4]+A[2]

A[1]	A[2]	A[3]	A[4]
3	**10**	**15**	**2 + 5 +10 = 17**

In next iteration the value of A[2] is updated from 5 and the final result will come out to be:

A[1]	A[2]	A[3]	A[4]
3	**10**	**15**	**17**

Merge Sorted List

Many PRAM algorithms achieve low time complexity by performing more operations than an optimal RAM algorithm. For example, a RAM algorithm requires at most n-1 comparisons to merge two sorted lists of n/2 elements. The time complexity of merging two sorted list is O(n).

CREW PRAM algorithm is used for assigning each list element its own processor for n processors. The processor knows the index of the element in its own list. It finds the index in the other list using binary search. It adds the two indices to obtain the final position. The total number of operations increased to $O(n\log_2 n)$.

Example: Merging Two Sorted List

Consider two sorted arrays as

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
2	**6**	**8**	**14**	**18**	**20**	**24**

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
3	**5**	**9**	**12**	**13**	**22**	**25**

As mentioned in the algorithm, there is participation of all the processors because of the use of "spawn" function. We can illustrate this algorithm by choosing any processor value as P_3

Working of P_3

A[i = 3] is greater than [i – 1] => [3 – 1] => 2, i.e., 2nd position element in the lower array (A [9] is 2nd index position of lower array : 8 > 5

Now evaluate (high – n/2) because 8 is larger than two elements of lower array and larger than (high – n/2) => 9 – 7 => 2, i.e. 2 elements in the upper array.

Perform the binary search with A[3] in upper array and find the position as: high = index of the largest integer value which is less than 8 => high is 9

No, P3 can calculate the position of 8 in the merged array after computing: (i-1) + (high – n/2). So, the position is (i + high – n/2) => (3 + 9 – 7 = 5), i.e., 5th position.

A[1]	A[2]	A[3]	A[4]	A[5]	…	A[14]
				8		

Similarly, the process will run for all the position using individual processing element.

Matrix multiplication

Matrix multiplication is a binary operation of two matrices of order M1xN1 and M2xN2. The operation thus produces a matrix of order M1xN2 if and only if N1=M2. The result matrix of order M1xN2 is known as the matrix product. The parallel algorithm follows:

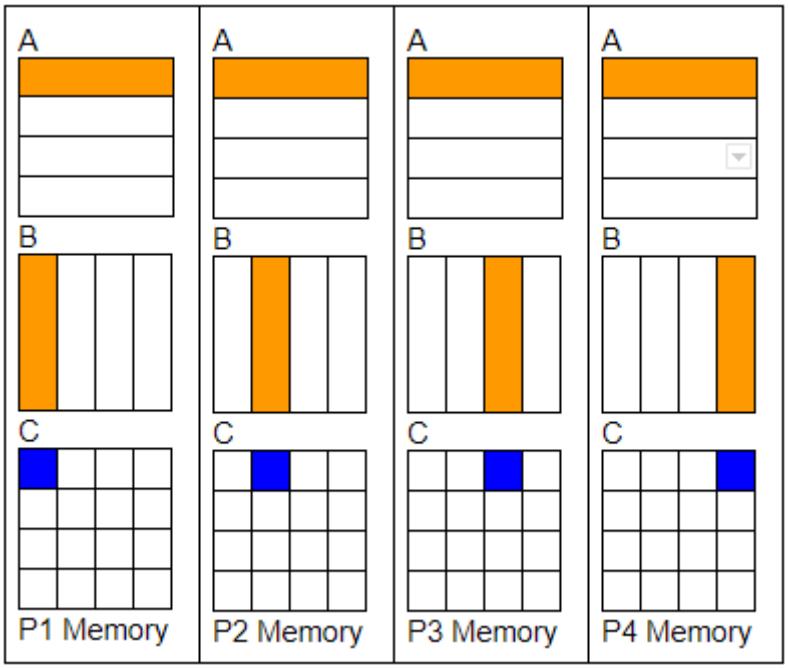
$$\begin{bmatrix} -2 & 1 \\ 0 & 4 \end{bmatrix} \times \begin{bmatrix} 6 & 5 \\ -7 & 1 \end{bmatrix} = \begin{bmatrix} -2\times6+1\times-7 & -2\times5+1\times1 \\ 0\times6+4\times-7 & 0\times5+4\times1 \end{bmatrix}$$

$$= \begin{bmatrix} -19 & -9 \\ -28 & 4 \end{bmatrix}$$

Matrix Multiplication for Big Data

The matrix multiplication becomes complex if you are dealing with big data. In case of Big Data, loading the complete matrix into the memory is not possible because of limited memory. In this case, we load just one row at a time from matrix A to all nodes (processors). The big data may have multiple dimensions. The matrix multiplication process has to be repeated for each dimension. The figure below depicts the big data two-dimensional multiplication. The algorithm for the same is given:

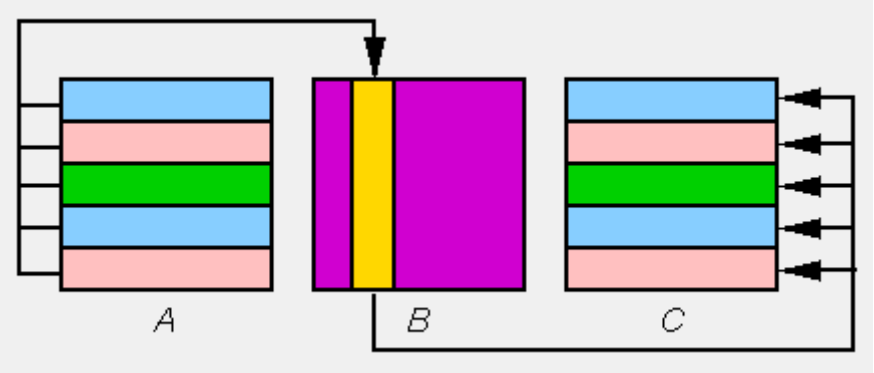
```
PARALLEL_MATRIX_MULTIPLICATION (A, B, C, M1, N1, M2, N2, DIM)

BEGIN
   for each dimension in DIM:
      for each row in A:
copy row to nodes
for each col in B:
   copy col to nodes
   Initialize
Crow col← 0
   for s = 1 to N1:
      Crow col ← Crow col + rows* cols
   end for
end for
      end for
   end for
END
```



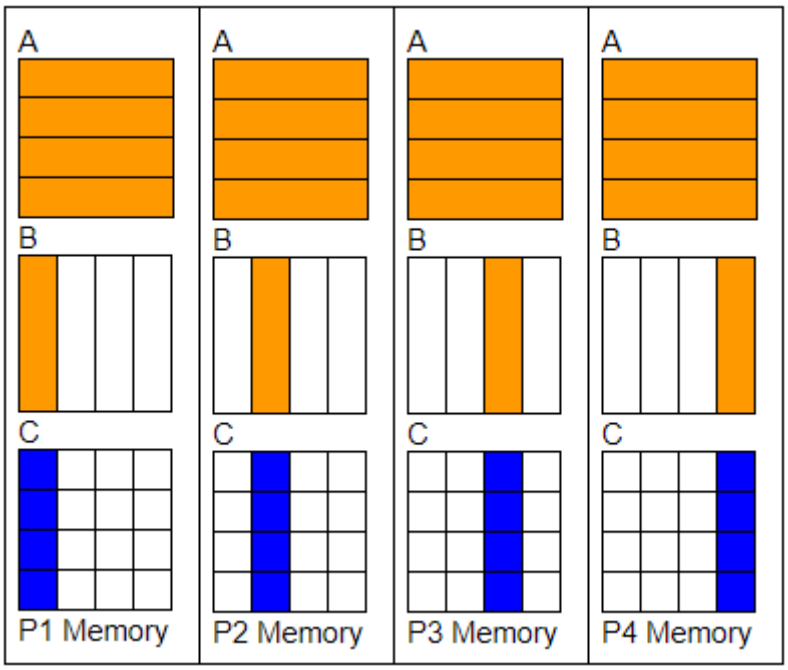
The row of matrix A is copied to all PEs along with different columns of matrix B which then calculates the one cell of C. On combining the result from all PEs we get a complete row of resulting matrix C. In this manner, we calculate for all rows. Also because we are calculating for n dimension matrix, the similar processing has to take place for each dimension repetitively. Suppose we have k PEs, then the number of PEs to be employed, k_e, will be calculated as:

$k_e = N_1 \ mod \ k$ where k>0. Broadcasting of each B[j] takes O(N logN) time.

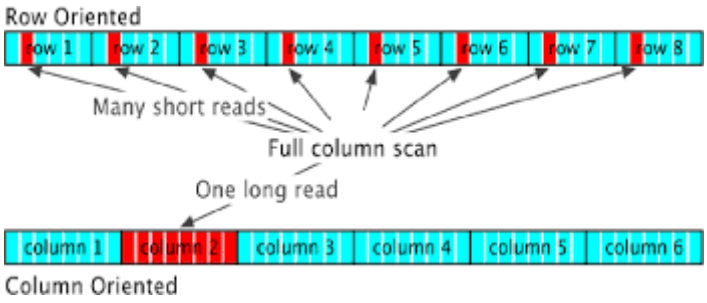
Row/ Column oriented algorithm




```
PARALLEL_MATRIX_MULTIPLICATION (A, B, C, M1, N1, M2, N2)
Do if N1=M2
Load A[i] on every processor P[i]
For all P[i] do:
   For j=0 to N-1:
      Load B[j]
      C[i][j] = A[i] * B[j]
Collect C[i]
```



Input Data | Output Data



Block oriented Matrix Multiplication

Block matrix multiplication is analogous to scalar matrix multiplication. In this approach sub-matrices are multiplied and shifted.

A=

A_{11}	A_{12}	...	A_{1n}
A_{21}	A_{22}		
...		...	
A_{m1}			A_{mn}

B=

B_{11}	B_{12}	...	B_{1n}
B_{21}	B_{22}		
...		...	
B_{m1}			B_{mn}

C=

A_{11} B_{11} + A_{12} B_{21}	A_{11} B_{12} + A_{12} B_{22}	...	B_{1n}
A_{21} B_{11} + A_{22} B_{21}	A_{21} B_{12} + A_{22} B_{22}		
...		...	
B_{m1}			B_{mn}

Analysis of Algorithm

n^2 matrix, p processors.

- Row-Column-oriented algorithm
    1. Computation: $n^2/p * n/p = n^3/p^2$.
    2. Communication: $2(\lambda + \beta n^2/p)$
    3. p iterations.
- Block-oriented algorithm
    1. Computation: $n^2/p * n/p = n^3/p^2$.
    2. Communication: $4(\lambda + \beta n^2/p)$
    3. $\sqrt{p}-1$ iterations.
- Comparison
  $2p (\lambda + \beta n^2/p) > 4 (\sqrt{p}-1) (\lambda + \beta n^2/p)$
  $\lambda p + \beta n^2 > 4 \lambda (\sqrt{p}-1) + 4 \beta (\sqrt{p}-1) n^2/p$
    1. $p > 4 (\sqrt{p}-1) \rightarrow p > 4$
    2. $1 > 4 (\sqrt{p}-1)/p \rightarrow p > 4$