

## 410250 : High Performance Computing

Any 4 Assignments and 1 Mini Project are Mandatory

### Group 1

1.	Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .
2.	Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.
3.	Implement Min, Max, Sum and Average operations using Parallel Reduction.
4.	Write a CUDA Program for : 1. Addition of two large vectors 2. Matrix Multiplication using CUDA C
5.	Implement HPC application for AI/ML domain.

### Group 2

6.	Mini Project: Evaluate performance enhancement of parallel Quicksort Algorithm using MPI
7.	Mini Project: Implement Huffman Encoding on GPU
8.	Mini Project: Implement Parallelization of Database Query optimization
9.	Mini Project: Implement Non-Serial Polyadic Dynamic Programming with GPU Parallelization

# Group 1

## Assignment 1 :

Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

# Parallel Breadth First Search

1. To design and implement parallel breadth first search, you will need to divide the graph into smaller sub-graphs and assign each sub-graph to a different processor or thread.
2. Each processor or thread will then perform a breadth first search on its assigned sub-graph concurrently with the other processors or threads.
3. Two methods : Vertex by Vertex OR Level By Level

**Procedure Parallel-Breadth-First-Search-Vertex(ALM, EM, U)**

begin

mark every vertex "unvisited"

 $v \leftarrow$  start vertexmark  $v$  "visited"instruct processor( $i$ ) where  $1 \leq i \leq k$ for  $j = 1$  to  $k$  doif  $(k * (j - 1) + i) \leq EM(v)$ then delete  $v$  from  $U(ALM(v, k * (j - 1) + i))$ 

endif

endfor

end-instruction

initialize queue with  $v$ 

while queue is not empty do

begin

 $v \leftarrow$  first vertex from the queuefor each  $w \in U(v)$  do

begin

mark  $w$  "visited"instruct processor ( $i$ ) where  $1 \leq i \leq k$ for  $j = 1$  to  $k$  doif  $(k * (j - 1) + i) \leq EM(w)$ then delete  $w$  from  $U(ALM(w, k * (j - 1) + i))$ 

endif

endfor

end-instruction

add  $w$  to queue

end

endfor

endwhile

end

**End Parallel-Breadth-First-Search-Vertex**

# Breadth First Search

To design and implement parallel breadth first search using OpenMP, you can use the existing breadth first search algorithm and parallelize it using OpenMP's parallelization constructs.

## Program Sample :

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>
```

```
using namespace std;
```

```
void bfs(vector<vector<int>>& graph, int start, vector<bool>&
visited) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    #pragma omp parallel
    {
        #pragma omp single
        {
            while (!q.empty()) {
                int vertex = q.front();
                q.pop();
```

```
#pragma omp task firstprivate(vertex)
{
    for (int neighbor : graph[vertex]) {
        if (!visited[neighbor]) {
            q.push(neighbor);
            visited[neighbor] = true;
            #pragma omp task
            bfs(graph, neighbor, visited);
        }
    }
}
}
}
}
}
```

```
void parallel_bfs(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    bfs(graph, start, visited);
}
```

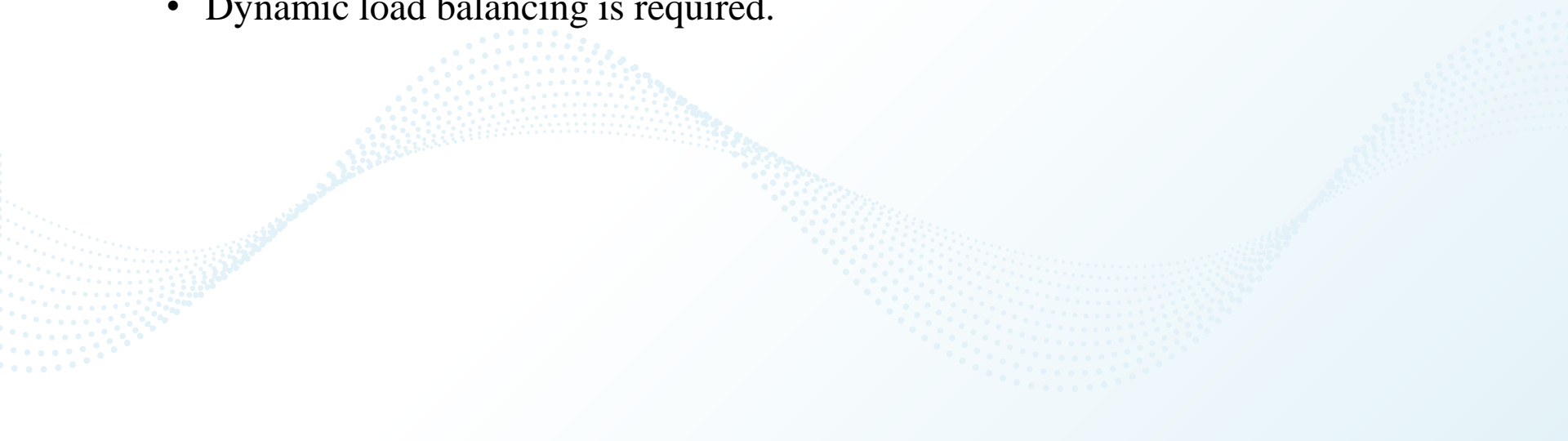
1. In this implementation, the **parallel\_bfs** function takes in a graph represented as an adjacency list, where each element in the list is a vector of neighboring vertices, and a starting vertex.
2. The **bfs** function uses a queue to keep track of the vertices to visit, and a boolean visited array to keep track of which vertices have been visited. The **#pragma omp parallel** directive creates a parallel region and the **#pragma omp single** directive creates a single execution context within that region.
3. Inside the while loop, the **#pragma omp task** directive creates a new task for each unvisited neighbor of the current vertex.
4. This allows each task to be executed in parallel with other tasks. The **firstprivate** clause is used to ensure that each task has its own copy of the **vertex** variable.
5. This is just one possible implementation, and there are many ways to improve it depending on the specific requirements of your application. For example, you can use **omp atomic** or **omp critical** to protect the shared resource queue.
6. This is just one possible implementation, and there are many ways to improve it depending on the specific requirements of your application. For example, you can use **omp atomic** or **omp critical** to protect the shared resource queue.



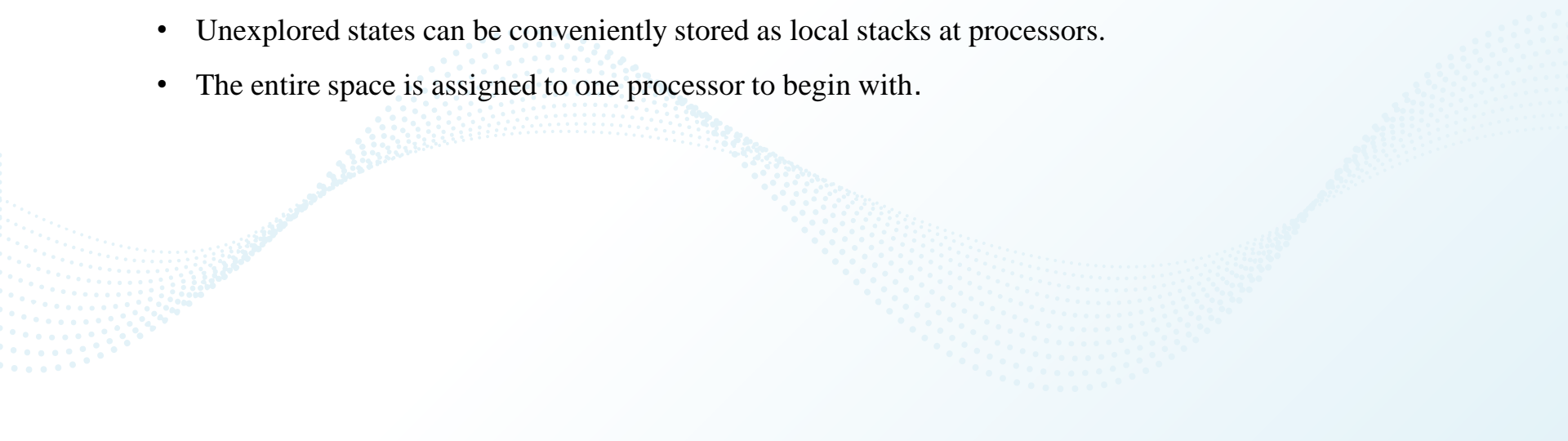
# Parallel Depth First Search



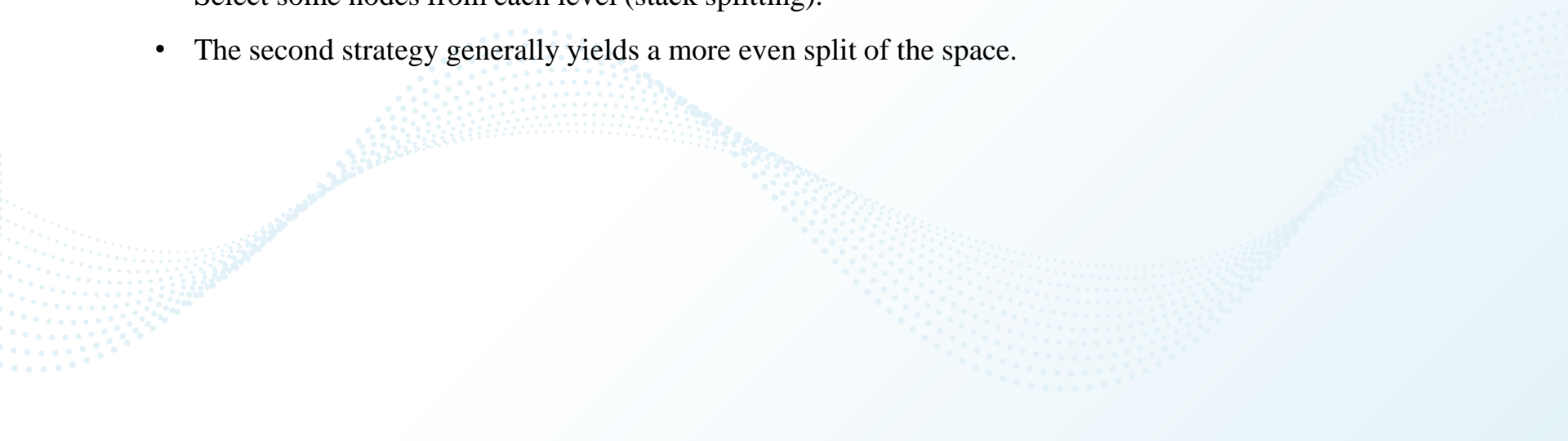
## Parallel Depth-First Search

- Different subtrees can be searched concurrently.
  - Subtrees can be very different in size.
  - Estimate the size of a subtree rooted at a node.
  - Dynamic load balancing is required.
- 

# Parallel Depth-First Search: Dynamic Load Balancing

- When a processor runs out of work, it gets more work from another processor.
  - This is done using work requests and responses in message passing machines and locking and extracting work in shared address space machines.
  - On reaching final state at a processor, all processors terminate.
  - Unexplored states can be conveniently stored as local stacks at processors.
  - The entire space is assigned to one processor to begin with.
- 
- A decorative graphic consisting of several overlapping, wavy lines of light blue dots, creating a sense of motion and depth, located in the lower half of the slide.

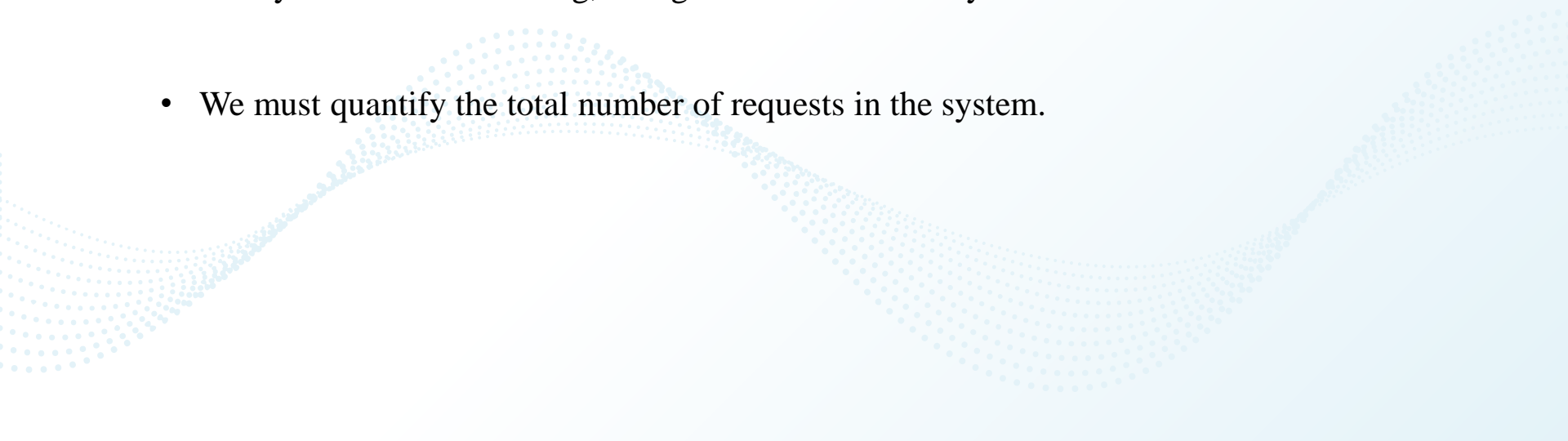
## Parameters in Parallel DFS: Work Splitting

- Work is split by splitting the stack into two.
  - Ideally, we do not want either of the split pieces to be small.
  - Select nodes near the bottom of the stack (node splitting), or
  - Select some nodes from each level (stack splitting).
  - The second strategy generally yields a more even split of the space.
- 


## Load-Balancing Schemes

- Asynchronous round robin: Each processor maintains a counter and makes requests in a round-robin fashion.
- Global round robin: The system maintains a global counter and requests are made in a round-robin fashion, globally.
- Random polling: Request a randomly selected processor for work.

## Analyzing DFS

- We can't compute, analytically, the serial work  $W$  or parallel time. Instead, we quantify total overhead  $T_o$  in terms of  $W$  to compute scalability.
  - For dynamic load balancing, idling time is subsumed by communication.
  - We must quantify the total number of requests in the system.
- 

## Load-Balancing Schemes

- Asynchronous round robin has poor performance because it makes a large number of work requests.
  - Global round robin has poor performance because of contention at counter, although it makes the least number of requests.
  - Random polling strikes a desirable compromise.
- 

## Termination Detection

- Processor  $P_0$  has all the work and a weight of one is associated with it. When its work is partitioned and sent to another processor, processor  $P_0$  retains half of the weight and gives half of it to the processor receiving the work.
- If  $P_i$  is the recipient processor and  $w_i$  is the weight at processor  $P_i$ , then after the first work transfer, both  $w_0$  and  $w_i$  are 0.5.
- Each time the work at a processor is partitioned, the weight is halved. When a processor completes its computation, it returns its weight to the processor from which it received work.
- Termination is signaled when the weight  $w_0$  at processor  $P_0$  becomes one and processor  $P_0$  has finished its work



# Program

```
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>
```

```
using namespace std;
```

```
void dfs(vector<vector<int>>& graph, int start,
vector<bool>& visited) {
    stack<int> s;
    s.push(start);
    visited[start] = true;
    #pragma omp parallel
    {
        #pragma omp single
        {
            while (!s.empty()) {
                int vertex = s.top();
                s.pop();
```

```
#pragma omp task firstprivate(vertex)
{
    for (int neighbor : graph[vertex]) {
        if (!visited[neighbor]) {
            s.push(neighbor);
            visited[neighbor] = true;
            #pragma omp task
            dfs(graph, neighbor, visited);
        }
    }
}

}

}

}

void parallel_dfs(vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false);
    dfs(graph, start, visited);
}
```

1. In this implementation, the **parallel\_dfs** function takes in a graph represented as an adjacency list, where each element in the list is a vector of neighboring vertices, and a starting vertex.
2. The **dfs** function uses a stack to keep track of the vertices to visit, and a boolean visited array to keep track of which vertices have been visited.
3. The **#pragma omp parallel** directive creates a parallel region and the **#pragma omp single** directive creates a single execution context within that region.
4. Inside the while loop, the **#pragma omp task** directive creates a new task for each unvisited neighbor of the current vertex.
5. This allows each task to be executed in parallel with other tasks. The **firstprivate** clause is used to ensure that each task has its own copy of the **vertex** variable.
6. This implementation is suitable for both tree and undirected graph, since both are represented as an adjacency list and the algorithm is using a stack to traverse the graph.
7. This is just one possible implementation, and there are many ways to improve it depending on the specific requirements of your application. For example, you can use **omp atomic** or **omp critical** to protect the shared resource stack.
8. The **dfs** function uses a stack to keep track of the vertices to visit, and a boolean visited array to keep track of which vertices have been visited. The **#pragma omp parallel** directive creates a parallel region and the **#pragma omp single** directive creates a single execution context within that region.

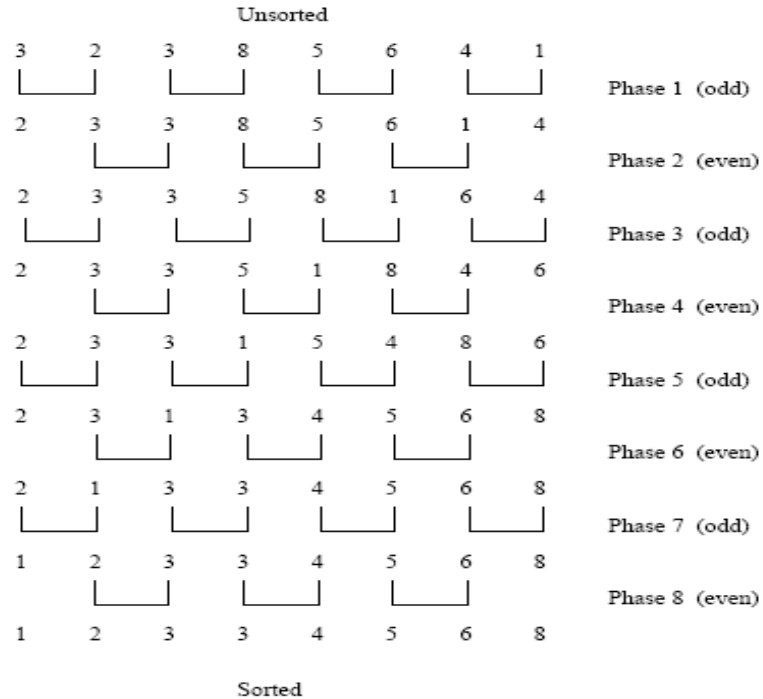
## Assignment 2 :

Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

# Bubble sort

1. The complexity of bubble sort is  $\Theta(n^2)$ .
2. Bubble sort is difficult to parallelize since the algorithm has no concurrency.
3. A simple variant, though, uncovers the concurrency.

# Odd-Even Transposition



Sorting  $n = 8$  elements, using the odd-even transposition sort algorithm. During each phase,  $n = 8$  elements are compared.

## Odd-Even Transposition

```
1.  procedure ODD-EVEN( $n$ )
2.  begin
3.      for  $i := 1$  to  $n$  do
4.          begin
5.              if  $i$  is odd then
6.                  for  $j := 0$  to  $n/2 - 1$  do
7.                      compare-exchange( $a_{2j+1}, a_{2j+2}$ );
8.              if  $i$  is even then
9.                  for  $j := 1$  to  $n/2 - 1$  do
10.                     compare-exchange( $a_{2j}, a_{2j+1}$ );
11.          end for
12.  end ODD-EVEN
```

Sequential odd-even transposition sort algorithm.

# Odd-Even Transposition

1. After  $n$  phases of odd-even exchanges, the sequence is sorted.
2. Each phase of the algorithm (either odd or even) requires  $\Theta(n)$  comparisons.
3. Serial complexity is  $\Theta(n^2)$ .



# Parallel Odd-Even Transposition

1. Consider the one item per processor case.
2. There are  $n$  iterations, in each iteration, each processor does one compare-exchange.
3. The parallel run time of this formulation is  $\Theta(n)$ .
4. This is cost optimal with respect to the base serial algorithm but not the optimal one.

# Parallel Odd-Even Transposition

```
1.  procedure ODD-EVEN_PAR( $n$ )
2.  begin
3.     $id :=$  process's label
4.    for  $i := 1$  to  $n$  do
5.      begin
6.        if  $i$  is odd then
7.          if  $id$  is odd then
8.             $compare\_exchange\_min(id + 1);$ 
9.          else
10.            $compare\_exchange\_max(id - 1);$ 
11.        if  $i$  is even then
12.          if  $id$  is even then
13.             $compare\_exchange\_min(id + 1);$ 
14.          else
15.             $compare\_exchange\_max(id - 1);$ 
16.        end for
17.      end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

# Parallel Odd-Even Transposition

1. Consider a block of  $n/p$  elements per processor.
2. The first step is a local sort.
3. In each subsequent step, the compare exchange operation is replaced by the compare split operation.
4. The parallel run time of the formulation is

$$T_P = \overbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

# Bubble Sort Odd Even Transposition

```
#include <iostream>
#include <vector>
#include <omp.h>
```

```
using namespace std;
```

```
void bubble_sort_odd_even(vector<int>& arr) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        #pragma omp parallel for
        for (int i = 0; i < arr.size() - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
        #pragma omp parallel for
        for (int i = 1; i < arr.size() - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}
```

```
int main() {  
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};  
    double start, end;  
  
    // Measure performance of parallel bubble sort using odd-  
even transposition  
    start = omp_get_wtime();  
    bubble_sort_odd_even(arr);  
    end = omp_get_wtime();  
    cout << "Parallel bubble sort using odd-even transposition  
time: " << end - start << endl;  
}
```

1. This program uses OpenMP to parallelize the bubble sort algorithm.
2. The **#pragma omp parallel for** directive tells the compiler to create a team of threads to execute the for loop within the block in parallel.
3. Each thread will work on a different iteration of the loop, in this case on comparing and swapping the elements of the array.
4. The **bubbleSort** function takes in an array, and it sorts it using the bubble sort algorithm. The outer loop iterates from 0 to  $n-2$  and the inner loop iterates from 0 to  $n-i-1$ , where  $i$  is the index of the outer loop. The inner loop compares the current element with the next element, and if the current element is greater than the next element, they are swapped.
5. The **main** function creates a sample array and calls the **bubbleSort** function to sort it. The sorted array is then printed.
6. This is a skeleton code and it may not run as is and may need some modification to work with specific inputs and requirements.
7. It is worth noting that bubble sort is not an efficient sorting algorithm, specially for large inputs, and it may not scale well with more number of threads. Also parallelizing bubble sort does not have a significant improvement in performance due to the nature of the algorithm itself.
8. In this implementation, the **bubble\_sort\_odd\_even** function takes in an array and sorts it using the odd-even transposition algorithm. The outer while loop continues until the array is sorted. Inside the loop, the **#pragma omp parallel for** directive creates a parallel region and divides the loop iterations among the available threads. Each thread performs the swap operation in parallel, improving the performance of the algorithm.
9. The two **#pragma omp parallel for** inside while loop, one for even indexes and one for odd indexes, allows each thread to sort the even and odd indexed elements simultaneously and prevent the dependency.

# Merge Sort



## Parallel Merge Sort

Given a set of elements  $A = \{a_1, a_2, \dots, a_n\}$ ,

$A_{\text{odd}}$  and  $A_{\text{even}}$  are defined as the set of elements of  $A$  with odd and even indices, respectively.

For example,  $A_{\text{odd}} = \{a_1, a_3, a_5, \dots\}$  and  $A_{\text{even}} = \{a_2, a_4, a_6, \dots\}$  regarding a set of elements  $A = \{a_1, \dots, a_n\}$

Similarly, let a set of elements  $B = \{b_1, \dots, b_n\}$ . We can then define the merge operation as:

$$\text{Merge}(A, B) = \{a_1, b_1, a_2, b_2, a_3, b_3, \dots, a_n, b_n\}$$

For example,

if  $A = \{1, 2, 3, 4\}$  and  $B = \{5, 6, 7, 8\}$

then  $\text{Merge}(\{1, 2, 3, 4\}, \{5, 6, 7, 8\}) = \{1, 5, 2, 6, 3, 7, 4, 8\}$

$$\text{Join}(A, B) = (\text{Merge}(A, B), \text{Odd-Even}(A, B))$$

**Algorithm: Odd-Even(A,B,S)**

begin

if A and B are of length 1

then

Merge A and B using one Compare-and-Exchange operation

else

begin

compute  $S_{\text{odd}}$  and  $S_{\text{even}}$  In Parallel do

$S_{\text{odd}} = \text{Merge}(A_{\text{odd}}, B_{\text{odd}})$

$S_{\text{even}} = \text{Merge}(A_{\text{even}}, B_{\text{even}})$

$S_{\text{odd-even}} = \text{Join}(S_{\text{odd}}, S_{\text{even}})$

end

endif

end

Example: Suppose the set of elements  $S = \{2,3,6,10,15,4,5,8\}$  and we start with  $A = \{2,6,10,15\}$  and  $B = \{3,4,5,8\}$ , two sorted sets of elements. Then

$$\text{Merge}(A_{\text{odd}}, B_{\text{odd}}) = \{2,3,5,10\}$$

$$\text{Merge}(A_{\text{even}}, B_{\text{even}}) = \{4,6,8,15\}$$

The join operation:

$$\text{Join}(A,B) = \{\text{Merge}(A,B), \text{Odd-Even}(A,B)\}$$

requires a merge operation, which results in  $\text{Merge}(A,B) = \{2,4,3,6,5,8,10,15\}$ , and an odd-even operation, which obtains the final sorted list of elements,

$$\text{Odd-Even}\{2,4,3,6,5,8,10,15\} = \{2,3,4,5,6,8,10,15\}$$

```
#include <iostream>
#include <vector>
#include <omp.h>
```

```
using namespace std;
```

```
void merge(vector<int>& arr, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);

    for (i = 0; i < n1; i++) {
        L[i] = arr[l + i];
    }
    for (j = 0; j < n2; j++) {
        R[j] = arr[m + 1 + j];
    }
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }
}
```

```
void merge_sort(vector<int>& arr, int l, int r) {  
    if (l < r) {  
        int m = l + (r - l) / 2;  
        #pragma omp task  
        merge_sort(arr, l, m);  
        #pragma omp task  
        merge_sort(arr, m + 1, r);  
        merge(arr, l, m, r);  
    }  
}
```

```
void parallel_merge_sort(vector<int>& arr) {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        merge_sort(arr, 0, arr.size() - 1);  
    }  
}
```

```
int main() {  
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};  
    double start, end;  
  
    // Measure performance of sequential merge sort  
    start = omp_get_wtime();  
    merge_sort(arr, 0, arr.size() - 1);  
    end = omp_get_wtime();  
    cout << "Sequential merge sort time: " << end - start <<  
endl;  
  
    // Measure performance of parallel merge sort  
    arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};  
    start = omp_get_wtime();  
    parallel_merge_sort(arr);  
    end = omp_get
```

# Assignment 3 : Implement Min, Max, Sum and Average operations using Parallel Reduction

```
#include <iostream>
#include <vector>
#include <omp.h>

using namespace std;
void min_reduction(vector<int>& arr) {
    int min_value = INT_MAX;
    #pragma omp parallel for reduction(min: min_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    cout << "Minimum value: " << min_value << endl;
}
void max_reduction(vector<int>& arr) {
    int max_value = INT_MIN;
    #pragma omp parallel for reduction(max: max_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
    cout << "Maximum value: " << max_value << endl;
}
```

```
void sum_reduction(vector<int>& arr) {
```



```
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
}
cout << "Sum: " << sum << endl;
}

void average_reduction(vector<int>& arr) {
    int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < arr.size(); i++) {
    sum += arr[i];
}
cout << "Average: " << (double)sum / arr.size() << endl;
}

int main() {
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

    min_reduction(arr);
    max_reduction(arr);
    sum_reduction(arr);
    average_reduction(arr);
}
```

1. The **min\_reduction** function finds the minimum value in the input array using the **#pragma omp parallel for reduction(min: min\_value)** directive, which creates a parallel region and divides the loop iterations among the available threads. Each thread performs the comparison operation in parallel and updates the **min\_value** variable if a smaller value is found.
1. Similarly, the **max\_reduction** function finds the maximum value in the array, **sum\_reduction** function finds the sum of the elements of array and **average\_reduction** function finds the average of the elements of array by dividing the sum by the size of the array.
1. The **reduction** clause is used to combine the results of multiple threads into a single value, which is then returned by the function. The **min** and **max** operators are used for the **min\_reduction** and **max\_reduction** functions, respectively, and the **+** operator is used for the **sum\_reduction** and **average\_reduction** functions. In the main function, it creates a vector and calls the functions **min\_reduction**, **max\_reduction**, **sum\_reduction**, and **average\_reduction** to compute the values of min, max, sum and average respectively.

# Assignment 4 : Write a CUDA Program for :

1. Addition of two large vectors
2. Matrix Multiplication using CUDA C

# Vector Addition



```
#include <iostream>
#include <cuda_runtime.h>

__global__ void addVectors(int* A, int* B, int* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int n = 1000000;
    int* A, * B, * C;
    int size = n * sizeof(int);

    // Allocate memory on the host
    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);

    // Initialize the vectors
    for (int i = 0; i < n; i++) {
        A[i] = i;
        B[i] = i * 2;
    }
}
```

// Allocate memory on the device

```
int* dev_A, * dev_B, * dev_C;  
cudaMalloc(&dev_A, size);  
cudaMalloc(&dev_B, size);  
cudaMalloc(&dev_C, size);
```

// Copy data from host to device

```
cudaMemcpy(dev_A, A, size,  
cudaMemcpyHostToDevice);  
cudaMemcpy(dev_B, B, size,  
cudaMemcpyHostToDevice);
```

// Launch the kernel

```
int blockSize = 256;  
int numBlocks = (n + blockSize - 1) / blockSize;
```

```
// Copy data from device to host  
cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);
```

```
// Print the results  
for (int i = 0; i < 10; i++) {  
    cout << C[i] << " ";  
}  
cout << endl;
```

```
// Free memory  
cudaFree(dev_A);  
cudaFree(dev_B);  
cudaFree(dev_C);  
cudaFreeHost(A);  
cudaFreeHost(B);  
cudaFreeHost(C);
```

```
return 0;
```

1. In this program, the `addVectors` kernel takes in the two input vectors `A` and `B`, the output vector `C`, and the size of the vectors `n`. The kernel uses the `blockIdx.x` and `threadIdx.x` variables to calculate the index `i` of the current thread. If the index is less than `n`, the kernel performs the addition operation `C[i] = A[i] + B[i]`.
1. In the `main` function, the program first allocates memory for the input and output vectors on the host and initializes them. Then it allocates memory for the vectors on the device and copies the data from the host to the device using `cudaMemcpy`.



# Matrix multiplication



# Matrix-Matrix Multiplication

- Consider two  $n \times n$  matrices  $A$  and  $B$  partitioned into  $p$  blocks  $A_{i,j}$  and  $B_{i,j}$  (0  
 $\leq i, j < \sqrt{p}$ ) of size each.  $(n/\sqrt{p}) \times (n/\sqrt{p})$
- Process  $P_{i,j}$  initially stores  $A_{i,j}$  and  $B_{i,j}$  and computes block  $C_{i,j}$  of the result matrix.
- Computing submatrix  $C_{i,j}$  requires all submatrices  $A_{i,k}$  and  $B_{k,j}$  for  $0 \leq k < \sqrt{p}$ .
- All-to-all broadcast blocks of  $A$  along rows and  $B$  along columns.
- Perform local submatrix multiplication.

# Example

A =

2	1	5	3
0	7	1	6
9	2	4	4
3	6	7	2

B =

6	1	2	3
4	5	6	5
1	9	8	-8
4	0	-8	5

# Example

A =

2	1	5	3
0	7	1	6
9	2	4	4
3	6	7	2

B =

6	1	2	3
4	5	6	5
1	9	8	-8
4	0	-8	5

# Metrics divided into 4 squares

2	1
0	7

5	3
1	6

9	5
2	3

4	4
7	2

6	1
4	5

2	3
6	5

1	9
4	0

8	-8
-8	5

# Metrics alignment

2	1
0	7

5	3
1	6

9	5
2	3

4	4
7	2

6	1
4	5

2	3
6	5

1	9
4	0

8	-8
-8	5

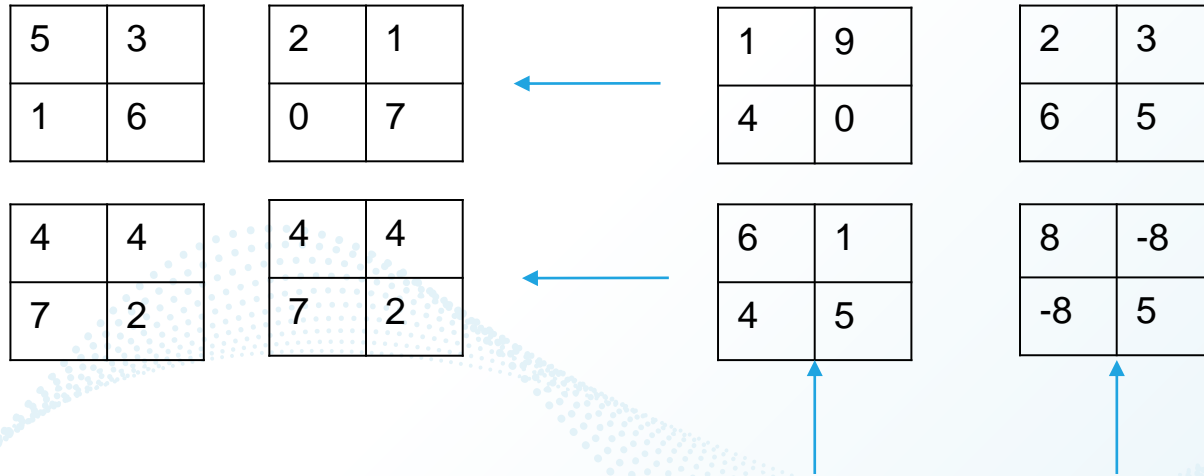
$$C_{0,0} = \begin{bmatrix} 2 & 1 \\ 0 & 7 \end{bmatrix} \times \begin{bmatrix} 6 & 1 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 16 & 7 \\ 28 & 35 \end{bmatrix}$$

$$C_{0,1} = \begin{bmatrix} 5 & 3 \\ 1 & 6 \end{bmatrix} \times \begin{bmatrix} 8 & -8 \\ -8 & 5 \end{bmatrix} = \begin{bmatrix} 16 & -25 \\ -40 & 22 \end{bmatrix}$$

$$C_{1,0} = \begin{bmatrix} 4 & 4 \\ 7 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 9 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 36 \\ 15 & 63 \end{bmatrix}$$

$$C_{1,1} = \begin{bmatrix} 9 & 2 \\ 5 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 6 & 5 \end{bmatrix} = \begin{bmatrix} 30 & 37 \\ 42 & 39 \end{bmatrix}$$

Now next step shifting operation is performed on A and B , Shift A one step left and B one step Up





# After performing local matrix multiplication

$$C_{0,0} = \begin{bmatrix} 16 & 7 \\ 28 & 35 \end{bmatrix} \times \begin{bmatrix} 17 & 45 \\ 25 & 9 \end{bmatrix} = \begin{bmatrix} 33 & 52 \\ 53 & 44 \end{bmatrix}$$

$$C_{0,1} = \begin{bmatrix} 16 & -25 \\ -40 & 22 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 42 & 35 \end{bmatrix} = \begin{bmatrix} 26 & -14 \\ 2 & 57 \end{bmatrix}$$

$$C_{1,0} = \begin{bmatrix} 20 & 36 \\ 15 & 63 \end{bmatrix} \times \begin{bmatrix} 62 & 19 \\ 42 & 33 \end{bmatrix} = \begin{bmatrix} 82 & 55 \\ 57 & 96 \end{bmatrix}$$

$$C_{1,1} = \begin{bmatrix} 30 & 37 \\ 42 & 39 \end{bmatrix} \times \begin{bmatrix} 0 & -12 \\ 40 & -46 \end{bmatrix} = \begin{bmatrix} 30 & 25 \\ 82 & -7 \end{bmatrix}$$

```
#include <cuda_runtime.h>
#include <iostream>

__global__ void matmul(int* A, int* B, int* C, int N) {
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if (Row < N && Col < N) {
        int Pvalue = 0;
        for (int k = 0; k < N; k++) {
            Pvalue += A[Row*N+k] * B[k*N+Col];
        }
        C[Row*N+Col] = Pvalue;
    }
}

int main() {
    int N = 512;
    int size = N * N * sizeof(int);

    int* A, * B, * C;
    int* dev_A, * dev_B, * dev_C;

    cudaMallocHost(&A, size);
    cudaMallocHost(&B, size);
    cudaMallocHost(&C, size);
```

```
cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

// Initialize matrices A and B
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        A[i*N+j] = i*N+j;
        B[i*N+j] = j*N+i;
    }
}

    cudaMemcpy(dev_A, A, size,
cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size,
cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
    matmul<<<dimGrid, dimBlock>>>(dev_A, dev_B,
dev_C, N);

    cudaMemcpy(C, dev_C
```

```
// Print the result
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        std::cout << C[i*N+j] << " ";
    }
    std::cout << std::endl;
}
```

```
// Free memory
cudaFree(dev_A);
cudaFree(dev_B);
cudaFree(dev_C);
cudaFreeHost(A);
cudaFreeHost(B);
cudaFreeHost(C);
```

```
return 0;
```

- In this program, the ``matmul`` kernel takes in the two input matrices ``A`` and ``B``, the output matrix ``C``, and the size of the matrices ``N``. The kernel uses the ``blockIdx.x``, ``blockIdx.y``, ``threadIdx.x``, and ``threadIdx.y`` variables to calculate the indices of the current thread. If the indices are less than ``N``, the kernel performs the matrix multiplication operation ``Pvalue += A[Row*N+k] * B[k*N+Col]`` and store the Pvalue in ``C[Row*N+Col]``.
- In the ``main`` function, the program first allocates memory for the input and output matrices on the host and initializes them. Then it allocates memory for the matrices on the device and copies the data from the host to the device using ``cudaMemcpy``.
- Next, the program launches the kernel with the appropriate grid and block dimensions. The kernel uses a 2D grid of thread blocks to perform the matrix multiplication in parallel.
- Finally, it copies the data from device to host using `cudaMemcpy` and prints the result using nested for loop. And it also frees the memory used.