PD_10_Kushagra Suryawanshi

HPC LAB ASSIGNMENT 3

Aim: Write a C program for bubble sort. Calculate its time complexity. Identify the hotspots in the program. Write an optimized code for bubble sort using OpenMP.

Objective:

1.Write sequential bubble sort
2. Calculate complexity of the program
3. Measure time taken by the sequential program.
4. Write program for Parallel Bubble sort
5. Measure and compare time taken by the parallel program

Theory:

Write about the serial bubble sort.

1. Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst-case complexity are of O(n2) where n is the number of items.
2. in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
3. To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e., the array requires no more processing to be sorted, it will come out of the loop.
4. After every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Write about the parallel bubble sort (Odd-Even Transposition).

Implemented as a pipeline.

Let local_size = n / no_proc. We divide the array in no_proc parts, and each process executes the bubble sort on its part, including comparing the last element with the first one belonging to the next thread.

Implement with the loop (instead of j<i)

```
 for (j=0; j<n-1; j++)
```

For every iteration of i, each thread needs to wait until the previous thread has finished that iteration before starting. We'll coordinate using a barrier. The local loop is the interior loop of the sequential algorithm, implemented in the region of the array assigned to the process/thread with the given id.

```
Local_loop(my_start, my_end)
{
 for (j=my_start; j< my_end; j++)
 if (a[j] > a[j+1])
```

```
 Swap(a[j], a[j+1]);
}
```

The pipeline organized is with a barrier. To create a delay in the global iteration in which each thread enters the pipeline, we sart with i having the value -id. Since i is incremented every time, and the local loop is not executed until i is non negative, thread 0 can start the local loop right away, thread 1 must wait until the second iteration, thread 2 until the third, and so on.

To avoid confusion in the number of threads entering the barrier after each iteration, the barrier always takes a parameter equal to the total number of threads. The delay in the parameter it ensures that all the threads are active and participating in the barrier every time, but only the ones that are supposed to execute the local loop do so.

The upper limit for i in the for loop insures that the total number of iterations for every thread (including those where it does nothing but the barrier), is identical to all the threads, and equal to the size of the array plus the number of threads.

The way my_start and my_end are computed, there is an overlap of 1 element of the array between each two threads in the array. The thread to the left of this element will compare it with the one before it, while the thread to the right of this element will compare it with the one after it.

```
void *Parallel_bubble_sort(void *arg)
{
 int id, i;
 Get_id(id);
 int lsize = size/no_threads;
 if (size % no_threads != 0)
 lsize++;
 int my_start = id*lsize;
 int my_end = min(size-1, (id+1)*lsize);
 for (i=-id; i<size+no_threads-id; i++) {
 if (i >= 0 && i<size)
 Local_loop(my_start, my_end);
 Barrier(no_threads);
 }
 if (id == no_threads-1)
 Output_array();
}
```

A version using conditional variables. This model doesn't work because for the bubble sort the dependency goes both ways: thread 1 should not start the iteration for i=0 until thread0 has finished this same iteration, but thread 0 should also not compare the last two elements in its array range in iteration 1 until thread 1 has compared the first two elements in iteration 0. The conditional variables insure that the dependencies going forward are respected, so the model could apply to any problem that does not present backward dependencies.

Each thread marks the iteration that is has finished the most recently (the value of i at the end of the local loop). In the function Synch_ith_previous, thread id makes sure that the thread id-1 has finished the iteration number i before proceding to start it itself. If the

previous thread hasn't reached that point, it goes into waiting for a conditional variable. In the synch_with_next, the thread writes its own index of the iteration it has finished, then signals the next thread that it can proceed.

Justify the usage of OpenMP in the parallel Bubble sort.

1) OpenMP is a widely adopted shared memory parallel programming interface providing high level programming constructs that enable the user to easily expose an application's task and loop level parallelism in an incremental fashion. The range of OpenMP applicability was significantly extended recently by the addition of explicit tasking features.

2) The idea behind OpenMP is that the user specifies the parallelization strategy for a program at a high level by providing the program code. OpenMP is the implementation of multithreading, a parallel execution scheme where the master thread assigns a specific number of threads to the slave threads and a task is divided between them.

3) Parallel loop: executing each iteration concurrently is the same as executing each iteration sequentially. No loop carries dependencies (an iteration does not produce any data that will be consumed by another iteration).

Test on data set of sufficiently large size.

Compute Total cost and Efficiency as :-

Total Cost = Time complexity × Number of processors used

Time Complexity = O(N) & Number of Processors used = 4

= (N * 4)

Efficiency = Execution time of sequential algorithm/Execution time of the parallel algorithm

= (338.989/47.527)

= 7.132556

Mention the number of processors / processor cores of your machine.

Processor Cores = 4

Consider data points as follows :-

Sr. No.	Data points/Data values	Time Taken for serial program	Time Taken for parallel program
1	500	**0.001097**	**0.038503**
2	1000	**0.008657**	**0.107103**
3	10000	**0.654536**	**0.873075**
4	50000	**17.05213**	**5.078123**
5	70000	**26.023348**	**10.437234**
6	100000	**45.28372**	**16.23812**

FAQs:

Q1. What are Various Decomposition techniques

A] Data Decomposition

1. Decompose the data

2. The decomposition is used to induce computational tasks Partitioning output data: each part of the output can be computed independently of the rest Each task computes parts of the output Partitioning input data: each tasks works on part of the input data It may not be possible to partition based on the output; e.g sorting a sequence, finding largest number

B] Recursive decomposition
Recursively subdivide the problem Solve subproblems in parallel Combine the results E.g. parallel quicksort

C] Exploratory decomposition
Assume we have a large solution space, and we need to search for a solution in it Decompose this space into smaller parts Search in parallel in these smaller parts to find a solution E.g. solving the 15-puzzle problem

D] Speculative Decomposition
This decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. Example: Speculative Decomposition Parallel discrete event simulation The nodes of a directed network have input buffer of jobs. After processing the job, the node put results in the input buffer of nodes which are connected to it by outgoing edges. A node has to wait if the input buffer of one of its outgoing neighbors is full. There is a finite number of input job types.

2. What are different task parameters.
1. Task Generation The tasks that constitute a parallel algorithm may be generated either statically or dynamically.
2. Static task generation refers to the scenario where all the tasks are known before the algorithm starts execution. Data decomposition usually leads to static task generation.
3. Task Sizes The size of a task is the relative amount of time required to complete it.
The complexity of mapping schemes often depends on whether or not the tasks are uniform; i.e., whether or not they require roughly the same amount of time.
4. If the amount of time required by the tasks varies significantly, then they are said to be non-uniform
5. Size of Data Associated with Tasks
6. Size of Data Associated with Tasks Another important characteristic of a task is the size of data associated with it.
7. Data associated with a task must be available to the process performing that task, the size and the location of these data may determine the process that can perform the task without incurring excessive data-movement overheads.
8. An interaction pattern is considered to be regular if it has some structure that can be exploited for efficient implementation.
9. On the other hand, an interaction pattern is called irregular if no such regular pattern exists

3. Elaborate on OpenMP private variable and their use.

Data scope attribute clause	Description
private	The **private** clause declares the variables in the list to be private to each thread in a team.
firstprivate	The **firstprivate** clause provides a superset of the functionality provided by the private clause. The private variable is initialized by the original value of the variable when the parallel construct is encountered.
lastprivate	The **lastprivate** clause provides a superset of the functionality provided by the private clause. The private variable is updated after the end of the parallel construct.

CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
    int SIZE = 100000;
    int A[SIZE];
    for(int i=0;i<SIZE;i++)
    {
        A[i]=rand()%SIZE;
    }
    int N = SIZE;
    int i=0, j=0;
    int first;
    double start,end;
int cores = omp_get_num_procs();
printf("Cores : %d",cores);
//serial
  start=omp_get_wtime();
    for( i = 0; i < N; i++ )
    {
        for( j = first; j < N-1; j += 1 )
        {
            if( A[ j ] > A[ j+1 ] )
            {
                int temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
            }
        }
    }
end=omp_get_wtime();
    // for(i=0;i<N;i++)
    // {
```

```
    //  printf(" %d",A[i]);
    // }

printf("\n Serial time = %f \n",(end-start));


//parallel
    start=omp_get_wtime();
    for( i = 0; i < N; i++ )
    {
        first = i % 2;
        if(first){
      #pragma omp parallel for default(none),shared(A,first,N)
        for( j = 0; j < N-1; j += 2 )
        {
            if( A[ j ] > A[ j+1 ] )
            {
                int temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
            }
        }
    }
    else{
      #pragma omp parallel for default(none),shared(A,first,N)
        for( j = 1; j < N-1; j += 2 )
        {
            if( A[ j ] > A[ j+1 ] )
            {
                int temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
            }
        }
    }
    }
end=omp_get_wtime();
    // for(i=0;i<N;i++)
    // {
    //  printf(" %d",A[i]);
    // }

printf("\n Parallel time = %f \n",(end-start));
}
```

```
kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ gcc -fopenmp -o hpc_bubble hpc_bubblesort.c

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./hpc_bubble.exe
Cores : 4
 Serial time = 0.001097

 Parallel time = 0.038503

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ gcc -fopenmp -o hpc_bubble hpc_bubblesort.c

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./hpc_bubble.exe
Cores : 4
 Serial time = 0.008657

 Parallel time = 0.107103

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ gcc -fopenmp -o hpc_bubble hpc_bubblesort.c

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./hpc_bubble.exe
Cores : 4
 Serial time = 0.654536

 Parallel time = 0.873075
```