Parallel Merge Sort

Title of the Assignment: Write a program to implement Parallel Merge Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Objective of the Assignment: Students should be able to Write a program to implement Parallel Merge Sort and can measure the performance of sequential and parallel algorithms.

Prerequisite:

1. Basic of programming language
2. Concept of Merge Sort
3. Concept of Parallelism

Contents for Theory:

1. **What is Merge? Use of Merge Sort**
2. **Example of Merge sort?**
3. **Concept of OpenMP**
4. **How Parallel Merge Sort Work**
5. **How to measure the performance of sequential and parallel algorithms?**

What is Merge Sort?

Merge sort is a sorting algorithm that uses a divide-and-conquer approach to sort an array or a list of elements. The algorithm works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves to produce a sorted output.

The merge sort algorithm can be broken down into the following steps:

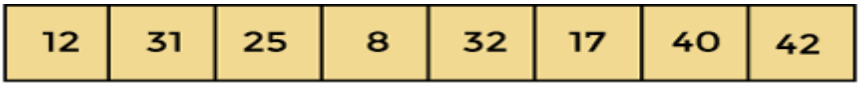
1. Divide the input array into two halves.
2. Recursively sort the left half of the array.
3. Recursively sort the right half of the array.
4. Merge the two sorted halves into a single sorted output array.

- The merging step is where the bulk of the work happens in merge sort. The algorithm compares the first elements of each sorted half, selects the smaller element, and appends it to the output array. This process continues until all elements from both halves have been appended to the output array.
- The time complexity of merge sort is O(n log n), which makes it an efficient sorting algorithm for large input arrays. However, merge sort also requires additional memory to store the output array, which can make it less suitable for use with limited memory resources.
- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.
- One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.
- Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Example of Merge sort

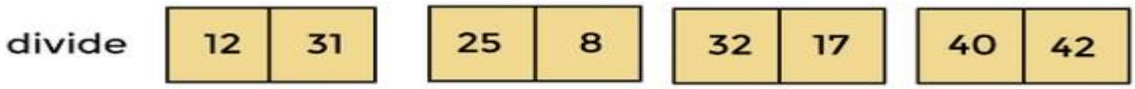
Now, let's see the working of merge sort Algorithm. To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -



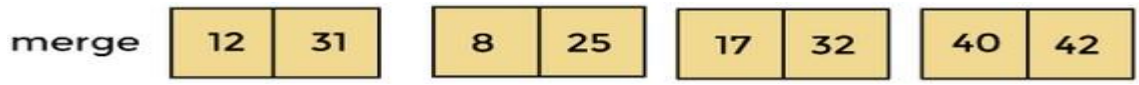
- According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.
- As there are eight elements in the given array, so it is divided into two arrays of size 4.

- Now, again divide these two arrays into halves. As they are of size 4, divide them into new arrays of size 2.

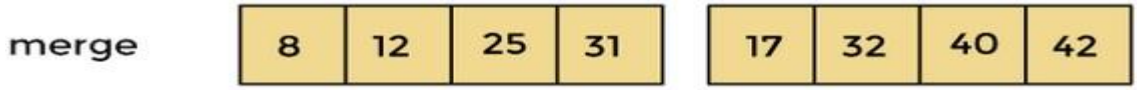


- Now, again divide these arrays to get the atomic value that cannot be further divided.

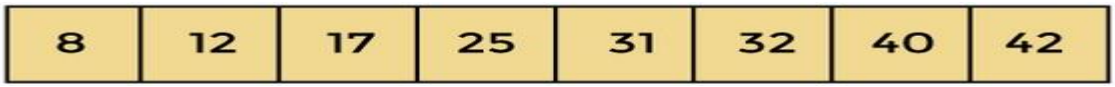
- Now, combine them in the same manner they were broken.
- In combining, first compare the element of each array and then combine them into another array in sorted order.
- So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



- In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



- Now, there is a final merging of the arrays. After the final merging of above arrays, the array will



look like -

Concept of OpenMP

- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.
- OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.
- OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuingwith the sequential part of the code.

How Parallel Merge Sort Work

- Parallel merge sort is a parallelized version of the merge sort algorithm that takes advantage of multiple processors or cores to improve its performance. In parallel merge sort, the input array is divided into smaller subarrays, which are sorted in parallel using multiple processors or cores. The sorted subarrays are then merged together in parallel to produce the final sorted output.
- The parallel merge sort algorithm can be broken down into the following steps:
- Divide the input array into smaller subarrays.
- Assign each subarray to a separate processor or core for sorting.
- Sort each subarray in parallel using the merge sort algorithm.
- Merge the sorted subarrays together in parallel to produce the final sorted output.
- The merging step in parallel merge sort is performed in a similar way to the merging step in the sequential merge sort algorithm. However, because the subarrays are sorted in parallel, the merging step can also be performed in parallel using multiple processors or cores. This can significantly reduce the time required to merge the sorted subarrays and produce the final output.
- Parallel merge sort can provide significant performance benefits for large input arrays with many elements, especially when running on hardware with multiple processors or cores. However, it also requires additional overhead to manage the parallelization, and may not always provide performance improvements for smaller input sizes or when run on hardware with limited parallel processing capabilities.

How to measure the performance of sequential and parallel algorithms?

There are several metrics that can be used to measure the performance of sequential and parallel merge sort algorithms:

1. **Execution time:** Execution time is the amount of time it takes for the algorithm to complete its sorting operation. This metric can be used to compare the speed of sequential and parallel merge sort algorithms.
2. **Speedup**: Speedup is the ratio of the execution time of the sequential merge sort algorithm to the execution time of the parallel merge sort algorithm. A speedup of greater than 1 indicates that the parallel algorithm is faster than the sequential algorithm.
3. **Efficiency:** Efficiency is the ratio of the speedup to the number of processors or cores used in the parallel algorithm. This metric can be used to determine how well the parallel algorithm is utilizing the available resources.
4. **Scalability**: Scalability is the ability of the algorithm to maintain its performance as the input size and number of processors or cores increase. A scalable algorithm will maintain a consistent speedup and efficiency as more resources are added.

To measure the performance of sequential and parallel merge sort algorithms, you can perform experiments on different input sizes and numbers of processors or cores. By measuring the execution time, speedup, efficiency, and scalability of the algorithms under different conditions, you can determine which algorithm is more efficient for different input sizes and hardware configurations. Additionally, you can use profiling tools to analyze the performance of the algorithms and identify areas for optimization

Conclusion- In this way we can implement Merge Sort in parallel way using OpenMP also come to know how to how to measure performance of serial and parallel algorithm

Assignment Question

1. **What is parallel Merge Sort?**
2. **How does Parallel Merge Sort work?**
3. **How do you implement Parallel MergeSort using OpenMP?**
4. **What are the advantages of Parallel MergeSort?**
5. **Difference between serial Mergesort and parallel Mergesort**

Reference link

- https://www.geeksforgeeks.org/merge-sort/

- https://www.javatpoint.com/merge-sort

Explanation

```c++
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;
```

This block of code is a set of header files and a namespace declaration. iostream and stdlib.h are header files, and omp.h is the header file for OpenMP (Open Multi-Processing) library that is used for parallel programming.

```c++
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

These are function declarations for mergesort and merge functions.

```c++
void mergesort(int a[],int i,int j)
{
    int mid;
    if(i<j)
    {
        mid=(i+j)/2;

        #pragma omp parallel sections
        {
            #pragma omp section
            {
                mergesort(a,i,mid);
            }

            #pragma omp section
            {
                mergesort(a,mid+1,j);
            }
        }

        merge(a,i,mid,mid+1,j);
    }

}
```

This is the mergesort function, which implements the merge sort algorithm recursively. a[] is the input array that is to be sorted, i and j are the starting and ending indices of the subarray being sorted.

First, it checks whether i is less than j because if i is greater than or equal to j, it means that there is only one element in the subarray, which is already sorted. Then it calculates the midpoint of the subarray using (i+j)/2.

The #pragma omp parallel sections directive creates a parallel region that specifies two parallel sections that can be executed concurrently. The mergesort function is then recursively called on the left and right halves of the subarray in two separate parallel sections.

Finally, the merge function is called to merge the sorted halves of the subarray.

```c
void merge(int a[],int i1,int j1,int i2,int j2)
```

This is the function signature which takes an integer array a and four integer variables i1, j1, i2, and j2. i1 and j1 define the start and end indices of the first sorted subarray, and i2 and j2 define the start and end indices of the second sorted subarray.

```c
int temp[1000];
int i,j,k;
i=i1;
j=i2;
k=0;
```

Here, a temporary array temp is created with a size of 1000. Three integer variables i, j, and k are initialized. i and j are set to the start indices of the two subarrays, while k is set to 0.

```c
while(i<=j1 && j<=j2)
{
    if(a[i]<a[j])
    {
        temp[k++]=a[i++];
    }
    else
    {
        temp[k++]=a[j++];
    }
}
```

This is a while loop that runs as long as i is less than or equal to j1 and j is less than or equal to j2. Inside the loop, if the element at index i of the first subarray is less than the element at index j of the second subarray, then the element at index i is copied to the temp array at index k, and i and k are incremented. Otherwise, the element at index j is copied to the temp array at index k, and j and k are incremented.

```
while(i<=j1)
{
    temp[k++]=a[i++];
}

while(j<=j2)
{
    temp[k++]=a[j++];
}
```

After the above loop terminates, there may be some elements left in one of the subarrays. These loops copy the remaining elements into the temp array.

```
for(i=i1,j=0;i<=j2;i++,j++)
{
    a[i]=temp[j];
}
```

Finally, the sorted temp array is copied back to the original a array. The loop runs from i1 to j2 and copies the elements of temp array to the corresponding indices in the a array. The loop variable j starts from 0 and increments alongside i.