Lab Assignment No. : 1

LP-V High Performance Computing

Aim :

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

Requirement :

64-bit Open source Linux or its derivative, C/C++ programming language; OpenMP

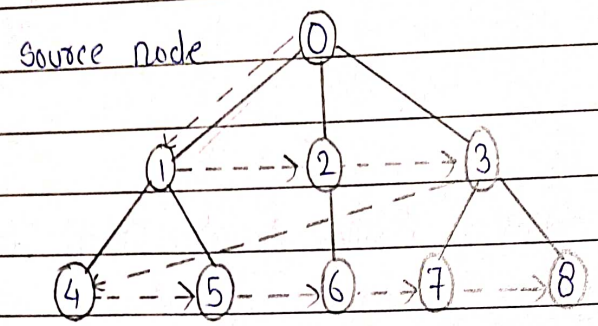
Theory :

Breadth First Search :

There are many ways to traverse graphs. BFS is the most commonly used approach. BFS is a traversing algorithm where we should start traversing from a selected node (source node or starting node) and traverse the graph layerwise. Thus exploring the neighbour nodes (nodes which are directly connected to source node). We must then move towards the next-level neighbour nodes.

As the name BFS suggests, we are required to traverse the graph breadthwise as follows :

① First move horizontally and visit all the nodes of the the current layer.



The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS you must traverse all the nodes in layer 1 before we move to the nodes in layer 2.

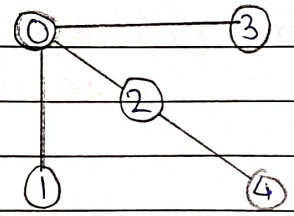
Depth First Search :

We parallelize DFS by sharing the work to be done among a number of processors. Each processor searches a disjoint part of seach done in a depth first search pattern. Since each processor searches the space in a depth first search manner, the state space to be searched is efficiently represented by a stack. The depth of the stack is the depth of the node being currently explored.

To test the effectiveness of parallel DFS, we have used it to solve the 15-puzzle problem. 15 puzzle problem is a 4×4 square tray containing 15 square tiles. The remaining 16th square is uncovered

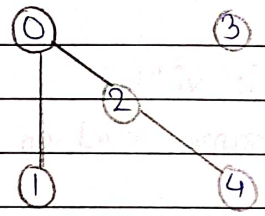
DFS is a pervasive algorithm, often used as a building block for topological sort, connectivity and planarity testing, among many other applications.

Example :



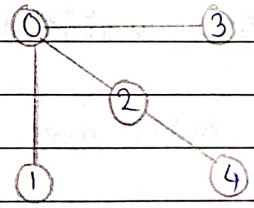
visited

stack



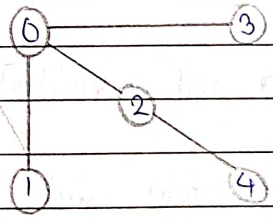
visited 0

stack 1 2 3



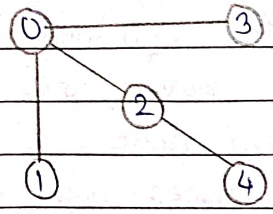
visited 0, 1

stack 2 3



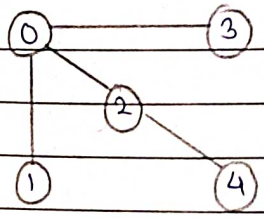
visited 0, 1, 2

stack 4, 3



visited 0, 1, 2, 4

stack 3



visited 0, 1, 2, 4, 3

Stack

Steps for searching :

In parallel BFS :

Step 1 : Start with the root node, mark it visited

Step 2 : As the root node has no node in same level, go to next level.

Step 3 : Visit all adjacent nodes and mark them visited.

Step 4 : Go to the next level and visit all the unvisited adjacent nodes.

Step 5 : Continue this process until all the nodes are visited or founded the required element.

In parallel DFS :

Step 1 : Consider a node (root node) that is not visited previously and mark it visited.

Step 2 : Visit the first adjacent successor node and mark it visited.

Step 3 : If all the successors node of the considered node are already visited or it doesn't have any more successor node, return to its parent node.

Step 4 : Continue this process until all the nodes are visited or founded the required elements.

Conclusion :

Conclusion :

Hence, we successfully studied and implemented parallel Breadth First Search and parallel Depth First Search using OpenMp.

Lab Assignment No. :-2

LP-V High Performance Computing

Aim :

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.

Requirements :

64-bit Open Source Linux or its derivative.
C / C++ programming, OpenMP

Theory :

Sorting :- Sorting is the process of arranging elements in a group of in a particular order, i.e. ascending order or in descending order, alphabetic order etc.

Parallel Sorting :
A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Bubble Sort :
The idea of bubble sort is to compare two adjacent elements if they are not in right order, switch them. Do this comparing and switching (if necessary) until we reach to end of the array. Repeat this process from the beginig of array n times.

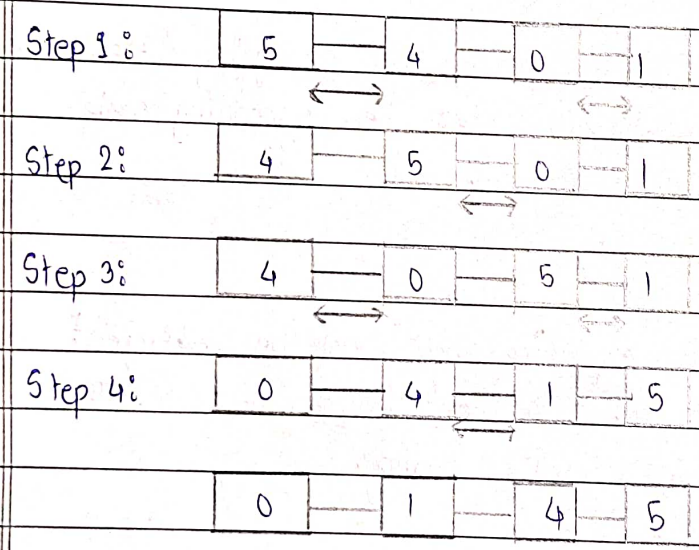
Parallel Merg
In parallel

Parallel Bubble Sort:

It is 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 it 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.

Example for Parallel Bubble Sorting:

[5, 4, 0, 1]



Merge Sort:

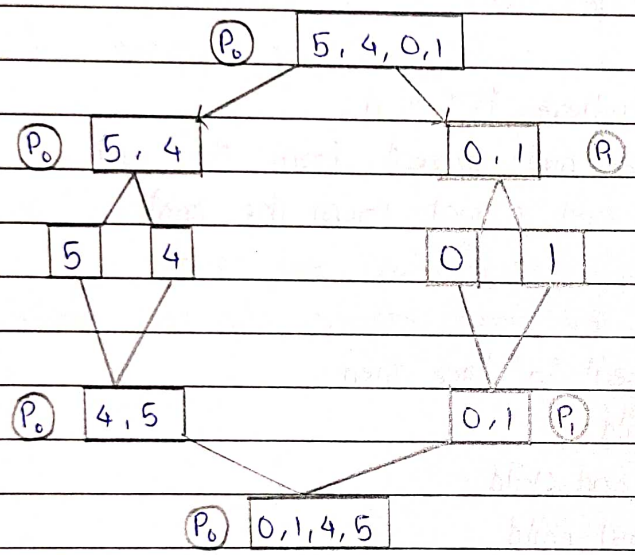
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity O(n log n). Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Parallel Merge Sort :

In parallel Merge sort we parallelise processing of sub-problems. Max parallelization achieved with one processor per-node (at each layer / height).

Example for Parallel Merge Sort :

[5, 4, 0, 1]



Algorithms :

For Parallel Bubble Sort :

Step 1 : For $i = 0$ to $n-2$

Step 2 : If i is even then

Step 3 : For $j = 0$ to $(n/2)-1$ in parallel

Step 4 : If $A[2j] > A[2j+1]$ then

Step 5: Exchange $A[2j] \leftrightarrow A[2j+1]$
Step 6: Else
Step 7: For $j = 0$ to $(n/2)-2$ do in parallel
Step 8: If $A[2j+1] > A[2j+2]$ then
Step 9: Exchange $A[2j+1] \leftrightarrow A[2j+2]$
Step 10: Next i.

Parallel Merge Sort:

```
1. Procedure parallel merge sort
2. Begin
3. Create processors Pi where i = 1 to n
4. if i > 0 then receive size and parent from the root.
5. Receive the list, size and parent from the root
6. Endif
7. midvalue = listsize / 2
8. if both children is present in tree then
9. send midvalue, first child
10. send listsize - mid, second child
11. send list, midvalue, first child
12. send list from midvalue, listsize - midvalue, second child.
13. call mergelist (list, 0, midvalue, list, midvalue+1, listsize, temp, 0, listsize
14. store temp in another array list 2
15. else
16. call parallel merge sort (list, 0, listsize)
17. endif
18. if i > 0 then
19. send list, listsize, parent
20. end if
21. end.
```

Algorithm Analysis :

1. Time complexity of parallel merge sort and parallel bubble sort in best case is (when all data is already in sorted form) $O(n)$.

2. Time complexity of parallel merge sort and parallel bubble sort in worst case is $O(n \log n)$.

3. Time complexity of parallel merge sort and parallel bubble sort in average case is $O(n \log n)$

Conclusion :

Hence, we successfully studied and implemented parallel Bubble sort and parallel Merge sort using OpenMP.

Lab Assignment No.: 3

LP-V High Performance Computing

Aim:

Implement Min, Max, Sum and Average operations using Parallel Reduction.

Requirements:

64-bit Open Source Linux or its Derivative,
C / C++ programming, OpenMP

Theory:

OpenMP:

OpenMP is a parallel programming model that allows for the creations of parallel programs using a set of compiler directives, library routines, and environment variables. OpenMP is particularly well suited for shared memory architectures where multiple processors can access the same memory space. In OpenMP, parallelism is achieved through the use of threads, which are lightweight processes that can be created and destroyed dynamically at runtime.

While implementation we used the OpenMP "parallel for" construct to parallelize the for-loop that iterates over the elements of the array. This construct splits the loop into smaller chunks, which are then executed in parallel by multiple threads.

The number of threads used is determined by the value of the 'OMP_NUM_THREADS' environment variable or by the

'omp_set_num_threads()' function. By default, OpenMP will use as many threads as there are processing cores avaible on the system.

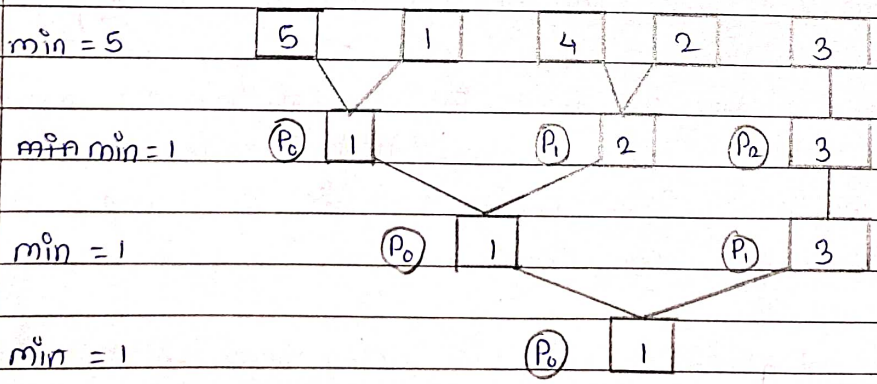
Overall, the use of OpenMP in this code allows us for efficient parallelization of the loop which can lead to significant speedup on multicore systems. The 'reduction' clause ensures that the results of the parallel computation are correct, even when multiple threads are updating the same variables simultaneously.

Operation Minimum (min):

The minimum value is the smallest value in the array. In implementation of this operation, the minimum value is initialised to the first element in array, an then compared to each subsequent element in loop. If smaller element is found the minimum value is updated accordingly.

Example:

[5, 1, 4, 2, 3]

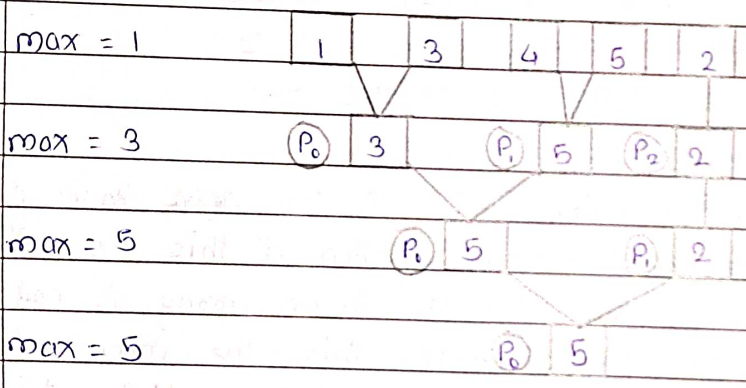


Operation Maximum (max):

The maximum value is the largest value in the array. In implementation of this operation, the maximum value is initialised to to first element in the array, an then compared to each subsequent element in the loop. If a larger element is found, the maximum value is updated accordingly.

Example:

[1, 3, 4, 5, 2]

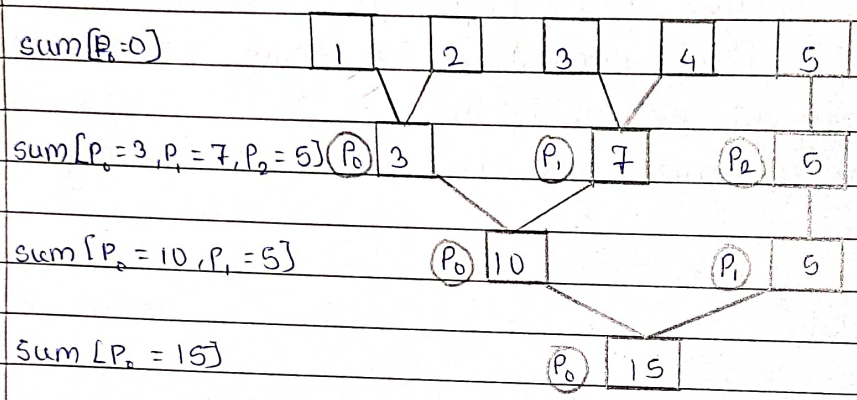


Operation Sum of values (sum):

The sum of values is the total of the values in the array. In implementation of this operation, the sum is initialised to zero, and then by using parallel threads we do summation of two array elements in one thread. After that the summation value of all threads are added into pairs. By performing this simultaneously we get the answers stored in only one thread, then the sum value is updated accordingly.

Example :

[1, 2, 3, 4, 5]

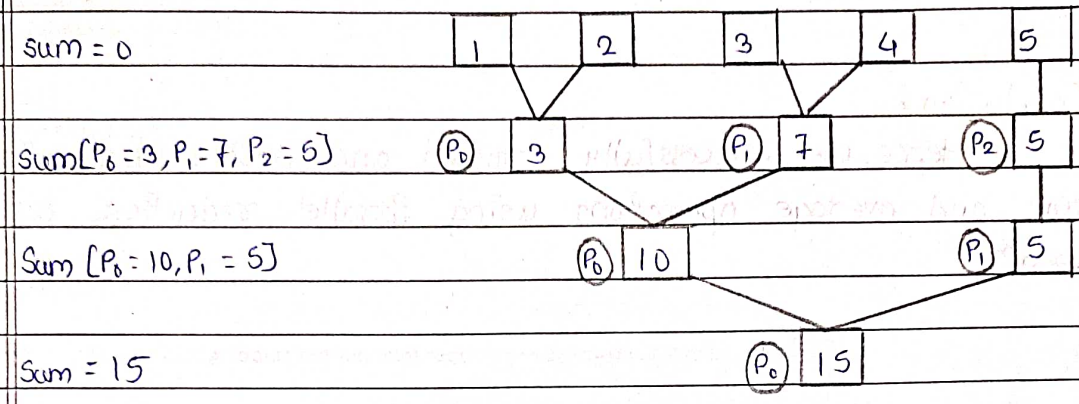


Average Operation (avg) :

The average operation value is the mean value of all elements in the array. In implementation of this operation, the ~~sum of all~~ mean of two elements in an array is calculated and stored in multiple threads. After this the mean values calculated are meaned by pairs and ranned on different threads. After that, by performing this simultaneously we get the final mean value stored in one thread and then the variable avg is updated by this new value generated i.e. average value of the elements present in the array.

Example :

[1, 2, 3, 4, 5]



Now, avg = sum / n

where, avg → average value

sum → summation of all elements in array

n → number of elements in array

∴ average avg = sum / n

= 15 / 5

avg = 3

∴ average = 3

All of the above operations are commonly used in data analysis and processing tasks. By using OpenMP to parallelize the loop that performs these operations, the code can be executed much faster than if it were executed serially. This can be especially useful when working with large datasets or computationally-intensive tasks.

Conclusion :

Hence, we successfully studied and implemented min, max, sum and average operations using Parallel reduction using OpenMP.

Lab Assignment No. : 4

LP-V High Performance Computing

Aim :

Implement HPC application for AI/ML domain.

Requirements :

64-bit Open Source Linux or its derivatives

C / C++ programming

OpenMP

Theory :

Here we will develop an application using OpenMP libraries and developing an application which will perform tokenization in C language.

Tokenization :

Tokenization is the process of breaking down a text into smaller components called tokens. Tokens can be words, phrases, sentences or any other meaningful unit of text.

Tokenization is a crucial step in many Natural Language Processing (NLP) applications, such as text classification, named entity recognition, and machine translation. There are many libraries available for tokenization. Here are some ways

OpenMP can be used ~~to~~ because OpenMP is a set of compiler directives and library routines for parallel programming in C and other languages. OpenMP can be used to parallelize the tokenization process, which can improve the speed of the process when dealing with large texts. The basic idea behind parallel tokenization using OpenMP is to split the text into smaller chunks and process each chuck in parallel using multiple threads.

To tokenize a text using OpenMP, we can split the text into smaller chunks and assign each chunk to a different thread. Each thread can then process its chunk of the text independently and generate a set of tokens. ~~To~~ To ensure that each thread processes an equal amount of text, we can use the 'omp_get_thread_num' function to get the ID of each thread, and use it to determine the chunk of text that it should process. We can also use the 'num_threads' clause to specify the numbers of threads to use.

While OpenMP can be a powerful tool for parallelizing tokenization, it's important to note that the performance gains will depend on the specific use case and the size of the text being tokenized. In some cases, the overhead of dividing the text into chunks and combining the results may outweigh the benefits of parallelization. As with any optimization technique, its important to measure the performance gains of parallelization and compare them to the overhead of implementing it.

The code reads lines of text from a file called 'example.txt' and uses openMP to tokenize each line in parallel. Here's a breakdown of how it works:

① The 'tokenize' function takes a single line of text as input and tokenizes it using 'strtok' function from the 'string.h' library. The function uses the 'omp_get_thread_num' function to get the ID of the current thread and uses a critical section to print the line being processed and the tokens generated by each thread.

② The 'main' function initializes the number of threads to use and opens the file for reading.

③ The '#pragma omp parallel' directive creates a parallel region and specifies the number of threads to use.

④ The 'while (fgets(line, MAX_LINE_LENGTH, file) != NULL)' loop reads each line of text from the file and passes it to the 'tokenize' function to be tokenized. Each thread processes its own chunk of the file independently.

⑤ Once all threads have finished processing their chunks of text, the program closes the file and returns 0.

Note that this code uses a critical section to ensure that each thread prints its output in a synchronized manner. This is because the 'printf' function is not thread-safe and can result in garbled output if multiple threads try to print to the console simultaneously. The critical section ensures that only one thread can access the 'printf' function at a time, thus avoiding race conditions.

#include <stdio.h>
#include <string.h>
#include <omp.h>
#define MAX_TOKENS
int toke

Note that this code assumes that each thread will generate only one token. If a thread generates multiple tokens, you may need to modify the code to allocate more memory for 'tokens' or to use a different data structure to store the tokens. Also, as with any optimization technique, it's important to measure the performance gains of parallelization and compare them to the overhead of implementing it.

Conclusion :

Hence we successfully developed an application using HPC & using OpenMP for AI/ML domain