Concurrent Programming

Lab 3 Report

Author: Sarayu Managoli

Program Overview:

The program is intended to implement of merge sort using OpenMP libraries for parallelization. The input file is dependent upon the inputs provided by the user. This program is expected to function like “sort -n” Linux command and output the sorted values onto the required file.

Brief Introduction of Merge Sort

It is said to be one of the most efficient sorting algorithms. Similar to quick sort, this too works on the principle of divide-and-conquer. It breaks down an array into smaller arrays until each of the arrays consists of only one element. Until the arrays are down to one element, merge sort is called. These single elements are later merged into forming a single sorted array. Here, the array is divided exactly from the middle element. Here additional storage is used as compared to quick sort. Merge sort does not work on the right array until the left array is sorted.

Brief Introduction of OpenMP

OpenMP consists of certain set of pragmas and APIs. A sequential program can be parallelized by the use of these pragmas. A key aspect of pragmas is that if a compiler does not recognize a given pragma, it must ignore it (according to the ANSI C and C++ standards). OpenMP allows the programmer to: create threads by specifying parallel regions, parallelize loops, specify scope of variable in parallel section and scheduling. OpenMP **uses -fopenmp** flag. OpenMP also allows the user to specify private and shared variables between threads.

Difference between OpenMP and Pthreads

Pthreads are POSIX standards whereas OpenMP is an industry standard. While OpenMP is used for one on one applications, pthreads are used for applications which consist of many short tasks for a few processors. OpenMP is used majorly in data analysis while pthreads are mainly used in operational environment. OpenMP does not consist of explicit locks or synchronizations. It also is not strictly built on pthreads. One of the concerns with OpenMP is that the synchronization is expensive compared to Pthreads. OpenMP code is usually faster than the ones run with pthread because OpenMP is seen to have lower cache misses. In general, the execution efficiency is higher with Pthreads than with OpenMP.

Parallelization Strategies

The omp pragma usage information is provided here. As a part of sort_merge() function, the OpenMP pragmas are used. The parallel sections construct is a shortcut for specifying a parallel construct containing a “sections” construct and no other statements. In C/C++, the semantics are identical to explicitly specifying a parallel directive immediately followed by a sections directive. The merge functionality is parallelized between the two halves of the array in this function by the use of **#pragma omp parallel sections**. **#pragma omp section** is used to define the scope of each section. Finally these parallelly sorted part of the array are sorted using the mergefunction().

Deliverables

As a part of the deliverables, the following artefacts are present in the ZIP folder:

1. Lab write-up
2. Source Files (in folder Code)
3. Makefile (in folder Code)

Source File - Code Organization

This file contains the source code for the implementation of Lab 0. The following functions are included as a part of the source file:

1. read_from_file(char *read_file_name,vector<int> &readarray) - Reads the integer values from the file specified in the command line argument.
2. write_to_file(char *write_file_name,vector<int> &writearray) - Writes the integer values from the arrays and stores it in a file as mentioned.
3. main(int argc, char **argv) - Initialises the variables and calls other functions.

Merge sort functions:

4. mergefunction(vector<int>& mergearray, int left, int mid, int right) - Merges the left and the right subarrays onto the main array.
5. sort_merge(vector<int>& mergearray, int left, int right) - Implements merge sort by arranging the elements in an ascending order. OpenMP libraries are used in this function.

Makefile

Makefile consists of two targets, all and clean. The steps to execute are also printed in the file.

Compilation steps:

Open command prompt in the folder Lab3/Code, type "**make**". This will create the executable "mysort". To delete the executable, type "**make clean**".

Execution steps:

To print the author's name, type "**./mysort --name**"

To run the program in normal mode, type "**./mysort [sourcefile.txt] [-o outfile.txt]**"

Error handling and extant bug:

Error handling has been done if incremental inputs are missing. Passing unexpected arguments has been classified as an error.

Extant bugs: The program does not handle the errors related to the inputs from the user being out of order. This can be considered as future scope of work.

Output:

```
jovyan@jupyter-sama2321:~/Lab3$ make
g++ -pthread -o mysort -fopenmp mysort.cpp mergesort.cpp
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To run the code, please follow the usage as mentioned below.
To print the author's name in the sorting program,
Usage: ./mysort [--name]
To run the sorting program in normal mode,
Usage: ./mysort [sourcefile.txt] [-o outfile.txt]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort --name
Sarayu Managoli
jovyan@jupyter-sama2321:~/Lab3$
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort
No arguments passed!
Usage: ./mysort [--name] [sourcefile.txt] [-o outfile.txt]
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort test_case1.txt
Num count = 5

Unsorted array is:
9
3
1
2
4
Sorted array is:
1
2
3
4
9

Output file not specified. Sorted output successfully written to stdout!
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort test_case1.txt -o
./mysort: option requires an argument -- 'o'
Invalid!
Num count = 5

Unsorted array is:
9
3
1
2
4
Sorted array is:
1
2
3
4
9

Output file not specified. Sorted output successfully written to stdout!
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort test_case1.txt -o soln1.txt
Num count = 5

Unsorted array is:
9
3
1
2
4
Sorted array is:
1
2
3
4
9

Sorted output successfully written to the specified file!
```

```
jovyan@jupyter-sama2321:~/Lab3$ ./mysort test_case1.txt -o soln1.txt klj
Too many arguments passed!
Usage: ./mysort [--name] [sourcefile.txt] [-o outfile.txt]
```

References

https://stackoverflow.com/questions/25833541/reading-numbers-from-file-c

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

https://codeyarns.com/2015/01/30/how-to-parse-program-options-in-c-using-getopt_long/

http://madhugnadig.com/articles/parallel-processing/2017/02/25/parallel-computing-in-c-using-openMP.html

http://www.spscicomp.org/ScicomP12/Presentations/IBM/Tutorial_5.OpenMP.pdf

https://software.intel.com/content/www/us/en/develop/articles/threading-models-for-high-performance-computing-pthreads-or-openmp.html?wapkw=(hp)

https://www.openmp.org/spec-html/5.0/openmpsu65.html