**Implementation and Performance Analysis of Parallel and Serial Counting Sort Algorithm using OpenMP**

Manish Padmannavar, Varra Charishma, Spoorthi Ashok Karani

(Usn: 1MS19CS406, 1MS18CS131, 1MS18CS123)

**Abstract:** A counting sort (ultra sort or math sort) is a sorting algorithm that sorts an array’s elements by calculating the number of occurrences of each distinct array element. The count is stored in an auxiliary array, and sorting is accomplished by mapping the count to an index in the auxiliary array.

**Keywords-** OpenMP, Parallel processing, parallelism, Counting, Aggregation.

**I. Introduction**

When comparing elements, no sorting algorithm can sort n elements in less than O(n log n) time. Fortunately, if we know some information about the elements ahead of time, we can sort them in other ways. Assume we are asked to sort n elements but are told that each element is in the range 0-k, where k is much smaller than n. We can use the situation to create a linear O(n) sorting algorithm. That’s the Counting type.

Counting sort uses the range (k) of the numbers in the array **arr** to be sorted. It makes an array of this length using this range. Each index I in array Bucket is then used to count how many elements in **arr** have the value I the counts stored in Bucket can then be used to place the elements in **arr** in their proper order in the resulting sorted array.

Counting Sort can be parallelized by partitioning the input array into as many partitions as there are readily accessible processors.

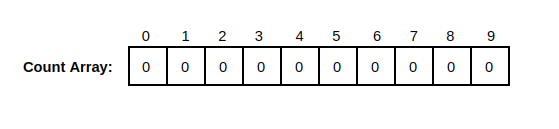
Each processor counts the elements of “its” partition in a separate auxiliary array during Counting the Elements. During Aggregating the Histogram, all auxiliary arrays are incorporated together to form one. During Writing Back Sorted Objects, each processor copies “its” partition’s elements to the target array. The fields in the auxiliary array must be decremented and read atomically.

**Input Array**

Table

Description automatically generated

**Initializing the Count array**

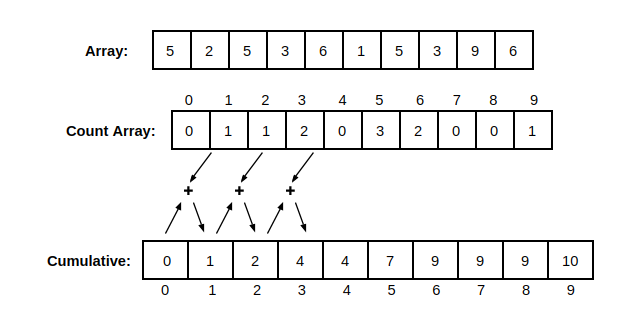


**Filling the Count Array**

A picture containing diagram

Description automatically generated

**Calculating Cumulative Frequency**



**Placing values in sorted array**

**Diagram

Description automatically generated**

**Print sorted Array**

**Table

Description automatically generated**

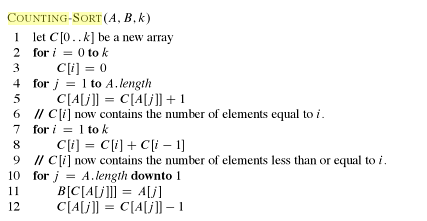
**Counting Sort can be parallelized** by partitioning the input array into as many partitions as there are readily accessible processors.

Each processor counts the elements of “its” partition in a separate auxiliary array during Counting the Elements. During Aggregating the Histogram, all auxiliary arrays are incorporated together to form one. During Writing Back Sorted Objects, each processor copies “its” partition’s elements to the target array. The fields in the auxiliary array must be decremented and read atomically.

**II. Literature Survey**

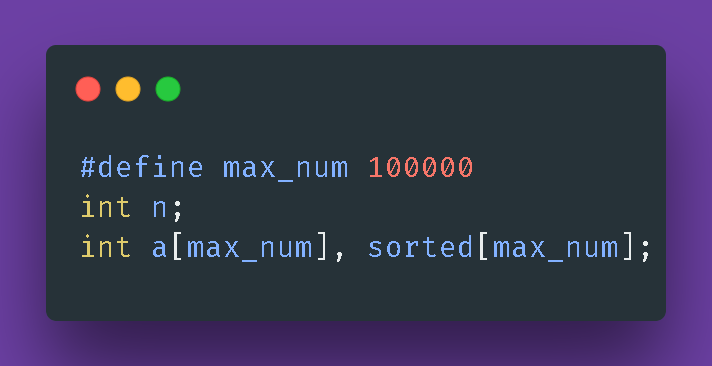
OpenMP is a shared-memory multiprocessing Application Program Inference (API) . It provides easy development of shared memory parallel programs. It provides a set of compiler directives for us to create threads, on top of pthreads manage the shared memory and synchronise the operations. The programs are compiled into multithreaded programs and the communications between threads is very systematic as threads share the same memory address space. A set of libraries is provided by the runtime as it maintains a thread pool as well [1]. To switch between sequential and parallel sections, which follow the fork/join model it uses a block-structured approach . A single thread is split into a number of threads at the entry of a parallel block, and a new sequential thread is started when all the split threads have finished . A fine-grained control over the threads is allowed by the directives[2]. Willy Zwaenepoel, Y.Charlie Hu, Honghui Lu, and Alan L Cox (1999), implemented OpenMP on a network of shared memory multiprocessors. The programmer relies on a single, standard shared-memory API for parallelization within a multiprocessor and between multiprocessors. John Bircsak, Zarka Cvetanovic, Peter Craig, Jonathan Harris, RaeLyn Crowell, Carl D. Offner and C. Alexander Nelson (2000), in their paper, describes extensions to OpenMP that implement data placement features needed for Non-Uniform Memory Access (NUMA) architectures. It is required that a programmer controls the placement of data in memory and the placement of computations that operate on that data in order to write efficient parallel programs for NUMA architectures, which have characteristics of both shared-memory and distributed-memory architectures. When computations occur on processors that have fast access to the data optimal performance is obtained . OpenMP does not by itself address these issues. It is supported on various platforms like LINUX, Windows and UNIX and various languages like C, C++, and Fortran [2]. Some of the advantages of OpenMP are 1) The compiler takes care of transforming the sequential code into parallel code according to the directives, therefore OpenMP is much easier to use[2]. 2) Without serious understanding of the multithreading mechanism, the programmer can write multithreaded programs [1].

**III. Method**

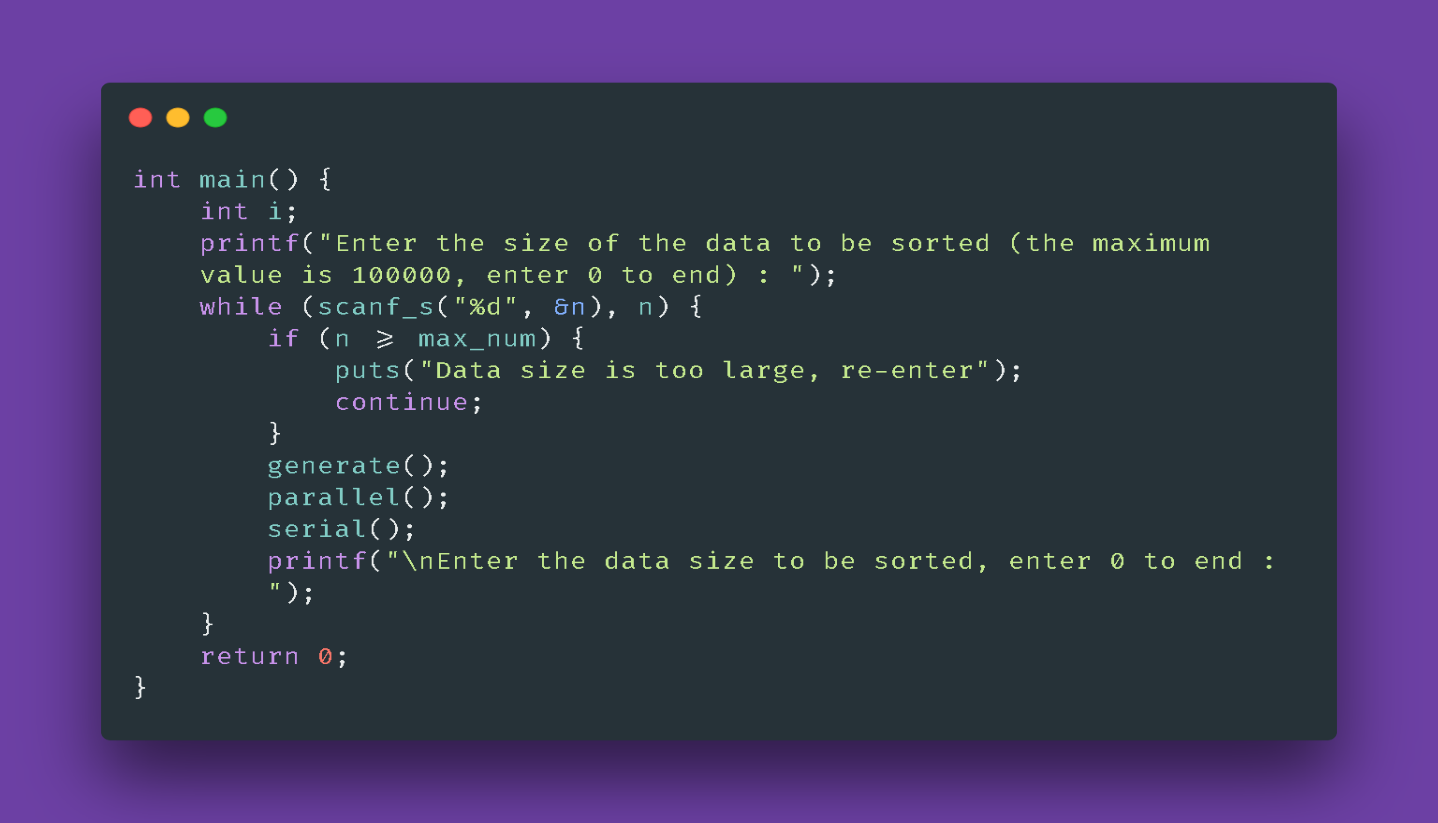
****

**Implementation in C**

We define the upper bound and arrays needed to the process. Here, a is the original array and sorted is the sorted array.

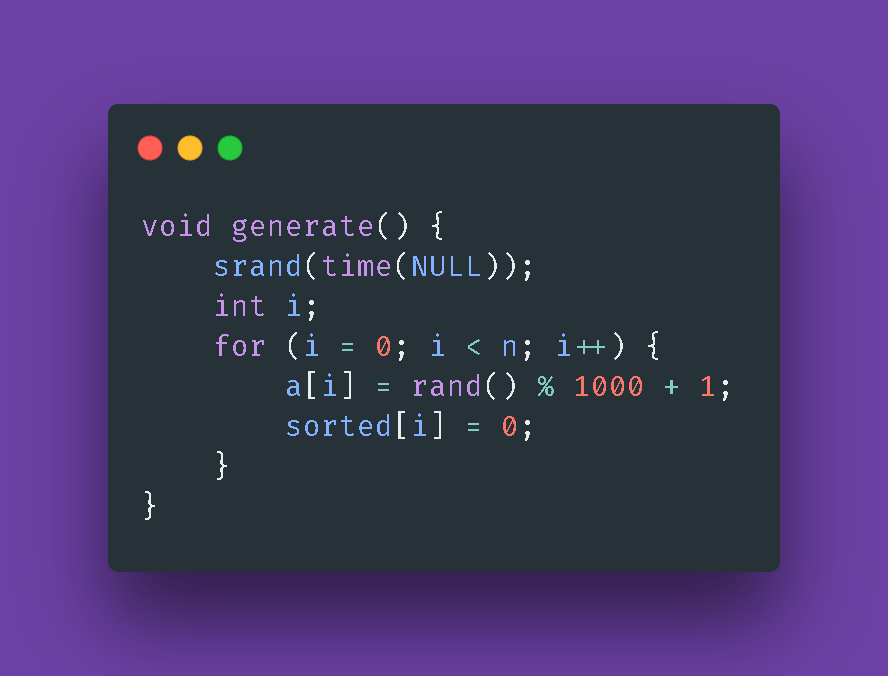


We define the main method as follows.

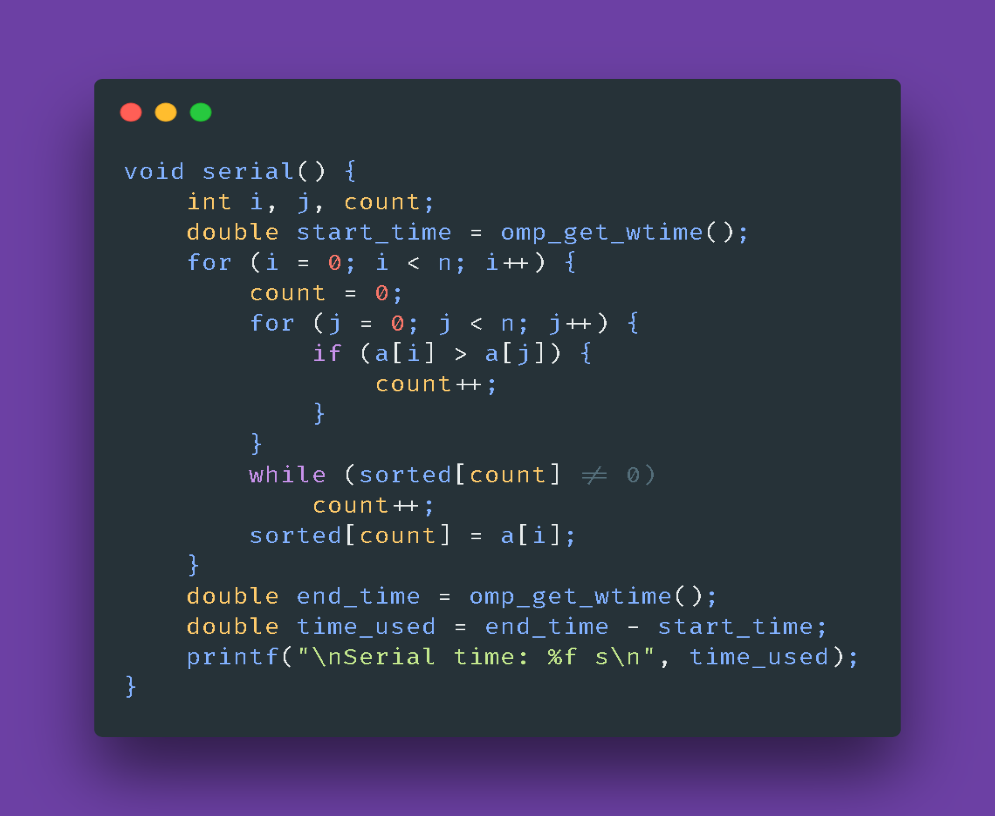


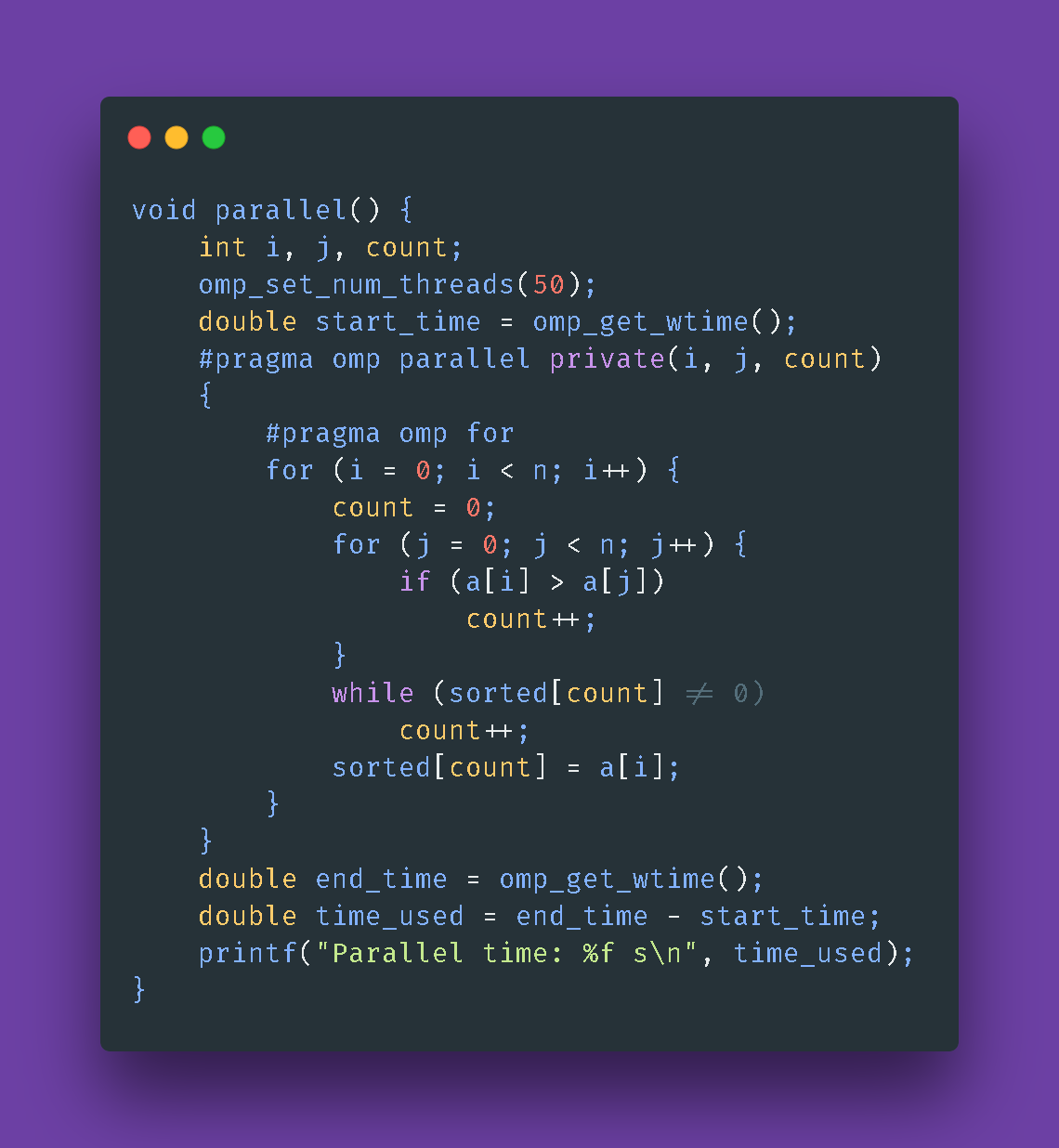
Then we create Three methods generate(), serial() and parallel():

generate() is to create array with random numbers. parallel() to execute parallel Counting Sorting and serial() is to execute serial Counting Sorting.



**serial()**



**parallel()**

**IV. Results**

Output using 10 threads

Graphical user interface, text

Description automatically generated

Output using 20 threads

Graphical user interface, text

Description automatically generated

Output using 50 threads

Graphical user interface, text

Description automatically generated

From the above outputs we can infer that in serial mode(10 threads) the algorithm took about 0.73 seconds to sort the array of 10,000 random numbers while in parallel while in parallel it took only about 0.18 seconds to sort the same array. We also notice improvement in performance after increasing the number of threads with 20 threads serial time was 0.72s and parallel was 0.17s and for 50 threads it was 0.72s for serial and 0.16 for parallel.

The results were obtained on an Intel i5 7200u processor (2 cores and 4 threads).

**References:**

[1] OpenMP Architecture Review Board, ―OpenMP Application Program Interface,‖ 2008, <http://www.openmp.org/mp-documents/spec30.pdf>.

[2] Parallel Programming with OpenMP in C | by Samir Huseynzade | Medium

[3] Parallel programming with OpenMP, <https://web.njit.edu/~shahriar/class_home/HPC/omp2.pdf>

[4] Comparison and Analysis of Parallel Computing Performance Using OpenMP and MPI, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1000.940&rep=rep1&type=pdf>

[5] Counting Sort Algorithm Explained with Examples | Medium Article.