CSC2002S: Assignment 1 report 2021

Aug 22

Authored by: Kevin Chiloane CHLKEV001

Parallel programming with java Fork/join FrameWork:1D Median Filter

Introduction

An Introduction, comprising of a short description of the aim of the project, the parallel algorithms and their expected performance.

The aim of the program is to accepts raw data with outliers and filter the raw data by it's median as to remove noise. It is of importance that the filtering process run within a specific time and parallel programming can achieves this through a good rate of acceleration.

The following is the requirement that need to be met to successfully to obtained a working and precise parallel program of the solution

-Code a parallel program of the medianFilter
-The use of the java Fork/join framework
-Using the recursive divide and conquer algorithm
-Through the basis of good
:Filtering size
:Sequential cutoff
:performance CPU
Use of precise timing to evaluate the run time of the parallel and Sequential program, across a varied range of input sizes.

Methods

This section details the approach taken to successfully accomplish the given problem with the best results ever. The following are the algorithms used to product the solution of the task at hand.

Public class SequentialMedianFilter{}

SequentialMedianFilter is the first class of the serial version of the solution. The class uses one thread to process the raw data.

The class consist of a method called:

ReadInData that accepts a filename and read content of the file to an array.

Start and stop to time the runtime of the program in milliseconds

Run method to run the main program

writeToFile to save our output data to a text file

filter that accepts array of unfiltered data and filtering size, use filter width to determine the number of neighbours to consider for each element both right and left, which ensures that the program remains within it's domain.

Public class ParallelMedianFilter extends RecursiveAction{}

The parallel class provide a parallel version of the solution to the filtering problem. It extends the recursiveAction, parallel gives context of parallel computing as it encapsulates tasks and divide into subtasks and the final outcome of the task is combination of the subtask.

The class consist of a method called:

protected Double compute() which has the code that represents the computational portion of the task and overrides the compute() of RecursiveTask.

The method uses the algorithm divide and conquer recursively to obtain median of the right and left boundary to trim the data.

The first step for validating the algorithm is to compare the output produced after providing sampleinput.txt file for varied datasets with both serial and parallel programes.

Performance measure is implemented by looking at the difference in run times for the serial and parallel method, across a range of input sizes supplied.

Looking at the parallel method within ParallelMedianFilter.java in particular the speedup is measured from the moment forkJoinPool invokes ParallelMedianFilter.java which performs the parallel computation till it ends.

The architecture used:
Lenovo: intel core i7-106567, 10th Gen , 4 cores and 8 Logical processors

Problems encountered:

- Noticed that the performance of both methods appeared to change rapidly for the same input data for some time.

- I'm using virtual box for ubuntu, turns out the 10GB reserved for ubuntu was depleted "No space left on device" had to go on a youTube spiral to get info to allow more space to run my tests.

RESULTS AND DISCUSSION

The following results are influenced by data size and number of threads on parallel performance.

Sequential program:

A varied number of input sizes was fed into the sequential program to establish high-precision timing for evaluating the run times.

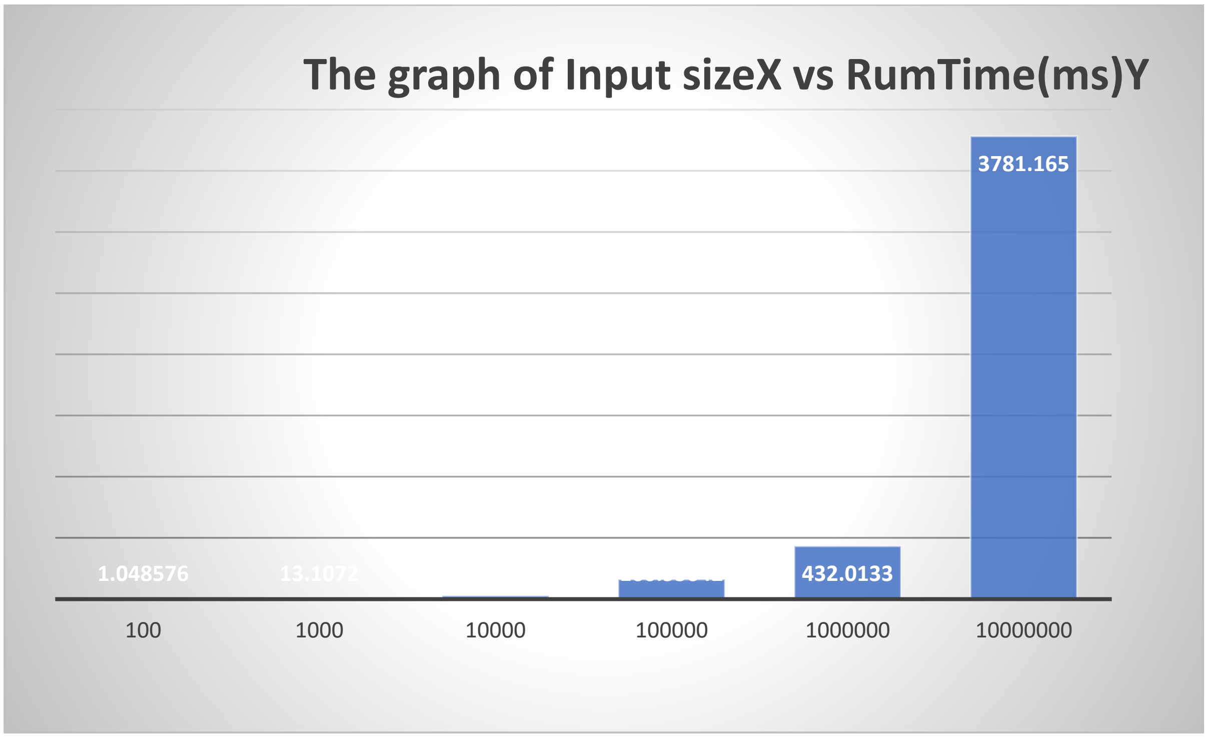
Experimenting with different parameters to establish the limits at which sequential processing should begin.

Figure 1A:

Shows the range of input file sizes with respective run times for sequential method.

Input Sizes	100	1_000	10_000	100_000	1_000_000	10_000_000
RunTime(Ms)	1.048576	13.1072	27.787264	160.95642	432.0133	3781.165

Figure 1B:



The run time starts to rapidly increase after the input size of 100_000 and the algorithm behavior is acceptable given the huge dataset . This is an exponential graph of run time against different input file sizes.

<u>Parallel program:</u>

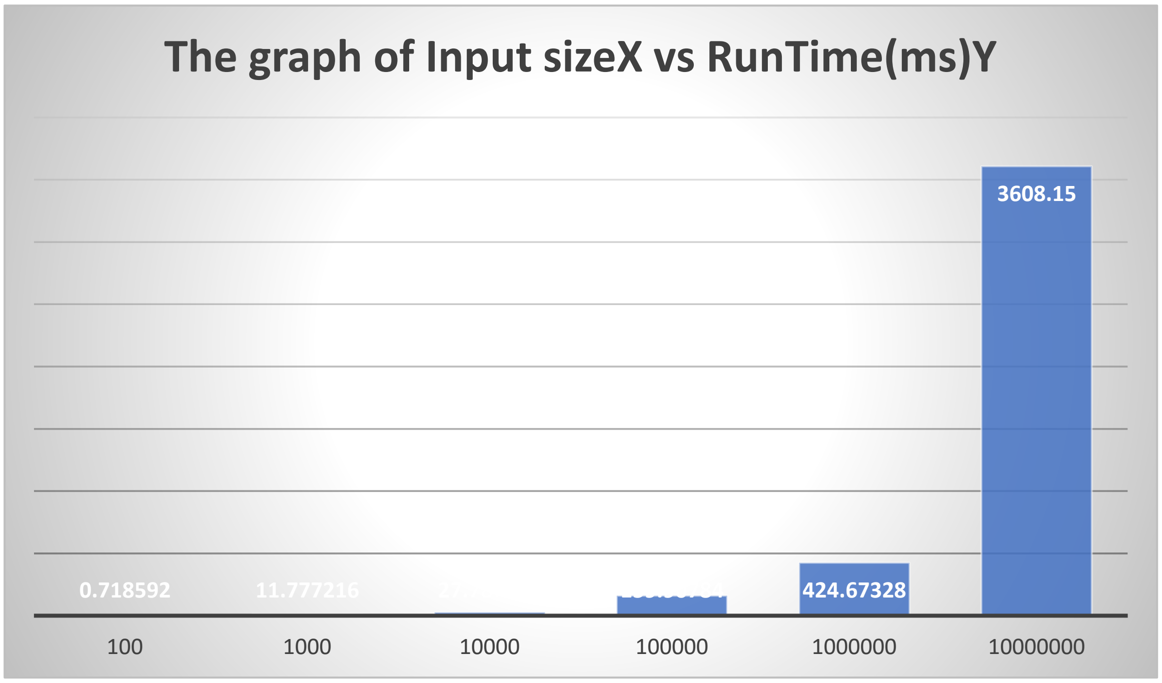
A varied number of input sizes was fed into the parallel program to establish high-precision timing for evaluating the run times. Experimenting with different parameters to establish the limits at which sequential processing should begin.

Figure 2A:

Shows the range of input file sizes with respective run times for a recursive divide and conquer method.

Input size	**100**	**1_000**	**10_000**	**100_000**	**1_000_000**	**10_000_000**
RunTime(ms)	**0.718592**	**11.777216**	**27.787264**	**159.90784**	**424.67328**	**3608.15**

Figure 2B:



Shows an exponential relationship between time and input size using a constant sequential cut-off, rapid increase in time given the dataset.

<u>Discussion:</u>

It is advantageous to solve this problem by parallelization using Fork/join framework whilst implementing the recursive divide and conquer algorithm as of result the task is divided and processed way faster than sequentially.

The parallel program performs best at wide range of input sizes, that is from 100_000 input sizes to 10_000 _000, Perhaps the architecture has a different optimal number of threads, as the sequential did compete slightly with the parallel approach.
It is believed that for each core there is a thread so the number of threads for the 4 core architecture is 4.

Conclusion(s):

Presented with the facts of the experiment and outputs produced whilst testing, one can recommend that the task of MedianFiltering for the purpose of removing noise from data should be tackled using parallelization supported by effective time scale in milliseconds sequential cut-off Bar.

This claim is supported by the experimental data displayed in the results and Discussion section.

Firstly, consider benchmarking using a high- precision timer by varying problem size and high performing architecture for both the serial and parallel methods:

-In overall the architecture produced better speed-up as
-The architecture managed to read in data from large input

file sizes and write it out without hassle.

- The sequential method approximately similar to the serial version excelled for small input text files.

Secondly,

The comparison of the sequential version of the program and the parallel version:

- The parallel version has good run times across a wide range of input sizes mainly because the algorithm performs a divide-and-conquer approach to the tasks.

- The two classes perform almost similar up until the sequential cut-off becomes 100_000 and greater