Assignment 1: Threaded KNN

Steven Hernandez
Department of Computer Science
Virginia Commonwealth University
Richmond, VA, USA
hernandezsm@vcu.edu

I. Introduction

For this assignment, the goal was to develop a k-nearest neighbor (KNN) classifier using both sequential as well as parallel techniques. Figures in this report use a KNN classifier with *k=5* using the provided 'medium.arff' dataset.

II. Analysis

Results of the KNN classifier in both sequential and parallel configurations can be seen in Table I. Parallel execution takes longer than sequential when the number of threads *N=1*. This is intuitive because the parallel version of the code must handle additional logic such as the splitting of the work between each thread even when configured to only use one thread as well as the calculation of work to be completed by each thread. Overall, threads take more overhead to instantiate the thread along side additional logic required for handling multiple parallel threads.

However, we can see that this additional work and overhead pays off when $N > 1$. As we increase *N* increases, CPU time decreases immensely. Equation (1) calculates the speedup achieved by the use of additional threads by comparing the time it takes to run in serial T_S, along with the time it takes to run in parallel T_P.

$$speedup = \frac{T_S}{T_P} \tag{1}$$

A. How much code was parallel?

In a theoretical fully-parallel program, doubling the number of threads would equally double the speedup of the program. However, our results do not reveal this linear relationship between number of threads and speedup. As we know, threads require some overhead in addition to the fact that the program here is not fully-parallel; instead relying on sequential code for initialization as well as termination.

Estimating the percentage of parallel code in our program $P_{estimated}$ can be achieved through (2) where *N* is the number of processors and *speedup* is the speedup calculated for the given number of threads.

$$P_{estimated} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1} \tag{2}$$

Table I presents the estimated results of this calculation, showing a best case of 96.12% parallelism achieved. Fluctuations of estimated parallelism arise as a result of thread creation overhead. Given 96.12% parallelism, we can use Amdahl's Law (3); where *P* is the percentage of parallelism and *N* is the number of threads, to calculate a theoretical parallel-system capable of running an infinite number of threads without any overhead.

$$speedup = \frac{1}{\left((1-P) + \frac{P}{N}\right)} \tag{3}$$

Of course, given $\lim_{N \to \infty} \frac{P}{N} = 0$ this can be rewritten simply as:

$$speedup_{maximum} = \frac{1}{(1-P)} \tag{4}$$

Thus, with the parallel code written for this report our theoretical parallel-machine can achieve a maximum *25.77* times speedup when compared to its sequential counterpart resulting in an optimal CPU time of *355 ms*. Do note however that our code could be further optimized for such a theoretical parallel-machine. Understanding the implemented KNN classifier runs in $O(n^2)$, the code here only parallelizes the classifier down to $O(n)$ time. However, it is possible to further reduce this given the infinite number of threads in this theoretical parallel-machine to achieve $O(1)$ time.

B. Result of 2048 Threads

Given theoretical speedup of *25.77*, we should see CPU time drop down to at the minimum *355 ms*. To test this, the parallel code is forced to run with 2048 threads in place of infinite threads to determine whether this is in fact the case. As we can see in Table I, CPU time does in fact reach very close to the theoretical optimal minimum CPU Time expected of *355 ms*.

TABLE I. Run Time for Each Configuration

Number of Threads	*CPU time (ms)*	*Accuracy*	*Speedup*	*$P_{estimated}$*
0 (sequential)	8878	0.5521	N/A	N/A
1	8940	0.5521	0.993	N/A
2	4763	0.5521	1.864	0.9270
4	2536	0.5521	3.501	0.9525
8	1411	0.5521	6.292	0.9612
2048	379	0.5521	23.42	0.9578