Parallel Computer Architecture

Project 1
OpenMP and PThreads Implementations

Apostolos-Nikolaos Vailakis
2014030174

Tzanis Fotakis
2014030054

Introduction

Within the context of this assignment, different ways of executing and parallelizing a simple hamming algorithm where explored. The requested algorithm would compare two arrays of strings (arrays “a” and “b”) and create a two dimensional matrix registering their hamming distances. Using this basic concept, the EP[1] algorithm was then implemented in different granularities using both the OpenMP API and PThread standard comparing each implementation’s time of execution.

The algorithm

Between two strings of equal length, the number of positions at which the corresponding symbols are different is called their “Hamming Distance”. In this example the strings are composed using only the symbols “1” and “0” (and are actually implemented using arrays of integers) making this a Binary Hamming Distance. The two arrays of strings, as well as the strings themselves, are of length m , n and l respectively. The above variables, including the number of threads executing the algorithm, are requested as arguments by the user.

[1] Embarrassingly Parallel

Implementation

The algorithms are implemented in C language and compiled using gcc. After allocating the desired space in memory each array is filled with random strings of 1s and 0s to be later used for the Hamming Distance calculations. Using those strings an m*n matrix is filled with all the strings' Hamming Distances using a serial method (nested loops) and it's time of execution is recorded to be used as a benchmark for later tests. Following the Serial method are the Parallel OpenMP and PThreads methods, each using 3 different granularities of tasks and measuring each time of execution. The 3 levels of granularities are as follows (Coarser to Finer):

1. Each taks compares one string from array "a" with all strings from array "b".
2. Each task compares one string from array "a" with one string from array "b".
3. Each task compares one character from a string of array "a" with the corresponding character from a string of array "b".

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Using these directives the algorithm was modified and the three different granularities of parallelization where implemented and tested on different sizes of datasets. This modification required the use of locks in the finest-grain implementation for possible race conditions between two or more threads comparing characters from the same string. The Chart #1 demonstrates the algorithm's efficiency in time/comparison as the number of total comparisons increase. We can see the "by row" method being the most effective as it has the least communication and synchronization overhead. It is also evident by the Char #2 that the third method "by char", which is the finest-grain method, is the one yielding the least speedup with each increase of physical processors, due to communication overhead of the locks mentioned above. This comes to highlight the importance of correctly chosen granularity level, something that will be even more evident in the next implementation using PThreads.

Chart #1

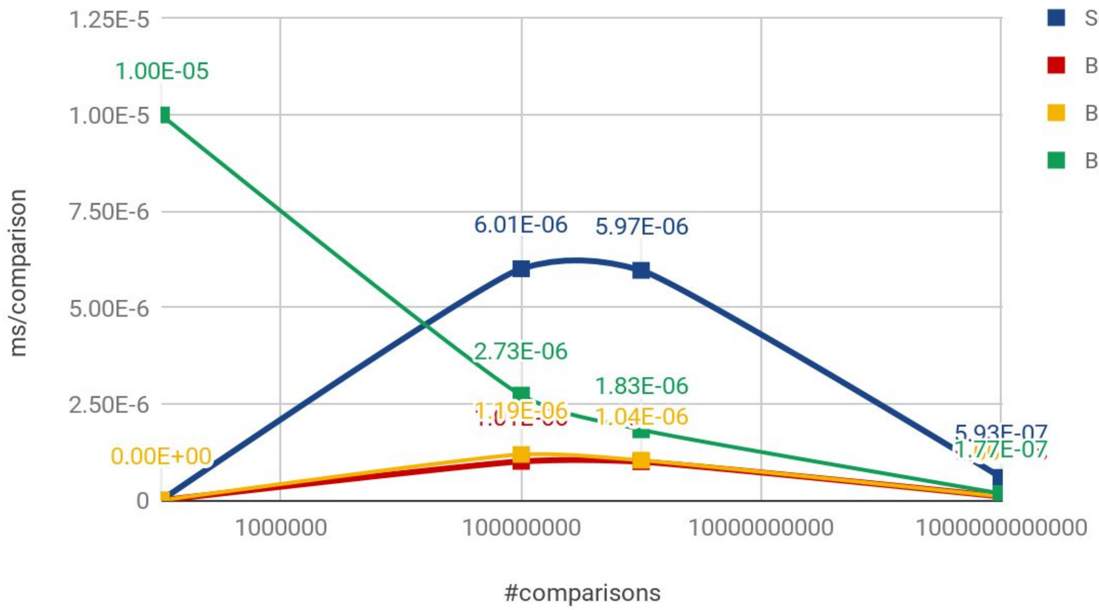
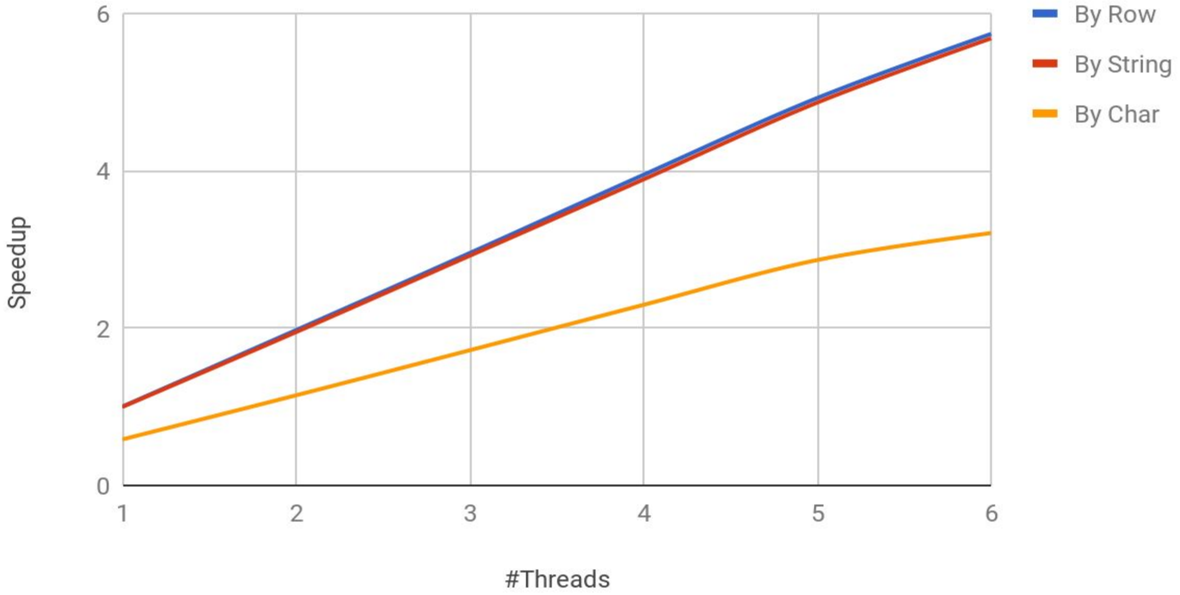


Chart #2



PThreads

POSIX Threads, usually referred to as **pthreads**, is an execution model that exists independently from a language, as well as a parallel execution model. This was also used to implement the three cases of granularity. The implementations were done by using mutexes to ensure the atomicity of read and write access of specific variables, avoiding with such a way possible race conditions. The idea behind each granularity's case algorithm is that there is a bank of tasks that all spawned threads have to take care of and complete. When a thread finishes its current task it has to try and find another non-completed task if it exists. This is easily implemented using a single counter variable that each thread increases when it undertakes a new non-completed task. This counter represents an index on the 2-dimensional array where the hamming distances are stored for each string of arrays a and b. This counter has to be accessed atomicly so that no more than one thread undertakes a new task and so all threads have different to each other tasks to complete. There comes the need for a mutex to lock when each thread need to access the counter so that no other thread can read or increase it, and unlock it when the task is assigned. In the finest granularity case where a task is to compare a character in a string between the input arrays a and b, there is a need for a number of mutexes, equal to n*m, one for each string comparison. Those are the ones used to ensure that no other thread writes to the same cell as the current does a given moment so that no thread will be "skipped" -> possibly write the same number as the other thread. An example for this case is shown below. Let us assume that register R1 represents the current cell of the hamming distances array used and its value equal to 5.

Thread 1	Thread 2
Read R1	
	Read R1
Add 1 to R1	Add 1 to R1
	Write Result to R1
Write Result to R1	

The result of the above example will be R1=6 although both threads 1 and 2 added to its value the number 1. So atomicity is needed, which is implemented by uses the already mentioned mutexes.

From chart #3 and #4, it can be observed that the most efficient way of implementing into a multithreading algorithm the hamming distance problem using PThreads is by using the second granularity method, each task compares two strings, one from each input array. Similar results is given by the first, "by-row" method. However, the significance of the communication and synchronisation overhead can be recognized by observing the third or "by character" method where each thread has to try and lock the mutex to undertake a task, which means waiting until the mutex is unlocked, then unlock the locked mutex and afterwards try to lock the cells mutex, which also means waiting until it is unlocked, then unlock it and repeat this process again. At last, not only this method makes the process to some great extent serial, but also it adds up to the whole computation time a great overhead for locking and unlocking the mutexes and waiting for others to finish.

From chart #4, it is a fact that the speedup increases with the number of threads for the first and second granularity case, whereas for the third case the speedup decreases as a result of the great overhead added as mentioned above. However, there will always be a maximum speedup (peak) using either the first or the second method, which cannot be demonstrated because of the lack of available cores.

Chart #3

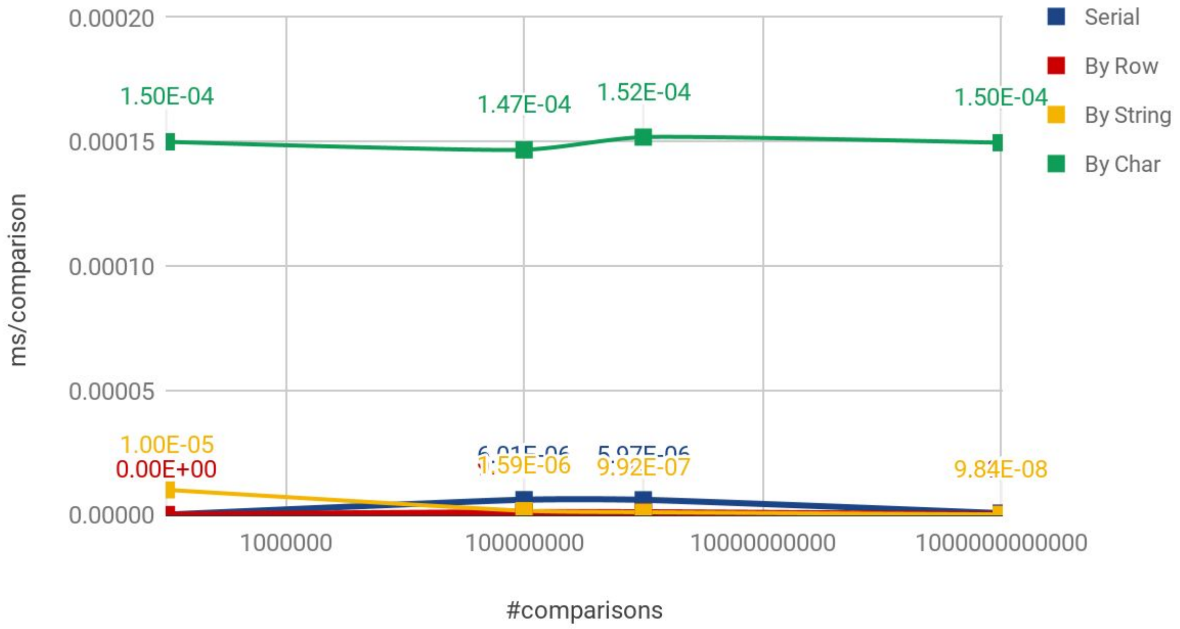
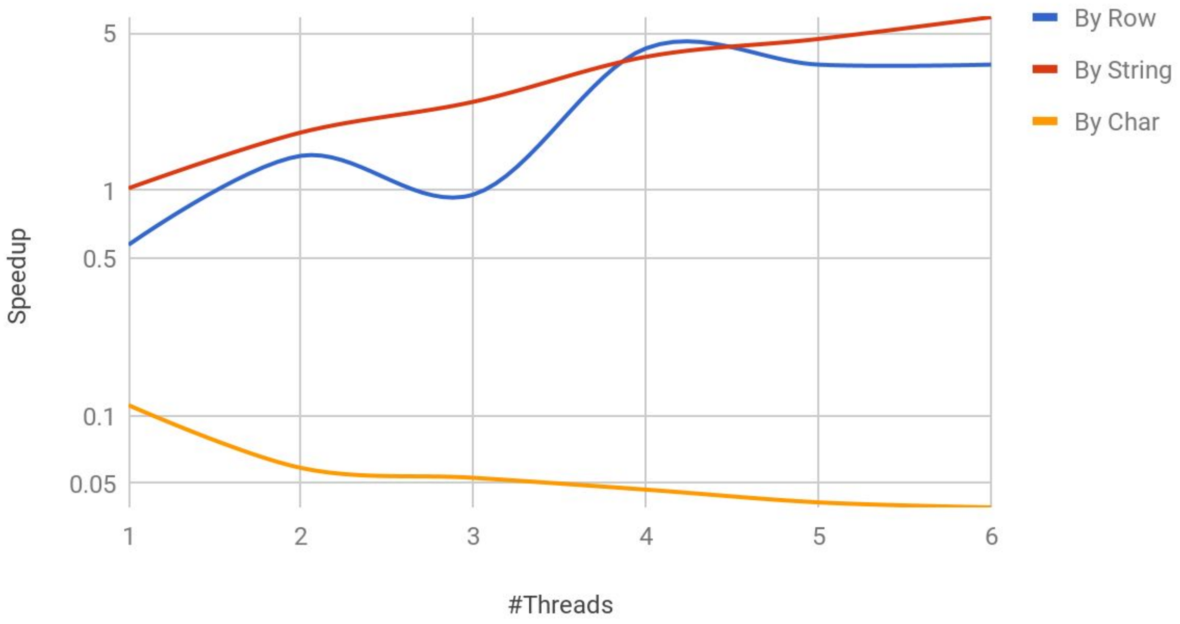


Chart #4



Conclusion

Given the data, for such an application, parallelism is highly recommended using the “by row” method for the OpenMP implementation and the “by string” method for the PThreads implementation. The current application is an example where parallelism can not only be a bad and developer unfriendly solution, but also several times slower than a simple serial one considering tasks such as the “by char” method. Comparing the two Implementations, PThreads performed slightly better in general to its competitor, OpenMp, but yielded catastrophic results when used badly, in contrast with the last one being almost linear to all granularity methods, capable of coping with code imperfections. This comes to show that with great power comes great responsibility.

	Tgreads: 1				Threads: 6							
	Serial				OpenMP				Pthreads			
m	100		1000	10000	100		1000	10000	100		1000	10000
n	100		1000	10000	100		1000	10000	100		1000	10000
l	10	100	1000	10000	10	100	1000	10000	10	100	1000	10000
#comparison	1.00E+05	1.00E+08	1.00E+09	1.00E+12	1.00E+05	1.00E+08	1.00E+09	1.00E+12	1.00E+05	1.00E+08	1.00E+09	1.00E+12
Serial	0	601	5967	593132								
By Row					0	101	1002	98916	0	98	974	96947
By String					0	119	1040	100383	1	159	992	98416
By Char					1	273	1831	176914	15	14677	151877	149619423
	m = n = l = 1000						m = n = l = 200					
	OpenMP						Pthreads					
#threads	1	2	3	4	5	6	1	2	3	4	5	6
Serial	5797	5797	5797	5797	5797	5797	47	47	47	47	47	47
By Row	5788	2936	1962	1468	1177	1010	82	33	49	11	13	13
By String	5810	2974	1985	1490	1191	1020	46	26	19	12	10	8
By Char	9909	5060	3373	2525	2023	1807	422	802	887	1001	1140	1200