[Homework Assignment 3]

Searching for Order among Chaos Using Threads

Submitted by:
Shubham Jain
sjain39@uic.edu

Problem Statement:

For this assignment I wrote a program using PTHREADS to analyze a data file looking for patterns. Each thread analyzes a different section of the file in parallel, recording their findings in global variables. I used pthread mutex locks to synchronize the read-write problem on these global variables between threads. A separate thread reads from and prints the latest results to the screen as they arrive.

How to run the program:

$: make

$: orderSearcher [inputDataFile] [nThreads]

inputDataFile: {any file to be analyzed}

nThreads: range{0, 5000}

Note: For nThreads = 0, the program finds the pattern in the data without using threads.

Evaluation Criteria:

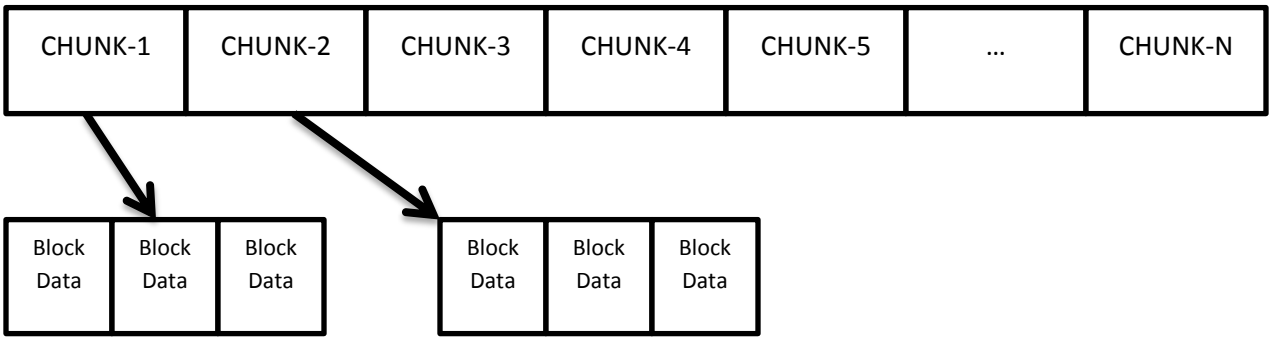
= (10% of Range) + (15% of Max Absolute Change) + (20% of Sum Absolute Change) + (25% of Standard Deviation) + (30% of Standard Deviation Change)

Note: The weights defined here for different criteria's are based on the study of different statistical measures in well-known research papers. I explored few of these research papers from internet and came to define the evaluation criteria as mentioned above.

How data broken among Threads:

Program simply split the files into equal sized chunks, and each thread works on single chunk. For a particular chunk, thread tries to find a pattern examining 80 blocks of data at a time. Separate threads compete with each other in parallel to find the best data of block in their respective chunk.

INPUT FILE



Note: Block size = 80

Boundary Condition: The pattern matching at boundary of chunks is taken care by running the threads till the chunksize[index-1] except the last chunk to avoid buffer overflow.

Synchronization:

Synchronization of global data between threads is maintained by use of pthread_mutex_lock and pthread_mutex_unlock that enabled signaling between threads for wait and signal.

Analyzing Results:

I have my personal Amazon EC2 instance to work on school projects. Hence, all the results are based on tests performed on my EC2 virtual machine instance.

System Specifications

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 23
model name      : Intel(R) Xeon(R) CPU          E5430  @ 2.66GHz
stepping        : 6
microcode       : 0x60b
cpu MHz         : 2660.026
cache size      : 6144 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu de tsc msr pae cx8 sep cmov pat clflush mmx fxsr sse sse2 ss ht syscall nx lm
constant_tsc up rep_good nopl pni ssse3 cx16 sse4_1 hypervisor lahf_lm dtherm
bogomips        : 5320.05
clflush size    : 64
cache_alignment : 64
address sizes   : 38 bits physical, 48 bits virtual
```

As the specification specifies, it has only **1 processor (single core)**. Hence, as expected, the effect on multi-threading was better than not having any threads in the program. However, I was not able to observe any significant improvement in performance after creation of certain number of threads. There was only a slight improvement with each added thread.

Data Observations:

I have tested my program on all the given files; however I am attaching results for all raw files only to limit the report length within limit.

File	nThreads	Time Taken	Best Location	Best Criteria Value
light_drops.raw	1	0.94	11623	48.67
Size = 49152	10	0.55	11623	48.67
	100	0.58	11623	48.67
diver_resize.raw	1	1.61	50844	39.77
Size = 82944	10	1.14	50844	39.77
	100	1	50844	39.77
	1000	0.85	50844	39.77
Chevalier_473.raw	1	50.18	2031843	32.93
Size = 2105280	10	25.88	2031843	32.93
	100	23.01	2031843	32.93
	1000	22.55	2031843	32.93
GrandeJatte.raw	1	30.15	686190	40.86
Size = 1289418	10	16.27	686190	40.86
	100	14.02	686190	40.86
	1000	13.59	686190	40.86
LUG_Newbies.raw	1	27.81	1116	0
Size = 1196640	10	15.43	1116	0
	100	13.64	1116	0
	1000	13.43	1116	0
Esher.raw	1	44.68	1823634	29.72
Size = 1825740	10	23.02	1823634	29.72
	100	20.33	1823634	29.72
	1000	19.34	1823634	29.72
diver.raw	1	230.11	3576768	0
Size = 9437184	10	115.75	3576768	0
	100	103.56	3576768	0
	1000	102.46	3576768	0

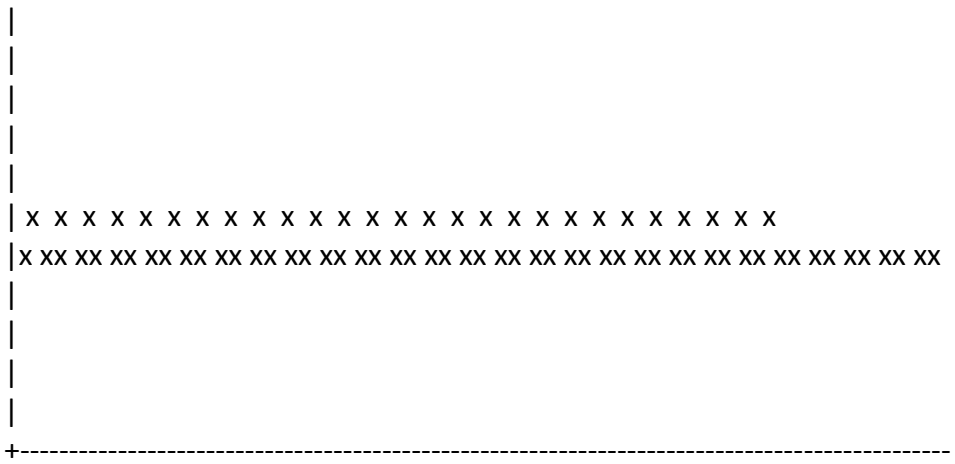
VRUPL_Logo.raw	1	363.11	0	0
Size = 15222627	10	181.5	0	0
	100	161.67	0	0
	1000	159.91	0	0
JohnBell.raw	1	450.19	7523014	37.89
Size = 18874368	10	228.69	7523014	37.89
	100	204.95	7523014	37.89
	1000	203.51	7523014	37.89

The observations made above clearly shows how even on a single core processor, the multithreading improves the performance and decreases the execution time by almost half. As expected on a single core system, there weren't any significant improvements with more creation of threads, since the CPU usage is almost 100% and the threads are scheduled efficiently then.

Sample Outputs:

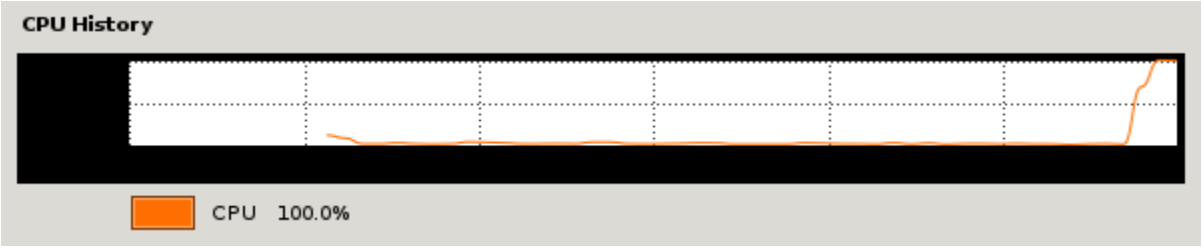
Test File: *Chevalier_473.raw*

Best candidate found is as follows:
Best Location = 686190
Criteria Value = 40.86



Number of Threads = 1

Number of Threads = 10; Time Execution Reduces by half



Number of Threads = 100; Time execution decreases very marginally thereafter

Conclusion:

Depicted the use of multithreading on a single cpu without hyperthreading and came to the following conclusion.

You can consider using multithreading on a single CPU:

- If you use network resources
- If you do high-intensive IO operations
- If you pull data from a database
- If you exploit other stuff with possible delays

When you should not use multithreading on a single CPU:

- High-intensive operations which do almost 100% CPU usage
- If you are not sure how to use threads and synchronization
- If your application cannot be divided into several parallel processes

Things Learnt:

A thread is essentially a single sequence of instructions. A process (a running instance of a program) consists of one or more threads. A processor core is a unit capable of processing a sequence of instructions. So there is a direct relation between threads and cores. For the OS, a thread is a unit of workload which can be scheduled to execute on a single core.

In order to make a single core able to run multiple threads, a form of time-division multiplexing was used. To simplify things a bit: the OS sets up a timer which interrupts the system at a fixed interval. A single interval is known as a time slice. Every time this interrupt occurs, the OS runs the scheduling routine, which picks the next thread that is due to be executed. The context of the core is then switched from the currently running thread to the new thread, and execution continues.