Edgar Granados

CpE 142- Section 01

May 1, 2019

Dr. Faroughi

Project 6: Performance of Processing Loop Processing in Parallel

Overview: The main objective of this project was to perform a series of runs from the provided programs; and compare the performance of each program as the number of executing threads was increased. In our analysis, we wanted to compare the program running with 1, 2, 4, and 10 threads and how the total executing time was affected as we increased the amount of threads.

1. ***NOTE*: Because systems these days are very fast; the problem size of the program was increased. Specifically, the number of rows was increased from 128 to 256 and the number of columns from 100k to 1M.**

2. The following data was collected from each program and used to create the performance charts.

Program	**1 Thread Program**	**2 Thread Program**	
Threads	**1**	**1**	**2**
Start Time	1556867035	1556867159	1556867159
End Time	1556867038	1556867160	1556867160
Execution Time(ms)	2564	1445	1453
Rows Processed	256	126	130
Avg. Time Spent on Each Row(ms)	1289	736	730

4 Thread Program			
1	**2**	**3**	**4**
1556867307	1556867307	1556867307	1556867307
1556867308	1556867308	1556867308	1556867308
928	930	925	926
62	66	62	66
494	464	495	459

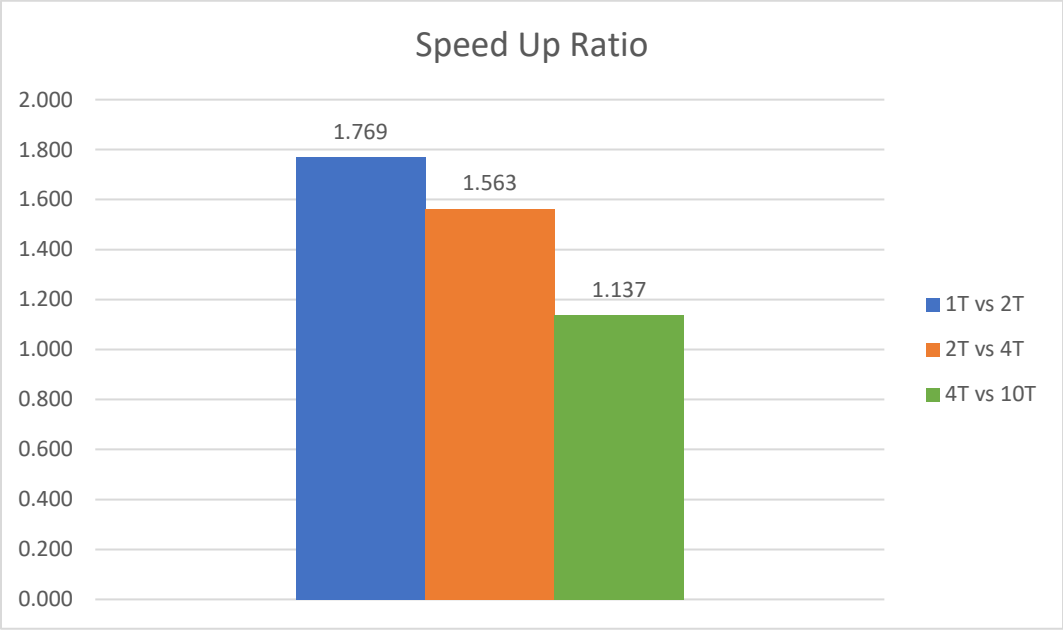
10 Thread Program									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1556867538	1556867538	1556867538	1556867538	1556867538	1556867538	1556867538	1556867538	1556867538	1556867538
1556867539	1556867539	1556867539	1556867539	1556867539	1556867539	1556867539	1556867539	1556867539	1556867539
812	822	820	814	824	808	818	800	821	817
26	27	37	25	20	25	23	23	26	24
426	437	319	444	460	473	452	421	439	446

Program	1 Thread Program	2 Thread Program	4 Thread Program	10 Thread Program
Overall Program Execution Time (ms)	2564	1449	927.25	815.6
Overall Avg. Time Spent on Each Row (ms)	1289	733	478	431.7

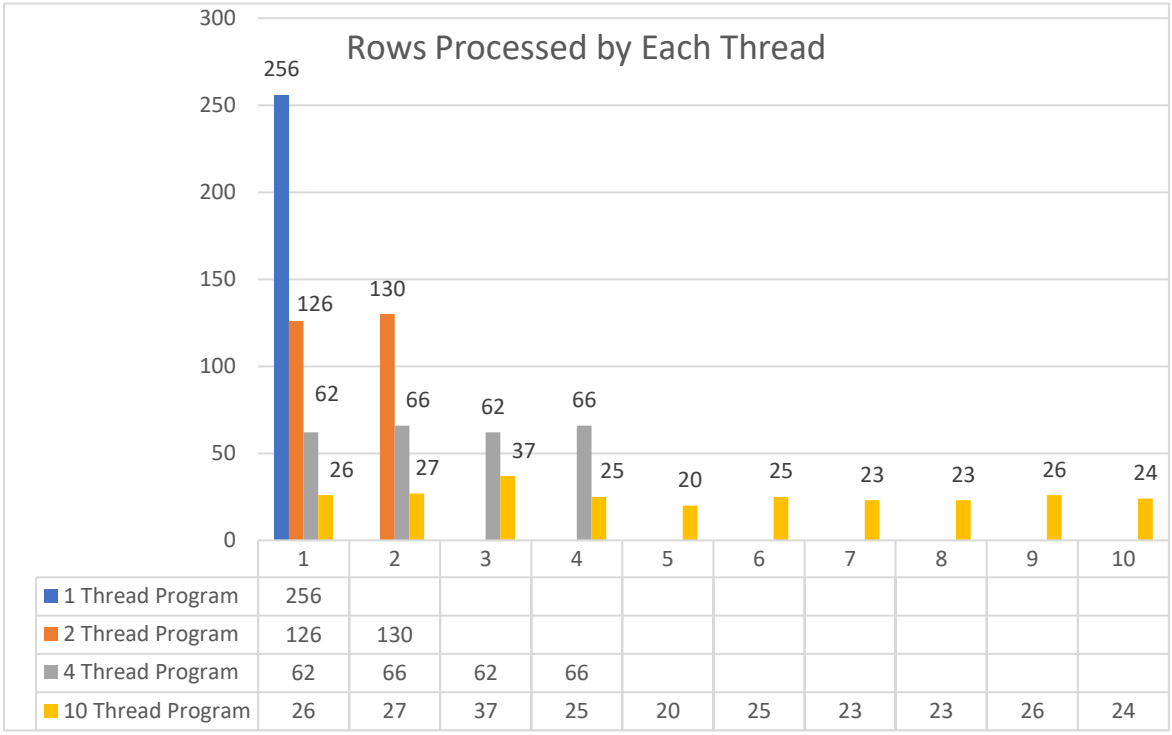
Speedup Ratio	
1T vs 2T	1.769
1T vs 2T	1.563
4T vs 10T	1.137

Overall Avg. Time Spent on Each Row (ms)	
1 Thread Program	1289
2 Thread Program	733
4 Thread Program	478
10 Thread Program	431.7

3. The following chart shows the speedup data for 1T vs 2T Speedup, 2T vs 4T Speedup, and 4T vs 10T speedup. Each speedup ratio was calculated by dividing the total execution time of one program over the other program.

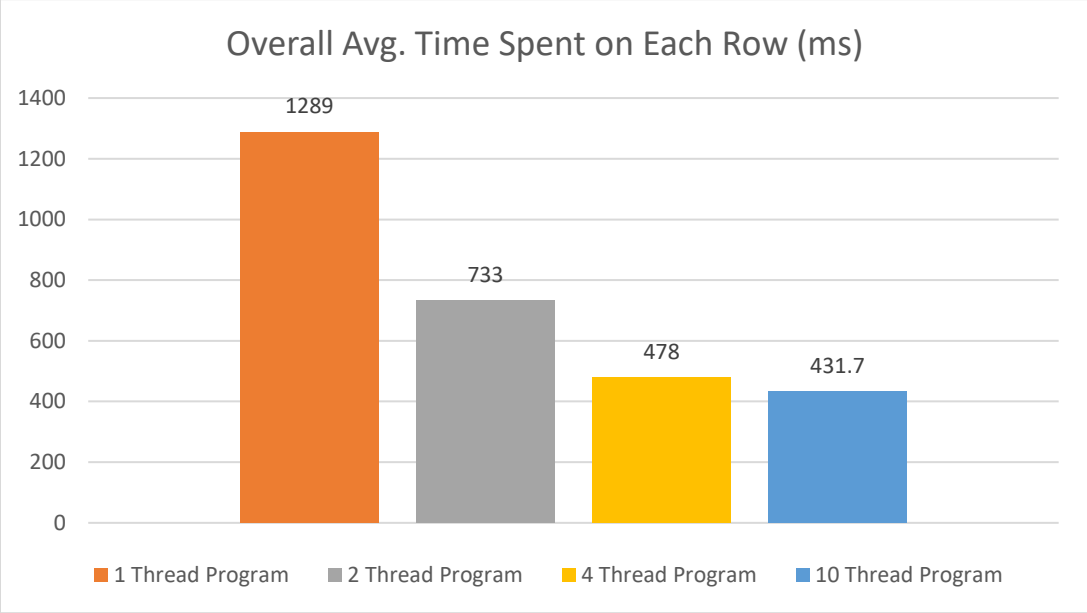


4. The next chart shows the number of rows each thread processed for each program. One important thing to note is that not every thread processed the same number of rows, which may indicate that some threads got "lucky" were indexed rows that required less processing. However, the total amount of rows processed for every program was 256; as expected.



	1	2	3	4	5	6	7	8	9	10
1 Thread Program	256									
2 Thread Program	126	130								
4 Thread Program	62	66	62	66						
10 Thread Program	26	27	37	25	20	25	23	23	26	24

5. Finally, the following chart demonstrates the amount of time each program spent processing each row in the matrix. The chart demonstrates that as the amount of threads in the program increased, the overall time spent on each row in the matrix began to decrease. One possible explanation for this observation, is that since you have more threads executing in parallel, the rows that require more processing could be executing together, thus lowering the amount of time being spent on each row.



6. **Analysis:** From Chart 1, we see that the plot that had the greatest speedup ratio was the **1T vs 2T** plot. From there forward, the speed up ratio began decreasing for the **2T** vs **4T** and **4T** vs **10T** plots. What can be inferred from Chart 1, is that after having 4 Threads executing the program, there is really very little benefit to the performance. In other words, at 4T, we have reached an upper limit of speedup performance. Any further parallelization (adding more threads) would give very little gain in performance. In terms of Amdahl's Law, this is known as the law of diminishing returns.

   Next, for Chart 2 we see the number of rows each thread processes. An important observation is that not all threads process the same number of rows. A possible explanation for this is that some threads are indexed at rows that require very little arithmetic processing (such as row 0, 1, 2 etc.); therefore, these threads are able to finish faster and fetch another row to process. Nonetheless, for the most part, the number of rows processed are evenly distributed between each thread.

   Finally, Chart 3 displays the overall amount of time each program spent on processing a row. Again, we see a decreasing trend as in Chart 1. That is, as the amount of threads in the program increased, the overall time spent processing each row decreased. However. the gap in speed between the 4 Threaded Program and 10 Threaded Program is very small. Which shows that if we increase the amount of threads after 4, the gain in performance for row processing time becomes very small. Once again, demonstrating Amdahl's law of diminishing returns.

7. The Following Source Code was written to create a 10 Thread Program that summed the up the columns of each row.

SOURCE CODE

```
//Created By: Edgar Granados
//Project 6: Looping parallelism and cache traffic
//Dr. Faroughi

/*On Simics compile with "gcc thread.c -pthread"
on CygWin, -pthread option is not required */

#include <pthread.h>
#include <stdio.h>
#include <sys/timeb.h>



#define NUMBER_OF_ROWS 256 //2^8

#define NUMBER_OF_COLS 1000000
#define NUMBER_OF_ADDITIONAL_THREADS 9



//shared arrays named "array" and "sum",  variable "count", and lock location named
"lock_var"
int array[NUMBER_OF_ROWS][NUMBER_OF_COLS];
int rowsProcessed[NUMBER_OF_ADDITIONAL_THREADS+1];
double sum[NUMBER_OF_ROWS];

long threadTime[NUMBER_OF_ADDITIONAL_THREADS+1];//Used to store the time it took for thread
to complete in ms
int avgRowTime[NUMBER_OF_ADDITIONAL_THREADS+1];
double timeComplete[NUMBER_OF_ADDITIONAL_THREADS+1];//Used to store the final time of a
thread.
int count = 0;  //define a shared count, initialized to zero
pthread_mutex_t lock_var;  //define a shared lock variable (a system level variable)
long gRefTime; //For timing

//Timing functions
long GetMilliSecondTime(struct timeb timeBuf);
long GetCurrentTime(void);
void SetTime(void);
long GetTime(void);
int FetchAndIncrement()
{
       int t;
       pthread_mutex_lock(&lock_var);    //lock
              t = count;
              count = count + 1;
       pthread_mutex_unlock(&lock_var); //unlock
       return(t);
}
void *FindRowSum(void *arg)
{
       int *pid = (int *) arg;
       int index, j, counter;
       long avgTime, Time;
       avgTime = 0;
       counter = 0;
       SetTime();
       index = FetchAndIncrement();

       while(index < NUMBER_OF_ROWS) {
```

```
		for (j=0; j< NUMBER_OF_COLS; j++)
		array[index][j] = index; //assume array data are read here

		sum[index] = 0; //start computations (assume a complex one)
		for (j=0; j<NUMBER_OF_COLS; j++)
		{
			sum[index] = sum[index] + array[index][j];
		}
		Time = GetTime();
		printf("pid = %d Sum[%d] = %f Took Time = %ld ms to complete row\n", *pid,
index, sum[index], Time);
		avgTime = avgTime + Time;
		counter++;
		index = FetchAndIncrement();
	}
	threadTime[*pid] = GetTime();
	rowsProcessed[*pid] = counter;
	avgRowTime[*pid] =  avgTime / counter;
	timeComplete[*pid] = (double) time(NULL);
	return NULL;
}
int main (void)
{
	pthread_t threadID[NUMBER_OF_ADDITIONAL_THREADS];
	void *exit_status;
	int i, pid[NUMBER_OF_ADDITIONAL_THREADS+1];


	for(i=0; i<NUMBER_OF_ADDITIONAL_THREADS; i++) { //create 1 aditional threads to run
		pid[i] = i;
		printf("10T: PID %d started: time = %f\n", pid[i], (double) time(NULL));
		SetTime();
		pthread_create(&threadID[i], NULL, FindRowSum, &pid[i]);  //&i would also work
but print pid info above will be incorrect
	}

	pid[9] = 9;
	printf("10T: PID %d started: time = %f\n", pid[9], (double) time(NULL));
	SetTime();
	FindRowSum(&pid[9]); //main is the 2th thread
	for(i=0; i<NUMBER_OF_ADDITIONAL_THREADS; i++)
		pthread_join(threadID[i], &exit_status);
	for( i = 0; i < (1+NUMBER_OF_ADDITIONAL_THREADS); i++)
	{
		printf("10T: PID %d ended: time = %f\n",pid[i], timeComplete[i]);
		printf("10T: PID %d took: time = %ld ms to complete\n", pid[i], threadTime[i]);
		printf("10T: PID %d Processed: %d rows and spent an Average Time of : %ld ms
per row\n",pid[i], rowsProcessed[i], avgRowTime[i]);
    }
	return 0;
}
long GetMilliSecondTime(struct timeb timeBuf){
	long mliScndTime;
	mliScndTime = timeBuf.time;
	mliScndTime *= 1000;
	mliScndTime += timeBuf.millitm;
	return mliScndTime;
}

long GetCurrentTime(void){
	long crntTime=0;
	struct timeb timeBuf;
	ftime(&timeBuf);
	crntTime = GetMilliSecondTime(timeBuf);
	return crntTime;
```

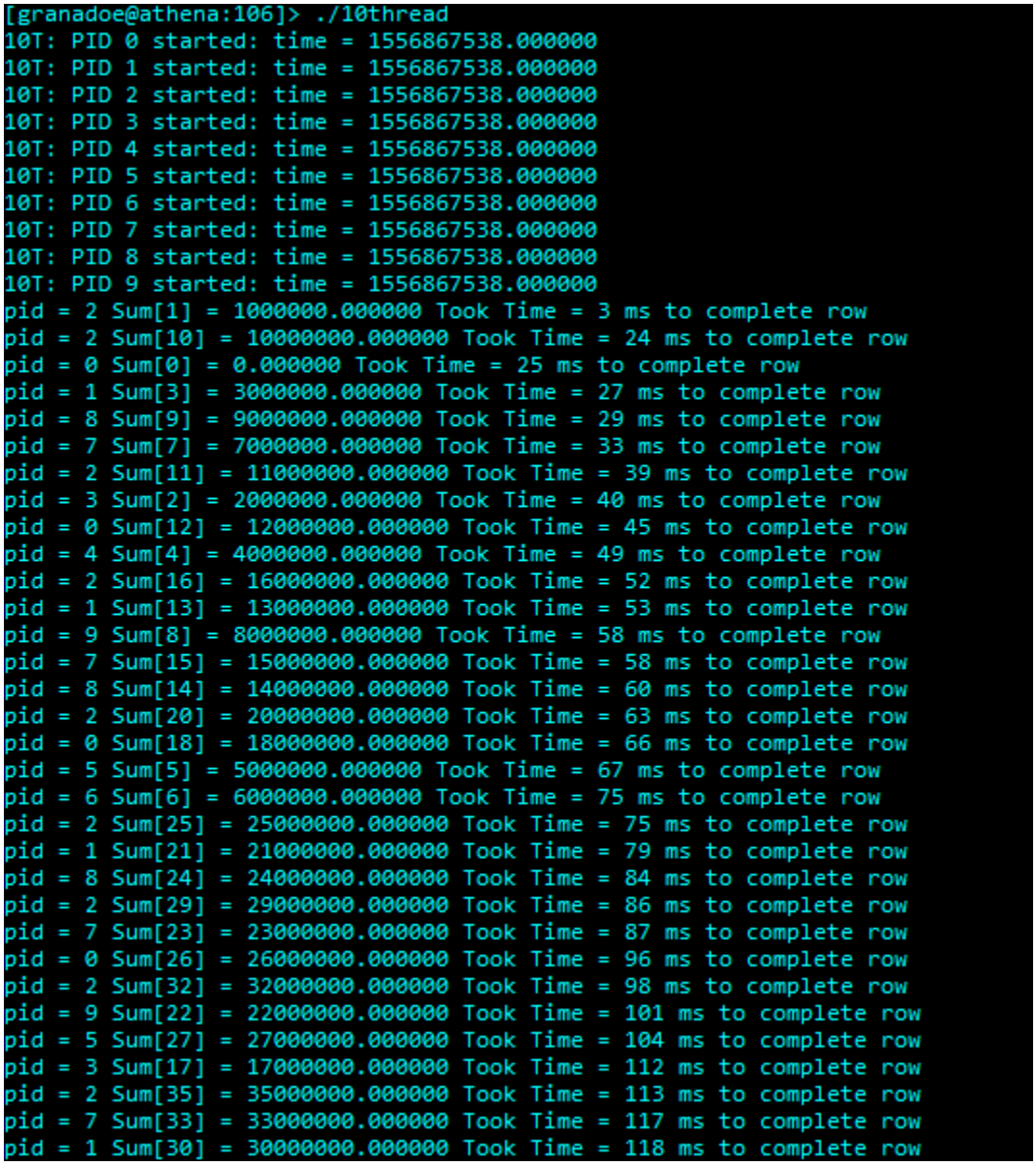
```
}

void SetTime(void){
	gRefTime = GetCurrentTime();
}

long GetTime(void){
	long crntTime = GetCurrentTime();
	return (crntTime - gRefTime);
}
```

OUTPUT

The following figures show snippets of the program after being executed on the Athena Environment.



Snippet 1: Beginning of Program after Executing

```
10T: PID 0 ended: time = 1556867539.000000
10T: PID 0 took: time = 812 ms to complete
10T: PID 0 Processed: 26 rows and spent an Average Time of : 426 ms per row
10T: PID 1 ended: time = 1556867539.000000
10T: PID 1 took: time = 822 ms to complete
10T: PID 1 Processed: 27 rows and spent an Average Time of : 437 ms per row
10T: PID 2 ended: time = 1556867539.000000
10T: PID 2 took: time = 820 ms to complete
10T: PID 2 Processed: 37 rows and spent an Average Time of : 319 ms per row
10T: PID 3 ended: time = 1556867539.000000
10T: PID 3 took: time = 814 ms to complete
10T: PID 3 Processed: 25 rows and spent an Average Time of : 444 ms per row
10T: PID 4 ended: time = 1556867539.000000
10T: PID 4 took: time = 824 ms to complete
10T: PID 4 Processed: 20 rows and spent an Average Time of : 460 ms per row
10T: PID 5 ended: time = 1556867539.000000
10T: PID 5 took: time = 808 ms to complete
10T: PID 5 Processed: 25 rows and spent an Average Time of : 473 ms per row
10T: PID 6 ended: time = 1556867539.000000
10T: PID 6 took: time = 818 ms to complete
10T: PID 6 Processed: 23 rows and spent an Average Time of : 452 ms per row
10T: PID 7 ended: time = 1556867539.000000
10T: PID 7 took: time = 800 ms to complete
10T: PID 7 Processed: 23 rows and spent an Average Time of : 421 ms per row
10T: PID 8 ended: time = 1556867539.000000
10T: PID 8 took: time = 821 ms to complete
10T: PID 8 Processed: 26 rows and spent an Average Time of : 439 ms per row
10T: PID 9 ended: time = 1556867539.000000
10T: PID 9 took: time = 817 ms to complete
10T: PID 9 Processed: 24 rows and spent an Average Time of : 446 ms per row
[granadoe@athena:107]>
```

Snippet 2: End of Program with the Output Results.