Austin Brown

CPE 434-01

1/28/2021

Lab 6

**Theory:**

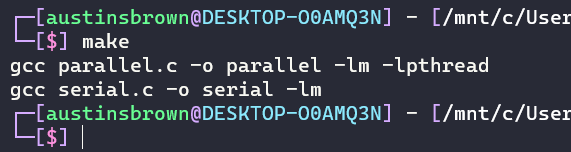
A process is a program that is being executed. A thread is just a segment of a process, or a subprocess. They are much more lightweight than processes are and can be created quicker that processes can. While processes have their own memory space, threads share memory. Although they share a space, they do have their own stack for local storage.

The rectangular method is a way of approximating a definite integral. It involves breaking the area under the curve into rectangles and summing them up. The more rectangles that are used, the more accurate the result will be.

When you are using a mutex, only one thread can access the memory space at a time. They are either locked or unlocked. A semaphore has a wait and signal operation. Instead of locking a memory region, they tell threads to wait or signal them.

**Observations:**

**Compilation**

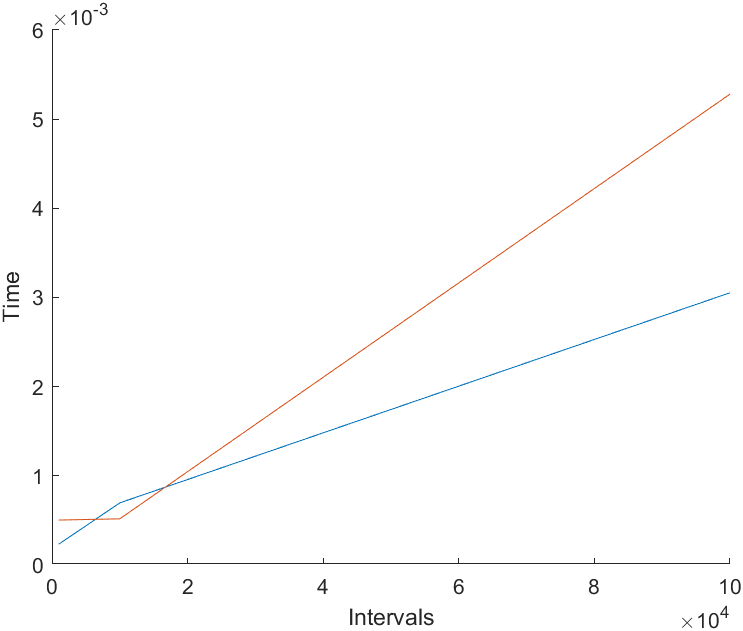
****

|  |  |  |
| --- | --- | --- |
| Intervals | Parallel (2 Threads) | Serial |
| **1000** |  |  |
| **10000** |  |  |
| **100000** |  |  |

|  |  |  |
| --- | --- | --- |
| Intervals | Parallel (4 Threads) | Serial |
| **1000** |  |  |
| **10000** |  |  |
| **100000** |  |  |

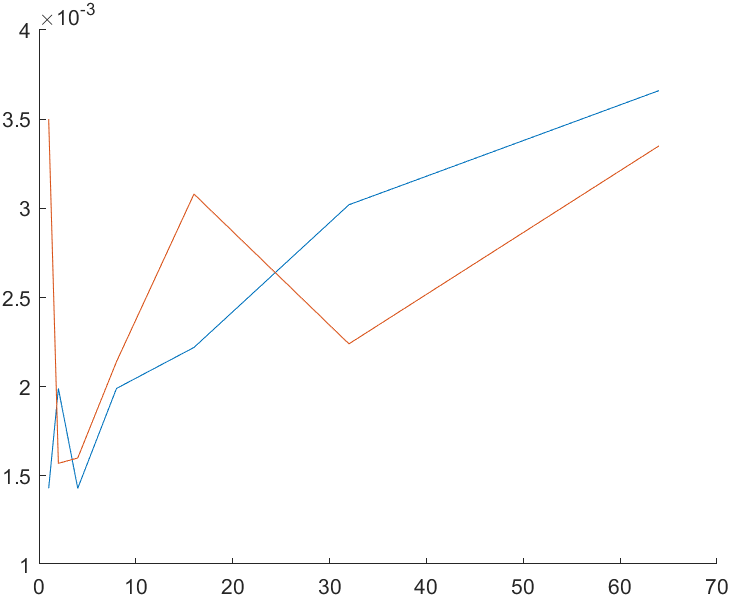
|  |  |  |
| --- | --- | --- |
| Intervals | Parallel (4 Threads) | Serial |
| **1000** |  |  |
| **10000** |  |  |
| **100000** |  |  |

|  |  |  |
| --- | --- | --- |
| Intervals | Parallel (16 Threads) | Serial |
| **1000** |  |  |
| **10000** |  |  |
| **100000** |  |  |



100000 Intervals

|  |  |  |
| --- | --- | --- |
| Number of Threads | Parallel | Serial |
| 1 |  |  |
| 2 |  |  |
| 4 |  |  |
| 8 |  |  |
| 16 |  |  |
| 32 |  |  |
| 64 |  |  |



**Conclusion:**

Threads can offer better performance for certain programs. There are limitations, however. There are only so many threads that you can create without slowing the program down. This is partially due to overhead, but you can also generate race conditions is synchronization is not implemented properly. You can use mutexes or semaphores to help this.

**Appendix:**

**serial.c**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <math.h>  *int* main(*int* argc, *char* \*\*argv)  {      if(argc != 2)      {          printf("Usage: ./b.out intervals\n");          exit(-1);      }  *struct* *timespec* start, stop;      clock\_gettime(CLOCK\_REALTIME, &start); *// start timer*  *int* intervals = atoi(argv[1]);      if(intervals <= 0)      {          printf("Interval needs to be greater than 0.\n");          exit(-1);      }  *double* width = (*double*)1/intervals;  *double* sum = 0;  *double* temp1, temp2;  *int* i = 0;      temp1 = sqrt(1 - pow(i \* width, 2));      for (i = 0; i < intervals; i++)      {          temp2 = sqrt(1 - pow((i+1) \* width, 2));          sum += ((temp1 + temp2) / 2) \* width;          temp1 = temp2;      }      sum \*= 4;      clock\_gettime(CLOCK\_REALTIME, &stop);  *unsigned* *long* *long* totalSeconds = (*long* *long*)(stop.tv\_sec - start.tv\_sec);  *unsigned* *long* totalNanoseconds = stop.tv\_nsec - start.tv\_nsec;      printf("Execution Time: %llu.%.9lu\n", totalSeconds, totalNanoseconds);      printf("Final Calculation: %.12f\n", sum);      return 0;  } |

**integralArgs.h**

|  |
| --- |
| typedef *struct* *integralArgs*  {  *int* threadID;  *int* threadCount;*// total number of threads*  *int* intervalCount;*// total number of intervals*  *int* intervalStart;*// what interval each thread starts at*  *int* intervalsPerThread;  *double* width;  }*integralArgs*; |

**parallel.c**

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <time.h>  #include <stdlib.h>  #include <math.h>  #include "integralArgs.h"  *double* globalSum; *// sum for every thread*  *pthread\_mutex\_t* mutex; *// used to lock access to globalSum*  *void* \*decomp(*void* \*); *// calculate the integral*  *int* main(*int* argc, *char* \*\*argv)  {      if (argc != 3)      {          printf("Usage: ./parallel intervals threads\n");          exit(-1);      }  *struct* *timespec* start, stop;      clock\_gettime(CLOCK\_REALTIME, &start); *// track the execution time*    *integralArgs* \*args;  *int* intervals = atoi(argv[1]);  *int* threads = atoi(argv[2]);      if (intervals <= 0) *// perform error checking*      {          printf("Number of intervals must be greater than 0.\n");          exit(-1);      }      if (threads <= 0)      {          printf("Number of threads must be greater than 0.\n");          exit(-1);      }      if (threads > intervals)      {          printf("You cannot have more intervals than threads.\n");          exit(-1);      }  *double* width = (*double*)1/intervals;  *int* remainder = intervals%threads;  *// certain threads will get extra intervals*  *int* status; *// error checking for pthread\_create*  *int* currentInterval = 0;      globalSum = 0;    *pthread\_t* threadArray[threads];        for (*int* i=0; i<threads; i++)      {          args = malloc(sizeof(*integralArgs*));          args->threadID = i+1;          args->intervalCount = intervals;          args->intervalStart = currentInterval;          args->intervalsPerThread = intervals/threads;   *// intervals per thread*          currentInterval += args->intervalsPerThread;          if (remainder > 0) *// handle the remainder*          {              remainder--; *// use one less interval to calculate*              args->intervalsPerThread++; *// threads will calculate an extra interval*              currentInterval++;          }          args->threadCount = threads;          args->width = width;          status = pthread\_create(&threadArray[i], NULL, decomp, (*void*\*)args); *// create pthread*          if (status)          {              printf("Error in creating the threads: %d\n", i+1);              return -1;          }      }        for (*int* i = 0; i < args->threadCount; i++)      {          pthread\_join(threadArray[i], NULL);      }    *double* finalCalc = 4\*globalSum; *// get final answer*        clock\_gettime(CLOCK\_REALTIME, &stop);  *unsigned* *long* *long* totalSeconds = (*long* *long*)(stop.tv\_sec - start.tv\_sec);  *unsigned* *long* totalNanoseconds = stop.tv\_nsec - start.tv\_nsec;      printf("Total execution time: %llu.%.9lu\n", totalSeconds, totalNanoseconds);      printf("Final calculation: %.12f\n", finalCalc);        free(args);      return 0;  }  *void* \*decomp(*void* \*argument)  {  *integralArgs* \*threadArgs = (*integralArgs*\*)argument;  *double* sum = 0;  *int* myInterval = threadArgs->intervalStart; *// keep up with what interval you are on*    *double* temp1, temp2;      temp1 = sqrt(1-pow(myInterval\*threadArgs->width, 2)); *// use trapezoid theorem to solve the integral*      for (*int* i=0; i < threadArgs->intervalsPerThread; i++)      {          temp2 = sqrt(1-pow((myInterval+1)\*threadArgs->width, 2));          sum += ((temp1+temp2)/2)\*threadArgs->width;          temp1 = temp2;          myInterval++;      }        pthread\_mutex\_lock(&mutex);     *// get access to globalSum*      globalSum += sum;      pthread\_mutex\_unlock(&mutex);   *// release exclusive access to globalSum*  } |