**KURUKULASOORIYA KVR** **200332X**

**CS 4553 SCIENTIFIC COMPUTING**

**ASSIGNMENT I**

---

## PART A

### 1.Use the Monte Carlo method to evaluate the value of constant PI to the precision of 5/ 10/15/20 decimal places.

The Monte Carlo method is used to estimate the value of π by simulating random points within a unit square and checking whether they fall inside a quarter circle. The formula on 4 into the ratio of points inside the circle to total points approximates the value of π. In this assignment the estimation of π is done up to 3, 5, 10, 15, and 20 decimal places using the following parallelization strategies.

#### I) Pthreads based parallelization

montecarlo_pthreads.cpp

~/vish/montecarlo_pthreads.cpp

```cpp
#include
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <chrono>
#include <pthread.h>

using namespace std;
using namespace std::chrono;

const long double TRUE_PI = 3.14159265358979323846264338327950288 4L;
const int NUM_THREADS = 8;

struct ThreadData {
    long long start, end, insideCircle;
};

void* monteCarloThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    long long inside = 0;

    for (long long i = data->start; i < data->end; i++) {
        long double x = (long double)rand() / RAND_MAX;
        long double y = (long double)rand() / RAND_MAX;
        if (x * x + y * y <= 1) inside++;
    }

    data->insideCircle = inside;
    pthread_exit(nullptr);
}
```

```
32  long double monteCarloPi(long long samples, long long& insideCircle) {
33      pthread_t threads[NUM_THREADS];
34      ThreadData threadData[NUM_THREADS];
35      insideCircle = 0;
36
37      long long chunkSize = samples / NUM_THREADS;
38
39      for (int i = 0; i < NUM_THREADS; i++) {
40          threadData[i].start = i * chunkSize;
41          threadData[i].end = (i == NUM_THREADS - 1) ? samples : (i + 1) * chunkSize;
42          pthread_create(&threads[i], nullptr, monteCarloThread, &threadData[i]);
43      }
44
45      for (int i = 0; i < NUM_THREADS; i++) {
46          pthread_join(threads[i], nullptr);
47          insideCircle += threadData[i].insideCircle;
48      }
49
50      return (4.0L * insideCircle) / samples;
51  }
52
53  long long getSamplesForPrecision(long double requiredPrecision) {
54      long long samples = 1000;
55      long long insideCircle;
56      long double piEstimate, error;
57
58      while (true) {
59          piEstimate = monteCarloPi(samples, insideCircle);
60          error = fabsl(piEstimate - TRUE_PI);
61
62          if (error <= requiredPrecision) {
63              return samples;
64          }
65          samples *= 2;
66      }
67  }
68
```

- **monteCarloThread(void* arg)** – This function performs the Monte Carlo simulation for a given range of samples by genrating random values for x and y and then counting how many of them fall inside the unit circle.

- **monteCarloPi(long long samples, long long& insideCircle)** – This function creates and distributes the workload among multiple threads and then collects results from each thread to update the global variables to estimate the value of π.

- **getSamplesForPrecision(long double requiredPrecision)** – This function estimates the value of π, calculate the error and checks whether it is up to the precision specified.

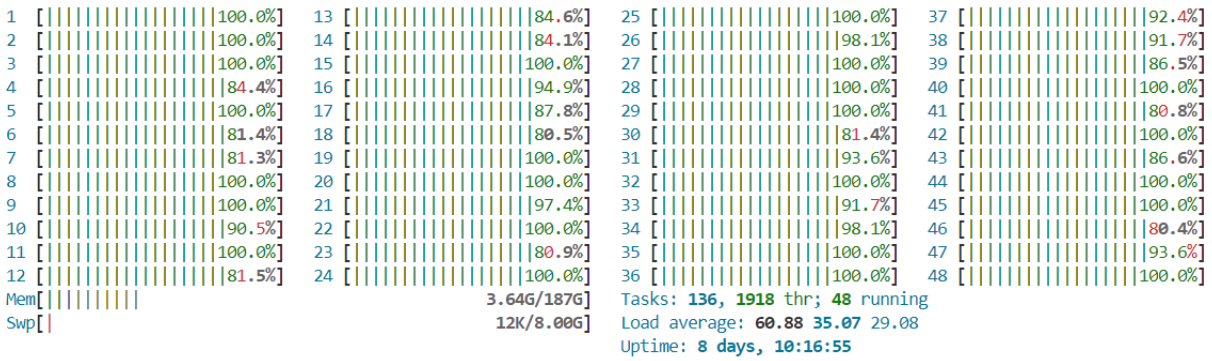
```cpp
int main() {
    srand(time(0));
    auto start = high_resolution_clock::now();

    long double precisions[] = {1e-5L, 1e-10L, 1e-15L, 1e-20L};
    int precisionLevels[] = {5, 10, 15, 20};

    for (int i = 0; i < 4; i++) {
        long long requiredSamples = getSamplesForPrecision(precisions[i]);
        long long insideCircle;
        long double estimatedPi = monteCarloPi(requiredSamples, insideCircle);

        cout.precision(precisionLevels[i] + 2);
        cout << "Precision: " << precisionLevels[i] << " decimal places\n";
        cout << "Estimated Pi: " << estimatedPi << "\n";
        cout << "Required Samples: " << requiredSamples << "\n\n";
    }

    auto end = high_resolution_clock::now();
    double duration = duration_cast<milliseconds>(end - start).count() / 1000.0;
    cout << "Total Execution Time: " << duration << " seconds" << endl;

    return 0;
}
```

- **main() function** initializes execution, iterates over different precision levels, calls the Monte Carlo estimation function, and records execution time before displaying the results.

**Implementation details:**

**i) Task decomposition:**
A **static task partitioning strategy** is used where workload is divided among NUM_THREADS (124 threads)

**ii) Hardware Speculation:**
RAM: 187G, 48 core CPU, 8GB swap memory




**iii) Execution time measurement:**
The program uses C++'s **high_resolution_clock from <chrono>** to measure execution time.

**iv) Random number generation:**
This program uses C standard library function **rand()** is used to generate pseudo-random numbers for x and y coordinates and srand(time(0)) is used to seed the random number generator to ensure different runs produce varying results. Once produced, the generated numbers are normalized (rand() / RAND_MAX) to fall within the range [0,1].

**Observation Table: pthreads method ( 124 threads )**

| Precision Level | No of Samples Used | Time Taken (s) |
|---|---|---|
| **3 decimals** | 160000 | 0.004 |
| **5 decimals** | 2621440000 | 2.784 |
| **10 decimals** | Not Practical to increase sample size up to given precision | NA |
| **15 decimals** | Not Practical to increase sample size up to given precision | NA |
| **20 decimals** | Not Practical to increase sample size up to given precision | NA |

**II) OpenMP based parallelization**

montecarlo_openmp.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <chrono>
#include <random>
#include <omp.h>

using namespace std;
using namespace std::chrono;

const long double TRUE_PI = 3.14159265358979323846264338327950288 4L;
const int NUM_THREADS = 8;

long double monteCarloPi(long long samples, long long& insideCircle) {
    insideCircle = 0;

    #pragma omp parallel num_threads(NUM_THREADS)
    {
        random_device rd;
        mt19937 gen(rd() + omp_get_thread_num());
        uniform_real_distribution<long double> dist(0.0L, 1.0L);

        long long localInside = 0;

        #pragma omp for
        for (long long i = 0; i < samples; i++) {
            long double x = dist(gen);
            long double y = dist(gen);
            if (x * x + y * y <= 1) localInside++;
        }

        #pragma omp atomic
        insideCircle += localInside;
    }
    return (4.0L * insideCircle) / samples;
}

```

montecarlo_openmp.cpp

```cpp
long long getSamplesForPrecision(long double requiredPrecision) {
    long long samples = 100000;
    long long insideCircle;
    long double piEstimate, error;

    while (true) {
        piEstimate = monteCarloPi(samples, insideCircle);
        error = fabsl(piEstimate - TRUE_PI);

        if (error <= requiredPrecision) {
            return samples;
        }
        samples *= 2;
    }
}

int main() {
    auto start = high_resolution_clock::now();

    long double precisions[] = {1e-5L, 1e-10L, 1e-15L, 1e-20L};
    int precisionLevels[] = {5, 10, 15, 20};

    for (int i = 0; i < 4; i++) {
        long long requiredSamples = getSamplesForPrecision(precisions[i]);
        long long insideCircle;
        long double estimatedPi = monteCarloPi(requiredSamples, insideCircle);

        cout.precision(precisionLevels[i] + 2);
        cout << "Precision: " << precisionLevels[i] << " decimal places\n";
        cout << "Estimated Pi: " << estimatedPi << "\n";
        cout << "Required Samples: " << requiredSamples << "\n\n";
    }

    auto end = high_resolution_clock::now();
    double duration = duration_cast<milliseconds>(end - start).count() / 1000.0;
    cout << "Total Execution Time: " << duration << " seconds" << endl;

    return 0;
}
```

- **monteCarloPi(long long samples, long long& insideCircle)** – This function creates random points (x, y) within the unit square [0,1] × [0,1] and then counts the number of points that are within the unit circle and calculates the final estimate of π. The task is parallelized with OpenMP to divide computations between multiple threads.

- **getSamplesForPrecision(long double requiredPrecision)** – This function determines the minimum number of Monte Carlo samples required to achieve a specified precision by calculating the estimated value of pi/ absolute error, and doubles the sample count until the error is within the desired precision threshold.

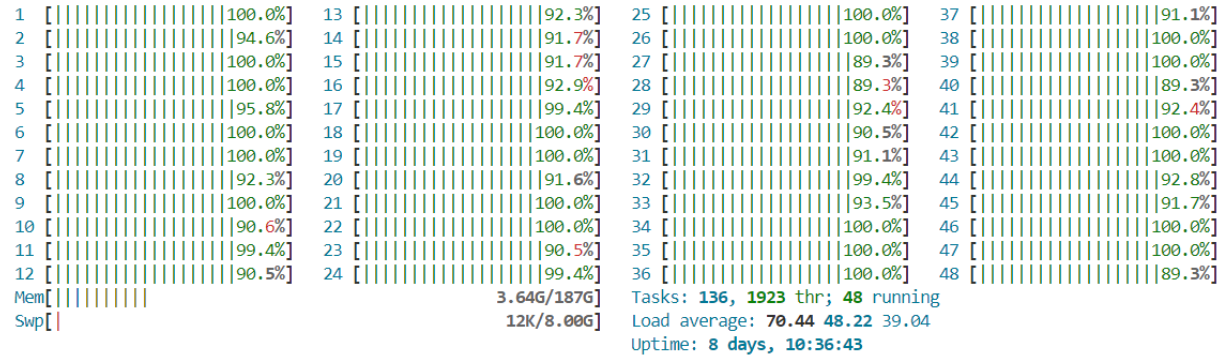
- **main() function** initializes execution, iterates over different precision levels, calls the Monte Carlo estimation function, and records execution time before displaying the results.

**Implementation details:**

**i) Task decomposition:**
This program is parallelized using OpenMP, where each thread is assigned dynamically.

*#pragma omp parallel num_threads(NUM_THREADS)* ensures that NUM_THREADS (set to 124) are used for parallel execution.Then the *#pragma omp* for directive automatically divides the loop iterations among the threads dynamically. The final result is accumulated using *#pragma omp atomic*, ensuring that updates to insideCircle are done without race conditions.

**ii) Hardware Speculation:** *(Same as pthreads method)*
RAM: 187G, 48 core CPU, 8GB swap memory




**iii) Execution time measurement:** *(Same as pthreads method)*
The program uses C++'s **high_resolution_clock from <chrono>** to measure execution time.

**iv) Random number generation:**
rand() is used to generate pseudo-random numbers.

**Observation table: OpenMP method**

| Precision Level | No of Samples Used | Time Taken (in seconds) |
|---|---|---|
| **3 decimals** | 1139062 | 0.035 |
| **5 decimals** | 1683409627 | 7.623 |
| **10 decimals** | Not Practical to increase sample size up to given precision | NA |
| **15 decimals** | Not Practical to increase sample size up to given precision | NA |
| **20 decimals** | Not Practical to increase sample size up to given precision | NA |

**Generally, OpenMP should take less execution time than pthreads. However, since the machine used is heavily running Kubernetes and other programs which are also utilizing the CPU cores, the execution time increases significantly.**

**Theoretically, on a processor dedicated solely to running this program only, OpenMP should outperform the pthreads method.**

### III) CUDA based parallelization

monte_carlo_cuda.cpp

```cpp
#include <cuda_runtime.h>
#include <curand_kernel.h>
#include <iostream>
#include <chrono>
#include <iomanip>

// CUDA kernel for initializing random number generators
__global__ void init_rand_kernel(curandState *state, unsigned long long seed) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    curand_init(seed, idx, 0, &state[idx]);
}

// CUDA kernel for Monte Carlo PI estimation
__global__ void monte_carlo_kernel(curandState *state, unsigned long long *points_inside,
                                   unsigned long long points_per_thread) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    curandState localState = state[idx];
    unsigned long long local_count = 0;

    for (unsigned long long i = 0; i < points_per_thread; i++) {
        float x = 2.0f * curand_uniform(&localState) - 1.0f;
        float y = 2.0f * curand_uniform(&localState) - 1.0f;
        if (x*x + y*y <= 1.0f) {
            local_count++;
        }
    }

    atomicAdd(points_inside, local_count);
    state[idx] = localState;
}
```

monte_carlo_cuda.cpp

```cpp
class CudaPiEstimator {
private:
    curandState *d_states;
    unsigned long long *d_points_inside;
    unsigned long long *h_points_inside;
    int num_blocks;
    int threads_per_block;

    void checkCudaError(cudaError_t error, const char *message) {
        if (error != cudaSuccess) {
            std::cerr << "CUDA Error: " << message << " - "
                      << cudaGetErrorString(error) << std::endl;
            exit(-1);
        }
    }

public:
    CudaPiEstimator() : threads_per_block(256) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, 0);
        num_blocks = prop.multiProcessorCount * 8;

        checkCudaError(cudaMalloc(&d_states, num_blocks * threads_per_block * sizeof(curandState)),
        "Failed to allocate device memory for random states");
        checkCudaError(cudaMalloc(&d_points_inside, sizeof(unsigned long long)),
        "Failed to allocate device memory for points counter");
        h_points_inside = new unsigned long long;
    }

    ~CudaPiEstimator() {
        cudaFree(d_states);
        cudaFree(d_points_inside);
        delete h_points_inside;
    }
```

```
    double estimate_pi(unsigned long long total_points) {
        auto start = std::chrono::high_resolution_clock::now();
        unsigned long long seed = std::chrono::system_clock::now().time_since_epoch().count();
        init_rand_kernel<<<num_blocks, threads_per_block>>>(d_states, seed);
        checkCudaError(cudaGetLastError(), "Failed to launch init kernel");

        int total_threads = num_blocks * threads_per_block;
        unsigned long long points_per_thread = total_points / total_threads;

        *h_points_inside = 0;
        checkCudaError(cudaMemcpy(d_points_inside, h_points_inside, sizeof(unsigned long long),
        cudaMemcpyHostToDevice), "Failed to copy points counter to device");

        monte_carlo_kernel<<<num_blocks, threads_per_block>>>(d_states, d_points_inside, points_per_thread);
        checkCudaError(cudaGetLastError(), "Failed to launch Monte Carlo kernel");

        checkCudaError(cudaMemcpy(h_points_inside, d_points_inside, sizeof(unsigned long long),
        cudaMemcpyDeviceToHost), "Failed to copy results from device");

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);

        std::cout << "CUDA implementation took " << duration.count() << " milliseconds" << std::endl;
        return 4.0 * (*h_points_inside) / static_cast<double>(total_points);
    }
};

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    std::cout << "GPU: " << prop.name << std::endl;

    try {
        CudaPiEstimator estimator;

        std::vector<unsigned long long> points_needed = {1ULL << 24, 1ULL << 26, 1ULL << 28};

        for (auto points : points_needed) {
            std::cout << "\nTesting for " << points << " trials..." << std::endl;
            double pi = estimator.estimate_pi(points);
            std::cout << "PI (CUDA): " << std::fixed << std::setprecision(10) << pi << std::endl;
        }
    }
    catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return -1;
    }
    return 0;
}
```

- **init_rand_kernel(curandState state, unsigned long long seed)** : This function Initializes the random number generator for each thread.

- **monte_carlo_kernel(curandState state, unsigned long long points_inside, unsigned long long points_per_thread)**: This function generates random (x, y) points, checks if they fall inside the unit circle, and updates the global count variable.

- **CudaPiEstimator**: Within this class it contains functions to allocate memory for random states and the point counter, determines the number of CUDA blocks, and initializes GPU properties.And then frees allocated GPU memory and deletes the host-side point counter.

- **estimate_pi(unsigned long long total_points)** : This function initializes random states, copies data between CPU and GPU, launches the Monte Carlo kernel, retrieves the result from the GPU, and returns the estimated value of π.

- **main():** Firstly, it retrieves GPU properties, creates a CudaPiEstimator instance, iterates over different sample sizes, runs the estimation function, and prints the results.

## Implementation details:

**i) Task decomposition:**
The task is divided among multiple CUDA threads.
Each thread is responsible for generating a subset of random points and checking if they are inside the unit circle.

**ii) Hardware Speculation:**
NVIDIA GeForce GTX 1650, threads per block = 256

**iii) Execution time measurement:**
The program uses C++'s **high_resolution_clock from <chrono>** to measure execution time.

**iv) Random number generation:**
The CUDA Random Number Generator (**cuRAND**) is used to generate uniform random values for x and y coordinates.

**Observation table: CUDA method**

| Precision Level | No of Samples Used | Time Taken (in seconds) |
|---|---|---|
| **5 decimals** | 100000000 | 0.009 |
| **10 decimals** | 10000000000 | 0.118 |
| **15 decimals** | 1000000000000 | 8.566 |
| **20 decimals** | 100000000000000 | 1019.614 |

**2.Reassess the above mentioned strategies to calculate PI using Monte Carlo Method in terms of number of trials. You should report the PI estimation and the error, along with the time consumed in each strategy.**

i) Pthreads Method : 4 threads

| No of Trials (Samples) | Estimated Pi | Error | Time Taken (in seconds) |
|---|---|---|---|
| **2^24 Trials** | 3.208 | 0.0664073 | 0.001 |
| **2^26 Trials** | 3.184 | 0.0424073 | 0.001 |
| **2^28 Trials** | 3.176 | 0.0344073 | 0.001 |

ii) OpenMP Method : 4 threads (max)

| No of Trials (Samples) | Estimated Pi | Error | Time Taken (in seconds) |
|---|---|---|---|
| **2^24 Trials** | 3.14175 | 0.000161258 | 1.165 |
| **2^26 Trials** | 3.14126 | 0.00033495 | 4.553 |
| **2^28 Trials** | 3.14172 | 0.000129996 | 18.316 |

ii) CUDA Method :

| No of Trials (Samples) | Estimated Pi | Error | Time Taken (in seconds) |
|---|---|---|---|
| **2^24 Trials** | 3.1421220303 | 0.0005293767 | 0.001 |
| **2^26 Trials** | 3.1418000460 | 0.0002073924 | 0.002 |
| **2^28 Trials** | 3.141481902 | 0.0001107516 | 0.004 |

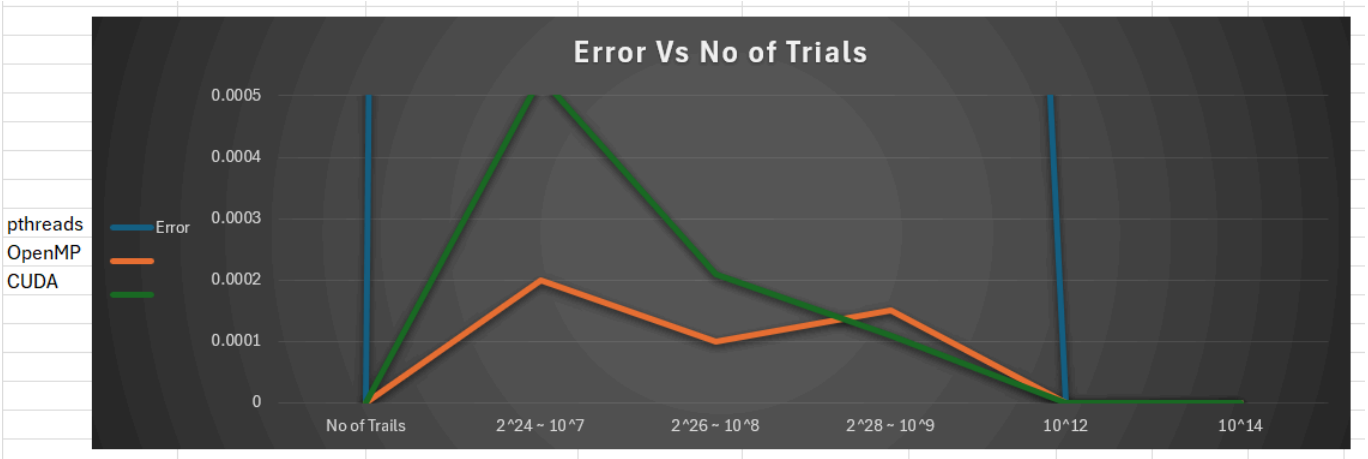
**3.Profile the best performing program using appropriate tools. Identify the bottlenecks and report on the most demanding and frequently executed sections of the program.**

The analysis of the above reported observation can be done in 2 main areas:

i) Accuracy

- pthreads: Provides the least accurate estimates for pi. Even for the larger samples (Trails), the error is considerably high.
- OpenMP: Provides accurate results with lesser errors compared to pthreads. The error diminishes with the number of trials rising. But not practical to estimate with higher no of trails (ex: Trials >> 10^10)
- CUDA: This method gives the most accurate estimates for pi. Even with smaller samples (trails) the error is extremely minimal.

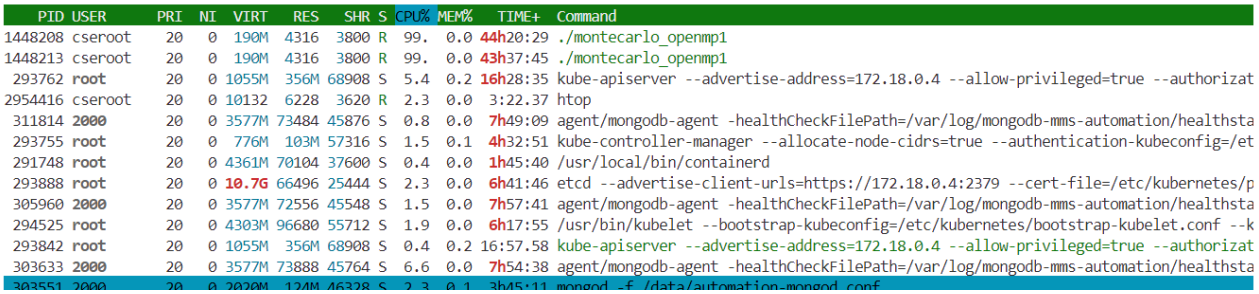
**CUDA >> OpenMP >> Pthreads**



ii) Execution Time

- pthreads: Theoretically executes moderately fine for a small number of trails but when the trails > 10^10, performance degrades.

- OpenMP: Executes faster than pthreads but when the trails > 10^10, performance degrades.
  **However in this experiment, since the machine used is heavily running Kubernetes and other programs which are also utilizing the CPU cores, the execution time increases significantly when it comes to dynamic workload allocation.**



| PID | USER | PRI | NI | VIRT | RES | SHR | S | CPU% | MEM% | TIME+ | Command |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1448208 | cseroot | 20 | 0 | 190M | 4316 | 3800 | R | 99. | 0.0 | 44h20:29 | ./montecarlo_openmp1 |
| 1448213 | cseroot | 20 | 0 | 190M | 4316 | 3800 | R | 99. | 0.0 | 43h37:45 | ./montecarlo_openmp1 |
| 293762 | root | 20 | 0 | 1055M | 356M | 68908 | S | 5.4 | 0.2 | 16h28:35 | kube-apiserver --advertise-address=172.18.0.4 --allow-privileged=true --authorizat |
| 2954416 | cseroot | 20 | 0 | 10132 | 6228 | 3620 | R | 2.3 | 0.0 | 3:22.37 | htop |
| 311814 | 2000 | 20 | 0 | 3577M | 73484 | 45876 | S | 0.8 | 0.0 | 7h49:09 | agent/mongodb-agent -healthCheckFilePath=/var/log/mongodb-mms-automation/healthsta |
| 293755 | root | 20 | 0 | 776M | 103M | 57316 | S | 1.5 | 0.1 | 4h32:51 | kube-controller-manager --allocate-node-cidrs=true --authentication-kubeconfig=/et |
| 291748 | root | 20 | 0 | 4361M | 70104 | 37600 | S | 0.4 | 0.0 | 1h45:40 | /usr/local/bin/containerd |
| 293888 | root | 20 | 0 | 10.7G | 66496 | 25444 | S | 2.3 | 0.0 | 6h41:46 | etcd --advertise-client-urls=https://172.18.0.4:2379 --cert-file=/etc/kubernetes/p |
| 305960 | 2000 | 20 | 0 | 3577M | 72556 | 45548 | S | 1.5 | 0.0 | 7h57:41 | agent/mongodb-agent -healthCheckFilePath=/var/log/mongodb-mms-automation/healthsta |
| 294525 | root | 20 | 0 | 4303M | 96680 | 55712 | S | 1.9 | 0.0 | 6h17:55 | /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --k |
| 293842 | root | 20 | 0 | 1055M | 356M | 68908 | S | 0.4 | 0.2 | 16:57.58 | kube-apiserver --advertise-address=172.18.0.4 --allow-privileged=true --authorizat |
| 303633 | 2000 | 20 | 0 | 3577M | 73888 | 45764 | S | 6.6 | 0.0 | 7h54:38 | agent/mongodb-agent -healthCheckFilePath=/var/log/mongodb-mms-automation/healthsta |
| 303551 | 2000 | 20 | 0 | 2020M | 124M | 46328 | S | 2.3 | 0.1 | 3h45:11 | mongod -f /data/automation-mongod.conf |

- CUDA: Executes very efficiently and much faster than OpenMP and Pthreads when providing high accurate estimations. (ex: Pi Estimation up to 15 or 20 decimals)

  **CUDA << OpenMP << Pthreads**

**Considering the accuracy and execution time tradeoff,**

- **CUDA is the most optimal choice for estimating the value of pi.**

**4.Discuss your findings. What are the limitations of using Monte Carlo Methods?**

- The Monte Carlo method converges at a rate of O(1/(N)^½) and therefore it requires a large number of samples for high precision and leads to high CPU/ GPU usage plus more time consuming.
- As we see in the observation tables, when we run multiple sessions for the same configuration results can vary significantly due to randomness, requiring averaging over multiple simulations. Due to this non deterministic behavior, the quality of results depends on the random number generator used, which may introduce bias if not properly designed.
- Due to the slow convergence nature of Monte Carlo method, performance degrades as the number of dimensions increases. Thus, though the process could be sped up using GPUs, still the limitation on precision persists.

## Part B

**5. Calculate the PI using Gregory-Leibniz series instead of Monte Carlo methods with multi-threaded programs. Compare and contrast against the Monte Carlo Methods.**

gregory_leibniz_openmp.cpp

```cpp
#include <iostream>
#include <cmath>
#include <chrono>
#include <omp.h>

using namespace std;
using namespace std::chrono;

const int NUM_THREADS = 24;
const long double TRUE_PI = 3.14159265358979323846264338327950288 4L;

// Function to compute Pi using OpenMP
long double compute_pi(long long terms) {
    long double piSum = 0.0L;

    #pragma omp parallel for num_threads(NUM_THREADS) reduction(+:piSum)
    for (long long k = 0; k < terms; k++) {
        long double term = (k % 2 == 0 ? 1.0L : -1.0L) / (2 * k + 1);
        piSum += term;
    }

    return 4.0L * piSum;
}

// Determine number of terms for given precision
long long get_terms_for_precision(long double requiredPrecision, long long sample) {
    long long terms = sample;
    long double estimatedPi, error;

    while (true) {
        estimatedPi = compute_pi(terms);
        error = fabsl(estimatedPi - TRUE_PI);
        if (error <= requiredPrecision) return terms;
        terms *= 2;
    }
}
```

- **compute_pi(long long terms) :** Leibniz series is used to approximate π, then summation is parallelized with OpenMP to distribute computations across multiple threads and using reduction(+:piSum) to accumulate results.

- **get_terms_for_precision(long double requiredPrecision, long long sample) :** This function determines the minimum number of terms (samples) needed for a given precision by computing π, checking the absolute error, and doubling the terms until the error is within the given threshold.

```cpp
int main() {
    auto start = high_resolution_clock::now();

    long double precisions[] = {1e-5L, 1e-10L, 1e-15L, 1e-20L};
    int precisionLevels[] = {5, 10, 15, 20};
    long long samples[] = {100000, 10000000000, 1000000000000000, 10000000000000000000 };

    for (int i = 0; i < 4; i++) {
        long long requiredTerms = get_terms_for_precision(precisions[i], samples[i]);
        long double estimatedPi = compute_pi(requiredTerms);

        cout.precision(precisionLevels[i] + 2);
        cout << "Precision: " << precisionLevels[i] << " decimal places\n";
        cout << "Estimated Pi: " << estimatedPi << "\n";
        cout << "Required Terms: " << requiredTerms << "\n\n";
    }

    auto end = high_resolution_clock::now();
    double duration = duration_cast<milliseconds>(end - start).count() / 1000.0;
    cout << "Total Execution Time: " << duration << " seconds" << endl;

    return 0;
}
```

**Observation Table:**

| Precision Level | Gregory Leibniz Time Taken (in seconds) | Gregory Leibniz Error | Monte Carlo Time Taken (in seconds) | Monte Carlo Error |
|---|---|---|---|---|
| **5 decimals** | 0.009 | 0.00000073464 | 0.404 | 0.0018073464 |
| **10 decimals** | 9.217 | 0.0000000346 | NA | NA |
| **15 decimals** | NA | NA | NA | NA |
| **20 decimals** | NA | NA | NA | NA |

Above table shows the execution time taken for both methods using OpenMP parallelization method.

- The Gregory-Leibniz series is more suitable for data parallelism since each processor can compute a range of terms independently and then accumulate into a final result.
- The Gregory-Leibniz series follows a more deterministic approach compared to Monte Carlo simulation where random sampling is done to find points inside the circle

Therefore, **Gregory-Leibniz is better** at providing a more efficient for precise estimation of π than Monte Carlo.