CS 301
High-Performance Computing

Lab 7

Pi Using Random Numbers

Aditya Nawal (202001402)
Divya Patel (202001420)

April 19, 2023

Contents

1 Introduction

In this report, we explore the use of the Monte Carlo method to estimate the value of π. We use the random function from the C library to generate random numbers and analyze the speedup achieved by using multiple processors. We conduct our experiments on both a lab and a cluster, plotting the speedup against the number of processors for a large number of points (10^9). Additionally, we investigate any potential issues in parallelizing this code and report on our findings, including our solutions to any problems encountered. In particular, we examine the problem of parallel random number generation using OpenMP.

2 Hardware Details

2.1 Lab 207 PC

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 4
- On-line CPU(s) list: 0-3
- Thread(s) per core: 1
- Core(s) per socket: 4
- Socket(s): 1
- NUMA node(s): 1
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 60
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- Stepping: 3
- CPU MHz: 3300.000
- CPU max MHz: 3700.0000
- CPU min MHz: 800.0000
- BogoMIPS: 6585.38
- Virtualization: VT-x
- L1d cache: 32K

- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K
- NUMA node0 CPU(s): 0-3
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm epb invpcid_single tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts

```
[student@localhost ~]$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 60  bytes 5868 (5.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 60  bytes 5868 (5.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

p4p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.100.64.86  netmask 255.255.255.0  broadcast 10.100.64.255
        inet6 fe80::b283:feff:fe97:d2f9  prefixlen 64  scopeid 0x20<link>
        ether b0:83:fe:97:d2:f9  txqueuelen 1000  (Ethernet)
        RX packets 32826  bytes 46075919 (43.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8015  bytes 586362 (572.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 52:54:00:3a:16:71  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 1: IP address of Lab PC

2.2 HPC Cluster

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 16
- On-line CPU(s) list: 0-15
- Thread(s) per core: 1
- Core(s) per socket: 8

- Socket(s): 2
- NUMA node(s): 2
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 63
- Model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- Stepping: 2
- CPU MHz: 1976.914
- BogoMIPS: 5205.04
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 20480K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15

3 Problem

3.1 Description of the problem

The Monte Carlo method is a statistical technique that uses random sampling to approximate numerical solutions to mathematical problems. In this case, we use the Monte Carlo method to estimate the value of Pi by generating random points within a square and counting the number of points that fall within a quarter circle inscribed within the square.

Mathematically, this can be expressed as follows: Let x and y be random variables uniformly distributed on the interval $[0, 1]$. The probability that a point (x, y) falls within the quarter circle is equal to the ratio of the area of the quarter circle to the area of the square, or $\frac{\pi}{4}$. Therefore, if we generate n random points and count the number of points k that fall within the quarter circle, we can estimate the value of Pi as $\hat{\pi} = \frac{4k}{n}$.

To analyze the speedup achieved by using multiple processors, we can use Amdahl's Law. Let P be the number of processors and f be the fraction of the code that is serial (i.e., cannot be parallelized). Then, the speedup S is given by $S(P) = \frac{1}{f+\frac{1-f}{P}}$. We can plot this function against the number of processors to analyze the speedup achieved.

In our experiments, we use both a lab and a cluster to conduct our experiments. We generate a large number of points (10^9) and plot the speedup against the number of processors. Additionally, we investigate any potential issues in parallelizing this code and report on our findings. In particular, we examine the problem of parallel random number generation using OpenMP and report on any solutions we have found to address this issue.

3.2 Compute to Memory Access Ratio

The compute to memory bound access ratio is the ratio of the number of arithmetic operations to the number of memory accesses in the code. To calculate this ratio, we need to count how many times each operation and access occurs in the code.

In the code, the arithmetic operations are:

- 1 addition and 1 multiplication in $rand\_x = (rand\_r(\&seed)/(double)RAND\_MAX)$;
- 1 addition and 1 multiplication in $rand\_y = (rand\_r(\&seed)/(double)RAND\_MAX)$;
- 2 multiplications and 1 addition in $origin\_dist = rand\_x * rand\_x + rand\_y * rand\_y$;
- 1 multiplication and 1 division in $pi = (double)(4 * circle\_points)/square\_points$;

The only memory accesses in the code are the reads and writes to the shared variable seed in the calls to $rand\_r(\&seed)$.

The total number of arithmetic operations per iteration of the loop is 8, and the total number of memory accesses per iteration is 2. Therefore, the compute to memory bound access ratio per iteration is $\frac{7}{2} = 3.5$. This means that the code is compute-bound, as it performs more arithmetic operations than memory accesses.

3.3 Memory Bound vs Compute Bound

The variables *rand_x*, *rand_y*, *origin_dist*, and *circle_points* are declared as private to each thread. This means that each thread has its own copy of these variables stored in its registers. As a result, the memory accesses to these variables are actually register accesses and do not involve main memory.

The only memory accesses in the code are the reads and writes to the shared variable seed in the calls to *rand_r*(&*seed*). However, these memory accesses are infrequent compared to the number of arithmetic operations performed in each iteration of the loop.

Therefore, the code is compute-bound, as the time to complete the computation is decided primarily by the number of elementary computation steps.

3.4 Serial Complexity

In the serial version of the program, we generate n random points and check whether each point falls within the quarter circle inscribed within the square. This requires n iterations of a loop, where each iteration involves generating two random numbers (for the x and y coordinates of the point) and checking whether the point falls within the quarter circle. The time complexity of generating a random number is constant, so the time complexity of generating two random numbers is also constant. Checking whether a point falls within the quarter circle involves computing the distance from the point to the origin and comparing it to the radius of the circle. This can be done in constant time using basic arithmetic operations.

Therefore, the overall time complexity of the serial version of the program is $O(n)$, where n is the number of points generated. This means that the running time of the program increases linearly with the number of points generated.

3.5 Parallel Complexity

In the parallel version of the program, we divide the work of generating n random points and checking whether they fall within the quarter circle among p processors. Each processor generates $\frac{n}{p}$ random points and checks whether they fall within the quarter circle. As in the serial version of the program, each iteration of the loop involves generating two random numbers and checking whether the point falls within the quarter circle. These operations can be performed in constant time.

Therefore, the time complexity of the parallel version of the program on each processor is $O(\frac{n}{p})$. Since all p processors are working simultaneously, the overall time complexity of the parallel version of the program is $O(\frac{n}{p})$. This means that by using p processors, we can reduce the running time of the program by a factor of p compared to the serial version.

3.6 Optimization Strategy

Our code uses parallelization with OpenMP to speed up the calculation of pi using the Monte Carlo method. The Monte Carlo method involves generating random points within a square and counting how many of these points fall within a circle inscribed within the square. The ratio of the number of points inside the circle to the total number of points can be used to estimate the value of pi.

One of the key features of this algorithm is that there is no dependency between the points being generated. This means that each point can be generated and checked independently of all other points. As a result, this algorithm is highly parallelizable.

Our code takes advantage of this by using OpenMP to create a team of threads that will execute the loop that generates the random points and counts how many of these points fall within the circle. Each thread generates its own random numbers using a thread-local random number generator initialized with a different seed based on its thread number.

The #pragma omp for directive is used to distribute the iterations of the loop among the threads in the team. Each thread executes a portion of the loop iterations, generating random points and counting how many of these points fall within the circle.

The private clause is used to specify that certain variables should be private to each thread. This means that each thread will have its own copy of these variables, which it can modify without affecting other threads. In Our code, the i, rand_x, rand_y, and origin_dist variables are declared as private.

The schedule(static) clause is used to specify that the iterations of the loop should be distributed among the threads using a static schedule. This means that each thread will be assigned a fixed number of iterations at compile time.

The reduction(+:circle_points) clause is used to specify that the circle_points variable should be subject to a reduction operation. This means that each thread will keep its own private copy of this variable and update it as it counts the number of points that fall within the circle. At the end of the loop, all these private copies will be combined into a single value using the + operator.

After all threads have completed their work, the value of pi is calculated using the formula pi = 4 * circle_points / square_points.

By dividing the work among multiple threads and executing these threads in parallel on multiple cores, our code can potentially achieve a significant speedup compared to a sequential version of the same algorithm.

In addition to parallelization using OpenMP, this code could also be optimized by using SIMD instructions with the immintrin.h library from Intel. This library provides a set of intrinsic functions for Intel processors that support SIMD instructions such as AVX. These functions allow you to use SIMD instructions in our C or C++ code to process multiple data elements in parallel (Refer end of the report).

By using SIMD instructions, you could generate and process multiple random numbers at once, rather than one at a time. This could result in an even faster calculation of pi.

4 Runtime and Speedup on LAB207 PCs

4.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

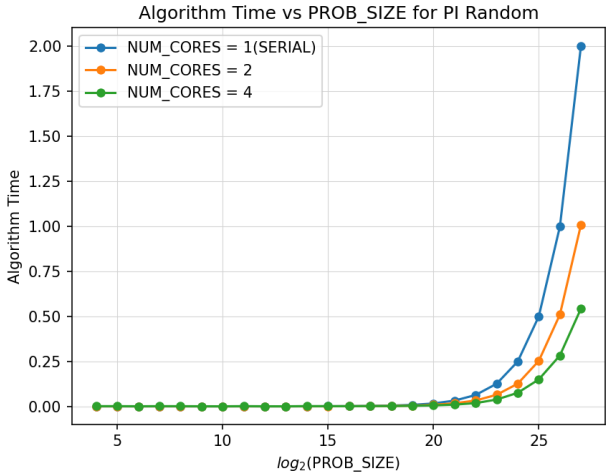

Figure 2: Graph of Problem Size vs Algorithm Runtime for Lab PC

4.2 Graph of Problem Size vs End-to-End Runtime for LAB207 PCs

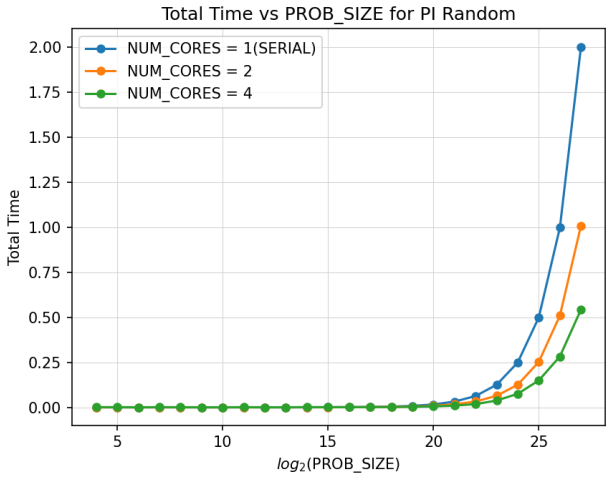

Figure 3: Graph of Problem Size vs End-to-End Runtime for Lab PC

4.3 Speedup on LAB207 PCs

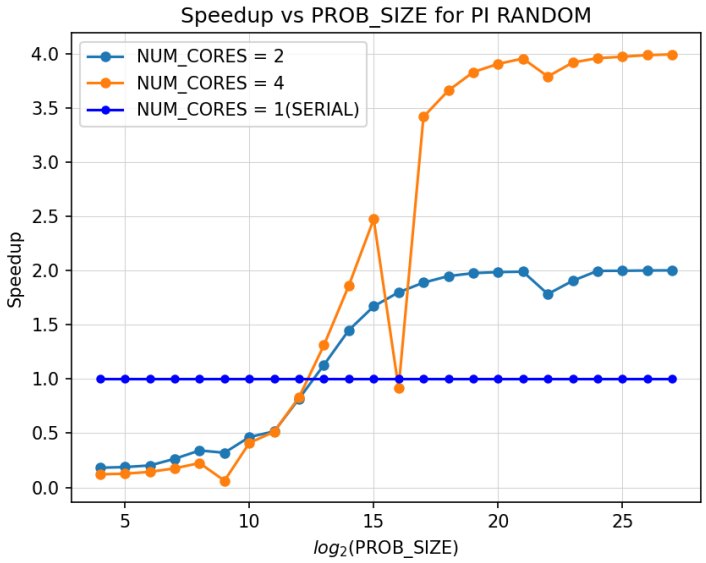

Figure 4: Graph of Speedup for Lab PC

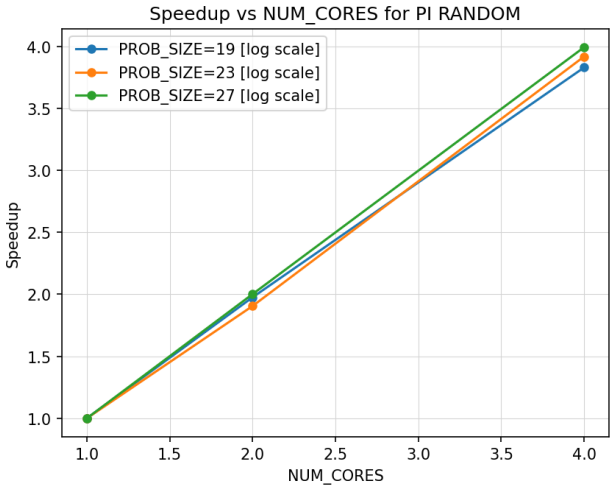

Figure 5: Processor vs Speedup on Lab PC

As we can see here speedup is increasing with the increase in number of processors and the speedup is almost linear. Also we can notice that speedup is improving with the increase in problem size as well thus we can say that the algorithm is scaling well.

5 Runtime and Speedup on HPC Cluster

5.1 Graph of Problem Size vs Algorithm Runtime for HPC Cluster

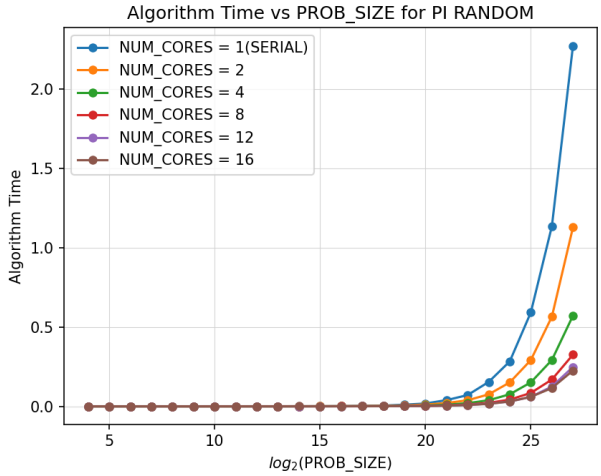

Figure 6: Graph of Problem Size vs Algorithm Runtime for HPC cluster

5.2 Graph of Problem Size vs End-to-End Runtime for HPC Cluster

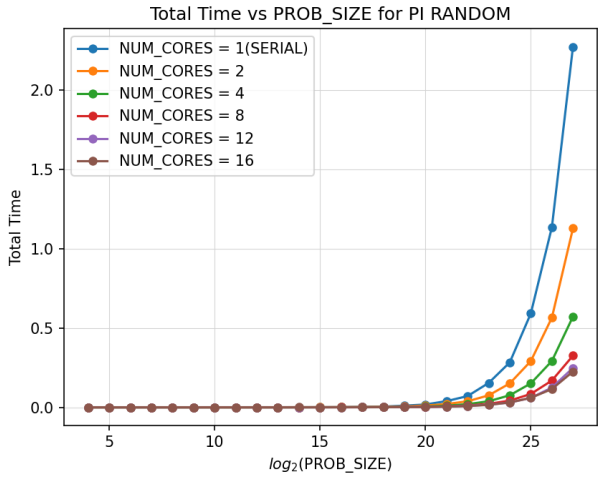

Figure 7: Graph of Problem Size vs End-to-End Runtime for HPC cluster

5.3 Speedup on HPC Cluster

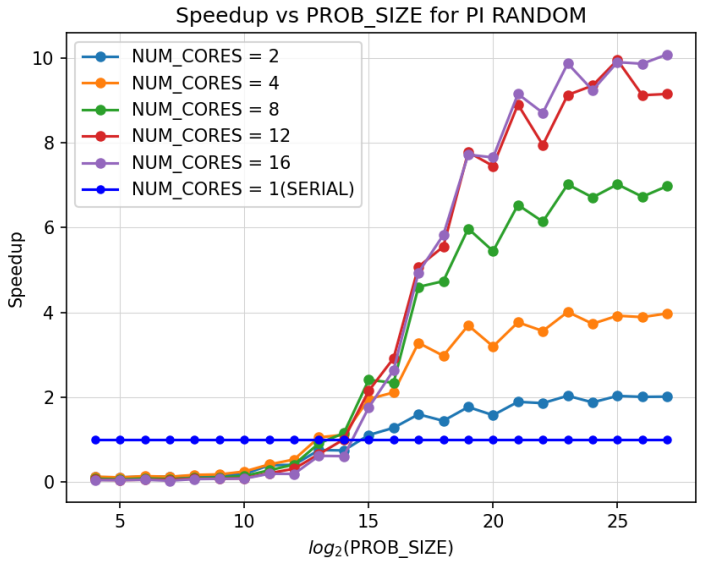

Figure 8: Processors vs Speedup for HPC cluster

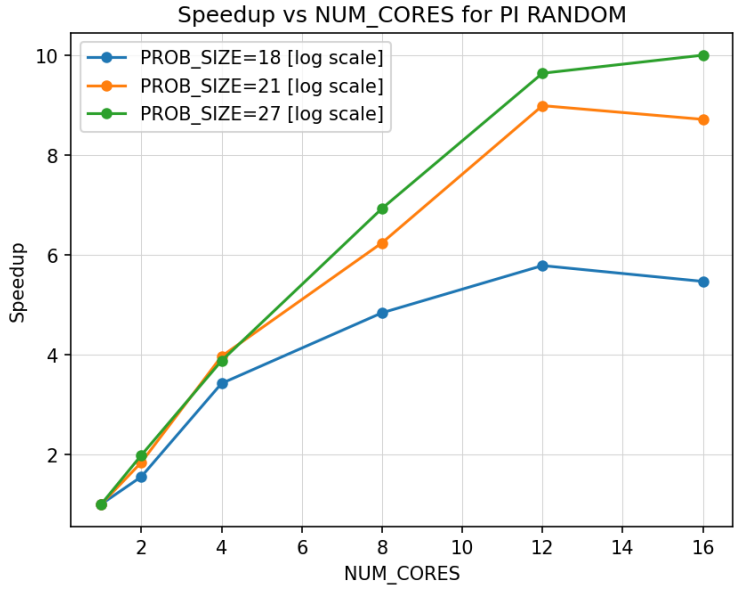

Figure 9: Processor vs Speedup on HPC Cluster

As we can see here speedup is increasing with the increase in number of processors and the speedup is almost linear. Also we can notice that speedup is improving with the increase in problem size as well thus we can say that the algorithm is scaling well.

6 Using SIMD using AVX Instructions

```
double rand_x, rand_y, origin_dist;
__m256d v_rand_x, v_rand_y, v_origin_dist;
__m256d v_one = _mm256_set1_pd(1.0);
long long int v_circle_points = 0;
for (i = 0; i < N; i += 4) {
    // Randomly generated x and y values
    v_rand_x = _mm256_set_pd((double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX);
    v_rand_y = _mm256_set_pd((double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX,
                             (double)rand_r(&seed) / RAND_MAX);
    // Distance between (x, y) from the origin
    v_origin_dist = _mm256_add_pd(_mm256_mul_pd(v_rand_x, v_rand_x),
                                  _mm256_mul_pd(v_rand_y, v_rand_y));
    // Check if the point is inside the circle
    v_circle_points += _mm_popcnt_u32(_mm256_movemask_pd(_mm256_cmp_pd(v_origin_dist,
                    v_one, _CMP_LE_OS)));
}
circle_points += v_circle_points;
pi = (double)(4 * circle_points) / square_points;
```

The code above uses the Monte Carlo method to estimate the value of pi. It generates N random (x, y) pairs and checks whether they fall within a unit circle centered at the origin. The number of points that fall inside the circle is used to estimate pi. The code takes advantage of SIMD (Single Instruction, Multiple Data) instructions using the AVX (Advanced Vector Extensions) instruction set to speed up the calculation.

The code initializes several variables, including rand_x, rand_y, and origin_dist, which will be used to store the random values generated and the distance between the point and the origin. It also initializes the AVX vector v_one to 1.0 and sets the variable v_circle_points to 0, which will be used to count the number of points that fall within the unit circle.

The loop iterates through N random (x, y) pairs. Within the loop, four pairs of random values are generated simultaneously using the rand_r() function, which returns a random integer value between 0 and RAND_MAX. The random values are then converted to doubles and stored in the AVX vectors v_rand_x and v_rand_y using the _mm256_set_pd() function.

Next, the distance between each point and the origin is calculated and stored in the AVX vector v_origin_dist using the _mm256_add_pd() and _mm256_mul_pd() functions to compute the square of the distance.

Finally, the code uses the _mm256_cmp_pd() function to compare the distance to the value of 1.0 stored in the AVX vector v_one. The _CMP_LE_OS parameter specifies that the comparison should be less than or equal to, and the result is stored in an AVX mask. The _mm256_movemask_pd() function is used to convert the AVX mask to a 32-bit integer mask, and the _mm_popcnt_u32()

function is used to count the number of set bits in the mask. The resulting count is added to the variable v_circle_points.

After the loop has completed, the total number of points that fell inside the circle is added to the variable circle_points, and the value of pi is estimated using the formula pi = 4 * (number of points inside the circle) / (total number of points).

Using SIMD instructions with the AVX instruction set improves performance by allowing multiple calculations to be performed simultaneously. In this code, the use of AVX vectors allows four pairs of random (x, y) values to be generated and their distances to be calculated in a single operation. The _mm_popcnt_u32() function is used to count the number of set bits in an AVX mask, which can also be done more efficiently than counting bits in a regular integer. By taking advantage of the parallelism provided by SIMD instructions, the code is able to process more data in less time, resulting in faster execution and improved performance.

6.1 Graph of Problem Size vs Total Runtime for HPC Cluster using SIMD

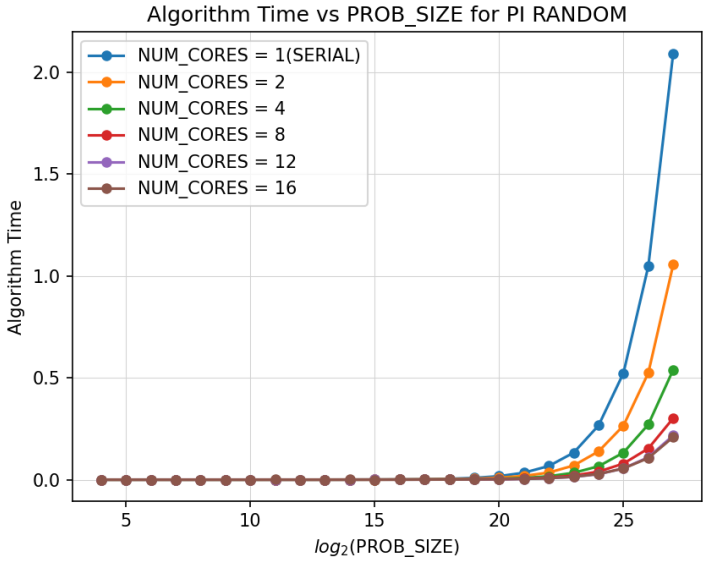

Figure 10: Graph of Problem Size vs Total Runtime for HPC Cluster with SIMD

6.2 Speedup on HPC Cluster using SIMD

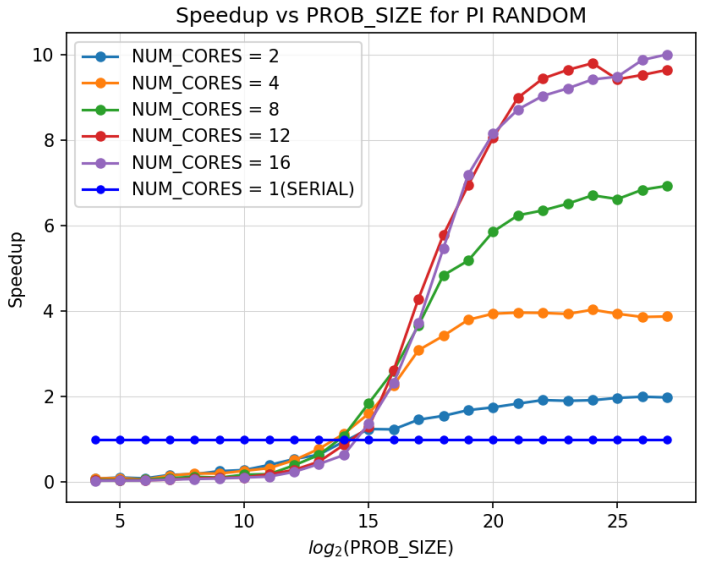

Figure 11: Graph of Speedup for HPC Cluster using SIMD

7 Discussion

One potential issue that may arise when parallelizing the code to estimate the value of pi using random numbers is related to parallel random number generation using OpenMP. When generating random numbers in parallel, it is important to ensure that each thread uses a different seed value for its random number generator. Otherwise, all threads will generate identical sequences of random numbers, which will affect the accuracy of the estimate of pi.

A random number generator is an algorithm that generates a sequence of numbers that are not actually random, but share many properties with completely random numbers. The sequence of numbers generated by a random number generator is determined by its initial seed value. If two random number generators are initialized with the same seed value, they will generate identical sequences of numbers.

In a parallel code using OpenMP, each thread has its own copy of the variables declared within its scope. This means that if each thread initializes its own random number generator with a different seed value, it will generate a different sequence of random numbers. However, if all threads initialize their random number generators with the same seed value, they will all generate identical sequences of random numbers.

In order to ensure that each thread uses a different seed value for its random number generator, one approach is to use the thread number as part of the seed value. For example, the seed value for each thread can be calculated as $time(NULL) * omp\_get\_thread\_num()$. This ensures that each thread uses a different seed value for its random number generator and generates a different sequence of random numbers.

In conclusion, when parallelizing code that involves generating random numbers using OpenMP, it is important to ensure that each thread uses a different seed value for its random number generator. Otherwise, all threads will generate identical sequences of random numbers, which will affect the accuracy of the results.