CS-301
High Performance Computing

Assignment 2

Amarnath Karthi 201501005
Chahak Mehta 201501422

September 10, 2017

1 Hardware and Compiler specifications

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	12
On-line CPU(s) list	0-11
Thread(s) per core	1
Core(s) per socket	6
Socket(s)	2
NUMA node(s)	2
Vendor ID	GenuineIntel
CPU family	6
Model	63
Model name	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping	2
CPU MHz	1211.531
BogoMIPS	4804.10
Virtualization	VT-x
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	15360K
NUMA node0 CPU(s)	0-5
NUMA node1 CPU(s)	6-11

Language	C++14
Compiler	g++
Flag(s)	-fopenmp

2 Scalar Product

Scalar product of 2 vectors has the following logic :

$$a \cdot b = \sum_{i=1}^{n} a_i b_i \tag{1}$$

The following is the C++14 implementation to calculate the scalar product of 2 vectors:

```
long int dotProduct(vector<int> &a, vector<int> &b) {
    long int product = 0;
    for(int i=0;i<n;i++) {
        product += a[i]*b[i];
    }
    return(product);
}
```

2.1 Parallelizing the serial code

If we take a close look at the serial code, we see that the output is just a summation of vector quantities of a and b. A naive approach would be to parallelize the for loop amongst various threads by using `omp parallel for` directive with the summation done as a **critical** or an **atomic** operation by using `omp critical` or `omp atomic` directives respectively.

However, the introduction of critical or atomic regions will degrade the performance, because only one thread can access the critical section of the code. A better solution would be to apply **reduction**. We can apply reduction here because summation(+) is an associative and a commutative operation. We apply reduction using the `omp reduction(+:sum)` clause along with the `omp parallel for` directive. The following is the optimal parallel implementation of the code in C++14:

```
long int dotProduct(vector<int> &a, vector<int> &b) {
    long int product = 0;
    #pragma omp parallel for reduction(+:product)
    for(int i=0;i<n;i++) {
        product += a[i]*b[i];
    }
    return(product);
}
```

The above implementation essentially provides a faster multithreaded solution by first distributing the load equally among all the cores. Each core then does a summation of $a_i b_i$ from index i_1 to i_2, which are different for each core. After all the cores have completed the operation, the summations are reduced to 1 variable.

2.2 Parallel overheads

Using OpenMP for task distribution and synchronization adds some overhead time to the code. We measure this time as the difference between the serial code and the parallel code run on 1 core.

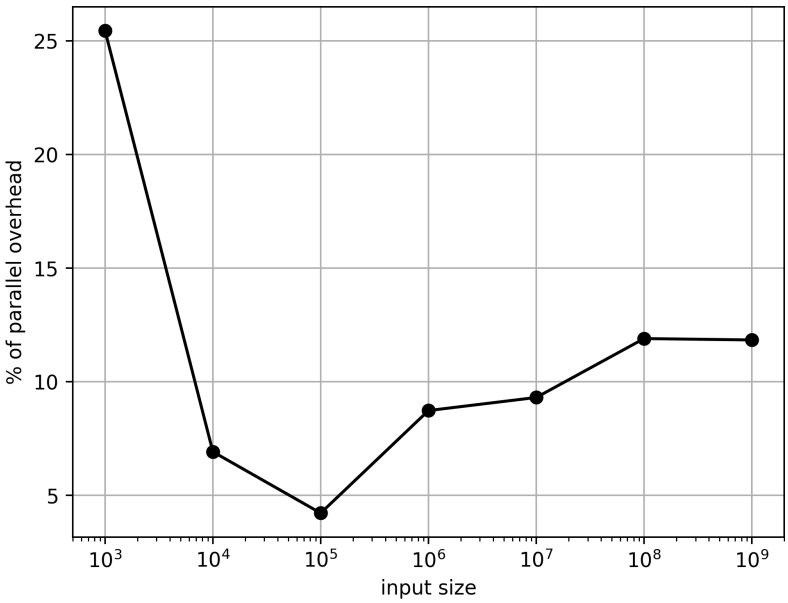

Figure 1: input size vs % of parallel overhead

We define overhead percentage O as the percentage of difference between serial and parallel code execution time on 1 core with respect to the serial execution time.

$$O = \frac{T_{p_1} - T_s}{T_s} \cdot 100 \tag{2}$$

Figure 1 shows the variation of the parallel overhead with input size. Parallel overhead is measured as % of the serial time for the corresponding input size. We see that for a size of 10^3 the overhead time by the parallel code is 25%. As the problem size increases, this overhead decreases, with it being at about 5% for input size 10^5. After this it again starts increasing gradually, becoming about 12% for input size 10^9.

For small input sizes, we see a greater overhead because for these problems, the actual compute time is very low, therefore the parallel overhead is a larger percentage of the serial time. Whereas, for larger problems, the compute time is very high and therefore the parallel overhead percentage is very low.

2.3 Input size vs execution time

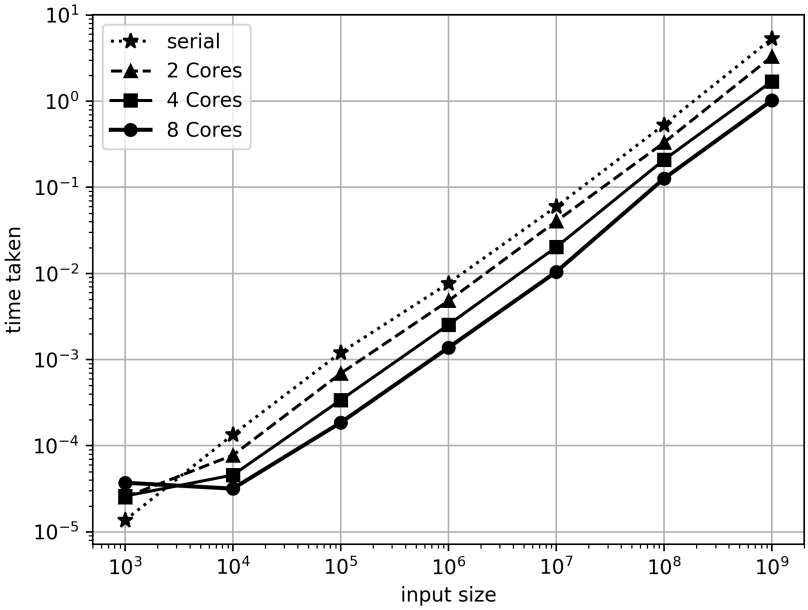

Figure 2: input size vs execution time in log-log scale

As we increase the problem size by a factor of 10, we expect the execution time to also increase more or less proportionately. Off course this is a gross estimation which does not take into account the caching in the architecture. Also as we increase the number of cores, for a given input size, the execution time should decrease (for an optimal parallel algorithm). **Figure 2** is the actual curve we get from our experiments. Not that it is plotted in log-log scale considering the huge ranges of data.We see that this more or less adheres to our observations, with only one exception.

For low input sizes, of the order of 10^3, the execution time increases as we increase the number of cores. For small input sizes, the actual serial computation time is very low. Parallelizing the code only adds some more overheads, which dominate for small input sizes. The time taken for synchronization, and load distribution is greater than the time saved by parallelizing the code. Whereas, this is not the case for large input sizes, because here although the compute time increases significantly, the parallel overhead remains more or less the same.

2.4 Speedup vs input size

$$speedup = \frac{T_{serial}}{T_{parallel}} \tag{3}$$

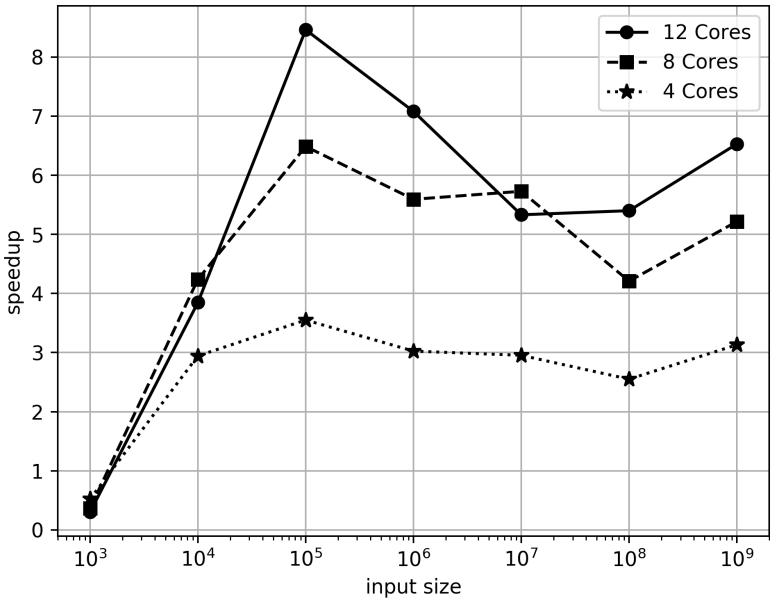

Figure 3: speedup vs input size

Figure 3 shows the variation of speedup with problem size. Notice that speedup first increases till problem size becomes 10^5 and then starts decreasing till problem size becomes 10^8. After this, it increases slightly till problem size becomes 10^9. An important pattern emerges here. Notice that irrespective of the number of processors, speedup is maximum when input size is of the order of 10^5.

One possible reason for this might be the NUMA model which the computer follows. Initially as the input size is small, the parallel overhead dominates over the time saved by parallelizing the code. As the problem size increases, the compute time becomes larger , whereas the parallel overhead remains more or less the same. Hence more compute time is saved by parallelizing the code thereby giving us greater and greater speedup.

After the array sizes becomes sufficiently large, each processor's cache cannot hold the array by itself. Therefore, it needs to access the array from the main memory, thereby increasing the time taken to compute.

2.5 Speedup vs number of cores

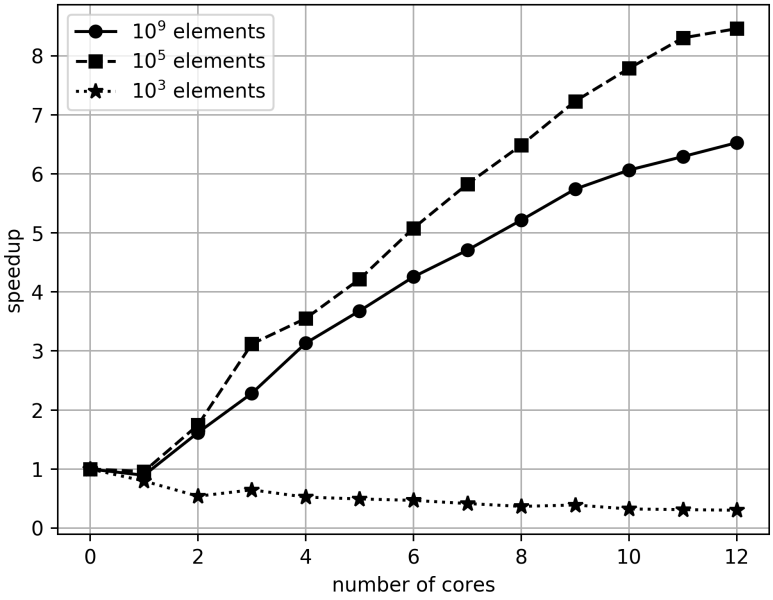

Figure 4: speedup vs number of cores

Figure 4 shows the variation of speedup with the number of cores for different problem sizes. 2 notable patterns can be seen here.

When the problem size is small(of the order of 10^3), increasing the number of cores just adds more parallel overhead because of increased inter-processor synchronization and communication. This is not compensated by the time saved in parallelization since the compute time is already very low incase of the serial code.

For large problem sizes, the case is opposite. Adding a new processor increases the speedup. This is because the compute time is inversely proportional to the number of processors. For large input sizes, the compute time is very high. So even though the parallel overhead increases slightly, a huge amount of time is saved by adding one extra processor. However note that the marginal gain in speedup by addition of 1 processor decreases as the number of processors increase. Thus speedup follows the **law of diminishing marginal returns** in this problem. We take a look at this in the next subsection where we discuss the efficiency.

2.6 Efficiency vs the number of cores

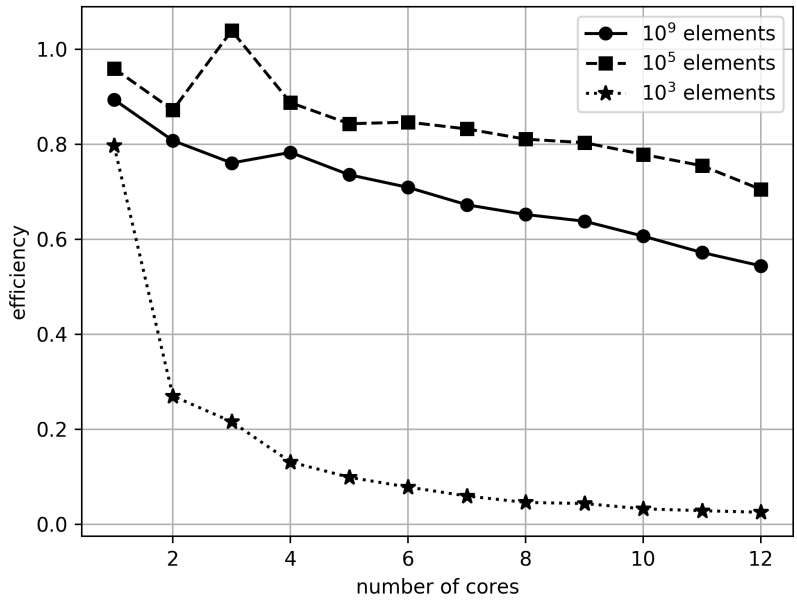

Figure 5: efficiency vs number of cores

Efficiency e is defined as speedup per unit processors. It is the ratio of s_p, the speedup in case of p processors and p.

$$e = \frac{s_p}{p} \tag{4}$$

As seen in the previous section, the speedup follows the law of diminishing marginal returns in this problem. Therefore when we plot the efficiency vs the number of cores, we expect a decreasing behavior. This argument is validated in **figure 5**.

A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.

The efficiency decreases with increase in number of cores, hence this is a **weakly scalable**. One possible reason for this might be the fact that the load might not get equally distributed across all processors. Another reason might be that all the processors are accessing the same memory i.e. array in this case. Hence the access speed decreases as a lot of processors try to fetch the array elements, thereby further decreasing the speedup.

3 Calculation of π : Monte Carlo method

The Monte Carlo method is a stochastic method which can be used to calculate the value of π.

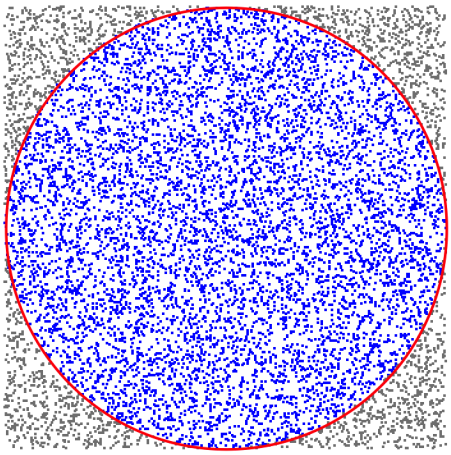

Figure 6: Monte Carlo method for the calculation of π

Basically in this method, random "darts" are projected in a 2x2 square space. Let number of darts projected on the circular region be n_c. Let total number of points be N. Therefore the probability of a random dart landing on the central circular region is $\frac{n_c}{N}$. Also the same probability is equal to $\frac{A_c}{A_s}$ where A_c is the area of the circle and A_s is the area of square. A_c has the value π since the radius of the circle is 1 and A_s has the value 4. Therefore:

$$\frac{\pi}{4} = \frac{n_c}{N} \tag{5}$$

The following is the serial implementation of the algorithm in C++14:

```
double getPi(long numPoints) {
    long n_c = 0;
    for(int i=0;i<numPoints;i++) {
        double x = -1 + (2.0*rand())/(RAND_MAX);
        double y = -1 + (2.0*rand())/(RAND_MAX);
        if(sqrt(x*x + y*y)<=1)
            n_c++;
    }
    double pi = (4.0*n_c)/(num*1.0);
    return(pi);
}
```

Notice that we are using the `rand()` function in the serial implementation.

3.1 Parallelizing the serial code

If we take a close look at the serial code, we see that the output is just summation on `n_c`. A naive approach would be to parallelize the for loop amongst various threads by using `omp parallel for` directive with the summation on `n_c` done as a **critical** or an **atomic** operation by using `omp critical` or `omp atomic` directives respectively.

However, the introduction of critical or atomic regions will degrade the performance, because only one thread can access the critical section of the code. A better solution would be to apply **reduction**. We can apply reduction here because summation(+) is an associative and a commutative operation. We apply reduction using the `omp reduction(+:n_c)` clause along with the `omp parallel for` directive. The following is the initial implementation of the code in C++14:

```
double getPi(long numPoints) {
        long n_c = 0;
        #pragma omp parallel for reduction(+:n_c)
        for(int i=0;i<numPoints;i++) {
            double x = -1 + (2.0*rand())/(RAND_MAX);
            double y = -1 + (2.0*rand())/(RAND_MAX);
            if(sqrt(x*x + y*y)<=1)
                n_c++;
        }
        double pi = (4.0*n_c)/(num*1.0);
        return(pi);
    }
```

However we encounter a problem. This code takes a much greater execution time as compared to even the naive serial code. This is because the function *rand*() is not thread safe.

3.1.1 `rand()`

This function returns a random number from 0 to `RAND_MAX`. It is a **linear congruential generator** which uses the following algorithm:

$$X_n + 1 = (aX_n + c) mod m \tag{6}$$

The value X_n is saved as the state of the generator for the next use. THe problem with `rand()` is that its a non reentrant function i.e. entering into the function every time changes the state it has. Basically whenever a thread uses the `rand()` function, it modifies the state. To avoid multthreaded ace conditions, this step is made critical, thereby making it act as a bottleneck in the parallel code.

3.2 rand_r()

This function is just a modification of the `rand()` function. We just pass the state as an integer pointer during function call, making it a thread safe method, since each thread can use a different seed. Thus the most optimal parallel code is :

```
double getPi(long numPoints) {
        long n_c = 0;
        #pragma omp parallel{
            int seed = time(NULL)
            #pragma omp for reduction(+:n_c)
            for(int i=0;i<numPoints;i++) {
                double x = -1 + (2.0*rand())/(RAND_MAX);
                double y = -1 + (2.0*rand())/(RAND_MAX);
                if(sqrt(x*x + y*y)<=1)
                    n_c++;
            }
        }
        double pi = (4.0*n_c)/(num*1.0);
        return(pi);
    }
```

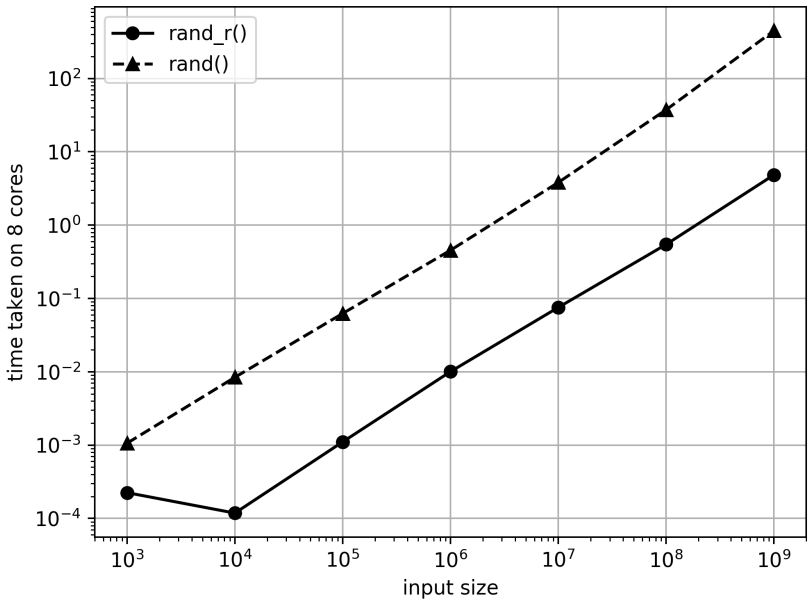

Figure 7: Time taken by rand vs rand_r (log log scale)

3.3 Parallel overheads

We define overhead percentage O as the percentage of difference between serial and parallel code execution time on 1 core with respect to the serial execution time.

$$O = \frac{T_{p_1} - T_s}{T_s} \cdot 100 \tag{7}$$

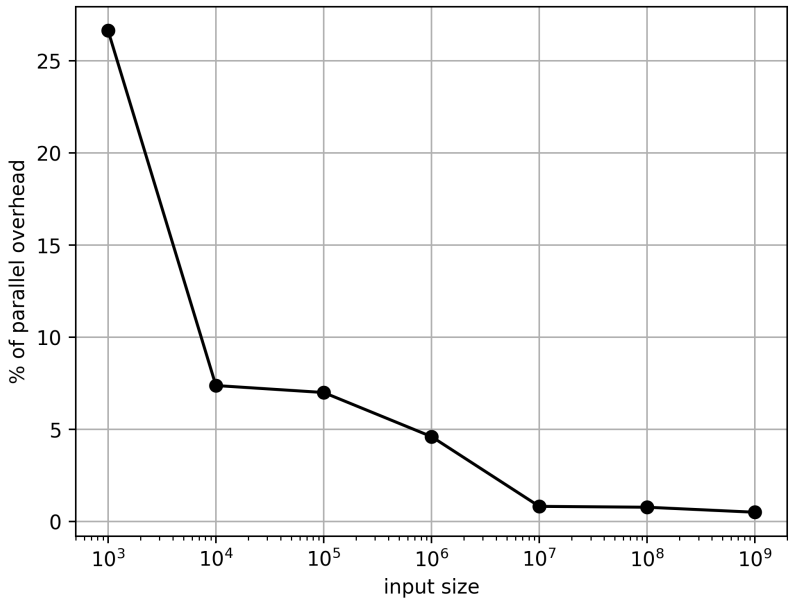

Figure 8: Parallel overhead vs input size

In the above figure we see variation of parallel overhead with respect to the input size.

For small input sizes, we see a greater overhead because for these problems, the actual compute time is very low, therefore the parallel overhead is a larger percentage of the serial time. Whereas, for larger problems, the compute time is very high and therefore the parallel overhead percentage is very low.

For large input sizes the overhead is barely 2%, whereas for very small inputs it can be as large as 25%.

3.4 Input size vs execution time

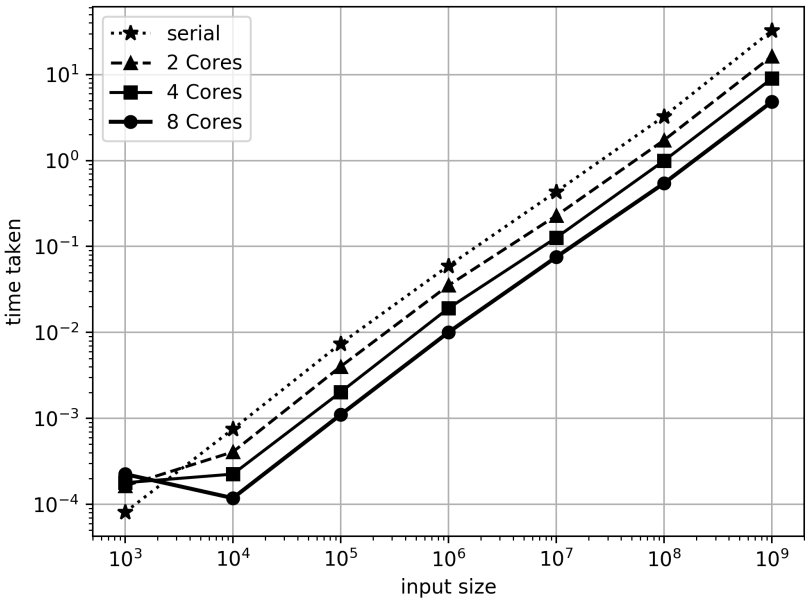

Figure 9: input size vs execution time

As we increase the problem size by a factor of 10, we expect the execution time to also increase more or less proportionately. Off course this is a gross estimation which does not take into account the caching in the architecture. Also as we increase the number of cores, for a given input size, the execution time should decrease (for an optimal parallel algorithm). **Figure 9** is the actual curve we get from our experiments. Not that it is plotted in log-log scale considering the huge ranges of data.We see that this more or less adheres to our observations, with only one exception.

For low input sizes, of the order of 10^3, the execution time increases as we increase the number of cores. For small input sizes, the actual serial computation time is very low. Parallelizing the code only adds some more overheads, which dominate for small input sizes. The time taken for synchronization, and load distribution is greater than the time saved by parallelizing the code. Whereas, this is not the case for large input sizes, because here although the compute time increases significantly, the parallel overhead remains more or less the same.

3.5 Speedup vs input size

$$speedup = \frac{T_{serial}}{T_{parallel}} \tag{8}$$

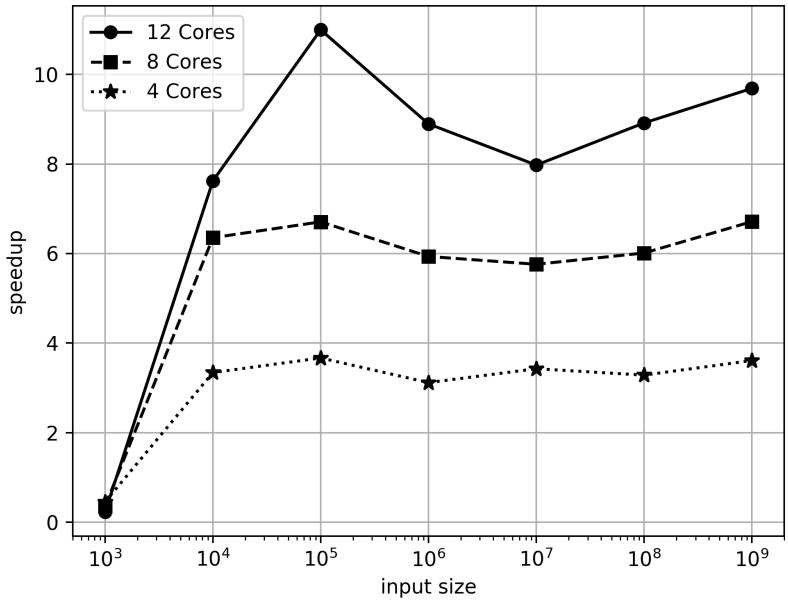

Figure 10: speedup vs input size

Figure 3 shows the variation of speedup with problem size. Notice that speedup first increases till problem size becomes 10^5 and then remains the same more or less.

Initially as the input size is small, the parallel overhead dominates over the time saved by parallelizing the code. As the problem size increases, the compute time becomes larger , whereas the parallel overhead remains more or less the same. Hence more compute time is saved by parallelizing the code thereby giving us greater and greater speedup.

Unlike the previous question, there is not much memory access required here. Hence for large input sizes, speedup remains roughly the same.

3.6 Speedup vs number of cores

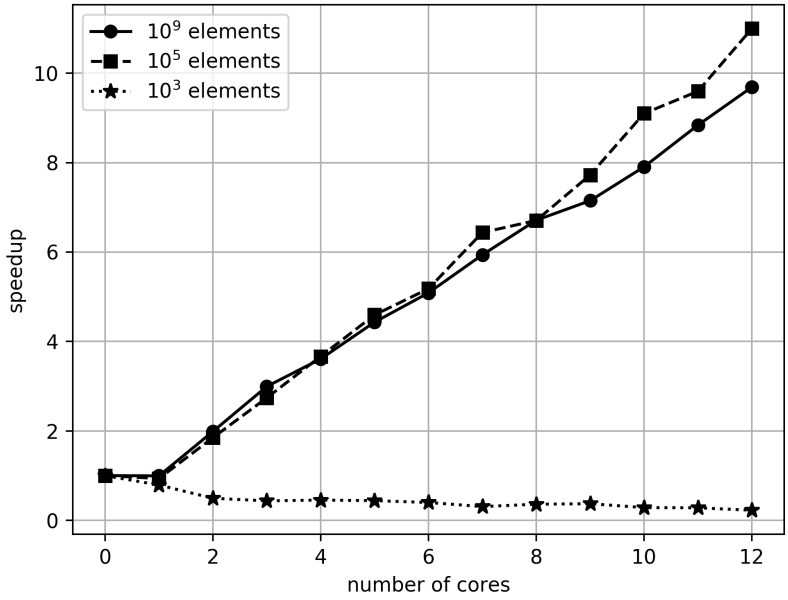

Figure 11: speedup vs number of cores

Figure 11 shows the variation of speedup with the number of cores for different problem sizes. 2 notable patterns can be seen here.

When the problem size is small(of the order of 10^3), increasing the number of cores just adds more parallel overhead because of increased inter-processor synchronization and communication. This is not compensated by the time saved in parallelization since the compute time is already very low incase of the serial code.

For large problem sizes, the case is opposite. Adding a new processor increases the speedup. This is because the compute time is inversely proportional to the number of processors. For large input sizes, the compute time is very high. So even though the parallel overhead increases slightly, a huge amount of time is saved by adding one extra processor. However note that the marginal gain in speedup by addition of 1 processor remains the same as the number of processors increase. Thus speedup does not follow the **law of diminishing marginal returns** in this problem. Hence this problem is **highly scalable** since speedup remains the same.

3.7 Convergence to π

On serial execution, the algorithm takes 32.439 seconds to calculate the value of π as 3.1416. On 12 cores the parallel algorithm gives the same accuracy in 3.36 seconds. Giving a speedup of roughly 10.