TEHNICAL REPORT

Evaluating parallel algorithm scalability

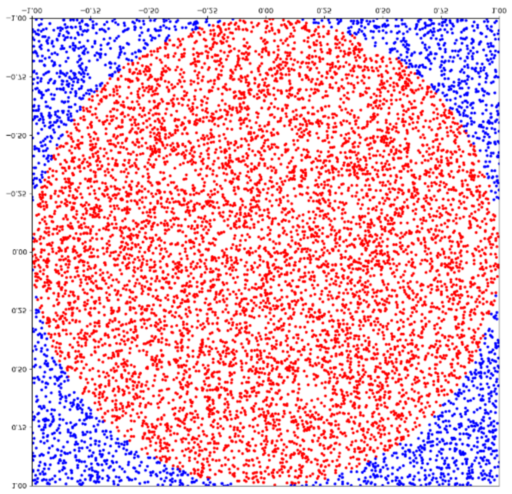
Larisa-Maria Biriescu, Big Data, year 1

Algorithm 1: Pi estimation using Monte Carlo Method

Description of algorithm: *(Problem area, importance, need for parallelization)*

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. One examples of usage of the Monte Carlo algorithm is the estimation of Pi.

The **idea** is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit. After the simulation we calculate the ratio of the number of points that lied inside the circle (inscribed into the square that has the same domain and with the same diameter) and the total number of points generated.



$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$

This **algorithm** is very popular due to it's simplicity. One important thing to be mentioned is that the accuracy of the estimation increases when the number of points increases as well.

Due to the fact presented above, the need for parallelization arises in order to minimize the time needed to get the result and also get a result as accurate as possible(this implies generating billions of random numbers).

Pseudocode: *(sequential version, parallel version concept - figure)*

The Algorithm

```
1. Initialize circle_points, square_points to 0.
2. Generate random point x.
3. Generate random point y.
4. Calculate d = x*x + y*y.
5. If d <= 1, increment circle_points.
6. Increment square_points.
7. If increment < NO_OF_ITERATIONS, repeat from 2.
8. Calculate pi = 4*(circle_points/square_points).
9. Terminate.
```

As it can be seen from the pseudo-code above, this algorithm can easily be parallelized due to the fact that the current iteration in which a point is generated doesn't depend on the previous iterations at all.

Aspects of security and data confidentiality

All the tests were run on Google Cloud Platform, using the Google Compute Engine.

They offer:

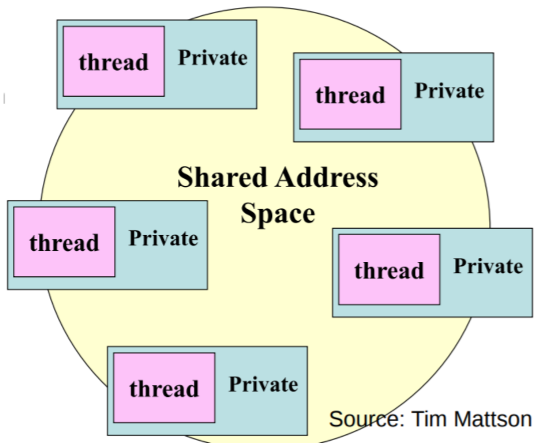
- **Network security**: Help secure the network with products that define and enforce your perimeter and allow for network segmentation, remote access, and DoS defense.
- **Endpoint security**: Help secure endpoints and prevent compromise with device hardening, device management, and patch and vulnerability management.
- **Data security**: Make sensitive data more secure with data discovery, data governance, and controls to prevent loss, leakage, and exfiltration.
- Identity and access management: Manage and secure employee, partner, customer, and other identities, and their access to apps and data, both in the cloud and on-premises.
- **Security monitoring and operations**: Monitor for malicious activity, handle security incidents, and support operational processes that prevent, detect, and respond to threats.

Also, the Google privacy team participates in every product launch, reviewing design documentation and performing code reviews to ensure that privacy requirements are followed. They help ensure that their Google Cloud products and services always reflect strong privacy standards that protect the customers.

Testing scenarios:

1. OpenMP

OpenMP (Open Multi-Processing) is a popular shared-memory programming model. It is supported by popular production C (also Fortran) compilers: Clang, GNU Gcc, IBM xlc, Intel icc.



Main concepts:

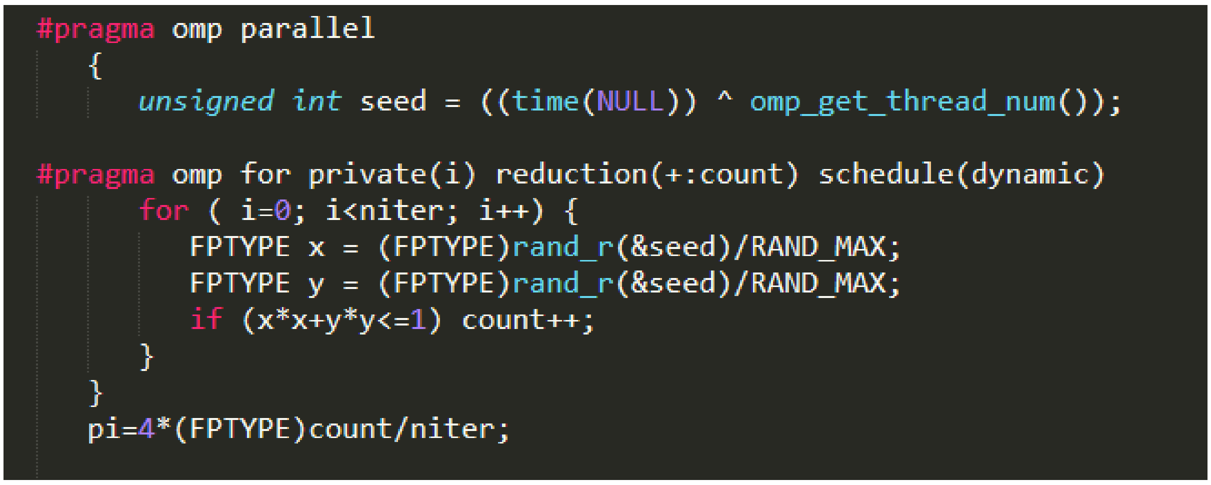
– **Parallel** regions: where parallel execution occurs via multiple concurrently executing threads

– Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution

– Shared and private data: shared variables are the means of communicating data between threads

– **Synchronization**: Fundamental means of coordinating execution of concurrent threads

– Mechanism for automated work distribution across threads



```
#pragma omp parallel
  {
    unsigned int seed = ((time(NULL)) ^ omp_get_thread_num());

#pragma omp for private(i) reduction(+:count) schedule(dynamic)
    for ( i=0; i<niter; i++) {
      FPTYPE x = (FPTYPE)rand_r(&seed)/RAND_MAX;
      FPTYPE y = (FPTYPE)rand_r(&seed)/RAND_MAX;
      if (x*x+y*y<=1) count++;
    }
  }
  pi=4*(FPTYPE)count/niter;
```

Parallel region PI estimation with OpenMP

In the above figure pe used the parallel and for regions. Each thread generates a specified number of points(niter) and after that using **reduction(+:count)** the count of points that are in the circle are summarized between all threads in order to compute the values of pi after the generation of all needed points. I used **dynamic** scheduling in order to have a better workload distribution.

Tested for: 1E8, 1E9, 1E10 number of **data points**

On a VM created on Google Cloud, using 1, 2, 4, 8, 16, 24, 32 **threads.**

For each testing scenario the same test has been ran 10 times and the average of executions times and pi values were plotted in the next section.

2. MPI

MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. The goals of MPICH are: (1) to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters (desktop systems, shared-memory systems, multicore architectures), high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet, Quadrics) and proprietary high-end computing systems (Blue Gene, Cray) and (2) to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations

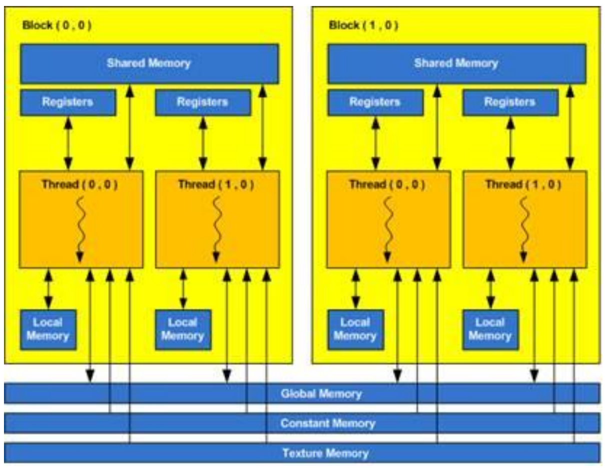
Tested for: 1E8, 1E9, 1E10 number of **data points**

On 1-8 **VMs** created on Google Cloud**.**

For each testing scenario the same test has been ran 10 times and the average of executions times and pi values were plotted in the next section

3. CUDA

CUDA is Nvidia's vision for the use of graphics processing units for general purpose computing ('GPGPU').

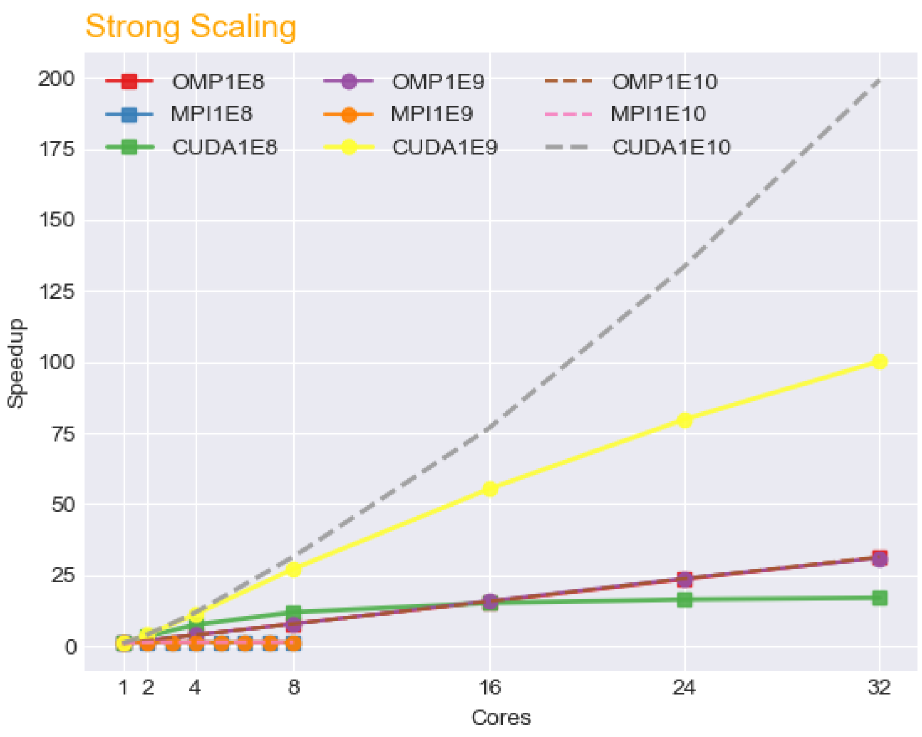


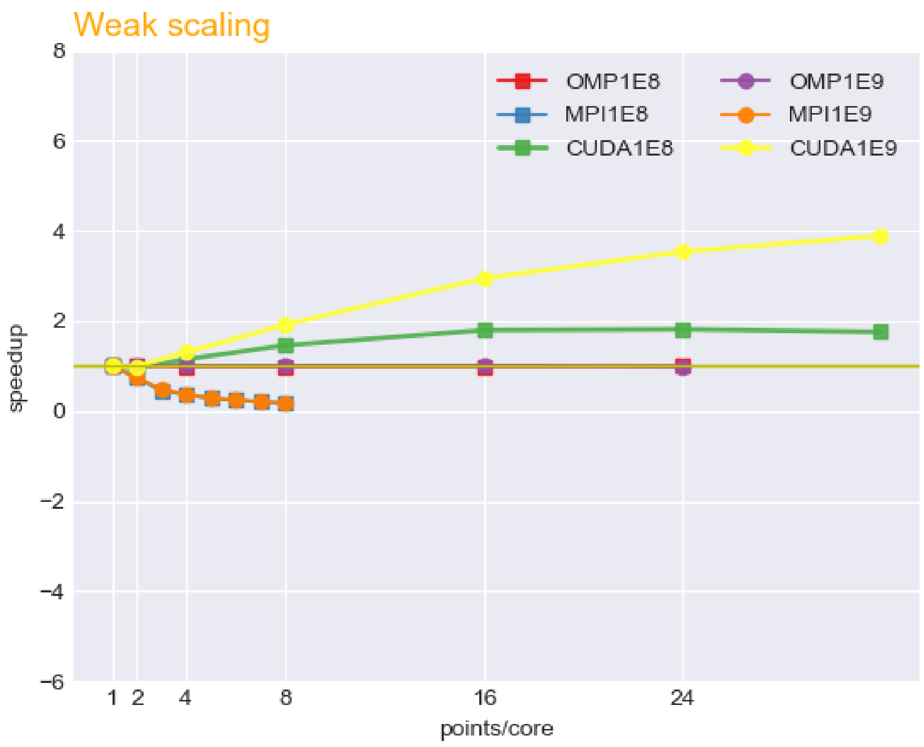
Tested for: 1E8, 1E9, 1E10 number of **data points**

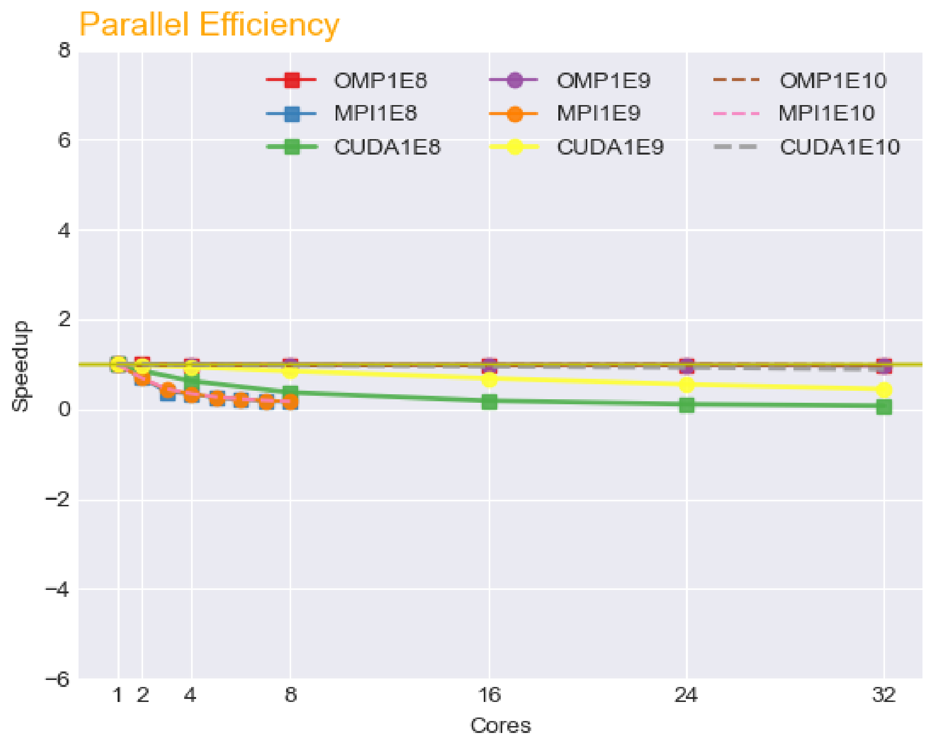
On NVIDIA TESLA P100 with 1GPU created on Google Cloud**.** The test were made by creating 1-8 **blocks** and for each block having 1, 2, 4, 8, 16, 32 **threads**.

For each testing scenario the same test has been ran 10 times and the average of executions times and pi values were plotted in the next section

Results:


Strong Scaling
OMP1E8
MPI1E8
CUDA1E8
OMP1E9
MPI1E9
CUDA1E9
OMP1E10
MPI1E10
CUDA1E10
Speedup
200
175
150
125
100
75
50
25
0
1
2
4
8
16
24
32
Cores


Weak scaling
OMP1E8
MPI1E8
CUDA1E8
OMP1E9
MPI1E9
CUDA1E9
speedup
8
6
4
2
0
−2
−4
−6
1
2
4
8
16
24
points/core



Discussions:

1. **Strong scaling**

For the strong scaling evaluation we wanted to see how OpenMP, MPI and CUDA performs while running our algorithm on a constant number of points, but varying the number of cores that are doing the work.

We can see clearly that CUDA performs the best for bigger number of points (1E9 and 1E10) but for a smaller number of data points, OpenMP performs slightly better. MPI in this case is on the last place, performing poorly no matter the number of data points.

2. **Weak scaling**

For weakly scaling evaluation we wanted to see if there exists a decrease in performance when we increase the number of cores but at the same time we keep the work/core constant. Eg: 1M data points generated/ 1 core, 2M data points generated/2 cores etc.

For CUDA we can see a speedup increase as the number of cores increases. On the other hand, OpenMP performes by far the best in both ranges of data points.

MPI is having the same drop in performance as CUDA does.

We can see that for 1E9 data points to be generated, there is no significant difference between CUDA's performance and OpenMP's performance for up to 8 cores.

3. Parallel efficiency

For parallel efficiency evaluation we can definitely see a drop in performance at MPI and also at CUDA, but for the former it's only for the smaller number of datapoints to be generated. As the number of datapoints increases (eg: 1E10), CUDA performs as better as OpenMP which doesn't seem to suffer any degradation in performance regardless of the number of data points to be generated.

Optimal configuration

OpenMP is by far the winner because it scales weakly as well as being parallel efficient.

CUDA scales strongly having a speed-up of almost 200 seconds when having 7 blocks with 32 threads each.

Although, if we ran them on max 8 cores, the increased speedup given by CUDA is not that significant for up to 1E9 data points to be generated.

Other remarks

We don't need just an algorithm that run the fastest, we want also one that gives us the correct results.

PI value, first 20 digits: **3.1415926535 8979323846**

	1E8	1E9	1E10
OpenMP	3.1415990000000003	3.141584000000000	3.141591
CUDA	3.1415900000000003	3.1415900000000003	3.1415900000000003
MPI	3.14159	3.1415910000000005	3.1415700000000006

It seems like MPI event though it performs the worst at parallelization, we get a result that is closer to the real value of PI, but with an insignificant degree compared to OpenMP.

CUDA and OpenMP remain the best choices regarding the paralellization and the delivery of valid results.

Algorithm 2: Complement of a DNA sequence

Description of algorithm:

Because of the nature of complementary base pairing, if you know the sequence of one strand of DNA(G, A, T, C), you can predict the sequence of the strand that will pair with, or "complement" it. The rule is simple:

G -> C

A -> T

This operation is very important in bioinformatics because it allows for analysis, like comparing the sequences of two different species.

The need for parallelization is crucial when dealing with DNA sequences because of the huge dimension of files containing this type of information. Manipulating them sequentially, on a single machine is not to be desired, being a very time consuming task.

Pseudocode:

The Algorithm

```
1. Read the initial file
2. Split the file in t smaller files where t is the number of threads that will do the computation
3. Process each file and do the complement
4. Merge the smaller processed files into one single file
5. Terminate.
```

As it can be seen from the pseudo-code above, this algorithm can easily be parallelized due to the fact that the current iteration in which a file is processed doesn’t depend on the previous iterations at all.

Aspects of security and data confidentiality:

All the tests were run on Google Cloud Platform, using the Google Compute Engine.

They offer:

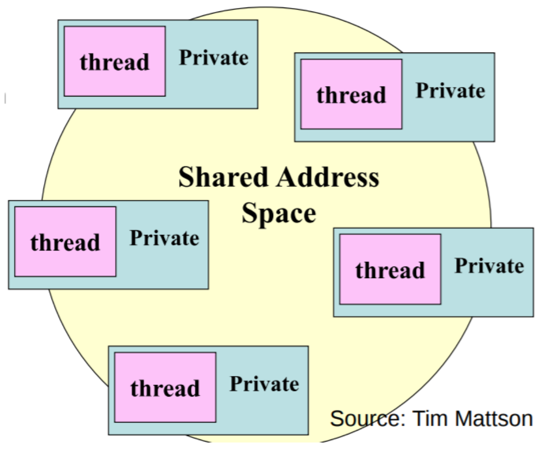
- **Network security**: Help secure the network with products that define and enforce your perimeter and allow for network segmentation, remote access, and DoS defense.
- **Endpoint security**: Help secure endpoints and prevent compromise with device hardening, device management, and patch and vulnerability management.
- **Data security**: Make sensitive data more secure with data discovery, data governance, and controls to prevent loss, leakage, and exfiltration.
- Identity and access management: Manage and secure employee, partner, customer, and other identities, and their access to apps and data, both in the cloud and on-premises.
- **Security monitoring and operations**: Monitor for malicious activity, handle security incidents, and support operational processes that prevent, detect, and respond to threats.

Also, the Google privacy team participates in every product launch, reviewing design documentation and performing code reviews to ensure that privacy requirements are followed. They help ensure that their Google Cloud products and services always reflect strong privacy standards that protect the customers.

Testing scenarios:

1. OpenMP

OpenMP (Open Multi-Processing) is a popular shared-memory programming model. It is supported by popular production C (also Fortran) compilers: Clang, GNU Gcc, IBM xlc, Intel icc.



Main concepts:

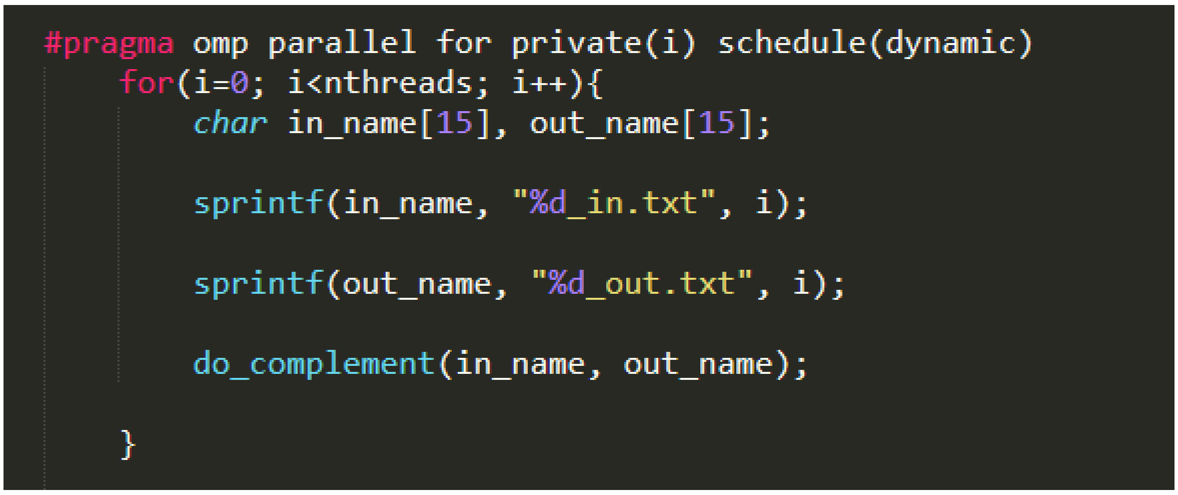
– **Parallel** regions: where parallel execution occurs via multiple concurrently executing threads

– Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution

– Shared and private data: shared variables are the means of communicating data between threads

– **Synchronization**: Fundamental means of coordinating execution of concurrent threads

– Mechanism for automated work distribution across threads



```c
#pragma omp parallel for private(i) schedule(dynamic)
    for(i=0; i<nthreads; i++){
        char in_name[15], out_name[15];

        sprintf(in_name, "%d_in.txt", i);

        sprintf(out_name, "%d_out.txt", i);

        do_complement(in_name, out_name);

    }
```

Parallel region DNA sequence complement with openmp

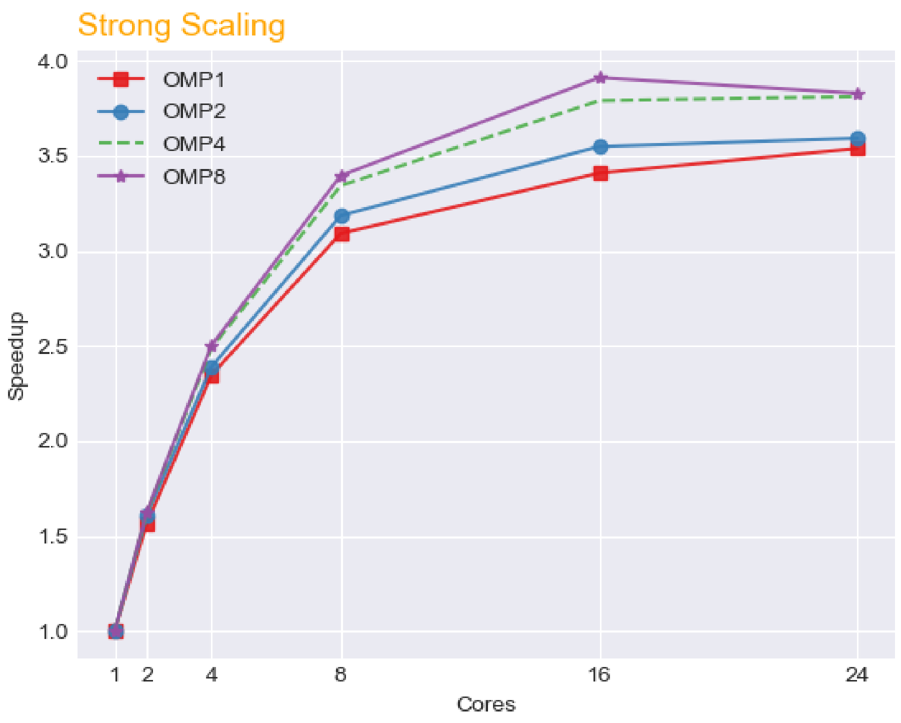
In the above figure pe used the parallel and for regions. Each thread processes a small chunck of the big file that has been splitted before. I used **dynamic** scheduling in order to have a better workload distribution.

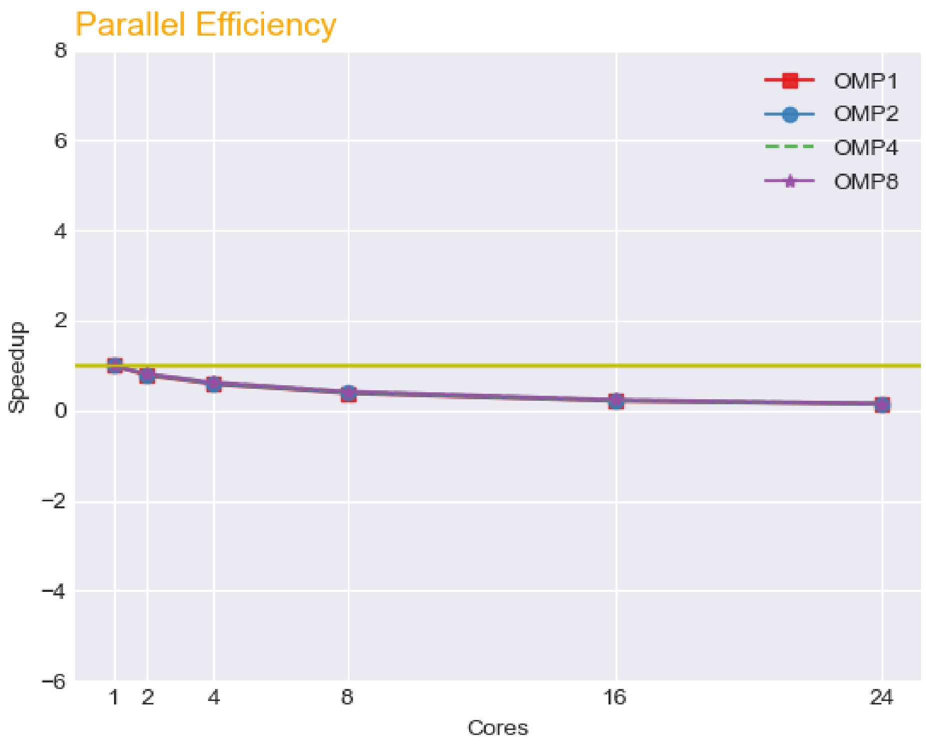
Tested for: 1GB, 2GB, 4GB, 8GB **text files** containing randomly generated sequences of g, c, a, t

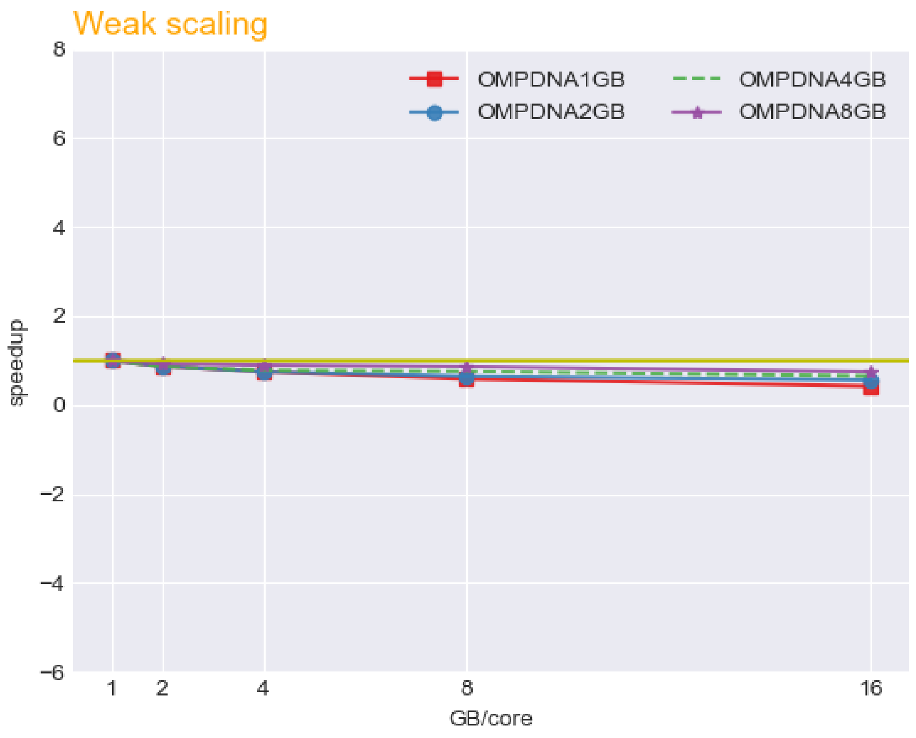
On a **VM** created on Google Cloud, using 1, 2, 4, 8, 16, 24 **threads.**

For each testing scenario the same test has been ran 10 times and the average of executions times were plotted in the next section.

Results:


Strong Scaling
Speedup
Cores
OMP1
OMP2
OMP4
OMP8
1.0
1.5
2.0
2.5
3.0
3.5
4.0
1
2
4
8
16
24


Parallel Efficiency
Speedup
Cores
OMP1
OMP2
OMP4
OMP8
−6
−4
−2
0
2
4
6
8
1
2
4
8
16
24



Discussions:

1. **Strong scaling**

We can see that for OpenMP regarding de strong scaling, „the sweet spot" is at 16 threads after that we can see a slightly decrease on bigger files (2GB, 4GB, 8GB).

2. **Parallel efficiency**

We can see a drop in performance regardless of the file size.

3. **Weak scaling**

We can see that OpenMP scales weakly for larger files, the smallest decrease in performance being on the 8GB file.