Approximation of π : A parallelization algorithm

1. Introduction

In mathematics, we can approximate the famous π using numerical integration by following the *Riemann's* method:

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \tag{1}$$

We approximate π by computing the area of a quarter-circle that is described by the function:

$$f(x) = \sqrt{1 - x^2}$$

where $x_i = i\Delta x$ and $\Delta x = 1/N$.

This approximation becomes fairly accurate as N becomes larger until reaching infinity. The benefits of calculating π in less time are noticeable as it is a crucial part of various formulas. We aim to parallelize the algorithm to make it more performant than the sequential version from a time complexity perspective.

2. Parallel Approach

In this section, we give an overview of the parallelization paradigms used in our solution.

Open MP (3) let us maximally use the multiple cores of nowadays processors, allowing us to have k threads that do the work in parallel. The machine that we used had only four cores, therefore $k = 4$. The threads will independently and simultaneously calculate their summations. These summations are stored locally and later added to the total sum.

Vectorization is the second paradigm that we introduced where operations found in loops are applied in parallel to multiple elements through special vector hardware found in CPUs such as AVX/SSE/MMX for Intel processors (2). The effect of vectorization is a speedup that ideally is proportional to the vector length. Among all the ways that are available to apply vectorization, we decided to use intrinsics functions.

3. Theoretical performance

The time complexity of the sequential algorithms is *O(N)* where N is defined in equation (1). The sequential source code is shown below (3).

```
double approximate_pi(unsigned long long int
    N) {
    double sum = 0;
    for (long long unsigned int i = 0;i <
    N;i++) {
        sum += ((double)((double)sqrt(1.0 -
    ((((double)i) / ((double)N)) *
    (((double)i) / ((double)N))))) /
    ((double)N));
    }
    return 4.0 * sum;
}
```

Regarding the parallel version, we assume that the workload is split between k threads with time complexity of *O(N/k)*. Also, by using vectorization, we will be narrowing it down even further to *O(N/2k)* because we will be using intrinsics functions, namely because *_m128d* has two double numbers. It can be noticed that we have nested loops, but the second loop has a constant number of iterations and is independent from N. The resulting span of the parallel algorithm would then be:

$$t(N,k) = O(2 * N/2k) = O(N/k)$$

By analysing our parallel solution we conclude that the work *w(N,K)* is equal to *O(N)* since all of our k threads have a time complexity of *O(N/k)*, so the total work computed by all the threads is *O(N)*.The cost *c(N)* is defined as *k*t(N,k)* and for would solution is equal to *O(N)*.

For the sequential algorithm instead we know that $t_s = O(N)$ and hence the cost and the work are also equal to O(N).

So we can conclude that is terms of work and cost there is not any difference in weather we pick the parallel or the sequential algorithm.

We will continue with performing strong-scalability analysis, where the problem size N is fixed and the number k of processors varies from 1 to N. As soon as it starts running our algorithm will already have a speedup of k. As it approaches N, the speedup will be the following:

$$\lim_{k\to N}\frac{t_s}{t}=\lim_{k\to N}\frac{O(N)}{O(N/k)}=\lim_{k\to N}\frac{O(N)}{O(N/N)}=O(N)$$

On the other side the weak scaling scales the size N to the amount of k resources used, meaning that the work $\overline{w}$ done by each processor will be kept constant. This means that $N = \overline{w}k$ so that $t_S(N) = O(\overline{w}k)$ grows proportionally to k. For our solution the time spent is kept constant, namely $\overline{w}$, so the scaled speedup is:

$$\lim_{k\to\infty}\frac{t_s}{t}=\lim_{k\to\infty}\frac{O(\overline{w}k)}{O(\overline{w})}=O(\overline{w})=O(1)$$

4. Implementation

The approach described in Section 2 was implemented as below (4). We used the OpenMP API (Reference 3) and SSE instructions (Reference 2). The data were aligned since vectorization works faster if the vector is aligned.

```
#define ALIGNMENT 16
#define NUM_THREADS 4
#define ELEMENTS_IN_VECTOR 2

double approximate_pi(unsigned long long int
    N){
    const double dx = 1.0/(double)N;
    double return_value=0;
    #pragma omp parallel for
    num_threads(NUM_THREADS)
    for(long long unsigned int
    p=0;p<N/ELEMENTS_IN_VECTOR;++p) {
        alignas(ALIGNMENT) double
    x[ELEMENTS_IN_VECTOR];
        for (long long unsigned int i =
    ELEMENTS_IN_VECTOR*p; i <
    ELEMENTS_IN_VECTOR*p+ELEMENTS_IN_VECTOR;
    ++i) {
            x[i-ELEMENTS_IN_VECTOR*p] = 1 - i
    * dx * i * dx;
        }
        __m128d f_xi =
    _mm_sqrt_pd(_mm_load_pd(x));
        __m128d tmp_results =
    _mm_mul_pd(_mm_set1_pd(dx), f_xi);
        tmp_results = _mm_add_pd(tmp_results,
    _mm_move_sd(tmp_results, tmp_results));
        double tmp =
    _mm_cvtsd_f64(tmp_results);
        #pragma omp atomic
        return_value+=tmp;
    }
    return return_value*4;
}
```

The first loop is the one that is parallelized to gain speedup. The second loop is constant and independent and always executes only 2 times to calculate the x_i value.

5. Experiments

We ran some experiments while varying the size of N on a four-core processor with k = 4 threads. As expected, the parallel algorithm performed better. With the increase of N, the results are more noticeable. An extract of the results is shown below (5), the full result is accessible trough the Reference 4, also 2 plots are provided to show the results for time (Figure 1) and correctness of the results (Figure 2) of both, the parallel and the sequential algorithms.

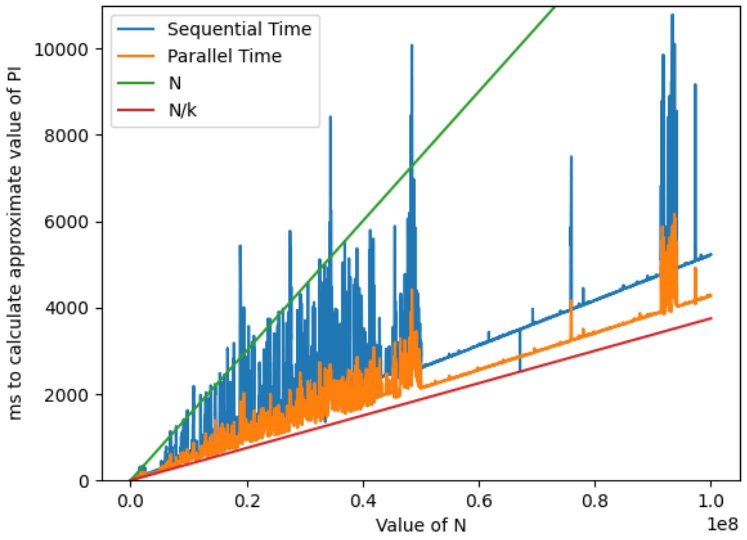

Figure 1: Time taken by the algorithms to give the result

```
with N=32768:
  sequential took: 1 ms and got result:
    3.14165
  parallel took: 1 ms and got result: 3.14171
-----------------------------------------------------
with N=65536:
  sequential took: 8 ms and got result:
    3.14162
  parallel took: 3 ms and got result: 3.14165
-----------------------------------------------------
with N=98304:
  sequential took: 5 ms and got result:
    3.14161
  parallel took: 5 ms and got result: 3.14163
-----------------------------------------------------
with N=131072:
  sequential took: 10 ms and got result:
    3.14161
```

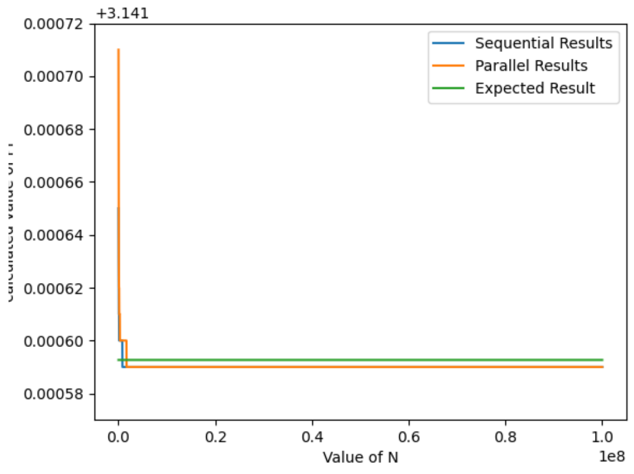

Figure 2: Result given

```
  parallel took: 7 ms and got result: 3.14162
-------------------------------------------------
with N=163840:
  sequential took: 9 ms and got result: 3.1416
  parallel took: 8 ms and got result: 3.14162
-------------------------------------------------
with N=196608:
  sequential took: 10 ms and got result:
    3.1416
  parallel took: 8 ms and got result: 3.14161
-------------------------------------------------
[...]
with N=4227072:
  sequential took: 220 ms and got result:
    3.14159
  parallel took: 181 ms and got result:
    3.14159
-------------------------------------------------
with N=4259840:
  sequential took: 222 ms and got result:
    3.14159
  parallel took: 178 ms and got result:
    3.14159
-------------------------------------------------
with N=4292608:
  sequential took: 227 ms and got result:
    3.14159
  parallel took: 180 ms and got result:
    3.14159
-------------------------------------------------
with N=4325376:
  sequential took: 219 ms and got result:
    3.14159
  parallel took: 180 ms and got result:
    3.14159
-------------------------------------------------
with N=4358144:
  sequential took: 221 ms and got result:
    3.14159
  parallel took: 187 ms and got result:
    3.14159
-------------------------------------------------
with N=4390912:
```

```
  sequential took: 223 ms and got result:
    3.14159
  parallel took: 193 ms and got result:
    3.14159
----------------------------------------------------
[...]
with N=52887552:
  sequential took: 2755 ms and got result:
    3.14159
  parallel took: 2261 ms and got result:
    3.14159
----------------------------------------------------
with N=52920320:
  sequential took: 2762 ms and got result:
    3.14159
  parallel took: 2261 ms and got result:
    3.14159
----------------------------------------------------
with N=52953088:
  sequential took: 2761 ms and got result:
    3.14159
  parallel took: 2255 ms and got result:
    3.14159
----------------------------------------------------
with N=52985856:
  sequential took: 2768 ms and got result:
    3.14159
  parallel took: 2258 ms and got result:
    3.14159
----------------------------------------------------
with N=53018624:
  sequential took: 2766 ms and got result:
    3.14159
  parallel took: 2260 ms and got result:
    3.14159
----------------------------------------------------
[...]
with N=66125824:
  sequential took: 3445 ms and got result:
    3.14159
  parallel took: 2825 ms and got result:
    3.14159
----------------------------------------------------
with N=66158592:
  sequential took: 3446 ms and got result:
    3.14159
  parallel took: 2829 ms and got result:
    3.14159
----------------------------------------------------
with N=66191360:
  sequential took: 3450 ms and got result:
    3.14159
  parallel took: 2823 ms and got result:
    3.14159
----------------------------------------------------
with N=66224128:
  sequential took: 3443 ms and got result:
    3.14159
  parallel took: 2835 ms and got result:
    3.14159
----------------------------------------------------
[...]
with N=75366400:
  sequential took: 3926 ms and got result:
    3.14159
  parallel took: 3208 ms and got result:
    3.14159
----------------------------------------------------
with N=75399168:
  sequential took: 3929 ms and got result:
    3.14159
  parallel took: 3204 ms and got result:
    3.14159
----------------------------------------------------
with N=75431936:
  sequential took: 3931 ms and got result:
    3.14159
  parallel took: 3203 ms and got result:
    3.14159
----------------------------------------------------
with N=75464704:
  sequential took: 3921 ms and got result:
    3.14159
  parallel took: 3229 ms and got result:
    3.14159
----------------------------------------------------
[...]
with N=83460096:
  sequential took: 4355 ms and got result:
    3.14159
  parallel took: 3578 ms and got result:
    3.14159
----------------------------------------------------
with N=83492864:
  sequential took: 4347 ms and got result:
    3.14159
  parallel took: 3553 ms and got result:
    3.14159
----------------------------------------------------
with N=83525632:
  sequential took: 4351 ms and got result:
    3.14159
  parallel took: 3554 ms and got result:
    3.14159
----------------------------------------------------
with N=83558400:
  sequential took: 4356 ms and got result:
    3.14159
  parallel took: 3564 ms and got result:
    3.14159
----------------------------------------------------
[...]
```

6. References

1) Github repository of the project, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/tree/main/Project2, Accessed 2021-01-28
2) Intel SSE instructions, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Accessed 2021-01-28
3) OpenMP API, [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, Accessed 2021-01-21
4) File containing the complete results, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/blob/main/Project2/test_code_output.txt, Accessed 2021-01-28