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. The threads then will put their partial results in a common variable. The resulting span of the parallel algorithm would then be:

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

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 //using sse
#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) {
        __m128d f_xi =
    _mm_sqrt_pd(_mm_set_pd(1 -
    (ELEMENTS_IN_VECTOR * p * dx *
    ELEMENTS_IN_VECTOR * p * dx),

    1 - (ELEMENTS_IN_VECTOR * p + 1) * dx *
    (ELEMENTS_IN_VECTOR * p + 1) * dx));
        __m128d tmp_results =
    _mm_mul_pd(_mm_set1_pd(dx), f_xi);

        //now do horizontal reduce to obtain
    the final value
        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 loop is parallelized to gain speedup. And in the end we need to use the *pragma atomic* to correctly save the data evaluated by the threads.

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.

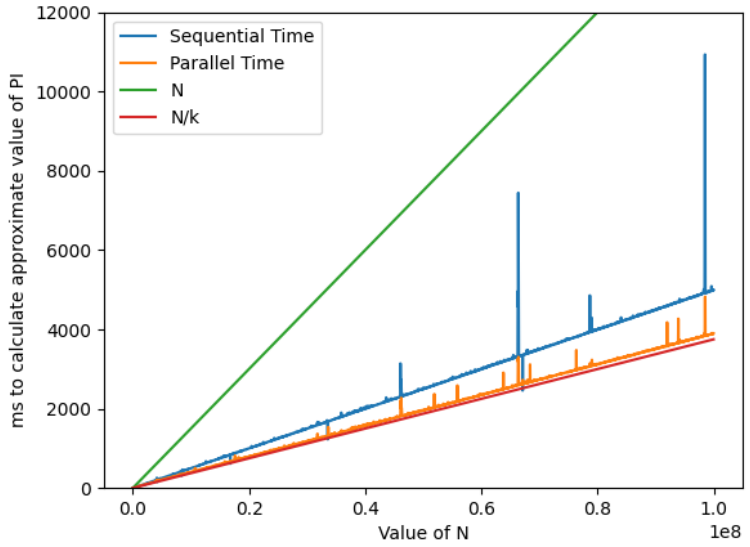

Figure 1: Time taken by the algorithms to give the result

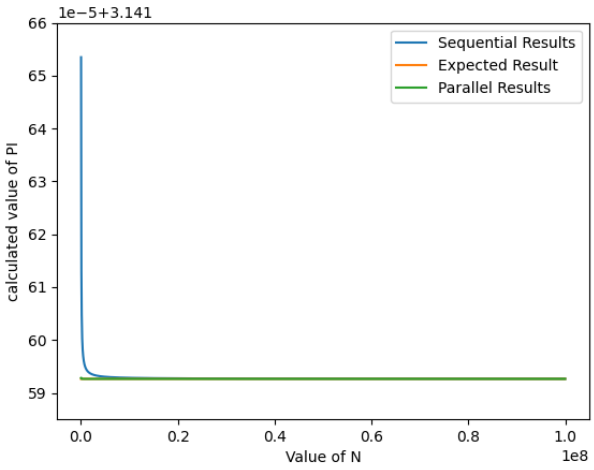

Figure 2: Result given

with N=32768:

```
  sequential took: 1 ms and got result:
    3.1416534904904934
  parallel took: 1 ms and got result:
    3.1415928178295482
-------------------------------------------------
with N=65536:
  sequential took: 2 ms and got result:
    3.1416231010739644
  parallel took: 2 ms and got result:
    3.1415927116574411
-------------------------------------------------
with N=98304:
  sequential took: 4 ms and got result:
    3.1416129604875431
  parallel took: 4 ms and got result:
    3.141592685197828
-------------------------------------------------
with N=131072:
  sequential took: 7 ms and got result:
    3.1416078875968902
  parallel took: 5 ms and got result:
    3.1415926741198241
-------------------------------------------------
with N=163840:
  sequential took: 9 ms and got result:
    3.141604842888515
  parallel took: 6 ms and got result:
    3.1415926682799067
-------------------------------------------------
with N=196608:
  sequential took: 12 ms and got result:
    3.141602812626243
  parallel took: 7 ms and got result:
    3.1415926647649424
-------------------------------------------------
with N=229376:
  sequential took: 13 ms and got result:
    3.1416013621930117
  parallel took: 10 ms and got result:
    3.1415926624579575
-------------------------------------------------
with N=262144:
  sequential took: 11 ms and got result:
    3.1416002742226006
  parallel took: 11 ms and got result:
    3.1415926608482594
-------------------------------------------------
with N=294912:
  sequential took: 15 ms and got result:
    3.1415994279309674
  parallel took: 12 ms and got result:
    3.1415926596727521
-------------------------------------------------
[...]
with N=25952256:
  sequential took: 1297 ms and got result:
    3.1415927306456317
  parallel took: 1027 ms and got result:
    3.1415926535968524
-------------------------------------------------
with N=25985024:
  sequential took: 1301 ms and got result:
    3.1415927305482869
  parallel took: 1025 ms and got result:
    3.1415926535972125
-------------------------------------------------
with N=26017792:
  sequential took: 1296 ms and got result:
    3.1415927304517934
  parallel took: 1029 ms and got result:
    3.141592653597483
-------------------------------------------------
with N=26050560:
  sequential took: 1305 ms and got result:
    3.1415927303555935
  parallel took: 1032 ms and got result:
    3.1415926535972147
-------------------------------------------------
with N=26083328:
  sequential took: 1305 ms and got result:
    3.1415927302591391
  parallel took: 1069 ms and got result:
    3.1415926535970051
-------------------------------------------------
with N=26116096:
  sequential took: 1302 ms and got result:
    3.1415927301626181
  parallel took: 1027 ms and got result:
    3.1415926535973195
-------------------------------------------------
[...]
with N=50888704:
  sequential took: 2541 ms and got result:
    3.1415926928876283
  parallel took: 1994 ms and got result:
    3.1415926535927161
-------------------------------------------------
with N=50921472:
  sequential took: 2543 ms and got result:
    3.1415926928632953
  parallel took: 2046 ms and got result:
    3.1415926535934009
-------------------------------------------------
with N=50954240:
  sequential took: 2545 ms and got result:
    3.1415926928368592
  parallel took: 1997 ms and got result:
    3.1415926535929057
-------------------------------------------------
with N=50987008:
  sequential took: 2544 ms and got result:
    3.1415926928119307
  parallel took: 1990 ms and got result:
    3.1415926535930638
-------------------------------------------------
[...]
with N=55934976:
  sequential took: 2791 ms and got result:
    3.1415926893431951
  parallel took: 2190 ms and got result:
    3.1415926535921042
-------------------------------------------------
with N=55967744:
  sequential took: 2797 ms and got result:
    3.1415926893220525
  parallel took: 2185 ms and got result:
    3.1415926535921987
-------------------------------------------------
with N=56000512:
```

```
  sequential took: 2794 ms and got result:
    3.1415926893010924
  parallel took: 2191 ms and got result:
    3.1415926535921312
-------------------------------------------------
with N=56033280:
  sequential took: 2800 ms and got result:
    3.1415926892796451
  parallel took: 2188 ms and got result:
    3.1415926535908221
-------------------------------------------------
with N=56066048:
  sequential took: 2799 ms and got result:
    3.1415926892591126
  parallel took: 2193 ms and got result:
    3.1415926535921952
-------------------------------------------------
[...]
with N=64126976:
  sequential took: 3206 ms and got result:
    3.1415926847746096
  parallel took: 2507 ms and got result:
    3.1415926535923431
-------------------------------------------------
with N=64159744:
  sequential took: 3216 ms and got result:
    3.1415926847592321
  parallel took: 2511 ms and got result:
    3.141592653591156
-------------------------------------------------
with N=64192512:
  sequential took: 3218 ms and got result:
    3.1415926847441713
  parallel took: 2513 ms and got result:
    3.1415926535913146
-------------------------------------------------
with N=64225280:
  sequential took: 3209 ms and got result:
    3.141592684726711
  parallel took: 2513 ms and got result:
    3.1415926535921179
-------------------------------------------------
with N=64258048:
  sequential took: 3210 ms and got result:
    3.141592684710822
  parallel took: 2512 ms and got result:
    3.1415926535914735
-------------------------------------------------
with N=64290816:
  sequential took: 3218 ms and got result:
    3.1415926846958317
  parallel took: 2509 ms and got result:
    3.1415926535911378
-------------------------------------------------
[...]
with N=77332480:
  sequential took: 3862 ms and got result:
    3.141592679450143
  parallel took: 3020 ms and got result:
    3.1415926535910375
-------------------------------------------------
with N=77365248:
  sequential took: 3861 ms and got result:
    3.1415926794398481
  parallel took: 3012 ms and got result:
    3.1415926535920149
-------------------------------------------------
with N=77398016:
  sequential took: 3867 ms and got result:
    3.141592679428308
  parallel took: 3017 ms and got result:
    3.1415926535911076
-------------------------------------------------
with N=77430784:
  sequential took: 3871 ms and got result:
    3.1415926794171005
  parallel took: 3015 ms and got result:
    3.1415926535903802
-------------------------------------------------
with N=77463552:
  sequential took: 3872 ms and got result:
    3.1415926794079936
  parallel took: 3022 ms and got result:
    3.1415926535913758
-------------------------------------------------
[...]
with N=99844096:
  sequential took: 4992 ms and got result:
    3.1415926736215529
  parallel took: 3889 ms and got result:
    3.1415926535911138
-------------------------------------------------
with N=99876864:
  sequential took: 4999 ms and got result:
    3.1415926736124393
  parallel took: 3896 ms and got result:
    3.1415926535904752
-------------------------------------------------
with N=99909632:
  sequential took: 5003 ms and got result:
    3.1415926736066875
  parallel took: 3891 ms and got result:
    3.1415926535915459
-------------------------------------------------
with N=99942400:
  sequential took: 4980 ms and got result:
    3.141592673598459
  parallel took: 3886 ms and got result:
    3.1415926535902883
-------------------------------------------------
with N=99975168:
  sequential took: 4991 ms and got result:
    3.1415926735947037
  parallel took: 3899 ms and got result:
    3.1415926535898957
-------------------------------------------------
```

6. References

1) Github repository of the project, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/tree/main/Project2, Accessed 2021-02-01
2) Intel SSE instructions, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Accessed 2021-02-01

3) OpenMP API, [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, Accessed 2021-02-01
4) File containing the complete results, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/blob/main/Project2/test_code_output.txt, Accessed 2021-02-01