# Approximation of $\pi$ : A parallelization algorithm

## 1. Introduction

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

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

We approximate $\pi$ 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 $\pi$ 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 (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/4k)* because we will be using intrinsics functions, namely because *_m128* has four float numbers. Also, adding the temporary results to the total sum will require a time complexity of *O(N/4)* for the same reason. The resulting span of the parallel algorithm would then be:

$$O(N/4k) + O(N/4)$$

For the parallel algorithms to run faster we should have that:

$$O(N/4k) + O(N/4) < O(N)$$

By solving the equation we get:

$$k > 1/3$$

Therefore we conclude that if we have more than 1/3 threads working concurrently, our algorithm performs faster, which is always true since we must have at least one thread.

## 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.

```
float approximate_pi(unsigned long long int
    N){
    const float dx = 1.0/(float)N;
    alignas(ALIGNMENT) float out[N/4];

    #pragma omp parallel for
    num_threads(NUM_THREADS)
    for(int p=0;p<N/4;++p) {
        alignas(ALIGNMENT) float x[4];
        for (int i = 4*p; i < 4*p+4; ++i) {
            x[i-4*p] = 1 - i * dx * i * dx;
        }
        __m128 f_xi =
    _mm_sqrt_ps(_mm_load_ps(x));
        __m128 tmp_results =
    _mm_mul_ps(_mm_set1_ps(dx), f_xi);
        tmp_results = _mm_add_ps(tmp_results,
    _mm_movehl_ps(tmp_results, tmp_results));
        tmp_results = _mm_add_ss(tmp_results,
    _mm_shuffle_ps(tmp_results, tmp_results,
    _MM_SHUFFLE(0, 0, 0, 1)));
        out[p] = _mm_cvtss_f32(tmp_results);
    }
    float return_value = 0.0f;
    for (int i = 0; i < N / 4; ++i) {
        return_value += out[i];
    }
    return return_value*4;
}
```

The first loop is the one that is parallelized to gain speedup. Temporary results are calculated by the thread to be added later on to the total sum

## 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.

```
[...]
with N=16384:
  sequential took: 0 ms and got result:
    3.14171
  parallel took: 0 ms and got result: 3.14171
-------------------------------------------------
with N=20480:
  sequential took: 1 ms and got result:
    3.14169
  parallel took: 0 ms and got result: 3.14169
-------------------------------------------------
with N=24576:
  sequential took: 1 ms and got result:
    3.14167
  parallel took: 0 ms and got result: 3.14168
-------------------------------------------------
with N=28672:
  sequential took: 1 ms and got result:
    3.14166
  parallel took: 1 ms and got result: 3.14166
-------------------------------------------------
with N=32768:
  sequential took: 1 ms and got result:
    3.14165
  parallel took: 2 ms and got result: 3.14166
-------------------------------------------------
[...]
with N=114688:
  sequential took: 10 ms and got result:
    3.14161
  parallel took: 4 ms and got result: 3.14161
-------------------------------------------------
with N=118784:
  sequential took: 6 ms and got result:
    3.14161
  parallel took: 3 ms and got result: 3.14161
-------------------------------------------------
with N=122880:
  sequential took: 7 ms and got result:
    3.14161
  parallel took: 3 ms and got result: 3.14161
-------------------------------------------------
with N=126976:
  sequential took: 6 ms and got result:
    3.14161
  parallel took: 5 ms and got result: 3.14161
-------------------------------------------------
[...]
with N=2138112:
  sequential took: 547 ms and got result:
    3.14159
  parallel took: 137 ms and got result:
    3.14163
-------------------------------------------------
with N=2142208:
  sequential took: 601 ms and got result:
    3.14159
  parallel took: 177 ms and got result: 3.1414
-------------------------------------------------
with N=2146304:
  sequential took: 661 ms and got result:
    3.14159
  parallel took: 150 ms and got result:
    3.14102
-------------------------------------------------
with N=2150400:
  sequential took: 535 ms and got result:
    3.14159
  parallel took: 159 ms and got result:
    3.14109
-------------------------------------------------
with N=2154496:
  sequential took: 592 ms and got result:
    3.14159
  parallel took: 131 ms and got result: 3.141
-------------------------------------------------
with N=2158592:
  sequential took: 548 ms and got result:
    3.14159
  parallel took: 157 ms and got result: 3.1412
-------------------------------------------------
with N=2162688:
  sequential took: 606 ms and got result:
    3.14159
```

```
  parallel took: 149 ms and got result:
    3.14152
-------------------------------------------------
[...]
with N=3645440:
  sequential took: 300 ms and got result:
    3.14159
  parallel took: 175 ms and got result:
    3.14129
-------------------------------------------------
with N=3649536:
  sequential took: 217 ms and got result:
    3.14159
  parallel took: 109 ms and got result:
    3.14098
-------------------------------------------------
with N=3653632:
  sequential took: 254 ms and got result:
    3.14159
  parallel took: 161 ms and got result:
    3.14049
-------------------------------------------------
with N=3657728:
  sequential took: 270 ms and got result:
    3.14159
  parallel took: 152 ms and got result:
    3.13972
-------------------------------------------------
with N=3661824:
  sequential took: 276 ms and got result:
    3.14159
  parallel took: 144 ms and got result:
    3.13996
-------------------------------------------------
[...]
with N=4022272:
  sequential took: 474 ms and got result:
    3.14159
  parallel took: 151 ms and got result:
    3.14276
-------------------------------------------------
with N=4026368:
  sequential took: 508 ms and got result:
    3.14159
  parallel took: 157 ms and got result:
    3.14283
-------------------------------------------------
with N=4030464:
  sequential took: 284 ms and got result:
    3.14159
  parallel took: 169 ms and got result:
    3.14301
-------------------------------------------------
[...]
with N=8212480:
  sequential took: 417 ms and got result:
    3.14159
  parallel took: 254 ms and got result: 3.1451
-------------------------------------------------
with N=8216576:
  sequential took: 418 ms and got result:
    3.14159
  parallel took: 255 ms and got result:
    3.14524
-------------------------------------------------
with N=8220672:
  sequential took: 414 ms and got result:
    3.14159
  parallel took: 256 ms and got result:
    3.14529
-------------------------------------------------
with N=8224768:
  sequential took: 416 ms and got result:
    3.14159
  parallel took: 256 ms and got result:
    3.14537
-------------------------------------------------
with N=8228864:
  sequential took: 418 ms and got result:
    3.14159
  parallel took: 254 ms and got result:
    3.14578
-------------------------------------------------
with N=8232960:
  sequential took: 413 ms and got result:
    3.14159
  parallel took: 256 ms and got result:
    3.14617
-------------------------------------------------
with N=8237056:
  sequential took: 419 ms and got result:
    3.14159
  parallel took: 256 ms and got result:
    3.14652
-------------------------------------------------
[...]
```

## 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