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 $\Theta(N)$. 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 *p* processors with time complexity of $\Theta(N/p)$. Also, by using vectorization of a scale v we will narrow it down to $\Theta(N/(p * v))$. Every processor then will horizonatally reduce the results from vectorization into *one* variable which represents the partial sums. This is done in $\Theta(log(v))$. In the end the threads will atomically store their result into the total sum variable in $\Theta(p)$ of time. Hence the resulting span of our algorithm is:

$$t(N, p, v) = \Theta(N/(p * v) + log(v) + p)$$

By analysing our parallel solution we conclude that the work *w(N,p,v)*=$\Theta(N + pv * log(v) + p^2v)$. That is because every processor computes $pv * (N/pv)$ in the first phase or $\Theta(N)$, in the second phase $\Theta(pv * log(v))$, and in the third phase $\Theta(p^2v)$. The cost is defined as *pv*t(N,p,v)* and for would solution is equal to $\Theta(N + pv * log(v) + p^2v)$. For the sequential algorithm instead we know that $t_s = \Theta(N)$ and hence the cost and the work are also equal to $\Theta(N)$.

So we can conclude that our solution is both cost and work-optimal when we choose a level of parallelism of $pv(log(v) + p) < N$

We will continue with performing strong-scalability analysis, where the problem size *N* is fixed and the degree of parallelism pv varies from

1 to N. As soon as it starts running our algorithm will already have a speedup of k. As soon as $pv(log(v) + p) > N$, the running time of the second and third phase will become dominant and when approaches pv=N, the speedup will be the following:

$$\lim_{pv\to N} \frac{t_s}{t} = \lim_{pv\to N} \frac{\Theta(N)}{\Theta(N/(p*v) + log(v) + p)}$$

$$= \lim_{pv\to N} \frac{\Theta(N)}{\Theta(N/N + log(v) + p)}$$

$$= \frac{\Theta(N)}{\Theta(log(v) + p)}$$

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

$$\lim_{pv\to\infty} \frac{t_s}{t} = \lim_{pv\to\infty} \frac{\Theta(\overline{w}pv)}{\Theta(\overline{w} + log(v) + p)}$$

$$= \Theta(\overline{w}) = \Theta(1)$$

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).

```
#define NUM_THREADS 4
#define ELEMENTS_IN_VECTOR 2

double approximate_pi(unsigned long long int
    N){
    double return_value=0;
    #pragma omp parallel
    num_threads(NUM_THREADS)
    {
        __m128d dx = _mm_set1_pd(1.0 /
    (double) N);
        __m128d base = _mm_set_pd(0.0, 1.0);
        __m128d sum = _mm_set1_pd(0.0);
        __m128d one = _mm_set1_pd(1.0);
        __m128d elements_in_vector =
    _mm_set1_pd((double)ELEMENTS_IN_VECTOR);
        #pragma omp for nowait
        for (long long unsigned int p = 0; p
    < N / ELEMENTS_IN_VECTOR; ++p) {
            __m128d i = _mm_add_pd(base,
    _mm_mul_pd(elements_in_vector,
    _mm_set1_pd((double)p)));
            __m128d tmp_results =
    _mm_mul_pd(i, dx);
            tmp_results =
    _mm_mul_pd(tmp_results, tmp_results);
            tmp_results = _mm_sub_pd(one,
    tmp_results);
            tmp_results =
    _mm_sqrt_pd(tmp_results);
            tmp_results = _mm_mul_pd(dx,
    tmp_results);
            sum = _mm_add_pd(sum,
    tmp_results);
        }
        sum = _mm_add_pd(sum,
    _mm_move_sd(sum, sum));
        double tmp = _mm_cvtsd_f64(sum);
        #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.

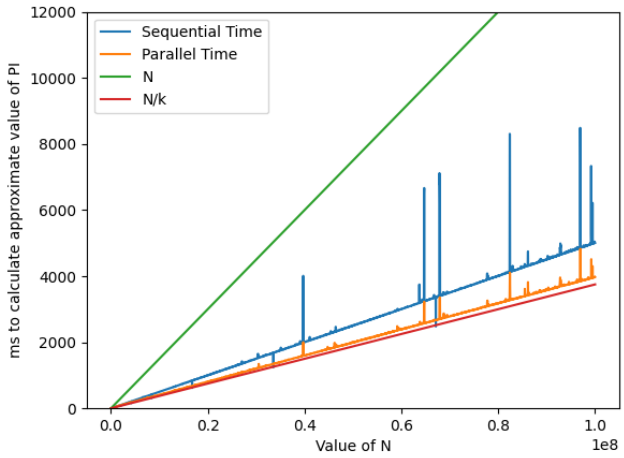

Figure 1: Time taken by the algorithms to give the result

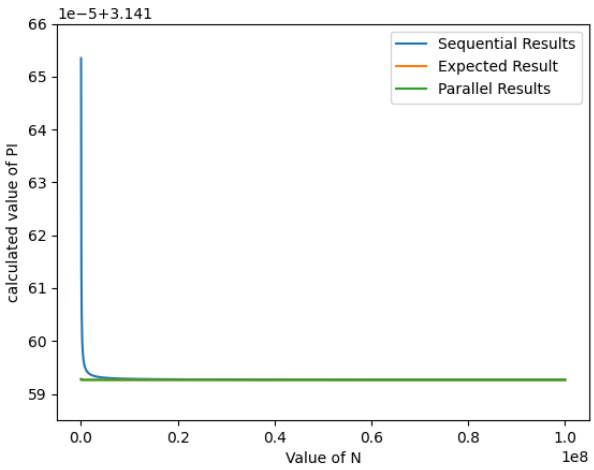

Figure 2: Result given

```
with N=32768:
  sequential took: 1 ms and got result:
    3.1416534904904934
  parallel took: 1 ms and got result:
    3.1415928178295633
-------------------------------------------------
with N=65536:
  sequential took: 2 ms and got result:
    3.1416231010739644
  parallel took: 2 ms and got result:
    3.1415927116574371
-------------------------------------------------
with N=98304:
  sequential took: 5 ms and got result:
    3.1416129604875431
  parallel took: 4 ms and got result:
    3.1415926851978369
-------------------------------------------------
with N=131072:
  sequential took: 5 ms and got result:
    3.1416078875968902
  parallel took: 6 ms and got result:
    3.1415926741198432
-------------------------------------------------
with N=163840:
  sequential took: 8 ms and got result:
    3.141604842888515
  parallel took: 6 ms and got result:
    3.1415926682798996
-------------------------------------------------
with N=196608:
  sequential took: 10 ms and got result:
    3.141602812626243
  parallel took: 8 ms and got result:
    3.1415926647649375
-------------------------------------------------
with N=229376:
  sequential took: 12 ms and got result:
    3.1416013621930117
  parallel took: 10 ms and got result:
    3.1415926624579522
-------------------------------------------------
with N=262144:
  sequential took: 11 ms and got result:
    3.1416002742226006
  parallel took: 10 ms and got result:
    3.1415926608482692
-------------------------------------------------
with N=294912:
  sequential took: 16 ms and got result:
    3.1415994279309674
  parallel took: 12 ms and got result:
    3.1415926596727597
-------------------------------------------------
[...]
with N=17793024:
  sequential took: 889 ms and got result:
    3.1415927659774976
  parallel took: 717 ms and got result:
    3.1415926536027396
-------------------------------------------------
with N=17825792:
  sequential took: 894 ms and got result:
    3.1415927657716729
  parallel took: 723 ms and got result:
    3.1415926536027325
-------------------------------------------------
with N=17858560:
  sequential took: 892 ms and got result:
    3.1415927655650395
  parallel took: 721 ms and got result:
    3.141592653602677
-------------------------------------------------
with N=17891328:
  sequential took: 896 ms and got result:
    3.1415927653602114
  parallel took: 724 ms and got result:
    3.1415926536026757
-------------------------------------------------
with N=17924096:
  sequential took: 895 ms and got result:
    3.14159276515639
  parallel took: 723 ms and got result:
    3.1415926536026695
-------------------------------------------------
with N=17956864:
  sequential took: 901 ms and got result:
    3.1415927649520605
  parallel took: 728 ms and got result:
    3.1415926536025678
-------------------------------------------------
[...]
with N=55672832:
  sequential took: 2787 ms and got result:
    3.1415926895115422
  parallel took: 2221 ms and got result:
    3.1415926535921104
-------------------------------------------------
with N=55705600:
  sequential took: 2786 ms and got result:
    3.1415926894901856
  parallel took: 2218 ms and got result:
    3.1415926535922107
-------------------------------------------------
with N=55738368:
  sequential took: 2786 ms and got result:
    3.1415926894692334
```

```
  parallel took: 2222 ms and got result:
    3.1415926535921788
-------------------------------------------------
with N=55771136:
  sequential took: 2794 ms and got result:
    3.1415926894474624
  parallel took: 2220 ms and got result:
    3.1415926535924878
-------------------------------------------------
with N=55803904:
  sequential took: 2804 ms and got result:
    3.1415926894263699
  parallel took: 2230 ms and got result:
    3.1415926535922689
-------------------------------------------------
with N=55836672:
  sequential took: 2797 ms and got result:
    3.1415926894061204
  parallel took: 2219 ms and got result:
    3.1415926535920402
-------------------------------------------------
[...]
with N=99778560:
  sequential took: 5004 ms and got result:
    3.1415926736327489
  parallel took: 3962 ms and got result:
    3.1415926535908665
-------------------------------------------------
with N=99811328:
  sequential took: 5000 ms and got result:
    3.1415926736268336
  parallel took: 3967 ms and got result:
    3.1415926535905632
-------------------------------------------------
with N=99844096:
  sequential took: 5007 ms and got result:
    3.1415926736215529
  parallel took: 3967 ms and got result:
    3.1415926535906982
-------------------------------------------------
with N=99876864:
  sequential took: 5045 ms and got result:
    3.1415926736124393
  parallel took: 3971 ms and got result:
    3.141592653590918
-------------------------------------------------
with N=99909632:
  sequential took: 4998 ms and got result:
    3.1415926736066875
  parallel took: 3969 ms and got result:
    3.1415926535909069
-------------------------------------------------
with N=99942400:
  sequential took: 5007 ms and got result:
    3.141592673598459
  parallel took: 3976 ms and got result:
    3.1415926535906147
-------------------------------------------------
with N=99975168:
  sequential took: 5005 ms and got result:
    3.1415926735947037
  parallel took: 3969 ms and got result:
    3.1415926535909149
-------------------------------------------------
with N=100007936:
  sequential took: 5007 ms and got result:
    3.141592673587513
  parallel took: 3973 ms and got result:
    3.1415926535908949
-------------------------------------------------
```

6. References

1) Github repository of the project, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/tree/main/Project2, Accessed 2021-02-03
2) Intel SSE instructions, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/, Accessed 2021-02-03
3) OpenMP API, [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, Accessed 2021-02-03
4) File containing the complete results, [Online]. Available: https://github.com/HarlockOfficial/dva336_labs/blob/main/Project2/test_code_output.txt, Accessed 2021-02-03