Parallel Leibniz Series

Zhiheng Yang* and Hang Xu*

Informatics Institute, University of Amsterdam

1 Background

In this report we are going to analysis the performance of a parallel OpenMP program, which calculates π based on leibniz series. Original leibniz series algorithm is described as follow:

Algorithm 1: Calculate π using leibniz series

Data: *niter*
Result: *pi*
$pi \leftarrow 0.0$;
for $i \leftarrow 0$ **to** $niter - 1$ **do**
  $pi \leftarrow pi + \text{pow}(-1, i) \times \left(\frac{4}{2 \times i + 1}\right)$;
end

It's a sequential for loop iteratively calculates the value of π(pi) over the specified number of iterations(niter), where each iteration contributes a term alternating in sign and inversely proportional to odd integers.

Algorithm 2: Parallel calculating leibniz series

Data: *niter*
Result: *pi*
$pi \leftarrow 0.0$;
#pragma omp parallel for reduction(+ : *pi*);
for $i \leftarrow 0$ **to** $niter - 1$ **do**
  $pi \leftarrow pi + \text{pow}(-1, i) \times \left(\frac{4}{2 \times i + 1}\right)$;
end

Above algorithm 2 describes the parallel implementation uses a `#pragma omp parallel for` directive to distribute original for loop across multiple threads to perform parallel execution, and perform reduction using `reduction(+ : pi)` to accumulate calculated pi when threads finished execution.

2 Experiment Setup

We conducted two experiments on Snellius to explore the computation with varying number of thread and iteration(workload). All job is running on Snellius partition `thin_course`, with 1 node reserved and 32 CPU cores, using 2022 software stack.
Partition is reserved with command: `#SBATCH --partition=thin_course`.
Program is compiled using `GCC 11.3.0` with command: `gcc -fopenmp -o pi pi.c -lm`.
The number of running threads is configured by exporting the environment variable `OMP_NUM_THREADS` for the submitted job.

3 Experiment Result

3.1 Multi Thread

In this section we are going to discuss the result of experiment with different number of thread. Total iteration *niter* is set to constant of 1000000000, and we ran several experiment with thread amount of $\{1, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48\}$. Experiment was repeated 20 times for robust data, result is report with min-max ranges and mean.
We use speedup S to measure the relative performance of parallel program comparing to sequential program, and parallel efficiency E to measure the efficiency of performance improvement by introducing additional threads. Where t_{seq}:sequential execution time, t_{par} parallel execution time, n_{thread}: amount of thread. Full experiment result see appendix table 1.

$$S = \frac{t_{seq}}{t_{par}}, E = \frac{S}{n_{thread}}$$

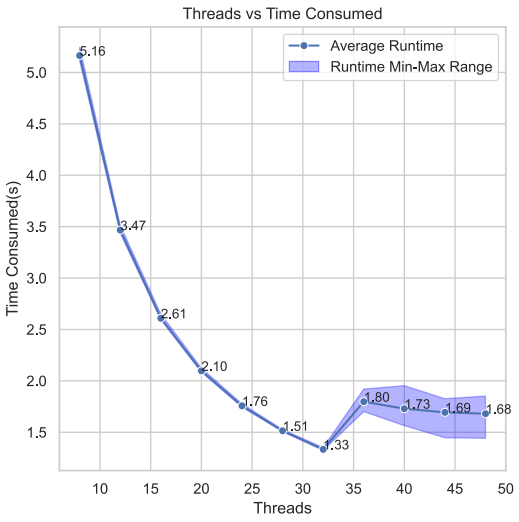

Fig. 1: Runtime with multi-threading

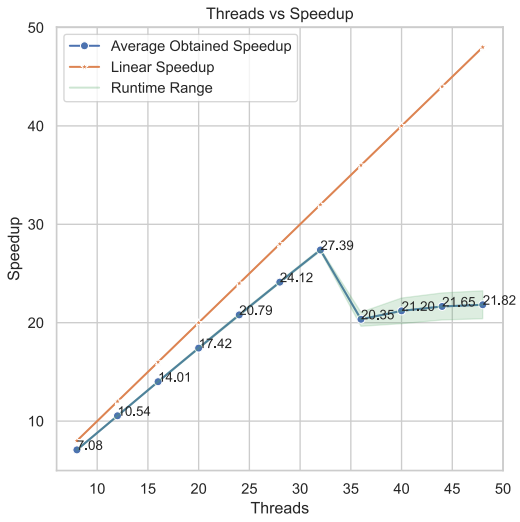

Fig. 2: Speedup with multi-threading

From the result we can see that parallel execution time significantly reduced comparing to sequential execution. This is because each thread concurrently executes a portion of the iterations, and each concurrent execution time is proportional to sequential execution, thereby reducing the overall execution time.

As shown in figure 3, with *niter* iteration running sequentially, `omp parallel for` distribute equal amount of $niter/n$ iteration to n threads and execute concurrently. Because majority of the computation is arithmetic operation and accessing local variable, reduction of *pi* is after parallel execution, there is no shared memory competition, so we obtained near-linear speedup by adding threads until 32. We cannot achieve linear speedup(as seen in Figure 2) due to the overhead introduced by `OpenMP` coordinating between

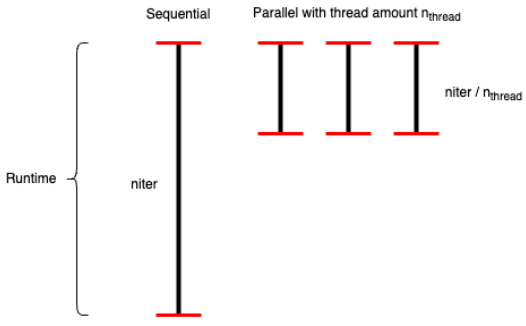

Fig. 3: Iteration Distribution

threads and the uneven execution times of threads, `OpenMP` needs to wait for the thread with the longest execution time to finish before exiting the parallel region.
Furthermore, the effectiveness of using more threads decreases (Figure 4) after the number of threads exceeds 32. This is because each submitted job reserves 32 physical cores, when the number of threads exceeds the available physical cores, more than one thread runs on a single core. Note that it is the physical core that executes instructions, adding more threads would introduce extra context-switching overhead while the core is already busy executing instructions. Figure 5 illustrates the efficiency with 48 cores, showcasing a notable improvement for thread numbers greater than 32. However, a gradual decline in efficiency is still observed, caused by finer job granularity for each thread and increased prominence of parallel overhead.

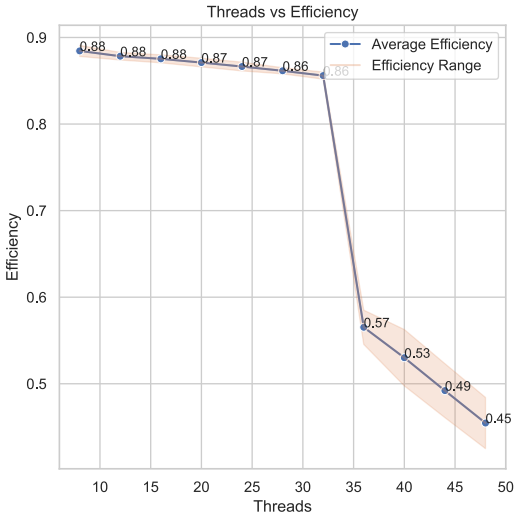

Fig. 4: Efficiency with 32 cores

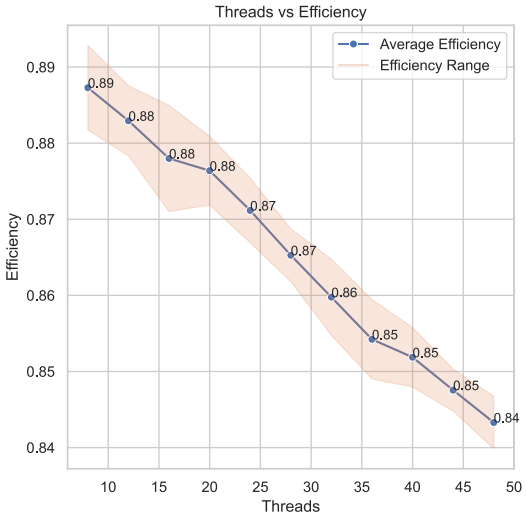

Fig. 5: Efficiency with 48 cores

3.2 Multi Workload

In this section we are going to discuss the result of experiment with different size of workload. Experiment was performed with iteration *niter* of $\{31250000, 62500000, 125000000, 250000000, 500000000, 1000000000, 2000000000\}$ and thread number of $\{1, 32\}$. Program was repeated 20 times for better performance accuracy, result is report with min-max and mean.
Figure 1 and Figure 7 show the experiment result of runtime and speedup respectively. We observe that the runtime scales near-linearly with the problem size, but employing speedup provides us with deeper insights into the performance. The comparison of speedup among various problem sizes was conducted using the mean runtime when executing with 1 thread and 32 threads.
In Figure 7 the orange line represents the ideal speedup of 32, while the blue line depicts the achieved speedup. A closer proximity of the blue line to the orange line indicates improved efficiency from multi-threading. It is evident that, with larger problem sizes, the program attains higher speedup and better efficiency. As problems size grows bigger, each thread gets more coarse-grain job. This reduces the

ratio of overhead introduced by using `OpenMP` and job execution time, making the benefits of employing parallelism more significant.

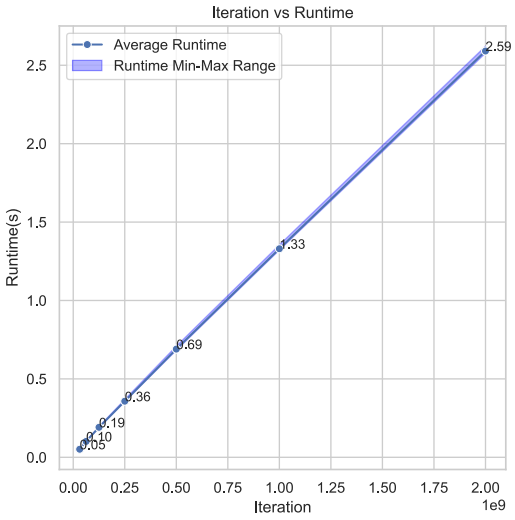

Fig. 6: Runtime with Various Iteration

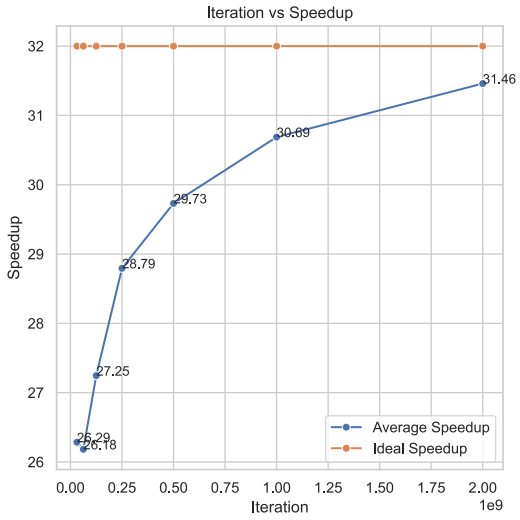

Fig. 7: Speedup with Various Iteration

4 Appendix

Table 1: 1000000000 iteration with different amount of thread

Thread Amount	**Run Time(s)**	**Speedup**	**Efficiency**
1	36.537154	1	1
8	5.163905	7.075489	0.884436
12	3.466438	10.540260	0.878355
16	2.608874	14.004953	0.875310
20	2.097447	17.419823	0.870991
24	1.757137	20.793570	0.866399
28	1.514647	24.122551	0.861520
32	1.333962	27.389960	0.855936
36	1.797663	20.324809	0.564578
40	1.729339	21.127809	0.528195
44	1.693780	21.571373	0.490258
48	1.680643	21.739989	0.452916

Table 2: 32 thread with different size of iteration

Thread Amount	**Run Time(s)**	**Speedup**	**Efficiency**
31250000	0.051399	26.285562	0.821424
62500000	0.101264	26.182658	0.818208
125000000	0.191414	27.245639	0.851426
250000000	0.357741	28.794448	0.899827
500000000	0.689985	29.731185	0.929100
1000000000	1.329853	30.687796	0.958994
2000000000	2.590814	31.460024	0.983126