**Parallel Programming Exercise 5 – 11**

|  |  |
| --- | --- |
| **Author:** | 蕭彧 ([r11922138@ntu.edu.tw](mailto:r11922138@ntu.edu.tw)) |
| **Student ID** | R11922138 |
| **Department** | Computer Science |

# Problem and Proposed Approach

This programming exercise is to calculate harmonic progression with arbitrary precision. Calculate harmonic progression itself is simple, we can use the cyclic allocation to let processors calculate partial sum and use one MPI\_Gather to accumulate these partial sums. The difficult part here is to achieve arbitrary precision. I use a GNU library called mpfr (multiple-precision floating-point computations with correct rounding). It supports arbitrary precision so that we can calculate harmonic progression for designated precision.

My approach is to use mpfr to calculate partial sums in each processor, and use MPI\_Gather to gather these data to processor 0. To send mpfr instance between processors, we need to do some serialization. MPFR has built-in function to transform mpfr instance to char array.



Also support transform char array back to mpfr instance.



Using these functions, we can exchange mpfr instances between processors. After gathering to processor 0, processor 0 just need to sum all these partial sums and we can get the final result.

The 1000000th harmonic progression with 100 digits of precision calculated by mpfr is 14.3927267228657236313811274931949395277963270695860320913581195580860372373876998608466237783432006835.

# Theoretical Analysis Model

mpfr\_d is time needed to do mpfr divide

mpft\_a is time needed to do mpfr add

λ is the message latency

β is network bandwidth

Sequential execution time: (n - 1)(mpft\_a + mpfr\_d)

Reduction time: (*λ* + d/β) ⎡log *p*⎤

Expected execution time: ⎡(n - 1)(mpft\_a + mpfr\_d)/p⎤ + (*λ* + d/β) ⎡log *p*⎤ + (mpft\_a) ⎡log *p*⎤

# Performance Benchmark

I pick process 0’s execution time, since it need to do the reduction. The message latency is λ = 0.001521 sec, and the network bandwidth is β = 1658 MB/sec

Table . The execution time

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Processors | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| Real execution time | 0.200204 | 0.100339 | 0.05124 | 0.028984 | 0.018675 | 0.010426 | 0.009566 | 0.009766 |
| Estimate execution time | 0.199967 | 0.099996 | 0.05001 | 0.026231 | 0.01654 | 0.00876 | 0.00896 | 0.00956 |
| Speedup | 1 | 1.995 | 3.9 | 6.9 | 10.72 | 19.2 | 20.92 | 20.5 |
| Karp-flatt metrics | - | 0.002506 | 0.008547 | 0.02277 | 0.03283 | 0.0215 | 0.03268 | 0.04129 |

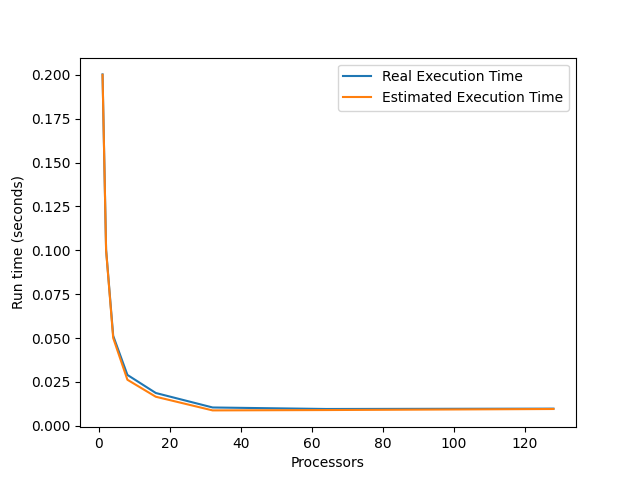


Figure . The performance of diagram

# Conclusion and Discussion

The concept of this problem is quite same as the previous pi calculation, but harmonic progression here. Each processor calculates the partial sum and then do a gather to get final output. The obstacle this time is how to cope with the arbitrary precision. I spend most of the time learning the api of mpfr and how to transfer mpfr instance between processors.

In the performance figure, we can see we did have good parallelism compared to previous problem. This is mainly because the problem size is bigger this time, so we can see nearly two-fold speedup when we double the processor amounts. Network latency doesn’t play an important role as before.