Parallel Computing System

Exercise 7

November 26 2019

s1250250

Phea Sinchhean

Computer Specifications

CPU	Intel(r) Core(tm) i5-6500 CPU @ 3.20GHz × 4
Memory	8GB
Base System	Oracle Solaris 11.4 64-bit
GCC version	9.1.0
L1 Cache	256 KB
L2 Cache	1 MB
L3 Cache	6 MB

Source Code Outline

The program try to calculate the value of PI. After we get the result, we compare our PI value with 3.14159265358979323846264 3.
The program used MPI library for parallelism. Processor with ID=0 is considered the main process while others are considered as work processes.
The number of processes used to run can be specified using "-np" followed by the number of processes.

MPI Broadcast is used to distribute "h" value, start and end indexes for summation calculation to each processes.
MPI Gather is used to gather all partial sums from each process to calculate the total summation.

The parameter for the program is x which is used to multiply n=1000 (The default value). In theory, the higher the n the higher the accuracy our pi is.
E.g.
Compilation for MPI in c programming

```
--------------------------------------------------------------------------
>mpicc -o p1_2 p1_2_c
--------------------------------------------------------------------------
```

Running the program

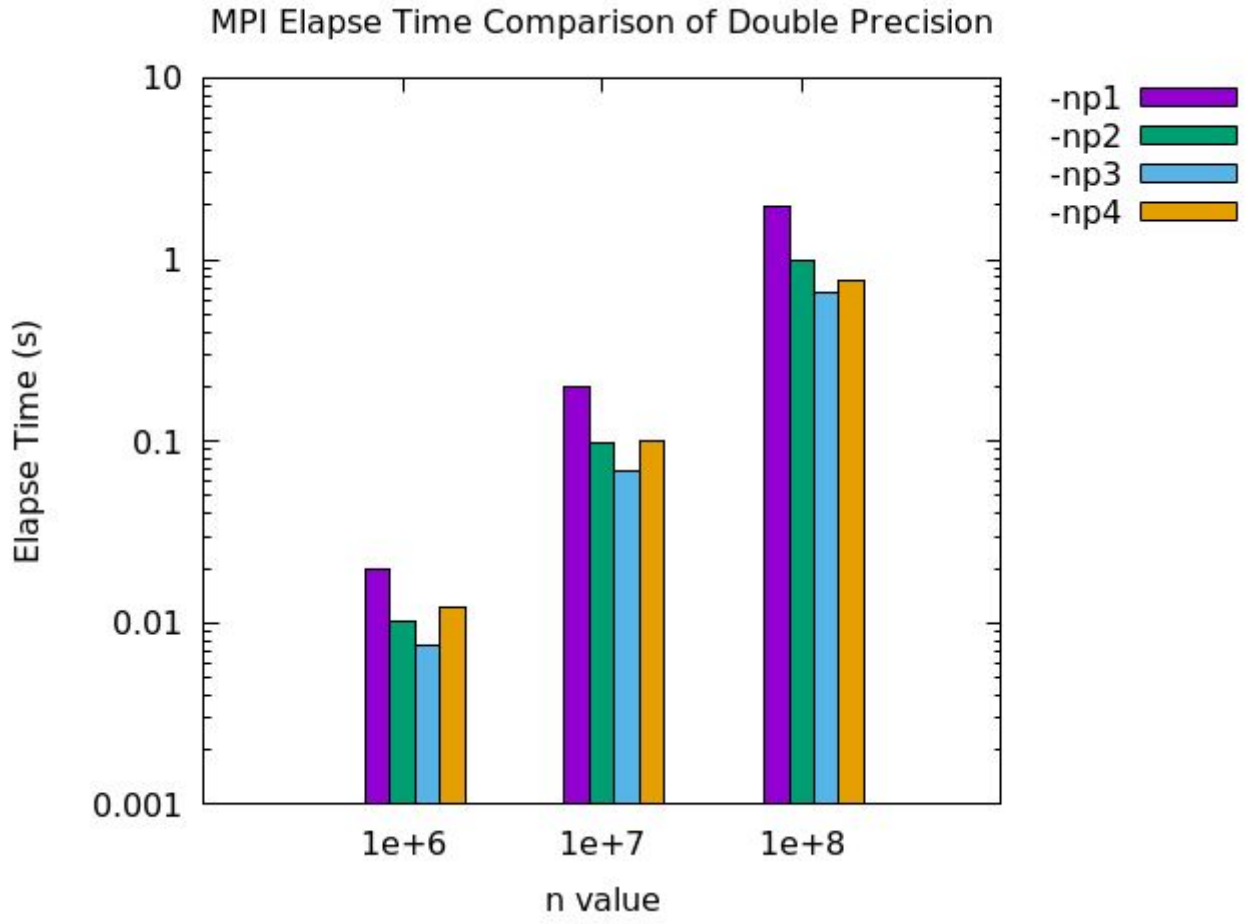
```
--------------------------------------------------------------------------
>mpirun -np 2 ./p1_2 1000
pi is approximately 3.1415946535888990, Error is 0.0000006366194878
wall clock time = 0.010185
--------------------------------------------------------------------------
```

Note: This command runs with 2 processes and n = 1000*1000.

Result

MPI Elapsed Time in second of Double precision based on number of processes and value of n.

n value	-np 1	-np 2	-np 3	-np 4
1.00E+06	0.019606	0.010185	0.007561	0.012183
1.00E+07	0.200901	0.097755	0.067924	0.099334
1.00E+08	1.952215	0.997775	0.65843	0.760592



Numerical Errors (-np 4)

n value	Double Error Rate (%)	Single Error Rate (%)
1.00E+03	0.06363275904184	0.06365024963284
1.00E+04	0.00636590589974	0.00635136162478
1.00E+05	0.00063661685449	0.00053225901799
1.00E+06	0.00006366194803	0.00447478570238
1.00E+07	0.00000636619347	0.57493016158980
1.00E+08	0.00000063661623	14.55434107223956

Double:
The higher the n value the lower the error rate. This follows the theory of the mathematics calculation. Double precision uses 64 bits to store data; in that 52 bits used for fraction part. It can contain 15-18 significant digits, typically 16. As a result, double precision can store and calculate precisely on data and has no information loss error in this case. So, in this case calculation on the partial sums gets us the desired result.

Single:
When n is between 1000 and 100000, we can see the increase in accuracy as n gets higher but after that the higher the n value the higher the error rate. This is due to the fact that Single precision uses only 32 bits to store data and only 23 bits for the fraction part. It can contain 6-9 significant digits, typically 7. This causes it to be able to work only a small range of numbers. If more than 7 digits are present, value is rounded off. In this case when n is smaller we can get accurate result with no problem but when n is too big, the number got rounded off and has information loss error when doing summation. When debugging the program, we can see that the partial sums could only increase upto 6.7108864000000000e+07 after that it ignores the much smaller numbers adding onto it, which leads to higher error rate contrary to the theory.