Optimizing Program Performance: Analyzing Minimum Data Size for Parallelization on a Four-Core System for the Histogram program

Exploration of parallelism

2023

Kai Koo

Introduction

In order to achieve optimal program performance, it is often necessary to consider parallelization, a technique that allows multiple processes to be executed simultaneously. However, determining the appropriate data size for parallelization can be a complex and challenging task. This analysis aims to provide a concise report on the results obtained from the investigation of the minimum data size required for effective parallelization of a program on the company's four-core system. The paper will present a brief statement of the objective, a concise description of the methodology used, the primary results obtained, and a conclusion drawn from the analysis. Ultimately, the findings of this analysis paper will help in the decision of whether this particular program is a useful addition to the company's software base.

Methodology

The histogram program, which achieved parallelisation through the MPI library, was analyzed via taking execution run-times of the program, while varying dataset size as well as number of processes. Timing was taken using the MPI function MPI_Wtime() which returns the current wall-clock time, hence calling MPI_Wtime() twice, once at the beginning of the code section to be timed, and then again after the code section to be timed; allowed us to get a *start* and *finish* time. The *elapsed* (run-time of parallelised part of code) was calculated via $finish - start$. These *elapsed* times were used to determine efficiency and speed up of the program. Bash scripts were used to run the program with constant *bin_number*, *min_measurment*, and *max_measurment*, only varying the *data_count* (doubling from 1024 till 16384). Three scripts were used - 1 process, 2 process, and 4 processes; one script for each. Each script was run five times to flatten variance in run-time due to outside influence from other users/programs running on the system and the minimum run-time was taken for runs of 1, 2 and 4 processes at each *data_count* size. The histogram program was modified such that raw run-time data was output to text files for convenient data collection. There is a text file for each number of processes in the "perfdata" folder; where runtimep1.txt holds data for 1 process, runtimep2.txt for 2 processes, and runtimep4.txt for 4 processes.

```
MPI_Barrier(comm);
local_start = MPI_Wtime();

Find_bins(bin_counts, local_data, loc_bin_cts, local_data_count,
    bin_maxes, bin_count, min_meas, comm);

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;

MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
```

Figure 1. Taking the run-time in Histogram.c

In **Figure 1**, MPI_Barrier() is used to ensure all processes are at the same point before the timing commences. Because Find_bins() executes the code in which most of the parallelization takes place, we time this section. Each process will calculate their *local_elapsed* time, before MPI_Reduce() is called, to collect every process' *local_elapsed* time and return it to process 0 to be output. It is important to note that we use **MPI_MAX** in MPI_Reduce(), this is because we are only concerned with the largest *local_elapsed* time (the time of the slowest process) as the program will run for as long as its slowest process does. Speed up of the parallelized histogram program was calculated using $speedup = \frac{Tserial}{Tparallel}$ and efficiency was calculated using $efficiency = \frac{speedup}{number\ of\ processes}$. These results were used to determine whether parallelization of the histogram program was worthwhile when run on the company's four core system.

Results

Data set size	1 process	2 processes	4 processes
1024	0.0365ms	0.0193ms	0.0144ms
2048	0.0718ms	0.0372ms	0.0200ms
4096	0.142ms	0.0722ms	0.0374ms
8192	0.282ms	0.142ms	0.0744ms
16384	0.563ms	0.283ms	0.145ms

Figure 2. Run- Times in milliseconds of Serial and Parallel Histogram Program

As seen in **Figure 2,** run-time does improve as more processes are run using MPI, however this improvement in run-time is less apparent at lower data set sizes, than it is at larger data set sizes. The difference between 2 processes and 4 processes for a data set size of 1024 is only 4.9 microseconds; where as at a data set size of 16384, the difference between 1 process and 2 processes is a reduction in run-time by almost half (0.563ms to 0.283ms) and then from 2 processes to 4 processes is almost half again (0.283ms to 0.145ms).

comm_sz	1024	2048	4096	8192	16384
1	1.00	1.00	1.00	1.00	1.00
2	1.89	1.93	1.97	1.99	1.99
4	2.54	3.59	3.79	3.80	3.90

Figure 3. Speedup of Parallel Histogram Program

The speedup of running the program at 1, 2 and 4 processes with varying data set sizes is also clearly shown in **Figure 3**, following a similar pattern as **Figure 2**, increasing the number of processes at lower data set sizes resulted in less apparent speed up (also see **Appendix 2**). This is especially true when running 4 processes, which at data set size 1024, only achieved a speed up of 2.54x, despite having 4x more processes than running with 1 process. Running the program with 2 processes achieved very reasonable speedup, even at the smallest data set size, with 1.89x speed up at only 1024 data set size. Eventually both 2 and 4 processes even out to almost 2x and 4x speed up respectively; at 16384 data set size, 2 processes achieved 1.99x speed up, and 4 processes achieved a 3.90x speed up.

comm_sz	1024	2048	4096	8192	16384
1	1.00	1.00	1.00	1.00	1.00
2	0.95	0.97	0.99	0.99	0.99
4	0.64	0.89	0.95	0.95	0.97

Figure 4. Efficiencies of Parallel Histogram Program

The efficiencies are shown in **Figure 4**, parallelization with 2 processes is immediately strongly beneficial with an efficiency of 0.95 even at the smallest data set size. Parallelization with 4 processes however, performs poorly at 1024 data set size, achieving only a 0.64 efficiency of parallelization (also see **Appendix 3**). This is likely due to the overhead of communication between 4 processes not being offset by the small computation time, leading to diminishing returns on the additional processes added.

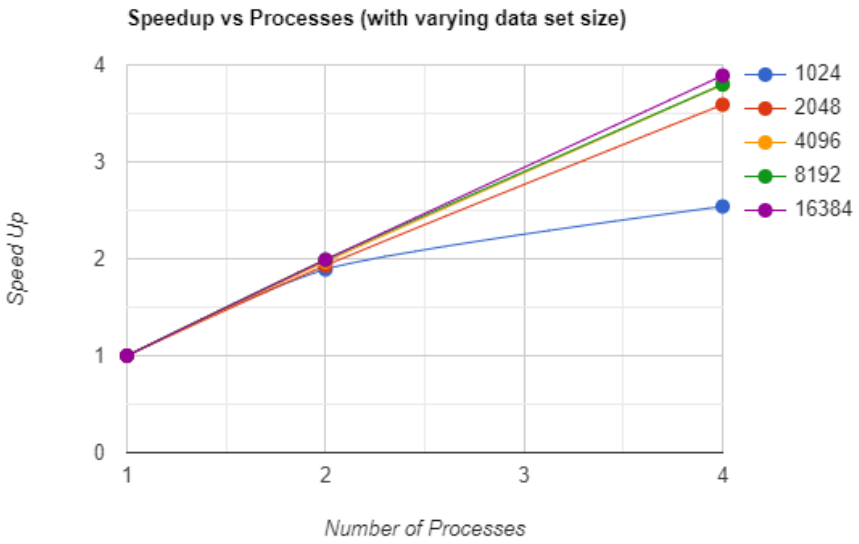
Conclusion

Based on the results, parallelizing the histogram program with 2 processes is immediately beneficial, even when given a small data set size. Even at a data set size of 1024, efficiency of parallelization for 2 processes is already > 0.7 (E = 0.95). For parallelization using the entire 4 core USS system, at 1024, efficiency is only 0.64 and not deemed worthwhile to parallelize with 4 processes. However, at data set sizes of 2048 and above, utilizing the entire 4 core system with 4 processes is worthwhile, with efficiency > 0.7 (E = 0.89 and a speedup of 3.59x at 2048 data set size), gradually increasing as data set sizes does. Overall, this program is weakly scalable as efficiency can be kept stable by increasing data set size as we increase the number of processes.

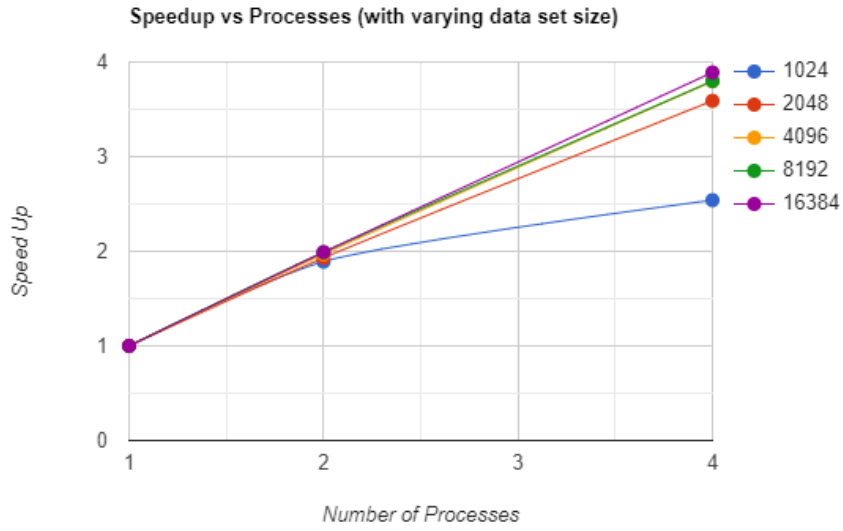
Appendix

Data set size	1 process	2 processes	4 processes
1024	3.65e-05s	1.93e-05s	1.44e-05s
2048	7.18e-05s	3.72e-05s	2.00e-05s
4096	1.42e-04s	7.22e-05s	3.74e-05s
8192	2.82e-04s	1.42e-04s	7.44e-05s
16384	5.63e-04s	2.83e-04s	1.45e-04s

Appendix 1. Raw Run-Time data in seconds



Appendix 2. Graph of Speedup vs Number of Processes (with varying data set size)



Appendix 3. Graph of Efficiency vs Number of Processes (with varying data set size)