COL380 Assignment 1 Report

Contents

1 Description of programs

1.1 Serial program

The serial program is almost the same as the simple version that was given to us.

1.2 Parallel version 1

In this version, we give a local variable representing the partial sum of some subset of iterations to each thread, which is then added to the sum in the end using a critical section, which will be potentially run 8 times.

1.3 Parallel version 2

Suppose T is the number of threads. In this version, we run the algorithm for $1 + \lceil \log_2 T \rceil$ steps. In the first step, which is the computationally most heavy step, we run a parallel for loop which has a local variable for each thread that stores the sum of elements accessed in the iterations assigned to that thread. We use $O(T)$ extra memory to store these results. Then in the next $\lceil \log_2 T \rceil$ iterations, we accumulate the partial sums of chunks of contiguously numbered threads in chunks of powers of two.

1.4 Parallel version 3

This version is quite similar to the previous version, but here we do static assignment of chunks for each thread using thread numbers and distributing the load somewhat equally.

2 Total time taken by programs

2.1 Serial program

N	10^3	10^5	10^7
Time taken (in ms)	0.010	0.875	77.191

2.2 Parallel version 1

Number of threads, N	10^3	10^5	10^7
2	0.101	0.945	74.458
4	0.181	0.897	65.951
8	0.328	0.900	65.885

2.3 Parallel version 2

Number of threads, N	10^3	10^5	10^7
2	0.140	0.787	65.378
4	0.174	0.796	58.019
8	0.324	0.930	56.788

2.4 Parallel version 3

Number of threads, N	10^3	10^5	10^7
2	0.084	0.810	68.155
4	0.170	0.811	59.482
8	0.333	1.141	65.414

3 Time taken by the parallelizable part of the programs

3.1 Serial program

N	10^3	10^5	10^7
Time taken (in ms)	0.004	0.364	26.402

3.2 Parallel version 1

Number of threads, N	10^3	10^5	10^7
2	0.096	0.278	16.601
4	0.176	0.301	10.489
8	0.322	0.408	9.514

3.3 Parallel version 2

Number of threads, N	10^3	10^5	10^7
2	0.134	0.257	14.520
4	0.168	0.271	10.647
8	0.174	0.406	8.590

3.4 Parallel version 3

Number of threads, N	10^3	10^5	10^7
2	0.079	0.278	16.137
4	0.170	0.286	11.957
8	0.333	0.492	14.040

4 Speedup, Efficiency, and Amdahl's law calculations

We compute the fraction f used in the statement of Amdahl's law for each value of N using the serial program.

N	10^3	10^5	10^7
f	0.4	0.416	0.342

Using Amdahl's law, if the maximum possible speedup is s, and the number of processors is p (equal to the number of threads for this assignment, since the number of threads is small and less than the number of processors on my machine), then

$$s = \frac{1}{(1-f) + \frac{f}{p}}$$

The ideal speedups would then be

Number of threads, N	10^3	10^5	10^7
2	1.25	1.26	1.21
4	1.43	1.45	1.34
8	1.54	1.57	1.43

For the real speedups, the tables are as follows:

4.1 Parallel version 1

Number of threads, N	10^3	10^5	10^7
2	0.099	0.926	1.037
4	0.055	0.975	1.170
8	0.030	0.972	1.171

4.2 Parallel version 2

Number of threads, N	10^3	10^5	10^7
2	0.071	1.112	1.181
4	0.057	1.099	1.313
8	0.031	0.941	1.359

4.3 Parallel version 3

Number of threads, N	10^3	10^5	10^7
2	0.119	1.080	1.132
4	0.059	1.079	1.298
8	0.031	0.767	1.180

As far as the efficiency is concerned, it is given by

$$e = \frac{s}{p}$$

So the efficiencies are as tabulated below:

4.4 Parallel version 1

Number of threads, N	10^3	10^5	10^7
2	0.049	0.463	0.518
4	0.014	0.244	0.293
8	0.004	0.122	0.146

4.5 Parallel version 2

Number of threads, N	10^3	10^5	10^7
2	0.035	0.556	0.590
4	0.014	0.275	0.428
8	0.004	0.117	0.170

4.6 Parallel version 3

Number of threads, N	10^3	10^5	10^7
2	0.056	0.540	0.566
4	0.030	0.270	0.325
8	0.004	0.096	0.147

5 Analysis

We can observe the following things:

1. Amdahl's law is never followed accurately if it is interpreted to give an estimate of the real speedup, but if it is interpreted as a theoretical upper-bound, then yes, it is followed. It is still quite imprecise in terms of predicting the correct speedup, but as N increases, we get a better and better approximation (due to overhead being less significant than the actual computation).

2. When the data size is small, the main bottleneck in the program is the overhead associated with threads, including context switches, waiting and so on. For moderate data sizes, this is still a bottleneck, but not so prominent. The usage of threads shines when the data is large (i.e., when $N = 10^7$), giving us good speedups, somewhat close to the ideal speedups as well. This can be further verified from the time taken by the parallelizable part of the programs for each version.

3. The second version is more efficient than the first version. This is because in the first version, the critical section is run by T threads, and hence requires more time (since any code run on it would in a way run sequentially, while in the second version, there is no critical section, but $\lceil \log_2 T \rceil$ iterations where the threads all run in parallel, so it runs faster.

4. The second version and the third version work at similar speeds, except for the cases where $N = 10^3$, and the case where there are 8 threads. In the first case, the effect of dynamic scheduling overhead is clearly visible, and in the second case, it seems like there is some anomaly. Also, by explicitly carrying out a comparison with static and dynamic scheduling provided by OpenMP, there was a similar trend there too.

6 Running the code

The code is in the files `sum.c` (for the serial code), and `sumX.c` where `X` is the version of the parallel code.

To create an executable `e` from file `e.c`, run `make e`.

To run the serial code, run `make 0`.

To run version `X` of the parallel code, run `make X`.

To clean up any executable files in the directory, run `make` or `make clean`.