Homework 1

CS 406

Parallel Programming

Spring 2020-2021

Instructor:

Kamer Kaya

Faculty of Engineering and Natural Sciences

Sabanci University

Efe Şencan

25083

Table of Contents

1) Problem Statement

The goal of this project is to implement parallel Sinkhorn-Knopp scaling algorithm which is used to convert a given matrix to a doubly stochastic form, where every row and column sum is equal to one. We used OpenMp library for the parallel implementation of this problem.

2) Sequential Implementation

When we implemented the algorithm with a sequential manner, in other words, using only a single thread, the obtained results with different compile flags are as follows.

Sequential Sinkhorn-Knopp Algorithm:

Input: A: n x n matrix

Output: rv, cv : row / column scaling arrays (entries of the diagonal scaling matrices)

```
for i ← 1 to n do
        rv[i] ← 1
        cv[i] ← 1
for x ←1 to iterations do
        for i ← 1 to n do
                start ← xadj[i]
                end ← xadj[i + 1]
                for j ← 1 to (end - start) do
                        rsum += cv[adj[start + j]]
                rv[i] ← 1/rsum
        for j ← 1 to n do
                start ← txadj[j]
                end ← txadj[j + 1]
                for i ← 1 to (end - start) do
                        csum += rv[tadj[start + i]]
                cv[j] ← 1/csum
        global_error ←-1
        for i ← 1 to n do
                start ← xadj[i]
                end ← xadj[i + 1]
                for j ← 1 to (end - start) do
                        cur_value += rv[i] * cv[adj[start + j]]
                cur_value = abs(1 – cur_value)
                if cur_value > global_error do
                        global_error = cur_value
```

Compile flag -O3 , Total iterations = 20

Results	Hamrle3	atmosmodl	vas_stokes	cage15	stokes
Time (s)	0.62	0.91	2.90	8.50	29.69

Compile flag -O0 , Total iterations = 20

Results	Hamrle3	atmosmodl	vas_stokes	cage15	stokes
Time (s)	2.08	3.32	9.73	29.62	101.47

3) Parallel Implementation

3.1) Implementation Details

When we take a look at the sequential algorithm, we first observe the for loop that is used to initialize the 'rv' and 'cv' arrays and set each of their value to 1. Since every task in the loop iteration is independent from each other, we can directly parallelize that for loop. After parallelizing that loop, there is an outer loop which runs 'iteration' many times that is specified by the user. However, we cannot parallelize that loop because at each iteration, we update the value of rv and cv and use them for the next iteration which are further used to compute the error value. Therefore, we have a task dependency there. When we observe the first inner loop which is used to update the elements of rv, we can see that at every loop iteration, different index of the array is updated and there is not any task dependency between the iterations. For this reason, we can parallelize that for loop. When we take a look at the loop variables, we have 'rsum', 'start', and 'end'. We preferred to declare these variables inside the inner loop to avoid them being shared among the threads which can cause performance issues in the program. The second inner loop is used to update the value of the cv array. Similar to the first inner loop, we can parallelize that loop, since every iteration is independent from each other, and set the variables 'csum', start, and end thread spesific. Finally, we have the inner loop for computing the error value for that iteration. In order to compute that value, 'rv' and 'cv' arrays should be updated, and we use these arrays in order to compute the 'cur_value'. At each iteration in the last

inner loop, we compute different error values by taking abs(1 – cur_value), and at the end of the loop, we set the global error value as the maximum of these current values. Therefore, the important point in that loop is that, there is a race condition while updating the global error value, since it is declared outside of that last inner loop and it is shared among the threads. To ensure that only a single thread is updating the value of the global error at a certain time, we used '#pragma omp atomic' which prevents that race condition.

Parallel Sinkhorn-Knopp Algorithm:

```
# pragma omp parallel for num_threads(nt)
for i ← 1 to n do
    rv[i] ← 1
    cv[i] ← 1
for x ←1 to iterations do
    # pragma omp parallel for num_threads(nt)
    for i ← 1 to n do
            start ← xadj[i]
            end ← xadj[i + 1]
            for j ← 1 to (end - start) do
                    rsum += cv[adj[start + j]]
            rv[i] ← 1/rsum
    # pragma omp parallel for num_threads(nt)
    for j ← 1 to n do
            start ← txadj[j]
            end ← txadj[j + 1]
            for i ← 1 to (end - start) do
                    csum += rv[tadj[start + i]]
            cv[j] ← 1/csum
    global_error ←-1
    # pragma omp parallel for num_threads(nt)
    for i ← 1 to n do
            start ← xadj[i]
            end ← xadj[i + 1]
            for j ← 1 to (end - start) do
                    cur_value += rv[i] * cv[adj[start + j]]
            cur_value = abs(1 – cur_value)
            if cur_value > global_error do
                    #pragme omp atomic
                    global_error = cur_value
```

Compile flag -O3 , Total iterations = 20, Measured time = second (s)

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3	0.41	0.21	0.12	0.10
atmosmodl	0.53	0.28	0.16	0.11
vas_stokes	1.63	0.88	0.47	0.31
cage15	4.92	2.74	1.44	0.97
stokes	17.42	9.22	4.86	3.29

Compile flag -O0 , Total iterations = 20, Measured time = second (s)

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3	1.10	0.56	0.28	0.15
atmosmodl	1.83	0.87	0.44	0.24
vas_stokes	4.98	2.51	1.30	0.67
cage15	15.45	7.99	4.02	2.04
stokes	51.76	29.87	13.40	6.89

3.1.2) Further Optimizations

We can further improve the performance of the algorithm by re-scheduling the place of the first inner for loop which is used to update the elements of ‘rv’ array. As we can observe from the algorithm, both the first and the last inner loop computes cv[adj[start + j]], and this value is used to compute the ‘rsum’ for the first inner loop while it is being used to compute the error for the last inner loop. Therefore, we can update the ‘rv’ array just inside the last inner loop. Thus, we could get rid of the burden of first inner loop, which

runs ‘n’ (number of rows/columns of the matrix) many times at every iteration. However, we still need to update the ’rv’ array only for the first iteration, because the updated version of rv array is required for the computation of cv array and that array ‘cv’ is also required for the computation of error. For this reason, before starting the iterations, we updated the ‘rv’ array using the same inner for loop. The results of the optimized algorithm is as follows.

Optimized Parallel Sinkhorn-Knopp Algorithm:

```
# pragma omp parallel for num_threads(nt)
for i ← 1 to n do
    rv[i] ← 1
    cv[i] ← 1
# pragma omp parallel for num_threads(nt)
    for i ← 1 to n do
            start ← xadj[i]
            end ← xadj[i + 1]
            for j ← 1 to (end - start) do
                    rsum += cv[adj[start + j]]
            rv[i] ← 1/rsum
for x ←1 to iterations do
    # pragma omp parallel for num_threads(nt)
    for j ← 1 to n do
            start ← txadj[j]
            end ← txadj[j + 1]
            for i ← 1 to (end - start) do
                    csum += rv[tadj[start + i]]
            cv[j] ← 1/csum
    global_error ←-1
    # pragma omp parallel for num_threads(nt)
    for i ← 1 to n do
            start ← xadj[i]
            end ← xadj[i + 1]
            for j ← 1 to (end - start) do
                    cur_value += rv[i] * cv[adj[start + j]]
                    rsum ← cv[adj[start + j]]
            cur_value = abs(1 – cur_value)
            if cur_value > global_error do
                    #pragme omp atomic
                    global_error = cur_value
            rv[i] ← 1/rsum
```

Compile flag -O3 , Total iterations = 20, Measured time = second (s)

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3	0.27	0.16	0.09	0.06
atmosmodl	0.39	0.19	0.10	0.08
vas_stokes	1.42	1.00	0.57	0.35
cage15	3.63	1.98	1.13	0.73
stokes	14.20	9.54	5.85	3.41

Compile flag -O0 , Total iterations = 20, Measured time = second (s)

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3	1.02	0.52	0.26	0.16
atmosmodl	1.43	0.72	0.36	0.19
vas_stokes	5.52	3.93	2.30	1.27
cage15	14.48	7.70	4.07	2.18
stokes	56.36	38.43	20.88	11.84

3.3) Effect of Scheduling Options on Performance

In this section, we compared the effect of different scheduling options on performance of our algorithm.

3.3.1) Static

When we define schedule(static,chunk-size) next to openmp for loop, OpenMP divides the iterations into that defined chunk-size and assigns these chunks to the threads in a circular manner. If chunk-size is not defined, then each thread will approximately have equal

amount of chunk size. I experimented with different chunk-sizes with the optimized algorithm and obtained the best result when chunk size is equal to 16384. Since the execution time of the program could slightly differ within different iterations, I executed the program **30** times and computed the average time. The average results of the program with static scheduling and with chunk size = **16384** with **20** iterations are as follows.

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	0.29	0.15	0.08	0.06
atmosmodl -O3	0.42	0.23	0.12	0.08
vas_stokes -O3	1.13	0.63	0.37	0.25
cage15 -O3	3.52	1.91	1.11	0.59
stokes -O3	12.41	6.36	3.76	2.34
Hamrle3 -O0	0.91	0.46	0.24	0.14
atmosmodl -O0	1.47	0.74	0.38	0.20
vas_stokes -O0	4.38	2.34	1.28	0.76
cage15 -O0	13.13	6.66	3.38	1.78
stokes -O0	44.30	22.78	11.81	6.52

3.3.2) Guided

In guided schedule, each thread executes the assigned chunks and then request another chunk until the chunks are finished. However, the size of the assigned chunks is determined based on the unassigned iterations and total number of threads. Since, guided schedule could not beat the static and dynamic schedule in terms of execution time, I did not take the average of the results. The results of the program with guided schedule and with 20 iterations are as follows.

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	0.29	0.15	0.08	0.06
atmosmodl -O3	0.39	0.22	0.14	0.08
vas_stokes -O3	1.21	0.71	0.44	0.33
cage15 -O3	3.36	1.77	0.97	0.66
stokes -O3	12.59	6.80	3.98	2.63
Hamrle3 -O0	1.03	0.52	0.26	0.13
atmosmodl -O0	1.45	0.73	0.36	0.18
vas_stokes -O0	4.58	2.50	1.44	0.85
cage15 -O0	13.24	6.64	3.34	1.69
stokes -O0	47.13	24.70	13.37	6.95

3.3.3) Dynamic

With dynamic scheduling, OpenMP divides iterations into chunk sizes and every thread is assigned to a particular chunks. When threads are done with their execution, they request another chunk until there are no more chunks left. The default chunk size is 1. When I made experiments with dynamic schedule with chunk size equals to 16384, I obtained the best results. Since my results were close to the static schedule result, I took the average of **30** runs to have a better evaluation of the results. The average results of the dynamic schedule with chunk size = **16384** and with **20** iterations are as follows.

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	0.30	0.15	0.08	0.05
atmosmodl -O3	0.44	0.23	0.12	0.07
vas_stokes -O3	1.15	0.60	0.34	0.21
cage15 -O3	3.85	2.08	1.12	1.09
stokes -O3	12.41	6.37	4.89	3.26
Hamrle3 -O0	0.92	0.47	0.24	0.12
atmosmodl -O0	1.47	0.74	0.39	0.21
vas_stokes -O0	4.37	2.24	1.17	0.63
cage15 -O0	13.33	6.83	3.41	1.75
stokes -O0	44.57	22.46	11.35	5.74

3.3.4) Auto

With auto scheduling, the scheduling type of the program is determined by the compiler or runtime of the system. Since the performance of auto scheduling could not beat the performance of static scheduling in general, I did not take the average of the results. The results of the auto scheduling with 20 iterations are as follows.

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	0.27	0.15	0.11	0.06
atmosmodl -O3	0.39	0.19	0.10	0.07
vas_stokes -O3	1.43	1.00	0.56	0.35
cage15 -O3	3.62	1.99	1.14	0.70
stokes -O3	14.44	9.61	5.63	3.38
Hamrle3 -O0	1.01	0.51	0.26	0.15
atmosmodl -O0	1.43	0.72	0.36	0.19
vas_stokes -O0	5.52	3.94	2.28	1.27
cage15 -O0	14.56	7.70	4.12	2.18
stokes -O0	56.35	38.33	20.76	11.87

3.4) Speed Up

To measure the speedup, we divided the sequential execution time (T1) of the program by the parallel execution time (Tp) with respect to different number of threads, datasets, and compiler flags. Since we obtained the best parallel results using the static scheduling (**3.3.1**) with chunk size 16384 and with the **optimized** parallel algorithm, it's execution time will be used for the comparison with the sequential program (**Section 2**). The results are as follows.

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	2.13	4.13	7.75	10.33
atmosmodl -O3	2.16	3.95	7.58	11.37
vas_stokes -O3	2.56	4.60	7.83	11.60
cage15 -O3	2.41	4.45	7.65	14.40
stokes -O3	2.39	3.99	7.01	11.27
Hamrle3 -O0	2.28	4.52	8.66	14.85
atmosmodl -O0	2.25	4.48	8.73	16.6
vas_stokes -O0	2.22	4.15	7.60	12.80
cage15 -O0	2.25	4.44	8.76	16.64
stokes -O0	2.29	4.45	8.59	15.56

Speed up Table 1

If we would apply the same loop optimization **(3.2)** to the sequential implementation, then the updated sequential algorithm table and speedup table would be as follows:

Updated Results	Hamrle3 -O3	atmosmodl-O3	vas_stokes -O3	cage15 -O3	stokes -O3
Time (s)	0.47	0.76	2.22	6.22	23.46

Updated Results	Hamrle3 -O0	atmosmodl-O0	vas_stokes -O0	cage15 -O0	stokes -O0
Time (s)	1.77	2.86	8.56	26.18	88.13

Matrix	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	1.62	3.13	5.87	7.83
atmosmodl -O3	1.80	3.30	6.33	9.5
vas_stokes -O3	1.96	4.60	6.0	8.88
cage15 -O3	1.76	3.25	5.60	10.54
stokes -O3	1.89	3.68	6.23	10.02
Hamrle3 -O0	1.94	3.84	7.37	12.64
atmosmodl -O0	1.94	3.86	7.52	14.3
vas_stokes -O0	1.95	3.65	6.68	11.26
cage15 -O0	1.99	3.93	7.74	14.70
stokes -O0	1.98	3.86	7.46	13.54

Speed up Table 2

3.5) GFLOP

FLOP is the measure of how many floating-point operations are done per second by the program, and GFLOPs is a billion FLOPs. In order to measure GFLOPs, after running the program, we executed '**perf stat ./scalesk**' command in Gandalf, so that we can observe the total number of instruction counts. By dividing the total instruction counts to the execution time of our program, we can obtain the FLOPs, and finally by dividing FLOPs to 10^9 we can measure the GFLOPs of our program. The **GFLOPs** result of the optimized version of our program with 20 iterations is as follows.

Matrix	1 thread	2 threads	4 threads	8 threads	16 threads
Hamrle3 -O3	0.005	0.008	0.015	0.027	0.036
atmosmodl -O3	0.003	0.005	0.009	0.019	0.027
vas_stokes -O3	0.001	0.002	0.053	0.006	0.007
cage15 -O3	0.0003	0.0007	0.001	0.0025	0.0028
stokes -O3	0.0001	0.002	0.004	0.0048	0.001
Hamrle3 -O0	0.001	0.002	0.005	0.01	0.017
atmosmodl -O0	0.0009	0.001	0.003	0.006	0.12
vas_stokes -O0	0.0002	0.0005	0.001	0.0019	0.003
cage15 -O0	0.00009	0.0001	0.0003	0.0007	0.001
stokes -O0	0.00002	0.0001	0.0002	0.00021	0.0004

4) Results

I implemented the parallelized version of Sinkhorn-Knopp using C++ OpenMp library. I observed that, my implementation is scalable with up to 16 threads. In addition, I optimized the loop structure of the naïve algorithm and further increased the speed of the both sequential and parallel execution of the program. I also tried to get rid of the second inner loop structure and perform everything in the error loop, however this approach did not bring performance gain. I also made experimentations with different scheduling options and observed that static scheduling with chunk size 16384 achieves the best running time performance. Furthermore, I measured the speedup value and the GFLOPs of the program.