Homework 2: OpenMP Programming

Student ID: 2022280387

Name: 陈英发

Date: 2022-09-25

Introduction

This is the report of the Homework 2 in the course *Introduction to Big Data Systems* during the Fall of 2022 at Tsinghua University.

In this coursework, we implement the parallel version of the PageRank algorithm using OpenMP in C++, and analyze the impact of scheduling strategies and hyperthreading on the speed up of the parallelization.

Method

The PageRank algorithm will iterate `MAXITER` number of times or until convergence, each iteration use the result from previous iteration, so it is difficult to parallelize these iterations. Instead, we parallelize the loops that is responsible for computing each node's new score, computing broadcast score and checking for convergence.

Specifically, we parallelize the two for loops inside the outer while loop. The pseudocode is as follows:

```
Repeat until convergence or MAXITER:
    for each node u:                    <--- Parallellize this
        score_new[u] = 0.0
        if u has no outgoing edge:
            broadcastScore += score_old[u]
        for each node v that has an edge pointing to u:
            score_new[u] += score_old[v] / outgoing_size(v)
        score_new[u] = damping * score_new[u] + (1.0 - damping) * equal_prob
    for each node u:                    <--- Parallellize this
        score_new[u] += damping * broadcastScore * equal_prob
        globalDiff += abs(score_new[u] - score_old[u])
    converged = (globalDiff < convergence)
    score_old = score_new
```

Each iteration in the second loop takes constant time, and is not the bottleneck, so we just use the default scheduling method, which is dividing iterations into 4 contiguous chunks to assign to each thread.

However, when computing the new score for a node u (the first for loop), we need to loop over the nodes with an edge pointing towards u, so the workload of iterations are unbalanced when nodes have very different number of edges. So we want to schedule the iterations to threads in a way such that the workload of each threads are as balanced as possible.

Due to the convenience of OpenMP macros, we just need to add two lines:

```
#pragma omp parallel for reduction(+:broadcastScore) schedule(method,
chunk_size)
// First for loop
#pragma omp parallel for reduction(+:globalDiff)
// Second for loop
```

Where `schedule(method, chunk_size)` specifies the scheduling method.

Scheduling Method

We consider three most common scheduling methods provided by OpenMP: static, dynamic and guided.

Let n be the number of iterations, and t be the number of threads.

Static: This is the default in OpenMP, iterations are divided into chunks and fed to thread in a round-robin manner. However, this simple assignment might still result in unbalanced workload. `chunk_size` defaults to n/t.

Dynamic: Chunks are dynamically assigned to threads after they finish processing their current chunk. While this method balanced workload, the scheduling logic results in significant overhead.

Guided: Similar to dynamic, but the chunk size of assignment to a certain thread is equal to r/t where r is the number of remaining iterations. The `chunk_size` argument specifies the minimum size of a chunk. This method can reduce the overhead of dynamic scheduling, but if the workload is concentrated in some contiguous iterations, it might still result in unbalanced workload.

Experiments

The implementation is tested on the server of the course, the environment is:

- OS: Ubuntu 20.04
- CPU: 4 cores (8 threads) of Intel Xeon 8176

We test the algorithm on the `com-orkut_117m.graph` graphs, which has 117,185,083 edges, and 3,072,441 nodes.

Each time result is the average of 3 runs.

Results

Different Scheduling Strategies

The time of sequential implementation on `com-orkut_117m.graph` is 12.831064 seconds.

Table 2: Performance of different scheduling methods on `com-orkut_117m.graph` with varying `chunk_size` argument using 4 threads.

Chunk size	Static	Dynamic	Guided
1	3.541293	5.685010	3.398209
16	3.421360	3.461200	3.274787
128	3.339067	3.345304	3.227207
1024	3.285014	3.288189	3.283748
10000	3.335913	3.255207	3.290994
n/t	3.748535	3.722884	3.729827

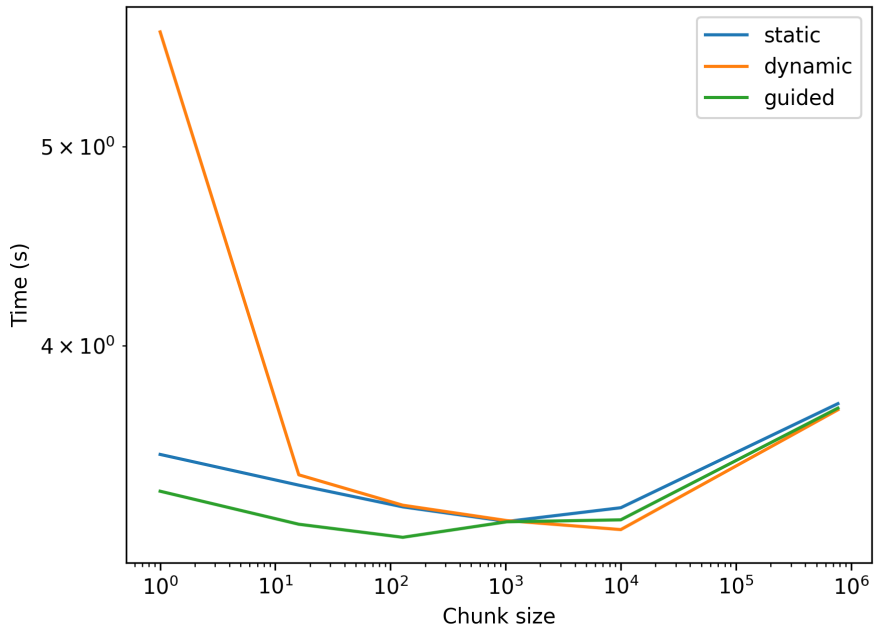

Figure 1: Plot of different scheduling methods on `com-orkut_117m.graph` with varying `chunk_size` argument using 4 threads.

> Note that for `com-orkut_117m.graph`, $n = 3072441$, so $n/t = 768110.25$, and to avoid small remainder iteration count in the end, we round upwards to 768111.

From the plot, we see that the chunk size is best set within the range $[10^2, 10^4]$, We also notice that the time of each scheduling method is almost the same when the chunk size is set to 1024, so we set that for the rest of the experiments.

Analysis

We observe that the time taken of all scheduling strategies is a U-shaped curve with respect to chunk size, which means that the time will increase if the chunk size is too small or too big.

For static scheduling, when the chunk size is too small, space locality is not well preserved, which may result in more cache misses, and thus more time taken. Additionally, in our experiments because we represent the graph using two arrays, where all edges of the same node are contiguous, we want to have such contiguous chunk assigned to the same thread. On the other hand, when the chunk size is too big, a thread is assigned very large contiguous iterations, which may result in unbalanced workload. This is due to the fact that vertices that are close to each other in the graph often have ID values that are close to each other, and we iterate vertices in ID order, so dense regions of the graph are likely to be assigned to the same thread.

For dynamic scheduling, assigning iterations to threads dynamically results in significant overhead, and the number of assignments is inversely proportional to chunk size, so the overhead is larger when chunk size is small. But, as we increase the chunk size, it falls back to static scheduling, and therefore has the same problem of unbalanced workload due regarding dense regions.

For guided scheduling, the number of assignments is reduced because of the dynamic chunk size, but towards the end, the chunk size can get too small, resulting in a smaller, but noticeable overhead. As the (minimum) chunk size increases, the overhead decreases, but the time taken increases due falling back to static scheduling.

Different Number of Threads

Table 2: Performance of different scheduling methods on `com-orkut_117m.graph` with varying number of threads using `chunk_size` of 1024.

t (# threads)	Time	Speed up
1	12.831064	
2	6.471210	1.98
3	4.361351	2.94
4	3.255207	3.94
8	2.828296	4.54
16	2.885661	4.45
32	2.852819	4.50

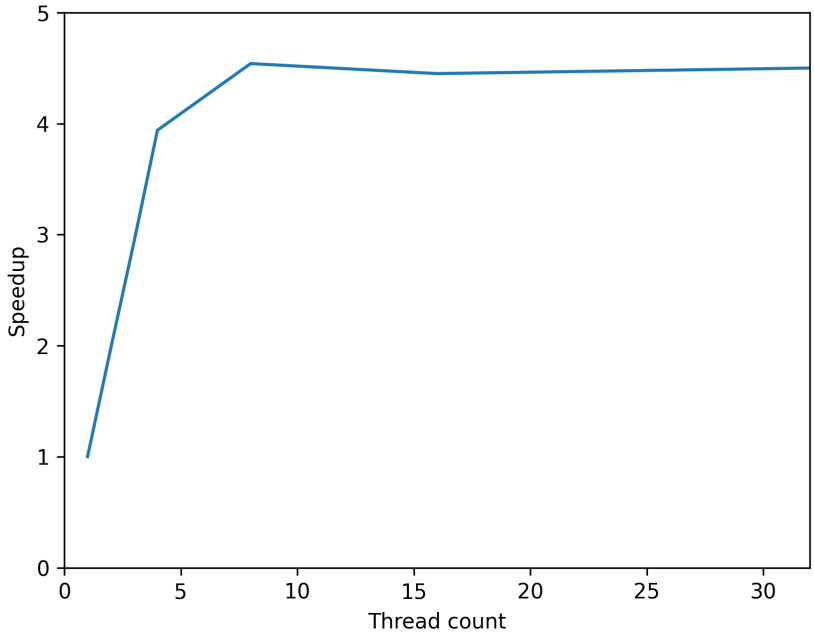

Figure 2: Relationship of speedup to thread count.

Speed Up

From Table 2 and Figure 2, we see that when the $t \leq 4$, the speed up is roughly the same as t, but always a bit lower, this is due to the sequential part that is not parallelized (which also includes the overhead introduced by the logic to perform the parallelization). The **Amdahl's law** states that the theoretical speedup S of the task is

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

where p is the proportion of the execution time that is being improved, s is the speedup of this the improvement. In parallelization, theoretically, we have $s = t$. And if we define $q = 1 - p$ as the proportion of execution time that is not being improved (the sequential part).

$$S = \frac{1}{q + \frac{1-q}{t}} = \frac{t}{tq + 1 - q} = \frac{t}{q(t-1) + 1} \Rightarrow q = \frac{t - S}{S(t-1)}$$

From experiments, we have an actual speedup of approximately $S \approx 3.94$ when $t = 4$, so

$$q = \frac{0.06}{3.4 \times 3} \approx 0.0051$$

So the sequential part only makes up 0.51% of the execution time.

Hyperthreading

The experiments are run on a CPU of 4 cores, but they support hyperthreading, so each core can execute 2 threads, resulting in a total number of 8 threads. But the hyperthreads are not as capable as ordinary threads. Therefore, in Table 2, we see that the time further decreases when we set the number of threads to 8, but the speed up is not as great as when the thread count is smaller (i.e. it's much lower than 8). In fact, using the deduction from previous section, we find out that the sequential part makes up $q = 8/(4.5 * 7) \approx 25.40\%$ of the execution time.

We also see that when the number of threads exceeds 8, which is the greatest number of hyperthreads, there is no additional speed up.

Conclusion

In this coursework, we implemented a parallelized the PageRank algorithm, and tested it to on a graph with 117M edges. The result shows a speedup that is quite close to the optimal speedup when the number of threads is less than or equal to 4. Because of hyperthreading, we see small additional speedup by increasing the number of threads to 8, but any more threads do not induce additional improvements.