Homework2 : Report

Kai Zhang 2023316026
Tsinghua University
zhang-k23@mails.tsinghua.edu.cn

Oct 2nd 2023

1 Implementation

In this section, we will briefly introduce the background of PageRank algorithm and then introduce our parallelization strategy and its code implementation.

1.1 PageRank

The PageRank algorithm was proposed as an algorithm to calculate the importance of Internet pages. PageRank is a function defined on the set of web pages, it gives a positive real number for each web page, indicating the importance of the web page, the whole constitutes a vector, the higher the PageRank value, the more important the web page, the ranking in the Internet search may be ranked first.

Assuming that the Internet is a directed graph, a random walk model is defined based on it, that is, a first-order Markov chain, which represents the process of random browsing of web pages on the Internet by web browsers. Suppose that the viewer jumps to the next page with equal probability according to the hyperlink connected out in each page, and continues to carry out such a random jump on the Internet, this process forms a first-order Markov chain. PageRank represents the smooth distribution of this Markov chain. The PageRank value of each page is the stationary probability.

Now we explain the PageRank algorithm by Markov chain transition matrix and stationary probability solution.

Taking the directed graph of Fig. 1 as an example, we assume that the PageRank value (the probability of accessing each node) of each point at some point in the state is vector $\mathbf{R}$, and the state transition matrix is M. Therefore, we can see from the figure and get the state transition matrix.

$$M = \begin{bmatrix} 0 & \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix} \tag{1}$$

The probability distribution of time t accessing each node at some time is represented by $\mathbf{R_t}$, then $t+1$ can be represented as $\mathbf{R_{t+1}}$,which satisfies the following properties:

$$\mathbf{R_{t+1}} = M\mathbf{R_t} \tag{2}$$

$$\lim_{t\to\infty} M^t \mathbf{R_0} = \mathbf{R} \tag{3}$$

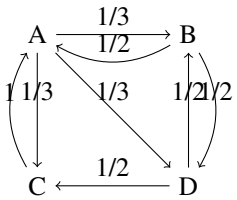

Figure 1: Example Directed Graph

$$M\mathbf{R} = \mathbf{R} \tag{4}$$

Where $\mathbf{R}$ is a stationary probability distribution.

$$R = \begin{bmatrix} PR(v_1) \\ PR(v_2) \\ \vdots \\ PR(v_n) \end{bmatrix} \tag{5}$$

$$PR(v_i) \geq 0, \quad i = 1, 2, \cdots, n \tag{6}$$

$$\sum_{i=1}^{n} PR(v_i) = 1 \tag{7}$$

Therefore, we can get the solution of the PR value as follows, where $M(v_i)$ is the set of nodes pointing to v_i, and $L(v_j)$ is the number of out directed edges of v_j. We can solve iteratively through this equation, which is the basic principle of the PageRank algorithm.

$$PR(v_i) = \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}, \quad i = 1, 2, \cdots, n \tag{8}$$

However, in practice, we need to introduce the damping coefficient to construct a model that guarantees the existence of a stationary probability, and we also need to consider the node without any out edge, that is, $L(v_j)$ is 0. Therefore, the final iteration formula is as follows:

$$PR(v_i) = d\left(\sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)} + \frac{broadcastScore}{n}\right) + \frac{1-d}{n}, \quad i = 1, 2, \cdots, n \tag{9}$$

Where d is the damping coefficient, $0 \leq d \leq 1$. And *broadcastScore* is the PR value sum of all nodes without any out edge, this means that they will link to other pages with a uniformly random probability.

1.2 Parallelization

From the above simple theoretical analysis, we know that the PageRank algorithm process is actually very simple, which is an iterative process of matrix operation. Since C++ does not have vector variable types and operation operations like Pytorch or Matlab, we have a lot of space for parallel optimization by using OpenMP.

The key code of PageRank algorithm in serial version is shown in Fig. 2, and it can be seen that there are many for loops that can be optimized in parallel.

After many attempts, we optimized it in the following code section.

1.2.1 Point 1: OpenMP optimizes the largest for loops

The largest for loop is the main iteration process of PageRank algorithm, which is actually the part of matrix multiplication. We added OpenMP optimization in the outer layer. Considering that broadcastScore is shared by all parallel threads and we hope to parallelize summation, **reduction** operation was adopted. As shown in Fig. 3.

1.2.2 Point 2: Access memory optimization

In the largest for loop, we can see that the **score_new[i]** variable is used many times, so to reduce the overhead of accessing memory, we add a local variable instead of using array index operation, which is also shown in Fig. 3.

1.2.3 Point 3: OpenMP optimizes smaller for loops

When calculating the iteration error in the last step, it is essentially a vector operation, which is implemented here through a for loop. We use OpenMP to optimize this function again, as shown in Fig. 4

```cpp
for (int i = 0; i < numNodes; ++i) {
    solution[i] = equal_prob;
}
while (!converged && iter < MAXITER) {
    iter++;
    broadcastScore = 0.0;
    globalDiff = 0.0;
    for (int i = 0; i < numNodes; ++i) {
        score_new[i] = 0.0;

        if (outgoing_size(g, i) == 0) {
            broadcastScore += score_old[i];
        }
        const Vertex* in_begin = incoming_begin(g, i);
        const Vertex* in_end = incoming_end(g, i);
        for (const Vertex* v = in_begin; v < in_end; ++v) {
            score_new[i] += score_old[*v] / outgoing_size(g, *v);
        }
        score_new[i] = damping * score_new[i] + (1.0 - damping) * equal_prob;
    }
    for (int i = 0; i < numNodes; ++i) {
        score_new[i] += damping * broadcastScore * equal_prob;
        globalDiff += std::abs(score_new[i] - score_old[i]);
    }
    converged = (globalDiff < convergence);
    std::swap(score_new, score_old);
}
```

Figure 2: Serial Version Key Code

```cpp
// point 1 : OMP
#pragma omp parallel for reduction(+ : broadcastScore)
for (int i = 0; i < numNodes; ++i)
{
    score_new[i] = 0.0;
    double local_score_new = 0.0; // point 2 : local parameter for access memory optimization
    double old_i = score_old[i];
    if (outgoing_size(g, i) == 0)
        broadcastScore += old_i;
    const Vertex *in_begin = incoming_begin(g, i);
    const Vertex *in_end = incoming_end(g, i);
    for (const Vertex *v = in_begin; v < in_end; ++v)
    {
        local_score_new += score_old[*v] / outgoing_size(g, *v);
    }
    local_score_new = damping * local_score_new + (1.0 - damping) * equal_prob;
    score_new[i] = local_score_new;
}
```

Figure 3: Parallel Version Key Code:Point 1-2

```cpp
// point 4 : Computational optimization
double damping_broadcastScore = damping * broadcastScore * equal_prob;
// point 3 : OMP
#pragma omp parallel for reduction(+ : globalDiff)
for (int i = 0; i < numNodes; ++i)
{
    score_new[i] += damping_broadcastScore;
    globalDiff += std::abs(score_new[i] - score_old[i]);
}
converged = (globalDiff < convergence);
std::swap(score_new, score_old);
```

Figure 4: Parallel Version Key Code:Point 3-4

1.2.4 Point 4: Optimization of the computational process

At the same time, we noticed that in the for loop, the **broadcastScore** corresponding iteration item would be calculated once each time, so we calculated the corresponding multiplication result directly outside the loop to reduce the performance loss caused by calculation operations in each loop, which was also shown in Fig. 4.

In our tests, the above optimization is basically the best OpenMP optimization strategy without considering more advanced optimization methods such as SSE.

2 Time Result

We tested time result of 1, 2, 3, and 4 threads on different graphs. **Each data point is the average of 10 runs of the program.** The test results on the **com-orkut_117m.graph** are shown in Table 1 and Fig 5. It can be seen that the running time of serial programs is basically unchanged, while the running time of parallel programs decreases significantly with the increase in the number of threads.

Table 1: Serial vs Parallel Program Times

Thread Num	Serial Program Time (s)	Parallel Program Time (s)	Speed up
1	5.591212	5.557057	1.0062
2	5.674214	3.864368	1.4683
3	5.568761	2.866733	1.9411
4	5.635632	2.292696	2.4580

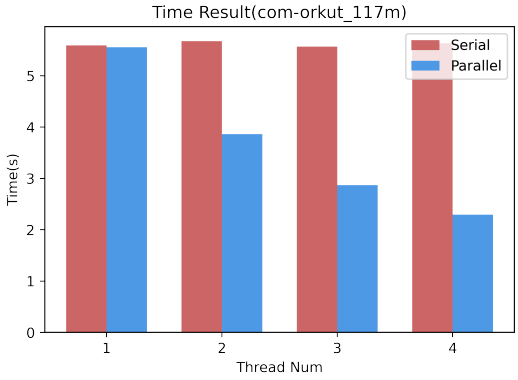

Figure 5: Time Result

As you can see, even in the case of single threads, our program performance has been optimized relative to serial programs, mainly due to our memory access optimization and computational optimization within the for loop.

The results of testing the other three graph data are shown in Fig. 6. We can see that as the amount of data decreases, the advantage of parallel becomes smaller and smaller, especially on the last smallest data, the program overhead of parallel exceeds that of serial, which should be due to the large number of thread scheduling operations.

3 Strategy comparison

During the programming process we tried different OpenMP strategies and found some problems.

1. If OpenMP optimization is added to the for loop inside the two-fold for loop, performance will drop sharply. The code is shown in Fig. 8 and the results are shown in Fig. 9.

As we can see, the efficiency of parallel programs is greatly reduced after adding the parallelism of inner loops, because nested parallelism leads to complex thread scheduling and additional overhead, even for single threads.

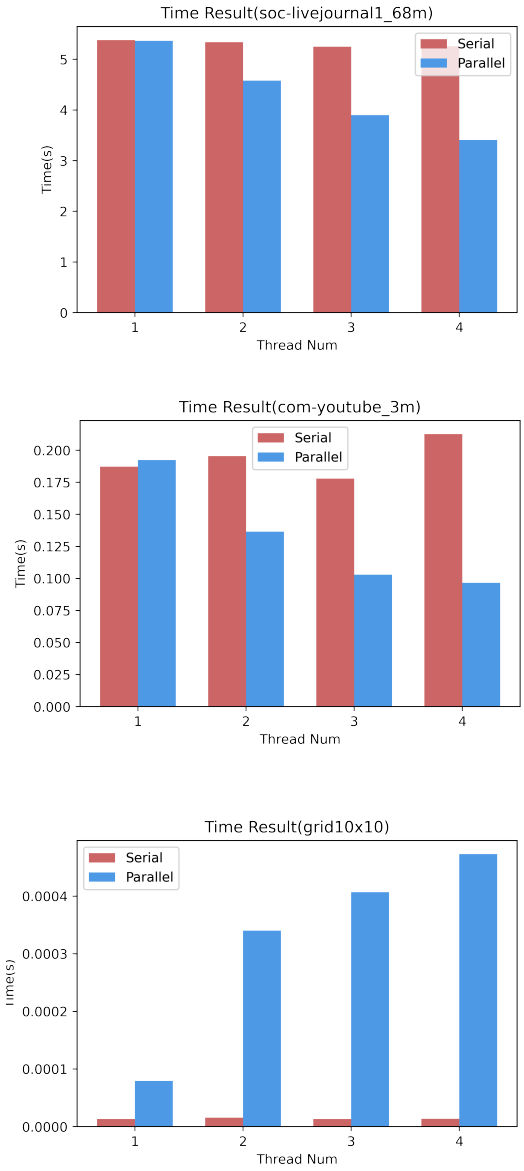

Figure 6: Time Result of other graphs

2. Array initialization for OpenMP optimization will also cause performance degradation, which should be mainly due to the cost of multithreading scheduling exceeds the acceleration brought by multithreading. The code is shown in Fig. 9 and the results are shown in Fig. 10 or Table 2.

We can see that after OpenMP optimization for array initialization, the acceleration results are not obvious, and the results will fluctuate. In many cases, the performance is not as good as serial initialization. Therefore, we did not adopt this strategy in the end, believing that parallel array initialization would introduce more thread scheduling overhead than its speedup.

3. Delete the corresponding reduction operation, as shown in Fig. 11, in an attempt to directly parallel the results. We found that the program did not run correctly, so this strategy is incorrect.

4. For load balancing strategies, we tried dynamic load balancing strategies.

For dynamic load balance, the code is shown in Fig. 12 and the results are shown in Table 3.

It can be seen that after the use of dynamic scheduling, the performance is not improved, but decreased, indicating that under the current data and hardware conditions, 4-thread parallel is not suitable for dynamic scheduling.

Since OpenMP defaults to static load balancing, we don't test static scheduling again.

```cpp
// point 1 : OMP
#pragma omp parallel for reduction(+ : broadcastScore)
for (int i = 0; i < numNodes; ++i)
{
    score_new[i] = 0.0;
    double local_score_new = 0.0; // point 2 : local parameter for access memory optimization
    double old_i = score_old[i];
    if (outgoing_size(g, i) == 0)
        broadcastScore += old_i;
    const Vertex *in_begin = incoming_begin(g, i);
    const Vertex *in_end = incoming_end(g, i);
    // add OMP to inner for loop
    #pragma omp parallel for reduction(+ : local_score_new)
    for (const Vertex *v = in_begin; v < in_end; ++v){
        local_score_new += score_old[*v] / outgoing_size(g, *v);
    }
    local_score_new = damping * local_score_new + (1.0 - damping) * equal_prob;
    score_new[i] = local_score_new;
}
```

Figure 7: Code of adding OpenMP to inner for loop

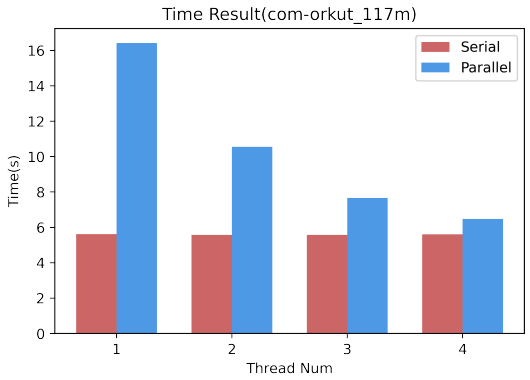

Figure 8: Time Result of adding OpenMP to inner for loop

```cpp
#pragma omp parallel for
for (int i = 0; i < numNodes; ++i)
{
    solution[i] = equal_prob;
}
```

Figure 9: Code of OpenMP for initialization

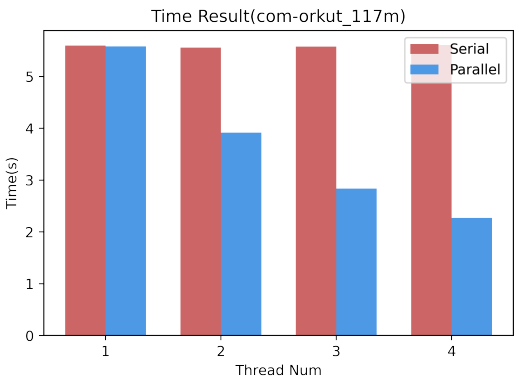

Figure 10: Time Result of OpenMP for initialization

```cpp
// point 1 : OMP
// #pragma omp parallel for reduction(+ : broadcastScore)
#pragma omp parallel for
for (int i = 0; i < numNodes; ++i)
{
    score_new[i] = 0.0;
    double local_score_new = 0.0; // point 2 : local parameter for access memory optimization
    double old_i = score_old[i];
    if (outgoing_size(g, i) == 0)
        broadcastScore += old_i;
    const Vertex *in_begin = incoming_begin(g, i);
    const Vertex *in_end = incoming_end(g, i);
    for (const Vertex *v = in_begin; v < in_end; ++v){
        local_score_new += score_old[*v] / outgoing_size(g, *v);
    }
    local_score_new = damping * local_score_new + (1.0 - damping) * equal_prob;
    score_new[i] = local_score_new;
}
```

Figure 11: Code of not using OpenMP reduction

```cpp
// point 1 : OMP
#pragma omp parallel for schedule(dynamic) reduction(+ : broadcastScore)
for (int i = 0; i < numNodes; ++i)
{
    score_new[i] = 0.0;
    double local_score_new = 0.0; // point 2 : local parameter for access memory optimization
    double old_i = score_old[i];
    if (outgoing_size(g, i) == 0)
        broadcastScore += old_i;
    const Vertex *in_begin = incoming_begin(g, i);
    const Vertex *in_end = incoming_end(g, i);
    for (const Vertex *v = in_begin; v < in_end; ++v){
        local_score_new += score_old[*v] / outgoing_size(g, *v);
    }
    local_score_new = damping * local_score_new + (1.0 - damping) * equal_prob;
    score_new[i] = local_score_new;
}
```

Figure 12: Code of using dynamic scheduling

Table 2: Time Result of OpenMP for initialization

Thread Num	Serial Program Time (s)	Parallel Program Time (s)	Speed up
1	5.597	5.582	1.0027
2	5.556	3.914	1.4195
3	5.578	2.835	1.9675
4	5.608	2.268	2.4726

Table 3: Time Result of using dynamic scheduling

Thread Num	Serial Program Time (s)	Parallel Program Time (s)	Speed up
1	5.596	7.154	0.7822
2	5.598	4.691	1.1932
3	5.636	3.769	1.4953
4	5.616	3.281	1.7119

4 Analysis of Speedup

In our code implementation, the maximum acceleration that can be achieved by 4 threads is about 2.5x, and the reasons why it is lower than 4x are as follows:

1. **The program is not completely parallel.** There is a certain serial part, so even theoretically it is not possible to achieve 4x acceleration.

2. **Multithreaded parallelism itself introduces additional thread scheduling overhead.** As we can see from our unsuccessful attempt at array initialization parallelism. There are many reasons for this: (1) Scheduling algorithms and mechanisms need to be introduced to allocate different threads, which introduces additional time overhead. (2) When multi-threading accesses memory, it may lead to cache failure, thereby reducing cache efficiency, so it will bring a lot of performance loss.

3. **The work distribution among threads may not be balanced.** As some nodes may have more incoming edges. I tried using dynamic scheduling in the OpenMP parallel loop, but it didn't work. Therefore, appropriate scheduling patterns may be required depending on the hardware and data situation to achieve the best load balancing.

5 Bonus

I tested the experimental results of 4,8,16,32,64 threads, as shown in Table 4.

Table 4: Time Result of using more threads

Thread Num	Serial Program Time (s)	Parallel Program Time (s)	Speed up
4	6.067	2.413	2.5140
8	6.637	1.318	5.0359
16	5.936	1.156	5.1332
32	6.046	1.134	5.3314
64	5.724	1.134	5.0454

The results of the experiment were quite surprising. Even though we only have 4 physical CPU cores, we do get more than 4x parallel acceleration when we take on more threads. This shows that hyperthreading technology is different from common multithreading technology. The reasons for the acceleration are as follows:

1. Hyperthreading is a hardware technology that allows for true multithreaded parallelism, unlike common multithreaded software technology. Therefore, hyperthreading technology can bring parallel acceleration.

2. Under our implementation, for large data sets, the benefit of increasing threads within a certain range is obviously greater than the overhead of thread scheduling. Therefore, 4 threads is far from enough, as the number of threads increases, the effect is better, but when the number of threads exceeds the total number of super-threads 32, there is basically no more acceleration.