Assignment 1

Sahand Sabour - 山姆 - 2022380024

Introduction

In this assignment, we were provided with an implementation of the PageRank algorithm in C++. Accordingly, we were tasked to use the OpenMP library to parallelize this algorithm on multiple CPU cores (i.e., Multi-core Parallelization).

PageRank

The PageRank algorithm, famously created and used by the Google search engine for searching and finding relevant search results, was created for measuring the importance of the web pages in a database given the search query. In this algorithm, every web page is considered a node in a large graph. Accordingly, its importance is counted by the number and the quality of links connected to the node.

OpenMP

Open specification for Multi-Processing (OpenMP) is an API for enabling multi-threaded programs that run on shared memories. This API enables us to efficiently parallelize our code by adding a few lines of code and ensuring no data races.

Implementation

The initial version of the PageRank algorithm assigned values to each node. However, it later changed to assigning each node with equal probabilities, which is implemented via the following for loop:

```
// First optimization
#pragma omp parallel for schedule(static)
for (int i = 0; i < numNodes; ++i)
{
  solution[i] = equal_prob;
}
```

By adding a scheduling clause, we can manage how loops for these assignments are mapped onto different threads, which could have minor improvements in the runtime. Statis scheduling was chosen as the number of nodes is known, and the same value is set to each node (i.e., the similar computational complexity for each iteration of the loop). Accordingly, two other loops in the implementation need to be parallelized. Like the above enhancement, the inner loop has constant computational time; thus, we leverage static scheduling. In addition, since the globalDiff variable is used to store the sum of values during iterations that are calculated on different threads, we use reduction for the addition operand with the starting value of 0 (i.e., reduction(+:)) to share this variable in the enclosing parallel region.

```
// Second optimization
#pragma omp parallel for reduction(+: broadcastScore) schedule(guided)
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;
}

// Third optimization
#pragma omp parallel for reduction(+: globalDiff) schedule(static)
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);
}
```

Similarly, the outer loop requires the reduction operation to calculate the broadcastScore as the sum of old scores. Additionally, the computational complexity for this operation is not the same for each iteration as the complexity of calculating the scores of outgoing edges depends on the number of these edges, which could defer for different nodes.

Hence, we leverage guided scheduling that initially sets a fixed-size chunk to each thread for processing and then dynamically assigns additional chunks to a thread based on its performance (i.e., whether the thread finishes processing the initially assigned chunks before other threads).

Results

The optimized PageRank algorithm was leveraged to process the com-orkut_117m graph.

Schedule Strategy

To investigate the time efficiency of different OpenMP schedule strategies, the guided scheduling for the outer loop was replaced with static and dynamic scheduling with varying chunk sizes. The obtained results are provided in the table below (time in seconds). All values are reported as the average runtime for 5 consecutive runs on a 4 threads. The best result is highlighted in bold.

Chunk Size	Static	Dynamic	Guided
1	3.723383	6.433059	3.257328
8	3.475185	5.670677	3.275761
64	3.362125	3.324762	3.282937
128	3.329298	3.275814	3.250586
512	3.359853	3.269315	**3.244027**
1024	3.309765	3.256879	3.253118

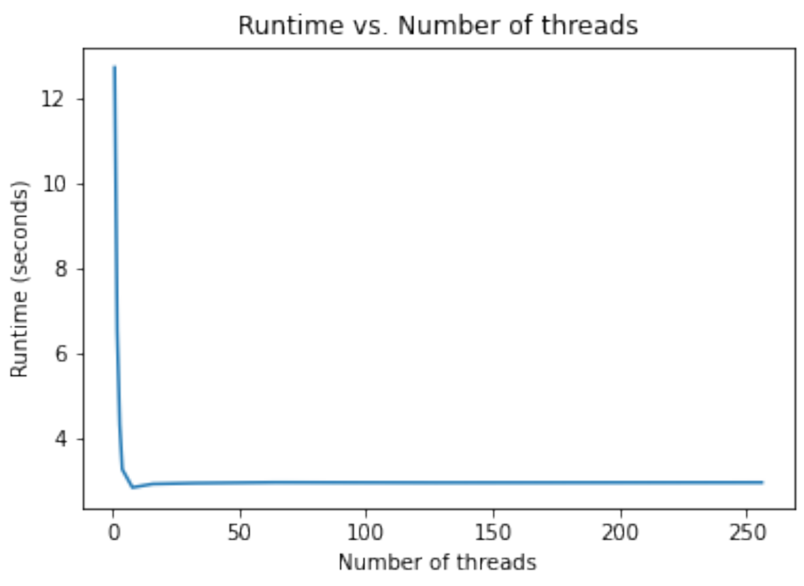
Thread Number

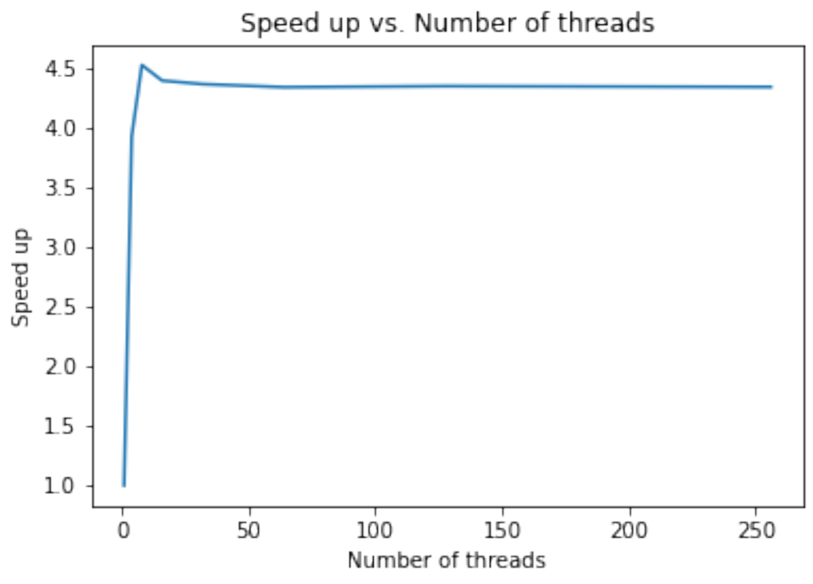
For parallelization, based on the above results, we implemented guided scheduling with a chunk size of 512. The runtime results for the different number of threads are reported in the table below.

Number of Threads	Runtime (seconds)
1	12.744576
2	6.524578
3	4.302668
4	3.244027
8	2.814931

Discussion

According to the obtained results, the speedup using 4 threads was approximately 3.93 (12.744576 / 3.244027). Thus, the speedup is not linearly proportional (i.e., they do not change at the same rate). We believe this is because by adding parallelization to the sequential implementation, we are adding overhead that is used for distributing processes between different threads, which has its own computational complexity.
Regarding hyperthreading, we are provided with a server with 4-core CPU that can run 2 threads on each core, which enables us to parallelize the program on 8 threads. The obtained results demonstrated that increasing the number of threads to 8 could speed up the runtime. However, the proportion of this speedup compared to 4 threads (approximately 1.15) is not as significant as the speedup achieved by increasing the number of threads from 2 to 4 (approximately 2.01), even though the rate of increase in the number of threads is the same (=2). Amdahl's law highlights this finding as it states that the speedup of a program is limited by the proportion of it that can be parallelized.





To investigate this further, we ran the experiment on an increasing number of threads and observed the corresponding speedup. The obtained results are illustrated in the above figures. As shown, there is an upper limit for the amount of speedup that we can achieve by increasing the number of threads. That is, after a certain point, increasing the number of threads increases the computational cost of the overhead to a value that significatly decreases the speedup.

Conclusions

In this assignment, we were tasked to parallelize the PageRank algorithm on multiple CPU cores using the OpenMP API. We implemented three minor improvements to the sequential implementation, which caused a speed up of approximately 4 times on 4 threads using guided scheduling with chunk size 512. Accordingly, we analyzed and discussed the obtained results.