Student Name: Karahan Sarıtaş, Elif Bayraktar
Student ID: 2018400174, 2018205141

Parallel Processing (CMPE 478)

Fall 2022 Homework #1

PageRank Calculation

We assume page A has pages $T_1 ... T_n$ which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also $C(A)$ is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$PR(A) = (1-d) + d * (\frac{PR(T1)}{C(T1)} + ... + \frac{PR(Tn)}{C(Tn)}) \tag{1}$$

Note that, when normalized by dividing $(1-d)$ by the total number of pages (say N) and again taking the initial distribution $(r^{(0)})$ as $1/N$ for each page, the PageRanks form a probability distribution over web pages, so that the sum of all web pages' PageRanks will be one.

The nodes of the graph are domain names, and two nodes, X and Y, are connected by a directed edge X $\rightarrow$ Y if there exists a document (URL) under the domain X that hyperlinks to a document (URL) under the domain Y.

It is also stated on Erdős WebGraph project that "The download contains one zipped file with the following format: two 26 character long string identifiers are written in a line separated with TAB character. Every line defines a directed edge of our webgraph. The identifiers correspond to nodes, first one describes the tail, second one does the head. For example, a row of the form "01324moja6i5ghdbhfe94iou9e 5ltb8q97ou2ui154lc4ohc9pqt" means that a URL in a domain, with ID "01324moja6i5ghdbhfe94iou9e" links to a URL in the domain of ID "5ltb8q97ou2ui154lc4ohc9pqt". If you want to know the actual domain names behind these ID's, you can translate them by using our Domain dictionary."

Implementation

First let's start with the contents of the matrix P. Let $C(A)$ denote the number of links going out of page A:

$$P_{i,j} = \begin{cases} \frac{1}{C(j)}, & \text{if there is a link from page } j \text{ to } i \\ 0, & \text{if no link from page } j \text{ to } i \end{cases}$$

What we have to do is to repeat the following operation until the stopping condition holds for an arbitrary value ϵ:

$$r^{(t+1)} = \alpha \cdot (P * r^{(t)}) + (1-\alpha) \cdot c$$

$$\text{Stopping condition: } \|r^{(t+1)} - r^{(t)}\|_1 \leq \epsilon$$

Within the code snippet provided below, we simply repeat the page rank calculation until the stopping condition holds. Since we want to calculate a column vector $r^{(t+1)}$ consisting of N many independent elements, this calculation can be parallelized easily. Furthermore, we can also parallelize the calculation of each $(P \cdot r^{(t)})_i$ easily. Examine the matrix multiplication shown below:

$$\begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} \\ P_{1,0} & P_{1,1} & P_{1,2} \\ P_{2,0} & P_{2,1} & P_{2,2} \end{bmatrix} \cdot \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

We know that $x_0 = P_{0,0} \cdot r_0 + P_{0,1} \cdot r_1 + P_{0,2} \cdot r_2$. So why not multiply each pairs in parallel and then sum them up atomically? Here you can inspect the corresponding code snippet:

```cpp
vector<double> calc()
{
    nIterations = 0;
    int N = row_begin.size() - 1;
    vector<double> r1(N, 1.0);
    vector<double> r2(N, 1.0);

    while (true)
    {
        double cSum = 0;
        #pragma omp parallel for default(shared) schedule(runtime) reduction(+: cSum)
        for (int i = 0; i < N; i++)
        {
            double sum = 0;
            #pragma omp parallel for default(shared) schedule(runtime) reduction(+: sum)
            for(int j = row_begin[i]; j < row_begin[i + 1]; j++)
            {
                int from = col_indices[j];
                sum += r1[from]*values[j];
            }
            r2[i] = alpha * sum  + (1 - alpha);
            cSum += abs(r1[i] - r2[i]);
        }
        nIterations++;
        if (cSum <= epsilon) break;
        r1 = r2;
    }

    return r1;
}
```

`from` represents a single page that has a link to A. `values[j]` represents the value of $\frac{1}{C_j}$ for the j^{th} page. We sum the results within an atomic construct to avoid race conditions. Also notice that we don't attempt to parallelize simple operations like `r1 = r2` as the parallelization doesn't increase the performance at this point.

Technical Details

We tested our program on three different machines with the following technical properties:
Computer #1:

- OS: Microsoft Windows 10
- CPU: Intel Core i5 8265U @ 1.60GHz 1800MHz 4 cores 8 threads
- Memory: 8GB
- C++ compiler: MinGW-w64 g++

Computer #2:

- OS: macOS Monterey
- CPU: Apple M1 Pro
- Memory: 32GB
- C++ compiler: clang++

Tables and Discussion

Computer #1 results (plugged-in):

Test No.	Scheduling Method	Chunk Size	No. of Iterations	Timings in seconds for each number of threads							
				1	2	3	4	5	6	7	8
0	omp_sched_static	20000	14	485	266	218	195	190	172	165	160
1	omp_sched_dynamic	20000	14	469	269	242	191	178	169	162	158
2	omp_sched_guided	20000	14	486	273	220	190	179	169	163	157

Computer #2 results:

Test No.	Scheduling Method	Chunk Size	No. of Iterations	Timings in seconds for each number of threads							
				1	2	3	4	5	6	7	8
0	omp_sched_static	20000	14	4	2	1	1	1	1	1	0
1	omp_sched_dynamic	20000	14	4	2	2	1	1	1	1	1
2	omp_sched_guided	20000	14	4	2	1	1	1	1	1	0

Chunk size is set to 2.000.000 (computer #2) (inefficient allocation):

Test No.	Scheduling Method	Chunk Size	No. of Iterations	Timings in seconds for each number of threads							
				1	2	3	4	5	6	7	8
0	omp_sched_static	2000000	14	4	4	4	4	4	4	4	4
1	omp_sched_dynamic	2000000	14	4	4	4	4	4	4	4	4
2	omp_sched_guided	2000000	14	4	4	4	4	4	4	4	4

Chunk size is set to 200 (computer #2):

Test No.	Scheduling Method	Chunk Size	No. of Iterations	Timings in seconds for each number of threads							
				1	2	3	4	5	6	7	8
0	omp_sched_static	200	14	4	2	1	1	1	1	1	0
1	omp_sched_dynamic	200	14	4	2	1	1	1	1	1	0
2	omp_sched_guided	200	14	4	2	1	1	1	1	1	0

The names of the first 5 hosts that have the highest rankings:

(i) 4mekp13kca78a3hfsrb0k813n9
(ii) 0491md82hej8u15vi98isrmuih
(iii) 3165mii1s1g0invqs94q303v0v
(iv) 46o3c5beh6kiojkvr1tvsk4ptt
(v) 2494c7mt12frm3c3go86abe13h

As expected, execution time of the algorithm decreases gradually as the number of threads is increased. Although data at hand might be insufficient to make any concrete remarks about the hierarchy between different scheduling options, it seems that overhead of dynamic scheduling is more apparent at low levels of parallelism (e.g. thread count 2, 3). Yet this effect goes away as we increase the level of parallelism and even changes (slightly) in favor of dynamic and guided scheduling. It is important to note here that mentioned variance in execution times is rather small, which is an indicator that different iterations of the for loop inside the parallel region, distributed amongst threads, take more or less the same time. Hence taking away the need for dynamic scheduling, and the very apparent effects of the overhead. It's also worth noting that the execution time changes drastically from one architecture to another as can be seen from the tables above.