IN4200 Partial Exam

Candidate number: 15350

April 2018

Computing PageRank

The program consist of a PE_function_15350.c file and an PE_main_15350.c file. The PE_function_15350.c consist of three functions: read_graph_from_file(filename), PageRank_iterations(d,eps) and top_n_webpages(xk,top_n).

The read_graph_from_file() function takes in a file name and reads the file, It is importeant that the file have the same layout as the web_NotreDame.txt file!. The important variables that the function gets from the file is n_nodes and n_edges which comes from the line:

```
err = fscanf(file, "%*s %*s %d %*s %d", &n_nodes, &n_edges);
```

Were it ignores the strings. The loop that saves the values to F_nodeId and T_nodeId, while excluding self-links is:

```
int s = 0;
int a,b;
while(s < n_edges) {
  err = getline(&line, &len, file);
  err = fscanf(file, "%d %d", &a, &b);
  if(a == b){
    n_edges--;
    continue;
  }
  F_nodeId[s] = a;
  T_nodeId[s] = b;
  s++;
}
```

n_nodes and n_edges gets used all the time in the program, while the other two is used to generate the A matrix as CRS format with arrays: row_ptr:

```
int cnt = 0;
for(int n=0; n < n_nodes; n++){
```

```
    cnt += total_T[n]; //total_T is sum of equal values in T_nodeid
    row_ptr[n+1] = cnt;
  }
```

col_idx and values:

```
  for (int j = 0; j < n_edges; j++) {
    count = row_ptr[T_nodeId[j]] + temp[T_nodeId[j]];
    col_idx[count] = F_nodeId[j];
    values[count] = (double)1/total_F[F_nodeId[j]];
    temp[T_nodeId[j]]++;
  }
```

The function also finds the danglings point and location, which is used later in PageRank function.

The Page_Rank() function with input values d and ϵ, does the most of the calculation. The function iterates true a loop, were it calculates W and x and returns x, with the functions:

$$W^{k-1} = \sum_{m \in \mathcal{D}} x_m^{k-1}, \quad \mathcal{D} : \text{set of indices for all dangling webpages}$$

$$x^k = \frac{1 - d + d * W^{k-1}}{N} * \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} + d * \boldsymbol{A} x^{k-1}$$

and stops at the stopping criterion:

$$\max_{1 \le j \le N} \quad |x_j^k - x_j^{k-1}| \quad < \epsilon$$

The maximumly achievable (The big O) speed of the page iterations depends on the for and while loops: The first for loop gives O(n), and from the while loop we get O(k*w+n*n), all this gives: O(n) + O(k*w+k*n*n) $\approx O(n^2)$. The factors that prevent reaching the ideal speed is that the threads runs in different speed and, some threads need to wait for them to finish before they can do their job. The way to calculated the iterations is inside a while loop who runs up to the stopping criterion hits, runs parallel:

```
    while (!array_ok) {
    array_ok = true;
    k++;
    sum_wk = 0;
    // calculating W
    #pragma omp parallel for reduction(+:sum_wk)
    for(int i=0; i<w_dang;i++){
      sum_wk += xk_1[w_place[i]];
    }

    tamp = (1 - d + (d * sum_wk))/((double)n_nodes);
```

```
    int j;
    //calculating x
    #pragma omp parallel for private(j)
    for (int i = 0; i < n_nodes; i++) {
      xk[i] = tamp; // calculating outside for loop since const
      for (j = row_ptr[i]; j < row_ptr[i+1]; j++) {
        xk[i] += d * (values[j] * xk_1[col_idx[j]]);
      }
      // Stopping crirerion
      if (fabs(xk[i]-xk_1[i])>eps){
        array_ok = false;
      }
    }
    //updating values
    tmp = xk_1;
    xk_1 = xk;
    xk = tmp;
  }
```

The last function **Top_n_webpages()**, gets the x from Page_Rank() and a chosen value from the user that indicates how many top values they want to print out. It start with calculating the first top value and uses it to find the others, and it only goes true the loop top_n times n_nodes time. The function is build with 2 parallel run, the first one calculates one top value and saves it as max, it is important to have a another variable outside the parallel to hold on the idx value, and to make sure that all the threads give out the max value:

```
  #pragma omp parallel
  {
    int idx=0;
    #pragma omp for reduction(max:max)
    for (int i = 0; i < n_nodes; i++) {
      if(xk[i] > max) {
        max = xk[i];
        idx = i;
      }
    }
    #pragma omp critical
    {
      if(xk[idx] == max){
        idx2 = idx;
      }
    }
  }

  top_n_index[0] = idx2;
  prevmax = max;
  max = 0;
```

The next loop goes true the program again and compares the value with new max and so on.

```
for (int i = 1; i < top_n; i++) {
    #pragma omp parallel
    {
      int idx =0;
      #pragma omp for reduction(max:max)
      for (int j = 0; j < n_nodes; j++) {
        if(xk[j] > max && xk[j] < prevmax) {
          max = xk[j];
          idx = j;
        }
      }
      #pragma omp critical
      {
        if(xk[idx] == max){
          idx2 = idx;
        }
      }
    }
      top_n_index[i] = idx2;
      prevmax = max;
      max = 0;
}
```

Results and OPenmp

As shown in the README.md file it is many ways to run the program, but i used the same optimization option, "-O2 -fopenmp" in every run with the same damping constant and only top 10 values were printed. I have also tested the program without any OpenMP, which takes a bit more time. When running the program with $\epsilon = 1e-10$ we get 89 iterations before the stopping criterie hits, and when using bigger ϵ it will be less iterations and more iteration when using smaller ϵ. The only changes was the OMP_NUM_THREADS. My pc have 4 cpu cores, and each cpu core can use two threads, that means that i should not run more then 8 threads. A run of the program with OMP_NUM_THREADS=8 (which gives the best running time):

```
15350 > export OMP_NUM_THREADS=8
15350 > gcc-8 -O2 -fopenmp PE_main_15350.c
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10
Reading file...
File read and closed...
Finding the CRS arrays ...
Finding dangling points ...
Total dangling points are: 188795
Iterating true x_k...
Calculating xk ...

Total  iterations : 89
Finding top 10 values...
    PAGE    | MAX VALUE
    1964  | 0.005627
       1  | 0.005405
  124803  | 0.003326
```

```
   12130  | 0.002857
  191268  | 0.002749
   32831  | 0.002732
    3452  | 0.002590
   83607  | 0.002460
    1974  | 0.002376
  142733  | 0.002341
Reading graph from file: took 0.487371 seconds
PageRank iterations:     took 0.234013 seconds
Top n webpages:          took 0.003070 seconds
ALL DONE!!,      TOTAL TIME = 0.724474 seconds
```

When running the program with 1,2 and 4 Threads, we se that the Page iteration function uses less and less time:

```
15350 > export OMP_NUM_THREADS=1
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10
Reading graph from file: took 0.481889 seconds
PageRank iterations:     took 0.363762 seconds
Top n webpages:          took 0.003147 seconds
ALL DONE!!,      TOTAL TIME = 0.848815 seconds

15350 > export OMP_NUM_THREADS=2
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10
Reading graph from file: took 0.485228 seconds
PageRank iterations:     took 0.278124 seconds
Top n webpages:          took 0.003123 seconds
ALL DONE!!,      TOTAL TIME = 0.766499 seconds

15350 > export OMP_NUM_THREADS=4
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10
Reading graph from file: took 0.489218 seconds
PageRank iterations:     took 0.238720 seconds
Top n webpages:          took 0.002538 seconds
ALL DONE!!,      TOTAL TIME = 0.730501 seconds
```

Ther is no point to try more threads, it will only take more times to load the threads then running the program. We see that it almost takes same time for 8 and 4 number of threads. Her is a run of the program with 10000 top n values with and without OpenMP:

```
15350 > gcc-8 PE_main_15350.c -fopenmp
15350 > export OMP_NUM_THREADS=4
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10000
Reading graph from file: took 0.515665 seconds
PageRank iterations:     took 0.607381 seconds
Top n webpages:          took 5.016241 seconds
ALL DONE!!,      TOTAL TIME = 6.139315 seconds

15350 > gcc-8 test2.c
15350 > ./a.out web-NotreDame.txt 0.85 1e-10 10000
WITHOUT OPENMP
Reading graph from file: took 0.529530 seconds
PageRank iterations:     took 1.130368 seconds
Top n webpages:          took 9.358309 seconds
ALL DONE!!,      TOTAL TIME = 11.018255 seconds
```

The time using OpenMP is much better then without openmp!.