IN-4200
High-Performance Computing and Numerical Projects

Project 1 - Computing PageRank

Fred Marcus John Silverberg

03 April 2019

Contents

1 Introduction

The aim of the project is to efficiently implement the PageRank algorithm used to highlight the most important webpages throughout a set of webpages. Focus is directed towards creating parallelized code with help of openMP. Generally, the project is supposed to be highly efficient in terms of computational speed and usage of available hardware. A deeper knowledge of theoretical and practical efficiency is also introduced.

2 Structure and implementation

2.1 File structure

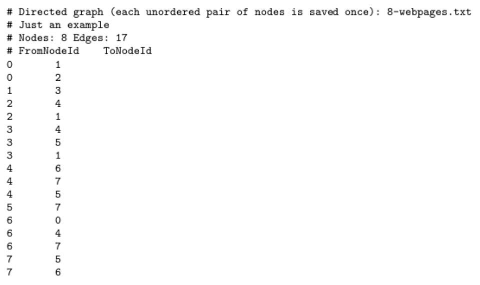
```
PE_main_15319.c
 - Contains the main code of the project
PE_functions_15319.c
 - Contains the functions used in the project
PE_pdf_15319.pdf
 - Contains this file
README.txt
 - Contains information about the project and how to compile
```

2.2 Data structure and Implementation

The raw data contains the number of webpages as 'Nodes' and the number of connections between webpages as 'Edges'. The column 'FromNodeId' holds information of the origin node of a outgoing connection, while the column 'ToNodeId' holds information about the destination node. this can be though of as a hyper-matrix: 'A[To,From]'. An example is show in figure[1]

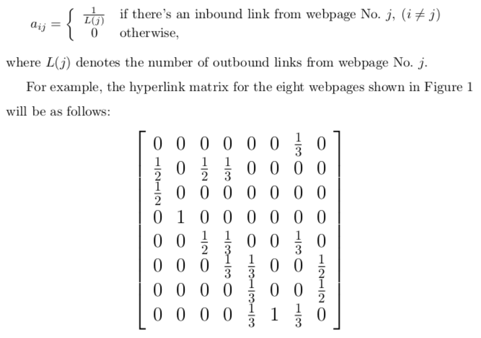
Figure 1

(a) Raw data example



```
# Directed graph (each unordered pair of nodes is saved once): 8-webpages.txt
# Just an example
# Nodes: 8 Edges: 17
# FromNodeId    ToNodeId
0       1
0       2
1       3
2       4
2       1
3       4
3       5
3       1
4       6
4       7
4       5
5       7
6       0
6       4
6       7
7       5
7       6
```

(b) Hyper-matrix example



$$a_{ij} = \begin{cases} \frac{1}{L(j)} & \text{if there's an inbound link from webpage No. } j, \ (i \neq j) \\ 0 & \text{otherwise,} \end{cases}$$

where $L(j)$ denotes the number of outbound links from webpage No. j.

For example, the hyperlink matrix for the eight webpages shown in Figure 1 will be as follows:

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

The raw data is obtained in an unsorted compressed storage fashion, eliminating storage of unnecessary elements in the data. This is achieved by the functions:

`* read_graph_from_file_0`

This function take in the filename and scans the data file row by row until the number of 'Nodes' and 'Edges' has been extracted.

`* read_graph_from_file_1`

This function receives three empty arrays. The file is once again scanned row by row until the end of the file. During this operation the data is extracted into the arrays and information about dangling webpages is obtained.

`* read_graph_from_file_2`

This last function uses the extracted data and constructs the array "val", representing the compressed hyper-matrix 'A[To,From]'.

The PageRank algorithm is then applied by the function:

`* PageRank_iterations`

This function takes the extracted data in unsorted compressed form and preform an iteration, implementing the PageRank algorithm described in:
`http://home.ie.cuhk.edu.hk/~wkshum/papers/pagerank.pdf`
When the final results have been obtained the scores can be found at the local address: `"../scores_unsorted.txt"`

A search for the number [M] of top webpages is started by the function:

`* top_n_webpages`

This function takes the final results and iterate through the list [M] times, obtaining the lists maximum score. For each iteration the score and representing index are stored and afterwards the identified score is 'deleted'. The same procedure continues until all [M] pages has been found, the results can then be found locally at: `"../scores_top.txt"`

3 Compiler and Hardware

```
* Compiler:
gcc - 4:7.3.0-3ubuntu2.1 - amd64 - GNU C compiler

* Hardware:
Intel i5-6500 Skylake: 3.2 Ghz
Operation capacity:     16 DP operations/s
Max memory bandwidth: 34.1 GB/s
CPUs:                    4
Threads:                 4
Cache:                   6 MB SmartCache
```

Source:`https://ark.intel.com/content/www/us/en/ark/products/88184/intel-core-i5-6500-processor-6m-cache-up-to-3-60-ghz.html`

4 Discussion

4.1 Theoretical vs practical speed for the PageRank function

My theoretical peak performance is: 3.2*16*4 = 204.8 $Gflops/s$.
In this discussion I will take the NotreDame webgraph as an example.
Edges: 1469679, Nodes: 325729

Table 1: Flops in each loop

Loop	Flops	Iterations	Total Flops
1	1	325729	325729
2	3	1469679	44090337
3	3	325729	977187

Total amount of flops for all loops during one 'while' iteration is then 0,045 $Gflops$. If reaching the error threshold requires 45 iterations we have a computational load of: 45*0,045 = 2,025 $Gflops$ The estimated peak performance time would then be 2,025/204.8 = 0,01 s.

However, this is not the case since we have to deal with latency, which is the time it takes for a none-byte transfer, or in other words the communication delay between parts (ns). In addition we need to take memory bandwidth in to the account, which is the rate at which data can be transferred from memory into the CPU ($Gbytes/sec$).

For a single value W the updated value of W stays in the CPU during the loop, by that logic, I assume that the updated values for an element in an array $X[i]$ also stays in the cache until the loop is finished, hence a store transfer for each iteration is not needed between RAM and cache. This is then restrained to the size of the objects.

Table 2: Size of objects

Objects	Size	Larger than cache?
val	11.7 MB	Yes
X_1	2.6 MB	No
X	2.6 MB	No
$W, d, term_1, delta$	0,000008 MB	No

Maximum achievable performance (P) can then be calculated by using machine balance (Bm) and code balance (Bc):

```
Bc           = data traffic / flops
Bm           = bmax / pmax
P            = min(pmax,bmax/Bc)
bmax         = 34.2 GB/s / 8  = 4.25 GWords/s
pmax         = 3.2*16*4  = 205 Gflops/s
flops        = (+,-,/,*) operators
data_traffic = Data transfers from slow memory (RAM) to fast memory (Cache)
```

Table 3: Loop structure and calculations

Loop	Flops	Data transfers	Bc	P	P<<pmax (?)
1	1	None	0/1	205 Gflops/s	No
2	3	'val'	1/3	12.8 Gflops/s	Yes
3	3	None	0/3	205 Gflops/s	No

If $[p << pmax]$ a loop is not optimized well and heavily memory bounded. An optimization idea for loop (2) would be an unrolled loop:

```
for (k = 0; k<*Edges; k++)
{
 X_1[row_idx[k]]   += ((*d)*val[k]*X[col_idx[k]]);
 X_1[row_idx[k+1]] += ((*d)*val[k+1]*X[col_idx[k+1]]);
 X_1[row_idx[k+2]] += ((*d)*val[k+2]*X[col_idx[k+2]]);
}
```

However, this is not possible since rowidx[k] could be the same as rowidx[k+2]. In such case the summation would end up to be wrong.

Table 4: Speed of the PageRank iteration

Loop	Iterations	Flops x 45	Theoretical time
1	325729	0,0147 $Gflops$	0.0147/205 = 0,00007 s
2	1469679	1.98 $Gflops$	1.98/12.8 = 0,154687 s
3	325729	0.44 $Gflops$	0.044/205 = 0,00021 s

As can be seen, the total theoretical time is then: 0,16 s

The theoretical speed of 0,16 s compared to the practical speed of 0.5 s highlights quite a difference. The theoretical speed assumes that the code uses all arithmetic units and cores in an optimal fashion. Even if the code is parallelized with openMP, the cores are not fully utilized during the whole 'while' iteration. It also assumes that data transfer and arithmetic overlap smoothly as well as neglegtion of latency. There is also a few if-statements, fabs and "pre-calculations" in the 'while' loop that contributes to the difference.

4.2 Time measurements for the PageRank function

The PageRank function have three loops (1,2,3). By openMP loop(1) and loop(2) was parallelized. The time data was collected by running the code five times at each thread setting and here presented with the mean value.

Table 5: Time measurements for various threads

Threads	t1	t2	t3	t4	t5	Mean	Efficency factor
1	0.512	0.517	0.509	0.512	0.511	0.5122	1.000
2	0.491	0.463	0.457	0.457	0.456	0,4648	1.102
3	0.440	0.437	0.454	0.450	0.439	0,4440	1.164
4	0.510	0.467	0.465	0.502	0.506	0,4900	1.045

It is obvious that the parallelization yielded some efficiency, especially for utilization of 3 threads. Notice how the efficiency decreases for the maximum nr of 4 threads, due to the need of a master thread which in turn results in constant spawning and killing of threads. Another setup was tested, were loop(1) was inserted in loop(3), but this decreased the efficiency due to the limitations of threads and was in the 0.55 s range.

4.3 Programming issues for optimization

A few programming issues observed while writing the code was:

1. Scanning the webgraph file twice since information about Edges and Nodes was needed for proper allocation of the data arrays. I assume that this is a problem an experienced c writer would solve.

2. The CRS configuration by the textbook implicit suggested a sorted webgraph or a sorting function of the data. Since we were to assume that the webgraph was not sorted and that a sorting algorithm is computational heavy, an unsorted version was implemented. However, further optimization was not found for that version and parallelization could not be applied to the heavy MVM operation.

3. The usage of 'fabs' in the error calculation, it seems to be the case that openMP do not like this extra function. A test was made on squaring the difference (X1-X), eliminating the need for 'fabs', but that yielded less accurate results.