THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 4

Due: 6:00pm Friday, 6 June 2014

This assignment is worth 10% of your final assessment

Task description

In this assignment we will implement the basic PageRank algorithm in the C programming language using a variety of parallel programming techniques to ensure peak performance is achieved.

The PageRank algorithm was developed in 1996 by Larry Page and Sergey Brin when they were graduate students at Stanford University. Google and other search engines compare words in search phrases to words in web pages and use ranking algorithms to determine the most relevant results.

PageRank assigns a score to a set of web pages that indicates their importance. The underlying idea behind PageRank is to model a user who is clicking on web pages and following links from one page to another. In this framework, important pages are those which have incoming links from many other pages, or have incoming links from other pages with a high PageRank score, or both.

Fetching the assignment files

To obtain the files required for this assignment, run the following command on `ucpu1`

```
$ /labcommon/comp2129/bin/get-assignment4
```

Hint: Develop and run your code on the undergraduate server to access the included testing tools.

Academic declaration

By submitting this assignment you declare the following:

The PageRank algorithm

PageRank is an iterative algorithm that is repeated until a stopping criteria is met. The last iteration gives us the result of the search, which is a score per web page. A high score indicates a very relevant web page whereas a low score indicates a not so relevant web page for a search. Sorting the web pages by their scores in descending order gives us the order for the result list of a search query.

For describing the PageRank algorithm we introduce the following symbols:

- S is the set of all web pages that we are computing the PageRank scores for
- $N = |S|$ is the total number of web pages
- $\mathbf{P} = [P_1^t, P_2^t, ..., P_N^t]$ is the vector of PageRank scores
- d is a dampening factor for the probability that the user continues clicking on web pages
- ϵ is the convergence threshold
- $\textsc{in}(p)$ is the set of all pages in S which link to page p
- $\textsc{out}(p)$ is the set of all pages in S which page p links to

For the PageRank vector, we use the notation $\mathbf{P}_p^{(t)}$ to represent the PageRank score for page p at iteration t. We initialise the scores for all pages to an initial value of $\frac{1}{N}$ so that the sum equals 1.

$$\mathbf{P}_u^{(0)} = \frac{1}{N} \tag{1}$$

During each iteration of the algorithm, the value of $\mathbf{P}$ is updated as follows:

$$\mathbf{P}_u^{(t+1)} = \frac{1-d}{N} + d \sum_{v \in \textsc{in}(u)} \frac{\mathbf{P}_v^{(t)}}{|\textsc{out}(v)|} \tag{2}$$

The algorithm continues to iterate until the convergence threshold is reached; that is, the algorithm terminates when PageRank scores stop varying between iterations. The PageRank scores have converged when the following condition is met:

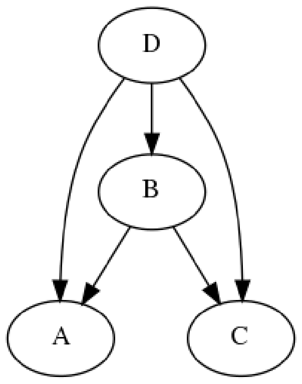
$$||\mathbf{P}^{(t+1)} - \mathbf{P}^{(t)}|| \leq \epsilon \tag{3}$$

The vector norm is defined to be the standard Euclidean vector norm; that is,
for some vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$:

$$||\mathbf{x}|| = \sqrt{\sum_i x_i^2} \tag{4}$$

Example

Our example has four web pages: $S = \{A, B, C, D\}$. In this example, $d = 0.85$ and $\epsilon = 0.005$. The referencing structure of the web pages A, B, C, and D is given in the graph below.



$$\begin{array}{rclcrcl}
\text{IN}(A) & = & \{B, D\} & & \text{OUT}(A) & = & \emptyset \\
\text{IN}(B) & = & \{D\} & & \text{OUT}(B) & = & \{A, C\} \\
\text{IN}(C) & = & \{B, D\} & & \text{OUT}(C) & = & \emptyset \\
\text{IN}(D) & = & \emptyset & & \text{OUT}(D) & = & \{A, B, C\}
\end{array}$$

Each node represents a web page.
Edges in the graph indicate that the source of the edge is linking to the destination of the edge.

Initialise $\mathbf{P}^{(0)} = \frac{1}{N}$. Then perform the first iteration for each page.

$$\begin{aligned}
\mathbf{P}_A^{(1)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(0)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(0)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.25}{2} + \frac{0.25}{3}\right) \approx 0.214 \\
\mathbf{P}_B^{(1)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_D^{(0)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.25}{3}\right) \approx 0.108 \\
\mathbf{P}_C^{(1)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(0)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(0)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.25}{2} + \frac{0.25}{3}\right) \approx 0.214 \\
\mathbf{P}_D^{(1)} &= \frac{1-0.85}{4} + 0.85\,(0) = \frac{0.15}{4} + 0 \approx 0.038
\end{aligned}$$

The initialisation and first iteration result in the following values for $\mathbf{P}$:

t	A	B	C	D
0	0.250	0.250	0.250	0.250
1	0.214	0.108	0.214	0.038

Next, we check whether or not the algorithm has converged.

$$\begin{aligned}
||\mathbf{P}^{(1)} - \mathbf{P}^{(0)}|| &= ||\{-0.036, -0.142, -0.036, -0.215\}|| \\
&= \sqrt{0.0677} \\
&\approx 0.260
\end{aligned}$$

Since $0.260 \not\leq 0.005$ (ϵ), we perform another iteration.

$$\begin{aligned}
\mathbf{P}_A^{(2)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(1)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(1)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.108}{2} + \frac{0.038}{3}\right) &\approx 0.094\\
\mathbf{P}_B^{(2)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_D^{(1)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.038}{3}\right) &\approx 0.048\\
\mathbf{P}_C^{(2)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(1)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(1)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.108}{2} + \frac{0.038}{3}\right) &\approx 0.094\\
\mathbf{P}_D^{(2)} &= \frac{1-0.85}{4} + 0.85\,(0) &= \frac{0.15}{4} + 0 &\approx 0.038
\end{aligned}$$

Which leaves us with the following three iterations of $\mathbf{P}$:

t	A	B	C	D
0	0.250	0.250	0.250	0.250
1	0.214	0.108	0.214	0.038
2	0.094	0.048	0.094	0.038

Again, we check whether or not the algorithm has converged:

$$||\mathbf{P}^{(2)} - \mathbf{P}^{(1)}|| \approx 0.180 \not\leq \epsilon$$

Since convergence has not been reached, we perform another iteration.

$$\begin{aligned}
\mathbf{P}_A^{(3)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(2)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(2)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.048}{2} + \frac{0.038}{3}\right) &\approx 0.069\\
\mathbf{P}_B^{(3)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_D^{(2)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.038}{3}\right) &\approx 0.048\\
\mathbf{P}_C^{(3)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(2)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(2)}}{|\{A,B,C\}|}\right) &= \frac{0.15}{4} + 0.85\left(\frac{0.048}{2} + \frac{0.038}{3}\right) &\approx 0.069\\
\mathbf{P}_D^{(3)} &= \frac{1-0.85}{4} + 0.85\,(0) &= \frac{0.15}{4} + 0 &\approx 0.038
\end{aligned}$$

Again, we check whether or not the algorithm has converged:

$$||\mathbf{P}^{(3)} - \mathbf{P}^{(2)}|| \approx 0.040 \not\leq \epsilon$$

Since convergence has not been reached, we perform another iteration.

$$\begin{aligned}
\mathbf{P}_A^{(4)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(3)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(3)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.048}{2} + \frac{0.038}{3}\right) \approx 0.069 \\
\mathbf{P}_B^{(4)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_D^{(3)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.038}{3}\right) \approx 0.048 \\
\mathbf{P}_C^{(4)} &= \frac{1-0.85}{4} + 0.85\left(\frac{\mathbf{P}_B^{(3)}}{|\{A,C\}|} + \frac{\mathbf{P}_D^{(3)}}{|\{A,B,C\}|}\right) = \frac{0.15}{4} + 0.85\left(\frac{0.048}{2} + \frac{0.038}{3}\right) \approx 0.069 \\
\mathbf{P}_D^{(4)} &= \frac{1-0.85}{4} + 0.85\,(0) = \frac{0.15}{4} + 0 \approx 0.038
\end{aligned}$$

Again, we check whether or not the algorithm has converged:

$$||\mathbf{P}^{(4)} - \mathbf{P}^{(3)}|| = 0 \leq \epsilon$$

We have now reached convergance, so the algorithm terminates and the values of $\mathbf{P}^{(4)}$ are the final PageRank scores for each of the pages. If this were a query, the resulting ranks would be A, C, B, D where we arbitrarily rank page A before page C since they have the same score.

Implementation details

In the header file **pagerank.h** we have provided the function **`read_input`** that will process the input file for you. The **`read_input`** function will output **error** if the input is malformed in any way and will free any memory that has been previously allocated. We have also provided various structs in the header file that may be useful, including a struct for holding pages and a struct that is a linked list of pages. The function **`read_input`** returns the input data in these structs, with the following form:

```c
/* singly linked list to store all pages */
struct list {
  node* head; /* pointer to the head of the list */
  node* tail; /* pointer to the tail of the list */
  int length; /* length of the entire list */
};

/* struct to hold a linked list of pages */
struct node {
  page* page; /* pointer to page data structure */
  node* next; /* pointer to next page in list */
};

/* struct to hold a page */
struct page {
  char name[21]; /* page name */
  int index;      /* index of the page */
  int noutlinks; /* number of outlinks from this page */
  list* inlinks; /* linked list of pages with inlinks to this page */
};
```

Warning: Do not add other files or modify the included header file or main function.
We will replace these files with our own copy when marking your code.

Program input and output

```
<number of cores>
<dampening factors>
<number of pages>
<name of web page>
...
<number of edges>
<source page> <destination page>
...
```

The first line specifies the number of cores that are available on the machine. You will need to take this into account when you are optimising the performance of your program when using parallelism.

This is followed by the the dampening factor to be used in the algorithm and the total number of web pages. The names of the web pages follow, one per line, where each name may be a maximum of 20 characters long. After declaring the names of the web pages, the number of edges in the graph is given, followed by each edge in the graph specified by its source and destination page.

The **`read_input`** function outputs **error** and will terminate if there is invalid input, or there are any memory allocation errors, or a page name exceeds the maximum length, or the dampening factor is not in the range $0 \leq d \leq 1$, a page is declared twice, or an edge is defined to a nonexistent page.

```
2
0.85
4
A
B
C
D
5
D A
D B
D C
B A
B C
```

This example has four web pages with names A, B, C, and D
There are five edges: $D \rightarrow A$, $D \rightarrow B$, $D \rightarrow C$, $B \rightarrow A$, and $B \rightarrow C$

The output is the list of scores per web page

```
A 0.0686
B 0.0481
C 0.0686
D 0.0375
```

The scores are ordered in the same order they were defined in the input.

For printing the score, use the format string **`"%s %.4lf\n"`**
For all test cases set $\epsilon = 0.005$, use the constant defined as **`EPSILON`**

The Makefile

We have provided you with a Makefile that can be used to compile and test your code.
The included Makefile will only run correctly on the undergraduate server and lab machines.

- `$ make`
  will compile your source file into an executable file `pagerank`
- `$ make rain`
  will compile your program and check whether your program has any memory leaks using the test cases that you define in your `tests/` directory. If any test cases require investigation then run valgrind on each test, e.g. `$ valgrind ./pagerank < sample.in`
- `$ make test`
  will compile your program and check the correctness of your program against the input and output files that you created in your `tests/` directory. If any test cases require investigation then run sdiff on each test, e.g. `$ sdiff sample.obs.out sample.exp.out`
- `$ make submission`
  will submit your program to the benchmarking queue and also for marking

Marking

In this assignment correctness and performance are equally weighted. However, your submission will only receive marks for the performance component if it passes all of the correctness test cases.

Before you attempt to write optimised code, you should first complete a working unoptimised version. Once you have the basic algorithm correct you can then begin to implement various optimisations.

We have provided a set of test cases in the included `tests/` directory. It is important that you thoroughly test and benchmark your code on both your local machine and the submissions page. In this assignment only **valid inputs** are tested, and your program will be checked for memory leaks. If your program leaks memory you will not pass the test case even if the output is correct. In addition to the provided tests we will run your program against a substantial collection of hidden test cases.

5 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will use our own hidden test cases that cover every aspect of the specification. To pass test cases your solution must produce the correct output within the imposed time limit.

5 **marks** are assigned based on the *performance* of your code relative to other students.
This component is tested on a separate machine in a consistent manner. The fastest submission will receive 5 marks, with successively slower solutions receiving lower marks. Any student who is faster than our own parallel reference implementation will receive at least 2.5 marks.

Warning: Any attempts to deceive the automatic marking will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specification, or your code is unnecessarily or deliberately obfuscated.