CSC352 Project 1

Spring 2010
Yang Li

In this project, I implemented a searching heuristic in both the multi-threading approach and the multi-processing approach. The performance comparison between the two implementations shows little difference.

This program queries a database [1] using a search term and retrieves 20 urls results. For each url, it reads the linked file and computes a relevance score, which is a function of the frequency ranking of the search term relative to other words in the same file. The output of this program is a list of search results ranked by their relevance scores. Each entry also shows some context from the file where the searched word are first encountered.

The two parallel implementations are very similar. Both of them have a sequential part for querying the database and displaying the output, and a parallel part that uses 20 threads or processes for retrieving individual files and computing the scores . The code are based on my solution to Homework 2, with the additional feature of finding word frequencies, scoring search results, and changing the output. Note that this program often does not yield all 20 search results, since some of the urls are broken. This can result in HTTP errors and is ignored by the program using an exceptions handler. Also, it does not search for multiple keywords, hence any extra command-line arguments are be ignored.

Sample output of `proj1MultiProc.py: (proj1Thread.py produces exactly the same result.)`:

```
$ ./proj1MultiProc.py love

---------------------------
Rank:  0 Relavence: 1.0
fetching url: http://xgridmac.dyndns.org/~thiebaut/www_etext_org/Zines_9537/ASCII/UXU/
uxu-259.txt
that the most important thing in the
world is love. The poor know that it is

---------------------------
Rank:  1 Relavence: 1.0
fetching url: http://xgridmac.dyndns.org/~thiebaut/www_etext_org/Zines_9537/ASCII/ATI/
ATI124.TXT
love love lo ve love love lo
...
---------------------------
Rank:  19 Relavence: 0.025
fetching url: http://xgridmac.dyndns.org/~thiebaut/www_etext_org/Religious_357/
Polyamory/Keys2LovingUnity.html
CA 92244 <BR> oldservant@delphi.com <BR>
lovenchosen@hotmail.com<BR> <BR>
&quot;No purer
```

1. Swish-e database used in this assignment: http://xgridmac.dyndns.org/~thiebaut/swish-e/swishe.php

```
===========================
1 url(s) can not be opened.
```

To measure the performance, I tested the two implementations on a Dell XPS laptop with duo processors and Linux OS. I used a script to record the average and maximal running time of the two programs 3 times with search term "love" and 3 times with search term "parallelism". The script also computes the average and least number of search results retrieved per second, (i.e. 20 entries divided by the running time).

Sample output of the script running `proj1MultiProc.py with search term "love":`

```
$ ./timeIt.py
------ Time spent for retrieving 20 messages ------
best time: 9.180000  average time: 9.263333
---------- Number of messages per second ----------
max: 2.178649  average: 2.159050
```

The results are recorded the results as below:

Search term: love

	Average (entry/sec)	Maximum (entry/sec)
Multi-Processing	2.159	2.178
Multi-threading	2.258	2.262.

Search term: parallelism

	Average (entry/sec)	Maximum (entry/sec)
Multi-Processing	9.900	10.000
Multi-threading	10.380	10.471

According to the above table, the number of searches per second in both programs are quite stable when the search items are the same. Also, the multi-threaded version of the program appears to be a little more efficient than the multi-processing version, however, this is not a significant difference. One may notice that search term "Parallelism" receives the result much faster than search term "Love" does. In fact, 13 out of 20 urls returned from querying "Parallelism" can not be opened. This shows that the most time consuming part of this program is analyzing the file after retrieving it.