Project Writeup

A Turbocharged Web Crawler built on V8

ECSE 420 -- Parallel Computing

Group 1

Fall 2011

Patrick Desmarais (260329253)

Iain Macdonald (260270134)

Guillaume Viger (260309396)

Presented to: Michael Rabbat

1. Problem Statement and Motivation

Our group designed and implemented a web crawler this semester. We investigated techniques that would allow us to fetch a webpage, gather the URL links on the page, and recursively repeat this procedure for all URLs discovered, storing our results in a graph data structure, with nodes representing web pages, and directed edges links between these webpages.

This is by no means an original endeavour; there are many companies using web crawlers, including Google, Microsoft, Baidu, and others. There are many different reasons for wanting to crawl the web, from commercial to private. One might want to index all content on the Internet, or create a graph of the interconnections between webpages, for the purpose of searching for content on the web.

Crawling the web is a problem which is highly parallel in nature. The sheer size of the Internet, and the volume of content online is reason enough to investigate parallel solutions. Furthermore, there is no clear starting point, or ordering of the webpages, rather the simple requirement that they should all be fetched and parsed exactly once. Using a simple graph search algorithm such as breadth-first offers a naturally parallel implementation pattern.

Any efficient web-crawler must exploit parallelism, and in that regard our solution doesn't distinguish itself either. There is a clear indication that the process of fetching webpages, parsing them for URLs, and storing these URLs is a very time consuming process, largely due to the network and disk access costs. Indeed memory access and computations are in the order of units of CPU cycles, whereas disk access can be in the millions of CPU cycles and network access in the hundreds of millions of CPU cycles [1]. Coupled with the sheer amount of work to do—Google's index of the web is at least 50 billion webpages [2]—these bottlenecks are motivation enough (if not requirements) to use parallelism in solving this problem.

Our solution is original in how we approached the problem, how we came to solve it, and the tools we used. We employed Nodejs and MongoDB to parallelize the tasks involved which take up the most time, namely network access. Unfortunately, Nodejs is not as effective at parallelizing serial computation, such as regular expression application, so this portion of our program remained a serial bottleneck. These new technologies allowed us to reach incredible speeds at the cost of some complexity in the software. In some sense we have built a proof of concept for home web crawling, subject to bandwidth restrictions. In this report, we discuss the tools we used, the architecture of our software, some performance results and testing benchmarks, and look at the parallel nature of the code.

2. Solution structure

Framework

Our web crawler is written in the Nodejs programming language. Nodejs is an event-driven implementation of the JavaScript programming language. Nodejs programs run and are influenced by events, which are created by the operating system, and handled by the program. The runtime environment is highly parallel, with an implicit event-loop, ensuring events are dealt with serially while blocking calls are dealt with asynchronously. This parallel framework is very good for web-accessing programs, and the http module in Nodejs is very efficient. Furthermore, Nodejs does not use locks, so no input/output operations block [3]. We chose to use Nodejs because it promises high parallelism with little overhead. Additionally, the http access modules in Nodejs are very efficient, and we wanted a platform that would permit us to access the web in parallel. Furthermore, two of the three members of our group were familiar with JavaScript at the beginning of the semester, so there was not as much training required as with some other options.

To store the results obtained from the higher-level downloading and parsing code, we used MongoDB. MongoDB is a noSQL database management system, which uses JavaScript as its primary interface and user shell, rather than the popular structured query language (SQL). MongoDB stores data using a binary serialization of the JavaScript Object Notation (JSON), which is known as BSON. It stores larger objects using GridFS, a specialized file storage mechanism. One of the primary focuses in MongoDB is on speed [4]. We elected to use MongoDB due to the focus on high-speed operation, in hopes that the database would be able to scale well with a highly parallel web crawler. We were curious to see how high performance MongoDB was in practice, with real world data, and compare this performance with standard database mangement systems like MySQL, PostgreSQL, or SQLite. None of the members of our group had used MongoDB before this project, and we were excited to write a project using it, to get a bit of experience on a new and upcoming platform.

Architecture

The Nodejs language and runtime environment are inherently parallel, since they are based on the concept of executing all normally blocking function calls in parallel. The major challenges in designing Nodejs programs are efficiently distributing work using callback functions, and coordinating implicitly parallel operations. This approach is quite different from other environments, such as C programming with POSIX threads, Cilk, OpenMP, or CUDA, in that the parallel nature of the program is not explicitly delegated by the programmer, but abstracted away by Nodejs.

Figure 1 shows one iteration of web crawling with our program. The approach involves a simple breadth first search where a root web page leads to the exploration of its parsed references, or links on the page. Hence, each subtree of references is dynamically created by recursively parsing web pages of the URL found at their root. In order to record the relationships found for every web page based on their containing references, and also avoid possible cycles in the dynamic BFS search tree, each web page URL and all URLs linked to from that page are saved to a common database. Note that serializing data into the database presents an inevitable bottleneck in the program. However, not respecting database access timing could easily lead to cycles in the parse tree.

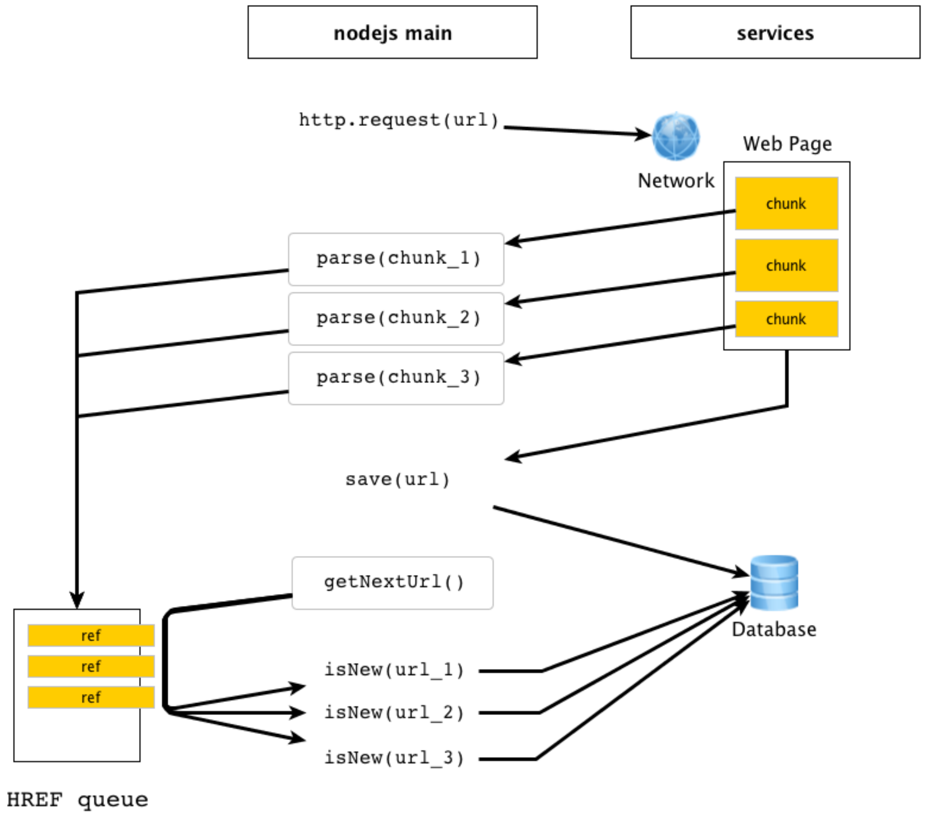

Figure 1: This is a diagram of one iteration of web crawling in our solution program. Each web page is requested using Nodejs' http module. Chunks of the web page are concurrently received and then parsed to find references. These references are saved to the database along with the url and placed in the HREF queue to be parsed later. This queue is emptied asynchronously at some time interval by the getNextUrl() call and checked against the contents of the database for creating the next subtree to be explored.

The lower-level database driver is a Nodejs program which uses the node-mongo-native binding. It offers simple facilities such as insertion, upsertion (insertion if the key does not exist, updating if it does), modification, deletion, and retrieval of data. All methods in the driver may be called synchronously, by passing a callback function as an argument to the program, or asynchronously, in which case the driver handles the data while the main execution thread continues. The driver attempts to be very efficient, while respecting the constraints of any database management system, and trying not to overload the database.

Discussion of Parallelism

The Nodejs http module offers a very nice feature to reach higher degrees of parallelism when parsing web pages. The request function of the http module has the ability to download webpages in chunks, and distribute these chunks to the callback function as soon as they are received. This allows the program to begin parsing a webpage before it has been entirely received, while it is fetching the rest of the webpage or other webpages. You will recall that parsing is the only serial bottleneck of our software. The program uses the parsing results to asynchronously save the URL and all parsed links to the database. Thus, the program can run in parallel using asynchronous database queries. In this sense, the program is using data decomposition to parallelize the task of retrieval. The webpage is divided into chunks at the source, and these chunks are passed through the program in parallel.

The database driver exploits parallelism, with concurrency managed by the Nodejs runtime environment. All methods in the driver may be called synchronously or asynchronously. This is done synchronously, by passing them as callbacks to functions using JavaScript anonymous function feature. This way execution of the database query will proceed once the operation has completed. Database queries may also be called asynchronously, in which case the driver will take care of the operation in the background while execution of the calling thread continues. Also note that functions can be passed as callbacks to the database driver methods in order to execute those synchronously after the database has been queried.

The database driver exploits the innate parallelism of Nodejs, but not without some drawbacks. MongoDB uses a global read-write lock to monitor access to the database. This means that under write-heavy load, such as in our application, a highly parallel driver is not possible, as most database accesses will block waiting for others. For this reason, the higher level program must synchronize database access. By staggering database access in time and gathering data obtained between accesses into chunks which are all added at the end of a stagger interval, this difficulty can be overcome. This requires some coordination by the higher-level program. This coordination is effectively a barrier which is triggered at set time intervals, rather than when all threads of execution reach the barrier.

Using this architecture on top of Nodejs, our web crawler can flip the advantages of parallelism around and allow network and disk access to be done in parallel while parsing is done concurrently. By effectively splitting each webpage into many smaller webpages, which can be handled much more quickly than requiring the program to wait for the download of an entire webpage before proceeding, we can mitigate the drawbacks of serial parsing and provide much higher speedups than other candidate solutions.

3. Analysis and experiments

Theoretical

If we model the time required to retrieve a webpage as $t_s + mt_w$, then the time required to retrieve one-kth of that webpage is $t_s + \frac{m}{k}t_w$. Let us also assume the time required to parse a webpage for all links using standard POSIX regular expression is proportional to the size of the webpage in bytes, with constant of proportionality c. For this analysis, we will consider retrieving and parsing a page with L links, and then retrieving and parsing each of those L pages (we assume L to be the same at each level). In serial, the time required would be:

$$(1 + L)[(t_s + mt_w) + cm]$$

More generally, going to depth d,

$$(1 + L^d)[(t_s + mt_w) + cm]$$

In a theoretically optimal version using parallelized network access, this would require time (for two levels):

$$2(t_s) + (1 + L)(k \times max\left\{\frac{m}{k}, \frac{cm}{k}\right\} \times t_w)$$

Assuming that we can parse a chunk of data faster than we can download it, this implies c < 1, making the parallel time as below:

$$2(t_s) + (1 + L)(mt_w)$$

Thus, the speedup we achieve is $\frac{1+L}{2}$ for one-level of depth. Generalizing to depth d,

$$d(t_s) + (1 + L^d)(mt_w)$$

The speedup of the theoretically optimal version of the web crawler is thus:

$$[(1 + L^d)((t_s + mt_w) + cm)] \ / \ [d(t_s) + (1 + L^d)(mt_w)]$$

It is difficult to calculate the efficiency of this system, as the main parallel benefit isn't in scaling over several processors, but rather in multiplexing the Internet connection, and parallelizing network access. This could be calculated assuming that processing elements refers to network cards, but with the important realization that this is also dependent upon the network access speed of each of these network cards. If we assume they all have constant network speed b bits per second, and we ignore the limiting factor in the access link connecting the machine to the rest of the network, we can derive an expression for efficiency in terms of these quantities and the number of network cards, n,

$$[(1+L^d)((t_s+\frac{m}{b}t_w)+cm)]\ /\ [d(t_s)+\frac{(1+L^d)}{n}(\frac{m}{b}t_w)]$$

Practical

Running the web crawler on a machine with a 2.8 GHz Intel Core 2 Duo, and the Mac OS X 10.7.2 operating system, for seven seconds, the program retrieved and parsed 830 chunks (of approximately 4096 bytes each), finding 102 URLs. During this execution, the program used six threads and the CPU usage on one core was at 100 per cent.

In a test of the database driver, a crawl was started at the McGill University home page, www.mcgill.ca, on a 7 Mbps connection on campus, and the database became overloaded within ten seconds. Introducing a stagger to database access alleviated this problem, and allowed the database to properly complete operations without flooding.

Results from execution to execution are highly variable, and depend on the network traffic and the order in which references are found in parsing and the order they are added to the queue. Requesting through Nodejs's http module also does not always provide consistent chunk sizes, so execution time also depends on the root page. For example, youtube.com respond with chunks of 4096 bytes, but alumnilive365.mcgill.ca returns chunks between 8000 and 11000 bytes.

Figure 2 shows the results of executing the program from the McGill campus, on the Mac OS X with a 2.8 GHz Intel Core 2 Duo, and a root page of www.mcgill.ca. For this execution, database accesses were staggered every 200 milliseconds. The program proceeded to run for five seconds, and found 169 unique URLs in this time.

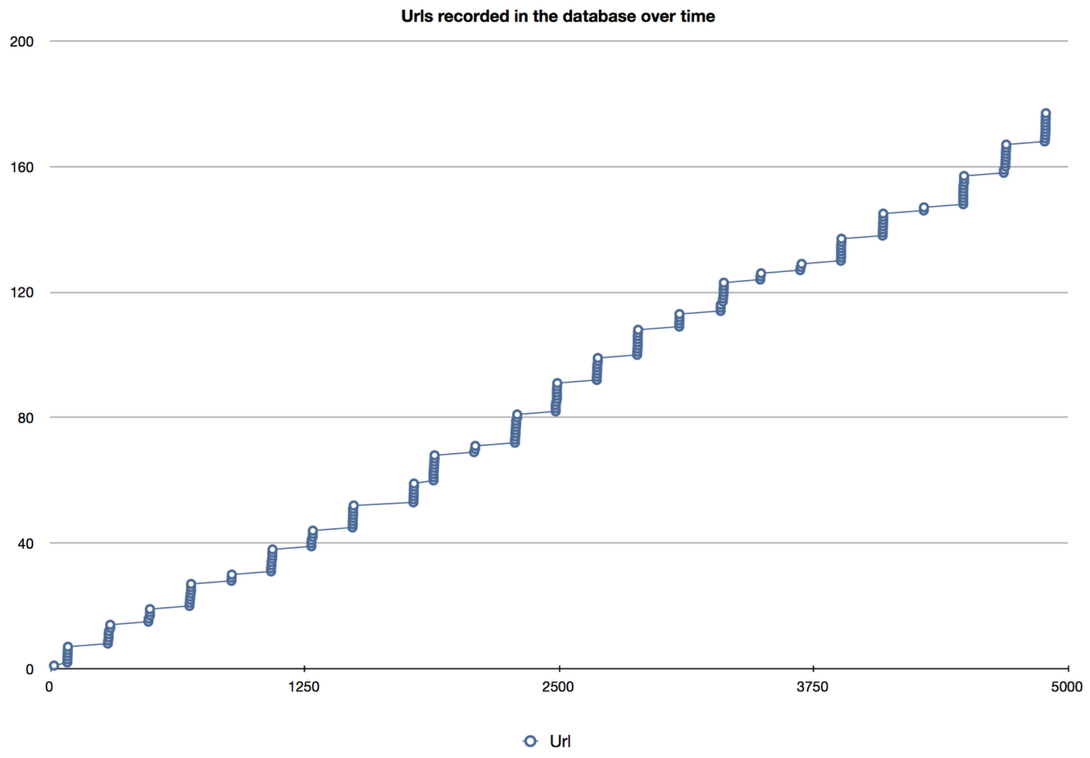

Figure 2: Web crawling results from a location close to campus, starting at the McGill home page. The process of at most 10 urls every 200 milliseconds for 5000 milliseconds resulted in 169 different found. The flat lines between every column of circles which represent unique URLs denotes the waiting period.

Validation

In order to test if our program was crawling the web correctly, we wrote a test script in Python using mongoDB bindings (through pymongo) that created a series of webpages (5000) and stored the relationship between the webpages (references) in a database. We then ran our program on this set of pages locally (to remove the potential networking issues and only test functionality). Another python script verified that the entries stored in the database were the same as the ones in the original database, the one created when the webpages were created. This process validated the basic functionality of our program.

Now, to verify that the results fetched from the Internet were correctly saved to the database, another Python script using the pymongo module read the collections entered in the database. The list of URLs sent to the database was compared to the one printed at the terminal. Our main objective was to observe the parallel performance of Nodejs for web crawling, so we didn't spend too much time to take into account all parsing cases, such as when chunks cut references in two. Hence, in the end, a good percentage, but not 100%, of references contained in web pages analyzed were stored.

Performance Evaluation

To analyze the previously cited examples, we observed the time at which chunks are parsed and web pages are fetched, as well as the size of chunks parsed. All of this information was printed to the terminal, for recording. CPU utilization information was observed using the 'top' command in a terminal, and the network connection speed was evaluated using bandwidthplace.com. Using all of this, we were able to gather information about the performance of our program. As stated earlier, our web crawler is capable of finding hundreds of URLs in only seconds, in one run it found 169 URLs in five seconds, in another, it found 102 URLs in seven seconds.

To analyze the performance of MongoDB, we conducted an experiment, comparing the performance of different database management systems using mock data modelling data from crawling the web. To generate the data, we created random string of length 100 characters, and called these the URLs. We took a subset of these strings and called this subset the base URLs, corresponding to webpages crawled. For each of these base URLs, we took three more of the random strings, and called these the pages referenced on the base URL. We added this information to the various database systems within a Python script, and measured the time taken for each database to perform the task.

The databases tested include MongoDB (version 1.4.4), MySQL (version 5.1.49), and SQLite (version 2.8.17). These were all installed from the Debian package repository on a Debian 6 machine running the Linux Kernel version 2.6.32-5. The machine has an Intel core 2 duo T9300 at 2.50 GHz, with 4 GB of DDR2 RAM at 667 MHz, and a SATA-2 hard disk at 5400 RPM.

For all database systems, we performed several tests. There were two fundamental families of testing: write-only performance, and write-then-read performance. In the write-only test, the data was passed to the database to be stored, and the time taken to do this was measured. In the write-then-read test, the write was performed, and then all data was read back from the database and compared with what was originally inserted. For SQLite, we performed each set on an in-memory database and an on-disk database. For each of these tests and each of the database systems, we performed the test with and without indexing frequently queried columns or fields, and with filesystem caching enabled and disabled. The results of the tests with indexing enabled, and filesystem caching disabled are shown below, in figure 3.

DBMS	Operation	Disk/Memory	Time (hh:mm:ss)
MongoDB	Write	Default	0:00:00.44
MongoDB	Write/Read	Default	0:00:03.08
MySQL	Write	Default	0:00:12.01
MySQL	Write/Read	Default	0:00:19.45
SQLite	Write	Disk	0:00:26.99
SQLite	Write	Memory	0:00:03.34
SQLite	Write/Read	Disk	0:02:14.87
SQLite	Write/Read	Memory	0:00:05.21

Figure 3: A comparison of different database systems, looking at the time taken to insert 10,000 records, and the time taken to insert and retrieve 10,000 records for each.

These results show that MongoDB is indeed very high performance, and well-suited for the type of data created in a web crawling application. Tests were performed with filesystem caching disabled, because in the case it is enabled, SQLite's performance for an on-disk and in-memory database are roughly equivalent, suggesting the database is actually being stored in memory. MongoDB narrowly outperforms SQLite in these tests, however, it did so consistently. Interestingly, the effect of indexing frequently queried columns is remarkable in the SQL databases. SQLite in-memory ran many thousands of times faster with these indices. The on-disk, non-indexed, read/write SQLite test had to be manually stopped after running for seven hours. Without indices, Mongo still performed very well, suggesting the out-of-the-box performance for Mongo is much better than that of SQL-based databases. It is worth noting that these tests were performed for data which model the structure obtained from a web crawling application, and are not necessarily generalizable. Full testing results can be found in our git repository, access to which is described in the appendix.

4. Discussion

Findings

Our goal was to design a webcrawler, with a secondary goal being to try out two new technologies: Nodejs and MongoDB, as both of these seem well-suited for crawling the web. This assumption proved to be true.

Nodejs is very fast and provides developers with the ability to use parallelism more naturally, despite the need to slightly change the programming paradigm in use. The performance gains make the Nodejs framework very appealing for many applications requiring a high degree of concurrency. Parallelizing access to database and network worked slightly too well requiring us to buffer access to the database due to a flood of simultaneous connections.

Similarly MongoDB proved to be a very high performance alternative to the more familiar family of SQL databases. Its ability to keep up with Nodejs (to some extent) was crucial, as other databases were shown to be too slow for our use. Additionally, MongoDB was much easier to set up than its SQL alternatives, and the data model much more intuitive. With a bit of research MongoDB also proved to be very easy to optimize.

Challenges

Despite choosing Nodejs because it abstracted away the process of creating threads explicitly and the task of dealing with granular synchronization as in C, there were still some issues to solve. Mainly, with Nodejs being a new technology (there has not yet been a 1.0 release), the driver work was left to us and we had to deal with an unpredicted high-level of concurrency. This is why we coded our own ‘driver’ to provide access to MongoDB. We also had to ungracefully throttle our application in order not to exceed the number of possible connections to the database. We discuss this aspect further next in the extensions section.

Extensions

The performance of the program could be enhanced by storing URLs and their references in chunks, rather than creating a single connection for every one of them. This would allow the program to deal with fewer database connections and let us remove the chunking time interval. The difficulty in creating this extension is that it requires the database driver maintain state, particularly, that it keep track of a database connection. This is difficult to do in Nodejs, due to the implicit parallelism, and difficulty in communicating between parallel threads of execution. We were satisfied with the performance of our driver, and program in general, and were impressed with the liberal use of parallelism employed by Nodejs.

Another important potential improvement that could have been applied to our software would have been to adapt our code for use on a cluster. In fact, we chose these technologies with this extension in mind. MongoDB has the ability to scale horizontally via sharding, where the database distributes itself across several machines, splitting data based on primary keys. Nodejs is simply a process, which could easily run on several different machines simultaneously, and use a form of message passing between the nodes to synchronize them, all saving data to a common database machine.

5. Appendix

Code

The code for our project can easily be obtained using git, or by downloading a prepared archive.

Git

To clone the git tree for our project, create a directory to clone into, and change to that directory. Then, ensure you have git installed on your system, and run:

```
git clone git@iain.hopto.org:ecse420.git
```

Archive

To download a tarball of our source code, visit:

```
http://iain.hopto.org/ecse420/code.html
```

And select your preferred archive format.

Note that several dependencies are required to run the code. The installation procedure is described in the README.txt file and the components necessary are mentioned in the requirements.txt. Following the README.txt file should be enough.

References

[1] R. Dahl. (2009, Nov.). node.js. Presented at JSCONF 2009. [Online]. Available: http://s3.amazonaws.com/four.livejournal/20091117/jsconf.pdf

[2] M. De Kunder. (2011). *The size of the World Wide Web (The Internet)* [Online]. Available: http://www.worldwidewebsize.com/

[3] Joyent Inc. (2011). *node.js* [Online]. Available: http://nodejs.org/#about

[4] 10Gen, Inc. (2011) *Philosophy* [Online]. Available: http://www.mongodb.org/display/DOCS/Philosophy