Building and Measuring the Efficiency of a Distributed Parallel Web Crawler

Teodor Muzychuk*, Roman Mutel*, Bohdan Ruban*, Mykhailo Bondarenko*
* *Faculty of Applied Sciences of Ukrainian Catholic University, Lviv, Ukraine*

***Abstract*—World Wide Web is one of the biggest sources of data in the modern world. Due to its size, in the last 30 years, there exists a challenge to effectively traverse and index the web pages it contains. Google Search is one of the well-known leaders in the industry because of its complex experience in the topic and huge computational power. However, less advanced systems are also worth the attention of developers and researchers. In this paper, we present our solution to the task of web crawling, describe the existing tradeoffs and problems we encountered, and analyze the efficiency of the parallel distributed system we developed.**

I. Introduction

A web crawler, also called a web spider or a spiderbot, is a computer program that visits websites, parses their HTML code to extract features such as page title, headers, links, and plain text, and continues crawling based on the links found on previous pages. The initial set of pages to start the crawling with is called a seed. The use case of a web crawler limits to the imagination of the developers: they are used to gather big amounts of data from domain-specific sites, generate recommendations in search engines, check for updates, test web service integrity, etc.

Crawling the web is a hard and composite task because of the inconsistency of the pages, tradeoffs between quality, quantity, and freshness of crawled pages, repetitive crawling (i.e., visiting the same page more than once), and crawler scalability. It is also worth noting that it is an embarrassingly parallel task. These two factors make the development of web crawlers perfect for educational, research, and interviewing purposes [1], [2].

II. Distributed System Architecture

The so-called microservice architecture (Fig. 1) is composed of nodes serving various functions. The entire architecture, except for the front-end, has been implemented in C++. All communication between individual nodes is performed using HTTP requests, facilitated by the cpr library [3]. The code for all parts, as well as instructions for compiling and running each of them, can be found in our GitHub repository [4]. Specifically, regarding the nodes related to crawling, we define three node types:

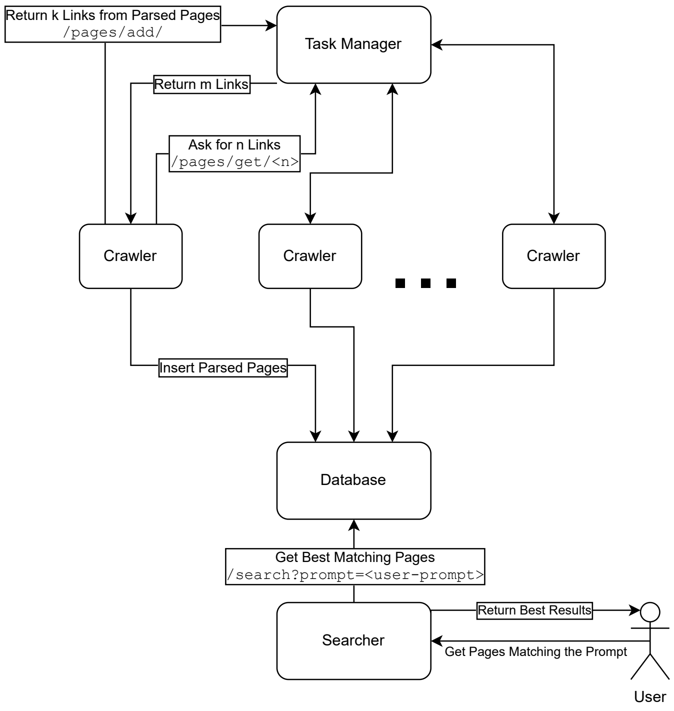

Fig. 1. Microservice architecture

- **Main Node (Task Manager):** This node is responsible for managing the queue of links to be crawled and distributing those among other nodes. Typically, we would only have one Task Manager; however, in certain environments, it might be beneficial to use an architecture comprised of many clusters of nodes, each containing its own Task Manager. In our case, we implemented the Task Manager with a RESTful API using the Crow library [5], which allows us to easily create a multithreaded asynchronous server.
- **Crawler/Parser:** This is the elementary building block of our system. Its purpose is to query the Task Manager for a list of links to be parsed and then parse them. Parsing involves fetching the HTML code of the web page and retrieving relevant information from it. We extract headings, titles, and links. Next, we save this data into our database and send the links back to the Task Manager. Fetching is an embarrassingly parallel task. Thus, we use the parallel pipeline tool from the Intel oneAPI Threading Building Blocks library [6], which allows us to quickly and efficiently parallelize the crawler by creating a pipeline that receives links, produces parsed web pages, and sends them to the database.
- **Database:** The database is the main data pool where all the indexed pages are stored. It is shared among all the other nodes but can be sharded (distributed) for load

balancing. We use MongoDB, which has full-text search capabilities out of the box, allowing for an easy searcher implementation. In the database, we only store the links, title, and headings of each web page to conserve storage. These are sufficient for a rudimentary search engine since the rest of the text usually repeats keywords mentioned in the headings or title.

There are also extra nodes unrelated to the crawling architecture, mostly used for user interaction:

- **Searcher:** This node is responsible for implementing a search API in the database. It accepts a request with a query and returns a list of search results. These results can then be displayed to the user or processed in some way by any front-end. We implemented this node using the Crow library.
- **Front-end:** This node provides the part visible to the user, including a simple UI with a search field and a list of results. It queries the searcher back-end and displays the results to the user.

Classically, the CAP theorem [7] is mentioned when designing and evaluating distributed systems. It states that any system cannot be consistent, available, and partition-tolerant simultaneously; only two of these guarantees can hold. In our design, certain hazards must be considered.

The first is consistency. The data between nodes must be consistent across the whole system. We achieve this by isolating most of the nodes from the Task Manager. The Task Manager operates independently of the status of the database, the number, and status of the crawlers, or the searcher. It is the crawlers' responsibility to request tasks from the Manager and complete them independently. Thus, the failure of the network between the crawlers and the Task Manager does not stop the entire system, providing resistance to partitioning. However, it will affect the data in transfer at the moment, as it is not duplicated in any buffers. In the case of millions of links, a few lost links are of small significance and can be traded for faster communication time between the services.

The real threat to the system's performance is the absence or sudden failure of the Task Manager. In such cases, nothing can continue the work. Thus, our Task Manager implementation can save its state upon crashing or at least return to a previous state.

III. Task Manager Heuristics

How exactly to distribute links among the crawlers expecting them is a difficult design question. It is logical that one should have a queue of links to be processed, which is mentioned by the majority of articles on web crawlers [2], [8]. However, the approach of a single global queue has a few flaws: namely, in practice, the queue tends to get filled up with links from the same domain since domains mostly reference themselves. In the long run, this means that we can exponentially dedicate more and more time to crawling the same exact domain, which is bad both for the heterogeneity of information and the performance of the whole system in case we get rate-limited. An exact description or solution of how this should be handled is out of the scope of the traditional papers on crawlers. That leaves the experimentation to ourselves. We settled on a single queue dedicated to each domain and a "master queue" of domain names. In there, we can prioritize them such that the crawling process is distributed evenly among the number of domains that are currently known. This accomplishes a few things: firstly, it somewhat ensures that we don't violate each domain's access policy: most domains have a `robots.txt` file that defines how automated web indexers should access the site. In case of violation, a rate limiter kicks in, preventing us from crawling the site. To avoid this, we ensure not to return more than 32 links from each domain to each crawler and try to provide each of them with different domains altogether. The prioritization of domains depends on two factors: the "priority" - more important, determined algorithmically by us, and the time of its entry into the queue - older domains that have been "waiting" for longer should be processed first. The "priority" of domains gets sorted out as follows. Upon entry into the queue, each of them gets priority 0. When a set of links from a particular domain is sent to a crawler, the priority of that domain rises - it is moved further down the queue. If its supply of links has been exhausted, it gets moved down even further in order not to impede other domains. This allows to distribute the crawling across, whilst accounting for the possibility of blockage and/or starvation.

IV. Efficiency Measurement

The measurement of efficiency was posed as one of the main tasks in this project. However, it is a non-trivial problem. The system is heterogeneous, and its components know little about each other. It is not difficult to measure a single web crawler's performance, however, it gives us very little data within the context of the whole system; depending on exactly what sort of work was given to the crawler, we may experience a very large deviation in its performance - sites can vary in size, in the time needed to access them, and in the number of pages per domain. Therefore, we need to gauge the performance at a global level.

Another possible solution is doing so at the Task Manager node: we can look at the size of its queue, or the set of all visited links, and see how it grows over time. This is okay, but also has its own caveats: the queue is hardly a good measurement since it is guaranteed to grow exponentially almost no matter what. The "visited" set is okay; however, the presence of a link in it does not guarantee that it was actually parsed. Due to the specifics of our architecture, in order to achieve isolation between crawlers and the task manager, all the invalid links or those that could not be parsed still end up in the "visited" set.

Therefore, the only solution to our problem is querying the database for the number of links periodically and seeing how many pages have been added to it. The optimal sampling rate for this is determined experimentally. We had three runs on the fixed seed of 600 links, dumped by the task manager. First run was with the sampling period of 2 seconds (3), second run had the sampling period of 5 seconds with the hope that it will smoothen the data (4. The third one had a period of 5 seconds, but it was run on the shuffled seed data, instead of ordered. The third one has the smallest standard deviation and is depicted below (5).

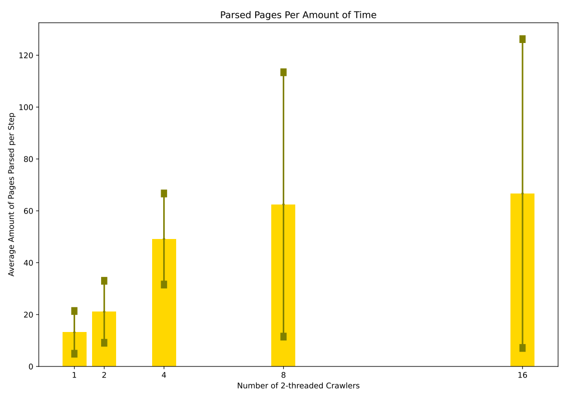

Fig. 2. Measured throughput of various amounts of two-threaded crawlers, 5-second sampling window, shuffled seed file

Even though the overall trend is seen on all three plots, the main problem of the high standard deviation remains. The results vary greatly based on the input data and network performance. Experiments have been held to determine the optimal starting conditions and sampling rate for our setup.

The results are as follows: measuring (5) the number of links parsed by a crawler per measure of time running on the same machine as the Task Manager, we get that 32 threads have the highest average throughput, with the standard deviation growing very quickly with the number of crawlers. This is to be expected, as with the growing number of parsed links, the number of those that cause unexpected delays grows as well. We can also see that the parallelization efficiency coefficient, defined as $E(s) = \frac{T(0)}{s*T(s)}$ (where $T(s)$ is the pages throughput of a system with s crawlers) (6) declines rapidly as the number of crawlers grows.

V. Conclusion

Web crawling can seem trivial at first glance. Nothing seems to be simpler than just ”clicking” on links recursively and logging the results. In practice, though, doing so efficiently brings a lot of fundamental system design questions and leaves one with many caveats and considerations. Even gauging a system’s performance is a non-trivial task and requires a lot of experimentation to get right - and even after that, the results are highly volatile (7). Even parsing a single site is a problem that is being tackled over and over by developers around the world, as the web gets more and more complicated. To conclude, a lot of credit has to be given to the designers of world-renowned search engines such as Google. Their breakthrough nature becomes apparent as soon as one tries to tackle a similar problem at the same scale.

References

[1] Educative, Inc., “Grokking modern system design interview for engineers & managers,” last accessed 1 June 2023. [Online]. Available: https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/7XxnzJxOX0r

[2] SystemDesign, “System design interview: Web crawler,” 2021. [Online]. Available: https://medium.com/double-pointer/top-5-videos-for-web-crawler-system-design-interview-75b7ac9c04ce

[3] Open Source Community, “libcpr: C++ requests,” 2020. [Online]. Available: https://github.com/libcpr/cpr

[4] R. M. T. Muzychuk, “Nyshporka web crawler,” 2023. [Online]. Available: https://github.com/rwmutel/nyshporka

[5] Open Source Community, 2021. [Online]. Available: https://crowcpp.org/master/

[6] Intel, “Intel oneapi threading building blocks: Scalable parallel programming at your fingertips.” [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html

[7] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT NewsVolume 33Issue 2*, 2002.

[8] N. El-Ramly, H. Harb, M. Amin, and A. Tolba, “More effective, efficient, and scalable web crawler system architecture,” *International Conference on Electrical, Electronic and Computer Engineering, 2004. ICEEC ’04*, 2004.

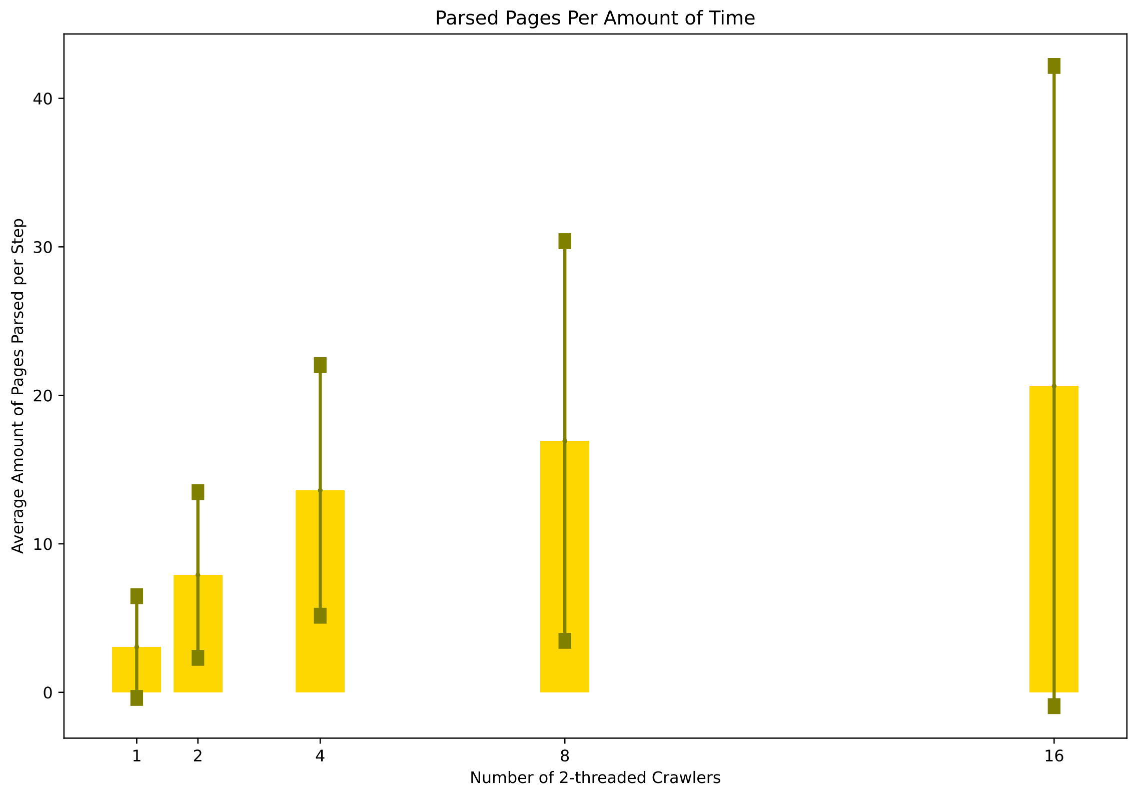

Fig. 3. Measured throughput of various amounts of two-threaded crawlers, 2-second sampling window, ”raw” seed file

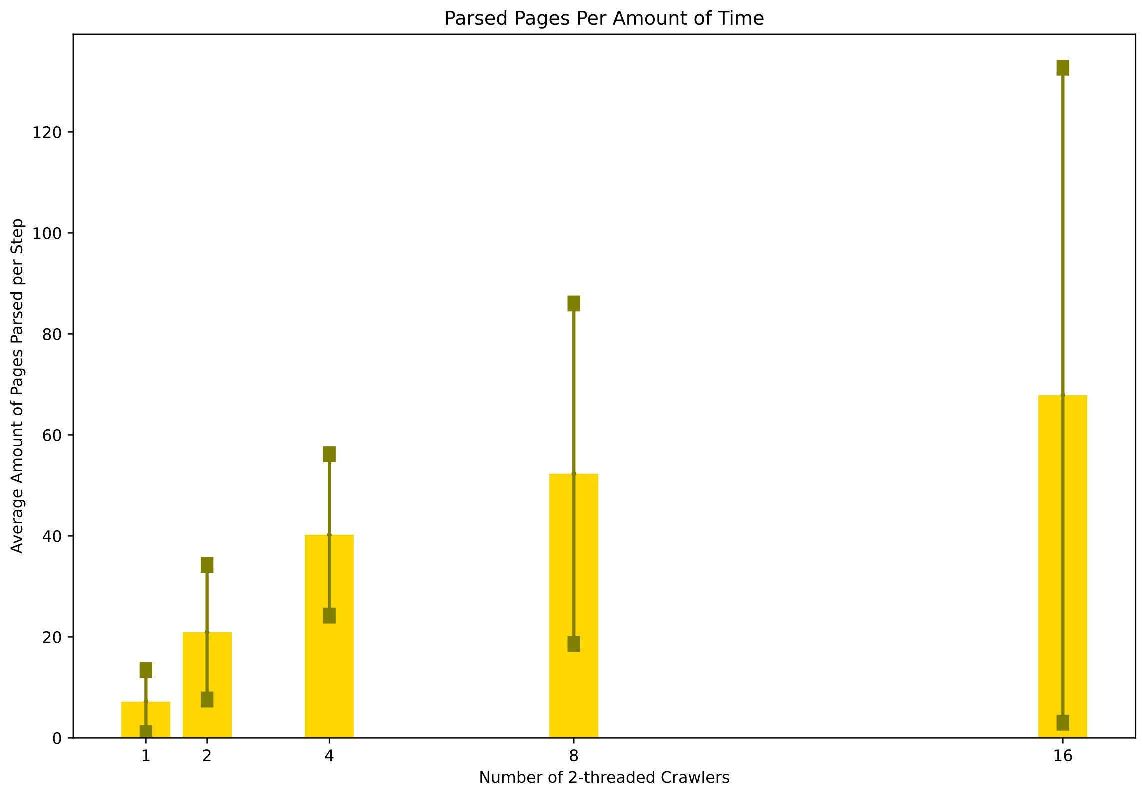

Fig. 4. Measured throughput of various amounts of two-threaded crawlers, 5-second sampling window, ”raw” seed file

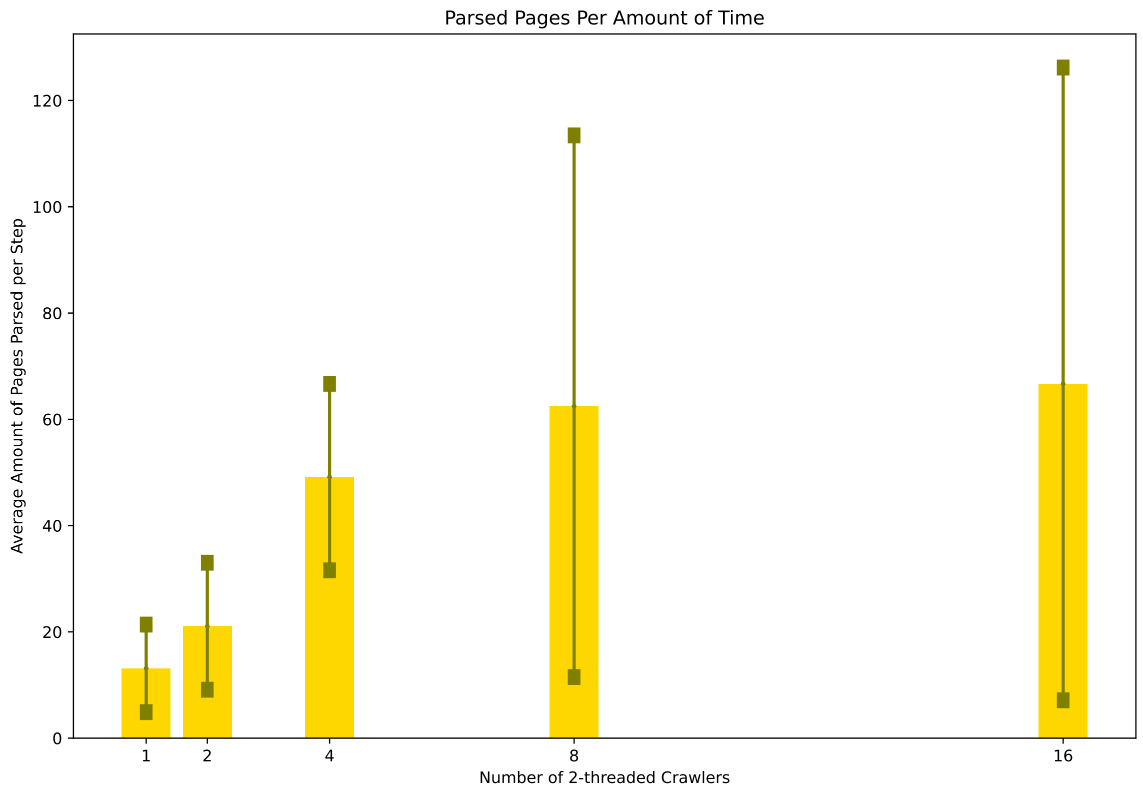

Fig. 5. Measured throughput of various amounts of two-threaded crawlers, 5-second sampling window, shuffled seed file

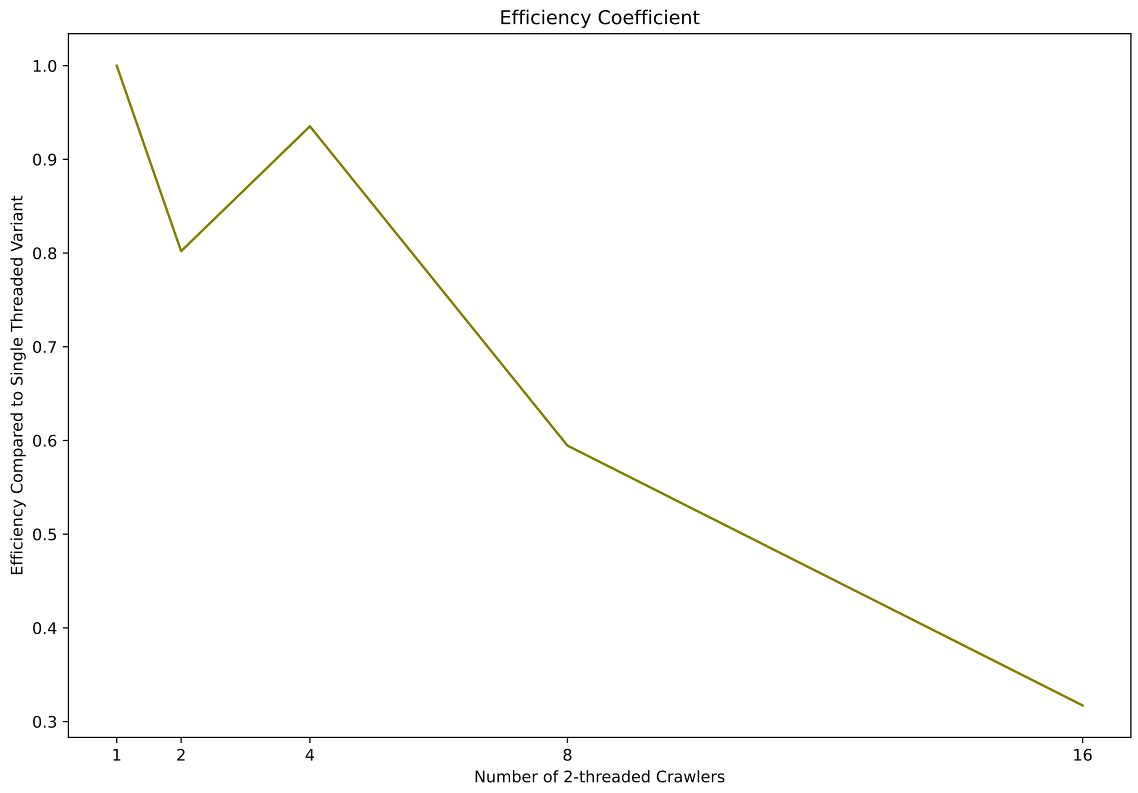

Fig. 6. Parallelization efficiency coefficient relative to the number of crawlers, 5-second sampling window, shuffled seed file

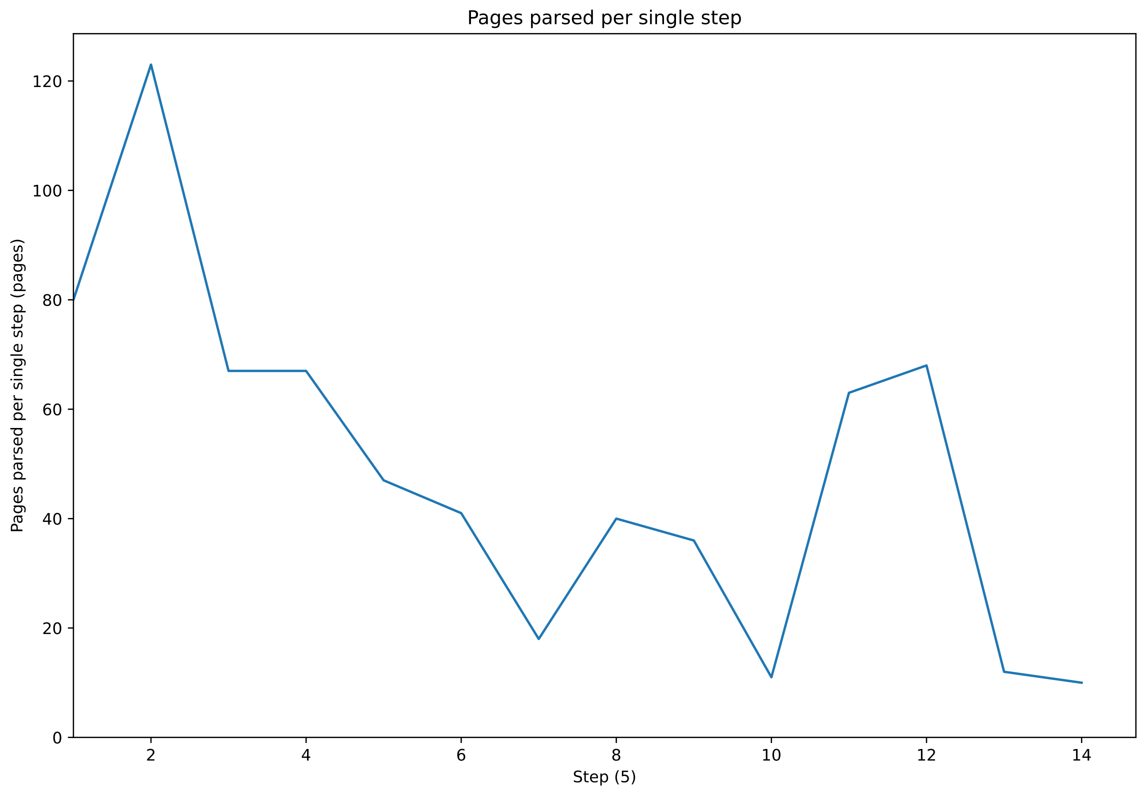

Fig. 7. Pages parsed per single sampling period, 8 two-threaded crawlers, 5-second sampling window, shuffled seed file