Real Life Application of Producer Consumer Problem

1st Ricardo Mangandi
Department of Computer Science
University of Central Florida
Orlando, USA
ricardo.mangandi@gmail.com

2nd Kyle Karacadag
Department of Computer Science
University of Central Florida
Orlando, USA
kylekaracadag1@gmail.com

3rd Nick Thiemann
Department of Computer Science
University of Central Florida
Orlando, USA
nathiemann1@gmail.com

4th Samuel Hearn
Department of Computer Science
University of Central Florida
Orlando, USA
shearn@knights.ucf.edu

5th Courtney Than
Department of Computer Science
University of Central Florida
Orlando, USA
courtthan@gmail.com

I. Abstract

The blocking queue data structure for multithreaded computing has many uses in real-world applications. Different applications of this data structure were explored and its benefits compared to equivalent single-threaded solutions to the same problems were evaluated. We gathered a list of real-world computational problems, then selected three to build a blocking queue solution from. The performance of each application was evaluated with only one worker thread, then with multiple worker threads. Experimental data was used to come to a conclusion on the efficiency of this data structure for these real-world problems. In the end, we found a minor but consistent speed-up in performance.

II. Introduction

In the real-world there are some systems and programs that compute heavy and long running tasks that consume lots of memory while processing. Sometimes these tasks are broken down to smaller tasks. It is discouraged at times to run multiple processes that consume lots of memory at one time because it could lead to the machine to crash. So, what exactly is available to automate how many processes we want the machine to process at one time? The answer is to use a blocking queue that would automate the first in first out concept for a real-world application. The blocking queue would only allow dequeues based on its available resources it has. Our research attempts to illustrate this by using a blocking queue that will block items from being dequeued based on how many resources it has available at a time. The queue is unbounded, so it can enqueue as many items at one time. The tool we used to simulate this environment is Redis. Redis is an in-memory data structure store it allows us to create multiple queues locally on our machine. Along with multiple queues and tasks inside the queues we can create N processes that consume the tasks. All of this is created locally on our machine. Our application was written in Python to support the Redis Q library. The tasks that are enqueued to their respective queue are video editing tasks, web scrapping tasks, and a small OpenCV app. These tasks were picked due to how much memory they take for a certain period. In short, our project is the producer and consumer problem where we have a system that produces content and threads/processes that consume the items in the queue. Dependent on how many threads are working.

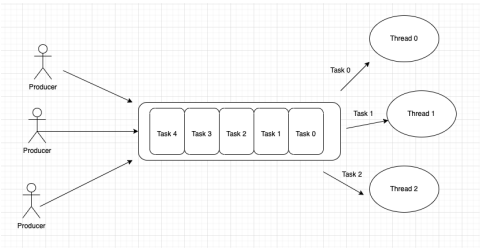


Fig. 1. Blocking Queue Diagram

III. Redis Breakdown

This section takes a look at the tool being implemented in this project. Redis is an in-memory cache and message broker. Its main appeal is its very low latency which is optimized by it not trying to send messages more than once, leading to no message guarantee, and the fact that in-cache memory allows for faster retrieval and messaging [4]. It is also worth mentioning that there is no native message persistence, meaning that any messages in the midst of being sent upon a crash are lost. It needs to be allowed to use the disk of whatever device it is stored in to obtain persistence.

Redis itself has a publish/subscribe system where there a masters and slaves, the slaves are all subscribed to different channels the master publishes to that they want to receive updates from. The slaves can unsubscribe from the channels at anytime given a message. Control signals are put into the

messages, with the first element of a message referring to one of the commands. This system allows for the network to be more dynamic [3].

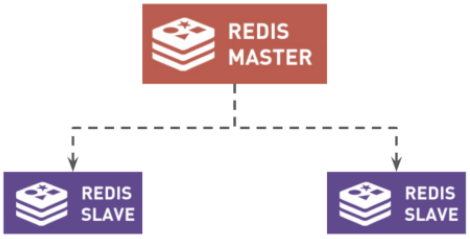


Fig. 2. Redis Architecture. Adapted from [4]

Compared to other message brokers Redis is known for having the lowest latency, as such was used for this project. This is particularly good for web scraping sites that see a lot of input, such as social media. When thousands of messages are uploaded at once, a messaging broker with low latency like Redis is valued as databases cannot handle processing the load by themselves [5]. Likewise Redis can be used by the social media platforms themselves, for loading the messages for the user quickly, or choosing not to load some messages to prioritize the videos. It's also ideal for systems that have hard architectural limits, that need small memory requirements because it's all an in cache memory. This ability of Redis allows it to run on computers such as a Raspberry Pi without much sacrifice [3].

IV. Discussion of Related Work

This paper talks about a model that is based on the producer/consumer paradigm that construct real-time applications that interact with external environment which is one of the most important features of a real-time system. The system is not an independent system, it responses to any external inputs that comes from devices such as sensors,.. then it process the data and sent the responses in order to control the overall system.

The producer-consumer paradigm is a standard paradigm for multitasking. The connection between the producer and consumer is synchronized using a monitor that wait for consumer to consume data and wait for the producer to produce data. This paradigm is considered as a model of interaction between internal and external processes where the producer acts as the external process since it collects data from external devices and the consumer as the internal process which create computations based on the data provided by the producer (external process).

However, to ask if the producer-consumer paradigm could be viable for the real-time system or not. No, it could not be and there are two main reasons for this. First of all would be the external process problem; if the producer is an external process which means that it is not under control of the computer and therefore, it would be impossible to control this external process to wait for the consumer to consume the data. For that reason, it's important to make sure that the consumer will consume all the data provided without the wait statement in the Insert function. Secondly, using wait statements in real-time systems is not acceptable.

As a result, the real-time systems producer-consumer model means that all consumer tasks must process data in a time frame imposed by its producer. In other words, in order to apply the producer-consumer paradigm in the real-time system, we need to consider all processes from our system to be periodic and the following characteristics for each computing:

- task identifier IDi that defines a task uniquely [2].
- task i execution or computing time Ci– the execution time of a task can vary within an interval [CBi, Cwi] where CBi and Cwi are the best respectively the worst case execution times for the task i (in most of the cases Worst Case Execution Time is denoted by WCETi); generally, only the best and the worst execution time of each task are known. When modelling the timing behaviour of tasks, this is a natural approach, because exact computation time of a task cannot be establish. Consequently, both times have to be considered when creating a task behaviour model, and results must be compared; another approach could analyze the timing behaviour starting from the idea of variable execution times. In our model we will rely on WCET [2].
- task deadline Di – deadline is a typical task constraint in real-time systems: is the time before which the task must complete its execution. Usually, the deadline of the task is relative, meaning that, from the moment when a task Ti arrives (from its arrival time), it should finish within Di time units [2].
- task period Tiper – for periodic tasks, task period Tiper is considered to be equal to task deadline Di. Because in most real-time control and monitoring systems the tasks typically arise from sensor data or control loops are periodic, we will consider in our model that all tasks are periodic [2].
- task ready (arrival) time Ri – represents the moment of time when the task Ti is ready for execution [2].

One of the algorithms that is an optimal algorithm for periodic tasks is the Earliest Deadline First algorithm. Earliest Deadline First is a priority driven algorithm where higher priorities are assigned to the requests that have the earliest deadlines.

V. Contributions of Our Paper to the Problem

The producer-consumer problem is a problem that can be applied to real life applications. Our application has multiple queues that consists of items being enqueued and items that are being dequeued. When there are no consuming threads then the items become idle inside the queue. A system at times has a limited number of resources available to it or we want to make the most out of a system's resources. Our application attempts to show this by using all the available resources it has and block items from continuously being consumed by

threads if the working threads are not available. The real-life tool that helped us accomplish this was Redis, Redis has a Python library called RQ which consists of multiple tools to enqueue items inside a queue, task schedulers, thread killers etc. The automation of our application is completed by the first in first out concept, so we do not have to redo the enqueue process. Whenever the items become available again the items are consumed. Our paper shows only a select few of experiments which are the CSV reading, video editing, and the web scrapping application. However, these small experiments are proof on how the producer-consumer problem can be applied to small real-world problems to speed up performance and use all the available resources. The results of these experiments may be somewhat misleading regarding performance due to how large the n inputs were. If the n inputs were much larger, we could probably see a much dramatic increase in performance.

Some programs that can benefit from using Redis queue are long running tasks that run in the background. An example of this is machine learning algorithms. In a project that I have been a part of consisted of being able to manage a backend that would have machine learning tasks running in the background and the producer-consumer problem was perfect for this problem. In addition, in the development community there is a large support for the producer-consumer problem to solve tasks such as machine learning tasks running in the background. The following article by Rosebrock showcases a tutorial regarding how to scale the Keras library and deep learning using Redis as a tool. [1] The article showcases the usage of Redis as a broker/message queue for the usage of a deep learning REST AP. The figure below showcases how similar the image is to the producer-consumer problem. The image below has clients (producers) producing content into the Redis queue and then we have the server (consumer) consuming the items.

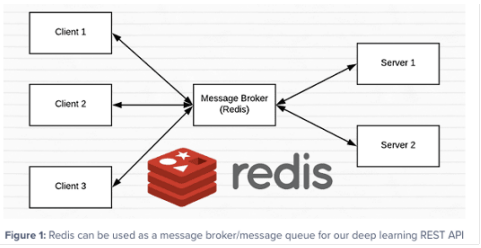


Fig. 3. Redis Server Architecture. Adapted from [1]

VI. Our Application

The application consists of two main python files that are the catalyst to each main feature: producer-consumer. There is a producer_main.py file which initially checks how many Redis workers there are, this essentially checks if there are any consuming threads available to do the work that our producers are producing. If there are no consumers, then the program lets the user know that there are no workers available and kills the program. If there are workers available, the program will ask the user which queue they would like to use. Each queue has an associated number of workers listening to it. If there are n workers and these workers listen on queue a, but the user selects to use queue b then no consuming threads will accomplish work. This is due to no workers listening on the queue b.

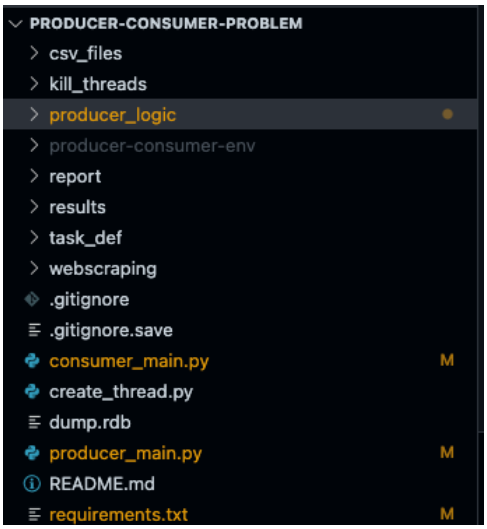


Fig. 4. File directory

The first step would be to run the consumer_main.py because this will initialize all the working threads that we need and have them listen to a specific queue based on what task the user would like to accomplish. Once that is completed, we can run the producer_main.py file where the user picks the queue that has active workers listening to it. Each experiment we ran has a queue that is assigned to it.

The application supports three different queues. Queue one contains the ability to parse a csv file and look for a specific value. Queue two contains the ability to split videos into n number of active workers with FFMPEG. Queue three contains the ability to web scrape three different sites at the same time. Then we have a default queue called queue zero which puts the process to sleep for 10 seconds. The queue zero is placed as a place holder to make sure the program can capture undefined behavior.

Redis comes with a monitoring system that allows us to see the workers who are currently working and see how each respective queue can block. This feature is handy and allows us to see the usage of how blocking queues are used in the real-world. The Redis dashboard can be loaded up on localhost:9181. We can see in figure 4 how the queue that is called queue_one has items inside of it. However, there are no workers listening therefore the queues will keep the items from being popped.

Figure 5 and 6 show how the dashboard presents the workers registered and their respective queues. When there are no workers present it will look like figure 5. However upon adding workers it would look like figure 6 showing the active workers and what queues the workers are working on.

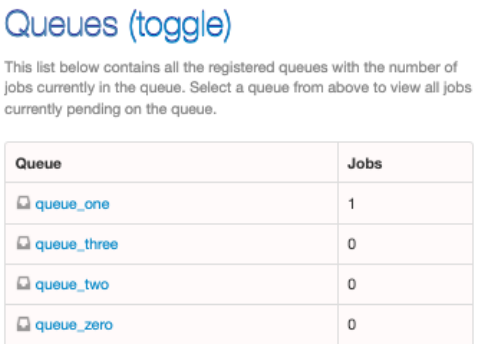


Fig. 5. Queue Presentation

Fig. 6. Workers before

VII. Our Experiments

As mentioned in the ”Our Application” section, we used the Redis library to parallelize python algorithms and see if they run more efficiently than the non-parallel version. To do this we created three algorithms using python: a web scraping program, reading from csv files, and video editing. These three programs were first created without parallel computing using python and the execution time was recorded given different inputs with different sizes. The programs were then split into multiple sections so the threads in the Redis program can take on each section and assign it to a different thread inside a blocking queue. Each was implemented as a specific module within the same project. Each application had its own designated queue in the redis memory. The execution times for each program were recorded again. The execution times of the three programs can be seen in the ”Results” section of this paper. Below are a more detailed description of what each one of our three programs did:

A. CSV Parsing

The first and simplest application was the creation of an algorithm that could parse through multiple CSV files to search for a keyword between multiple workers simultaneously. A CSV file is a data format which has values separated by commas. This is a simple but relevant application in real-world computing. If a business has a large set of data files and needs to search for specific data, a multithreaded approach using a queue data structure may be beneficial.

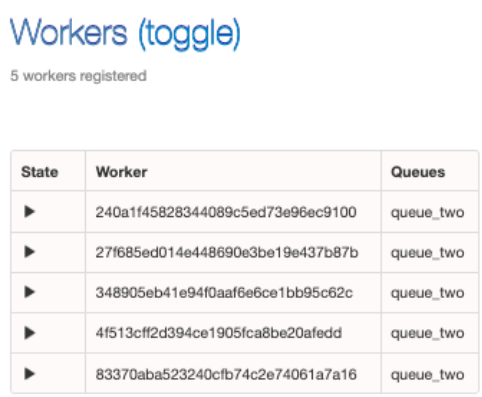


Fig. 7. Workers after

The file path of each CSV file was put into the Redis queue as a job and then picked up by workers which looped through the lines to count the total occurrences of this keyword. Each worker returned the number of occurrences of the keyword in the file it read. To test, there were 12 CSV files of stock data and the keyword to search for was “2014”. 8 different trials were used for 1 worker vs 3 workers. After running and getting the averages, the CSV parse saw a significant improvement in run-time, from an average of 0.03789 seconds when testing with 1 worker compared to an average of 0.0195 seconds with 3 workers. This is an average of 48.53% time improvement. Possible improvements in finding its more accurate time savings could be made by creating more files and/or longer files to parse, or testing with different numbers of workers to study Amdahl’s law.

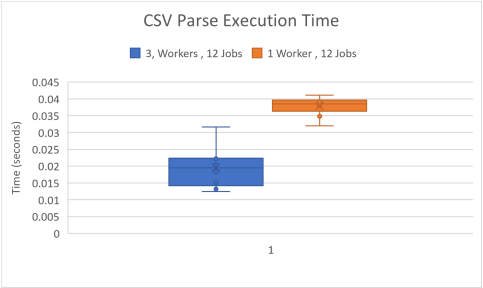
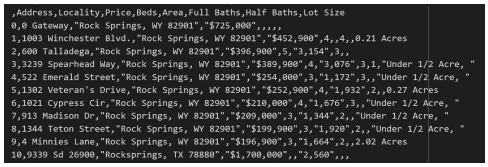


Fig. 8. CSV Parsing Results

B. Web Scraping

The next application was the use of the Redis queue for web scraping. Web scraping entails extracting useful, relevant information from a website or multiple websites. It is an especially powerful and relevant real-world task and thus we decided to choose it to demonstrate how to improve it using multithreading.

We created this web scraping program using Python and the libraries BeautifulSoup, requests, pandas, and time. BeautifulSoup and requests were used for the web scraping aspect of our program. Pandas was used to store the web scraped



```
,Address,Locality,Price,Beds,Area,Full Baths,Half Baths,Lot Size
0,0 Gateway,"Rock Springs, WY 82901","$725,000",,,,,
1,1003 Winchester Blvd.,"Rock Springs, WY 82901","$452,900",4,,4,,0.21 Acres
2,600 Talladega,"Rock Springs, WY 82901","$396,900",5,"3,154",3,,,
3,3239 Spearhead Way,"Rock Springs, WY 82901","$389,900",4,"3,076",3,1,"Under 1/2 Acre, "
4,522 Emerald Street,"Rock Springs, WY 82901","$254,000",3,"1,172",3,,"Under 1/2 Acre, "
5,1302 Veteran's Drive,"Rock Springs, WY 82901","$252,900",4,"1,932",2,,0.27 Acres
6,1021 Cypress Cir,"Rock Springs, WY 82901","$210,000",4,"1,676",3,,"Under 1/2 Acre, "
7,913 Madison Dr,"Rock Springs, WY 82901","$209,000",3,"1,344",2,,"Under 1/2 Acre, "
8,1344 Teton Street,"Rock Springs, WY 82901","$199,900",3,"1,920",2,,"Under 1/2 Acre, "
9,4 Minnies Lane,"Rock Springs, WY 82901","$196,900",3,"1,664",2,,2.02 Acres
10,9339 Sd 26900,"Rocksprings, TX 78880","$1,700,000",,"2,560",,,
```

Fig. 9. CSV Files obtained from web scraping

data into a dataframe and then convert that dataframe into a csv file. The time library was used to get the execution time of the whole program. The website that was used to web scrape was a real estate website. There are three pages in the website and the data extracted were Address, Locality, Price,Beds, Area, Full Baths, Half Baths, Lot Size. We have try and except statements for each of the data extracted in case the program found any null values. This ensures that the program won't crash in runtime. We gathered a total of 36 rows of data.

To parallelize our algorithm, we split it into three "tasks" so the threads in the Redis program can add it to the blocking queue. Each task took on one page from the website and web scraped it individually. Each task creates its own dataframe from the data gathered from each page and these data frames are sent to the main.py class so they can be combined together and outputted to one CSV file. Before we used Redis the execution time we got was 1.3472 seconds after running the program 10 times and getting the average. After using Redis the execution time decreased to 0.9863 seconds on average. We can clearly see that Redis was able to execute the program more efficiently.

- HomeInfo.cvs: CSV file that holds all the data extracted from the web scraping program
- main.py: The main class that runs all three producer classes to combine them into one CSV folder. This program also gathers the execution time data.
- producer_one.py: The first class that the thread in the Redis queue takes to webscrape the real estate website. This class extracts data from the first page in the website.
- producer_two.py: The second class that the thread in the Redis queue takes to webscrape the real estate website. This class extracts data from the second page in the website.
- producer_three.py: The third and last class that the thread in the Redis queue takes to webscrape the real estate website. This class extracts data from the third page in the website.
- realestate_scraper.py: The original web scraping program that does not use any parallel computation. This program gets the execution time for using our webscraping program without Redis.

C. Video Editing

Video editing was a task assigned to a queue due to how intensive the process can be and wanted to see how we can parallelize this and have multiple workers accomplish this task

Fig. 10. Web Scraping file directory

at the same time. The idea was to split a video into n number of workers. Each worker would get a certain parameter to start editing from and to end editing at. The library used was FFMPEG which is a video editing command line tool that is very popular due to it high performance and flexibility. To see any significant difference in performance between the tasks being parallelized and a video being edited on a single thread the task did not need to be run on a large video. The video that was used to run both sequential and parallel was a 40 second clip. The results were consistently the same. A video of 15 minutes was also tested to see how the run time would change and the performance difference between them was dramatic.

The editing process did not need to be compared to such large videos to view the difference between the parallelized method and the sequential method. The video would get split into the n number of workers and we could see the performance difference. For a 40 second video that would get split into 5 8 second videos had a performance of 10-13 seconds compared to the editing of FFMPEG on one thread this process would take 27 seconds. This is roughly a 27% - 30% performance increase. I ran this process a total of 10 times with consistent results every time on each run. The way that the editing works is that FFMPEG goes frame by frame and concatenates each frame as it goes on, this process is linear. With the parallel execution we have 5 workers looking for each frame and concatenating its own video. The process to look for frames is still linear in run time; however, we have more processes at one time accomplishing work which roughly increases the performance by roughly 27%.

Another interesting task to try would be to see how fast the items would be able to concatenate a video together with n frames given. The frames would be loaded into memory so to access each frame instead of a linear operation to find them would be O (1). Even though this was not tested logically the parallelized version should do a lot better due to it not having to search for the frames thus increasing the runtime performance.

VIII. Experiment Results

After the results of the three applications were recorded, we came to the conclusion that the Redis queue with multiple workers did speed up the execution times of the applications overall by a slight margin compared to one worker. Overall we found a 20% - 30% speedup for execution times when providing one worker per job of the task compared to one worker overall. It shows that the Redis queue can be used for more efficient computing on real-world computing problems, but our data may have been too limited to analyze the effects of very large inputs. We provided what we deemed as large inputs to analyze execution times, but computation was still somewhat trivial for single-threaded solutions for the input we provided.

It was also observed that the amount of threads executing tasks did not have a linear correlation with the speedup of the execution. This could be due to Amdahl's Law, which explains that the overhead to maintain multiple threads can have a negative impact on performance. That is, while overall execution time was better, the overhead came with a cost.

A. Possible Improvements

In retrospection, improvements to the data collection could have been made in the following ways:

- Providing a larger data set for each problem to gain more accurate data on time savings
- Providing a larger number of workers to show the adverse effects of Amdahl's law on efficiency
- Building solutions for more real-world problems
- Testing other frameworks besides Redis to compare different runtimes of different blocking queues

IX. Appendix

A. Challenges

Challenges faced include being able to coordinate with other thread workers to reduce redundant work. This would include coming up with the logic to split the workload between each thread. Since our project allows the user to decide how many working threads they would like, this brings a challenge to coordinate based on N number of threads. The problem would be much easier if we had a set number of threads and knew what tasks would be run on those threads.

The redis data structure queue has a timer in seconds, when tasks get inserted they get a set timer imposed on them, the default is currently two minutes. If the timer expires before the tasks are able to get enqueued they all get dequeued at the same time. Coordinating the logic with the timer is also a challenge.

Simulating a blocking-queue involves the usage and understanding the thread lifecycle. The thread lifecycle in Redis workers is different compared to real threads. When a task gets accomplished a normal thread would go to terminate itself. When a task gets done with Redis workers they stay up. Coordinating a creative way to kill them once they are done is a challenge.

B. Continuation

As stated previously, the N inputs of the experiments were too small going forward it is ideal to use bigger N inputs. Using bigger N inputs would give us consistent reliable results. Something odd about the Redis program is in programming terms it is difficult to tell when all the workers have finished working and we never had a way to tell this, but to look at the Redis flask dashboard. We attempted to implement a way that would loop while the workers run inside the producer_main.py, but this proved to be inefficient and never exactly accurate. We kept noticing how the script would finish, but when we would check the Redis dashboard there would be some workers that are still working. From the Redis documentation it does not seem they have an out of the box solution for this. Moving forward implementing a clever way to automate this process to get perfect runtimes would be ideal. An idea is having the results be written to a database after they are done, and whenever an event such as write to database would occur the user would get notified by some sort of way.

We did not get to attempt to implement a machine learning task that would take a lot of resources. Implementing some sort of Computer Vision task that would showcase the usage of the system not getting clogged up by multiple requests would be an ideal experiment to show as well. Computer vision tasks are known to take periods of time and the size of N can drastically affect the outcome in the end.

References

[1] A. Rosebrock, "A scalable keras + deep learning rest api," PyImageSearch, 17-Apr-2021. [Online]. Available: https://pyimagesearch.com/2018/01/29/scalable-keras-deep-learning-rest-api/. [Accessed: 28-Apr-2022]

[2] D Zmaranda, G. A. Gabor, A. Nicula, "Producer-Consumer Paradigm in Real-Time Applications," Gianina Adela Gabor, 2009.

[3] Redis. [Online]. Available: https://redis.io/. [Accessed: 28-Apr-2022].

[4] R. G. Hedge and N. G. S, "Low Latency Message Brokers," International Research Journal of Engineering and Technology, vol. 7, no. 5, pp. 2731–2738, May 2020.

[5] R. K. Singh and H. K. Verma, "Redis-based messaging queue and cache-enabled parallel processing social media analytics framework," The Computer Journal, vol. 65, no. 4, pp. 843–857, 2020.