CS5600 Computer System Final Project Report

Parallel Zip
An Application of the Producer-Consumer Concurrent Design Pattern

Aly Badary

0. Abstract

In the second part of this class we got introduced to a new abstraction besides the virtualization of CPUs and memory; that of a thread. We learned about how multi-threaded programs can have multiple points of execution, and how that sounds like a potential boost in performance but in fact can lead to quite the opposite if not handled properly. Nevertheless, such a powerful concept had me intrigued and I decided to dig deeper to learn more about how to properly and effectively employ multi-threading to boost a program's performance. The program in question was one suggested by the author of the textbook we've been following (OSTEP): Remzi H. Arpaci-Dusseau, and it's that of a simple premise: zipping files using RLE (run length encoding). The RLE algorithm, while not very powerful, is simple to understand and implement, which helps focus much of the efforts into the concurrent programming challenge. I start with the simple, linear/sequential program which I then attempt to optimize to a concurrent-yet-inefficient program, and finally rely on a classic concurrent programming design pattern to achieve an efficient parallel version.

I will be using the C programming language for all things code here, yet I believe C++ can be used to achieve similar results in arguably more compact code due to the availability of native data structures. While other languages can be also be used, the third (optimal) version of the program takes advantage of a system call that directly maps files into memory, a procedure that is most straightforward using a low-level language like C/C++.

This project is suggested by the OSTEP course owner/designer, the link for the prompt will be posted in the appendix.

1. Prerequisite: RLE Compression

RLE (run length encoding) is a lossless compression algorithm that is very easy to understand and implement but only tends to offer a decent compression ratio to data with certain patterns/sequences. Those patterns are essentially repeated ones, as the algorithm works by replacing a sequence of repeated characters by a number representing the count or number of instances of that character, followed by a single instance of the character itself. For example, the following string:

"AAAAAAAAAABBBBBBCCCD"

Would be encoded as:

“10A6B3C1D”

It should be easy to notice that such premise would only effectively compress very niche types of data, and for the other vast majority it might actually end up expanding the original data. A string of 4 unrepeated characters only takes up 4 bytes of memory (one for each character), but if we were to prefix every one of those character with a number (a 4-byte integer), we’d end up with a compressed version that’s roughly 4x the original! Perhaps text files with a lot of spaces for indenting or line-art images that contain large black/white areas might be the ones best suited for: RLE.

The compression algorithm itself is not the focus of this project and accordingly the convivence provided by RLE in terms of implementation is all that matters for our purposes. We are not attempting to devise the best compression tool but to learn how to successfully parallelize a program to achieve better performance.

A snapshot of the code used to design this algorithm is shown below:

```c
void compressRLE(FILE* fp){
    if (fp == NULL){
        exit(EXIT_FAILURE);
    }

    int curr;
    int prev = fgetc(fp);
    uint32_t count;

    while (prev != EOF){
        curr = fgetc(fp);
        count = 1;
        // Count matching consecutive characters
        while (curr == prev){
            count++;
            curr = fgetc(fp);
        }

        // Write to stdout, 4-bit integer count followed by 1 bit character
        fwrite(&count, sizeof(count), 1, stdout);
        fwrite(&prev, sizeof(char), 1, stdout);

        prev = curr;    // for next iteration
    }
}
```

Figure 1 - RLE Algorithm in C
(file pointer argument being the file to encode)

Our compression tool will work with file inputs, where the file data will be read into memory, encoded using our algorithm, and finally written (as bytes) into *stdout*. This output can be

redirected into another file instead of printing on the terminal, and an unzipping tool will also be developed to test the correctness of our zipper. If multiple files are passed in as arguments, then they will all be encoded and written out to the same output channel (*stdout*), which means that the unzipping tool will not be able to restore exactly which data belonged to which file, but rather decode the whole stream. Later versions of this program can attempt to achieve the former.

2. Linear (Sequential) to Parallel

We discussed how our single threaded program will work in the last paragraph but let's layout the process again with a simple visual summary in pseudocode:

Linear (Sequential):

```
int main(files){
        for file in files{
            FILE* filePointer = fopen(file)
            compressRLE(filePointer)
        }
}
```

Looking at the setup above and starting to think about how to parallelize this can seem simple at first. The first thing that comes to mind would be to recycle this setup almost without change except for introducing additional threads; one to handle each file. This will require very little effort in terms of rewriting/refactoring code (or none really), and the idea itself remains quite simple and intuitive. So our new visual summary would look like:

Simple Parallel:

```
int main(files){
        pthread_t threads[no. of files]
        for i in range(no. of files){
            FILE* filePointer = fopen(i^th file)
            pthread_create(&threads[i], NULL, compressRLE, filePointer)
        }
}
```

Obviously, there are a few things that need to be altered for this to work correctly (strictly speaking in terms of correctness here and not performance).

The most important one is how the output stream (*stdout*) and the order of writing to it is handled. We know threads run in parallel to one another (unless locks/condition variables are introduced),

so there's no reason to believe that the order in which those files' encoded data are being written to *stdout* will be preserved. This could be handled via a myriad of ways, including (but not limited to); each thread writing the encoded data to a separate structure, be it a temporary file, temporary array or even a temporary custom structure, and in the end the main function can combine write those temporary buffers to *stdout*. For our own simple parallel tool, we will use custom buffer structures (which we will need for our next design iteration of the tool anyway).

Once this issue is handled, we can now say that we have successfully designed a concurrent version of our zipping tool. Can we also say that it is an *efficient* one? Not really.

You see, earlier on in the semester when we were discussing the virtualization of CPUs and how the operating system needs to make sure all the limited (hardware) resources are being fairly shared along all processes, we now run into a similar kind of issue plaguing our simple concurrent tool: fairness.

For simplicity's sake, let's say we have 4 CPUs on our machine and that we ask our simple parallel tool to zip 4 files at once. Our tool will process to create 4 threads, each of which handles a single file and is supposedly running on its own CPU to achieve maximum throughput. Now picture this, out of those 4, files 3 are ~50 MBs while the remaining one is an astounding 300 MBs! This poor thread that unluckily gets assigned this unproportionally large file will almost certainly take much longer than all other threads to conclude, ultimately being the ***straggler*** lingering behind an slowing the whole group's procession to the finish line, not to mention the toll it takes by having to do so much more work compared to the others.

Can we do something to help resolve this **straggler** bottleneck that seems to hinder our ability to achieve an efficient concurrent program design? Turns out yes we can, and we can do so using one of the classic concurrent programming design patterns: **Producer-Consumer**.

3. Efficient Parallel Zip: Producer-Consumer Problem

One of the classic concurrent programming design patterns is producer-consumer. Under that pattern, a process is designated as either a *producer*; a process that is responsible for adding to some fixed-size data structure (usually called a buffer), or a *consumer;* a process that is responsible for removing from that same data structure ("consuming" from it). Figure 2 below shows a simple graph that represents this ordeal.

To achieve consistency with such design, certain conditions need to be met:

- Access to the shared buffer (data structure) must be **mutually exclusive** (i.e. only one process, be it a producer or a consumer, can access it at a time).
- A consumer must not consume from an empty buffer.
- A producer must not add (produce) to a full buffer.

Those conditions should be intuitive when you think about the design we're trying to achieve. They are also rather easily implemented with the aid of locks (mutex) and condition variables.

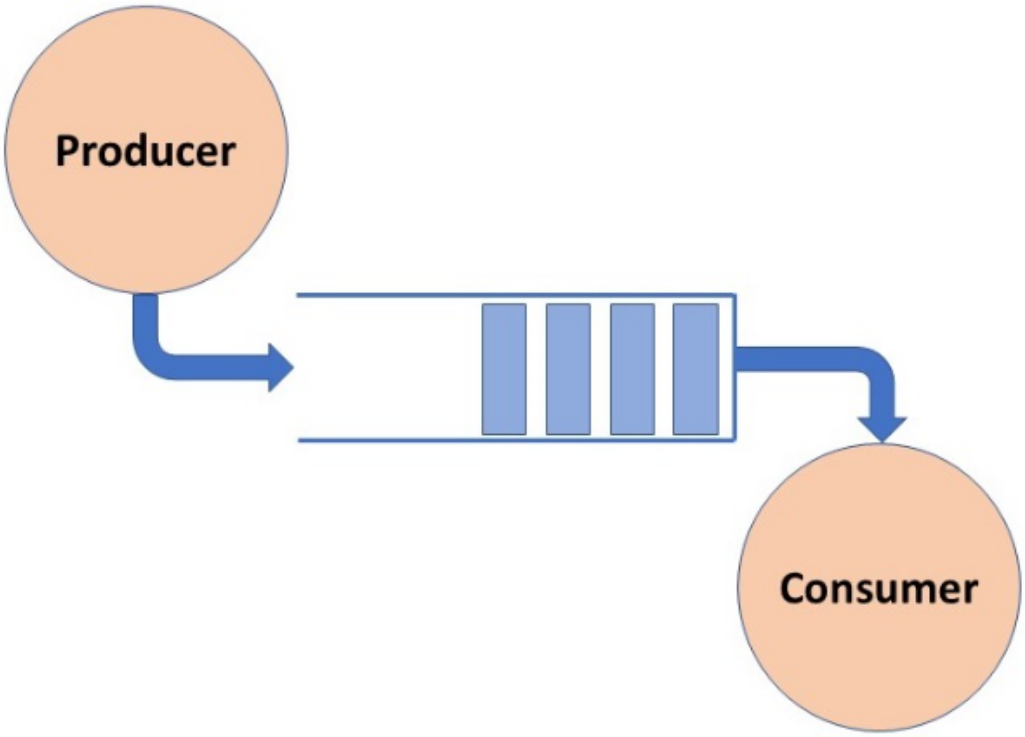


Figure 2[1] - The Producer-Consumer approach visualization

The question remains though: how can this pattern help us resolve the 'straggler' problem in our parallel zip?

The answer is simple; instead of handling files entirely at a time, break them down into *pages* that can then be handled simultaneously and concurrently, by any (consumer) thread. Let's expand a little on this.

Our problem earlier was the bottleneck we run into when a certain thread working on a certain file takes much longer and handles much more work compared to the other threads, who might just be idle in that extra time. In very high-level terms, what we want to ideally happen in this case is for those idle threads to chip in on that straggler thread and help finish up the remaining work. The problem here is; how can they do so without disrupting the consistency of the file being encoded?

By using a single *producer* thread to map those files to memory then break them up to ***pages***:

(fixed-size bytes of information that include metadata about the file itself and the offset location of this page within it), we can then just keep adding those pages into our fixed-buffer queue, which in turn would be handled by the any available *consumer* thread.

Consumer threads would use the information stored as metadata in a page to access the relevant memory address and extract the information, encode it, and store this encoded data in another temporary structure which also includes some referential metadata to help organize the final output written to *stdout*.

To sum up our order of execution:

- Producer Thread:
    - Loops through file list and for every file do the following:
        - Memory maps file contents using mmap() system call.
        - Breaks up file data into fixed-size pages (bytes of memory) based on file size.
        - Use pointers/reference variables to keep track of file number, page number, and memory address offset of page.
        - Store all that information in a page structure and checks whether buffer queue full. If full

            ⇒ Global signal all consumer threads to start working and put oneself to sleep until signalled empty from any of them

            Otherwise acquire global lock and directly insert to queue, then release lock and wake up a single, potentially sleeping consumer thread.

        - Once all files are processed, set global *Done* flag/Boolean as True.
- Consumer Threads:
    - Check whether buffer queue is empty and global *Done* flag is False. If so, signal empty to Producer thread and put oneself to sleep until signalled.
      If not empty:

        - Acquire global lock and dequeue a page from the queue.
        - Call zipping function on that page.
        - Use the metadata in the page struct and encoded data returned from the compression function to create another output struct, store relevant output information, and add it to an output stream temporary structure.
        - Release global lock.

Once all producer and consumer threads finish executing, the main function thread can proceed to handle the temporary output stream structure to consistently write the output to *stdout*.

Now a couple of variables to consider when implementing this design are:

- Number of threads
- Page size
- Buffer size

It's also usually problem-specific how many producer/consumer threads are assigned out of the total threads, but in our case it makes sense to use a single producer thread and assign all others as consumers. For the actual number of threads (all), it makes sense to create a number that is equal to the number of CPUs on the machine we're using to run our code. We will be doing that, but we'll also try different numbers and collect some data to analyse.

Regarding page size, we'll start with the native page size of the machine we're using but then again try different sizes and collect data to analyse. Last but not least, buffer size should be big enough to accommodate all threads working concurrently (so at least as big as the number of threads), and so we'll start with a number around 10 but also try a few others and observe the difference in output.

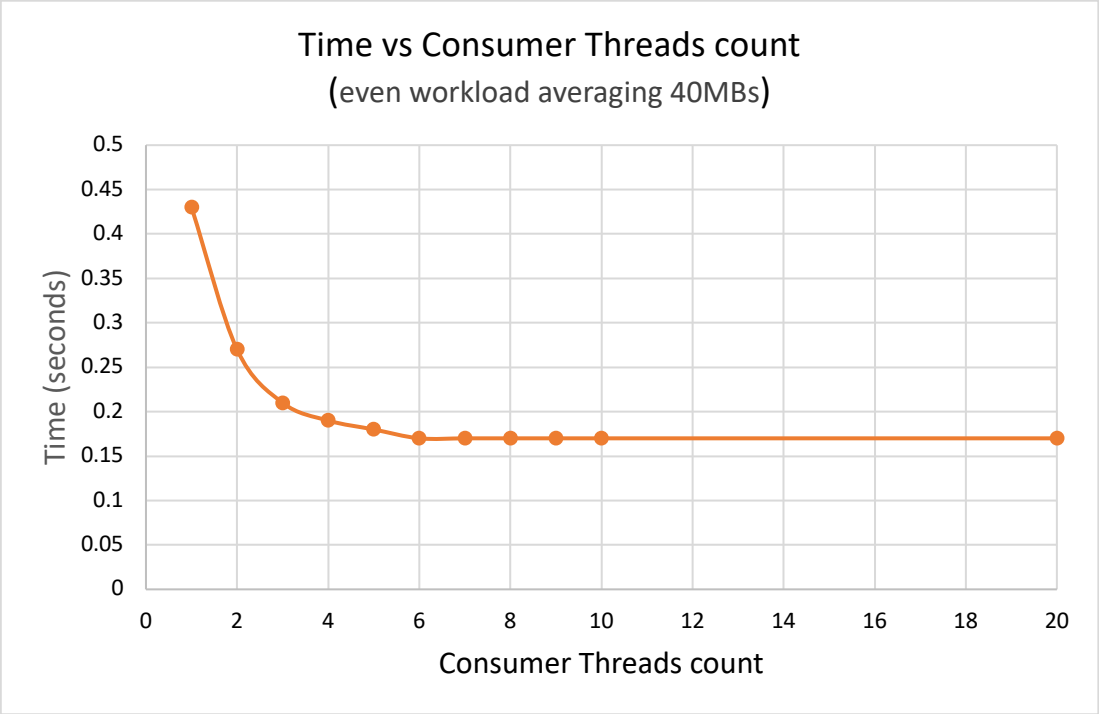
Finally, after finding a combination of values that works well and optimizes our concurrent program, we'll test an identical workload using our 3 versions of the zipping tool: basic linear program, naïve parallel program, and finally our efficient concurrent program. We'll collect data and from multiple test runs and analyse the results.

4. Code Implementation

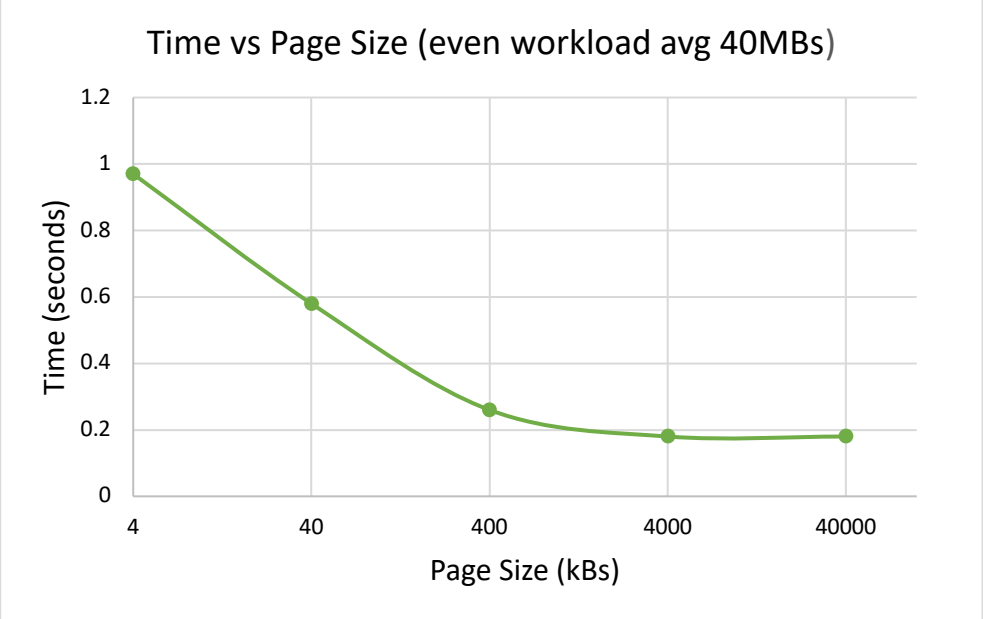
Please see the GitHub repository link in the Appendix for details.

5. Running the Code using different Configurations and Results:

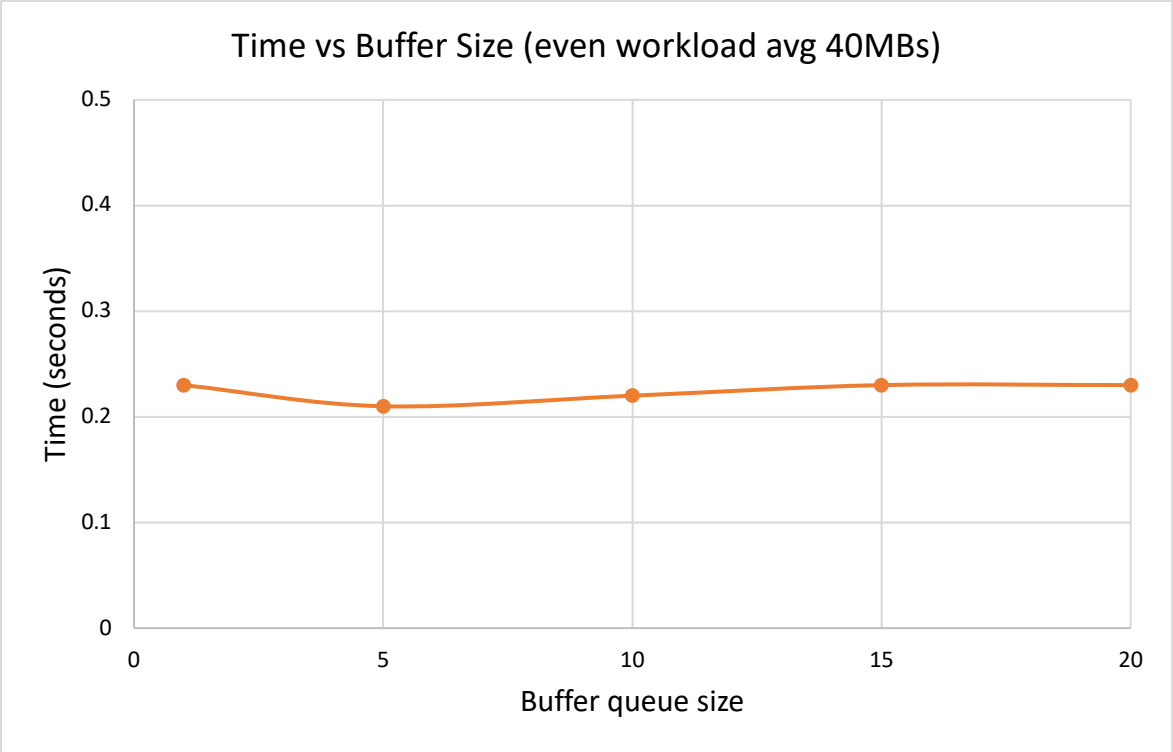
- **Varying number of consumer threads:**



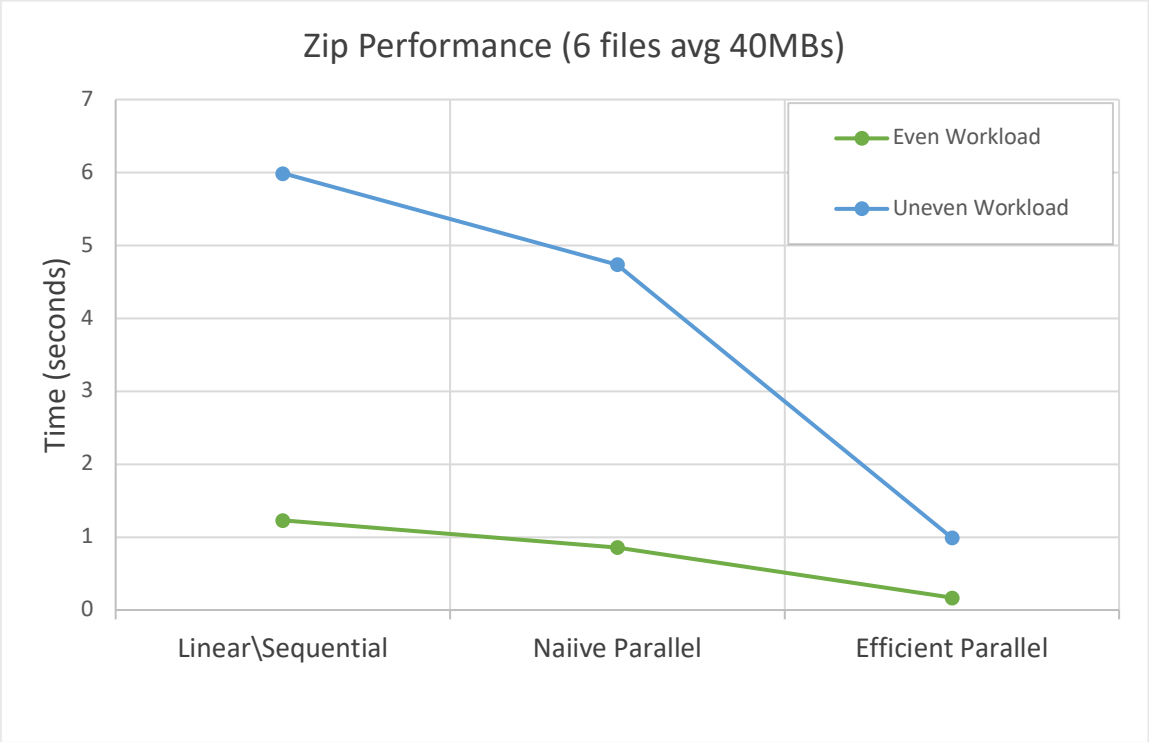
- Varying page size:



- **Varying buffer queue size:**

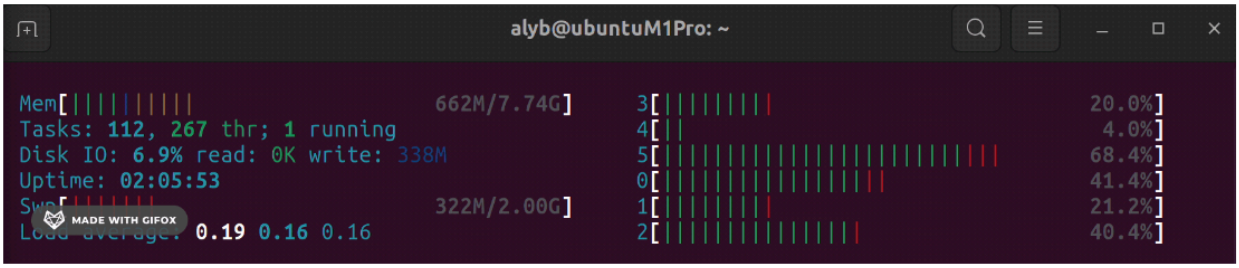


- Comparing our efficient parallel program with the naïve parallel and sequential programs (using the same workload):

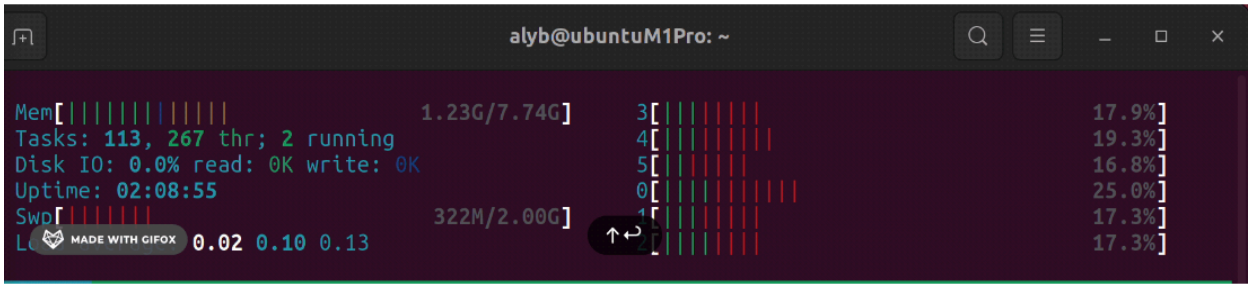


- htop (process viewer) readings (CPUs usage meters) while running our programs

- **Naïve parallel:**



- **Efficient Parallel (producer-consumer):**



6. Discussion and Conclusion:

As you can probably tell from the charts above, we've decided to use a number of threads that's equal to our machine's available CPUs: 6. We've also started our tests with a buffer of size 10, and a page size that's native to our system's page size: 4kBs.

Our tests have shown no particular trends with varying the buffer size starting from a size of 1 to a size of 20. On the other hand, increasing the page size beyond 4kBs showed substantial performance gains, plateauing around a page size of 4MBs. For an identical workload, the program was more than 4x faster with a page size of 4MBs vs 4kBs.

The reason behind this might be the fact that 4kBs of data and a buffer queue size 10 only would mean so much wasted time in the process of signalling back and forth between the producer thread and the consumer threads, along with the acquiring and releasing of the global mutex lock for the buffer.

Finally, we can see the fruits of our labor in how substantially better the performance of the efficient concurrent version of our program compared to the naïve and linear versions. The disparity is even more clear when tested using uneven workloads, which essentially exposes and magnifies the weakness of the naïve parallel design in the straggler problem.

We also used ***htop***; an interactive process viewer software that offers a GUI with options to view CPU readings and memory IOs, to verify that our parallel program is indeed utilizing all CPUs available versus the naïve parallel, which does not.

In conclusion, the process of redesigning a simple linear program to become effectively multi-threaded is one that needs to be approached with caution. It might seem simple at first but simple designs might come at a substantial performance costs.

Even when still relatively better than the linear version, a poor concurrent design might be severely limiting the potential performance gains of a proper concurrent version, as we saw in our tests above. Said proper design might require a major paradigm shift when thinking of the same simple program, and perhaps require the introduction of a lot more aspects in the program, which come with their own challenges and limitations, yet as we saw when all of this is thoughtfully and cautiously planned and handled, the results speak for themselves.

7. References

1. "What is the producer-consumer problem?," *Educative*. [Online]. Available: https://www.educative.io/answers/what-is-the-producer-consumer-problem. [Accessed: 10-Jan-2023].

2. "Lecture 18: Concurrency-producer/consumer pattern and thread pools," *Lecture 17: Concurrency-Producer/Consumer Pattern and Thread Pools*. [Online]. Available: https://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html. [Accessed: 09-Jan-2023].

3. *Operating systems: Three easy pieces*. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/. [Accessed: 09-Jan-2023].

4. *Index of /~amer/651/assignments/project2.sample.data.files*. [Online]. Available: https://www.eecis.udel.edu/~amer/651/Assignments/Project2.sample.data.files/. [Accessed: 11-Jan-2023].

8. Appendix

- GitHub repository link for code:

  https://github.com/alyATB/cs5600-finalproject-parallelzip

- Project prompt as introduced by OSTEP creator:

  https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-pzip