We can use Load Balancing as a technique of ensuring that all processes/threads in a particular [problem are treated in such a fashion that they all finish at approximately the same time. We frame the problem in the form of a map/reduce problem, and then use Static Load Balancing (S-LBAN) to adjust the process load for optimal net performance.

The question is how accurate can we do this? Is it worth the effort? Clearly Amazon's Elastic Computing (EC2) tells us that this is so on a large scale, which typically includes a lot of data movement and networking. But what about in a real-time system, with no significant networking – the data arriving in a batch before the program begins, and having repeatable statistical properties?

In other words, we measure, adjust, and deliver optimally.

The central task of this assignment is to produce an evidence-based report answering the above questions.

Methodology

We propose to answer the RQ by creating a series of tasks. First some simple ones in order to establish a baseline of result expectations, then to rewrite the tasks using the map/reduce software pattern to produce a set of concurrent threads of different complexity. We can then adjust the scheduling in order to answer the research question.

2. Overview of Methodology

In this assignment, you must write C/C++ code to implement a map / reduce problem. Map() is a process or function that maps the input problem to the available resources which may mean subdividing the problem into components and solving each of those component problems using prioritized job scheduling.

Then reduce() gathers the component solutions and combines them to form the global solution.

You will be provided with "dirty data" and the assignment is then divided into a number of tasks, each of which purports to solve the problem a different way. In each case, there will be a performance measuring phase, and for the later methods, an adjustment phase based on the performance metrics obtained.

Throughout, you will be measuring and properly documenting performance (and hopefully improving it)

In the last task, where streaming data is used, it will be important to reduce idle time by ensuring that all subtasks created by map() finish at the same time, so that reduce() is not kept waiting.

General

Your solution to the problem you choose must not use busy waiting, which is where a program sits in a tight loop until some condition is met as this waste's CPU time. Instead, you must have each thread sleep until a condition is met (use a condition variable), wake up and do some processing and then perhaps signal other threads that the job has been done. Please ensure sufficient output so that it is clear when your program performs any actions such as adding to an array, locking a lock, etc.

You should ensure in your design of the program that you only share between threads the minimum state (variables) possible as the more information that is shared the more likely there is to be a conflict (deadlocks, race conditions, etc). Please see the courseware material for more detail on these problems. If your algorithm requires randomness, you should ensure you seed the random number generator correctly.

To ease marking of your assignment, you must call your executables "taskN", where N is the task number as described below. All materials necessary for the markers to build your tasks must be included in the submitted ZIP file. The markers will use the CS servers (titan, saturn, jupiter) for the marking, and the code is expected to run there.

Makefile

Your solution must be written in C / c++ and you must supply a Makefile that builds your solution incrementally and then links them together. If you are feeling a big rusty about make files or have not used them before, we recommend going through the following tutorial:

https://www.tutorialspoint.com/makefile/index.htm

Compiler Settings

Please note that as a minimum you must use the '-Wall -g' flags on gcc or equivalent on other compilers to generate warnings. You may use any supported c++ compiler on the server - if you wish to use a standard above c++ on the server with g++ you will need to use the scl command, e.g.:

```
scl enable devtoolset-9 bash
```

This should get you gcc version 9.3.1 (2020) instead of gcc 4.8.5 (2015). Use "gcc –version" to confirm.

Graceful Exit

In order to account for the possibility of thread starvation, you will need to gracefully exit your simulation at the end of around 10 seconds. We would normally expect even the slowest task – task1() – to not take longer than 10 seconds to execute.

Use the following method to do this: once you have launched all the required threads from main, call the sleep function, to specify sleep for ten seconds. Once the sleep function finishes, change the value of a global variable to indicate that all the threads should exit, which they should be checking regularly.

A less graceful exit might be to call exit() after 10 seconds, but that risks leaving zombies behind with unfinished business – potentially leading to data loss.

Valgrind

The solution you submit will ideally be as bug-free as possible including free of memory bugs. Your markers will mark your submission on the titan/jupiter/saturn servers provided by the university and we will use the tool "valgrind" to test that your program is memory correct (no invalid access to memory and all memory freed) and as such you should check your program on titan before submission with this tool. This is for debugging and memory testing only. When gathering performance data, you should do this on a dedicated machine or VM, since on titan, being a shared resource, the timings will obviously not be repeatable.

The following command:

```
valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all ./simulation
```

Will run your program with valgrind. Please note that in this case the name of the executable produced by compilation of your program is 'executable'.

Allowed Concurrency Functions

For the concurrency part of the assignment, you must limit yourself to the following functions:

- pthread_create
- pthread_join
- pthread_detach
- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_trylock
- pthread_mutex_lock
- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_init
- you may also need the pthread_mutexattr_* functions
- you may also use the scheduling priority function.

This list is not exhaustive and so if you are unsure, please ask on the discussion board about the function you wish to use.

Please note that in practice beyond this course, you would use higher-level c++ functions that call these functions, but part of what you are learning here is the underlying calls that are made as part of managing concurrency. That is, part of what you are learning is to apply the theory that you learn in the workshops to practical problems.

The Source Data

To start off, you may use a words list conventionally stored in '/usr/share/dict/linux.words' as a clean data file. You may need to install this in your distro. Titan does not have it.

The actual data to be used is at https://www.keithv.com/software/wlist/.

You will find a number of ZIP files containing text files containing (ideally) 1 word per line which you read into an array for subsequent processing. In fact, the data is 'dirty' and you will need to clean It up first. In your performance measurement using Amdahl's Law, this is the serial part.

Performance Measurement and Reporting

You will need to devise a way of measuring improvement using primarily execution time, but also resource usage. Consider some of the metrics discussed in class.

You then need to describe this using graphs and other ways to describe how what you did improved performance, and why. See reporting details below.

5. The Tasks in detail.

The fundamental task is to receive a set of words, clean it up, then use map/reduce ultimately to sort it. Words of length 3-15 characters are to be sorted on the third character onwards. Words of other lengths are to be ignored (filtered out). The tasks below achieve this in different ways.

Below you will find a selection of five tasks called task1 … task5 that must all be done in sequence. In each case, the simulations should not exceed about 10 seconds and terminate cleanly - no crashes, no deadlocks, no race conditions. This should be enough time to gather event statistics. You may need to replicate the data in order to make it run long enough. If so, then please describe what you did.

You should also print a line of text for each action in your program such as adding an element to an array. All normal messages should be printed to stdout (use cout) and error messages should be printed to stderr (use cerr). This will allow your marker to capture normal output and error output separately if they wish to. In bash, a command like:

./task2 (args) 1>outfile 2>errfile

would do the job (where args are whatever arguments deemed necessary).

Be sure to prefix each process/thread with its identity, where needed. I would also timestamp the data reported. An example of this is the additional output given when the -v or -vv or -vvv options on ssh are used.

Performance Measurement

Use Linux-supplied tools and functions and your judgement for performance measurement (for example: 'perf', 'profile', 'ftime', 'getrlimit', 'gccpg', 'perf stat -d', or others) . In the report you must justify the tool you are using and what it actually measures.

Map() / Reduce()

For each task, you will use a different mapN() to separate the words list into separate lists,, each to be executed according to the task. A corresponding reduceN() function will gather the distributed output produced by map() to a final outcome, again different for each task. The N above refers to the task number.

In all tasks , the same file should be produced as output for a given input. In that sense, all tasks are equivalent – the only difference being HOW they do their job.

You should call the function mapN() and reduceN() where N is the task number, to make this clear to the markers.

Task 1: Task 1 – *Manually* Capturing, Examining, and Filtering the data

- Use the coreutils programs ("grep, sed, sort, etc..") to discover ways of cleaning the data. The files may contain punctuation and other symbols. Also remove duplicates. Consider and document some simple filtering rules, In order to come up with a clean data file. Finally, since sorting an already sorted file is not a good test, use 'shuf' to shuffle the data into random order.
- Distill what you created into a shell file called Task1.sh
- Now replicate this filtering and shuffling in a C/C++ function called TaskFilter(), and compile this separately as you will link to it in the subsequent tasks. Make a main() that simply calls this function of the file supplied as an argument on the command line as

  Task1filter DirtyFile Cleanfile

  Where the resultant clean file should be identical to the file produced by the coreutils and shell script tools above.

- Report is to include:
  a. The combination of coreutils tools used to generate the equivalent of Task1filter.
  b. The source of Task1filter
  c. The number of words of length 3 to 15 letters in the data set you end up with. Include this data set in your submission. This forms the basis for the load balancing statistics you may use later. Ignore words of length 1,2, or more than 15 letters
  d. the performance data for this task

Task 2: Process based Solution

Read the data from the original file. Use *Task1filter* on the original data to produce a clean version . Then do the following.

- Use **map2()** to separate the words list into separate lists, one with words of length 3, the next of length 4, to 15. Ignore words of length 1,2 or more than 15.
- Map2() then uses fork() to call "sort" and have it sort each file on the **third** letter of each word using appropriate arguments, saving each result in its own file.

- Use reduce() to open each of the 13 files reading one line at a time for each file, and writing the lowest sort order word from each file, and then reading the next word from that file. In other words, a 13 → 1 merge sort.

Task 3: Threads using shared memory without conflicts

Here, we use FIFO files to transfer the data from map3() to reduce3(). Specifically

- The main program calls the filter function to clean the data. The function will return a global string array of valid words. then main() creates both the map3() and reduce3() threads.
- The map3() function creates 13 index arrays and threads, one for each word length, indexing into the main string array. So if the global array contained ["air", "airbag", "and"] an index array had [1,3,…] then global[index[0]] is 'air', global[index[1]] = 'and'.
- Global[] is only ever read by map before the threads, and only thread3 reads index3[]. Since no word can be two lengths at the same time, there will never be a conflict between threads .
- Having created the mapping, map3() now creates 13 threads. Each thread uses the C function qsort() to perform the same sort as before – on the third letter onwards – and when done, it creates 13 FIFO files opened for write and each thread will then output words to their corresponding file.
- In this case, the reduce3() function will wait, and then read those same 13 files, one line at a time in sort order. Note that qsort() sorts the indexes, not the array itself.
- This should produce the same file as Task2 (and Task 1 – after sorting.)
- At the end, each thread opens a FIFO file for writing, and dumps the global strings in sorted index order.
- reduce3() should keep a count of words for each thread.
- Meanwhile, reduce3() needs only to perform the reduction merge step. It waits for each map3() thread to signal its completion. When all are done, reduce3() will open all 13 FIFO files for read, and perform the merge step as in Task2,
- Report: Compare performance. Consider where the time was saved or lost.

Task 4 Optimizing thread performance

It should be self-evident that some word lengths are more frequent than others, so some threads will take longer to complete than others.

For this task

- When, map4() – same as map3() - creates the thread, it additionally assigns them a specific priority based on the performance ratios in Task3, so that all threads finish at roughly the same time. Think how you would do that? (Hint: 'mice')
- Does reduce4() need to change from reduce3*(?
- To get reasonably reliable performance data, you may need to pad out the input data with more duplicates.
- Can you tweak the thread performance so that they indeed finish at about the same time?
- Now compare the tweaked priorities with what you believe they should be. Do some research to discover how the priority levels are related to speed.
- Describe all that you did in the report and what conclusions you came to.

Report: Document the following.

- Where are the threads spending most of their time?
- Can you come up with a big O representation? Justify your reasons.
- For the larger word lengths the string compare will also take longer. Is that significant?
- By checking the performance of the Task 3 threads, how close was the ratio of thread performance with the big O expectation? For example if thread was $O(n^2)$, then for larger n, a 3000-word thread should run 4 times as long as a 1500-word thread. So we have execution time and word count as two metrics for comparison.

Task 5 Converting to Input Stream

Instead of the data coming from a fixed file, it comes from a streaming input source. There are several implications in this.

- Can you sort streaming data? Try oy with "cat - | sort". When do you get to see the sort output? Why?
- Now you cannot preselect historical ratios. Instead you must be adaptive. You have word count and cumulative execution time for each thread. What can you do with it?
- Let us suppose you block the data, and sort each block. How does that change things.?
- Report on all the above and any other relevant considerations.

For this final task

- Create a 'stream server' ,
    a. This reads the whole original (dirty) file into a string array.
    b. Then the server outputs a random entry to a FIFO file that is connected to the map5() input stream at some constant rate.
    c. In this case, since the rate is slow, map5() will need to know when the FIFO is non-empty. You cannot use a busy-wait loop to find out.
- Report:
    a. Give a detailed description of how you would change Task4 in order to make it streamable, but without sorting (so no blocking needed) So here map5() does not change, but the threads do.
    b. Describe what CPU scheduling algorithm could be used if there was no sort but there was a precedence rule that said that shortest words pass first in reduce5().
- Implement the above task rescheduling.

th