ECE 421 Design Document

Kirby Banman

Ryan Thornhill

Mar 1, 2015

1.) What is your definition of an object?

An object is a particular instance of a class where the object can be any combination of data and functions.

2.) What strategies should be deployed in terms of accepting input (i.e. the large number of objects.)?

Objects should be able to be supplied via the command line or from a file.

3.) Is your sort generic? What sorting criterion does your system use? Is this criterion flexible; i.e. changeable by the user at the point of execution? What limitations have you placed upon legal sorting criteria? To make it reusable to other people, how should we include it into the Ruby Class hierarchy?

The sort will be generic we will allow for the sorting of any object as the user will define the comparator. This means the criterion will be flexible and changeable by the user at runtime. A legal sorting criterion must only return -1, 0 or 1 for a comparison between any two elements of the passed set. Overriding the sort method from the enumerable class with our own implementation is an option, we could also ensure we only use features that come with the standard Ruby libraries so that our module could always be used.

4.) In reality we have a uniprocessor system, describe “equationally” what is happening to your solution as you increase the concurrency (e.g. produce a regression model (remember regression from Statistics) of processing time against the number of threads. Your solution can be modeled as a program which: (1) has a component which produces threads; and (2) a set of threads which undertake the same (small) task. This is in essence the basis of stress testing (discussed in ECE 322)

Symbolically, for a particular algorithm:

T_1	:= serial runtime of the algorithm, with 1 thread on 1 processor
p	:= number of concurrent executions (threads, in this case)
x	:= fraction of the algorithm that is executed in parallel
(1-x)	:= fraction of the algorithm that remains serial (thread creation overhead, etc)

Then the runtime of the algorithm as a function of the number of threads is:

$$\begin{aligned} T(p) &= xT_1 / p + (x-1)T_1 \\ &= T_1 (x/p + x - 1) \end{aligned}$$

This is Amdahl’s law, which expresses that an algorithm can be sped up by parallelization by a maximum factor of p (the number of threads), which requires perfect parallelization (x=1). Also, since thread *creation* is a serial task, creating more threads decreases the parallel fraction x. Hence, increasing the number of threads will eventually result in slower execution.

5.) Concurrent systems tend to crash frequently – what approach to exception handling have you devised? Consider the content of the library at: http://c2.com/cgi/wiki?ExceptionPatterns; which are applicable to this problem? Is Module Errno useful in this problem? What components of the Ruby exception hierarchy are applicable to this problem? Discuss in detail your strategy for exception-handling.

The ExceptionsTidyThreads pattern seems to be very applicable to this situation. We will want to make sure that any resources threads have allocated during their runtime are released before we simply kill the thread. The Errno module seems to have limited applicability for this system. However, if we are to have a highly accurate duration time, we will reuse the timing module from the last project, which does make use of the Errno module.

The ThreadError exception will be particularly important for us since the focus of this project is using a large number of threads, allowing them to fail gracefully will be important. RuntimeError and StandardError will also have applications in this sorting system as we will have to be on the look out for mismatched types that can't be compared and similar errors.

Our exception handling will rely on a let it fail methodology. If a thread runs into an exception, we will let the thread fail gracefully. We won't try to correct errors that have happened on a single thread. We will restart that thread to rule out transient faults and if the thread fails for a second time we will force the entire sort to fail. Our threads won't have much, if anything to clean up when they are being killed so we don't have to be extremely vigilant for things like memory leaks.

6.) What differences exist between thread-based and process-based solutions? How has this impacted the design of your solution?

Thread based solutions require only one heap space that is shared among the threads. Processes require global data constructs to share data and communicate with each other as the have distinct stacks, heaps and copies of the program. This means that the overhead for starting and stopping processes is higher than that of threads. In the interest of efficiency we feel this solution lends itself well to the threading model. Furthermore, threading will give us fine grained control over restarting a failed part of the sort.

7.) Do you have any race-condition or task synchronization concerns about your solution? How do we tidy-up a multi-threaded program, if stopped mid-execution?

The recursive calls to merge sort are going to require synchronized calls to merge as the merge cannot be performed until the sort is complete. Since merge itself is also a recursive algorithm there will be a level of synchronization between successive calls to merge. Our

“Thread Generator” will be notified of thread exceptions, if it is required that we must exit the program, we will have the generator kill the remaining threads.

While typically we would want the thread “destructors” to unwind and changes they have made and release any resources. We will not be writing files or allocating resources from our worker threads so we won’t have to be worried about cleanup as they are killed.

8.) As discussed in ECE 320: What is configuration management? What is version control? Are you using either concept? If “yes”, describe your process and any tool support what you utilize – illustrate your process with regard to Assignments 1 and 2; if “no”, justify your decision.

Configuration management (CM) refers to a discipline for evaluating, coordinating, approving or disapproving, and implementing changes in artifacts that are used to construct and maintain software systems. -
http://www.sei.cmu.edu/productlines/frame_report/config.man.htm

Version control is a smaller subset of Configuration management and is defined as:

the task of keeping a software system consisting of many versions and configurations well organized.

We make heavy use of version control, and thus Configuration Management primarily with the Git and GitHub software. Design by contract is another important concept under configuration management that we have utilized in both the previous projects. However, these projects aren’t quite large enough to warrant the use of issue trackers and other tools under the Configuration Management category. For projects one and two, version control enabled us to work on the project concurrently with minimal effort in merging our solutions. As well as giving us confidence to try unique solutions without fear of breaking the currently working code base.

9.) Briefly Explain:
a. What is refactoring (as discussed in ECE 320)?
b. Are you using refactoring in your development process? Justify your answer?
c. If “yes”, give examples, minimum of 2, of the refactoring “patterns” that you used in Assignment 1
d. If “no”, give examples of where your solution to Assignment 1 would be improved by applying refactoring patterns. Supply a minimum of two different (i.e. different refactoring patterns) as examples.

a. Refactoring is the process of restructuring existing code without changing its functionality.

b. We make only small use of refactoring. Typically these projects already require a significant portion of our very busy schedules so we don’t have much extra time to make the code nicer without improving the functionality. Occasionally we need to refactor in order to cleanly integrate the next part of the project into the current code base. But we typically plan our design from the beginning to avoid the need to refactor like this.
c. n/a
d. There are a couple of pieces that would benefit from refactoring. First we could refactor all of our “factory” methods into a single method that takes some kind of “type” parameter. Secondly, we could refactor some of the methods that use long iteration to only iterate over the non-zero elements, increasing efficiency.

Changes

Contracts:
The merge and partition methods were initially contracted to never mutate input arrays. However, for performance and for easy inter-thread communication, a thread- and recursion-global output array is mutated throughout.

We decided to make another module to handle the shell interaction, there were extra contracts added to support that functionality.

Design:
2.) We chose to accept only a file input, a path to a file containing a newline separated list of strings is supplied at the command line. If you use the sorting module on it’s own without the “driver” we created, you can supply a list of ANY object that can be compared using <=>

3.) In the interest of limited time we chose to implement the sort for any class that implements the “<=>” operator and hence uses the comparable module. In our solution a custom comparator cannot be passed in the form of a block to the sorting routine.

5.) Again, in the interest of time we did not implement the restarting of threads in order to rule out transient failures. We decided to do as much error handling as possible before starting the algorithm. A transient failure could cause an incorrect sort. However, our contracts would catch that error, so the result would not be passed off as a correct sort.

Additional Testing:
For the major parts of code (Parsing files and the sorting algorithm) we developed test suites that are compliant with the minitest library. We primarily used a method of error guessing and equivalence classes to develop comprehensive test cases. Please see the files postfixed with “_tests” for our testing code.

The driver program that interacts with the shell did not lend itself well to testing as it requires mutating the ARGV global variable in ruby. Because of this complexity we choose to test this module manually.

Problems

Because of the suggested thread model, a list of thousands of items may cause thread exhaustion on some machines. (i.e. Ruby is no longer able to create Thread objects, and the rest of your OS will not be able to create more processes.)