Parallel-Method-Project-Write-Up

By Robert Blum

How It Works

This project aims to measure the difference between two methods of parallelism and concurrency in web crawling. The project contains two methods, one using a breadth first search algorithm and futures, and the other uses the Actor model to send and receive messages between actors. In the main class, the primary method to run the project is `runBenchmark`, which takes in the number of runs each method will run, and the seed path.

```
def runBenchmark(n: Int, basePath: String)
```

For example, running the code below produce the following output.

```
val TEST_LINK1 = "https://cs.muic.mahidol.ac.th/courses/ooc/api/"
runBenchmark(10, TEST_LINK1)
```

```
TEST RESULTS ========================================================
testing link START: https://cs.muic.mahidol.ac.th/courses/ooc/api/
Future Mean: 35.60596308436364, Actor Mean: 36.151467606272725
Future stdDev: 3.3522533717624072, Actor stdDev: 2.671289226679779
```

Each iteration of each method also crawls each link for certain information, including:

- Number of files crawled in total
- Number of unique file extensions found
- A map of each extension pointing to how many files of that extension are found
- The total number of words found outside HTML tags

Each run outputs these stats into the console as well.

```
Future Crawler took: 37.410880155 s with result: WebStats(10349,5,HashMap(jpg -> 2, png -> 121, css -> 3, gif -> 45, js -> 5, html -> 10173),6728700)
Actor Crawler took: 35.114015914 with result: WebStats(10351,5,HashMap(jpg -> 2, png -> 123, css -> 3, gif -> 45, js -> 5, html -> 10173),6721017)
```

Method One: BFS & Futures

This method follows the method in assignment 3 of the Functional and Parallel Programming course at MUIC. However, this was improved code that, in my opinion, is much cleaner, as well as being faster than the code I wrote in my assignment. This method works by first extracting links from a page, then the collection of links is run through BFS to extract links that have not been visited before. This process then repeats until no new links are found.

Method Two: Actor Model

This method follows the actor model. Actors are objects that are able to send and receive messages, while also being able to do computation depending on the messages that the actor receives. Actors work behind the scenes (similar to futures) concurrently. This project uses Akka actors in its implementation. The implementation in this project uses a master actor with worker actors, where the master delegates tasks and sends links to each worker to crawl, and then the worker does computation and when it is finished extracting the relevant information from the page, sends all links it finds to the master. The master then filters these links and continues to delegate these tasks.

Findings

Each crawl of the site https://cs.muic.mahidol.ac.th/courses/ooc/api/, produces runtimes that are quite varying. As seen from above, the test (certainly without enough iterations) shows that each method has really similar runtimes. This is a little bit discouraging as I spend quite some time learning how to use Akka actors, which only resulted in barely noticeable differences in speed.

Unfortunately, I was unable to test out and eliminate variables such as latency and depth of crawl in the benchmarks.

Conclusion

On the other hand, I feel quite happy that I now have a new tool under my belt. Actors seem to be really useful in large distributed systems. Coding using actors was non-trivial but its patterns are quite intuitive and easy to understand.

Although actors and the future bfs implementations seem to result in similar runtimes and benchmarks in this small test, I still believe actors to be a superior coding design, due to them being thread-safe encapsulated units of computation.