Advanced Programming and Paradigms

Project report

Author: Marco Carlo Cavalazzi - N° Matr.: 0000644460
Professor: Alessandro Ricci
February 2015

1. Description of the problem

The purpose of the project is to create a utility that indexes and keeps indexed all the documents stored in a directory (including its subdirectories), so as to make it possible to retrieve the list of the documents containing some words in a very efficient and fast way. Here we consider as documents just files with extension ".txt". The utility should make it possible to:

- build the index, by suitably processing all the documents in all the directories specified by the users
- keep the index updated, monitoring periodically the specified directory
- execute queries, getting the words to search from input and displaying the list of the files containing the word in output

So the aim is to write a concurrent program implementing the utility. Moreover, it must have a suitable GUI to:

- select the directory to be indexed
- manually control the index creation process - providing controls for starting, stopping, pausing the process and components to show the state of the process
- execute the queries and show the results

2. Concurrent architecture adopted

The project has been approached using the different architectures studied during the course, choosing then the one that best addresses the problem.

At first, it has been considered the "Announcer-Listeners" design. Reading the definition it is clear that it is not meant to be used for a purpose like this, because the aim of the project is not the reaction to some input but the active analysis of folders and files.

The second architecture considered has been the "Filter-Pipeline". This configuration could be used in our case. It would be possible to create some threads to work with the files as *generators*[1], another one containing the function to map the words as the *filter*[2] and one or more threads as *sink agents*[3] that collect the output of the filter agent and store it properly. It could work. This said, this approach could also potentially create a bottleneck situation in which the filter agent, having to deal with the information gathered from all the generator agents, would slow the performance of the whole system down.

[1] - Generator agents are the agents starting the chain, generating the data to be processed by the filter agent.
[2] - The Filter agent is an intermediate agent of the chain, consuming input information from a pipe and producing information into another.
[3] - Sink agents are those that terminate the chain. They collect the results.

The final architecture considered has been the “Master-Workers” one. Starting from this solution, it has been created a personalized version of it in which there is a plurality of Master elements. The initial Master element can decompose the global task in subtasks for which:

- If there is a “.txt” file → a Worker agent is created
- If there is a sub-folder → a new sub-Master element is launched

We will refer to this approach as the “Masters-Workers” architecture.

The “Masters-Workers” architecture has been used in combination with the "Result Parallelism" design. Thus, the system is implemented so that all the agents can work in parallel and produce a piece of the final result simultaneously.

The kind of problem decomposition used is, thus, the "Recursive decomposition": each Master will divide the problem into sub-problems, which will be again divided into sub-sub-problems recursively until all the folders stored in the main one have been examined.

3. Structure and behaviour

In this paragraph are explained in more detail the main elements of the project, which has been developed using the NetBeans IDE 8.0.1 and as the rich client application package JavaFX.

The Masters-Workers architecture is implemented creating a first thread that analyzes the main folder, chosen from the user through the GUI, and creates:

- one sub-Master for every sub-folder and
- one Worker for every “.txt” file found in the selected folder.

Following the Recursive decomposition design, each sub-Master (called *FolderThread* in the code) will then follow the same steps. This will enable the system as a whole to look in every sub-folder and launch a Worker thread (called *FileThread* in the code) for every “.txt” file.

Thanks to the concurrent Java's Abstract Data Types, it has been possible to implement the Result Parallelism design through the ConcurrentHashMap (see *java.util.concurrent.ConcurrentHashMap*). This particular structure represents a hash map supporting full concurrency of retrievals and adjustable expected concurrency for updates.

The Workers will examine the file given in input and update new information to the shared ConcurrentHashMap, maximizing the parallelism while addressing the development of the solution. Every Worker modifies the same data structure calling basic methods already implemented in Java. For each entry, the ConcurrentHashMap will have:

- a word (as "key")
- the list of files containing the word (as "value")

- Here is represented an exemplification of the behaviour of the Master threads in pseudo-code:

```
foreach( sub-folder sf ){
	// Launches a new sub-Master thread for that sub-folder
	FolderThread( sf ).start()
}
foreach( txtFile tf ){
	// Launches a new Worker thread that will analyze the file
	FileThread( tf ).start()
}
```

- The following is a pseudo-code representation of the main part of the Worker threads:

```
foreach( word in file ){
	word ← next word in file
	if( word is not in the HashMap ){
		list ← new List()
		list.add(file)
		HashMap.add( word, list )
	}else{
		filesList ← list.getValue( word )
		if( filesList.contains( file ) ){
			// do nothing
		}
		else{
			filesList.add( file )
			HashMap.replace( word, filesList )
		}
	}
}
```

In order to be able to communicate to the user the progress of the program during the indexing process each Master and Worker passes a string to the thread called “guiUpdaterThread”. The latter develops the Producers/Consumers design, adding the received strings to a queue, which will then be iteratively read in order to post new messages on the GUI.
For this part of the program, we can think of N producers (the Masters and Workers) and one consumer (the GUIUpdaterThread), which will add the given strings to a *LinkedBlockingQueue*, which will order the elements as FIFO (first-in-first-out). The thread will then iteratively check if the queue is empty. If not, it will print each new statement in the GUI.

Once the index is created a new thread is launched that iteratively waits one minute and checks if a new index is being built. If not, it refreshes the previously created index automatically, simulating the press of the “Build index” button in the GUI.

It is important to notice that this particular approach, with the ConcurrentHashMap and the LinkedBlockingQueue, allows us to be certain that there will be no interference between threads nor errors while accessing the shared resources. Thus, no *deadlock* (or *deadly embrace*), no *livelock* and no *starvation* situation is going to happen.
The only situation possible in which an exception could be created would be while appending a huge amount of data to the LinkedBlockingQueue in a very short time, in order to use all the RAM of the computer. Nevertheless, this case will never happen thanks to the iterative removal

operations done by the GUIUpdaterThread and the limits imposed from the development environment and the operating system to the amount of RAM available for the program.

4. Testing

4.1 Indexing performance

In order to measure the performance of the system I studied the time needed to build the index for a small, a medium and a big amount of folders and files.

In numbers:

Test size	N° of files	N° of folders	N° of files ".txt"
Small	1512	272	14
Medium	17238	1558	31
Big	119174	27267	106

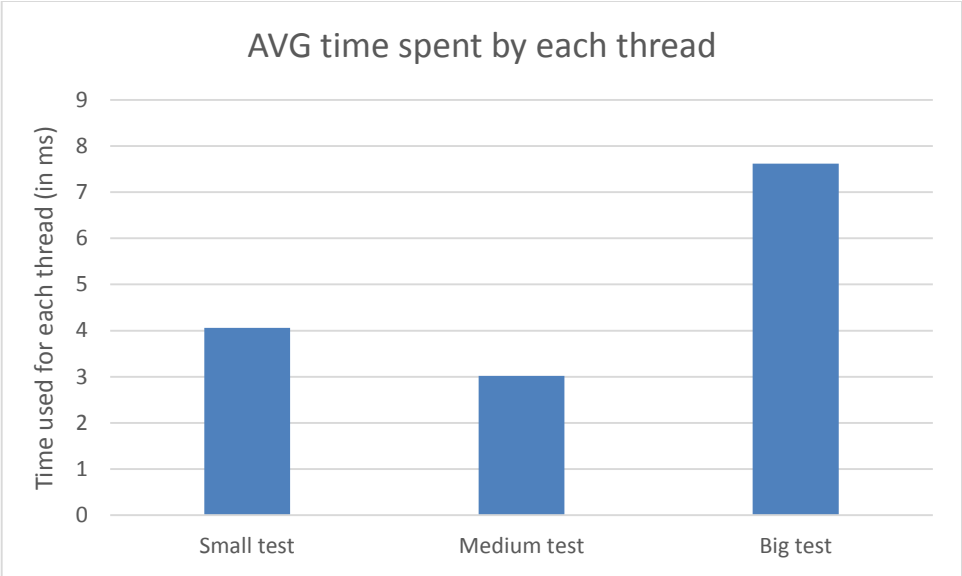
The tests have been run on a HP ENVY quad-core 2.2 GHz with 16GB of RAM and a 64-bit architecture mounting Windows 8.1. They have been repeated many times in order to be able to see if the results were consistent or if they may vary according to the Operating System's use of the scheduling and the volatile memory (cache, RAM).

The following table shows the results:

Tests	Small	Medium	Big
Indexing timings in ms	1252	5019	243949
	1272	4702	191766
	1228	4748	238463
	1188	4884	207183
	1225	4829	256303
	1157	4730	180665
	1154	4754	178542
	1170	4945	170216
	1177	4621	205692
	1151	4778	213267
Minimum elapsed time	1151 ms	5019 ms	170216 ms
Maximum elapsed time	1272 ms	4621 ms	256303 ms
Average elapsed time	1161 ms	4801 ms	208604.6 ms

From the data collected we can clearly see that the indexing process requires a little amount of time to examine a small or medium amount of folders and files, but it greatly lowers its performance when the amount of created threads is high.

In order to understand how much the time needed for each thread increases according to the size of the test, we can divide the average elapsed time by the sum of the number of folders and ".txt" files used for each test. Here the results:



As said, from the graph we can recognize that the amount of time used for each thread in the small and medium tests is similar, while the time used in the third one is the double.

4.2 Query performance

Thanks to the HashMap's design the duration of a query requires an average time of 15 ms, even with a big number of files and words considered.
This is because all the program needs to do to find the files with those words (given in input from the user through the GUI) is to look for the hypothetical list of files containing the chosen word in the HashMap, using the function *get(key)* where the *key* is the chosen word. If the word is present in the examined files the program will return the attached list of files containing it and it will show to the user every filename with a hyperlink to the file, useful to check the correctness of the results.
If more than one word is used the system will retrieve all the files containing all the words. If only some of them are stored in a file, that file will not be considered.

5. Conclusions

The development of this project has given me the opportunity to study in practice the possibilities of concurrent programming and its effects on a relatively limited hardware.
First, I noticed that even though Java may not be the fastest of the programming languages it manages to handle perfectly a relatively big amount of work in a small amount of time. The creation and management of new threads has been easy to implement, once chosen the proper data structures.

The only disadvantage found in the concurrent solution implemented has been the lack of hardware resources. In fact, even if the program keeps the time needed to analyze a small and medium amount of data low, it encounters some difficulties when launched on a big test set. The time needed to perform the duty increases drastically, most probably because of the garbage collector, that does not manage to release the occupied memory from the early threads fast enough to avoid having to use all the available RAM for the newly created ones.

Even though the project manages to solve any task (given enough time) we can notice that a more powerful architecture would have greatly speeded up the performance. Running this program on a computer with a higher number of CPUs and more RAM would definitely make it achieve results in a quicker way.

Using the concurrent approach it has been easy to apply the concept of divide-et-impera to the task, launching a new thread for each sub-task. This, nevertheless, has introduced other problems related to the amount of resources needed for each thread. In the end, I believe that the concurrent architecture is useful as long as the number of threads does not exceed the amount that is easily manageable from the available hardware.

As possible future developments for this project, I would configure the Master threads to not only launch new Masters for the sub-folders and new Workers for the “.txt” files, but to take care of the analysis of the “.txt” files present in their folder themselves, while launching new threads for the exploration of the sub-folders.