1. Task Parallel in C#

1.1 Getting Started

The task was to develope a Windows C# console application which finds and lists all duplicate files in given start paths and their subdirectories. The application should be optimized by process time and memory management with the use of tasks and other algorithms.

Specification

- The application should be built for batch processing. That means no other user inputs are necessary to run the application
- The application should print all console outputs and the count of read 0 and 1 bits to **stdout**
- Errors need to be printed to *stderr* output

Here is an example on how the application can be called and it's parameter:

cntFileBits [-r [n]] [-f fileFilter] [-t maxThreads] [-h] [-p] [-v] [-w] [-s startPath]

- **-s startPath:** Gibt das Startverzeichnis an, ab dem die Dateien gelesen werden sollen; die Option -s kann auch mehrfach angegeben werden, z.B. wenn zwei Partitionen durchsucht werden sollen
- **-r [n]:** Rekursives Lesen der Unterverzeichnisse; wenn n (bei n >= 1) angegeben, dann bestimmt n die Tiefe der Rekursion; wird n nicht angegeben, dann werden rekursiv alle unter dem Startverzeichnis stehenden Verzeichnisse und deren Dateien gelesen
- **-f fileFilter:** fileFilter gibt an, welche Dateien gelesen werden sollen; z.B. *.iso oder bild*.jpg; wird diese Option nicht angegeben, so werden alle Dateien gelesen
- **-t maxThreads:** maximale Anzahl der Threads; wird diese Option nicht angegeben, dann wird die Anzahl der Threads automatisch optimiert
- **-h:** Anzeige der Hilfe & Copyright Info; wird automatisch angezeigt, wenn beim Programmstart keinen Option angegeben wird
- **-p:** Ausgabe der Prozesserungszeit auf stdout in Sekunden.Millisekunden
- **-v:** Erweiterte Ausgabe etwaiger Prozessierungsinformationen auf stdout
- **-w:** Warten auf eine Taste unmittelbar bevor die applikation terminiert

1.2 Concept

These computation steps were considered in order to perform well:

1. At first find all files in the given start paths, this can be run in parallel for different start paths and subfolders
2. The second step was to group the found files by filesize
3. Removing those entries where only one file exists
4. Finding duplicates in each group can be run in parallel tasks
5. In a group of files, the first file will be checked against all other files in its group.
6. If a duplicate is found, say file2 is duplicate of file1, it will be removed form the list.
7. The reason is that it doesnt need to check if file2 is a duplicate on file3 because
8. if this would be the case then file1 is also a duplicate of file3 and this will be found when file1 and file3 are compared together.
9. The basic concept on comparing files is to calculate a blockwise MurMur3 hash and increasing the block size if the blocks are equal.
10. If duplicates are found they will be put into a new group. Then the next file will be checked.
11. When a file is processed and no duplicate is found, it will be left out.
12. It can occur that in a group of one file size, two or more groups of duplicates can exist.

1.3 Comparing files

Starting with a single byte of data from each file from the start and the end of the file. This may help to find early non duplicates in e.g. xml files or where the header looks the same for each file. The output is then a byte array with length 16. These two arrays will the be compare on each index. If one index from array1 is not equal to the same index of array2 this verifies that those files are not duplicates to eachother. When the first byte of data is equal, then a block size of 2 of data starting from the next index of data because the previously checked block of data doesn´t need to be checked again. After all files are processed and checked the file readers are disposed (with the use of the dispose pattern). This should close the opened streams of data and free the allocated memory.

Comparing two files

```
private bool Compare(FileReader cur, FileReader other) {
var equal = true;
for (var index = 0; equal; index++) {
	var itemCur = cur.ReadSection(index);
	var itemOther = other.ReadSection(index);
	if (itemCur == null || itemOther == null) {
		break;
	}
	equal = itemCur.Equals(itemOther);
	}
	return equal;
}
```

Reading blocks from a file

```
public FileItem ReadSection(int index) {
	if (_stream == null) {
		return null;
	}
	if (_readHashes.TryGetValue(index, out var result)) {
		return result;
	}
	var fileItem = new FileItem();
	var startIndex = 0;
	var bytecount = 0;

	bytecount = (int)Math.Pow(2, index);
	if (index > 0) {
		startIndex = (int)Math.Pow(2, index - 1);
	}
	if (FileSize / 2 < startIndex) {
		return null;
	}

	if ((FileSize / 2 + 0.5f) < (startIndex + bytecount)) {
		bytecount = (int)(Math.Ceiling((FileSize / 2.0) + 0.5f) - startIndex);
	}

	//Workaround for filesize 0
	if (FileSize == 0) {
		startIndex = 0;
		bytecount = 0;
	}

	byte[] bytes = new byte[bytecount];
	_stream.Position = startIndex;
	_stream.Read(bytes, 0, bytecount);
	fileItem.Front = Helper.GetMurMurHash(bytes);

	byte[] bytes2 = new byte[bytecount];
	_stream.Position = (FileSize - startIndex - bytecount);
	_stream.Read(bytes2, 0, bytecount);
	fileItem.Back = Helper.GetMurMurHash(bytes2);

	_readHashes.Add(index, fileItem);
	return fileItem;
}
```

If the file is a duplicate to another file then a hash of the whole file is calculated and it can be safely assumed that those files are duplicates.

It still can happen that two different files create the same hash. If this case should be considered, then a hash algoritm with more than 128bit is needed.

1.4 Testing

The first test was to check how fast all files from a given start path can be found.

Commandline call: ./searchDub -s "C:\Users" -w -p -v

Output can be found here:

Calculation from Windows can be found here:

It can be noticed that the file count from our application is not the same as from Windows. We assume that Windows may not count some system files and during the two calls to count files, temporary files can be created and deleted. Although 14k files is still a very big difference for those explanations.

A complete scan over C:\Users can be found here!

1.5 Authors

- Mike Thomas
- Andreas Reschenhofer

See also the list of contributors who participated in this project.

1.6 License

No license information