Lab 4

Due Mar 8 by 12:30pm **Points** 100 **Submitting** a file upload **File Types** zip
Available Feb 20 at 1:30pm - Mar 8 at 12:30pm 16 days

Purpose

You will be programming more advanced topics in C#. You will be using asynchronous functions using the await and async keywords using TAP for parallel programming. You will also use the Parallel class to execute loop iterations in parallel. Lastly, you will learn how to develop partial classes split among many files.

Assignment

You will be manually implementing the MapReduce class for a generic set of data. You must create a single class called MapReduce that takes a generic data type and is split among three different files: Map.cs, Reduce.cs, and MapReduce.cs. All code needed to construct and add data to memory will be contained in MapReduce.cs. All code needed to execute the Map function will be contained in Map.cs. All code needed to execute the Reduce function will be contained in Reduce.cs.

The Map function will execute a function for each individual element of the set of data. No data will influence another, so therefore, you can execute the Map() function completely in parallel. Since the developer supplies a function to your Map function, you must write a delegate, which will then be the parameter supplied to your Map function. Since Map is a 1-to-1 function, the delegate will take a single parameter and return the modified parameter of the same data type. Remember, Map is in a generic class, so the data type is specified in <>, such as MapReduce<double>.

The Reduce function will execute a function that reduces the entire data set into a single value. For example, calculating the average of a set of data or getting the product of a data set. Since the Reduce function requires data across the data set, it is executed serially. However, in your C# program, you will be using the TAP asynchronous pattern to allow the developers program to continue while your ReduceAsync function executes in the background. You must use the TAP pattern to solve this function. Since Reduce and ReduceAsync take a function to perform on the data set, you must create a delegate. This delegate will take two input parameters of the generic data type (accumulator and current element) and return a single value of the generic data type.

You will need to create a class that reads a set of data from a binary file. Call this class DataReader and put it in DataReader.cs. Your constructor will take a filename as a string and it will read the given filename. If the file cannot be found, throw the FileNotFoundException. DataReader will read only double data types, so DataReader will only work with MapReduce<double>. The binary file is simply a list of 8-byte, double values one after another without padding and in little-endian format. You must use an array (double[]) to store the

values read from a file. Since arrays are not resizable, you must read the entire file to count the number of values to be stored. Then, create the memory for the array, and finally, read the file into the array.

MapReduce.cs (1/3 for MapReduce class):

1. Must be a generic class (has a "template" parameter MapReduce<*type*>)

2. Must contain a generic container, such as a List to store the given data.

3. Must contain an Add function which takes a generic parameter and adds it to the generic container.

4. Must contain the 0-argument constructor, which simply creates the new generic container.

5. Must contain a property called Count which returns the number of elements added through the Add function.

6. Must contain an indexer that returns the given element in your list.

Map.cs (2/3 for MapReduce class):

1. Must contain a delegate for a map function. The map function takes a single generic parameter and returns the same data type.

2. Must contain the Map function. The Map function will take as a parameter the delegate from #1. The Map function must use the Parallel class to execute the delegate map function in parallel.

Reduce.cs (3/3 for MapReduce class):

1. Must contain a delegate for a reduce function. The reduce function takes two generic parameters (left and right), executes the given operation and returns a single generic data.

2. Must contain the Reduce function. The Reduce function will take as a parameter the delegate from #1. The reduce function will serially, and synchronously loop through the internal data and execute the function given by the delegate parameter. Notice that the Reduce function does not modify any of the internal data.

3. Must contain the ReduceAsync function. The ReduceAsync must use the TAP pattern and return a value of the generic data type. This function will execute the same Reduce function in #2, except it will do it asynchronously using the async and await keywords in C#.

DataReader.cs:

1. Must contain an internal array of doubles.

2. Must contain a constructor that takes a filename as a string parameter. The constructor will open the file and read all of the data into the internal array. If the file does not exist, throw FileNotFoundException. If you read the file and there are no doubles to read, throw EndOfStreamException.

3. Must implement IEnumerable. IEnumerable must have the GetEnumerator() function implement, and it must return a DataReaderEnum class.

4. Must contain a property called Count, which is readonly and returns the number of elements in your array of doubles.

5. Must contain an indexer that returns the data at the given index into the array of doubles. myclass[i] should return doubles_array[i]. Check the bounds of i before returning the data. If i is outside the bounds of the array, throw IndexOutOfRangeException.

DataReaderEnum.cs:

1. Must implement the IEnumerator interface.

2. The constructor will take an array of doubles, which is a reference to the array of doubles in the DataReader class.

3. IEnumerator requires that MoveNext(), Reset(), and the property Current be implemented.

4. MoveNext() will increment the internal iterator (best to use an integer, which will be the index into the array of doubles).

5. Reset() will set the internal iterator back to the default state. Remember that MoveNext() is called after the iterator is created, so the internal iterator must be set to -1 in its default state.

6. The Current property returns the value of the internal iterator and is read-only.

Asynchronous Programming

Asynchronous programming, aka concurrent programming, aka parallel programming, is a technique where multiple lines of code run at the same time. This introduces some challenges, especially when you need the result of an asynchronous operation before your program can continue.

There are three different asynchronous ways to program in C#: Asynchronous Programming Model (APM), Event-based Asynchronous Programming (EAP), and Task-based Asynchronous Pattern (TAP). You must use the TAP pattern for this lab.

Sample Code

The file **test.10000** provides 10,000 randomly generated doubles between the values 0.0 and 1.0 in binary format, 8 bytes at a time. If you execute the sample code below with this file, the result should be "Reduce returned value: 1.423...".

The following might be used in driver code to test your program:

```
using System;
using System.IO;

class lab4test
{
    public static int Main(string[] args) {
        if (args.Length < 1) {
            Console.WriteLine($"Usage: {AppDomain.CurrentDomain.FriendlyName} <filename>");
            return -1;
        }
        DataReader dr;
        try {
            dr = new DataReader(args[0]);
        }
        catch (FileNotFoundException) {
            Console.WriteLine($"{args[0]}: File not found");
            return -2;
        }
        Console.WriteLine($"Done reading {args[0]}, {dr.Count} records read.");
        var mr = new MapReduce<double>();
        foreach (double data in dr) {
            mr.Add(data);
        }
        mr.Map(i => { return (i+i) / 2.0; });
        var rtask = mr.ReduceAsync((accum, cur) => {
            return (accum + cur) / Math.Sqrt(accum + cur);
        });
        ulong iters = 0;
        while (rtask.IsCompleted == false)
             iters++;
        rtask.Wait();
        Console.WriteLine($"Reduce returned value: {rtask.Result:F3}. Reduce took {iters} iteration
s.");
        return 0;
    }
}
```

Submission

Zip all of your C# source files as lab4.zip. Do NOT include an entry point. This will be provided in the driver code.

WARNING: In order for your code to be graded, it must at least compile using the mcs compiler. If the TA (or I) cannot compile your code, you will not receive credit for it! If you cannot finish a certain function, make sure it does at least something to pass the compiler stage.

Submit lab4.zip on Canvas.

File Upload | Google Doc

Upload a file, or choose a file you've already uploaded.

File: Choose File No file chosen

+ Add Another File

Click here to find a file you've already uploaded

Comments...

Cancel Submit Assignment