Answers to Exercises

2. Parallel Loops

Exercises

1. Which of the following problems could be solved using the parallel loop techniques taught in this chapter?

    a. Sorting an in-memory array of numbers with a million elements.
    No, because the array cannot be divided into parts that can be sorted independently. See chapter 6 for an example of sorting in parallel using dynamic tasks.

    b. Putting the words in each line read from a text file in alphabetical order.
    Yes, because each line can be processed independently.

    c. Adding together all the numbers in one collection, to obtain a single sum.
    No, because the sum of the entire collection is needed, not the sums of separate parts. See chapter 4 for an example of summing collections in parallel using aggregation.

    d. Adding numbers from two collections pair-wise, to obtain a collection of sums.
    Yes, because all the pairwise sums are independent of each other.

    e. Counting the total number of occurrences of each word in a collection of text files.
    No, because a single count for all the files is needed, not counts for separate files.

    f. Finding the word that occurs most frequently in each file in a collection of text files.
    Yes, because each file can be considered independently, to find its most frequent word.

2. Choose a suitable problem from Exercise 1. Code three solutions, using a sequential loop, a parallel loop, and PLINQ.
This is a project, no answer is provided.

3. In the credit review example, what is the type of account?
   *Accounts are represented in the sample by instanced of the **Account** class.*

   What is the type of **accounts.AllAccounts**?
   *The AllAccounts method returns an enumerable of type Account, **IEnumerable<Account>.***

   What is the type of **accounts.AllAccounts.AsParallel()**?
   *The AsParallel method returns a **ParallelQuery<Account>**.*

4. Is it possible to use a PLINQ query as the source for a **Parallel.ForEach** loop? Is this recommended? Explain your answers.
   *Yes, it is possible because a PLINQ query belongs to the type **ParallelQuery<T>**, which implements **IEnumerable<T>**. However it is not recommended because it is inefficient: the output of the **ParallelQuery** is merged into one **IEnumerable**, and then partitioned again into **Parallel.ForEach**. The recommended alternative is to use PLINQ's **ParallelEnumerable.ForAll** method instead.*

5. Do a performance analysis of the credit view example code on the CodePlex site http://parallelpatterns.codeplex.com. Use command line options to independently vary the number of iterations (the number of accounts) and the amount of work done in the body of each iteration (the number of months in the credit history). Record the execution times reported by the program for all three versions, using several different combinations of numbers of accounts and months. Repeat the tests on different computers with different numbers of cores and with different execution loads (from other applications).
   This is a project, no answer is provided.

6. Modify the credit review example from CodePlex so that you can set the **MaxDegreeOfParallelism** property. Observe the effect of different values of this property on the execution time when running on computers with different numbers of cores.
   To use MaxDegreeOfParallelism, revise the code in UpdatePredictionsParallel as follows:

   ```
   var options = new ParallelOptions() { MaxDegreeOfParallelism = 2 }; // 2 or 4 or …
   Parallel.ForEach(accounts.AllAccounts, options, account => …
   ```

3. Parallel Tasks

Exercises

1. The image blender example in this chapter uses task parallelism: a different task processes each image layer. A typical strategy in image processing uses data parallelism: the same

computation processes different portions of an image, or different images. Is there a way to use data parallelism in the image blender example? If so, what are the advantages and disadvantages, compared to the task parallelism discussed here?
*Data parallelism would be easy if there were many images to process – simply make the entire body of **SequentialImageProcessing** the body of a **Parallel.ForEach** loop that iterates over pairs of images. This would have the advantage of easily scaling to more than two tasks.*

*Achieving data parallelism by subdividing the images would be more difficult. For example, the **Rotate** method uses the .NET **RotateFlip** method which uses the entire image; it cannot be applied independently to portions of the image. It is possible to use data parallelism for image rotation but would require writing new code, incurring the disadvantages of more complicated code and more code development.*

2. In the image blender sample, the image processing methods **SetToGray** and **Rotate** are void methods that do not return results, but save their results by updating their second argument. Why do they not return their results?
*Because the arguments to **Parallel.Invoke** are of type **Action**, which does not return results (not **Func**, which does return results).*

3. In the image blender sample that uses **Task.Factory.StartNew**, what happens if one of the parallel tasks throws an exception? In the sample that uses **Parallel.Invoke**?
*In the both cases, the runtime catches the exception, packages it as an inner exception in an aggregate exception, and rethrows the aggregate exception to the caller when the other task finishes. In the first case, this occurs when the caller is waiting at **Task.WaitAll**, and in the second case, when the caller is executing **Parallel.Invoke**.*

4 Parallel Aggregation

Exercises

1. Consider the small social network example (with subscribers 0, 1, 2). What constraints exist in the data? How are these constraints observed in the sample code?

   Constraints:

   The friends relation is symmetric; if x is a friend of y, then y is a friend of x.
   The friends relation is not reflexive: x is not a friend of x.

   *In **SubscriberRepository**, the **AssignRandomFriends** method enforces both constraints in the last if-statement block. **PotentialFriendsSequential** and **PotentialFriendsParallel** enforce them in the calls to the **RemoveWhere** method, and –Linq and –Plinq enforce them in the calls to **Where**.*

2. In the social network example, there is a separate post processing step where the multiset of candidates, which is an unordered collection, is transformed into a sequence that is sorted by the number of mutual friends, and then the top N candidates are selected. Could some or all of this post processing be incorporated into the reduction step? Provide answers for both the PLINQ and **Parallel.ForEach** versions.
   The combining operation in the reduction step must be associative, which means the result must not depend on the order in which the pairwise combinations are formed. So this question becomes, is there an associative combining operation which achieves the effect of the post processing? Yes. That operation is applied to pairs of collections: merge the two collections and retain the top N elements (or all elements if there are fewer than N). This could be incorporated into the reduce phase of both versions. It might result in better performance than the sequential post processing step shown in the examples.

3. In the standard reference on map/reduce (see the "Further Reading" section), the map phase executes a map function that takes an input pair and produces a set of intermediate key/value pairs. All pairs for the same intermediate key are passed to the reduce phase. That reduce phase executes a reduce function that merges all the values for the same intermediate key to a possibly smaller set of values. The signatures of these functions can be expressed as: map (k1,v1) -> list(k2,v2) and reduce (k2,list(v2)) -> list(v2). In the social network example, what are the types of k1, v1, k2, and v2? What are the map and reduce functions? Provide answers for both the PLINQ and **Parallel.ForEach** versions.
   In both versions, k1 is the subject, v1 is a friend of k1. k2 is the friend of v1, and v2 is the number of mutual friends that k2 has in common with v1. In the output of the map stage, list(k2, v2) is a collection where each k2 may occur multiple times, each time paired with a v2 value of 1. In the reduce stage, the v2 for each k2 are added up to find the total number of mutual friends, and the friend k2 is retained in the result, so in this example the signature is: reduce(k2,list(v2)) -> list(k2,sum(v2)).

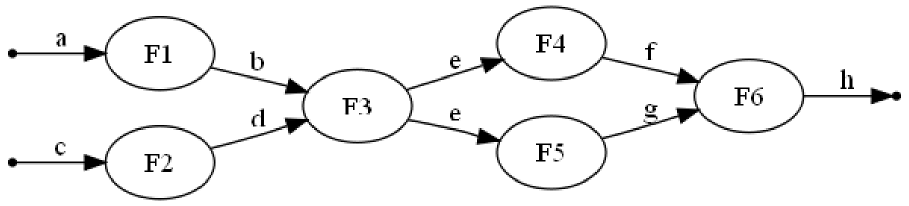
5. Futures

Exercises

1. Suppose you parallelize the following sequential code using futures in the style of the first example in the section, "The Basics."

```
var b = F1(a);  var d  = F2(c);  var e = F3(b,d);  var f =
F4(e);  var g = F5(e); var h = F6(f,g);
```

   Draw the task graph. In order to achieve the greatest degree of concurrency, what is the minimum number of futures you must define? What is the largest number of these futures that can be running at the same time?



You must define at least two futures (For F1 and F4, for example). At most one of the futures can be running at the same time as the main task, which runs the remaining operations; F2, F3, F5 and F6.You could add additional futures but the dependencies between the different operations limit the number of parallel operations to two.

2. Modify the BasicFutures sample from CodePlex so that one of the futures throws an exception. What should happen? Observe the behavior when you execute the modified sample.
This is a project, no answer given.

6. Dynamic Task Parallelism

Exercises

1. The sample code on Codeplex assigns a particular default value for the **Threshold** segment length. At this point, both the serial and parallel the QuickSort methods switch to the non-recursive **InsertionSort** algorithm. Use the command line argument to assign different values for the **Threshold** value, and observe the execution times for the sequential version to sort different array sizes. What do you expect to see? What is the best value for **Threshold** on your system?
This is a project, no answer given.
2. Use the command line argument to vary the array size, and observe the execution time as a function of array size for the sequential and parallel versions. What do you expect? Can you explain your observations?
This is a project, no answer given.
3. Suggest other measures to limit the number of tasks, besides the number of cores.
*The size of the segment in the array that is passed to the sort call (just as with the threshold for the non-recursive **InsertionSort** method). The number of tasks (that is, the number of **Parallel.Invoke** calls).*

7. Pipelines

Exercises

1. Write your own pipeline by modifying the example shown in the first section of this chapter.
   Project, no answer given.
2. Execute the code with the Concurrency Visualizer. View and interpret the results.
   Project, no answer given.