In cases where a large number of threads are used and are available for the reduction computation, an alternative implementation would be to divide the elements of the global array holding the partial results generated by each thread among four or eight threads. These threads would divide up the partial sum elements, compute a reduction on the assigned chunk, and then allow one thread to do the final reduction on these results in serial. The code for this suggested algorithm isn't as simple as the one using barriers, but it could be more efficient.

Applying MapReduce

I want to give you an idea about how to determine when MapReduce might be a potential solution to a concurrent programming problem. The task we're going to carry out here is finding all pairs of natural numbers that are mutually *friendly* within the range of positive integers provided to the program at the start of execution (this computation was part of the problem posed during the Intel Threading Challenge contest in July 2008). Two numbers are mutually friendly if the ratio of the sum of all divisors of the number and the number itself is equal to the corresponding ratio of the other number. This ratio is known as the *abundancy* of a number. For example, 30 and 140 are friendly, since the abundancy for these two numbers is equal (see Figure 7-2).

$$\frac{1+2+3+5+6+10+15+30}{30} = \frac{72}{30} = \frac{12}{5}$$

$$\frac{1+2+4+5+7+10+14+20+28+35+70+140}{140} = \frac{336}{140} = \frac{12}{5}$$

FIGURE 7-2. Friendly numbers

The serial algorithm for solving this problem is readily evident from the calculations shown in Figure 7-2. For each (positive) integer in the range, find all the divisors of the number, add them together, and then find the irreducible fractional representation of the ratio of this sum and the original number. After computing all the ratios, compare all pairs of ratios and print out a message of the friendly property found between any two numbers with matching ratios.

To decide whether this problem will fit into the MapReduce mold, you can ask yourself a few questions about the algorithm. Does the algorithm break down into two separate phases? Will the first phase have a data decomposition computation? Are those first phase computations independent of each other? Is there some "mapping" of data to keys involved? Can you "reduce" the results of the first phase to compute the final answer(s)?

This is a two-part computation. We can think of the first phase as a data decomposition of the range of numbers to be investigated, and there is a natural mapping of each number to its abundancy ratio. Factoring a number to compute the divisors of that number is independent

of the factorization of any other number within the range. Thus, this first phase looks like a good candidate for a map operation.

As for the reduce phase, each number-abundancy pair generated in the map phase is compared with all other pairs to find those with matching abundancy values. If a match is found within the input range of numbers, that match will be noted with an output message. There may be no matches, there may be only one match, or there may be multiple matches found. While this computation doesn't conform to the typical reduction operations where a large number of values are summarized by a single result, we can still classify this as a reduction operation. It takes a large collection of data and "reduces" the set by pulling out those elements that conform to a given property (e.g., the earlier document search application that finds the smaller set of pages containing keywords or phrases).

Thus, the serial algorithm for identifying mutually friendly pairs of integers within a given range can be converted to a concurrent solution through a MapReduce transformation. The code for an OpenMP implementation of this concurrent solution is given in Example 7-3.

EXAMPLE 7-3. MapReduce solution to finding friendly numbers

```
int gcd(int u, int v)
{
  if (v == 0) return u;
  return gcd(v, u % v);
}

void FriendlyNumbers (int start, int end)
{
  int last = end-start+1;
  int *the_num = new int[last];
  int *num = new int[last];
  int *den = new int[last];

#pragma omp parallel
 {int i, j, factor, ii, sum, done, n;
//  -- MAP --
#pragma omp for schedule (dynamic, 16)
  for (i = start; i <= end; i++) {
    ii = i - start;
    sum = 1 + i;
    the_num[ii] = i;
    done = i;
    factor = 2;
    while (factor < done) {
      if ((i % factor) == 0) {
        sum += (factor + (i/factor));
        if ((done = i/factor) == factor) sum -= factor;
      }
      factor++;
    }
    num[ii] = sum; den[ii] = i;
    n = gcd(num[ii], den[ii]);
    num[ii] /= n;
```

```
    den[ii] /= n;
  }  // end for

//  -- REDUCE --
#pragma omp for schedule (static, 8)
  for (i = 0; i < last; i++) {
    for (j = i+1; j < last; j++) {
      if ((num[i] == num[j]) && (den[i] == den[j]))
        printf ("%d and %d are FRIENDLY \n", the_num[i], the_num[j]);
    }
  }
 }  // end parallel region
}
```

Ignore the OpenMP pragmas for the moment while I describe the underlying serial code. The `FriendlyNumbers()` function takes two integers that define the range to be searched: `start` and `end`. We can assume that error checking before calling this function ensures that `start` is less than `end` and that both are positive numbers. The code first computes the length of the range (`last`) and allocates memory to hold the numbers within the range (`the_num`). It also allocates memory space for the numerator (`num`) and denominator (`den`) of the abundancy ratio for each number. (We don't want to use the floating point value of the abundancy since we can't guarantee that two ratios, such as 72.0/30.0 and 336.0/140.0, will yield the exact same `float` value.)

The first `for` loop iterates over the numbers in the range of interest. In each iteration, the code computes the offset into the allocated arrays (`ii`), saves the number to be factored, and finds the divisors of that number and adds them together (`sum`). The internal `while` loop finds the divisors of the number by a brute force method. Whenever it finds a factor, it adds that factor (`factor`) and the associated multiplicand (`i/factor`) to the running `sum`. The conditional test makes sure that the integral square root factor is not added in twice. The `done` variable is the largest value that potential divisors (`factor`) can be. This value is set to `i/factor` whenever a factor is found, since there can be no other divisors greater than the one associated with the factor value in `factor`.

After summing all the divisors of a number, the code stores `sum` in the appropriate numerator slot (`num`), and stores the number itself in the corresponding denominator slot (`den`). The `gcd()` function computes the greatest common divisor (GCD) for these two numbers (via the recursive Euclidean algorithm) and divides each by the GCD (stored in `n`) to put the ratio of the two into lowest terms. As noted in the comments, the factoring and ratio computations will be the map phase.

The nested `for` loops that follow compare the numerators and denominators between unique pairs of numbers within the original range. To ensure only unique pairs of ratios are compared, the inner `j` loop accesses the numerator and denominator arrays from the `i+1` position to the `last`.

If the [i] indexed numerator and denominator values match the [j] indexed numerator and denominator values, a friendly pair is identified and the two numbers stored in the_num[i] and the_num[j] are printed with a message about their relationship. This is the reduce phase.

The code in Example 7-3 uses OpenMP pragmas to implement concurrency. The code includes a parallel region around both the map and reduce portions. Within this region are the declarations of the local variables i, j, factor, ii, sum, done, and n.

The for loop of the map phase is located within an OpenMP loop worksharing construct to divide iterations of the loop among the threads. Notice that I've added a schedule clause to the pragma. I've specified a dynamic schedule, since the amount of computation needed to find divisors of numbers will vary widely, depending on the number itself. Within the inner while loop, the number of factors that must be considered will be smaller for a composite number than a prime of similar magnitude (e.g., 30 and 31). Also, larger numbers will take more time than smaller numbers, since there will be more factors to test and, likely, there will be more divisors of the larger number (e.g., 30 and 140). Hence, to balance the load assigned to threads, I've elected to use a dynamic schedule with a chunk size. Threads that are assigned subrange chunks that can be computed quickly will be able to request a new chunk to work on, while threads needing more time to continue with the assigned subrange chunk will continue factoring.

For the reduce phase implementation, another loop worksharing construct is placed on the outer for loop (Simple Rule 2). As with the worksharing construct in the mapping code, a schedule clause has been added to yield a more load balanced execution. In this case, I've used a static schedule. You should realize that the number of inner loop iterations is different for every iteration of the outer loop. However, unlike the inner loop (while) within the map phase, the amount of work per outer loop iteration is monotonically decreasing and predictable. The typical default for an OpenMP worksharing construct without the schedule clause is to divide the iterations into a number of similar-sized chunks equal to the number of threads within the OpenMP team. In this case, such a schedule would assign much more work to the first chunk than to subsequent chunks.

I visualize such a default static division of iterations like the triangle shown in Figure 7-3, where the vertical axis is the outer loop iterations, the horizontal axis is the inner loop, and the width of the triangle represents the number of inner loop iterations executed. The area of the triangle associated with a thread is in direct proportion to the amount of work that the thread is assigned. The four different shades of gray represent a different thread to which the chunk of work has been assigned.

FIGURE 7-3. Two static distributions of monotonically decreasing amounts of work among four threads

The more equitable division of area (i.e., work) among threads is shown in Figure 7-3 (b), which includes smaller and more numerous chunks. Those chunks are assigned to each thread in a round-robin fashion. The sum of the areas is then much more equal between threads and, thus, the load will be better balanced. You can accomplish the division of work shown in Figure 7-3 (b) by using the `schedule (static, 8)` clause given in Example 7-3. While a dynamic schedule may give a tighter overall load balance, the overhead associated with distributing chunks of iterations to threads might be more adverse than simply using the "good enough" static schedule.

There is no magic reason for using the chunk sizes that we used here. When using an OpenMP schedule clause (or implementing such behavior using an explicit threading library), test several different values to see whether there are significant performance differences. Choose the one that gives the better performance in the majority of potential input data set cases. Keep in mind the size of a cache line and choose a chunk size that will allow full cache lines to be used by a single thread whenever possible, especially when updates are required.

Friendly Numbers Example Summary

In this section, I've shown how you can apply the MapReduce framework to a serial code in order to find a concurrent equivalent. While the reduce phase of the friendly numbers problem might seem atypical, you need to be prepared to see past the standard many-to-one reduction case in order to be better equipped to translate serial codes to a MapReduce solution.

MapReduce As Generic Concurrency

I think the biggest reason that the MapReduce framework has gotten such a large amount of notoriety is that it can be handled in such a way that the programmer need not know much about concurrent programming. You can write a MapReduce "engine" to execute concurrently when it is given the specifications on how the mapping operation is applied to individual elements, how the reduction operation is applied to individual elements, and how the reduction operation handles pairs of elements. For the programmer, these are simply serial functions (dealing with one or two objects). The engine would take care of dividing up the computations among concurrent threads. The TBB `parallel_reduce` algorithm is an example of this, where the `operator()` code would contain the map phase computation over a subrange of items and the `join` method would implement the reduce phase.

This is the reason that I recommend structuring your map and reduce phases in such a way that you can apply the reduction computations to individual items and partial results of previous reductions. For example, in finding the maximum value from a data set, the definition of the reduction operation is to simply compare two items and return the value of the largest. Such code would work whether it was being applied to pairs of elements from the original collection or from partial results that had used this code to whittle down the original set into fewer partial results. If the MapReduce engine only has to deal with the details of dividing up the data and recombining partial results, the programmer simply supplies the comparison function to the engine. The limitations of a generic MapReduce engine might preclude the use of such a system if the reduction computation were more complex, such as the reduction computation used in the friendly numbers problem.

I predict that providing generic concurrency engines and algorithms that allow programmers to write only serial code or require a minimum of concurrency knowledge will become popular in the coming years. This will allow programmers who do not have the training or skills in concurrent programming to take advantage of multicore and manycore processors now and in the future. Of course, until we get to the point where we can program by describing our problem or algorithm in English (like in countless episodes of *Star Trek*), someone has to understand concurrent programming to build such engines, which could be you.