/Theory/In/Practice

The Art of Concurrency

A Thread Monkey's Guide to Writing Parallel Applications

O'REILLY®

Clay Breshears

The Art of Concurrency

Clay Breshears

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

The Art of Concurrency
by Clay Breshears

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Sarah Schneider
Copyeditor: Amy Thomson
Proofreader: Sarah Schneider

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
May 2009: First Edition.

ISBN: 978-0-596-52153-0

[V]

1241201585

To my parents, for all their love, guidance,
and support.

CONTENTS

CHAPTER SEVEN

MapReduce

MapReduce is an algorithmic framework, like divide-and-conquer or backtracking, rather than a specific algorithm. The pair of operations, *map* and *reduce*, is found in LISP and other functional languages. MapReduce has been getting a lot of buzz as an algorithmic framework that can be executed concurrently. Google has made its fortune on the application of MapReduce within a distributed network of thousands of servers (see "MapReduce: Simplified Data Processing on Large Clusters" in *Communications of the ACM* [2008] by Jeffrey Dean and Sanjay Ghemawat), which has only served to heighten awareness and exploration of this method.

The idea behind *map* is to take a collection of data items and associate a value with each item in the collection. That is, to match up the elements of the input data with some relevant value to produce a collection of key-value pairs. The number of results from a map operation should be equal to the number of input data items within the original collection. In terms of concurrency, the operation of pairing up keys and values should be completely independent for each element in the collection.

The *reduce* operation takes all the pairs resulting from the map operation and does a reduction computation on the collection. As I've said before, the purpose of a reduction is to take in a collection of data items and return a value derived from those items. Parallel sum (from Chapter 6) is an example of a reduction computation. In more general terms, we can allow the reduce operation to return with zero, one, or any number of results. This will all depend on what the reduction operation is computing and the input data from the map operation.

Before looking at the implementation and other details, let's look at an example of MapReduce in action. Consider the task of counting the number of vowels and consonants in the following sentence:

The quick brown fox jumps over the lazy dog.

In my head, I run through the sentence, character by character, once to count the consonants and then again to count the vowels. The results would then be two integers: the number of vowels (12) and the number of consonants (23). Did you only count 11 vowels? Since "lazy" has two syllables, which requires two vowel sounds, there must be at least two vowels. Thus, "y" is doing duty as a vowel in this sentence. Example 7-1 has pseudocode of the MapReduce operation that would compute these values. For this example, `S` is the string of characters (array) holding the sentence.

EXAMPLE 7-1. MapReduce pseudocode example

```
// Map
  for i = 1 to length(S) {
    if (S[i] is a consonant)
      generate_pair(key[i]=S[i], value[i]=1);
    else
      generate_pair(key[i]=S[i], value[i]=-1);
  }
```

```
// Reduce
  cCount = 0; vCount = 0;
  for i = 1 to length(S) {
    if (value[i] > 0) cCount += value[i];
    else              vCount += abs(value[i]);
  }
```

For the given data set, `S`, the MapReduce solution first maps a value to each letter in order to create 35 key-value (letter-integer) pairs. The choice of values depends on whether the letter is a consonant (`1`) or a vowel (`-1`). The reduction operation will take each key-value pair and add the value into one of two counters based on the type of letter contained in the key.

> NOTE
>
> **We will need some way to identify "y" as a vowel. This identification will need to influence the key-value pair created in the map phase so that the reduce computation can add the associated value to the correct counter.**

If we wrote code for this letter-counting operation in serial, we could have simply examined each letter in turn and incremented the proper counter. In the (serial) MapReduce variation (Example 7-1), we're required by the framework to use the values associated with the keys in the reduction computation. While it may seem to be more work, this difference creates the situation where the data is divorced from the algorithm and makes the reduction computations more independent.

Map As a Concurrent Operation

Look back at the pseudocode of the vowel/consonant counting algorithm in Example 7-1. Can you see that the creation of each key-value pair in the map phase would be independent of every other pair creation? Simply divide the letters among threads and create key-value pairs for each letter. One goal I would urge you to keep in mind when using MapReduce is to make the reduction computation as simple as possible. This is why the algorithm in Example 7-1 decided the category of each letter in the map phase.

An alternate mapping would assign a value of 1 to each key (character) and let the reduce phase decide whether the key is a vowel or consonant. The context of the keys that exists in the map phase may not be available when the reduction computation is executed. Or, if not unavailable, it may require extra data to preserve the context and correctly process values in the reduction. Determining whether to label "y" as a consonant or a vowel requires the context of the word itself. If the map and reduce operations were in different functions and we used the alternate mapping of `1` for all keys, we would need to send the reduce function the context (the sentence) in order to classify the "y" properly.

Just to hammer one more nail into this idea, consider the classic MapReduce example of finding pages from a document (or set of documents) that contain a key phrase or word of interest. Since we might not be able to find the exact phrase on any pages that we want to search, we can devise a search-rating scheme that could rank pages that might have some subset of words in our phrase. For example, pages containing the exact phrase will be given the highest rating, pages that contain a subphrase (subset of words from the original phrase in the same order and next to each other) will be given slightly lower ratings, and pages that have disjointed words from the phrase will be given even lower ratings. For the final results of this example, we could specify the output as a list of the 20 pages with the highest scores.

The mapping computation should create key-value pairs with a pointer to one page of the document(s) as the key and the search rating of that page as the value. The rating of each page with regard to the search phrase is completely independent of rating any other page. The reduce phase now simply selects the 20 pages with the highest scores.

Implementing a Concurrent Map

How do you implement the map phase for concurrent execution? I'm sorry to say that I can't tell you, because each application that uses MapReduce will likely be different. So, the details are going to be up to you and will be based on the computational needs of the code. However, I can give you some general guidelines.

Whenever you find MapReduce applicable, the mapping operation will always be a data decomposition algorithm. You will have to turn a collection of data into key-value pairs. The items from the collection of data may require some "massaging" to determine the consequent key value. Or the map might simply attach a key to the value, or vice versa if the data is to be used as a key.

Whatever processing needs to be done in the map phase, you must design it with the reduce phase in mind. By doing more work in the map phase, you lessen the amount of work needed in the reduction operation and make it easier overall to write and maintain the MapReduce algorithm. The best reduce is going to be a single operation that you can apply to both individual elements and partial reduction results (if the reduce algorithm uses them). Writing a two-stage reduction (e.g., deciding on the type of a letter and adding a value to the right counter) can needlessly complicate your concurrent implementation.

Since the mapping operation on individual elements is independent of the computation on any other data item (Simple Rule 1), there won't be any data races or other conflicts. Of course, there will always be the exception that will prove me wrong on this. If you find that synchronization is necessary to avoid a data race, reexamine the mapping computation to see whether you should handle that data race in the reduce phase. Also, you may find that MapReduce is not the best algorithm for the problem you are trying to parallelize.

Finally, be aware of load balancing issues in the map computations. If you have something as simple as attaching a count value to a data item key (e.g., the letter counting example), then

all of the individual map operations will take about the same amount of time, and you can easily divide them up with a static schedule. If you have a case where the computation time on individual elements will vary (e.g., finding key words and phrases within documents of different sizes), a more dynamic schedule of work to threads will be best.

Reduce As a Concurrent Operation

As I recommended in the previous chapter, if you have the chance, use either OpenMP or Intel TBB to do reduction computations. All of the grunt work and coordination of threads and partial values goes on "behind the scenes." Why would you want to do any heavy lifting if you don't have to? Plus, making use of code that is already written and debugged will give you more of a warm fuzzy feeling about your concurrent application.

If you don't have the option of using the reduction algorithms built into TBB or OpenMP, you will need an explicit threads solution. Example 6-4 includes a handcoded reduction for summing up all the items within an integer array. You can use this as a model for implementing reduce when there are few threads.

For this chapter, though, I want to present an alternative that doesn't use the serial processing at the final step to combine the partial results from the previous independent computations. Let's again take the summing of all elements from an array as the specific problem to be solved. As with the code in Example 6-4, we'll use Pthreads as the explicit threading library for implementation.

Handcoded Reduction

To get started, each thread working on the reduction is assigned a nonoverlapping chunk of the overall data set. Any of the previously discussed methods that ensures all of the data are assigned will work. A static division of data is best. Next, each thread calculates the sum of all the items assigned. This will yield a number of partial sums equal to the number of threads. At this point, we can use the PRAM version of the parallel sum algorithm presented in Example 6-2 (one thread per data item to be summed). However, since we probably don't have lockstep execution of threads built into the hardware, we will need to coordinate the threads' executions on the data and with each other.

As in Example 6-4, we will store the partial sums in separate elements of a global array. Threads use the assigned ID for the index into the global array `gSum`. We will compute the final sum concurrently using the global array data. The first element within the `gSum` array (index 0) will hold the computed total for the elements of the original array. Figure 7-1 shows a representation of the concurrent computations that we will need to take the partial sums and add them together for the final answer.

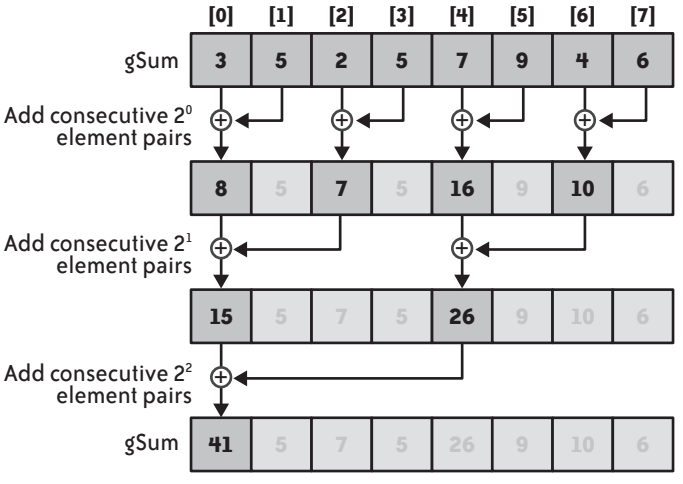

FIGURE 7-1. Reduction computation

Each array displayed in Figure 7-1 represents the contents of the `gSum` array after a round of computation. The arrows indicate where data is read from and where the results are stored in each round. The plus sign (+) within the circle is the combining operation (addition for this example) used in the reduction. The grayed numbers simply indicate that the data within that element of the array is no longer needed in further rounds of computation. For brevity, I've limited the example to eight threads. I think that once you've seen how two or three rounds are computed, you'll easily be able to extrapolate this algorithm to any number of threads.

In the first round (Add consecutive 2^0 element pairs), each thread whose ID is a multiple of 2 (2^1) reads the value stored in the element indexed by [thread ID] + 1 (2^0)—if such an element is part of the array—and adds this value to the value stored in the element indexed with the thread's ID. In the second round (Add consecutive 2^1 element pairs), each thread whose ID is a multiple of 4 (2^2) reads the value stored in the element indexed by [thread ID] + 2 (2^1)—if such an element is part of the array—and adds this value to the value stored in the element indexed with the thread's ID. In the third round (Add consecutive 2^2 element pairs), each thread whose ID is a multiple of 8 (2^3) reads the value stored in the element indexed by [thread ID] + 4 (2^2)—if such an element is part of the array—and adds this value to the value stored in the element indexed with the thread's ID, *und so veiter* (or "and so on" for my English readers). The pattern for each successive round simply repeats with the indexes involved growing by a factor of two.

The code to implement a reduction summation on an array of integers is given in Example 7-2. This code is modified from the code given in Example 6-4 with the differences highlighted in bold text.

EXAMPLE 7-2. Reduction code to sum elements of an array

```
#include <pthread.h>
#include <stdio.h>
#include "pth_barrier.h"

#define NUM_THREADS 128

int N;  // number of elements in array A
int *A;
int gSum[NUM_THREADS];  // global storage for partial results
pth_barrier_t B;

void *SumByReduction (void *pArg)
{
  int tNum = *((int *) pArg);
  int lSum = 0;
  int start, end, i;

  start = ((float)N/NUM_THREADS) * tNum;
  end =   ((float)N/NUM_THREADS) *(tNum+1);
  if (tNum == (NUM_THREADS-1)) end = N;
  for (i = start; i < end; i++)
    lSum += A[i];
  gSum[tNum] = lSum;

  pth_barrier(&B);
  int p2 = 2;
  for (i = 1; i <= NUM_THREADS; i *= 2) {
    if ((tNum % p2) == 0)
      if (tNum+i < NUM_THREADS) gSum[tNum] += gSum[tNum+i];
    p2 *= 2;
    pth_barrier(&B);
  }
  free(pArg);
}

int main(int argc, char* argv[])
{
  int j, sum = 0;
  pthread_t tHandles[NUM_THREADS];

  InitializeArray(A,&N);  // get values into A array; not shown
  pth_barrier_init(&B, NUM_THREADS);
  for (j = 0; j < NUM_THREADS; j++) {
    int *threadNum = new(int);
    *threadNum = j;
    pthread_create(&tHandles[j], NULL, SumByReduction, (void *)threadNum);
  }
// just wait for thread  with id=0 to terminate; others will follow
  pthread_join(&tHandles[0], NULL);
  printf("The sum of array elements is %d\n", gSum[0]);
  return 0;
}
```

The first thing to notice about the code in this example is the inclusion of another header file, `pth_barrier.h`, and a barrier object globally declared by `pth_barrier_t B`. A *barrier* is a synchronization that will pause threads at the barrier point in the code until all threads working in the computation have reached that same point. Once all threads have arrived, the barrier releases the threads to begin execution of the succeeding code. This is like a starting line at a race that doesn't allow the competitors to start until all racers have reached the start line; once all the racers are ready, they are allowed to begin the event. See "A Barrier Object Implementation" on page 134 for details about implementing such a barrier object with the POSIX threads library.

If you look at the `main()` function, you will see the initialization of the array, the initialization of the barrier object, and the creation of threads to execute the `SumByReduction()` function. The main thread then waits for only the first thread created. This thread will have an ID (`threadNum`) of 0, and the `pthread_t` value returned from `pthread_create()` is stored in the index 0 element of the `tHandles` array. Since the final result will be in the `gSum[0]` location, and this "0" thread computes that final value and stores it in the location, once the 0 thread has terminated, the final sum has been stored and is ready to be used.

The first half of the `SumByReduction()` function code is taken verbatim from the corresponding function found in Example 6-4. The second half (in the bold text) is the reduction of the partial sums generated by the threads into a single summation value. Before reduction of the partial sums can begin, all of the partial sums must be computed. Even though we could assume that an equal distribution of chunks of data array will be assigned to each thread, we cannot assume (Simple Rule 6) that all threads will finish at the exact same time to store the partial sum result in the proper `gSum` slot. While there could be more than enough resources to assign one thread to a core exclusively, there are myriads of other factors within the operating system or the processor hardware that can slow down or inhibit the computation of one or more threads. Thus, we need to place a barrier after the assignment of each thread's partial sum into the `gSum` array. After the last thread has reached the barrier, we know that all the partial sums have been stored and the computation can safely proceed to the reduction. The whole key to the reduction algorithm working correctly (and being able to prove that this is a correct algorithm) is the barrier synchronization.

The reduction computation mimics the combining of data in Figure 7-1. The `for` loop counts off the rounds, with the `i` variable serving as the offset into the `gSum` array from which a thread will read data during the current round. The loop variable is multiplied by 2 in each iteration.

The `p2` variable will be powers of 2 that are used to determine which threads need to read data from the `gSum` array and add that value to the value found in that thread's assigned `gSum` element. The outer `if-then` statement in the body of the loop makes this determination by dividing a thread's ID number by `p2` and allowing those threads that are evenly divisible by the current

p2 to proceed to the inner if-then statement. The inner conditional expression (tNum+i < NUM_THREADS) will ensure that the proposed element of the gSum array to be read actually exists within the array bounds and, if so, increments the value of the thread's gSum slot with the value accessed.

Regardless of whether or not a thread was allowed to participate in the addition operation, every thread multiplies the local copy of p2 by 2 and then waits at a barrier until all threads have completed whatever computation was allowed within the current round of the reduction. All threads will execute something within each round of the reduction, even though half as many are doing useful work in a given round than in the previous round. This may seem like a waste of resources. However, since all threads must meet up at the barrier, we keep each thread running and doing a minimal amount of work (doubling the value of p2 in each round) to keep in sync with those threads that are still doing constructive work.

NOTE

While I've never written one or seen an implementation of one, creating a barrier that could work with a different number of threads each time it was used sounds like such a complex and daunting task. I'm afraid that the execution of such a beast would have massive amounts of overhead—certainly much more than keeping some threads alive for a few microseconds past the time they are doing anything practical.

Once you've had a chance to digest the code in Example 7-2 and probably traced the concurrent execution of the code using the example given in Figure 7-1, you may be asking yourself if this will work with a number of threads that is not a power of 2. All the instructions seem to be predicated on powers of 2, but there may come a time when you can only use 14 or 57 threads for a reduction operation. Rest assured, the code does work for a number of threads that is not a power of 2.

To prove this to yourself, try tracing the algorithm with nine threads (#define NUM_THREADS 9). The gSum array will be indexed from [0] to [8] (visualize another element attached to the right of the gSum array shown in Figure 7-1). There will be four rounds to the reduction algorithm, where i will be assigned values 1, 2, 4, and 8. The first eight elements of the gSum array will be processed as shown in Figure 7-1 during the first three rounds of the algorithm. The ninth element (index [8]) will be unchanged, since all potential elements of gSum that would be read and used to add into the contents of that slot are not within the bounds of the array. At the fourth round, the 0 thread will read the contents of gSum[8] (0 + i = 8) and add that value to the contents of gSum[0]. You can reproduce this idea to any number of threads between 9 and 16 (or any other nonpower of 2) where the upper slots of gSum will be summed (via reduction) into the index [8] element during the first three rounds, and the fourth will bring the final total into gSum[0].

Speaking of odd numbers of threads, did you notice the use of the (float) cast in the computation of start and end in Example 7-2? Rather than hoping for a number of iterations

that will be evenly divisible by the number of threads, you can use this method to divide iterations more evenly than integer division. For example, suppose that you have a loop with 122,429 iterations to be divided among 16 threads. If you use integer division and an assignment statement for the last thread to use 122,429 for its `end` value, each thread would be assigned 7,651 iterations, except the last one, which would get 7,664 (if you don't have a calculator handy, just trust me on the arithmetic). When using floating-point division that truncates fractional parts when recast back to (`int`), 13 of the threads will be assigned 7,652 iterations and the other 3 get 7,651. If the time to compute one iteration is short, 13 extra iterations assigned to the last thread might not have much impact. If an iteration takes 30 seconds to compute, waiting six and a half minutes for one thread to finish is a serious load balance issue. Regardless of the time per iteration, for static scheduling of loop iterations, using (`float`) when computing the `start` and `end` bounds will always generate a better load balance between threads.

A Barrier Object Implementation

A barrier object can synchronize thread execution at a specific point within the code. Threads are blocked at a barrier until all threads have reached the barrier point, and then all threads are released. With this description, we can develop the code to implement a barrier object to be used in Pthreads codes.

A Pthreads condition variable will hold threads until they can be released. So, we need a condition variable and the associated mutex object. In addition, the barrier must know the total number of threads that are participating in the barrier and how many threads have arrived at the barrier. When the final thread has come to the barrier, use that thread to release all the other waiting threads and to reset the counters of the barrier for the next use.

In my initial implementations of the barrier object, since I was using the count of threads that have arrived as the condition to keep threads waiting, I had thought to have the last thread exiting the barrier do the reset of all the counters. By keeping the count of threads at the barrier equal to 0 until the last thread was ready to exit, I didn't take into account the chance of another thread reentering the barrier, acquiring the mutex, checking the conditional expression, and passing over the `pthread_cond_wait()`. When a thread external to the `while` loop usurps the mutex and isn't forced to wait as it should be, it is known as an *intercepted wait*. In this case, the intercepted wait of an external thread entering the barrier before all threads previously waiting on the barrier had left can lead to a deadlock.

Thus, I need a conditional expression for the condition variable that doesn't rely on the count of threads at the barrier. I've chosen to "color" each use of a barrier in a cyclic fashion. When threads enter a barrier, they must wait for all threads to enter the barrier while the barrier is the same color. When the final thread shows up, the color of the (future) barrier changes and the counter resets (for the next use). Threads that are signaled to wake up check the barrier color. If it is not the same color that they found when they entered, they know that the final

thread has arrived and they can now exit the barrier. The structure for the pth_barrier_t type is given here:

```
typedef struct {
  pthread_mutex_t m;
  pthread_cond_t c;
  int count, color, numThreads;
} pth_barrier_t:
```

Following the convention of other Pthreads synchronization objects, I have written an initializing function to set up a barrier. This function simply calls the initialization functions for the condition variable and mutex. The total number of threads that will always participate in each use of the barrier is sent as a parameter to the initialization function. This value sets the two integer counters within the object. The count is decremented as threads come into the barrier and will reach 0 when the final thread has arrived. The color will actually toggle between "0" and "not 0," but I've added a definition of RED to be used in the initialization, for some extra flair. The initialization code is given here:

```
#define RED 0

pth_barrier_init (pth_barrier_t *b, int numT)
{
  pthread_mutex_init(&b->m, NULL);
  pthread cond_init(&b->c, NULL);
  b->count = b->numThreads = numT;
  b->color = RED;
}
```

Upon entering the pth_barrier() function, a thread first gains control of the object's mutex and notes the current color of the barrier. The color is held in a variable (kolor) local to each thread entering the barrier function (since kolor is declared in a function called by a thread). The thread then determines whether it is the last to arrive. If not, it blocks itself on the condition variable (and releases the mutex). If the thread is the last to arrive, which it will know from the barrier's count being decremented to 0, it will reset the color of the barrier, set the count, and wake up all threads that have been waiting. The code for the pth_barrier() function is given here:

```
void pth_barrier (pth_barrier_t *b)
{
  pthread_mutex_lock(&b->m);
  int kolor = b->color;
  if (--(b->count)) {
    while (kolor == b->color) pthread_cond_wait(&b->c, &b->m);
  }
  else {  // last thread
    pthread_cond_broadcast(&b->c);
    b->count = b->numThreads;
    b->color = !b->color;
  }
  pthread_mutex_unlock(&b->m);
}
```

Could we still have a disastrous intercepted wait? Consider two threads running and needing to wait at the barrier. If T0 is already waiting, T1 will enter and realize it is the final thread. While still holding the mutex, T1 changes the `color` of the barrier (to `BLUE`, say), sets the `count`, and broadcasts the wake-up signal before releasing the mutex. If T1 races through the code following the barrier and encounters the barrier again (perhaps in a loop), it will see that it is not the last to arrive at a `BLUE` barrier. T1 decrements the barrier `count` and will call `pthread_cond_wait()`. From the fairness property of the interleaving abstraction, we know that T0 will eventually acquire the mutex. Upon return from waiting, it evaluates the `while` conditional expression. Since T0 was waiting at a `RED` (`kolor`) barrier, the expression is false (the current barrier `color` is `BLUE`) and the thread will exit the barrier. After running through the code following the barrier, the next instance of the same barrier encountered by T0 will be `BLUE`.

Threads under Windows Vista have added a `CONDITION_VARIABLE` object that works much like the Pthreads equivalent. A `CRITICAL_SECTION` object is associated and released when the thread waits by calling `SleepConditionVariableCS()`. Like many other Windows Threads functions that block threads, this function allows you to set a time limit. To wake threads sleeping on a `CONDITION_VARIABLE`, use a call to `WakeConditionVariable()`.

I can't take too much credit for the barrier implementation here. I had tried using two decrementing counters, the second of which counted threads leaving the barrier so that the final thread could reset all the counts. When this method kept deadlocking, I turned to *Programming with POSIX® Threads* (Addison-Wesley Professional, 1997) by David R. Butenhof and got the color inspiration. The barrier implementation given here is a simplification of the barrier code in Butenhof's book. Of course, his implementation is much more detailed and portable than the one I've cobbled together. I urge you to go over his codes and consider using his full implementation if you need to use barriers in more complex situations.

Design Factor Scorecard

How efficient, simple, portable, and scalable is the reduce code described earlier? Let's examine the algorithm with respect to each of these categories.

Efficiency

The code declares a local sum variable for each thread (`lSum`) to hold the ongoing computation of the local partial sum. Even though each thread will update a unique element from the `gSum` array, using the local sum variable avoids all the false sharing conflicts that could arise for each and every item within an assigned chunk of data. By updating a `gSum` slot once per thread, you can limit the number of false sharing conflicts to the number of threads, not the number of items in the original data array.

The barrier implementation will be an efficiency concern for the reduction algorithm. Besides getting a thread to wait on a condition variable in Pthreads or in Windows Threads, there is the overhead that comes from the extra code needed to decide whether a thread entering the barrier is the final thread and, if so, releasing all other waiting threads. The attendant bookkeeping that goes along with all of this is just more computation to be synchronized for correctness and more time spent not actually doing productive work.

Simplicity

With the help of Figure 7-1, the code is pretty simple and straightforward. The use of `i` and `p2` to determine which threads are allowed to proceed and from where data is gathered would be the most confusing parts to someone unfamiliar with the algorithm. (While I chose to handle them separately for the example, the `i` and `p2` variables could be combined into the `for` loop.)

Portability

OpenMP has an explicit barrier for threads within an OpenMP team. There are also implicit barriers at the end of OpenMP worksharing constructs that you can use (or turn off with the `nowait` clause, if not needed). Instead of using the `reduction` clause in OpenMP, you could write the explicit algorithm in OpenMP by attaching thread ID numbers to each thread in the team and using the explicit barrier. Normally, knowing someone was even contemplating such a use of OpenMP would elicit howls of derisive laughter, Bruce. But, if I can replace all the code from "A Barrier Object Implementation" on page 134 with the single line `#pragma omp barrier`, I would be willing to swallow my prejudices and take the simpler path.

In a message-passing system, you can write an algorithmic construction for reduction, similar to the one discussed previously. In this case, each process has a chunk of the data, the reduction computation on that portion of the data is computed locally, and the process ID numbers are used to coordinate messages between processes to pass the local partial results to other processes. There is no need for explicit barrier synchronizations, since the act of passing messages can guarantee the correct order of data is sent and sent only when it is ready. Receiving processes need to block until the data is received. These processes can easily compute from where the data will be sent. Within the MPI message-passing library, there is a reduction function that will, more often than not, give better performance than a handcoded reduction algorithm.

Scalability

Is this algorithm, with the barrier and all, the best way to do a reduction when there are a large number of threads involved? The implementation of the barrier object will be the principal limit to scalability of this reduction algorithm. The synchronization objects used within the implementation of a user-coded barrier object can be a bottleneck as the number of threads increases.

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.

About the Author

Dr. Clay Breshears has been with Intel Corporation since September 2000. He started as a senior parallel application engineer at the Intel Parallel Applications Center in Champaign, Illinois, implementing multithreaded and distributed solutions in customer applications. Clay is currently a courseware architect, specializing in multicore and multithreaded programming and training. Before joining Intel, Clay was a research scientist at Rice University helping Department of Defense researchers make the best use of the latest High Performance Computing (HPC) platforms and resources. Clay received his Ph.D. in computer science from the University of Tennessee, Knoxville, in 1996, but he has been involved with parallel computation and programming for over 20 years; six of those years were spent in academia at Eastern Washington University and the University of Southern Mississippi.

Colophon

The cover image is an aerial view of wheat-harvesting combines from Getty Images. The cover fonts are Akzidenz Grotesk and Orator. The text font is Adobe's Meridien; the heading font is ITC Bailey.

Software/Algorithms

The Art of Concurrency

"This is a luscious book that actually delivers on its title. You cannot teach art, but you can allow the apprentice glimpses of the adept. As someone who spent 30+ years working for supercomputer companies, and who has now been an academic for almost 10 years, I can say without question that this book rings true."

–Tom Murphy, Computer Science program chair, Contra Costa College

"Finally, a book with a practical focus on concurrency, including many real-world nontrivial algorithms that are analyzed in ways to improve performance using parallel programming techniques."

–Mike Pearce, Parallel Computing Scale Manager, Intel Software Network

If you're looking to take full advantage of multicore processors with concurrent programming, this practical book provides the knowledge and hands-on experience you need. *The Art of Concurrency* is one of the few resources that focuses on implementing algorithms in the shared-memory model of multicore processors, rather than just theoretical models or distributed-memory architectures. This book provides detailed explanations and usable samples to help you transform algorithms from serial to parallel code, along with advice and analysis for avoiding mistakes that programmers typically make.

Written by a senior Intel engineer with over two decades of parallel and concurrent programming experience, this book will help you:

- **Explore the differences between programming for shared memory and distributed memory**
- **Learn guidelines for designing multithreaded applications, including testing and tuning**
- **Discover how to best use different threading libraries, including Windows threads, POSIX threads, OpenMP, and Intel Threading Building Blocks**
- **Explore how to implement concurrent algorithms that involve sorting, searching, graphs, and other practical computations**

The Art of Concurrency shows you how to keep algorithms scalable to take advantage of new processors that will have more than two cores. For developing parallel code algorithms and for concurrent programming, this book is essential.

Dr. Clay Breshears, a Course Architect for Intel Corporation, specializes in multicore and multithreaded programming and training.

US $44.99 **CAN $56.99**

ISBN: 978-0-596-52153-0

9 780596 521530 54499