CHAPTER SIX

Parallel Sum and Prefix Scan

SUMMING THE ELEMENTS OF AN ARRAY OR FINDING ALL PARTIAL SUMS OF THE ELEMENTS in an array are basic algorithmic problems. The solution to these problems is easy to describe in a single sentence or two. The concurrent versions of these algorithms, known as *parallel sum* and *prefix scan* (or *parallel scan*), are simple and easy to understand. Since they are so simple, these problems have been extensively analyzed and are used as bellwether algorithms within the parallel programming community. Description, design, analysis, and implementation of these two algorithms will get our feet wet for the rest of the algorithms contained in the text.

Study of these two concurrent algorithms is all well and good, but if you can't find a use for them, you might think that reading through this chapter could be a waste of time. I'm sure you can imagine cases in which you might need to find the sum of an array of items or figure out the largest item within an array. These are examples of parallel sum. Prefix scan is a bit more abstract and its use as part of another algorithm is less obvious. So, after going over the design and implementation of these two concurrent algorithms, I'll point out some other algorithms where prefix scan is used.

Parallel Sum

You may need to compute the sum of all values within a given array. Example 6-1 shows a serial code that will perform such a summation of the `N` elements within the integer array `A`. After execution of the code, the `sum` variable will have the total sum of all elements from the `A` array (assuming no overflow or other exception was encountered).

EXAMPLE 6-1. Serial summation of integer array

```
int sum = 0;
for (int i = 0; i < N; i++)
  sum += A[i];
```

The parallel sum operation can work with any associative and commutative combining operation. For example, the operations of multiplication, maximum, minimum, and some logical operations would all be appropriate. In a serial program, the commutative property of the operation is not important, since the order with which the serial execution combines the elements is the same each and every time. Nevertheless, to have any chance of performing the computation concurrently, we rely on this property of the operation, since concurrent execution does not impose a fixed schedule on the order of independent computations. Throughout this chapter, I'll talk about addition or the summing of elements as the combining operation. Be aware that you can replace this with another appropriate operation when needed.

Looking closer at the serial code, since the value of `sum` at any single iteration depends on all the previous additions to `sum`, this might look like a situation that cannot be made concurrent. Luckily, this is a special case of an induction variable where the purpose of the computation is to "reduce" a collection of data into a single value. I've already told you that such a reduction operation can be made concurrent, so let's see how you can do it.

PRAM Algorithm

Imagine a complete binary tree with the same number of leaf nodes as there are elements in the vector to be summed. If we assign a vector value to individual leaf nodes and make the internal nodes of the tree correspond to the addition of child nodes, we can find the sum of all the leaf node values by working up the tree and carrying out the addition operations. Figure 6-1 shows an inverted binary tree with eight leaf nodes (squares) and assigned values. The internal nodes (circles) hold the sums of the node's children, and the root will have the sum of all vector values.

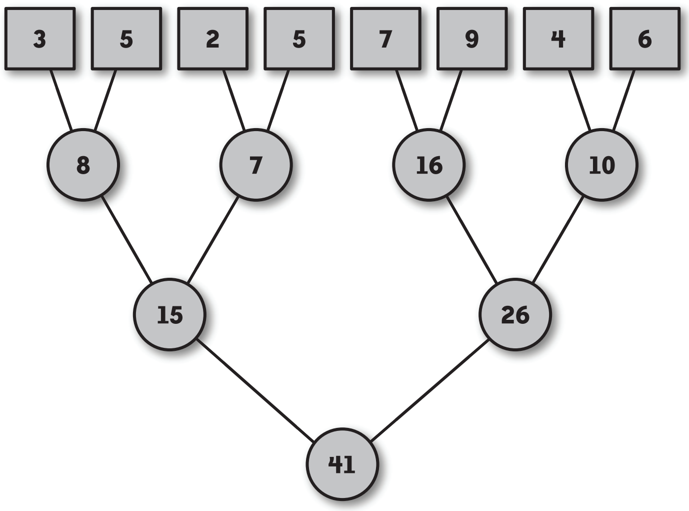

FIGURE 6-1. Binary tree illustration of parallel sum computation

Each addition within the same level of the tree is independent of all the other additions within that level. This is where the concurrency of the algorithm is to be found.

For a PRAM model, if you have a processor for each element of the array (leaf node), you can compute the parallel sum in a logarithmic number of steps. Example 6-2 shows the pseudocode for this computation adapted from the Daniel Hillis and Guy Steele paper, "Data Parallel Algorithms" (*Communications of the ACM*, 1986).

EXAMPLE 6-2. PRAM algorithm for parallel sum

```
for j := 1 to log_2(n) do
  for all k in parallel do
    if (((k + 1) mod 2^j) = 0) then
      X[k] := X[k - 2^(j-1)] + X[k]
    fi
  od
od
```

The parallel sum is done in-place (within the confines of the same array). Figure 6-2 shows the progression of the code in Example 6-2 on an eight-element array `X`. The arrows for each value of `j` point to the array element that receives the (partial) sum at each iteration of the outer loop. The final sum will be in the highest indexed element, `X[7]`.

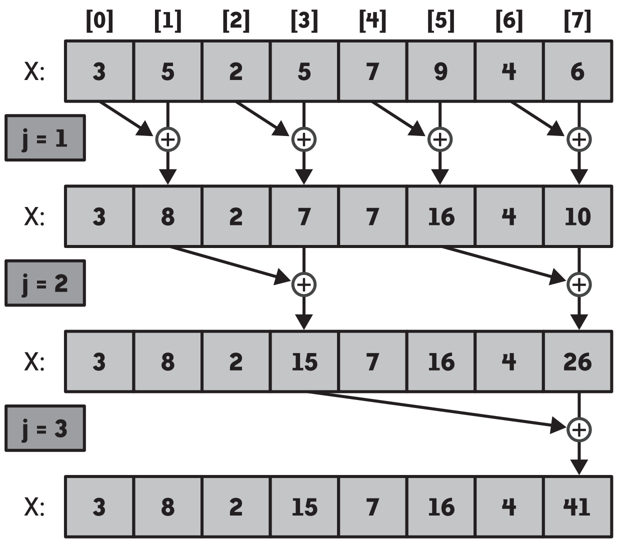

FIGURE 6-2. PRAM parallel sum algorithm example

A dash of reality

Can we use the PRAM algorithm for parallel sum (as given in Example 6-2) in a threaded code? Even overlooking the assumption of having a number of processors equal to the number of array elements available for any size array, a quick interleaving analysis of the algorithm with just two threads reveals a problem. If you divide up the inner loop between the threads, one of these can finish the assigned iterations and start up on the next set before the other thread has completed. Data races or the use of incorrect intermediate results can occur. Before you can compute any of the partial sums within a given iteration of the outer loop, you must be sure that all the previous inner loop iterations have been completed.

The pseudocode in Example 6-2 has no explicit indication of how to achieve this correctness property. The PRAM algorithm depends on having the execution of the body of the inner loop executed on separate processors all at the same time in lockstep synchronization. With the need for an unbounded number of processors and the reliance on lockstep execution, which is well nigh impossible without specialized hardware and operating system combinations, the PRAM algorithm isn't of much practical use.

A More Practical Algorithm

Parallel sum is a reduction algorithm. The concurrent algorithm for a reduction is based on data decomposition. To start, divide the data array into chunks equal to the number of threads to be used. Next, assign each thread a unique chunk and sum the values within the assigned subarray into a private variable. Finally, add these local partial sums to compute the total sum of the array elements.

OpenMP and Intel TBB have direct support for these operations built into each threading library. OpenMP includes a predetermined set of operations within the `reduction` clause that you can use to combine data. TBB allows more flexibility in that you must write your own code to sum items within chunks of data (inside the `operator()`) and how the results of those reduced chunk values are combined (through the `join` method). Example 6-3 has a summation code implemented with OpenMP.

EXAMPLE 6-3. Parallel sum using OpenMP reduction clause

```
int main(int argc, char* argv[])
{
  int sum = 0;
  int *X;
  int N;

  InitializeArry(X, &N);

#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < N; i++)
    sum = sum + X[i];

  printf("The sum of array elements is %d\n", sum);
  return 0;
}
```

After the initialization of the `X` array (`InitializeArray()`, not given), the loop worksharing construct divides the iterations of the loop into chunks and assigns those chunks to threads in the OpenMP team. The `reduction` clause ensures that each thread is allocated a properly initialized local copy of the `sum` variable. This local copy collects the partial sums of assigned iteration chunks. Once the threads have executed all the iterations, the `reduction` clause adds together (based on the `+` operator in the clause) and stores the sum of the partial sums into the global copy of the `sum` variable. The value within this global copy is then printed out.

The bulk of the work is done for you when using OpenMP or TBB for a reduction operation, especially when pulling together the partial sums into the final summation. Thus, citing Simple Rule 5, I recommend that you *use these to implement your parallel sum algorithms whenever possible*.

Should you find yourself in the position of writing your own reduction computation, you will need to divide the array elements and perform the final sums of partial sums explicitly. When there are a small number of threads, use a single thread to do the final summation in serial. This will require you to have the partial sums stored in globally accessible locations. Example 6-4 includes Pthreads code to implement the simple summation application. To keep things as uncomplicated as possible, I've set the number of threads to a fixed value (`NUM_THREADS`). Details about how the data is initialized (`InitializeArray()`) are left out of this code, too.

EXAMPLE 6-4. Parallel sum using POSIX threads and global partial sum storage

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4

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

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

  start = (N/NUM_THREADS) * tNum;
  end = (N/NUM_THREADS) * (tNum+1);
  if (tNum == (NUM_THREADS-1)) end = N;
  for (int i = start; i < end; i++)
    lSum += X[i];
  gSum[tNum] = lSum;
  free(pArg);
}

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

  InitializeArray(X, &N);  // get values into X array; not shown
  for (j = 0; j < NUM_THREADS; j++) {
    int *threadNum = new(int);
    *threadNum = j;
    pthread_create(&tHandles[j], NULL, Summation, (void *)threadNum);
  }
  for (j = 0; j < NUM_THREADS; j++) {
    pthread_join(tHandles[j], NULL);
    sum += gSum[j];
  }
  printf("The sum of array elements is %d\n", sum);
  return 0;
}
```

The main routine creates the threads that execute the Summation() function. Each thread first finds the boundaries of the X array chunk that it will be assigned. These boundaries are stored in local copies of start and end. The boundary computation requires that each thread be assigned a unique, consecutive integer ID number, starting with 0. The thread ID is passed to the thread via the single parameter allowed. The last thread (whose ID number is NUM_THREADS - 1) must be sure to pick up the final elements of the X array, since the code uses integer division. The if statement after the assignment of end will do this.

Once the boundaries are known, each thread loops over the chunk of X and sums those values into a local integer. After processing a chunk, each thread will copy the value of the locally computed partial sum into a unique global array element for the main thread to sum up into the final total.

As for the main thread, after initializing the X array, a number of threads are created (I've used the defined constant NUM_THREADS to set the number of threads in Example 6-4). Each thread's ID number is stored in a newly allocated integer and sent to the created thread through the single parameter. Before termination, threads must be sure to free the memory allocated for the thread ID to avoid a memory leak.

The main thread waits for each of the created threads to complete execution. Once a thread has terminated, the main thread will know that the associated partial sum for that thread is available in the global array and can be added to the overall sum variable. The total sum is printed after all partial sums have been combined.

Astute readers will realize that once the data has been reduced to a number of partial sums equal to the number of threads, you could use a concurrent algorithm, based on the PRAM algorithm. We'll take a look at another method for a reduction computation based more closely on the PRAM algorithm in Chapter 7.

Design Factor Scorecard

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

Efficiency

The addition of code to compute chunk boundaries is overhead to the concurrent execution. However, if the addition of two assignments and an if statement causes a noticeable degradation of performance, the data chunks may not be big enough to warrant concurrent execution. In this case, you need to increase the granularity of the chunks by reducing (or completely eliminating) the number of threads used.

You can completely eliminate the need for computing chunk boundaries by starting each thread at a different element of the array, based on thread ID, and accessing every fourth (or NUM_THREADSth) element. The code change for this scheme is given in Example 6-5.

EXAMPLE 6-5. Computation loop modification to eliminate need for start and end

```
for (int i = tNum; i < N; i += NUM_THREADS)
  lSum += A[i];
```

This change introduces a need to share cache lines between cores with threads accessing every fourth element (Example 6-4). However, since this will only involve read-access of the cache data, there should be no false sharing penalty. Still, this modification will make use of only a small set of items within a whole cache line. Thus, the time to read a full cache line will be practically wasted due to the fact that not all of the data in each line is used by a thread.

There will be some false sharing potential with multiple threads updating different elements of the `gSum` array. However, it is always better to have one possible false sharing incident per thread than to have one per data item if the threads are updating a single shared global variable for each addition (not to mention the multiple threads contending for the same synchronization object). As an alternative, you could offset each element in `gSum` used by the number of bytes in a cache line.

Simplicity

The two concurrent solutions here are very simple data decompositions of the original serial code. It is straightforward to compute partial sums of chunks from the array and then add those partial sums together into a final total.

While there appears to be a huge code explosion between the simple three-line serial version in Example 6-1 and the Pthreads version, the ratio in this case is a bit lopsided due to the very tiny amount of serial code that was used initially. Other algorithms that we will examine will start with more code, and, in my experience, the absolute number of added lines for implementing explicit threading will typically be close to the number of additional lines shown in Example 6-4.

Portability

The Pthreads algorithm translates pretty easily to other explicit threading models. There is nothing unique to Pthreads used here. The explicit threads algorithm uses the same idea as a distributed-memory algorithm. Processes are assigned a chunk of data, the data is summed locally, and the partial results are sent to a single process for final summation. If all processes require the answer, the final answer must be broadcast to all the other processes.

Scalability

With increases in the amount of data, the algorithm will scale well. If the data size remains fixed but the number of cores and threads increases, there will be a point (of diminishing returns) where dividing up the data into smaller chunks for more threads will begin to yield worse performance. TBB attempts to control the chunk size behind the scenes, while OpenMP has the `schedule` clause to give the programmer some control. For an explicitly threaded solution, you will need to determine the crossover point and build in limiting logic that controls either the number of threads that can be created or the minimum chunk size allowed in order to execute the parallel sum with multiple threads.

Prefix Scan

Prefix scan computes all partial sums of a vector of values. That is, the results of a prefix scan will be a vector—the same size as the original vector—where each element is the sum of the preceding elements in the original vector up to the corresponding position. Two uses of the results of this algorithm are to find the longest sequence of 1s in a binary array (sequence) and packing only the desired elements from an array (which I'll demonstrate later in this chapter). See *Principles of Parallel Programming* by Calvin Lin and Lawrence Snyder (Pearson Education, 2008) or Selim G. Akl's *Parallel Computation: Models and Methods* (Prentice Hall, 1997) for other uses of prefix scan.

Figure 6-3 shows an eight-element vector and the results of the prefix scan operation on that vector.

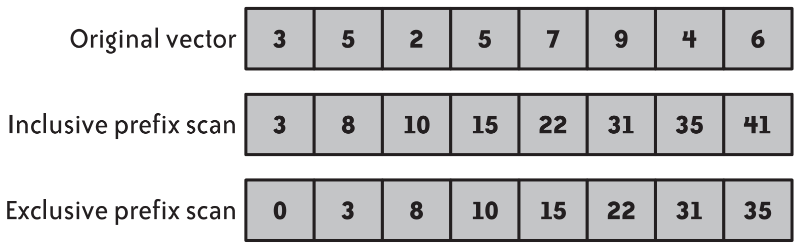
Original vector 3 5 2 5 7 9 4 6
Inclusive prefix scan 3 8 10 15 22 31 35 41
Exclusive prefix scan 0 3 8 10 15 22 31 35

FIGURE 6-3. Prefix scan examples

There are two versions of prefix scan: *inclusive* and *exclusive*. For an inclusive scan, the result is the sum of all preceding values, as well as the value of the element in the position under consideration. To compute the inclusive prefix scan of the fifth position in the vector, you need to add 3 + 5 + 2 + 5 + 7 = 22. The exclusive version does not include the value of the vector element at the position of interest. Thus, the exclusive scan result for the fifth position is 3 + 5 + 2 + 5 = 15.

As with parallel sum, the prefix scan operation can work with any associative combining operation. Besides addition, you can also use the operations of multiplication, maximum, minimum, and such logical operations as AND, OR, and eXclusive OR (but I'll continue to make reference to prefix scan using the addition operation). Unlike parallel sum, the combining operation does not need to be commutative because of the fixed order used to combine elements. Example 6-6 shows serial code to implement the inclusive prefix scan of an array of `N` integers, storing the results in corresponding elements of the array `prefixScan`.

EXAMPLE 6-6. Serial prefix scan computation of integer array

```
prefixScan[0] = A[0];
for (int i = 1; i < N; i++)
  prefixScan[i] = prefixScan[i-1] + A[i];
```

The code first stores the `A[0]` value into the `prefixScan[0]` slot to initialize the results array. Subsequent iterations of the loop use the `(i-1)`th value of the results array added to the `A[i]` value to compute the value to be stored in the `prefixScan[i]` element.

> **NOTE**
>
> **If you want to have an in-place version of the serial inclusive prefix scan, simply replace the `prefixScan` array with `A`. The exclusive version of the prefix scan is a bit more complicated, but I'll have a version of that later in the chapter.**

Upon examination of the serial code in Example 6-6, you should recognize the use of induction variables in the body of the loop. Unlike the parallel sum code, which used a single induction variable throughout, this algorithm uses a different induction variable for each iteration, and the value of that induction variable relies on the values of all the previously computed induction variables. It certainly looks less likely that we'll be able to create a concurrent solution for prefix scan than we were for parallel sum.

If you're familiar with TBB, you will know that one of the parallel algorithms included in the library is `parallel_scan`. This algorithm can compute the prefix scan of an array of values with multiple threads. So, citing Simple Rule 4, if you can use TBB and need a prefix scan operation, then make use of the library functions already written and optimized (and skip to "Selection" on page 112—of course, you never know what dollop of knowledge or juicy tidbit I might drop over the next few pages, so you might want to keep reading).

PRAM Algorithm

For those of you still reading, let's first take a look at how you can implement the prefix scan operation on the PRAM model. In "Data Parallel Algorithms" (*Communications of the ACM*, 1986), Hillis and Steele recognized that in their parallel sum algorithm (Example 6-2), most of the processors would be idle for most of the time. In fact, half of the processors would never be used at all. Making good use of the idle processors enables machines to carry out the computation of all partial sums in the same execution time as the parallel sum algorithm.

To illustrate this better utilization of processors, Figure 6-4 shows how data will be accessed (arrows) and combined in-place for computing the prefix scan on an eight-element array. Rather than showing the data, the figure shows the range of indexes that have been summed (using bracket notation) within the array element.

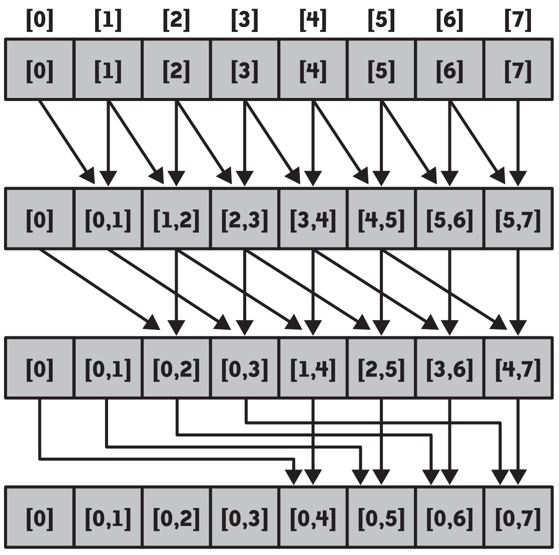

FIGURE 6-4. Example of PRAM computation for prefix scan

Before I give you the code for this operation, let's examine the operation of the PRAM algorithm by focusing on the slot of the array with index 5 and see how the prefix scan of the preceding elements is computed in that slot. The initial value is simply the data that is stored there (denoted by `[5]`). In the first step, the processor assigned to that position of the array will read the current contents of the index 4 element and will add that value to the contents of the index 5 slot, yielding the sum of the index 4 and 5 elements (`[4,5]`). The processor will then read the current contents of the index 3 element and will add that value (the sum of the original index 2 and 3 elements) to the contents of the index 5 slot. This will be the sum of the original elements from index 2 through and including index 5 (`[2,5]`). In the next step (the final step for eight elements), the processor will read the current contents of the index 1 slot and will add this value to the contents of the index 5 slot, which gives us the sum of all elements indexed from 0 through 5 (`[0,5]`). You can follow the arrows in the figure to see how each individual sum is computed from one step to the next.

We know that each step illustrated in Figure 6-4 is completed before the next step begins because the PRAM model works in lockstep execution. The algorithm assumes there is a separate processor assigned to compute each element within the array. All processors will take the same time to read the current contents of a lower-indexed slot, add the value found to the current contents of the assigned array element, and store the result back into the assigned array position. Thus, there is no need for explicit synchronization and we know there is no chance of data races in which one processor attempts to read a value at the same time the assigned processor is updating the contents of an array element.

The pseudocode for the PRAM inclusive prefix scan algorithm given in Example 6-7 is adapted from Hillis and Steele's "Data Parallel Algorithms." The only difference between this code and the code in Example 6-2 is the `if` test for determining which processors are active in the inner loop. This test uses the processor index (`k`) and determines whether looking back in the array a number of slots (2^{j-1}) is still within the lower bound of the array. If it is, the value in that lower indexed element is added to and stored in the current array element value.

EXAMPLE 6-7. PRAM algorithm for prefix scan

```
for j := 1 to log_2(n) do
  for all k in parallel do
    if (k ≥ 2^(j-1)) then
      X[k] := X[k - 2^(j-1)] + X[k]
    fi
  od
od
```

A less heavy dash of reality

Some interleaving analysis reveals that the practical problems we saw with the PRAM algorithm for parallel sum will also be problems when trying to directly implement the prefix scan algorithm from Example 6-7. We could simulate the PRAM algorithm after dividing the array into chunks for a finite number of threads. However, we'd still need to synchronize access to be sure data in lower indexed slots was read before it was updated. Also, we would need to make sure all computation for each iteration of the outer loop was complete before threads proceeded to the next outer loop iteration. This latter coordination task is further complicated as the number of array slots requiring computation shrinks for each iteration of the outer loop. In the last iteration, only half of the array elements are updated.

Looking back even further, the serial algorithm (Example 6-6) has far too many dependences and can't be directly adapted to a concurrent solution. If this is going to work concurrently, we'll need to invoke Simple Rule 8 and devise an equivalent algorithm that is more amenable to concurrency.

A More Practical Algorithm

Taking a cue from the parallel sum implementation, we can also approach prefix scan as a data decomposition design (there is no TBB version showing how to use the `parallel_scan` algorithm here; you'll have to look that one up yourself or wait until you get to the "Selection" section of this chapter). We can divide the array into a number of (roughly) equal-sized chunks, one per thread, and have each thread execute a serial prefix scan on the assigned chunk. Now we've got a set of "partial" prefix scans, one per chunk. How does that help? To compute the final answer for any one chunk, we need the sum of all the preceding chunks.

Luckily, the final element of an inclusive prefix scan is the total of all the elements in the array. If we run an exclusive prefix scan over just those final elements (maybe after copying them into an array), we will have the sums of all preceding chunks in the array location corresponding to a chunk (thread). We can then simply have each thread add this sum of preceding totals to all elements in the assigned chunk and compute the final prefix scan values. Figure 6-5 gives a pictorial example of this three-step algorithm on 16 elements using 4 threads.

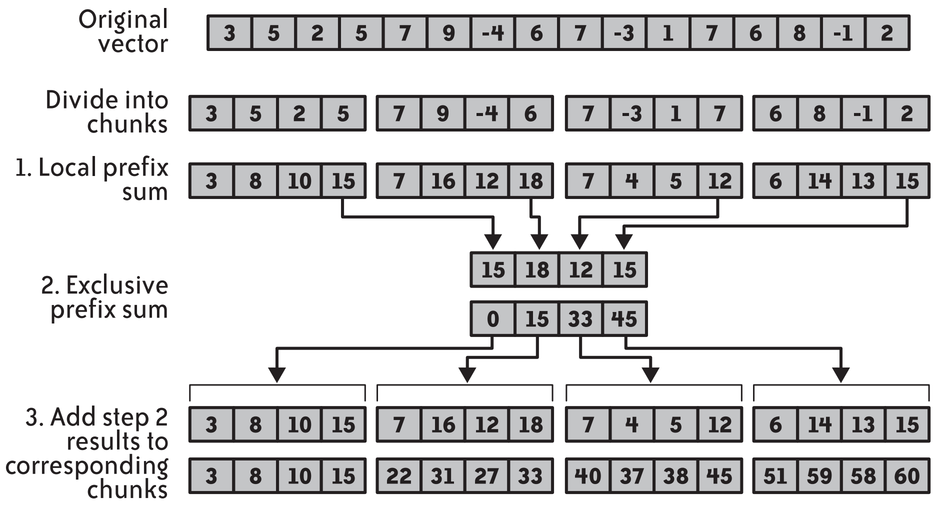

FIGURE 6-5. Prefix scan algorithm example

The first step (local prefix scan) and third step (adding the preceding chunk's sum to local chunk) can be done independently. The second step requires a prefix scan computation with a number of elements equal to the number of threads. We could implement an adaptation of the PRAM algorithm for this situation or take a hint from the parallel sum implementation and do this second step in serial.

If we do the second step in serial, we don't want to terminate the threads after completion of the first step, since we would need to recreate them for the third step. Starting and stopping threads is too much overhead and should be avoided if at all possible. Still, the serial thread executing the second step needs to know when all the chunk totals have been computed before it can start the prefix scan of the chunk totals. And, on that same side of the coin, the computational threads must wait until the serial prefix scan of chunk totals is complete before starting on the third step. Thus, we need some way to signal the completion of these two events.

Example 6-8 shows threaded code using Windows Threads and event synchronization objects to implement an inclusive prefix scan. As before, to keep things simple, I've written code that handles integers as the data on which to find the prefix scans, and set the number of threads to a fixed value (`NUM_THREADS`). Details about how the data is initialized (`InitializeArray()`) are omitted.

EXAMPLE 6-8. Prefix scan using Windows Threads

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
```

```
#define NUM_THREADS 4

int N, *A;
int inTotals[NUM_THREADS], outTotals[NUM_THREADS];
HANDLE doneStep1[NUM_THREADS];
HANDLE doneStep2;

unsigned __stdcall prefixScan(LPVOID pArg)
{
  int tNum = *((int *) pArg);
  int start, end, i;
  int lPrefixTotal;

  free(pArg);
  start = (N / NUM_THREADS) * tNum;
  end = (N / NUM_THREADS) * (tNum + 1);
  if (tNum == (NUM_THREADS-1)) end = N;

// Step 1
  for (i = start+1; i < end; i++)
    A[i] = A[i-1] + A[i];

  inTotals[tNum] = A[end-1];
  SetEvent(doneStep1[tNum]); //signal completion of Step 1

// wait for completion of Step 2
  WaitForSingleObject(doneStep2, INFINITE);

// Step 3
  lPrefixTotal = outTotals[tNum];
  for (i = start; i < end; i++)
    A[i] = lPrefixTotal + A[i];

  return 0;
}

int main(int argc, char* argv[])
{
  int i, j;
  HANDLE tH[NUM_THREADS];

  InitializeArray(A,&N);  // get values into A array; not shown
// Create manual reset events initially unsignaled
  for (i = 0; i < NUM_THREADS; i++)
    doneStep1[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
  doneStep2 = CreateEvent(NULL, TRUE, FALSE, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
    int *tnum = new int;
    *tnum = i;
    tHandles[i] = (HANDLE) _beginthreadex(NULL, 0, prefixScan, (LPVOID) tnum, 0, NULL);
  }
// wait for Step 1 completion
  WaitForMultipleObjects(NUM_THREADS, doneStep1, TRUE, INFINITE);
// Step 2
  outTotals[0] = 0;
  for (j = 1; j < NUM_THREADS; j++)
    outTotals[j] = outTotals[j-1] + inTotals[j-1];

  SetEvent(doneStep2);  //signal completion of Step 2

// wait for completion of Step 3
  WaitForMultipleObjects(NUM_THREADS, tHandles, TRUE, INFINITE);

  return 0;
}
```

What the main thread does

The main thread creates an array of event objects (`doneStep1`) for the computation threads to use in signaling when they have completed the work in step 1. Also, a single event object (`doneStep2`) is set up for the main thread to signal that it has completed the exclusive prefix scan computation of the chunk totals. After the events are set up, the computation threads are created and the process thread waits for all of the `doneStep1` events to be in the signaled state.

The exclusive prefix scan computation for step 2 uses two arrays: one to hold the original data, and one to hold the scan results. This makes the implementation simple, since it only requires one line of code. This one line of code works because we preserve the original data even when the corresponding result element is updated (I've included an in-place variation of this exclusive prefix scan in Example 6-9).

Once the exclusive serial prefix scan is complete, the main thread signals the computational threads that it is safe to begin step 3. The only thing left for the main thread to do is wait for the termination of the computational threads; this indicates that the scan of the original array is complete.

What the spawned threads are doing

Each computational thread first figures out which chunk is assigned to it using the thread ID number sent as a parameter to the `prefixScan()` function and stored locally as `tNum`. This is done in the same manner as the parallel sum code in Example 6-4. The threads compute the prefix scan on the chunk of assigned data. The value from the final chunk element (the total of all array elements within the chunk) is copied to the shared `inTotals[tNum]` slot. Since this completes processing for step 1, the thread sets the event object `doneStep1[tNum]` and waits for the signal that the main thread has completed step 2.

After receiving the `doneStep2` signal, each computational thread copies the value from the `outTotals[tNum]` slot and stores this value in the local `lPrefixTotal` variable. Each of the partial prefix scan values from step 1 in the assigned chunk is updated with the addition of `lPrefixTotal` to compute the final scan results. Termination of all threads clues in the main thread to the fact that the computation is complete and that it can use the results as needed.

NOTE

If you think back to the beginning of the prefix scan discussion, I stated that the combining operation must be associative but does not have to be commutative. This is due to the unambiguous order in which the results are combined. In "Data Parallel Algorithms," Hillis and Steele make note of this and point out that the code that does the combination within their PRAM pseudocode is written in a specific order, namely:

```
X[k] := X[k - 2^(j-1)] + X[k]
```

where the values and previous results that occur in lower indexed elements always appear on the left of the combining operator within the righthand side of the assignment statement. Faithfulness to this format will preserve the correctness of the algorithm if you ever find yourself using a noncommutative operator in place of the addition operation shown.

I've adhered to this ordering in all of the code presented in this chapter rather than using the shortcut operator +=.

Design Factor Scorecard

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

Efficiency

As with the parallel sum implementation, the code to compute chunk boundaries is overhead. The same analysis of chunk size applies here. Unlike the parallel sum implementation, the loop code modification to assign alternate iterations to threads shown in Example 6-5 cannot be used since the prefix scan algorithm relies on having contiguous chunks of data elements.

With multiple threads updating different elements of the `inTotals` array during the last part of step 1, there is a chance for false sharing. Concurrent access to the `outTotals` is read-only, so there should be no detriment from false sharing on that array in preparation for step 3.

If memory is tight and an additional totals array (`outTotals`) may be too much, you can use an in-place serial version of exclusive prefix scan. Code for this algorithm is given in Example 6-9.

EXAMPLE 6-9. In-place exclusive prefix scan in serial

```
int nexT = Totals[0], curr;
for (j = 1; j < NUM_THREADS; j++) {
  curr = Totals[j];
  Totals[j] = nexT;
  nexT = nexT + curr;
}
Totals[0] = 0;
```

Only two extra variables (rather than an entire array) are needed to keep track of the next prefix scan value (`nexT`) to be stored in the `Totals` array and to hold the current data value from the array (curr) before that value is overwritten. The final step of the code sets the first element in the array to 0.

Simplicity

Like parallel sum before, the concurrent solution in Example 6-8 is a data decomposition. This is not a straightforward decomposition of the serial code, since there are three steps, with one of those steps actually being a prefix scan computation across threads. Implementing something closer in structure to the PRAM algorithm for step 2 might be possible, but, due to the required synchronization to keep execution order, the code wouldn't be as simple as the serial version.

Even so, I think once you understand the three steps of the concurrent algorithm and what they accomplish, the code is easy to understand and to modify for some other operation besides addition. Lin and Snyder, in *Principles of Parallel Programming*, give an implementation for an abstract parallel prefix scan method (as well as one for reduction). Their algorithm isolates the combining operation of the scan and would allow you to easily reuse the code with a different combining operation.

Portability

The Windows Threads algorithm translates pretty easily to other explicit threading models. There is nothing uniquely Windows used here. You could implement this algorithm in a distributed-memory environment, too, with some message-passing between steps to gather data for the exclusive prefix scan of totals. The results of the exclusive scan are then scattered back out to processes for local processing.

You could also use this algorithm under OpenMP. The API finds the number of threads (thread ID number are required) and carries out the serial prefix scan of the Totals array under the single pragma. In my opinion, if you have to go to the OpenMP API functions and use those values to divide the data array into chunks, you have gone outside the realm of how best to use OpenMP. If you're going to go to that trouble, why not just write the code with an explicit threading library? If you've got other parts of your application in OpenMP and want to keep the entire application in OpenMP, then I'm OK with writing a small part of the code using OpenMP as if it were explicit threads. (You're free to do what you want, of course, I just ask that you don't tell me about it.)

Scalability

The prefix scan code given here is going to have the same scaling characteristics as the parallel sum. With an explicit threads implementation, you'll need to find where too little data in each chunk causes the implementation performance to degrade below acceptable bounds.