Sum3

Release 1.1

M McKenney, I Crk

March 11, 2015

A PDF (ebook) version of the module: Sum3.pdf

An Epub version of the module: Sum3.epub

CHAPTER

ONE

SUM3 PROBLEM OVERVIEW

1.1 Basic Description

The *Sum3* problem is described by a rather simple question: Given a set of n integers, how many triples of distinct elements in that list add up to 0.

For example, given the following list:

```
-1, -2, 0, 2, 3
```

the answer is 2:

```
-1 + -2 + 3 = 0
-2 + 0 + 2 = 0
```

The simplest way to code a solution to the *Sum3* problem is to use a double nested looping structure to generate the indices of all possible triples in the input array. Clearly, this is simple to express in code, but has an unfortunate time complexity of $O(n^3)$. For example, the following C++ code uses 3 `for` loops to achieve a correct solution.

```
#include <iostream>
using namespace std;

int main( )
{
        int dataSize = 5;
        int* data = new int[ dataSize ];
        data[0] = -1;
        data[1] = -2;
        data[2] = 0;
        data[3] = 2;
        data[4] = 3;
        // do the naive Sum3 computation.  O(n^3)
        int count = 0;
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0)
                                        count++;

        cout<< count <<endl;

}
```

1.2 Complete Naive Solution

The next code block uses the same solution as above, but includes a command line parameter that will generate an array of random numbers of a user-specified size. The `getRandInt()` function loads the vector passed into it with random integers. The `data` vector allocates space for the number of integers required, and is then passed to the `getRandInt()` function. This version of the program allows the user to get a feel for how and $O(n^3)$ program behaves in practice. Try compiling the program and running it.

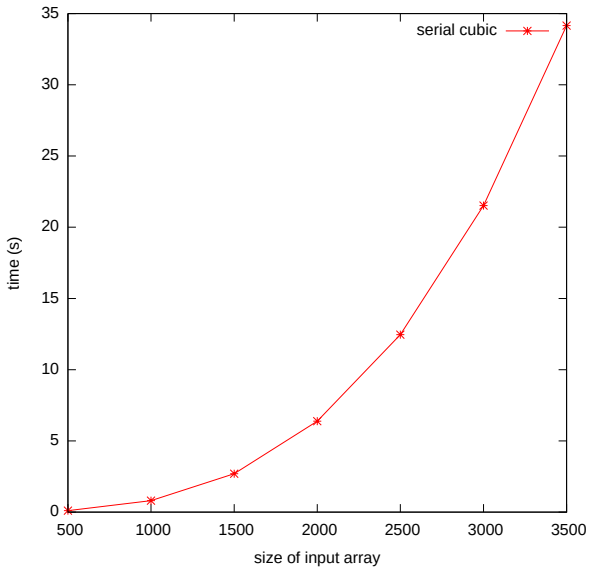
```
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt( int* dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int* data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );
```

```
51          int *dataPtr = &data[0];
52          // do the naive Sum3 computation.  O(n^3)
53          int count = 0;
54          for (int i = 0; i < dataSize-2; i++)
55                  for (int j = i+1; j < dataSize-1; j++)
56                          for (int k = j+1; k < dataSize; k++)
57                                  if (dataPtr[i] + dataPtr[j] + dataPtr[k] == 0){
58                                          count++;
59                                  }
60          cout<< count <<endl;
61  }
```

Here are some running times for the previous program:

Array Size	Time (s)
500	.106
1000	.804
1500	2.698
2000	6.382
2500	12.459
3000	21.519
3500	34.162

The running times are plotted. The cubic nature of the curve is clearly visible.



Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the `time` command on linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on linux, check out the `lspci` abd `lscpu` commands, and check out the `/proc` filesystem. Check the man pages and/or google for information on how to use/interpret these resources

CHAPTER

TWO

SUM3 SIMPLE IMPLEMENTATION (SERIAL)

For an input array of arbitrary integers (that is, we make no assumptions as to the range of the integers), *Sum3* can be solved in $O(n^2)$ time using relatively simple algorithms.

The purpose of this chapter is to emphasize the point that parallelism is not always the best solution to a problem. Sometimes, making algorithmic improvements (resulting in better theoretical complexity) can provide better speedups than parallelism; especially if you don't have the hardware resources to take advantage of a parallel algorithm.

2.1 Using a Hash Table

The easiest approach to code is to take advantage of hash tables. Hash tables are implemented in many languages, so they are cheap from a coding perspective, and they offer $O(1)$ time for inserting an element and determining if an element is in the hash table ... under ideal circumstances. The approach is to:

1. Load all numbers in a hash table
2. Generate all possible pairs of numbers
3. Check if the negation of the sum of each pair of numbers is in the hash table.

However, this approach suffers from some drawbacks. First, hash table operations can degrade to linear time under certain conditions (although this can be managed rather easily). Second, care must be taken to ensure that unique triples are identified. For example, if the input array was `-2, 0, 4`, one must ensure that `-2+-2+4 = 0` does not get counted as a valid triple (since -2 only appears once in the input).

2.2 Using a Sorted Input Array

A slightly more complicated $O(n^2)$ algorithm requires that the input array be sorted. The algorithm contains a loop that will traverse the input array from first to last.

Assume a sorted input array `array` containing `n` elements. We will first find all triples that add up to 0 and involve the first element in `array`. We set `a=array[0]`. A triple is then formed with `b=array[0+1]` and `c=array[n-1]`. If `a+b+c > 0`, then we need to reduce the sum to get it closer to `0`, so we set `c=array[n-2]` (decrease the largest number in the triple to get a smaller sum). If `a+b+c < 0`, then we need to increase the sum to get it closer to 0, so we set `b=array[0+2]` (increase the smaller number to get a larger sum). This process continues until `b` and `c` refer to the same array element. At the end of this process, all possible triples involving the first array element have been computed.

To compute all triples involving the first array element, we took a pair of numbers from the array. After examining the pair, one number was excluded from further consideration in future triples involving the first array element. Therefore, for `n` array elements, we will construct `n-2` triples due to the following: the first array element is used in every triple, leaving us to create pairs of the remaining `n-1` elements; the first pair uses two elements and eliminates one

from further consideration; all following pairs will reuse 1 element form the previous pair, and eliminate one element from consideration. Thus, computing all triples involving a single element requires $O(n)$ time. Repeating this for n elements in the input array requires $O(n^2)$ time. A C++ implementation of this algorithm follows:

```
#include <iostream>
#include <sstream>

using namespace std;
int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100+1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );
        // sort the data
        sort( data, data + dataSize );

        // do the Sum3 computation.  O(n^2)
        int count = 0;
        int a,b,c, sum; // array elements
```

```
56          int j,k;    // array indices
57          for (int i = 0; i < dataSize-2; i++){
58                  a = data[i];
59                  j = i+1;
60                  k = dataSize-1;
61                  while( j < k ) {
62                          b = data[j];
63                          c = data[k];
64                          sum = a+b+c;
65                          if( sum == 0 ){
66                                  cerr << a << " " << b << " " << c << endl;
67
68                                  count++;
69                          }
70                          if( sum < 0 ){
71                                  j++;
72                          }
73                          else {
74                                  k--;
75                          }
76                  }
77          }
78          cout<< count <<endl;
```

}

One pitfall in this algorithm is that duplicate values cause problems.

Example

Consider the input string `array = [-4, 1, 1, 3, 3]`. There are 4 triples that sum to 0:

```
array[0] + array[1] + array[3] = 0
array[0] + array[1] + array[4] = 0
array[0] + array[2] + array[3] = 0
array[0] + array[2] + array[4] = 0
```

However, the above code will only find 2 triples. The algorithm progresses as follows, with items being examined in bold:

-4, **1**, 1, 3, **3** sum = 0, so index **k** is decremented
-4, **1**, 1, **3**, 3 sum = 0, so index **k** is decremented
-4, **1**, **1**, 3, 3 sum = -2, so index **i** is incremented, resulting in j == k and the while loop exits

Effectively, the algorithm only finds triples involving the first **1**, and none involving the second **1**.

In fact, duplicates cause 3 problems that must be addressed:

1. 3 or more 0s will independently sum to 0. In fact, for `n > 2` 0s, there will be `n choose 3` triples that sum to 0.
2. A sequence of repeating numbers exists such that a pair of those numbers along with a third forms a triple that sums to 0. For example: ‘‘ -4, 1, 2, 2, 2‘‘ contains 3 triples that sum to 0. When such a sequence is detected

with length `n > 2`, there will be `n choose 2` triples that sum to 0. Note that a correct handling of this case can also handle the case where 3 or more 0s exist.

3. The situation in the above example when two sequences of repeating numbers exist such that the repeated element of each sequence along with a third number not equal to those elements sum to 0. For sequences of non-equal numbers `x` and `y` with respective lengths `m >= 1` and `n >= 1`, such that there exists a number `z | x+y+z = 0`, there will be `x*y` triples that sum to 0 for each individual copy of `z`.

The following program handles duplicates correctly. **Problem 1** and **Problem 2** are handled in a single conditional, the first `if` statement in the `while` loop. Note that when such a situation occurs, we have effectively found the point at which `j` and `k` converge, and so we break out of the while loop. **Problem 3** is handled in the second `if` statement in the `while` loop; the modification of `j` and `k` are again handled specially in this situation (since they may be increased or decreased by more than 1). The third `if` statement in the `while` loop handles the situation in which duplicates do not occur. The final `else` block in the `while` loop handles the normal increment of `j` or decrement of `k` when a triple with a non-zero sum is visited. The complete solution for handling duplicates is:

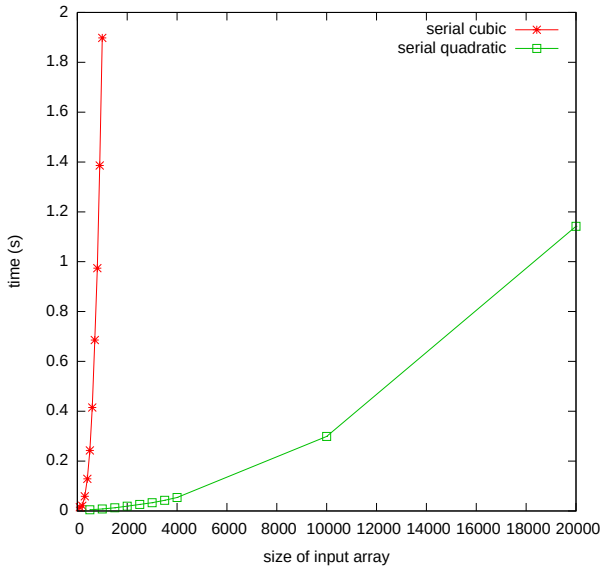
```
#include <iostream>
#include <sstream>

using namespace std;
int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }
```

```
44
45          // create a data vector
46          int *data = new int[ dataSize ];
47
48          // load it up with random data
49          getRandInt( data, dataSize );
50          // sort the data
51          sort( data, data + dataSize );
52
53          // do the Sum3 computation.  O(n^2)
54          int count = 0;
55          int a,b,c, sum; // array elements
56          int j,k;    // array indices
57          for (int i = 0; i < dataSize-2; i++){
58                  a = data[i];
59                  j = i+1;
60                  k = dataSize-1;
61                  while( j < k ) {
62                          b = data[j];
63                          c = data[k];
64                          sum = a+b+c;
65                          if( sum == 0 && b == c ) {
66                                  // case where b == c.  ie, -10 + 5 + 5
67                                  // or where a == b == c == 0
68                                  int num = k-j+1;
69                                  count += (num*(num-1))/2;
70                                  break;
71                          }
72                          else if( sum == 0 && (data[j+1] == b || data[k-1] == c )){
73                                  // case where there are multiple copies of b or c
74                                  // find out how many b's and c's there are
75                                  int startj = j;
76                                  while( data[j+1] == b ) j++;
77                                  int startk = k;
78                                  while( data[k-1] == c ) k--;
79                                  count += (j-startj+1) * (startk-k+1);
80                                  j++;
81                                  k--;
82                          }
83                          else if( sum == 0 ){
84                                  // normal case
85                                  count++;
86                                  j++;
87                          } else {
88                                  // if sum is not 0, increment j or k
89                                  if( sum < 0 ) j++;
90                                  else k--;
91                          }
92                  }
93          }
94          cout<< count <<endl;
95  }
```

Here are some running times for the previous program:

Array Size	Time (s)
500	.005
1000	.008
1500	.013
2000	.019
2500	.026
3000	.033
3500	.043
4000	.054
10000	.299
20000	1.142

Here we show the graph of running times for the cubic and quadratic versions of the algorithm. Notice the vast difference in running times for the two programs. Again, the cubic and quadratic nature of the algorithms are visible.



Exercise

Create a Makefile and use the Makefile to compile the program listed above. Run the program using the `time` command on linux to see some running times. Collect running times on two different machines. Write a brief report indicating your results and listing the machine specifications. List reasons why one machine is faster than the other.

To find out machine specs on linux, check out the `lspci` abd `lscpu` commands, and check out the `/proc` filesystem. Check the man pages and/or google for information on how to use/interpret these resources

CHAPTER

THREE

SUM3 MULTI-CORE OPENMP IMPLEMENTATION

At this point we have seen two possible algorithms and implementations for the Sum3 problem

1. A $O(n^3)$ algorithm that is extremely easy to implement, but that suffers from poor execution time.
2. A $O(n^2)$ algorithm that is much faster in terms of execution time, but that requires care to handle sequences of repeating values correctly.

The question now whether or not we can do better. It turns out that algorithmically, we cannot...at least not yet; there are currently no known algorithms to solve the Sum3 problem faster than On^2 time. In fact, a class of problems exists called Sum3 hard problems [GOM1995CG]. Any problem that is constant time reducible to a Sum3 problem is Sum3-hard; the implication being that a sub-quadratic time solution to any problem in the Sum3-hard class provides a sub-quadratic time solution to Sum3.

3.1 Introducing Parallelism with Multiple Threads

In terms of running time, we can do better with the Sum3 problem. One way to improve running time is to utilize a multi-core CPU with a multi-threaded program. The general approach is to divide the work that must be completed among multiple cores. Ideally, if we evenly divide the work among two processors, the program should run twice as fast. Another way to express this is to say that the program should have a 2x **speedup** over the serial version. The concept of speedup is a useful measure to gauge the effectiveness of a particular optimization, and is defined as follows:

Definition

Let t_{old} be the running time of a program and t_{new} be the running time of the same program that has been optimized. The **speedup** of the new program is equal to the running time of the old program divided by the running time of the new program:

speedup $= t_{old}/t_{new}$

3.1.1 Design of a Multi-Threaded Algorithm

Designing a multi-threaded program, or a parallel algorithm for that matter, requires a different thought process than defining a serial program. When defining a serial algorithm, the focus is usually to determine the steps that must be completed in succession to achieve an answer. When developing a multi-threaded program, one must think in terms of how the work can be divided among threads. It is sometimes useful to begin with a serial program and try to modify it to divide work, and other times it is easier to simply start from scratch.

Concept

How should the work be divided among threads?

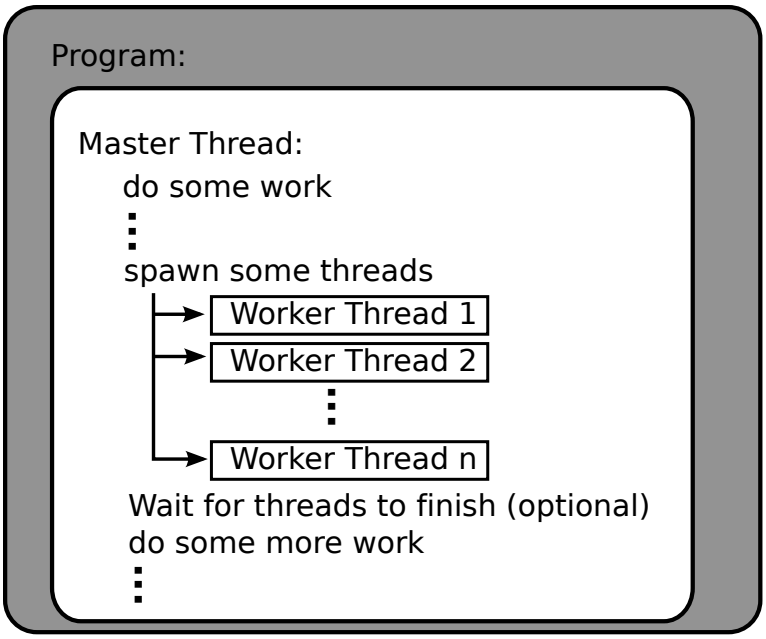
In order to effectively parallelize work, each thread must be able to do a portion of the work. Ideally, each thread will take roughly the same amount of time to execute; it is generally undesirable for a bunch of threads to complete their work quickly and sit around waiting on a single, slow thread to finish.

Lets take the cubic version of the Sum3 algorithm as an example (the core of the program is listed below). The main work of the program consists of the double nested loops that generate all possible triples of numbers from the input array. A reasonable method of dividing the work is to make 2 threads such that each thread generates roughly half of the triples that must be tested. We can achieve this by modifying the outer loop such that one thread will only compute triples in which the first number in the triple comes from the first half the array, and the second thread will compute triples in which the first number comes from the second half of the array. Such a modification requires minor changes to the core of the program.

```
#include <iostream>
using namespace std;

int main( )
{
        int dataSize = 5;
        int* data = new int[ dataSize ];
        data[0] = -1;
        data[1] = -2;
        data[2] = 0;
        data[3] = 2;
        data[4] = 3;
        // do the naive Sum3 computation.  O(n^3)
        int count = 0;
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0)
                                        count++;

        cout<< count <<endl;

}
```

3.1.2 Cubic Sum3 Using Pthreads

Pthreads are a low level threading mechanism that have been around for a long time. Pthreads are provided by a C library, and such, they do things in a very old-school, C way of thinking. The basic idea that the programmer defines a function, and then tells the pthread library to create a thread to run that function. Therefore, we have a concept of a master thread, which is the thread of the program that is started when the program is initially executed, and the concept of worker threads, which are spawned by the master thread. Typically, the master thread will spawn some worker threads, and then wait for them to complete before moving on, as illustrated in the following:



In order to achieve a pthread version of the program, we must first put the code that each thread will execute into a function. We will then call a pthread library call and tell it to use that function as the code the thread will run. The pthread library will then launch the thread. Our master thread can then launch additional threads, do other work, or wait on the launched threads to finish executing. Here are the function calls we will use... you should check your system's man pages for the most up to date information.

Pthread Library Calls

```
1 int
2 pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
3         void *(*start_routine)(void *), void *restrict arg);
```

Create a thread. The first argument will be assigned a pointer to a thread handle that is created. The second argument is a pointer to a thread attributes struct that can define attributes of the thread, we will simply pass a NULL to use defaults. The third argument is the function pointer to the function that this thread will execute. The final pointer is a pointer to a single argument that will be passed to the function. Note, to use the argument, the function must type cast it to the appropriate type. Also, if more than 1 arguments are required, you must pack them into a **struct**. Returns a

Returns a 0 on success, a non-zero error code on failure

```
1 int
2 pthread_join(pthread_t thread, void **value_ptr);
```

Join a thread. The join command will cause the calling thread to wait until the thread identified by the thread handle in the first argument terminates. We will pass NULL as the second argument

Returns a 0 in success, a non–zero error code on failure

Now that we have the ability to create and join threads, we can create a simple pthread program. The following example shows how to create threads to do some work, and pass arguments to them. Remember that pthreads require the function that will be executed as a thread to take a single argument; so we must wrap all the arguments we want to pass to the function into a single struct. We define the struct on line 23, and instantiate two instances of the struct on line 87. Because a pointer to the struct is passed to our `partial3SumThread` function as a `void *`, we must cast the pointer back to the correct type on line 33 before we can use it in the function.

We want to pass the input array to each thread, so we can simply pass it to the function.

Note

The function launched as a thread by the pthread library *must* take a single argument of type void pointer. This argument must be type-cast to a void to pass it into the function, and type-cast back to its original type within the function. Of course, this breaks type checking for the compiler, so be careful and double check the types yourself!

Here is the code:

```
#include <iostream>
#include <sstream>

using namespace std;

int getRandInt( int *dataVec, int dataSize )
{
        // load up a vector with random integers
        int num;
        for( int i = 0; i < dataSize; i++ ) {
                // integers will be 1-100
                num = rand() % 100 +1;
                if( rand( ) % 2 == 0 ) {
                        // make some integers negative
                        num *= -1;
                }
                dataVec[i] = num;
        }
}

// define a struct to pass information to the threads
struct threadInfo{
        int myID;
        int *dataPtr;
        int dataSize;
        int count;
};


void* partial3SumThread( void* arg ) {
        // type cast the argument back to a struct
        threadInfo * myInfo = static_cast<threadInfo*>( arg );


        // each thread only works on half the array in the outer loop
        // compute the bounds based on the ID we assigned each thread.
        // remember, we only have 2 threads in this case, so we will hard code a 2
        int start = ((myInfo->dataSize / 2) * myInfo->myID );
        int stop = ((myInfo->dataSize / 2)* (myInfo->myID+1));
        if( myInfo->myID == 1 )
                stop =myInfo->dataSize-2;

        // do the naive Sum3 computation.  O(n^3)
        for (int i = start; i < stop; i++)
                for (int j = i+1; j < myInfo->dataSize-1; j++)
                        for (int k = j+1; k < myInfo->dataSize; k++)
                                if (myInfo->dataPtr[i] + myInfo->dataPtr[j] + myInfo->dataPtr[k] ==
                                        myInfo->count++;
                                }
}
```

```
51
52  int main(int argc, char * argv[] )
53  {
54          int dataSize = 0;
55
56          if( argc < 2 ) {
57                  std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
58                  exit( -1 );
59          }
60          {
61                  std::stringstream ss1;
62                  ss1 << argv[1];
63                  ss1 >> dataSize;
64          }
65          if( argc >= 3 ) {
66                  std::stringstream ss1;
67                  int seed;
68                  ss1 << argv[2];
69                  ss1 >> seed;
70                  srand( seed );
71          }
72          else {
73                  srand( 0 );
74          }
75
76          // create a data vector
77          int *data = new int[ dataSize ];
78
79          // load it up with random data
80          getRandInt( data, dataSize );
81
82          // allocate thread handles
83          pthread_t       worker1tid, worker2tid;
84
85          // allocate and set up structs for 2 threads
86          threadInfo info1, info2;
87          info1.myID = 0;
88          info1.dataPtr = data;
89          info1.dataSize = dataSize;
90          info1.count = 0;
91          info2.myID = 1;
92          info2.dataPtr = data;
93          info2.dataSize = dataSize;
94          info2.count = 0;
95
96          // allocate space for a return value
97          int returnVal;
98          // call the worker threads
99          if (  returnVal = pthread_create( &worker1tid, NULL, partial3SumThread, &info1 ) ) {
100                 cerr<<  "pthread_create 1: "<< returnVal <<endl;
101                 exit( 1 );
102         }
103         if (  returnVal = pthread_create( &worker2tid, NULL, partial3SumThread, &info2 ) ) {
104                 cerr<<  "pthread_create 2: "<< returnVal <<endl;
105                 exit( 1 );
106         }
107         // now wait on the threads to finish
108         if ( returnVal = pthread_join( worker1tid, NULL ) ) {
```

```
109                 cerr<<  "pthread_join 1: "<< returnVal <<endl;
110                 exit( 1 );
111         }
112
113         if ( returnVal = pthread_join( worker2tid, NULL ) ) {
114                 cerr<<  "pthread_join 2: "<< returnVal <<endl;
115                 exit( 1 );
116         }
117
118         cout<< info1.count + info2.count <<endl;
119 }
```

The program has a relatively simple structure, the core of the Sum3 computation is done in the `partial3SumThread` function. 2 instances of the function are launched, and we assigned the first instance an ID of 0, and the second instance and ID of 1. These thread IDs are manually assigned and stored in a `threadInfo` struct that we create. The structs are created and initialized on lines 87-95. Based on the thread's ID, we determine which half of the array a thread should use in the outer loop of the Sum3 computation (lines 39-42). A thread is launched on line 100 and line 104. Once all the threads are launched, the main thread must wait for them to complete before printing the result, so the main thread calls one join function for each thread (lines 109 and 114). Once all worker threads have joined, we print the result.

Here are some running times on a machine with a 2 core cpu, and the results graphed along with the serial and serial quadratic versions of the algorithm:

Array Size	Time (s)
100	.01
200	.019
300	.029
400	.063
500	.125
600	.207
700	.324
800	.475
900	.668
1000	.912

The pthreads version is clearly faster. As expected, the version with two threads runs roughly twice as fast as the version with 1 thread. For an array of 500, the speedup is 1.9, for an array of 1000, the speedup is 2.08. A speedup of more than 2 is likely due to timing granularity. The linux `time` command was used, which is not the most accurate time keeping method. Other considerations should have an impact. For example, the two threads could have an effect on cache, causing a greater amount of cache hits or misses than the single threaded version. Cache misses cause the processor to stall, wasting time. More cache hits means the processor stalls less, and does the work in a shorter amount of time.

Exercise

The Pthreads version of the Sum3 problem is hard coded to handle 2 threads. Convert the program such that it will take the number of threads to use as a command line argument, and then generate that many threads. Hint: you will have an easier time of it if you use vectors to hold all thread information. For example, you will have a vector of `threadInfo` structs, on for each thread.

3.1.3 Thread Communication

In the Pthreads version of Sum3, very little thread communication took place. In fact, the only thread communication that happened was the master thread provided some data to the worker threads in the form of a function argument.

Usually, threads will communicate with each other through **shared memory**. The key to understanding shared memory is understanding scoping rules. Basically, a thread will have access to all memory locations that are in scope for that thread, and all memory locations accessible through pointers that are in scope for that thread. This means that if a variable has global scope, then ALL threads will be able to read and write to it. This means, two threads can communicate by reading and writing to such a value. In our program above, we declared two instances of the `threadInfo` struct in the `main()` function (line 87); those instances are `info1, info2`. Those structs now exist in memory, but the names `info1, info2` exist in scope for the `main()` function. We store information that the threads need in those structs. To provide access to that information, we pass the pointer to one of those structs to each thread during the `pthread_create` call. This means that a thread can access its struct through that pointer. Essentially, the master thread has communicated with the worker threads, letting them know their thread ID, the input array, etc.

Note that in the program, only 1 copy of the `data` array exists. It is created in the `main()` function. Pointers to that array are stored in the `threadInfo` structs, so that each thread can access the same array. This does not cause any problems because both threads only read the array, they are not making changes to it. Also note that each struct has its own memory location to store a `count`. Thus, each thread is keeping track of its own `count`. This is why all the `count` values from the `threadInfo` structs must be summed in line 119.

Exercise

Change the pthreads Sum3 program such that both threads access the same memory location for keeping track of their `count` variable. Don't use any other pthread library features. Run the program multiple times. What happens to the result value? why?

3.2 Doing Things a Little Differently with OpenMP

Pthreads is just one method to create multi-threaded programs. Most languages have some built-in mechanism or library to launch threads. Another mechanism in C/C++ is OpenMP. One drawback to pthreads is that although the core Sum3 computation changed very little in the pthread program, the program did require some somewhat significant structural changes. Recall that we had to put our computation into a function, set up a struct to pass data to the function, introduce create and join function calls, etc. Furthermore, we only implemented 2 threads, implementing more threads requires even more work. So, there is some programming overhead to converting a serial program to a parallel program using pthreads.

OpenMP takes a compiler-based approach to threading. Instead of inserting function calls and re-organizing code to create threads, OpenMP requires you to place compiler directives near portions of code that will be run in parallel so the compiler can generate a team of threads. These directives are known as **pragmas**. One of the goals of OpenMP is to provide a mechanism whereby serial code can be parallelized using multiple threads with only minor code modifications. The result is that you can write a serial program, debug it while it is serial (a much easier task than debugging parallel code), and then simply add a pragma to achieve parallelism using multiple threads.

As an example, we will write a OpenMP version of the Sum3 problem with a similar structure as the Pthread version: we will create 2 threads, and each thread will only iterate over a portion of the outer loop. We will need the following OpenMP pragmas and functions:

OpenMP Interface

```
1 #pragma omp parallel
2 {
3         // some code
4 }
```

Will create a team of *n* threads where *n* is the number of computational cores available on the computer on which this code is executed. Each thread will execute the code in the code block. No code will be executed beyond the code block until all threads have joined.

```
int omp_get_thread_num();
```

Returns the unique thread identifier of the thread that executes this function. If only a single thread exists, the function always returns 0. If multiple threads exist, the function returns a number in the range from `0` to `omp_get_num_threads()-1`.

```
int omp_get_num_threads();
```

Returns the number of threads in the current team. If only 1 thread exists (the serial portion of the code), it returns 1.

```
void omp_set_num_threads( int num_threads );
```

Sets the default number of threads to create in subsequent parallel sections (for example, sections defined by `#pragma omp parallel`).

```
g++ -fopenmp source.cpp
```

remember to use the OpenMP compiler flag to compile programs using OpenMP.

With these pragmas and functions, we can duplicate the Pthreads version of the program by forcing 2 threads to operate on the data. We will call `omp_set_num_threads(2)` (line 57) to force 2 threads, then break up the work similarly to what we did before. Much like Pthreads, OpenMP threads will communicate through memory locations; again, scoping rules apply: (1) any variable whose scope is external to a parallel section will be shared among all threads, and (2) all threads will have their own private copy of any variable declared within the scope of a parallel section. So, we will define an integer for each thread to keep track of the number of triples summing to 0 that it sees external to the parallel section, so we can add those counts together at the end (lines 54-55). Recall in the Pthread code, we had to explicitly set thread identifiers to 0 and 1; OpenMP will do this for us and each thread can find its ID using the `omp_get_thread_num()` function (line 62):

```
1  #include <iostream>
2  #include <sstream>
3  #include <omp.h>
4
5  using namespace std;
6
7  int getRandInt( int *dataVec, int dataSize )
8  {
9          // load up a vector with random integers
10         int num;
11         for( int i = 0; i < dataSize; i++ ) {
12                 // integers will be 1-100
13                 num = rand() % 100 +1;
14                 if( rand( ) % 2 == 0 ) {
15                         // make some integers negative
16                         num *= -1;
17                 }
18                 dataVec[i] = num;
19         }
20 }
21
22
23 int main(int argc, char * argv[] )
24 {
25         int dataSize = 0;
26
27         if( argc < 2 ) {
28                 std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
29                 exit( -1 );
```

```
30          }
31          {
32                  std::stringstream ss1;
33                  ss1 << argv[1];
34                  ss1 >> dataSize;
35          }
36          if( argc >= 3 ) {
37                  std::stringstream ss1;
38                  int seed;
39                  ss1 << argv[2];
40                  ss1 >> seed;
41                  srand( seed );
42          }
43          else {
44                  srand( 0 );
45          }
46
47          // create a data vector
48          int *data = new int[ dataSize ];
49
50          // load it up with random data
51          getRandInt( data, dataSize );
52
53          // do the naive Sum3 computation.  O(n^3)
54          int count1 = 0;
55          int count2 = 0;
56
57
58          omp_set_num_threads( 2 );
59  #pragma omp parallel
60          {
61                  int count = 0;
62                  int myID = omp_get_thread_num();
63                  // each thread only works on half the array in the outer loop
64                  // compute the bounds based on the ID we assigned each thread.
65                  // remember, we only have 2 threads in this case, so we will hard code a 2
66                  int start = (( dataSize / 2) * myID );
67                  int stop = ((dataSize / 2)* (myID+1));
68                  if( myID == 1 )
69                          stop =dataSize-2;
70                  for (int i = start; i < stop; i++)
71                          for (int j = i+1; j < dataSize-1; j++)
72                                  for (int k = j+1; k < dataSize; k++)
73                                          if (data[i] + data[j] + data[k] == 0){
74                                                  count++;
75                                          }
76                  if( myID == 0 )
77                          count1 = count;
78                  else
79                          count2 = count;
80          }
81          cout<< count1 + count2 <<endl;
82  }
```

Now, some running times. Again, speedup is roughly 2 for the OpenMP version, but the OpenMP version required much less code reorganization to implement. The running times are a few thousandths of a second higher than the Pthread versions. This is due to a different implementation of threading. In OpenMP, the compiler implements the threading, rather than OS system calls. Because of this, the threads get compiled a little differently. Also, different

OpenMP compilers will have slightly different performance results.

Array Size	Time (s)
100	.005
200	.014
300	.032
400	.066
500	.124
600	.212
700	.333
800	.492
900	.696
1000	.952

Exercise

Compile the serial cubic version of the program and the openmp version with compiler optimizations turned on. How do the running times compare then?

3.2.1 Creating a Lot of Threads

Because OpenMP is integrated into the compiler, the overhead of creating threads is small; thus, it is possible to create many more threads than available processor cores. An important concept of OpenMP is the idea of worker threads: on a computer with 2 cores, the number of worker threads defaults to 2. Even if many threads are created, only 2 will be able to run at any given time. Therefore, it is acceptable, although not always optimal, to create many more threads than there are processors available. One easy way to create a lot of threads is through the `omp parallel for` pragma. This pragma must be placed directly before a `for` loop. OpenMP will then generate a single thread for every iteration of that for loop. To be safe, make sure that the `for` loop contains the initialization, increment, and stopping condition in the loop declaration!

One problem with generating lots of threads is that we also need to then generate lots of `count` variables (lines 54-55 above). One alternative is to create a single global count variable, but then make sure that only 1 thread accesses it at a time. To ensure that only 1 thread accesses a variable at a time, we put that variable access in a **critical section** using an OpenMP `omp_critical` pragma. Anything in a code block directly following an `omp_critical` pragma is guaranteed to be accessed by exactly 1 thread at a time. Using these directives, our multi-threaded code looks very similar to our original code:

```
1  #include <iostream>
2  #include <sstream>
3  #include <omp.h>
4
5  using namespace std;
6
7  int getRandInt( int *dataVec, int dataSize )
8  {
9          // load up a vector with random integers
10         int num;
11         for( int i = 0; i < dataSize; i++ ) {
12                 // integers will be 1-100
13                 num = rand() % 100 +1;
14                 if( rand( ) % 2 == 0 ) {
15                         // make some integers negative
16                         num *= -1;
17                 }
18                 dataVec[i] = num;
19         }
```

```
}


int main(int argc, char * argv[] )
{
        int dataSize = 0;

        if( argc < 2 ) {
                std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
                exit( -1 );
        }
        {
                std::stringstream ss1;
                ss1 << argv[1];
                ss1 >> dataSize;
        }
        if( argc >= 3 ) {
                std::stringstream ss1;
                int seed;
                ss1 << argv[2];
                ss1 >> seed;
                srand( seed );
        }
        else {
                srand( 0 );
        }

        // create a data vector
        int *data = new int[ dataSize ];

        // load it up with random data
        getRandInt( data, dataSize );

        int count = 0;
        // do the naive Sum3 computation.  O(n^3)
        #pragma omp parallel for
        for (int i = 0; i < dataSize-2; i++)
                for (int j = i+1; j < dataSize-1; j++)
                        for (int k = j+1; k < dataSize; k++)
                                if (data[i] + data[j] + data[k] == 0){
                                        #pragma omp critical
                                        {
                                                count++;
                                        }
                                }

        cout<< count <<endl;
}
```

The one drawback of the critical section is that execution may serialize on it. Therefore, if every thread enters the critical section in every execution of the loop, then the program will behave much like a serial program in terms of running time. Therefore, placement of the critical section is important! Note that we placed it inside the `if` statement. Every triple must be tested to see if it sums to 0, but only a small portion of the triples actually sum to 0. If we placed the `if` statement in the critical section, we would get serial behavior. If every triple summed to 0 (an array of lots of 0s), we would also get serial behavior (regardless of the placement of the critical section with respect to the `if` statement). Thus, the previous method will have more reliable speedups in these edge cases, but this method will get speedups for input that we are likely to see, and requires only a few lines of modification to the code, and no logic changes!

Here are the running times. Note that the critical section does have an impact, but it is not too bad. A speedup of 1.9 vs 1.9 for the Pthreads version at an array size of 500 (the same!), and a speedup of 1.9 vs 2.1 for the Pthreads version at an array size of 1000:

Array Size	Time (s)
100	.005
200	.014
300	.032
400	.068
500	.130
600	.218
700	.343
800	.512
900	.722
1000	.989

Exercise

Change the OpenMP program using a critical section to serialize in 2 ways. First, use an input array of all 0's. Then move the critical section to contain the `if` statement. Compare running times to the serial version of the algorithm (the cubic one), and give a list of reasons as to why one is faster/slower than the other.

3.2.2 Reductions

OpenMP has a lot of other options, constructs, and functionality. Check the official documentation for more details. One of the advantages of OpenMP is that many of these constructs make it very easy to achieve tasks that one has to do manually with pthreads. A good example of this is the `reduction`.

A reduction simply means to combine many values into a single value using a specified operator. For example, we can reduce an array of numbers to a single sum of all the numbers in the array (by adding them up). Alternatively, we could reduce an array to its max or min value, by searching for the largest or smallest value, respectively, in the array. We can use a reduction to completely get rid of the critical section in the previous version of the code.

When declaring an `omp parallel for` pragma, we can identify values that will be reduced after all threads have executed. Each thread will then make its own local copy of any variable specified in a reduction, and OpenMP will automatically perform the specified reduction of all of those local variables into a single value. The reduction operation is also specified.

For example, to get rid of the critical section above, we should tell OpenMP to perform a reduction on `count`. Thus, every thread will get its own copy of count. We will specify a sum reduction, so that at the end of thread execution, all the local versions of the `count` variable get summed into a single variable. The result is that we only have to add a single line of code to achieve parallel implementation of Sum3. Line 55 specifies that a sum reduction on the variable `count` will be computed:

```
1  #include <iostream>
2  #include <sstream>
3  #include <omp.h>
4
5  using namespace std;
6
7  int getRandInt( int *dataVec, int dataSize )
8  {
9          // load up a vector with random integers
10         int num;
11         for( int i = 0; i < dataSize; i++ ) {
12                 // integers will be 1-100
13                 num = rand() % 100 +1;
```

```
14                  if( rand( ) % 2 == 0 ) {
15                          // make some integers negative
16                          num *= -1;
17                  }
18                  dataVec[i] = num;
19          }
20  }
21
22
23  int main(int argc, char * argv[] )
24  {
25          int dataSize = 0;
26
27          if( argc < 2 ) {
28                  std::cerr << "usage: exe  [num of nums] [optional seed value]" << std::endl;
29                  exit( -1 );
30          }
31          {
32                  std::stringstream ss1;
33                  ss1 << argv[1];
34                  ss1 >> dataSize;
35          }
36          if( argc >= 3 ) {
37                  std::stringstream ss1;
38                  int seed;
39                  ss1 << argv[2];
40                  ss1 >> seed;
41                  srand( seed );
42          }
43          else {
44                  srand( 0 );
45          }
46
47          // create a data vector
48          int *data = new int[ dataSize ];
49
50          // load it up with random data
51          getRandInt( data, dataSize );
52
53          int count = 0;
54          // do the naive Sum3 computation.  O(n^3)
55  #pragma omp parallel for reduction(+:count)
56          for (int i = 0; i < dataSize-2; i++)
57                  for (int j = i+1; j < dataSize-1; j++)
58                          for (int k = j+1; k < dataSize; k++)
59                                  if (data[i] + data[j] + data[k] == 0){
60                                          count++;
61                                  }
62          cout<< count <<endl;
63  }
```

And finally, here are some running times:

Array Size	Time (s)
100	.005
200	.013
300	.031
400	.068
500	.125
600	.213
700	.332
800	.505
900	.697
1000	.963

The running times are a bit slower than the Pthreads version, and a bit faster than the OpenMP version with a critical section. However, remember that the Pthreads version is hard coded to 2 threads, and the programmer essentially must implement their own reduction if you want to use more threads. This version required adding 1 pragma to an otherwise unchanged serial implementation and achieved threading that will automatically scale to the number of available processors with NO extra work! This code is easy to debug (you simply comment out the pragma and debug in serial), uses advanced features, and is easily adaptable to multiple hardware configurations. This should convince you of why OpenMP is a popular choice for threading.

Exercise

Look at the scheduling clauses for OpenMP (the wiki has a concise description). Try dynamic scheduling with various chunk sizes on the above program. Report the effects on execution time, and describe why those effects are occurring.

Convert the $O(n^2)$ version of the algorithm to a parallel algorithm using OpenMP. Try to get the fastest time for 100,000 numbers. Make sure to use a computer with multiple cores (and hyperthreading turned off!)

CHAPTER

FOUR

SEARCH

- *search*

[GOM1995CG] 1. Gajentaan, M.H. Overmars, “On a class of $O(n2)$ problems in computational geometry”, Computational Geometry: Theory and Applications, 1995, 5 (3): 165–185, doi:10.1016/0925-7721(95)00022-2.