Karl Ludwigsen

April 20, 2011

CS 515 Project 1

Design Documentation

There are three main arrays that are global to all threads of the program.

***candidate = (int \*) malloc(sizeof(int)\*((N+1)/2));***

***prime = (int \*) malloc(sizeof(int)\*((N+1)/2));***

***p\_sieves = (int \*) calloc( sizeof (int) , sieve\_upper\_bound);***

The candidate array contains all the numbers to be considered to be prime ( 2, 3, 5, 7, 9, 11, 13, 15, 17). The numbers in the prime array can be 1 (for prime) or 0 (for not prime). Each number corresponds to a number in the candidate array. The prime array is initialized as all primes:

***for (int i = 0; i< (N+1)/2; i++)***

***{***

***prime[i] = 1;***

***}***

The p\_sieves array contains the indexes of the prime sieves. The element of the prime sieve is 0 to indicate 2 as the first prime sieve. When finding sieve the first index is ignored as we go straight to index 1.

When we perform the crossing out we change the prime array and the corresponding index back to zero. As mentioned in the summary before the threads can go to work they need to be assign to their own block. There’s two arrays block\_index and block\_size. The block\_index array contains the starting number of the treads block. The block\_size contains the amount of numbers to be worked on.

***Int block\_sum = 1;***

***for ( int i= 0; i<num\_threads; i++)***

***{***

***int block;***

***block = calculate\_block\_size( i, N, num\_threads);***

***block\_index[i] = block\_sum;***

***block\_size[i] = block - 1;***

***block\_sum += block;***

***}***

***int calculate\_block\_size(int id, int range, int threads)***

***{***

***int base = range / threads;***

***int extra = range % threads;***

***return (id < extra) ? ++base : base;***

***}***

For example when threads is 6 and the range is 20. Calculate block size will return 4, 4, 3, 3, 3, 3. But the range we want is 2-4, 5-8, 9-11, 12-14, 15-17, 18-20 (note the first thread range always starts on 2).

The function calculate block size returns 4 but the block size will be 1 minus that (3). So we get a range from 1 to 4. Block\_sum is incremented by 4 (not 3) though. For the thread two it’s starting index is 5 and the block size is also 3. So it can be assigned 5 to 8. The block sum is then incremented by 4 again to account for the next thread starting number.

Block index and block size are used as comparisons to determine the numbers to work on during the worker function.

Once all thread data is initialized we can now spawn threads to the worker function.

***for(t=1; t< num\_threads; t++)***

***{***

***pthread\_create(&threads[t-1], &attr, worker, (void \*)t);***

***}***

***// Once all threads are spawned have main thread starts finding prime sieves***

***worker(0);***

We spawn threads and pass a thread id. Notice we start counting at 1 so we don’t pass thread id 0 to the worker function during thread spawning. That is reserved for the main thread, as indicated in the worker call immediately after the launching of threads.

Once we’ve spawned threads we can enter the worker function

**while (all\_sieves\_found == 0 || i < sieve\_count)**

**{**

**if (tid == sieve\_synchro)**

**{**

**find\_sieves (tid);**

**for (i = 1; i < sieve\_count; i++) // after a thread finds prime sieves**

**cross\_out\_multiples (tid, i); // check their block**

**}**

**else**

**{**

**for (i = 1; i < sieve\_count; i++)**

**cross\_out\_multiples (tid, i);**

**}**

**}**

The while loop continues until all sieves are found and the thread has crossed all multiples of the sieves. If a thread is assigned to finding sieves he needs to check his own block immediately afterward or the thread will be stuck in an infinite loop. Every time a sieve is found the variable sieve\_count is incremented letting other threads know that a sieve is available to cross out. Sieves are found in this manner:

***void find\_sieves(int id)***

***{***

***static int count = 1;***

***static int i = 1;***

***printf("Thread %d finding sieves... i is %d and count is %d\n" , id + 1, i, count);***

***int end = (block\_index[id] + block\_size[id]);***

***while ( (candidate[i] <= sieve\_upper\_bound) && (candidate[i] <= end) )***

***{***

***if(isPrime(i) == 1)***

***{***

***p\_sieves[count] = i;***

***count++;***

***sieve\_count++;***

***printf("Found prime sieve - %d\n", candidate[i]);***

***}***

***i++;***

***}***

***if (candidate[i] >= end && candidate[i] <= sieve\_upper\_bound)***

***{***

***printf("Have next thread help find the remaining sieves\n");***

***sieve\_synchro++;***

***}***

***else***

***{***

***all\_sieves\_found = 1;***

***printf("All prime sieves found!\n");***

***}***

***}***

When the first thread comes in we first need to find out what its range is. The thread looks for sieve until either the thread as reached the upper bound of the sieves or its own block range. If the sieve upper bound falls beyond the threads block the thread increments the global **static volatile int** *sieve\_synchro* variable to tell the next thread it’s his turn to look for sieves. Since the loop is initiated with static variables count, and i the next thread can begin immediately where the last thread left off. If the thread reaches the upper bound of the sieves the thread will tell everyone that all the prime sieves were found by setting the **static volatile int** *all\_sieves\_found* to 1.

The *find\_sieves* uses the utility function *isPrime* that passes an index of the candidate and prime arrays.

As said before once a sieve is found a thread can begin crossing out multiples

Here’s the loop call

**for (i = 1; i < sieve\_count; i++)**

**cross\_out\_multiples (tid, i);**

The value i that is passed in is the index of the p\_sieves array. The p\_sieves array contains indexes of the prime sieves that have been found so far.

***void cross\_out\_multiples (int id, int n) //sieve index is just 1 2 3 4...***

***{***

***int start = block\_index[id];***

***int end = start + block\_size[id];***

***int index = ( start%2 == 0) ? start/2: (start-1)/2;***

***// if the index is returned is lower than the sieve***

***//increment the index - we don't want to cross it out***

***if (index <= p\_sieves[n]){***

***index = p\_sieves[n] + 1;***

***//printf("(%d) index adjusted to ... %d\n", id, index);***

***}***

***// tells us which thread is working on which prime sieve***

***printf("(%d) Crossing out multiples of %d\n", id+1, candidate[p\_sieves[n]]);***

***while (candidate[index] <= end)***

***{***

***if (candidate[index] % candidate[p\_sieves[n]] == 0)***

***{***

***//printf("(%d) crossed out %d using %d \n", id, candidate[index], candidate[p\_sieves[n]]);***

***prime[index] = 0;***

***}***

***index++;***

***}***

***}***

The first step of the function is to translate the block\_index to an index related to the prime and candidate arrays. Say if a thread has a block from 51 to 100. 51 will translate to index 25 (since we have no even numbers). We first need to make sure that the number we’re checking against does not wipe out itself. Say we get 7 as a prime sieve and the first thread starts has range 2- 20 it’ll start checking at 9. In the while loop we continue crossing out numbers that are divisible with our current sieve number until we reach the end of the thread’s range.

After the while loop is done the thread will spin in the worker function awaiting a new sieve.

**Challenges:**

When starting out I first did things serially with one thread. Block sizes were tricky to implement because of the translation between the numbers to the candidate and prime arrays. When I first implemented the while loop in the worker function it was spinning in many scenarios. I didn’t always have the thread that was finding sieves check its own block afterwards making it spin. Little details like the sieve upper bound. It’s important to realize that the sieve upper bound is <= sqrt(range) rather than sieve upper bound < sqrt(range). In the later case of range 25, 5 would’ve been skipped. I considered having a variable prime count to be initiated at range-1 and then decremented whenever a multiple is crossed out. But this would involve a possible mutex that would slow performance and decided against it. Also in the crossing out if the sieve landed in the first block often times it would cross itself out by mistake then I realized I needed to ensure that multiples were higher than the sieve prime itself.

**Synchronization Issues:**

Synchronization was pretty straight forward actually. What was tricky was seeing the interleaving. Placing prints in certain places proved to be kind of bad. I also started with volatile variables in my while and for loops of the main function to ensure accuracy of data. I tried changing things around as I was worried if the variable all\_sieves\_found was a volatile variable and sieve\_count was not the thread doing the sieving might find all the primes before the other workers could start because they would take so long to check the while conditions. So it was a bit confusing. When I ran the script > results.txt like so, the prints **never** came out right. The prints and apparent interleaving always worked when doing single runs from a bash terminal. In either case I always able to find the correct amount of prime numbers.