Assignment 1.1,2,3: Wave equation
Steven Raaijmakers & Marcus van Bergen

Wave equation

We implemented the computation of a wave equation in C. This wave consists of two complexities; the space and time complexity. The current wave can be computed by taking the two predecessors of the wave, and applying some calculation to it. When we want to compute wave N, we need to know wave N-1 and wave N-2.

P-THREADS

First we made created a sequential implementation, so we could see which parts we could parallelise. Because the current wave depends on its predecessors we decided to parallelise the other complexity. We divided the equation in to N equal parts, where each thread computed one of these parts. We did this by taking i_max and dividing it by num_threads, and rounding down the outcome. So when i_max would have had a value of 100, and n_thread would be 10, the first thread will calculate the wave for I = 0 to I = 9, the second thread would take care of I = 10 to I = 19 etc.

Using a for-loop we computed the i_min and i_max for each thread depending on the num_thread and i_max arguments, and saved them in an array. By spawning the threads we requested the correct i_min and i_max from the array for each thread. After waiting on all threads for completion the process repeats itself.

OPENMP

Our OpenMP implementation is a mix of what we use in P-threads (data split between thread amount of the i complexity) and the sequential for-loop with the correct pragama definitions. As by the P-threads we divide the equation into N parts, and then put that in the sequential loop. After this we gave the for-loop the correct pragamas which gave us the OMP multi-threaded implementation.

Using these two, we are able to have a multithreaded application which performs faster than a p-thread implementation.

EXPERIMENT: SPEEDUP

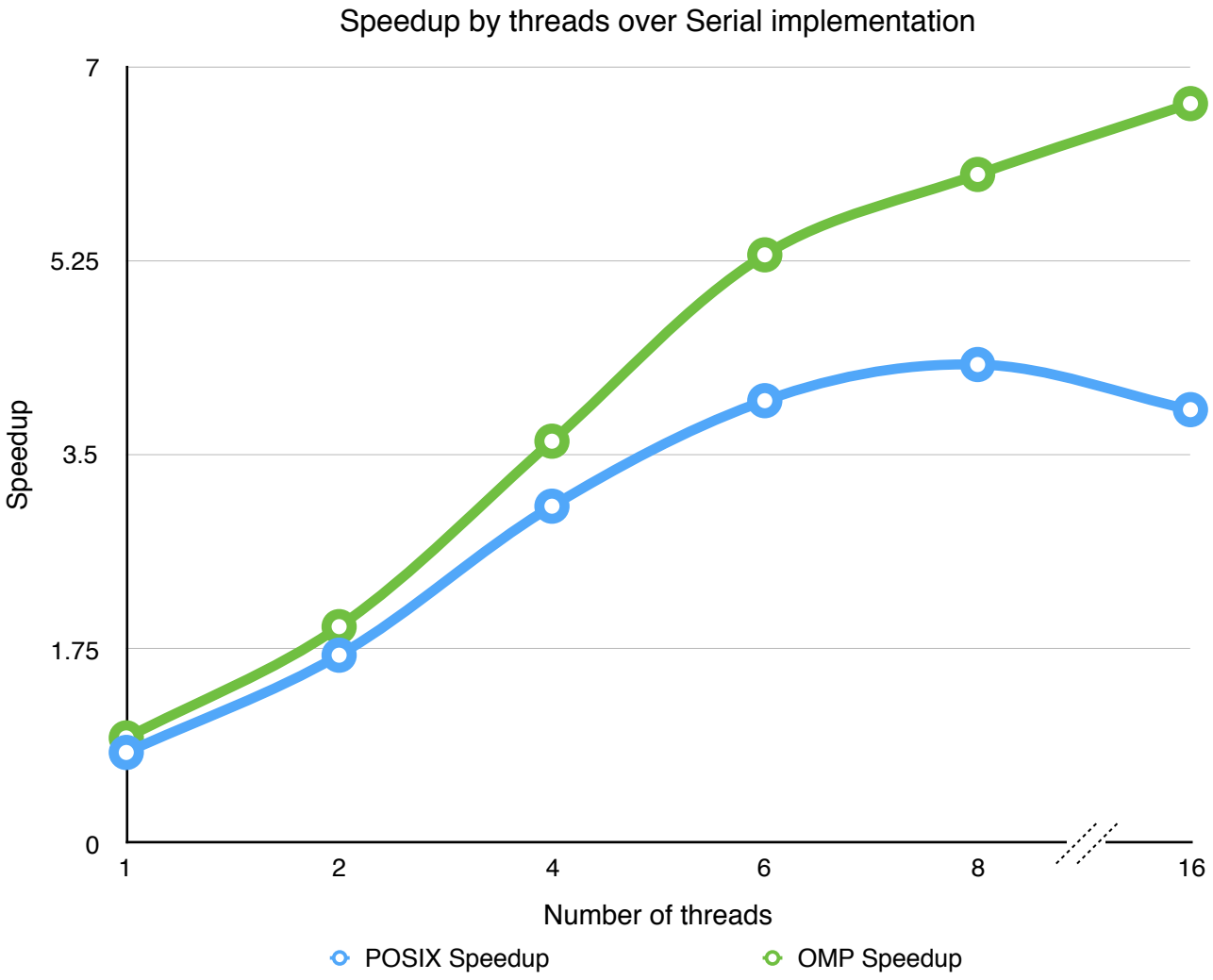
Running our implementations on DAS4 yielded the following results.

Speedup by threads over Serial implementation

Number Threads	1	2	4	6	8	16
P-Threads*	84.3596	40.747	22.7371	17.317	16.0096	17.6754
OMP*	72.7131	35.3778	19.0616	13.0256	11.4601	10.3623
Serial*	69.1114					
POSIX Speedup	0.819	1.696	3.040	3.991	4.317	3.910
OMP Speedup	0.950	1.954	3.626	5.306	6.031	6.670

*runtime in seconds

Using this information we're able to plot the following speedup graph.



The yielded results come some-what in line with what we were expecting. Both multithreaded implementations have a sub 1 speedup, meaning slower, in regard to the serial implementation. We expected this to be the case as both OMP and P-threads need to take time to spawn a thread, split the data, and then execute the same code as the serial implementation. Thus the total amount of operations that the multithreaded applications need to do with one thread is much higher than that of the serial; in other words: overhead.

When moving to 2, 4, or 6 threads we can see that there is an almost linear relation between the amount of threads being used and the speedup gained over the serial implementation. This was also expected behaviour, as we now start to take full advantage of the processor on this particular node, and the concurrent calculations being done.

When moving to 8 threads we can see the relationship between amount of threads and speedup yielded starts to diminish. We believe that for this wave application 8 would be the optimal amount for P-threads, seeing as this is where you can get the highest speedup, while not spawning too many threads to also cause overhead. Sixteen P-threads actually yields lower performance than 8 or 6. This is possibly due to there being so many threads per core that they have to be scheduled. This takes away from the amount of processing which would be able to be done *concurrently* had there been a lower thread amount. Instead, we now believe that the scheduling of so many P-

threads leads to time being wasted on switching and fetching resources, which leads to a longer run-time.

Interestingly enough, the same does not occur for OMP threads. At a higher amount of threads, such as 16, the yield does diminish, however the speedup seems to become higher. Our explanation for this, in comparison to P-threads slower performance, would have to with OMP's high-level abstraction.

We believe that OMP possibly finds a better way to store resources and data per thread, that even when scheduling threads, the resources needed to be swapped is less than by P-threads. An example for this could be that the i-points are only calculated one time for all T's. This could lead to less time lost on resource fetching, thus implying a lower run-time and a higher speedup.

Assignment 3: Sieve of Eratosthenes

We implemented a version of the “sieve of Eratosthenes” in parallel using pthreads in c. We used two concepts by designing this program; the consumer-producer concept and a pipeline.

At initialisation we created a generator-thread which produces an infinite amount of sorted natural numbers, starting with 2, and puts them in a buffer. The idea of the generator thread is that it acts as a producer. So for example, when the buffer is full, the generator stops with putting new natural numbers in the buffer until there’s space again. Or when its putting new values in it makes sure no other filters are consuming this buffer.

We also created a filter-thread, which takes a filter-number and an incoming buffer, from which it can consume values. It also has a outgoing buffer, so it also acts like a producer. Naturally we give the first filter thread the filter number 2. The filter thread will consume all values it gets from the incoming buffer. Depending on the obtained value we decide whether to drop the value or produce. If the obtained value is not divisible by the current-filter-thread-number, we produce by putting the obtained value in the outgoing buffer. We then check if the current filter thread has a successor; if it doesn’t we create a new filter-thread which takes the just obtained value as its filter-number and the current-thread’s outgoing buffer as an incoming buffer. This creates a pipeline, which keeps extending itself whenever a new prime is found. This also means that for every found prime number, there must be exist one thread.

So for example, we get the number 1999. This is a prime, but we only now this because we check if 1999 is divisible by any of the previous found primes. 1999 will enter the pipeline at filter-2, which forwards it to 3 etc. After a while 1999 reaches filter-1997, which decides 1999 is a prime because none of the previous found primes can divide 1999 without a remainder. Filter-1997 will then create a filter-1999. Because all of the filters in front of filter-1997 are free, the next prime-candidate can already be checked by them..

The following results were found when running our application on the DAS4 machine. 5000 threads was not possible to be run, as the machine had been capped at somewhere around 4990 maximum threads.

Primes	**100**	**1000**	**5000**
Time (seconds)	0.0151625	0.577358	Not Available