Parallel programming
Ville-Matti Saarelainen
1800137

Exercise 1: OpenMP

I decided to shorten my code by writing a function which can calculate the angles required, when given any two datasets. The only parallel part of the program is this function. This means I call the function three times, once for every histogram.

While this did work when using an atomic statement for incrementing the histogram bins, I did not find a way to easily get the reduction operation to work. I believe the issue is how the C language deals with pointers to arrays being passed into functions, and how they are used by OpenMP. According to my reading, this type of reduction operation is supported in newer versions of OpenMP, but it is not available on Dione (though I didn't try manually loading the newest GCC module, maybe it in fact is available). It may be possible to compile the program with a newer OpenMP version outside of Dione, then copy it in and execute it, but I have not tried it. Another potential workaround would be to make local copies of the histogram inside the function, then write out the results into the original array pointer passed into the function.

The reason this issue is important is that the reduction operation runs considerably faster than the atomic equivalent. I have included the results a friend of mine got using a reduction operation (and no abstracted calculation function) in the table of results for comparison.

Also, using the single-precision versions of the trigonometric functions gives a performance benefit (667s down to 441s) on my laptop, but on Dione there is no difference probably because the server CPUs in Dione have better hardware support for calculations with doubles.

Cores:	1 (Dione)	8 (i5-8250U)	40 (Dione)
Time:	3482s	441s	255s
Time (with reduction)	~3300s	-	~120s

Exercise 2: MPI

For solving the problem with MPI, I opted for a slightly different design than with OpenMP. In my design, every thread reads in the data files on startup, and determines independently which portion of the calculations it should perform. Thread ID 0 is special, in that all the data is gathered up on thread 0 in the end. Also, thread 0 only calculates angles between galaxies if the number of threads is such that the problem is not evenly divisible between the "worker" threads.

In this design, every thread calculates histograms DD, DR and RR consecutively, as opposed to the OpenMP solution where the calcangles()-function was parameterized with respect to input data. In theory, this may yield better caching effects with respect to initializing the input data, but in reality I made the change because it was easier to implement that way.

Cores (on Dione):	40	100	1000
Time:	48s	23s	10s

Exercise 3: CUDA

For the CUDA version of the program, I decided to try using two-dimensional thread blocks. This makes it rather easy to decompose the two-dimensional calculation into one calculation per thread, but as it turns out I also need to compute a one-dimensional thread ID for clearing shared memory and copying shared to global memory, so the benefit is marginal.

A constant N is used to control the dimension of a thread block. I got the best results with N = 32 (the maximum value for a GPU on Dione), but I have read that decreasing this value may result in performance gains in some cases. In my implementation I cannot go lower than N = 19, as a minimum of 360 threads are required for copying the local histograms into global memory. It might be faster to copy two histogram values per thread, and use e.g. N = 16.

The CUDA compiler has an extra optimization flag, --use_fast_math, which increases the speed of floating-point operations at the cost of accuracy. In my limited testing I saw differences as high as 0.1, but for this problem the results are still roughly the same. Of course, for real science loss of precision is undesirable.

Parameters:	N = 19, -O3, --use_fast_math	N = 32, -O3, --use_fast_math	N = 32, -O3
Time (on Dione):	1.6s	0.85s	1.13s