CHALMERS UNIVERSITY OF TECHNOLOGY

DAT280 Parallel Functional Programming Lab A

Johan Swetzén, 890916-5576
swetzen@student.chalmers.se
Sebastian Lagerman, 890608-5017
seblag@student.chalmers.se

March 31, 2014

TASK 1

We have implemented four different versions of a parallel `scanl1`. To make sure we used a slow function we also implemented a slowOp which is a glorified add function.

par & seq The first one we tryed was a `scanl1` implemented with the par & seq function. Our first attempt using only par & seq in the beginning of the function call ended up with a lot of spark that were overflowed. Directly afterward we added a depth to prevent the number of sparks. This lead to a hugh inprovment in spark-handling. During this time the function completed at about the same time as the normal scanl1 function. Lastly we forced the spark to evaluate to normal form which ended up making the function faster then the original.

Divide & conquer Our divide & conquer was inspired by Simon Marlow lecture which we borrowed the divConq-function. Once we had that function it was easy to send the right parameters.

Strategies, parList We started of our strategies with two functions. The first was called using the parListChunk-strategy and the second used the parList-strategy. Both of which were called with the rdeepseq to force them to normal form. During our testig phase we found that using the second alternative was more efficient.

Par monad Once we got the the par monad implementation we directly implemented that we spawn two processes to start working on the two halves of the list and then wait to recieve them. It gave us exelent results about the same as the strategies-function, but we felt we could get more out of it. So we decided to dig a bit deeper and make the mkFan-function parallel just to see how much this would effect the result. For those efforts came good results and the functions prossesing time halved once again.

TASK 2

par & pseq We started out with a par & pseq version of `fft`. that turned out to be pretty fast. A big speedup came when we added `rnf` to the left and right part of the list before merging, forcing them to be evaluated to normal form.

rpar & rseq It was difficult to get away from really small spark sizes which resulted in a lot of garbage collected and fizzled sparks. In the end we couldn't get any speedup on this one, but rather it slowed down.

Strategies Based on the divConq code shown in lecture 4, we made a very quick attempt at a Strategies-based solution. However, this only made thigs slower. Since par & pseq was fast already, we didn't spend much time on this.

Divide & conquer with Par Simon Marlow showed a `divConq` function based on Par in his lecture and we wanted to try that out. Using `fft` and `bflyS` that we were given, the simples possible application of the divConq function yielded very good results. In fact, this is our best version and it's probably the one we spend the least time on.

Par Monad This was one of the more involved implementations, but still quite straightforward. Both `fft` and `bflyS` were modified to work with `Par`, so that we could stay in the monad as much as possible. As in some of the other functions, we calculated the twiddle factors, `tws`, separately so that we could spawn that separately with parMap (replacing the list comprehension).

FINDING THE BEST PARALLELISATION

All our functions were benchmarked using Criterion, but also individually with the -s and -ls parameters to the RTS. In most cases we have tried to introduce a threshold parameter to control the chunk size, and experimented with different values to get the best speed. One exception is our best FFT function, which uses the divide and conquer function from Simon Marlow's lecture. One answer might be that the Par Monad does a good job, and it is sometimes worth letting it do its thing without meddling too much with the inner workings. This goes well along with the fact that our simple par & pseq was so good. We simply tried to run the two parts in parallel, which is pretty much the same as divConq does.

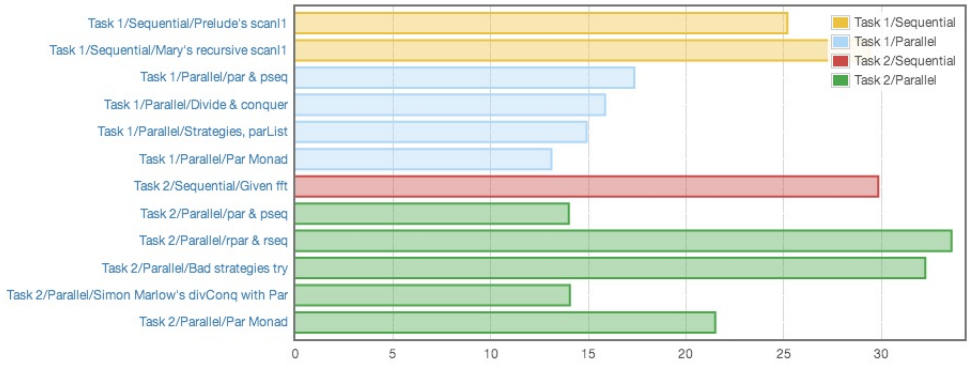

Figure 2.1: Our benchmarks

MEASUREMENTS

All these mesurments were aquired on a MacBook Pro with a dualcore. Our best speedup on task 1, according to Criterion, was 1.92. For FFT the figure was 2,13. This is quite remarkable since we would not expect more than twice the speedup on two cores, but we happily assume that we managed to improve the function as well as parallelising it.