Programming Languages

Assignments

Titouan Vervack

Bachelor of Science in Informatics

19 May 2014

Used hardware and software

All of the tests ran on the same hardware and software, which you can see below:

Hardware:

- **CPU:** Intel I7 2600k quad core @3.4Ghz
- **RAM:** Corsair Vengeance 8GB (2x4GB) CL9 @1600Mhz
- **HDD:** Western Digital Blue 1TB 7200RPM 64MB cache

Software:

- **OS:** Windows 8.1 Pro 64-bit
- **Compiler:** Mozart 1.4.0

Every program is a standalone program that was compiled using the command **ozc -x Code.oz**.

Assignment 1 Lazy execution

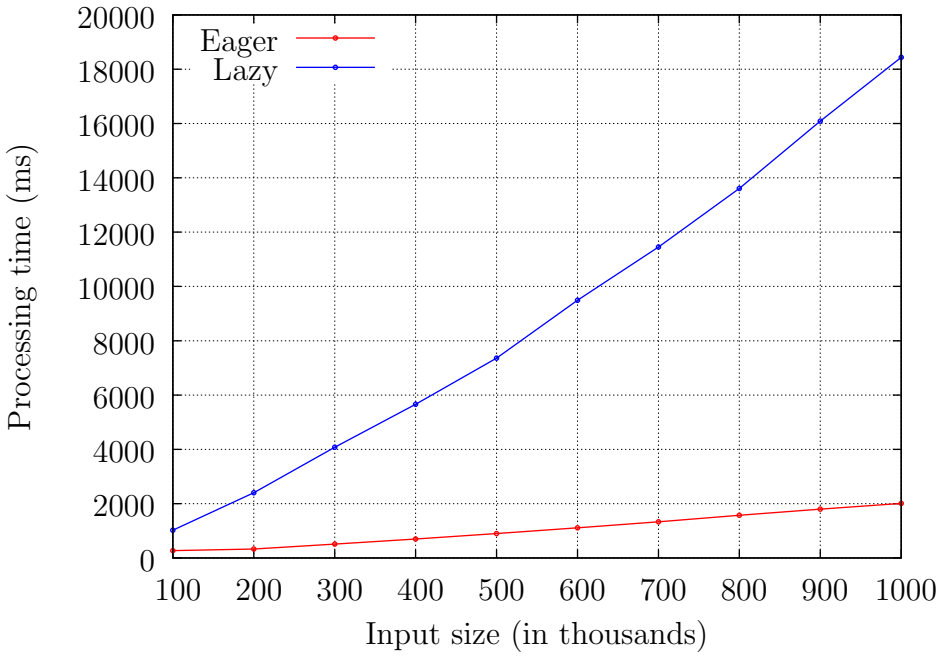
Assignment 1.1 Performance isssues

For this assignment I decided to implement the Sieve of Eratosthenes and the mergesort algorithm. The sieve commands expect one integer as input which represents the upper-bound for the sieve. Mergesort expects one integer as input which represents the amount of numbers to sort. Mergesort first creates a list of N random elements in Oz and then sorts it.

Sieve of Eratosthenes

In the lazy version everything is lazy, including the parsing of the input arguments. The eager version on the other hand does nothing lazily. The eager and lazy version are almost exactly the same. The only difference are the lazy keywords in front of every function and an added touch function, to force the list of prime numbers to be calculated.
Since both versions are so alike their time complexity is the same. We observe that the lazy version is slower though. Lazy evaluation should be preferred if it allows us to skip the calculation of certain parts in the program. This also means that when the same amount of information has to be calculated, lazy does worse than eager since it has more overhead. This is lazy evaluation's drawback. In the case of this Sieve, both lazy and eager have to calculate the same amount of information. This makes the eager version more preferable since it is as easy to write as the lazy version and performs better (less overhead).

We can see the results of the benchmark beneath.



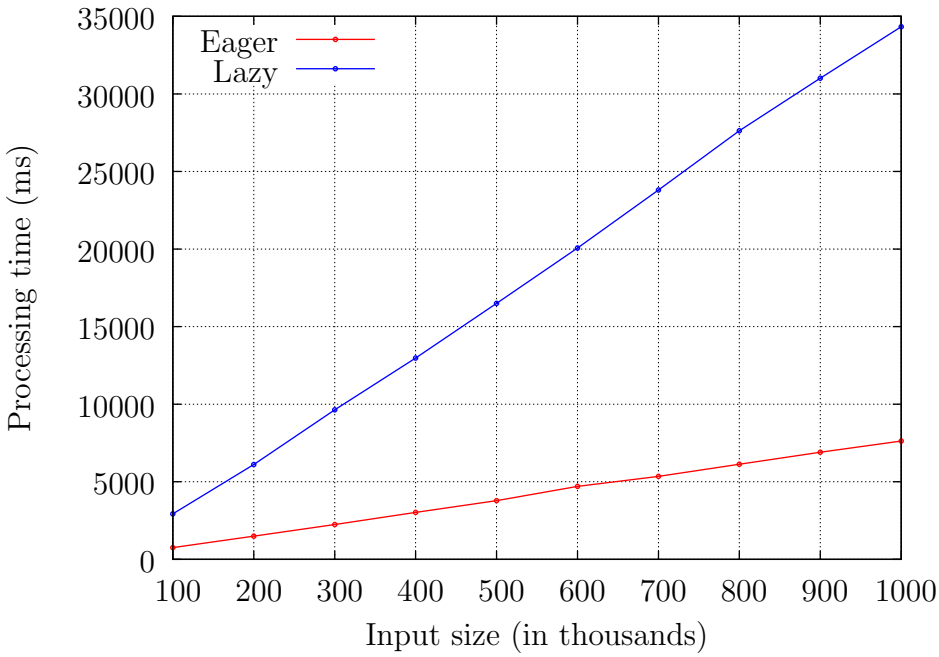
We ran the test 10 times over each the lazy and the eager version. The input parameter varied between 100.000 en 1.000.000 and was incremented in steps of 100.000. The batch

script that was used to benchmark the programs can be found in **Assignment 1.1/SieveTester.bat**. The data can be found in **EagerSieve.dat** and **LazySieve.dat**.

Mergesort

The lazy and eager version both contain a procedure(Split). As it is a procedure this is always eager. Every function in the lazy version is again lazy and there is an extra touch function as in the Sieve exercise. Because of this the time complexity of both versions is the same, but the lazy is slower because of the introduced overhead. So again, the eager version is preferred.

We can see the results of the benchmark beneath.



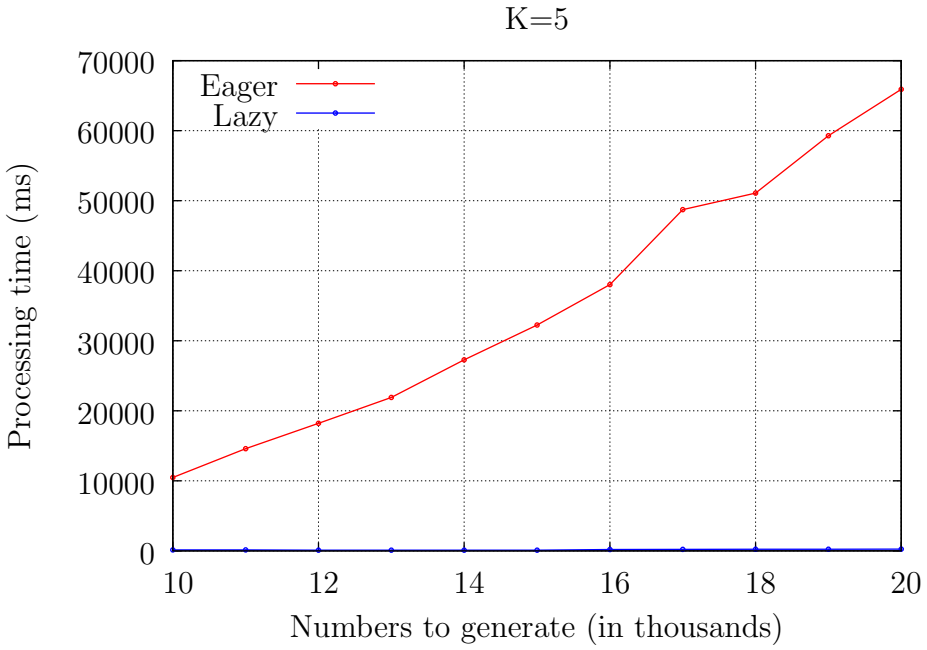
We ran the test 10 times over each the lazy and the eager version. The input parameter varied between 100.000 en 1.000.000 and was incremented in steps of 100.000. The batch script that was used to benchmark the programs can be found in **Assignment 1.1/SortTester.bat**. The data can be found in **EagerSort.dat** and **LazySort.dat**.

Assignment 1.2 The Hamming problem

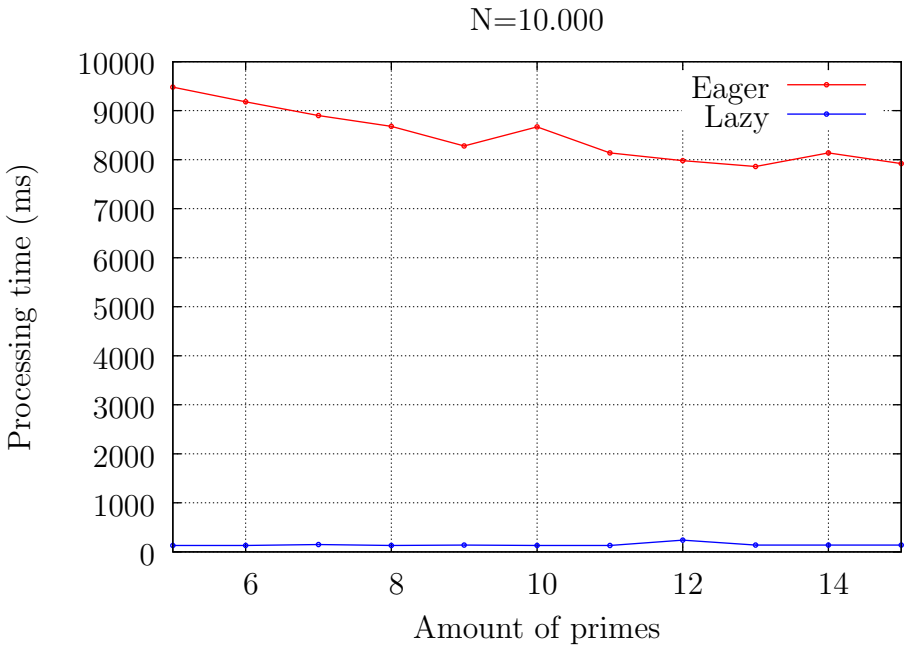
The lazy version is a more general version of the Hamming problem which is solved in the book (page 293-294). This solution makes use of an infinite stream H. An eager version for this problem can't use an infinite list which forces us to adjust the algorithm. This is why eager and lazy aren't completely comparable.
The eager version uses some lazyness. The first K prime numbers are calculated lazily (as in the lazy version) using the Sieve of Eratosthenes we wrote in the previous assignment. After that though, everything is eager. The way we solve this problem is as follows:

- Calculate the first K primes.
- While we don't have N elements in H (the current resultset): loop over every prime number x in the set of K prime numbers.
- Loop over the elements of H until you find the first element y which is bigger than the last element in H when multiplied with x. Return x * y.
- Save these minima for every prime.
- Calculate the minimum of these minima and add it to H. Start over from the first step until we have enough elements.

I ran one benchmark which went over the lazy and eager version a 100 times each. We observed how changing N(the amount of numbers to generate) and K(the amount of primes to use) influenced the processing time of the program. N ranged between 10.000 and 20.000 (which I increased in steps of 1.000) while K ranged between 5 and 15 (which I increased in steps of 1). We used some of this data to produce the graphs below.



For this graph we let N vary between 10.000 and 20.000 while K held the constant value 5.



For this graph we let K vary between 5 and 15 while N held the constant value 10.000.

We see that eager version is always a lot slower. This is because it isn't able to make use of the optimized algorithm. We can also observe that the processing time gets reduced

when we use more prime numbers.

We can conclude that the lazy version is a lot faster (because it can make use of the infinite stream) and is easier to write. Because of this the lazy version is preferred over the eager version. The lazy version is able to complete the task in $O(N)$ time. The eager version on the other hand does it in $O(N * (K * log(N)))$.

The batch script that was used to benchmark the programs can be found in **Assignment 1.1/HammingTester.bat**. The data can be found in the folder **Assignment 1.2** in the files **EagerHamming.dat** and **LazyHamming.dat**. The data for the graphs can be found in the same folder in the files **LazyHammingCstN.dat**, **LazyHammingCstK.dat**, **EagerHammingCstN.dat** and **EagerHammingCstK.dat**

Assignment 2 Declarative model and concurrency

For this assignment I turned the sudokus into csv's. Every cage gets appointed a character. For each element in the sudoku I write down the right character and a comma. This creates a file of 81 characters seperated by comma's, representing the sudoku. After the sudoku I write down the character of every cage and it's number, again seperated by commas. You end up with one very long, comma seperated line which can be used to let the programs solve the sudoku.

You can find these Sudoku files in **Assignment 2.1/Sudoku.csv**, **Assignment 2.2/Sudoku1.csv** and **Assignment 2.2/Sudoku2.csv**.

Assignment 2.1 Declarative model

This program was written in Oz. The way I solved this was by using Oz' constraints. Using this makes it quite easy to solve the KillerSudoku. We first parse the csv into a 9x9 matrix containing the sudoku and then parse the values of the cages and save those

in a list of tuples. After parsing, we create a new 9x9 matrix using **FD.list**. We add the regular sudoku constraints. The first of these constraints is that the numbers in a sudoku varie between 1-9. The second is that no two numbers in the same row,column or box have the same value. For this we use **FD.distinct**. The only thing that's different about a KillerSudoku is that it has cages. So we add a constraint for these cages. We use **FD.sum** in combination with the Sudoku matrix and the list of sum tuples to pull this off.

The solution is actually found within a fraction of a second but for completeness we did a test. This test can be found in **Assignment 2.2/KillerSudokuTester.bat**. After running the test we know that the solution is computed within 50 ms.
The solution we found is the folowing:

5	8	1	3	4	2	9	7	6
4	3	6	5	7	9	8	1	2
7	9	2	8	6	1	4	5	3
6	4	5	7	1	3	2	8	9
3	7	9	4	2	8	1	6	5
2	1	8	6	9	5	3	4	7
8	6	4	2	3	7	5	9	1
9	5	3	1	8	6	7	2	4
1	2	7	9	5	4	6	3	8

We are able to conclude that using constraints in Oz was the right choice for this problem. It was not too hard to implement and is very fast (50ms). Writing this in a language like Java would have taken a lot longer as it would've probably needed backtracking.

Assignment 2.2 Introducing concurrency

The concurrent model we used was the declarative concurrent model. The reason I chose this model is because it is easy to implement. The other models would have been a lot more work to implement and I don't think they would've have made the computation any faster.

The program runs two searches, each in it's seperate thread, and makes use of the dataflow variable **Sol** to communicate between these threads. Unification of the threads is done by the barrier defined in the book on page 277.

Sadly, I was unable to produce a solution using my concurrent program. I am also unable to conclude if this is because of a programming error or just because the problem takes too long to solve.
Even though my concurrent program didn't work, I was able to find the solution to the linked sudoku killer by using the program of the previous assignment. The solution I found is the following:

6	5	8	7	2	1	3	4	9
3	9	1	4	5	6	7	8	2
4	7	2	3	8	9	5	6	1
2	6	4	5	9	3	1	7	8
1	3	7	6	4	8	9	2	5
5	8	9	1	7	2	4	3	6
7	2	3	8	1	5	6	9	4
8	1	6	9	3	4	2	5	7
9	4	5	2	6	7	8	1	3