DSSC - Algorithmic Design

Homework 2

Patrick Indri

May 31, 2019

Chain Matrix Multiplication

The `code/` folder contains the implementation of the Chain Matrix Multiplication algorithm. `chain.c` implements a bottom up approach to determine the best parenthesization for the chain multiplication of a series of n matrices. Denoting such chain multiplication as $A_{1...n}$ the matrix `M` is defined as matrix where `M[i][j]` is the minimum number of scalar products needed to compute $A_{i...j}$. `S` denotes the matrix whose entries `S[i][j]` record the value for the best split of $A_{i...j}$. `MatrixChain` computes both matrices, printing and returning `S`. `PrintParens` exploits the info in `S` to print the optimal solution while `ChainMult` performs the actual computation.

Compile and run

The `makefile` provided can be used to compile the code. Using `make` or `make all` the code gets compiled in a verbose version `chain_test.x`; its output is shown in Lst. 2. Running `make benchmark` results in the compilation of `benchmark.x` which provides a cleaner output, ready to be plotted. Both executables read the sizes of the matrices from `input.txt`, and accept the desired length of the chain as argument. It should be noted that the provided input file supports up to 40 matrices (additional sizes can be added to `input.txt`). By default, the code prints a warning message if the desired length is larger than 40. `run.sh` can be used to perform a benchmark of the code.

Implementation and results

The algorithm has been implemented exploiting the `struct INT_MATRIX` which holds the pointers to the elements of the matrix and the matrix dimensions. Using integers as elements avoids the computational error which might derive from a long sequence of float multiplications. A general approach to actually compute the optimal solution has been implemented: the structure of such a general approach [1] is presented in Lst. 1.

Listing 1: Chain Multiplication Computation

```
Mult(i, j) {
if (i == j) return A[i];
  else {
    k = S[i,j];          // Identifies best split.
    X = Mult(i,k];       // X = A[i]...A[k].
    Y = Mult(k +1, j];   // Y = A[k+1]..A[j].
    return XY;           // Multiply matrices X and Y.

  }
}
```

The implementation has been tested on several instances of the problem, comparing the chain matrix algorithm with a naive, sequential multiplication. Lst. 2 shows the output for a chain of lenght 6.

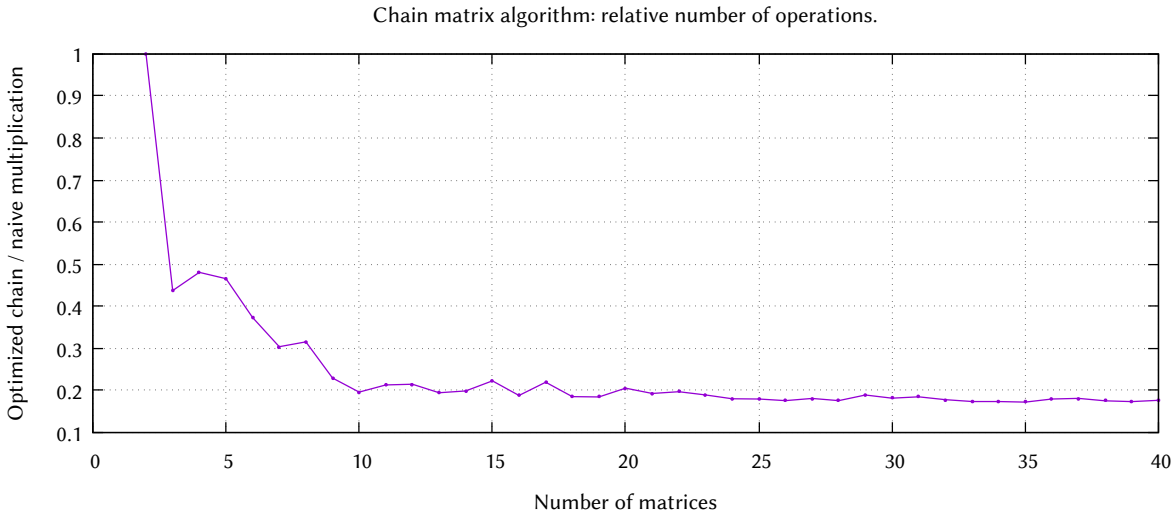
Listing 2: Output of the code for the test matrix chain

```
Number of matrices in the chain, n = 6

Matrix S:
1       1       3       3       3
0       2       3       3       3
0       0       3       3       3
0       0       0       4       5
0       0       0       0       5

Printing optimal solution:      ((A1(A2A3))((A4A5)A6))

NAIVE SOLUTION
Execution time: 0.000716s
Number of operations: 40500

CHAIN SOLUTION
Execution time: 0.000314s
Number of operations: 15125
```

```
19
20 Operations opt/naive: 0.373457
21
22 Is the result the same? 1
```

The code outputs the matrix `S` and the optimal parenthesization. On this test instance, the chain matrix algorithm shows a clear improvement in execution time and number of operation. Comparing the two results element by element, the correctness of the computation has been verified as well.

The script `run.sh` compiles `benchmark.x` and runs it on chains going from 2 up to 40 matrices. The resulting plot is shown in the following figure.



As expected, the optimised and the naive implementations perform the same for two matrices. However, increasing the length of the chain, the chain matrix algorithm shows an increasingly better performance, stabilising at $\approx 20\%$ the number of operations of the naive implementation.

References

[1] R. B. Muhammad. *Matrix Chain Multplication Problem*. 18.03.2010. URL: http://personal.kent.edu/%7Ermuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm.