HPC Assignment: Matrix Chain Multiplication

2024-03-17

1. Introduction and objective

It is common knowledge that, although matrix multiplication is not a commutative binary operation, it is associative. However, although due to said associativity $A \cdot (B \cdot C) = (A \cdot B) \cdot C$, depending on the dimensions of the matrices the total number of operations required to operate each side of the equality can be quite different.

The number of operations required to multiply to matrices with shapes $n \times p$ and $p \times q$ is npq. Thus, when given a sequences of non-square matrices, computing the product in different orders might entail vastly different computational complexities.

The problem of finding the best order to compute the product of a sequence of matrices is known as matrix chain multiplication. A direct solution to this problem is simply computing the complexity of every single possible order and selecting the order yielding the smallest one. Unfortunately, as the size of the sequence of matrices grows, so does the number of possibles order. In fact, its growth is given by the Catalan numbers $C_n = \frac{1}{n+1}\binom{2n}{n}$, that is, the growth is asymptotically exponential. To avoid such a wildly suboptimal solution, other approaches have been proposed with the years, achieving a complexities of $O(n^3)$ or even as low as $O(n \log n)$.

The objective of this project is to benchmark different implementations of the product of five matrices, and implement at least one of the well-known solutions to this optimization problem using `C++` backend through the `RcppArmadillo` library.

2. Task 1: Different orders lead to different complexities

To illustrate how different the total number of operations can be when altering the altering the order of a product of matrices, we will consider the following five matrices: $A(50 \times 6)$, $B(6 \times 45)$, $C(45 \times 10)$, $C(45 \times 10)$, $D(10 \times 15)$ and $E(15 \times 30)$.

```r
set.seed(100505652) # for reproducibility
# matrices with whole numbers to avoid problems
# with precision on floats
A = matrix(sample(1:5,size=50*6,replace=T),nrow=50,ncol=6)
B = matrix(sample(1:5,size=6*45,replace=T),nrow=6,ncol=45)
C = matrix(sample(1:5,size=45*10,replace=T),nrow=45,ncol=10)
D = matrix(sample(1:5,size=10*15,replace=T),nrow=10,ncol=15)
E = matrix(sample(1:5,size=15*30,replace=T),nrow=15,ncol=30)
```

First, let compare with a benchmark two different parenthesizations of the product: the naïve $A \cdot B \cdot C \cdot D \cdot E = (((A \cdot B) \cdot C) \cdot D) \cdot E$ and $(A \cdot (B \cdot C)) \cdot (D \cdot E)$.

```r
microbenchmark(
  "Naïve" = A %*% B %*% C %*% D %*% E,
  "Parenthesized" = (A %*% (B %*% C)) %*% (D %*% E),
  times=2e2
)
```

```
## Unit: microseconds
##           expr  min   lq    mean median    uq   max neval cld
##          Naïve 30.3 36.0 43.8955  37.80 44.55 181.5   200  a 
##  Parenthesized 15.4 16.6 24.5765  17.45 21.65 121.2   200   b
```

As can be seen from the output of the benchmark, the second parenthesization takes about 50% less time than the naïve one. To find the reason behind this, we can compute bny hand the number of operations that each parenthesization requires. For the naïve product:

$$\underbrace{50 \cdot 6 \cdot 45}_{A \cdot B} + \underbrace{50 \cdot 45 \cdot 10}_{(AB) \cdot C} + \underbrace{50 \cdot 10 \cdot 15}_{(ABC) \cdot D} + \underbrace{50 \cdot 15 \cdot 30}_{(ABCD) \cdot E} = 66000$$

For the second option:

$$\underbrace{6 \cdot 45 \cdot 10}_{B \cdot C} + \underbrace{50 \cdot 6 \cdot 10}_{A \cdot (BC)} + \underbrace{10 \cdot 15 \cdot 30}_{D \cdot E} + \underbrace{50 \cdot 10 \cdot 30}_{(ABC) \cdot (DE)} = 25200$$

As we can see, the second parenthesization requires about $\frac{25200}{66000} \approx 38\%$ less operations. From the benchmark, we can see that the average time of the second parenthesization is only 50% smaller instead of 72%, although the ratio between the maximum times is 39%. This probably indicates that this computation with the R language might not be as direct and might require some other underlying processing.

3. Task 2: `Rcpp` for naïve product

In an attempt to speed the product of our matrices, an `Rcpp` function can be developed. First, we will start by simply creating a function that multiplies the matrices in the naïve order.

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
arma::mat naiveProdCpp(arma::mat A, arma::mat B, arma::mat C,
                       arma::mat D, arma::mat E){
  // compute the product of 5 matrices in the naïve order
  return A*B*C*D*E;
}
```

The naïve parenthesization computed with this new function can be benchmarked against the ones computed in R.

```r
microbenchmark(
  "Naive-R" = A %*% B %*% C %*% D %*% E,
  "Parenthesized-R" = (A %*% (B %*% C)) %*% (D %*% E),
  "Naive-RcppArma" = naiveProdCpp(A,B,C,D,E),
  times=2e2
)
```

```
## Unit: microseconds
##             expr  min    lq    mean median   uq    max neval cld
##          Naive-R 28.0 31.45 40.6610   35.6 38.2  200.5   200   a
##  Parenthesized-R 14.0 15.75 20.0600   16.8 17.4  101.6   200   a
##   Naive-RcppArma 17.3 19.55 68.9225   20.2 21.1 8624.5   200   a
```

On average, the naïve parenthesization coded with `Rcpp` is slightly faster than the naïve product computed with R and much slower than the other parenthesization. This new function does not seem too consistent however, since in the worst case scenario has been as slow as the R naïve implementation and in the best case scenario it has proven to be only slightly slower than the R implementation with the alternative parenthesization.

4. Task 3: `Rcpp` to multiply in a given order

So far we have tried two different parenthesization, but it might be convenient to have a function that given a list of matrices computes their product in some order specified by a given vector. This new function will be implemented using `RcppArmadillo`.

The most delicate aspect when designing this function is the format of the vector that specifies the order. Given a sequence of n matrices, there are $n-1$ matrix product operations between them. This can be pictured as choosing without replacement $n-1$ numbers from 1 to $n-1$. Furthermore, once $n-2$ products have been computed, there are only two matrices remaining, and so there is only one possible product left. This means that the order of the product can be completely specified using a vector of length $n-2$.

The $n-2$-vector that this function will receive as input will contain the positions of the products within the expression $A_1 \cdot \ldots \cdot A_n$ in the order in which they are to be performed. That is, taking the previously used parenthesization $(A \cdot (B \cdot C)) \cdot (D \cdot E)$, within $A \cdot B \cdot C \cdot D \cdot E$, $B \cdot C$ would be represented by 2, $A \cdot (B \cdot C)$ would be represented by 1, $D \cdot E$ would be represented by 4, and finally $(A \cdot (B \cdot C)) \cdot (D \cdot E)$ would be represented by 3. Since the last product can be safely ignored, the vector that would represent this parenthesization within the function would be $(2, 1, 4)$.

Having established the format of the vector, the function will receive the list of matrices and iteratively perform their product in the specified order. In each iteration, the list of matrices will be altered so that if at the start it is `list(A,B,C,D,E)`, after the first product it would be `list(A,B·C,D,E)`. The orders specified by the vector will have to be updated accordingly after every product is performed, since the total number of products will have been reduced by 1.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
arma::mat orderedProd(List matrixList, arma::colvec orderVec){

  // matrix output
  arma::mat out;

  // left matrix of the product
  arma::mat matL;
  // right matrix of the product
  arma::mat matR;

  // deep copy to safely manipulate the list
  List listCopy = clone(matrixList);

  for (size_t i=0; i < orderVec.size(); i++){
    // index of the i-th product operation
    int prodIndex = orderVec[i];

    // matrices (in order) of the i-th product
```

```
    matL = as<arma::mat>(listCopy[prodIndex-1]); // extracting from list is an S4 object
    matR = as<arma::mat>(listCopy[prodIndex]);   // so need to transform to arma::mat

    out = matL * matR; // perform the product

    listCopy[prodIndex-1] = wrap(out); // update the remaining matrices

    listCopy.erase(prodIndex); // drop the already-computed product

    for (size_t j=i; j < orderVec.size(); j++){
      if (orderVec[j] >= listCopy.length()){
        // since there are one less matrix and one less product now
        // for remaining products to adapt to this
        orderVec[j] = orderVec[j] - 1;
      }
    }
  }

  // final two matrices
  matL = as<arma::mat>(listCopy[0]);
  matR = as<arma::mat>(listCopy[1]);

  // final product
  out = matL * matR;
  return out;
}
```

Before doing anything else, let us first check that the function performs the product correctly:

```
unique(as.vector(unique(orderedProd(list(A,B,C,D,E),c(2,1,4)) == A %*% B %*% C %*% D %*% E)))
```

```
## [1] TRUE
```

Finally, the new function can be benchmarked using the better parenthesization found earlier against the same product order in R:

```
microbenchmark(
  "Parenthesized-R" = (A %*% (B %*% C)) %*% (D %*% E),
  "Parenthesized-Rcpp" = orderedProd(list(A,B,C,D,E),c(2,1,4)),
  times=2e2
)
```

```
## Unit: microseconds
##                expr  min    lq    mean median    uq    max neval cld
##     Parenthesized-R 13.8 15.10 21.7810  17.00 18.20   95.8   200  a
##  Parenthesized-Rcpp 19.6 21.85 36.4025  23.75 33.25 1255.5   200   b
```

We can see from this benchmark that coding in `Rcpp`does not guarantee better performance, since our new function is slower than the R implementation. One reason the new function might be slower than the R implementation is because of the number of operations related with memory that it does, such as cloning the list of matrices, removing an element from the list in each step, updating the vector with the order through a nested `for` loop… This also hints that the performance of matrix products in R probably already has backend in a compiled language.

5. Task 4: Dynamic algorithm

Finally, an algorithm that solves the Matrix Chain Multiplication problem will be implemented. More specifically the dynamic approach, with complexity $O(n^3)$, has been chosen due to its good complexity-simplicity relation.

This algorithm starts by considering the matrix chain multiplication in a recursive way, that is, if $A_i A_{i+1} \dots A_j$ is a chain multiplication of matrices, each of size $p_{i-1} \times p_i$, any of its optimal parenthesizations divides it between A_k and A_{k+1} in subchains $A_i \dots A_k$ and $A_{k+1} \dots A_j$ that must be optimal themselves. Furthermore, the optimal cost of computing $A_i A_{i+1} \dots A_j$ must be the cost of computing $A_i \dots A_k$ plus the cost of computing $A_{k+1} \dots A_j$ plus the cost of computing the product of the two subchains. In this way, an optimal to solution to the multiplication of the initial chain can be built by obtaining optimal solutions to its subchains, and this can be repeated recursively.

Let $m \in \mathcal{M}^{n \times n}$ such that $m_{i,j}$ is the minimum number of scalar multiplications needed to compute $A_i \dots A_j$. Then $m_{i,j}$ can be defined as follows:

$$m_{i,j} = m_{i,k} + m_{k+1,j} + p_i p_k p_j,\ i < j$$

which is nothing more than the cost of computing $A_i \dots A_k$, $A_{k+1} \dots A_j$ and their product.

Although in this last expression k is not known, there are $j - i$ values for it, and so they can all be checked to find the optimal spot for parenthesization of the $A_i \dots A_j$ chain. In summary:

$$m_{i,k} = \begin{cases} 0 \quad \text{if} i = j \\ min_{i \leq k < j}\{m_{i,j} + m_{k+1,j} + p_{i-1} p_k p_j\} \quad \text{if} i < j \end{cases} \tag{1}$$

At the same time, another matrix $s \in \mathcal{M}^{n \times n}$ is defined, such that $s_{i,j}$ contains the value of k which yields the optimal parenthesization of $A_i \dots A_j$.

The algorithm computes the optimal cost of the parenthesization in a bottom-up manner, that is, calculating the optimal cost of subchains of increasing length until the whole chain has been reached.

Once the algorithm has finished, the optimal cost is contained in $m_{1,n}$, wheras the optimal parenthesization is contained in the first row of the s matrix.

5.1. R implementation

First, we can implement this algorithm using R:

```r
optimalOrderR = function(matrixList){
  # number of matrices in the chain
  n = length(matrixList)
  # initialize matrices
  # m[i,j] = minimum number of scalar multiplications in A_i···A_J
  m = matrix(0,n,n)
  # s[i,j] = optimal split to compute A_i···A_j
  s = matrix(0,n,n)

  # vector to keep track of the dimensions of the matrices
  p = numeric(length = n+1)

  for (j in 1:n){
    # only the rows matter for the first n-1 matrices
    p[j] = nrow(matrixList[[j]])
  }
```

```r
  # for the n-th matrices, the columns also matter
  p[n+1] = ncol(matrixList[[n]])

  # len is the length of the subchains
  # len=1 is trivial since m[i,i]=0
  for (len in 2:n){
    # i is the first matrix of the subchain
    for (i in 1:(n-len+1)){
      # j is the last matrix of the subchain
      j = i+len-1
      m[i,j] = Inf
      # k is the splitting position
      for (k in i:(j-1)){
        cost = m[i,k] + m[k+1,j] + p[i]*p[k+1]*p[j+1]
        if (cost < m[i,j]){
          m[i,j] = cost
          s[i,j] = k
        }
      }
    }
  }

  return(as.matrix(t(s[1,])))
}
```

```r
optimalOrderR(list(A,B,C,D,E))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    1    1    1    1
```

5.2. Rcpp implementation

This same algorithm can also be implemented using `RcppArmadillo`:

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
arma::mat optimalOrderArma(List matrixList){
  // n is the number of matrices in the chain
  int n = matrixList.length();
  int cost;

  // m(i,j) = minimum number of scalar multiplications in A_i···A_J
  arma::mat m;
  // s(i,j) = optimal split to compute A_i···A_j
  arma::mat s;

  // the diagonal and lower triangle are either unimportant or trivial
  m.zeros(n,n);
  s.zeros(n,n);

  // vector that contains the dimensions of the matrices
```

```
  IntegerVector p(n+1);

  // for the first n-1 matrices only the rows matter
  for (int j=0; j < n; j++){
    arma::mat mat = matrixList[j];
    p(j) = mat.n_rows;
  }
  // for the last matrix both rows and columns matter
  arma::mat mat = matrixList[n-1];
  p(n) = mat.n_cols;

  // length of the subchain
  for (int len=2; len <= n; len++){

    // first matrix of the subchain
    for (int i=0; i <= n-len; i++){
      // last matrix of the subchain
      int j = i+len-1;
      m(i,j) = arma::datum::inf;

      // position of the split
      for (int k=i; k <= j-1; k++){
        cost = m(i,k) + m(k+1,j) + p(i)*p(k+1)*p(j+1);
        if (cost < m(i,j)){
          m(i,j) = cost;
          s(i,j) = k+1;
          }
      }
    }
  }

  return s.row(0);
}
```

```
optimalOrderArma(list(A,B,C,D,E))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    1    1    1    1
```

5.3. Comparison and analysis

As we can see, both implementations yield exactly the same result for the parenthesization. Since $s_{1,j}$ gives the optimal split for the $A_1 \ldots A_j$ chain, we can start with $s_{1,5}$ and go backwards. $s_{1,5}$ indicates that the last split should be $A \cdot (B \cdot C \cdot D \cdot E)$. $s_{1,4}$ tells that the next one should be $A \cdot ((B \cdot C \cdot D) \cdot E)$, wheras $s_{1,3}$ says that $A \cdot (((B \cdot C() \cdot D) \cdot E)$ would come after. Since the next splitting would only lead to an isolated matrix, it is not necessary to consider it, and so, with the notation used before, the optimal order would be $(2, 3, 4)$.

Firstly, we compare the performance of both implementations when obtaining the optimal solution:

```
microbenchmark(
  "OptimalOrder-R" = optimalOrderR(list(A,B,C,D,E)),
  "OptimalOrder-Rcpp" = optimalOrderArma(list(A,B,C,D,E)),
  times=2e2
)
```

```
## Unit: microseconds
```

```
##               expr  min   lq   mean median   uq   max neval cld
##     OptimalOrder-R 21.3 22.2 23.254  22.70 23.2  58.3   200  a 
##  OptimalOrder-Rcpp  5.9  6.4 11.995   6.85  7.4 933.9   200   b
```

As we can see, the `RcppArmadillo` implementation is much faster than the R one, as is to be expected.

We can also bechmark this new optimal parenthesization with the rest of parenthesization tried earlier:

```r
microbenchmark(
  "Naive-R" = A %*% B %*% C %*% D %*% E,
  "Naive-RcppArma" = naiveProdCpp(A,B,C,D,E),
  "Parenthesized-R" = (A %*% (B %*% C)) %*% (D %*% E),
  "Parenthesized-RcppArma" = orderedProd(list(A,B,C,D,E),c(2,1,4)),
  "Optimal-R" = A %*% (((B %*% C) %*% D) %*% E),
  "Optimal-RcppArma" = orderedProd(list(A,B,C,D,E), c(2,3,4)),
  times=2e2
)
```

```
## Unit: microseconds
##                    expr  min    lq    mean median    uq    max neval cld
##                 Naive-R 28.1 32.30 39.1180  35.50 37.35  107.1   200   a
##          Naive-RcppArma 17.4 19.25 23.9225  20.40 21.35  112.5   200   a
##         Parenthesized-R 14.1 15.50 20.4505  16.65 17.15   95.5   200   a
##  Parenthesized-RcppArma 20.0 22.40 60.3630  23.50 25.45 5750.1   200   a
##               Optimal-R 10.6 12.20 16.5390  13.40 13.95   87.9   200   a
##        Optimal-RcppArma 17.6 20.00 26.3400  21.00 22.50  106.2   200   a
```

As we could have predicted from our results in earlier sections, the R direct product with the optimal ordering is the fasted by far, followed by the non-naïve parenthesization also in R, and then by the optimal parenthesization implemented with `RcppArmadillo`.

It is worth mentioning the existence of an algorithm developed by Hu and Shing that achieves a computational complexity of $O(n \log n)$ using an equivalence between the problem of finding the optimal parenthesization of a chain of matrices and the problem of triangulation of a regular polygon, which if it were implemented would be much quicker than any of the options implemented in this project.

References

Cormen, Thomas H; Leiserson, Charles E; Rivest, Ronald L; Stein, Clifford (2001). "15.2: Matrix-chain multiplication". Introduction to Algorithms. Vol. Second Edition. MIT Press and McGraw-Hill. pp. 331–338. ISBN 978-0-262-03293-3.

Hu, T. C.; Shing, M.-T. (1984). "Computation of Matrix Chain Products, Part II" (PDF). SIAM Journal on Computing. 13 (2): 228–251. doi:10.1137/0213017.