HPC Project

Perpetually Tired

2025-03-16

Setup

Let's first generate a bunch of matrices to test stuff on.

```
A=matrix(rexp(200, rate=.1), ncol=5)
B=matrix(rexp(400, rate=.1), ncol=80)
C=matrix(rexp(800, rate=.1), ncol=10)
D=matrix(rexp(600, rate=.1), ncol=60)
E=matrix(rexp(600, rate=.1), ncol=10)
m_list = list(A,B,C,D,E)
lapply(m_list, dim)
```

```
## [[1]]
## [1] 40  5
##
## [[2]]
## [1]  5 80
##
## [[3]]
## [1] 80 10
##
## [[4]]
## [1] 10 60
##
## [[5]]
## [1] 60 10
```

Now let's try comparing the default way, a pretty alright way, and a terrible way of multiplying them.

```
pathA= function(){A%*%B%*%C%*%D%*%E} #standard order
pathB= function(){A%*%(B%*%C)%*%D%*%E} #pretty good
pathC= function(){(A%*%B)%*%(C%*%D)%*%E} #very innefficient


benchmark(pathA(),
          pathB(),
          pathC(),
          order="relative", replications=10000)
```

```
##      test replications elapsed relative user.self sys.self user.child sys.child
## 2 pathB()         10000     0.91     1.000       0.77      0.14           NA         NA
## 1 pathA()         10000     1.60     1.758       1.35      0.25           NA         NA
## 3 pathC()         10000     3.78     4.154       3.25      0.53           NA         NA
```

We can see a good amount of difference between the three paths, particularly between the most and least efficient, with the default path performing a little worse than the fastest.

We can calculate the operations being done in both cases:

```
length_A =
40* 5* 80+
40* 80* 10+
40* 10* 60+
40* 60* 10

length_B =
40* 5* 80+
5* 80* 10+
5* 10* 60+
5* 60* 10

length_C =
40* 5* 80+
40* 80* 60+
80* 10* 60+
40* 60* 10

c(length_A,length_B,length_C)
```

```
## [1]  96000  26000 280000
```

And see that path B had vastly fewer operations than both A and C.

Applying C++

Now let's try it with some RcppArmadillo functions.

```
// [[Rcpp::export]]
mat basicmult(List X) {
  mat out = as<arma::mat>(X[0]);
  for (int i = 1; i<X.size();i++){ //start at 1 so we can skip out 0 starting index
    out =out*as<arma::mat>(X[i]);
  }
  return out;
}

// [[Rcpp::export]]
mat simplemult(List X) {
  mat out = as<arma::mat>(X[0]) * as<arma::mat>(X[1]) * as<arma::mat>(X[2]) * as<arma::mat>(X[3]) * as<a
  return out;
}
```

```
// [[Rcpp::export]]
mat evensimpler(const arma::mat A,const arma::mat B,const arma::mat C,const arma::mat D,const arma::mat
  mat out = A * B * C * D * E;
  return out;
}
```

```
benchmark(pathA(),
          pathB(),
          pathC(),
          evensimpler(A,B,C,D,E), #basic * operations in a row
          simplemult(m_list), #vector selecting and * in a row
          basicmult(m_list), #loop through vector in order and multiply
          order="relative", replications=10000)
```

```
##                          test replications elapsed relative user.self sys.self
## 5          simplemult(m_list)        10000    0.58    1.000      0.56     0.02
## 4 evensimpler(A, B, C, D, E)        10000    0.60    1.034      0.57     0.03
## 2                     pathB()        10000    0.89    1.534      0.75     0.15
## 6           basicmult(m_list)        10000    1.14    1.966      1.12     0.02
## 1                     pathA()        10000    1.59    2.741      1.31     0.28
## 3                     pathC()        10000    3.73    6.431      3.19     0.54
##   user.child sys.child
## 5         NA        NA
## 4         NA        NA
## 2         NA        NA
## 6         NA        NA
## 1         NA        NA
## 3         NA        NA
```

Now let's try it with a vector that dictates the order multiplications should be done in.

```
# Pattern vector
P_vec = c(2,3,1,4,5)
pathB() == order_mult(m_list,P_vec) #incredible
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [2,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [3,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [4,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [5,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [6,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [7,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [8,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
##  [9,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [10,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [11,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [12,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [13,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [14,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [15,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [16,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [17,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [18,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [19,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [20,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [21,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [22,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [23,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [24,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [25,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [26,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [27,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [28,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [29,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [30,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [31,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [32,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [33,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [34,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [35,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [36,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [37,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [38,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [39,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
## [40,] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  TRUE
```

```r
benchmark(pathA(),
          pathB(),
          pathC(),
          simplemult(m_list),
          basicmult(m_list),
          order_mult(m_list,P_vec), #our ordered cpp function
          order="relative", replications=10000)
```

```
##                        test replications elapsed relative user.self sys.self
## 4        simplemult(m_list)        10000    0.56    1.000      0.54     0.02
## 6 order_mult(m_list, P_vec)        10000    0.70    1.250      0.68     0.02
## 2                   pathB()        10000    0.89    1.589      0.81     0.08
## 5         basicmult(m_list)        10000    1.12    2.000      1.13     0.00
## 1                   pathA()        10000    1.56    2.786      1.27     0.29
## 3                   pathC()        10000    3.67    6.554      3.00     0.67
##   user.child sys.child
## 4         NA        NA
## 6         NA        NA
## 2         NA        NA
## 5         NA        NA
## 1         NA        NA
## 3         NA        NA
```

Pretty good overall, `simplemult` still beats it by sheer computational simplicity and speed, but it's close. Besides, `simplemult` requires writing out the multiplications by hand, like a loser.

Algorithm

Let's try implementing a version of the Hu and Shing approach, starting by making our vector of matrix dimensions.

```r
# extremely crude way of making said vector
vec = c()
vec=sapply(m_list,dim)[1,]
vec[6]=10
vec
```

```
## [1] 40  5 80 10 60 10
```

Now lets try making said implementation.

Basically, we write out our matrix dimensions as nodes and the matrices themselves as edges between them, like this:

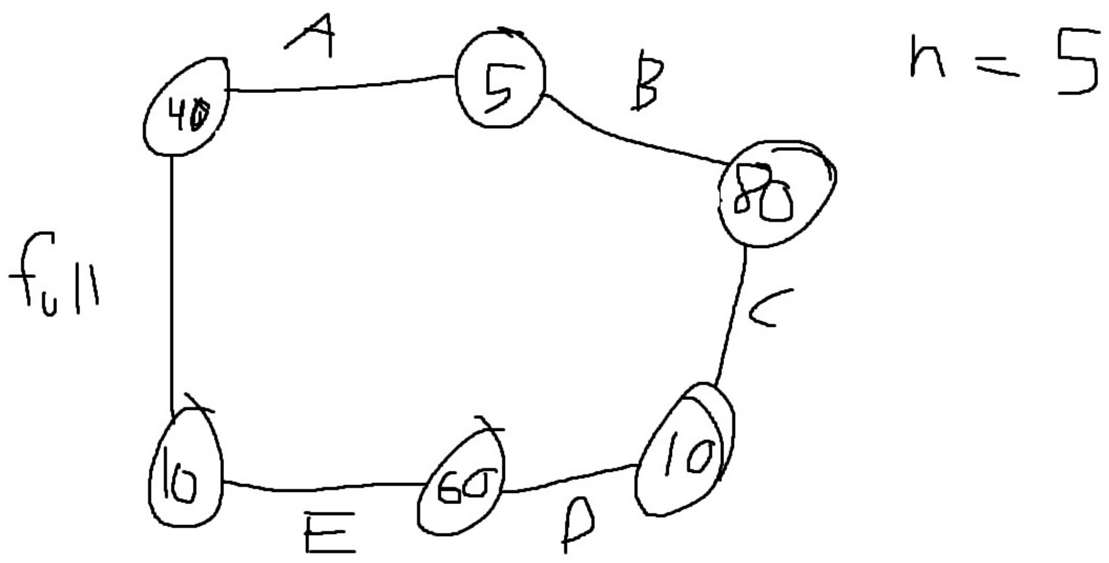

Figure 1: my cool drawing.

We then progressively loop through chain lengths and the nodes that can produce chains that long, so for instance, for length=2, the nodes we'd use as starting points would be 1,2,3 and 4. Inside these loops, we calculate the operation cost and keep the minimum costs we find for any start and end chain. We then progressively increase the length, using the previously saved smaller calculations to get the costs of reaching larger ones.

The diagonal for our operation cost matrix should be all 0's, it doesn't cost anything for a matrix to not be multiplied. Beyond that, we're only going to be using cells above that diagonal. Here's the function:

```r
algorithm = function(nodes){
  n=length(nodes)-1 #how many matrices/edges we have
  costs = matrix(0, n, n) #cost matrix, storing operation cost of each parenthesization
  splits = matrix(0, n, n)  # split matrix stores the optimal split point for the i to j line

  # first, we loop through all the possible chain lengths greater than 1 for our matrix set,
  # for us starting at 2 and going up to 5 (which would just be the pathA parenthesization)
  # len-1 tells you how many multiplications are going to happen inside these parentheses
  # because our matrices are the actual chains/edges and the nodes are just their dimensions,
  # len tells you how many matrices are being multiplied, 2,3,4 or 5, for us

  for (len in 2:n) {

    # then, within each length, we loop through all the nodes that, if set as starting node,
    # could produce a chain of the length set in the above loop
    # our i's are the ( of our parenthesis

    for (i in 1:(n - len + 1)){
      j <- i + len - 1
      costs[i, j] <- Inf #if you don't set it to inf, you can't identify minimums progressively later

      # next, you're looping through all your K+1s, which are your possible "joining" nodes
      # for the multiplication of your i and j nodes
      # for example, if youre multiplying node 1 (40) with node 5 (60), as your joining node,
      # you could have 2,3 or 4, 5,80, or 10, but you still have to account for the cost
      # of being able to use a specific joining node when you choose one.
      for (k in i:(j - 1)) {
        #the operation of cost multiplying your leftmost and rightmost nodes,
        # along with a joining middle one, your k+1
        # the idea is to loop through the possible join nodes and see the cost of each one
        edge_cost = nodes[i] * nodes[k + 1] * nodes[j + 1]

        #the operation cost of the matrices between the leftmost and rightmost
        # if the inner cost[i,j] is longer than 1, you're using the saved minimum cost of getting
        # here from prior loops, that's why it goes smaller to larger, to use those saved smaller
        # costs in the calculation of larger ones.
        path_cost = costs[i, k] + costs[k + 1, j]

        # we then combine the cost of the outermost nodes plus their joining node, and the path it
        # takes to reach that joining node, and see how it compares
        cost = path_cost + edge_cost
        if (cost < costs[i, j]) {
          costs[i, j] = cost
          splits[i, j] = k  # this is what we need for a parenthesizatino vector at the end
          }
        } #looping thru k's
      } #looping thru i's
    } #looping through lens
    return(list(costs = costs, splits = splits))
  }
out = algorithm(vec)
out$costs
```

```
##      [,1]  [,2] [,3]  [,4]  [,5]
## [1,]    0 16000 6000 19000 12000
## [2,]    0    0 4000  7000 10000
## [3,]    0    0    0 48000 14000
## [4,]    0    0    0     0  6000
## [5,]    0    0    0     0     0
```

```
out$splits
```

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

Let's draw out what our parenthesization should be from the splits matrix now:

1 to 5 = 1 so (A) B C D E which we rewrite as A (B C D E) now from 2 to 5 = 4 so A ((B C D) E) now from 2 to 4 = 3 A (((B C) D) E) Let's try it in action

```
pathD= function(){A%*%(((B%*%C)%*%D)%*%E)} #from function

benchmark(pathD(),
          pathB(),
          order="relative", replications=10000)
```

```
##      test replications elapsed relative user.self sys.self user.child sys.child
## 1 pathD()        10000    0.29    1.000      0.25     0.05         NA        NA
## 2 pathB()        10000    0.89    3.069      0.79     0.09         NA        NA
```

The parenthesization we get from the algorithm is vastly faster than even our previously fastest path. Now the question becomes how do we convert this vector into something useable by our C++ function?

```
N_vec = c(2,3,4,5,1) #what our input needs to look like
out$splits # what the output looks like
```

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

Let's write a function that does that.

```
properize = function(splits){
    out = c() #outvector
    d=5 #to loop through out
    matlist = c(1:5) #to keep track of matrices
    sets = NULL # to store subsequent parentheses
    init_split_point = splits[1,5] #should be ncol, but keeping it simple
    for (i in 1:nrow(splits)){ #loop thru rows
    # print(i)
      for (j in ncol(splits):i){ #counting backwards, working to shorter splits, capped by our row
        split = splits[i,j]
        # print(j)
        if(j==i){split =i}

        if(i!=j)out[d] = split+1 else {out[d] = split}
        d = d-1


        # all to check if we need to move to the next row

        right_set = matlist[split+1:j]
        left_set = matlist[i:split]
        if (sum(left_set==1) == 1){ #if i is in left set, that's the i_set, if not, its right
            i_set = left_set
        } else { i_set = right_set}

        if (length(i_set) == 1){
            d = d+1
            out[d] = out[d]-1
            d = d-1
            break # go to next i, u need to go deeper, basically
        }

        if (d==-1){break}
        }
    if (d==-1){break}
    }
    # print(out)
    out
  }
properize(out$splits)
```

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

Now let's wrap them all up in a function and see how they perform.

```
wrapup = function(m_list,nodes){
    splits = algorithm(nodes)$splits #get optimal splits
    vec = properize(splits) #clean into presentable shape
    order_mult(m_list,vec) #perform multiplication
}
# wrapup(m_list,vec) #vec is our vector of nodes/dimensions
```

Here's something awful, the numbers are pretty close, but not exactly identical, so we end up with things like this:

```
order_mult(m_list,P_vec)[1,1] == wrapup(m_list,vec)[1,1]
```

```
## [1] FALSE
```

```r
c(order_mult(m_list,P_vec)[1,1],wrapup(m_list,vec)[1,1])
```

```
## [1] 12217702948 12217702948
```

```r
all.equal(order_mult(m_list,P_vec), wrapup(m_list,vec))
```

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

Ha ha. Terrible.

Let's compare the performace against the others.

```r
benchmark(pathA(),
          pathB(),
          pathD(),
          simplemult(m_list),
          basicmult(m_list),
          order_mult(m_list,P_vec), #our ordered cpp function
          wrapup(m_list,vec), #our fancy rolled up function
          order="relative", replications=10000)
```

```
##                        test replications elapsed relative user.self sys.self
## 3                   pathD()        10000    0.30    1.000      0.20     0.09
## 4        simplemult(m_list)        10000    0.61    2.033      0.59     0.01
## 6 order_mult(m_list, P_vec)        10000    0.70    2.333      0.67     0.01
## 7       wrapup(m_list, vec)        10000    0.70    2.333      0.69     0.02
## 2                   pathB()        10000    0.90    3.000      0.75     0.16
## 5         basicmult(m_list)        10000    1.14    3.800      1.08     0.07
## 1                   pathA()        10000    1.55    5.167      1.34     0.20
##   user.child sys.child
## 3         NA        NA
## 4         NA        NA
## 6         NA        NA
## 7         NA        NA
## 2         NA        NA
## 5         NA        NA
## 1         NA        NA
```

As it stands, `wrapup` was faster than the most basic straightforward multiplication, but nowhere close to the fastest. The overheads for the benefit we get from this algorithm are rather costly, and only really beat the most basic of implementations. That said, except for `simplemult` (which, again, would require you typing each matrix to be multiplied in by hand), all the other functions it lost to already had some for of parenthesization applied to it. That is to say, the function loses against any decent form of manual pre-parenthesization, but if you don't have one at all, it performs pretty well. Additionally, it's likely this overhead is compensated for in larger chains of matrices, where calculating the optimal path by hand becomes impossible and the cost for sub-optimal parenthesization grows.

References

1. Hu, T.C. and Shing, M.T. (1981) Computation of matrix chain products. part I, part II. [Preprint]. doi:10.21236/ada113349.