Code Costs and Implementations

Martin Houde

February 16, 2022

Abstract

Analyzing the different costs in my code to possibly find the optimal number of chunks to break the domain into.

1 Code Costs

My code is broken into two different sections. In the first part, we pre-calculate matrix products of shorter possible domain configurations. A domain of size n has 2^n different possible combinations and for each of them we calculate $n-1$ matrix products. Denoting the cost of matrix multiplication as α, this step costs

$$cost_1 \approx 2^n(n-1)\alpha \tag{1.0.1}$$

In the second step, we loop through a full domain of size N and break it into N/n smaller domains. The smartest thing to do given $\mod(N,n) \neq 0$ is to just calculate the last left over portion at the end. The cost of this step is

$$cost_2 \approx \left(\frac{N}{n} - 1 + \mod(N,n) - 1\right)\alpha \tag{1.0.2}$$

The total cost is then

$$cost \approx \left(2^n(n-1) + \frac{N}{n} - 1 + n - 2\right)\alpha \tag{1.0.3}$$

Where I have taken the maximal cost when $\mod(N,n) = n-1$ giving rise to $n-2$ matrix multiplications.

2 Analysis

Initially, as we increase n the cost decreases, but it doesn't take long before it grow exponentially. I cheat a little bit and treat this as a continuous function. Doing so we can take the derivative and numerically find its roots for different values of N. Here are a few values I've looked at

N	n
200	3
1000	4
3000	5

Next we actually run the code and use the python function $\%timeit$, to evalutate run times. Here we take the full domain to be comprised of $N_{domain} = 1001$ segments. According to the table above, we expect the code to be optimal when we break our calculations into segments of $n = 4$.

3 Further Optimization: Recursion

As is, we are brute-forcing the initial pre-calculations. We could optimize this step by recursively calculating the different combinations. To do so, we simply need to multiply the previous set twice: Once by the matrix for $g(z) = 1$ and another for when $g(z) = -1$. In this sense, the total cost of calculating the n-th step is

$$cost_{rec,n} \approx 2 \cdot 2^{n-1}\alpha + cost_{rec,n-1} \tag{3.0.1}$$

n	$\%timeit$	Normalized Cost	t_{theory}
2	40.5 ± 1.54 s	1	40.5 s
3	28.4 ± 1.28 s	0.695	28.1 s
4	24.5 ± 0.54 s	0.594	24.1 s
5	28.2 ± 1.42 s	0.656	26.6 s
6	41.9 ± 1.08 s	0.97	39.3 s
7	1m18 ± 2.78 s	1.81	1m13

Figure 1: Time cost of running code for different segment lengths n. Second column are the different evaluations of our full code using $\%timeit$. Third column is the time cost according the the expression derived above (Eq. (1.0.3)), normalized such that the cost for $n = 2$ is 1. Fourth column are the theoretical prediction for different times based on normalized cost and starting with the time for $n = 2$

where 2^{n-1} is the number of elements that we need to multiply and we need to add the cost of the previous calculations. By recursion, we find that the total cost for this step simplifies to

$$cost_{rec,n} \approx \left(2^{n+1} - 4\right) \alpha \tag{3.0.2}$$

By direct comparison to $cost_1$, this scales much better as a function of n and we would expect this to further optimize our code.

In fact, if we consider the total recursive cost function to be continuous (take $cost_2$ to be unchanged), we find that $n_{opt} = 5$ for $N = 1001$ and we expect

$$\frac{cost_{n_{opt}}}{cost_{rec,n_{opt}}} \approx 1.14 \tag{3.0.3}$$

and we see that there should be some advantage to recursively generate our pre-calculated set.

3.1 Implementation

We implement two different codes to generate our set of pre-calculated matrix multiplications. The brute-force and recursive method. We then run $\%timeit$ for different values of n and compare both methods. As shown above, the cost of our brute-force method follows our theoretical prediction quite well (Fig. 1). Our brute-force implementation is

```
def product_brute(P,N,n):
    prod = np.ones((2**n,len(P),len(P)),dtype=np.complex128)
    for i, config in enumerate(product((P, N), repeat = n)):
        T = np.eye(len(P),dtype=np.complex128)
        for j in range(n):
            T = config[j] @ T
        prod[i] = T
    return prod
```

where n denotes the shorter domain lengths, P is the Heisenberg propagator for a segment with $g(z) = 1$, and N is the Heisenberg propagator for a segment with $g(z) = -1$. Here we use itertools to generate all the different combinations for a set of n segments and for each configuration we do the prescribed multiplication. Slight drawback to this coding method is that I am also additionally multiplying by the Identity matrix for each configuration (however, I think this multiplication is trivially implemented and doesn't add as much cost).

We implement the recursive method as follows:

```
def product_rec(P,N,n):
    return np.array((P,N)) if n==1
    else np.concatenate([product_rec(P,N,n-1) @ P, product_rec(P,N,n-1) @ N], axis=0)
```

where all inputs are the same. We now use the $\%timeit$ to evaluate the run times of the two different methods for different n's. We always take the total number of domains $N = 1001$

As we can see from the values in Fig. 2, the recursive function does not outperform our brute-force method. In fact, it does increasingly worse for increasing values of n and generally has a

n	$\%timeit_{rec}$	$\%timeit_{brute}$
2	201 ms ± 8.32 ms	305 ms ± 13.7 ms
3	804 ms ± 124 ms	891 ms ± 45.9 ms
4	2.62 s ± 461 ms	2.3 s ± 63.5 ms
5	7.64 s ± 326 ms	5.86 s ± 314 ms
6	18.5 s ± 752 ms	14.4 s ± 823 ms

Figure 2: Time cost of running codes to generate all possible products for different segment lengths n. Second column are the different evaluations of our recursive code using $\%timeit$. Third column are the different evaluations of our recursive code using $\%timeit$.

higher standard deviation than the brute-force method. When we consider the full code, We always find that the recursive method does worse.

4 Further Optimization: Domain Loop

In the second part of our code, we loop over all domain sets of size n and for each set we loop through the possible ways to configure ± 1 n times. Original implementation of this is done as

```
for i in range(0, Nd - remainder, n):
    for j, config in enumerate(product((1, -1), repeat=n)):
        if np.all(domain[i : i + n] == config):
            T = prod[j] @ T
            break
```

The first loop just runs over all domain subsets of size n (which fully divide the total domain of size Nd). The second loop runs over all configurations and finds which configuration the domain subset corresponds to. Once found, we multiply by the relevant Heisenberg propagator (prod[j], generated previously) and we break the loop as there is no reason to keep looping.

The itertool function "product" always generates the configuration such that the first half of them start with $+1$ and the second half start with -1. So if our domain subset starts with -1, we end up looping over half of all possible configurations uselessly. In the worst case scenario where all our domain subsets start with -1, this is a lot of useless looping. We could imagine optimizing by first checking the sign of the first element of the domain subset and then looping over the configurations that start with said sign. We could even imagine doing even better without having to loop over configurations. We can initially create a list of all configuration and then check which index our domain subset corresponds to. This only works if the domain subset is also a list. This is not a problem as we can convert an array to a list quite easily. We can rewrite this portion of the code as

```
x = list(product((1, -1), repeat = n))

for i in range(0, Nd - remainder, n):
    index = x.index(tuple(domain[i : i + n]))
    T = prod[index] @ T
```

We first create a list of the possible configurations. Note that "x" is a list of 2^n tuples of length n. When we loop over our domain subset we then convert said subset into a tuple and use the ".index()" function to find the proper index. Here we are finding which index our domain subset corresponds to and this index is then used to find the corresponding pre-calculated Heisenberg propagator.

We again analyze the time cost of both different methods and compare. The time we save by not looping is minimal (order of 0.1 s), but its does make the code look cleaner and more concise in my opinion.

5 Final Verdict

In the end, we keep the first method of pre-calculating the different possible combination of matrix products. Recursively calculating them led to longer times. In the second portion of the code, we are able to omit one loop by turning things into lists/tuples. Doing so does not give a great advantage but makes the code cleaner(advantage might be better for much larger domains).