A quantum control algorithm: numerical aspects

Philipp Klenze*
JASS 2008

April 17, 2008

Acknowledgment: This paper contains material adapted from the slides of the presentation "Numerical Linear Algebra Tasks in a Quantum Control Problem" by Konrad Waldherr.

Abstract

Various numerical challenges in the quantum control algorithm presented by Mr. Fischer are discussed. Particular regard is given to the efficient calculation of exponentials of sparse matrices and to algorithms for parallel matrix-matrix-multiplications.

*Any comments to this paper should be sent by email to ⟨the authors last name⟩@in.tum.de

Contents

1 Introduction

The quantum control algorithm which was described in depth in the talk and paper of Mr. Fischer offers some interesting numerical challenges if it is to be implemented efficiently.

One iteration step in the Gradient Flow Algorithm goes like this:

- Calculate the forward-propagation for all $t_1, t_2, ..., t_k$:
$$\mathbf{U}(t_k) = e^{-i\Delta t\mathbf{H}_k} \cdot e^{-i\Delta t\mathbf{H}_{k-1}} \cdots e^{-i\Delta t\mathbf{H}_1}$$
- Compute the backward-propagation for all $t_M, t_{M-1}, \ldots, t_k$
$$\mathbf{\Lambda}(t_k) = e^{-i\Delta t\mathbf{H}_k} \cdot e^{-i\Delta t\mathbf{H}_{k+1}} \cdots e^{-i\Delta t\mathbf{H}_M}$$
- Calculate the update
$$\frac{\partial h(\mathbf{U}(t_k))}{\partial u_j} = \mathrm{Re}\left\{\mathrm{tr}\left[\mathbf{\Lambda}^\dagger(t_k)(-i\mathbf{H}_j)\mathbf{U}(t_k)\right]\right\}$$

$\mathbf{H}_k$ are typically sparse matrices with a size of 1024×1024. The resulting exponentials are equally sized, but no longer sparse. M can be assumed to be 128.

Analyzing the algorithm, one discovers that the lion's share of the computing efforts will go to just two tasks:

- the computations of exponentials of sparse matrices
- matrix-matrix-multiplications

As the memory of a single ordinary computer is to small to contain all of the exponentials, special consideration will be given to parallelisation approaches for the second task.

2 Computations of exponentials of sparse matrices

2.1 Definition of the matrix exponential

The matrix exponential of a square matrix in defined using the Taylor series, just like it is defined for numbers:

$$e^{\mathbf{A}} := \sum_{k=0}^{\infty} \frac{\mathbf{A}^k}{k!}$$

It should be noted, however, that the functional equation of the exponential function is no longer generally true. Thus the computation of the matrix exponentials is non-trivial.

In the following part, various strategies to calculate the matrix exponential will be discussed.

2.2 Properties of the exponent matrix

The optimal algorithmic approach is dependent on the type of the argument matrix. In the given case, the matrix $\mathbf{H}$ is *sparse*, most entries are zero. This is the single most important property to choose the right algorithmic approach, as some matrix operations which are efficient for full matrices are more expensive on sparse matrices than other algorithms. [1]

The input matrix $\mathbf{H}$ also has some other exploitable properties. It is *hermitian* and *persymmetric*. With some sophisticated transformations, the problem of calculating the exponential of the complex matrix $\mathbf{H}$ can thus be reduced to the computation of the exponential of two real matrices of half the size.

2.3 Diagonalisation, Eigendecomposition

The exponential of a diagonal matrix

$$\mathbf{D} = \operatorname{diag}(d_1, \ldots, d_n) = \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix}$$

is trivial to calculate:

$$e^{\mathbf{D}} = \operatorname{diag}(e^{d_1}, \ldots, e^{d_n}) = \begin{pmatrix} e^{d_1} & & \\ & \ddots & \\ & & e^{d_n} \end{pmatrix}$$

Furthermore, if $\mathbf{A}$ is $\mathbf{SDS}^{-1}$, with $\mathbf{D}$ being a diagonal matrix, the exponential is also trivial to calculate:

$$e^{\mathbf{A}} = S\left(\operatorname{diag}(e^{d_1}, \ldots, e^{d_n})\right) S^{-1}$$

The decomposition of $\mathbf{A}$ into $\mathbf{SDS}^{-1}$ is called *Eigendecomposition*. Unfortunately, and unsurprisingly, it is quite expensive to calculate the Eigendecomposition of a matrix. Still, this approach is one possible candidate.

2.4 Approximations of the matrix exponential

In this section, different approximations of the matrix exponential are analysed. A naïve approach would be to take a partial sum of the

[1] For example, matrix inversion has the same complexity as matrix-matrix-multiplication only for full matrices. For sparse matrices, the cost for multiplications is significantly decreased while the cost for inversion is not decreased by a similar amount.

TAYLOR series, but due to slow convergence and numerical instability, this is not feasible. Another approximation with a polynomial, namely the CHEBYSHEV series expansion, has much better properties and will be examined more closely.

Furthermore, an approximation using rational functions, the PADÉ approximation will be discussed.

2.4.1 Scaling and Squaring

The quality of the approximations of either one of the two approximations discussed below is strongly dependent on the norm of the exponent being smaller than one. As one needs to compute the exponential of a matrix of arbitrary norm, a method to transform the general case of the problem to the special case for which the approximations work is needed.

Fortunately, such a method exists. It is called *scaling and squaring*.

The following equation is true for the matrix exponential:

$$e^{\mathbf{A}} = \left(e^{\mathbf{A}/2^k}\right)^{2^k}$$

That means that one can multiply the exponent with a factor of $\frac{1}{2^k}$ to decrease its norm, then use the approximation to calculate the exponential, and square that result k times to obtain the exponential.

Of course, these calculations increase both the time needed to run and the numerical error, but they mean that the following two approximations can generally be used.

2.4.2 Chebyshev series expansion

The Chebyshev polynomials form an orthogonal basis in the interval $[-1, 1]$ with a slightly unusual metric. Any well behaved function — particularly the exponential function — can be expanded in this basis. The basic idea of the Chebyshev series expansion is to approximate the exponential function by a partial sum of this expansion. The coefficients decreases as $\frac{1}{2^k k!}$, so even a polynomial of comparably low order can give a very good approximation.

Furthermore, this method can also be used to calculate matrix exponentials, provided that the norm of the argument matrix is smaller than one. Using the "Scaling and Squaring" technique, this prerequisite can be ensured.

As matrix polynomials of sparse matrices are comparably inexpensive to compute, especially when using elaborate multiplication schemes, the efficiency of this algorithm is rather high.

2.4.3 Padé approximation

Like the TAYLOR series expansion, the PADÉ expansion approximates a well-behaved function, like the exponential function, at a single point, which will be zero in the given case. But unlike the Taylor series expansion, the Padé approximation uses a rational function instead of a polynomial to emulate the given function.

As the Taylor method did completely fail in terms of numerical stability, it is somewhat surprising that the Padé method actually works quite well.

Like the Chebyshev method, the Padé method can be generalized to matrices. But there are similar restrictions on the norms of the argument matrices, as the Padé approximation is good only near zero. Thus the "Scaling and Squaring" steps have to be applied before and afterward respectively.

The division is replaced by a matrix inversion when switching from real numbers to matrices. Of course, the inverse matrix, which has rather poor numerical properties, in never calculated explicitly, but rather indirectly by solving a corresponding system of linear equations.

It should be noted that the matrix inversion is more costly than matrix-matrix-multiplications for sparse matrices, which means that the Padé approximation has some initial disadvantage when compared to the Chebyshev method. Whether or not this is balanced by better approximations with smaller polynomials is dependent on the size of the matrices.

2.5 Comparison of the methods

Figures 1 and 2 give an idea of the computation time and accuracy of the different methods[2]. It should be noted that the edge length of the input matrix doubles with each additional spin, so the plots are basically logarithmic on both axis.

The plots show that all of the methods are similar both in terms of computational effort and accuracy, with the Eigendecomposition being the worst and the Chebyshev method having a little advantage. This advantage is believed to increase for even greater matrices.

[2]The measurement was done by Mr. Waldherr, for more details, refer to his diploma thesis.

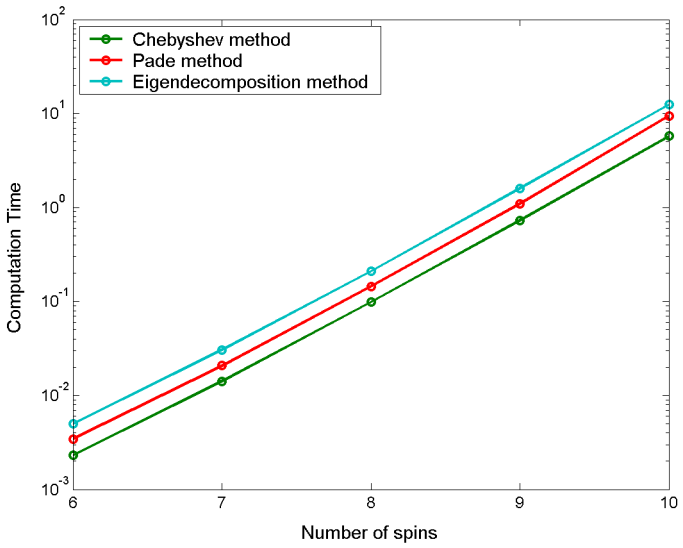

Figure 1: computation time comparison for the different methods of calculating the matrix exponential

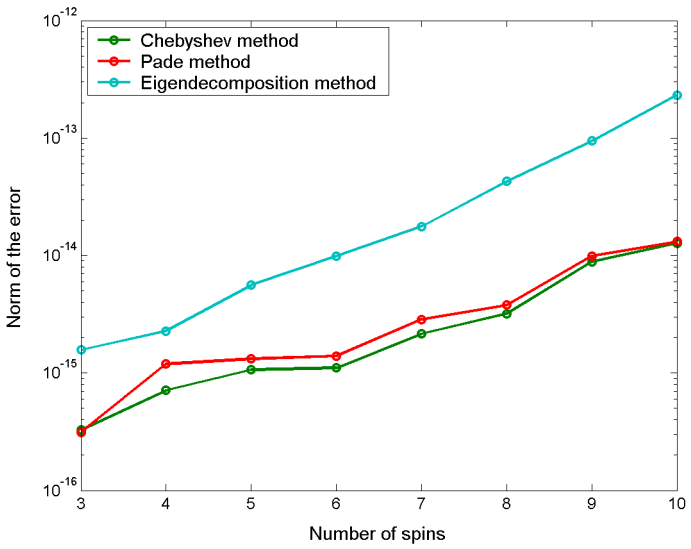

Figure 2: accuracy comparison for the different methods of calculating the matrix exponential

3 Parallel matrix-matrix-multiplication

3.1 Introduction

After calculating the exponentials $\mathbf{U}_k$, they have to be multiplied with each other. The following products are needed:

$$\begin{aligned} &U_0 \\ &U_0 \cdot U_1 \\ &U_0 \cdot U_1 \cdot U_2 \\ &\vdots \\ &U_0 \cdot U_1 \cdot U_2 \cdots U_M \end{aligned} \tag{1}$$

Unfortunately, the exponential of a sparse matrix is no longer sparse. In fact, these matrices can become so large that they no longer fit on the memory of a typical personal computer all at once. This calls for a parallel solution.

In the following sections, three different approaches will be discussed. The first one is a rather intuitive one, the second one is somewhat complicated and the third one is a sort of compromise between the first two ones which preserves the strength of both and reduces the drawbacks of either of the first.

3.2 The slice-wise approach

3.2.1 The algorithm

The idea behind this algorithm is the realisation that one needs only the upper half of the leftmost matrix to calculate the upper half of all of the products.

One divides the matrix U_0 vertically[3] into as many parts — or slices — as processors are present[4]. Each processor[5] gets both one slice of the matrix U_0 and all of the other matrices. Every CPU is now able to calculate a slice of all of the products, starting from the leftmost: $U_0 \cdot U_1$. Afterwards, all of the slices of all of the products have to be collected and reassembled.

[3] That means, one cuts horizontally, so to speak.

[4] The case where there are more CPUs than matrix rows is considered pathological.

[5] The terms processor and computer are used somewhat sloppy and interchangeable in the following sections. The parallelisation approaches all assume a cluster of different nodes, each with its own memory and a processor, which communicate over some one-to-one network.

3.2.2 Analysis

The algorithm depends on the matrices being broadcasted to all the computers. This creates both a memory problem for every node and a time problem for the one which has to broadcast to every other one. The latter problem can be addressed by starting to calculate before the node has all the matrices it will need later on, but the first one remains.

While the total number of scalar multiplications are equal to that of the sequential method for this parallelisation approach, the communication required is prohibitive.

3.3 The tree-like approach

3.3.1 The algorithm

The tree-like approach is most understandable when one just has to compute the product of all the matrices and not the products in between.

This algorithm requires the number of matrices to be a power of two and needs half as many nodes as there are matrices[6].

Basically, one builds one binary tree, starting with the branches and ending with the trunk. In the first step, one multiplies U_0 with U_1 on one CPU, U_2 and U_3 on a different one, the next two matrices on yet another one and so on in parallel. Afterwards, the first and the second product are multiplied on one processor, and so are the third and the fourth on another one again. This is repeated until only one product is left.

This idea has two problems. After the first step only half of the processors are still occupied. This is very inefficient. The other problem is that the intermediate products — which our algorithm requires — are not yet computed.

The obvious solution to these two problems is to have the unoccupied nodes calculate the intermediate products. This can be done, but the resulting rules which CPU has to multiply which matrices are quite complicated and beyond the scope of this text.

3.3.2 Analysis

This algorithm does not need as much communication as the slice-wise. Particularly, broadcasts are avoided. But there are drawbacks. First and foremost, the total number of multiplications ($O(n \log_2 n)$) are greatly increased when compared to the number of multiplications

[6]Naturally, it will also work for other cases, but not as optimal.

used by the sequential one ($O(n)$). It is true that parallelisation always has some price, but in this case, the price may well be having to spend three times as many CPU seconds just to get the result in a shorter time.

Another practical problem is that many multiplications depend on a single result from the previous step. In the final iteration, one result has to be sent to about half of all nodes. This tends to cause delays.

3.4 A pipeline based approach

This is the third algorithm which the author believes to be better suited to the Quantum Control Algorithm than the previous two. By taking the slice-wise idea from the first algorithm and combining it with a generic pipelining approach, one can produce an algorithm which has no more scalar multiplication than the sequential approach and which needs less communication even than the tree-like approach.

3.4.1 The algorithm

This algorithm requires about as many CPUs as there are matrices.

The leftmost matrix is again divided vertically like in the slice-wise approach. Every other matrix is sent to exactly one CPU, U_1 to the first, U_2 to the second and so on.

Now, the first slice of the matrix U_0 is sent to the CPU storing U_1. That node then calculates the first slice of the first product, stores it in its memory and sends it to the processor holding U_2, which in turn multiplies that slice by U_2 and passes the result on. In the meantime, the second slice of U_0 is sent to the first CPU again, where it is processed in the same way.

After the last slice has passed through the pipeline, each node has one product stored in its memory.

3.4.2 Analysis

While the author may be a bit biased towards this one as he independently thought of it[7], this section will mainly contain advantages.

The total amount of scalar multiplications is as good as it is for the sequential variant. And while the total required communication may or may not be optimal, it is at least asymptotically optimal.[8]

[7] Admittedly only after having seen the slice-wise one and having heard something about pipelining...

[8] The total amount of data transmitted is not more than twice as high as the minimum. The data transmitted consists of three portions of almost equal total size. The first one are the matrices U_1 to U_n and the second one is the communication during the pipelines, the

A minor drawback is that any pipeline will have some idle time on the nodes when it is not yet — or not any more — fully filled. This is considered a minor problem for large matrices which can be cut into many slices.

The many small data pieces which need to be transmitted might cause a performance decrease for some network setups.

Also, a number of CPUs equal to the number of matrices might be considered to high. In that case, any number of matrices could be stored on each CPU and could be processed sequentially.

3.4.3 Further improvements

While the algorithm might (or might not) work quite well in this way, it is the personal opinion of the author that its true beauty is only seen when it is fully integrated in the quantum control algorithm.

Each CPU is associated with one timestep, which means it is responsible for one Hamilton matrix H.[9]

First, each node computes the exponential of its current matrix H. Afterwards, the leftmost matrix U is divided into slices and pushed through the pipeline normally. After that, the rightmost matrix U is horizontally divided into slices and pushed backwards through the pipeline. Now — and this is the nice part about this improvement — every single CPU has exactly the two intermediate products it needs to update its Hamilton matrix. Apart from the pipeline, no further communication is required between the iterations of the algorithm.

third one is the collection of the results. The communication which would be absolutely needed (in any non-trivial parallelisation) is (at least) the sending of the input and the receiving of the output. This amounts to two thirds of the communication of the described algorithm.

[9] Or any number of timesteps, as discussed above.