Matrix exponentials and parallel prefix computation in a quantum control problem

T. Auckenthaler[a], M. Bader[a], T. Huckle[a], A. Spörl[b], K. Waldherr[*,a]

[a] *Technische Universität München, Department of Informatics, Boltzmannstr. 3, 85748 Garching, Germany*
[b] *Technische Universität München, Department of Chemistry, Lichtenbergstr. 4, 85747 Garching, Germany*

Abstract

Quantum control plays a key role in quantum technology, in particular for steering quantum systems. As problem size grows exponentially with the system size, it is necessary to deal with fast numerical algorithms and implementations. We improved an existing code for quantum control concerning two linear algebra tasks: The computation of the matrix exponential and efficient parallelisation of prefix matrix multiplication.

For the matrix exponential we compare three methods: the eigendecomposition method, the Padé method and a polynomial expansion based on Chebyshev polynomials. We show that the Chebyshev method outperforms the other methods both in terms of computation time and accuracy. For the prefix problem we compare the tree-based parallel prefix scheme, which is based on a recursive approach, with a sequential multiplication scheme where only the individual matrix multiplications are parallelised. We show that this fine-grain approach outperforms the parallel prefix scheme by a factor of 2–3, depending on parallel hardware and problem size, and also leads to lesser memory requirements.

Overall, the improved linear algebra implementations not only led to a considerable runtime reduction, but also allowed us to tackle problems of larger size on the same parallel compute cluster.

Key words: matrix exponential, Chebyshev polynomials, parallel prefix problem, parallel matrix multiplication, quantum control algorithm

1. Introduction

Quantum computation and Hamiltonian simulation offers solutions to computational problems which are hard to solve with a conventional device [5, 21,

*Corresponding author
Email addresses: auckenth@in.tum.de (T. Auckenthaler), bader@in.tum.de (M. Bader), huckle@in.tum.de (T. Huckle), waldherr@in.tum.de (K. Waldherr)

24]. In addition, the number of coherently controllable quantum systems is increasing [18, 3, 12], so the question arises to steer these systems in an optimal way to fight the limiting decoherence processes. For a small number of problems there exist analytical solutions [13, 14, 27], but for the vast majority only numerical solutions are available [23, 25]. To get these numerical solutions, the GRAPE (GRadient Ascent Pulse Engineering) algorithm was developed [15], which provides a method for optimal quantum control based on gradient flows. It is also used for NMR spectroscopy applications, such as the design of pulse shapes for problems involving transfer of coherence between coupled spins or the synthesis of unitary propagators in a network of coupled spins (see [15]).

To get a first impression about the complexity, the GRAPE algorithm is listed in algorithm 1. It describes a step-wise optimisation in conjugate gradients: In each iteration step, we compute the Hamiltonian quantum evolution which is given by the matrix exponentials at every time step k. Afterwards we calculate the forward (step 2) and backward propagation (step 3) given by a sequence of evolutions of the quantum system under M piecewise constant Hamiltonians H_k. The gradient $\frac{\partial h(U(t_k))}{\partial u_j}$ of the (real-valued) performance function h, which measures the deviation of the time evolution $U(T)$ from the desired target $W(T)$, is given by the real part of trace evaluations (calculated in step 4) and needed for the update of the controls in step 5. For the controls we may specify an arbitrary initial value $u_j^{(0)}(t_k)$ in step 1 (usually all values are chosen equal to one), or initialise the controls with a suitable initial guess, if respective previous calculations are available.

1: set initial controls $u_j^{(0)}(t_k)$ for all times t_k with $k = 1, \ldots, M$;
2: starting from $U_0 = I$, calculate the forward-propagation for all $t_1, \ldots, t_M$ (for simplicity $\Delta t := t_{k+1} - t_k$ uniform):

$$U(t_k) = e^{-i\Delta t H_k} e^{-i\Delta t H_{k-1}} \cdots e^{-i\Delta t H_1};$$

3: likewise, starting with $T = t_M$ and $W(T) = \text{const} \cdot U_G$ compute the back-propagation for all $t_M, t_{M-1}, \ldots, t_1$

$$W(t_k) = e^{i\Delta t H_k} e^{i\Delta t H_{k+1}} \cdots e^{i\Delta t H_M} W(T);$$

4: calculate $\frac{\partial h(U(t_k))}{\partial u_j} = \Re\left(\text{tr}\{W^{\text{H}}(t_{k+1})(-iH_j)U(t_k)\}\right)$
5: with $u_j^{(1)}(t_k) = u_j^{(0)}(t_k) + \epsilon \frac{\partial h}{\partial u_j} \mid_{t=t_k}$ update all the piece-wise constant Hamiltonians H_k and continue with step 2.

Algorithm 1: Gradient Flow Algorithm for Quantum Control

The computational effort for the GRAPE algorithm is dominated by

1. the computation of the matrix exponentials $U_k := e^{-i\Delta t H_k}$, where H_k denotes a large and sparse Hermitian matrix,
2. the matrix multiplications in the prefix and postfix problems (steps 2 and

3),

The trace evaluations in step 4 are about as expensive as the matrix exponentials, but they offer no possibility for algorithmic improvement.

The previously existing code, developed in 2006 ([4]), was restricted to treating quantum systems with up to 10 spins, because of runtime and storage limitations. Our goal for the present work was to improve the existing code in order to deal with systems of higher dimensions (11–15 spins).

In section 2 we present structure and properties of the matrices H_k, and we investigate three different methods for the computation of the exponentials. We compare the three methods in terms of accuracy and efficiency. In section 3 we focus on two approaches to compute the prefix problem (steps 2 and 3 of algorithm 1): The original method, as presented in [4], used parallel prefix multiplication, i.e. an approach based on a recursive tree scheme. We compare this approach with a sequential prefix scheme, where the individual matrix multiplications are parallelised, instead. The respective, more fine-grain parallelisation reduced both memory requirements and the total computational work, and therefore proved to be advantageous despite not achieving full speedup. In section 4 we show parallel performance results for our implementation of the entire GRAPE algorithm.

2. Computation of Matrix Exponentials

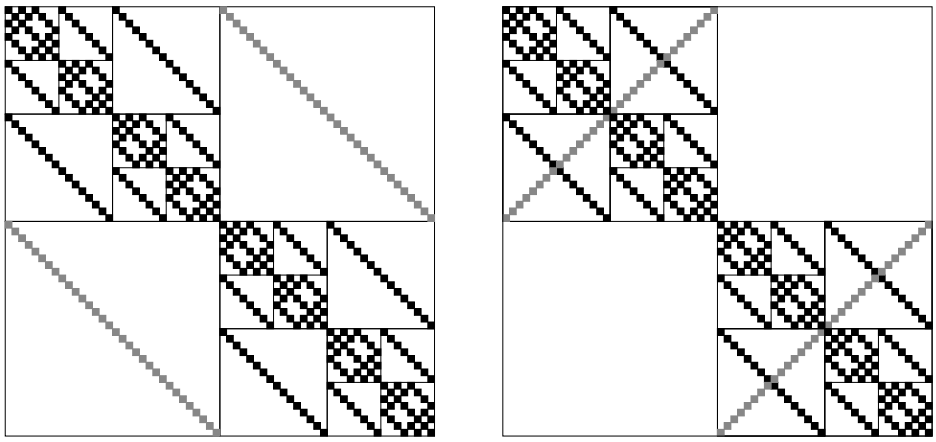
In this section we consider several methods for the computation of the matrix exponentials $U_k = e^{-i\Delta t H_k}$ appearing in steps 2 and 3 of algorithm 1. The so-called Hamiltonians H_k are of the form $H_k = H^{(\mathrm{drift})} + u_k H_k^{(\mathrm{control})}$, where $H^{(\mathrm{drift})}$ describes the constant and time-independent drift matrix and $H_k^{(\mathrm{control})}$ denotes a Hermitian matrix, which is given by a sum of tensor products involving the identity matrix I and the Pauli matrices σ_x and σ_y. In our applications, $H^{(\mathrm{drift})}$ is real, diagonal and persymmetric and the $H_k^{(\mathrm{control})}$ matrices can be brought to real-symmetric and persymmetric matrices by a diagonal and unitary transform D_k. The transformed matrix $\hat{H}_k = D_k^{\mathrm{H}} H_k D_k$ is real, symmetric, persymmetric and has the same sparsity pattern than H_k. Therefore, $\hat{H}_k$ can be brought to block diagonal form by a second linear transform V_k:

$$V_k^{\mathrm{T}} D_k^{\mathrm{H}} H_k D_k V_k = \begin{pmatrix} H_k^{(1)} & \mathbf{0} \\ \mathbf{0} & H_k^{(2)} \end{pmatrix}$$

Figure 1 illustrates the sparsity patterns of the original and transformed matrices.

In this manner, the problem of computing $U_k = e^{-i\Delta t H_k}$ can be reduced to two problems $e^{-i\Delta t H_k^{(l)}}$ of half size:

$$U_k = D_k V_k \begin{pmatrix} e^{-i\Delta t H_k^{(1)}} & \mathbf{0} \\ \mathbf{0} & e^{-i\Delta t H_k^{(2)}} \end{pmatrix} V_k^{\mathrm{T}} D_k^{\mathrm{H}}. \tag{1}$$



(a) Original Hamiltonian (b) Hamiltonian after linear transform

Figure 1: The sparsity pattern of the matrices to be exponentiated. In (a) the original matrix is illustrated, in (b) the pattern after the linear transform. As the matrices are given by tensor products of Pauli and identity matrices, we get the illustrated multi-level Toeplitz structure.

Hence, we gain a speedup of 4 by halving the matrix sizes (the application of the transformations D_k and V_k can be neglected), and – depending on the method used to compute the exponentials – gain an additional performance advantage from using real instead of complex arithmetics.

According to these considerations, we have to compute exponentials e^{iA}, where the matrix $A = -\Delta t H_k^{(l)}$, according to equation (1), is real-symmetric and sparse (the sparsity pattern is illustrated in figure 1b). There is a rich collection of algorithms to compute matrix exponentials ([19, 20, 9, 8], e.g.), however, as pointed out in [19, 20], the method of computation has to be carefully chosen for each individual problem setting. For the scenario in algorithm 1 ("compute e^{iA} for a given matrix A"), this leaves us three efficient methods for computation: the eigendecomposition method, the Padé approximation, and the Chebyshev expansion.

2.1. Eigendecomposition

This method was used in the original code [4]. As the matrix A is normal, we can diagonalise it by a unitary linear transform:

$$A = V \mathrm{diag}(\lambda_1, \ldots, \lambda_n) V^{\mathrm{T}}. \tag{2}$$

After computing this decomposition we simply get

$$e^{iA} = V \mathrm{diag}(e^{i\lambda_1}, \ldots, e^{i\lambda_n}) V^{\mathrm{T}}. \tag{3}$$

The most costly operations in this approach are the computation of the eigendecomposition of A (which can be done with non-complex arithmetic) in (2)

and one matrix-multiplication in (3). In our concrete implementation, LAPACK's `dsyev` routine was used for the real-symmetric eigenproblem. The performance of this method is not optimal: Although the matrix is sparse, a complete eigendecomposition is required. In other words the eigendecomposition method doesn't let us exploit the sparsity of A. The performance results in section 2.4 attest these considerations.

2.2. Padé approximation

The Padé approximation is a rational expansion of the form $r_{st}(x) = \frac{p_{st}(x)}{q_{st}(x)}$, where p_{st} and q_{st} denote polynomials of degree s and t respectively. In the case of the exponential function $x \mapsto e^x$ these polynomials are given by

$$\begin{aligned} p_{st}(x) &= \sum_{j=0}^{s} \frac{(s+t-j)!s!}{(s+t)!(s-j)!} \frac{x^j}{j!}, \\ q_{st}(x) &= \sum_{j=0}^{t} \frac{(s+t-j)!t!}{(s+t)!(t-j)!} \frac{(-x)^j}{j!}. \end{aligned} \tag{4}$$

We can generalize (4) to any quadratic matrix A, and obtain the formula

$$e^{iA} \approx r_{st}(iA) := (q_{st}(iA))^{-1} p_{st}(iA), \tag{5}$$

where $p_{st}(iA)$ and $q_{st}(iA)$ denote the accordant matrix polynomials.

In practice, the diagonal Padé approximants (i.e. $s = t$) are used, because in this case the polynomial evaluation of $p_t(iA) := p_{tt}(iA)$ and $q_t(iA) := q_{tt}(iA)$ can be done in a very efficient way (see e.g. [8]). The approach

$$r_t(iA) = (q_t(iA))^{-1} p_t(iA), \tag{6}$$

then forms a good approximation for matrices A near the origin, i.e. with small norm $\|A\|$. The so-called scaling & squaring technique provides us a simple means to reduce the norm of A.

Scaling and squaring

The scaling & squaring method exploits the relation $e^A = \left(e^{A/\beta}\right)^\beta$ and is therefore a helpful method to reduce the matrix norm of the exponent. The idea is to choose β as an integral power of 2, say $\beta = 2^\gamma$, to compute $e^{A/\beta}$, and then to undo the effect of the scaling by a repeated squaring procedure (γ squaring steps). Altogether, the Padé method computes e^{iA} via the formula

$$\left[r_t(iA/2^\gamma)\right]^{2^\gamma}. \tag{7}$$

The most time-consuming parts within this method are the evaluation of the matrix polynomials p_t and q_t in (6), the computation of the LU decomposition to solve equation (6), and the γ multiplications in the final squaring steps (7). Throughout the evaluation of the matrix polynomials, the sparsity pattern of

A is destroyed. Therefore, this evaluation had to be implemented using dense matrix library calls. In addition, the LU decomposition of $q_t(iA)$ and the final squaring steps are also computed for dense matrices. Hence, the sparsity pattern of A cannot be exploited with the Padé approximation method.

2.3. Chebyshev series expansion

In this section we investigate a method based on a polynomial expansion of the exponential function. This approach leads to the most efficient way for the computation of our matrix exponentials, also because it can exploit sparsity of the matrices.

For this method, we need the Chebyshev polynomials of the first kind, which are given by the formula

$$T_k(x) = \cos(k \arccos(x)) \tag{8}$$

and can be described by the three-term-recurrence formula

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x). \end{aligned} \tag{9}$$

With respect to the weight function $\omega(x) = (1 - x^2)^{-1/2}$, these polynomials are orthogonal. In this sense, a function $f(x)$ with arguments $|x| \leq 1$ can be represented by an infinite Chebyshev series according to

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x), \qquad \text{with} \quad a_k = \frac{2}{\pi} \int_{-1}^{1} f(x) T_k(x) \frac{dx}{\sqrt{1-x^2}}. \tag{10}$$

Details can be found in [22]. In our case $f(x) = e^x$ and the coefficients then take the special form $a_k = 2i^k J_k(-i)$ with the Bessel functions J_k. For $|x| \leq 1$ this leads to $e^x = J_0(i) + 2 \sum_{k=1}^{\infty} i^k J_k(-i) T_k(x)$. Therefore, the matrix exponential of iA, provided that the normalisation condition $||A|| \leq 1$ is satisfied, is given by

$$e^{iA} = J_0(i) I + 2 \sum_{k=1}^{\infty} i^k J_k(-i) T_k(iA). \tag{11}$$

For dealing with matrices A of arbitrary finite norm, the scaling and squaring technique is applied once again (as for the Padé approximation).

The Chebyshev method requires the computation of only one matrix polynomial per matrix exponential, and the update formula is of the form

$$T_{k+1} \leftarrow -T_{k-1} + 2iAT_k. \tag{12}$$

This rule can simply be implemented by using the BLAS routine for general matrix multiplication, $C \leftarrow \alpha C + \beta AB$. Moreover, in every summand of the Chebyshev series, the matrix product AT_k in equation (12) is of the form *sparse* $\times$ *dense* (as A is sparse), and can be computed by Sparse BLAS libraries. Therefore, within the Chebyshev method, matrix products of the form *dense* $\times$ *dense* only occur in the squaring steps, if scaling & squaring is necessary.

2.4. Performance results

In this paragraph we present numerical results which reveal our theoretical analysis in the last section. We compare the eigendecomposition method (see 2.1), the Padé approximation (see 2.2), and the Chebyshev expansion (see 2.3) in terms of computing time (see 2.4.1), but also in terms of accuracy (see 2.4.2). When dealing with the Padé or Chebyshev expansion methods, we have to choose two parameters: the length of the expansion series and the number of scaling & squaring steps. For the implementation of the Padé approximation method, we employed the parameter pairs given in [8]. Accordingly, we selected the parameters for the Chebyshev expansion such that we guarantee the same accuracy: For the number of scaling & squaring steps we take the smallest integer γ such that the 1-norm of the exponent is less or equal 1; the Chebyshev series itself is chosen of order 17.

2.4.1. Computing time

As pointed out in the introduction, the computation of the matrix exponentials is one of the three time-critical parts of the GRAPE algorithm. Due to the exploitation of the sparsity of the exponent matrices, we were to expect that the Chebyshev method would outperform both eigendecomposition and Padé method. This was backed up by the performance measurements plotted in figure 2. The respective measurements were obtained on an Intel Itanium2 processor (*Montecito*, 1.6 GHz). Figure 2 illustrates that all considered methods are of the same time complexity. Furthermore, we see that the Chebyshev method is more than two times faster than the eigendecomposition method which was used before.

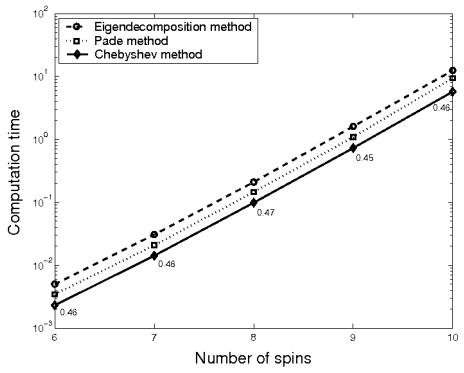


Figure 2: Comparison of the CPU times required for calculating one matrix exponential as a function of the system size, where n spins translate into a complex matrix of dimension $2^n \times 2^n$. The numbers in the plot denote the time reduction for the Chebyshev method compared to the eigendecomposition approach.

In terms of memory requirements, the Chebyshev method also gives a certain improvement: For the resummation of the matrix polynomial (11) only one

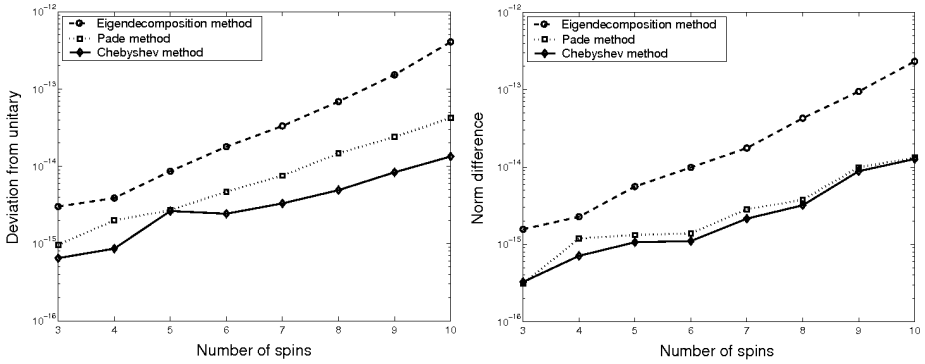
additional matrix is needed for temporary storage. In contrast, both eigendecomposition method and Padé approximation request the storage of three dense matrices for each matrix exponential. However, this is of minor influence, as the most storage-consuming part of the GRAPE algorithm is caused by the computation of the prefix problem described in chapter 3.

2.4.2. Accuracy

The accuracy of the approximations $\widetilde{U} \approx e^{iA}$, with sparse $A = -\Delta t H_k$, can be measured in two different ways:

- **deviation from unitary:** With A being complex Hermitian, the exponential e^{iA} fulfills the equation $e^{iA}\left(e^{iA}\right)^{\mathrm{H}} = e^{iA}e^{-iA^{\mathrm{H}}} = e^{iA}e^{-iA} = e^{0} = I$ and is therefore unitary. An appropriate error measure for $\widetilde{U} = e^{iA}$ is thus given by $\|\widetilde{U}\widetilde{U}^{\mathrm{H}} - I\|$.
- **deviation from exact solution:** If the drift term $H^{(\mathrm{drift})}$ of the Hamiltonian is set to zero, the eigendecomposition of H is analytically given by the Fourier transform matrix (see [7]). In this manner, we can apply our methods on specially constructed Hamiltonians H with exactly known exponential $U = e^{-iH}$, and then measure the approximation error $\|\widetilde{U} - U\|$ in a specific matrix norm.

Figure 3 illustrates the behaviour of the different methods in terms of accuracy. Again, the Chebyshev method performs best. Compared to the previously used eigendecomposition method, we gain a factor of 10^{-2} in both accuracy measures.



(a) deviation from unitary (b) deviation from exact solution

Figure 3: Comparison of the accuracy of different methods subject to the system size: (a) deviation from unitarity measured by $\|\widetilde{U}\widetilde{U}^{\mathrm{H}} - I\|$ and (b) errors $\|\widetilde{U} - U\|$.

Hence, the Chebyshev method outperforms the other methods both in terms of accuracy and efficiency and therefore allows us to speedup the GRAPE algorithm without losing accuracy.

3. Parallel Prefix Multiplication

After the computation of the matrix exponentials $U_k := e^{-i\Delta t H_k}$, the GRAPE algorithm will compute the sequence of evolutions $U(t_k)$, where

$$U(t_k) \;=\; e^{-i\Delta t H_k} e^{-i\Delta t H_{k-1}} \cdots e^{-i\Delta t H_1} \tag{13}$$

(step 2 of algorithm 1). Similar, the reverse products will be computed (step 3). Both the computation of the $U(t_k)$ and the computation of the reverse products $W(t_k)$ are instances of the so-called *prefix problem*, i.e. to compute all matrix products $U_1 U_2 \cdots U_k$ for $k = 1, \ldots, M$ for a given set of matrices U_k. In the following, we will focus on the parallel computation of the forward sequence, as the computation of the backward sequence works in exactly the same way.

Our total problem size is determined by the size n of the matrices, which grows exponentially with the number q of spins ($n = 2^q$), and by the number M of matrices, which is determined by the choice of the time step. Typical values of M are in the range of 100–1000. Hence, for simulations that can treat more than 10 spins – currently, system simulations with 10–15 spins are desired –, parallel implementation of the prefix computation on compute clusters or even supercomputers is necessary.

3.1. Coarse-grain vs. fine-grain parallelisation

The prefix problem offers both a fine-grain and a coarse-grain approach for parallelisation. The fine-grain approach is to simply compute the products $\prod_{j=1}^{k} U_j$ sequentially for $k = 1, \ldots, M$ in a loop over k, but to parallelise the individual matrix multiplications. The coarse-grain approach, in contrast, reorganises the multiplications using the following divide-and-conquer approach to compute a product $U_{k_1:k_2} := \prod_{j=k_1}^{k_2} U_j$:

1. Compute the products $U_{k_1:\kappa}$ for all $\kappa = k_1, \ldots, \hat{k}-1$, with $\hat{k} = \left\lceil \frac{1}{2}(k_1 + k_2) \right\rceil$.
2. Compute the products $U_{\hat{k}:\kappa}$ for all $\kappa = \hat{k}, \ldots, k_2$.
   Steps 1 and 2 can be executed in parallel.
3. Compute the products $U_{k_1:\kappa} := U_{k_1:(\hat{k}-1)} U_{\hat{k}:\kappa}$ for all $\kappa = \hat{k}, \ldots, k_2$. All $(k_2 - k_1 + 1)$ products of this step can be computed in parallel.

Recursive extension of this scheme leads to a tree-like multiplication scheme as given in figure 4. This so-called *parallel prefix scheme* was first presented in [17]. To compute the products $U_{1:k}$ for $k = 1, \ldots, M$, at most $\frac{M}{2}$ parallel processes can be used. For larger numbers of available processor cores, as might be the case on a massively parallel system, the individual matrix multiplications would therefore need to be parallelised, as well.

The coarse-grain, parallel prefix scheme features a simpler communication pattern than the fine-grain, parallel matrix multiplication. In addition, the individual matrix multiplications in the tree-scheme are heavy-weight sequential work units, such that good parallel speedups of the tree algorithm are to be expected, and were also achieved in the existing code [4]. However, the tree scheme increases the total computational work by a logarithmic factor: note

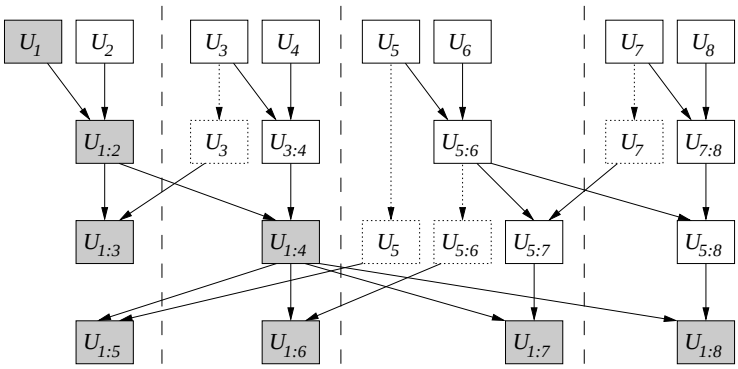


Figure 4: Divide-and-conquer scheme for the parallel prefix computation. The dashed lines define the processor scopes; solid lines denote communication between processors, and dotted lines and boxes indicate where matrices have to be retained for next-level computations. Note that the result matrices (grey boxes) are not balanced among the processes.

that the parallel computation requires $\mathcal{O}(\log M)$ subsequent steps (given by the horizontal levels in figure 4). This additional logarithmic factor is saved in the fine-grain approach, where the individual matrix multiplications are parallelised. On the other hand, for the comparably small matrix sizes (1024×1024, e.g.), it is difficult to achieve good parallel speedups on large numbers of processors.

In a previous performance study [1], we compared the overall parallel efficiency for fine-grain, coarse-grain, and hybrid parallelisation. In the hybrid approach, evaluation of the parallel prefix tree is restricted to the lower levels, whereas the products of the upper levels are computed via fine-grain, parallel matrix multiplication. We especially focused on the increased amount of total work for the parallel prefix scheme, and for hybrid schemes. We found that the coarse-grain parallel prefix scheme and also hybrid implementations, even when using a more efficient memory distribution than given in figure 4, are only advantageous for matrix sizes of 512×512 or smaller, which correspond to systems of up to 9 spins. The fine-grain approach with parallel matrix multiplication was consistently faster for matrices of size 1024×1024, or higher (i.e. 10 spins, or more), with the advantage growing for larger systems. Therefore, in the present work we switched to a fine-grain parallelisation approach, and used a 3D block-oriented method for parallel matrix multiplication.

3.2. 3D block-oriented parallel matrix multiplication

Parallel matrix multiplication is a problem that offers a multitude of well-established algorithms, e.g. [2, 26, 16]. Nevertheless, our problem setting poses a set of challenges that makes it important to choose the adopted approach carefully:

- For the forward and backward propagation, we need to compute many matrix products in sequence, but the size of the individual matrices is too

small to achieve good speedups on many processors (such as on the 128 CPUs of our compute cluster used in section 4). Hence, we require an algorithm that scales well for small matrices, in particular.

- After computation of the matrix exponentials – where each individual exponential is computed sequentially, but all exponentials are computed concurrently – the matrices will already be distributed over all available processors. However, each individual matrix is stored on a single processor. Hence, our algorithm will either need to work directly on such a distribution, or require only a comparably cheap redistribution of matrices to several or all processors.

In its most general form, matrix multiplication, for example $C = AB$, can be formulated via the algorithm

1: **for all** $(i, j, k) \in \{1, \ldots, N\} \times \{1, \ldots, N\} \times \{1, \ldots, N\}$ **do**
2: $\quad C_{ik} \leftarrow C_{ik} + A_{ij}B_{jk}$
3: **end for**

(assuming C is initialised to zero), where the C_{ik}, A_{ij}, and B_{jk} may be elements or matrix blocks, and where the body of the for loop, i.e. all block operations of step 2, may be executed entirely in parallel. If we visualise the block operations as a cube of size $N \times N \times N$, then the projections in k-, i-, and j-direction indicate which matrix blocks A_{ij}, B_{jk}, and C_{ik} are to be accessed, respectively – compare also figure 5. Depending on the data distribution and also the distribution of the block operations to the available processors, algorithms for parallel matrix multiplication are often classified into the following types:

1D algorithms parallelise the block operations into planes in the cube, which corresponds to column-wise computation of the result matrix.

2D algorithms distribute the block operations into columns; usually these columns work on individual blocks in the result matrix, such that all computations on that result block are executed by a single processor ("owner computes").

3D algorithms define a blocking on the three nested main loops of the algorithm. Hence, the parallelisation is then purely *work-oriented* in the sense that a 3D subblock of block operations is assigned to each processor. Both result and operand matrices may be distributed over several processors.

Naturally, the three classes lead to different performance properties of the algorithms. 3D algorithms, most important, feature the lowest communication effort – $\mathcal{O}(n^2 p^{1/3})$ vs. $\mathcal{O}(n^2 p^{1/2})$ for 2D, and $\mathcal{O}(n^2 p)$ for 1D algorithms (compare [6, 11]). The resulting lower bandwidth requirements are of particular importance in our problem setting, because we expect scalability problems due to the small matrix sizes. In addition, a 3D algorithm will produce larger block multiplications as sequential work units. For most matrix multiplication libraries, the sequential performance grows for increasing matrix size, especially

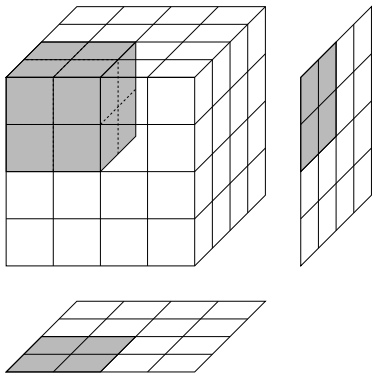


Figure 5: Block operations in the 3D block multiplication algorithm. The projections onto the planes below and right illustrate the accessed matrix blocks of the result matrix C and of one of the operand matrices, respectively.

within the range of small to moderate matrix sizes, which will be the dominant case in our problem setting.

For our application of prefix computation, a 1D algorithm had already been considered in the original work by Gradl et al. [4]. However, it was found to be inferior to the coarse-grain parallel prefix scheme, both with respect to runtime, and also due to increased memory and communication requirements. The use of 2D and 3D algorithms was examined in our performance study [1], were 3D algorithms showed the best runtime performance for our problem setting. The improved performance comes at the cost of a slightly higher memory requirement: $\mathcal{O}(n^2 p^{1/3})$ additional matrix elements distributed on p processors, which is less than an additional matrix per processor. As we typically store several matrices (up to 8 in our examples in section 4) on each CPU, this additional requirement is easily affordable.

We therefore implemented a 3D block algorithm, which compared to the performance study in [1] was tuned for our available hardware. The algorithm extends the classical 2D owner-computes idea in the sense that a given block of the result matrix may be computed by several processes. The number D of blocks per index dimension is chosen such that D is the smallest power of 2 with $D^3 \geq p$ (p the number of processes). Hence, each of the D^2 blocks of the result matrix C is computed by p/D^2 processes, and each processor performs D^3/p block operations, all of which work on a single matrix block C_{ik}. Altogether, each process will perform the following two steps:

for all local blocks operations (i, j, k) **do**
  Fetch A_{ij} and B_{jk} from remote processes
  $C_{ik} \leftarrow C_{ik} + A_{ij} B_{jk}$
end for
Accumulate all results for block C_{ik} (group-collective operation)

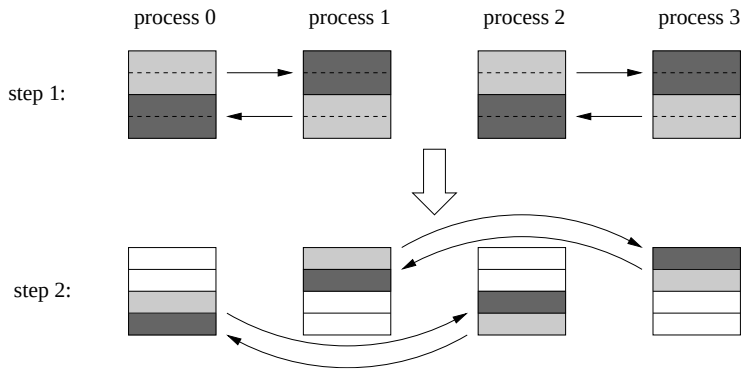


Figure 6: Communication pattern during the accumulation of results within a single matrix block C_{ik}. In each successive step, pairs of processes exchange and accumulate their partial results: the number of exchanged rows is halved and the distance between two communicating processes is doubled. Gray boxes represent rows that are sent to the corresponding process, black boxes represent rows that are received and accumulated to local data.

The accumulation of the results in C_{ik} is a group-collective procedure, which is performed in parallel by all p/D^2 processes that compute the same block C_{ik}. The accumulation is organised as a pairwise accumulation of data, as illustrated in figure 6.

After $\log_2(p/D^2)$ subsequent steps, each process has computed one part of the block C_{ik} – these parts are then broadcasted to the other processes. Note that, within the for-loop to compute the block operations, communication and computation may be overlapped to hide communication costs.

During a computation $U_{1:k} = U_{1:k-1}U_k$, only blocks of the matrices $U_{1:k-1}$ and U_k are accessed. Hence, if $U_{1:k-1}$ and U_k would be stored on only one process each, we would obtain a communication bottleneck. We therefore distribute each matrix onto all available processes before starting the prefix computation. The granularity of this distribution is given by the subblocks used in the accumulation step (compare figure 6). Subblock l of a matrix U_k will be stored on process $(l + k) \mod p$ (p the number of processes). After each accumulation process, the computed matrix $U_{1:k}$ will be correctly distributed to all processes. After the entire prefix loop, the result matrices $U_{1:k}$ will be assembled in a final global communication step, such that again all matrices are stored on a separate process.

3.3. Performance results

The performance of the 3D parallel matrix multiplication was evaluated on an *Infiniband cluster* with 32 Opteron nodes; each node contains four AMD Opteron 850 processors (2.4 GHz) connected to 8 GB or 16 GB of shared memory. Each node is equipped with one MT23108 InfiniBand Host Channel Adapter card, which is thus shared by 4 processors for communication. The sequential

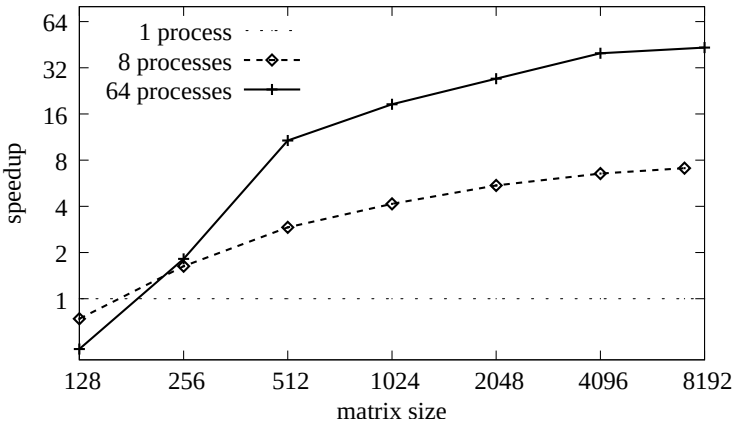


Figure 7: Speedup depending on matrix size for the 3D parallel matrix multiplication on 8 and 64 processes of the Infiniband cluster.

block multiplications were executed by the `dgemm` implementation of AMD's Core Math Library (ACML, v. 3.5.0).

Figure 7 shows the achieved speedups, when using 8 or 64 processors (on 2 or 16 nodes, respectively), for matrix sizes from 128×128 (7 spins) up to 8192×8192 (13 spins). We see that good speedups are only achieved for comparably large matrix sizes. For matrices of size 1024×1024, the achieved parallel efficiency is only about 50 % for 8 processes and about 30 % for 64 processors. This is both due to communication overhead and due to decreased performance of the sequential matrix multiplication for very small matrix blocks.

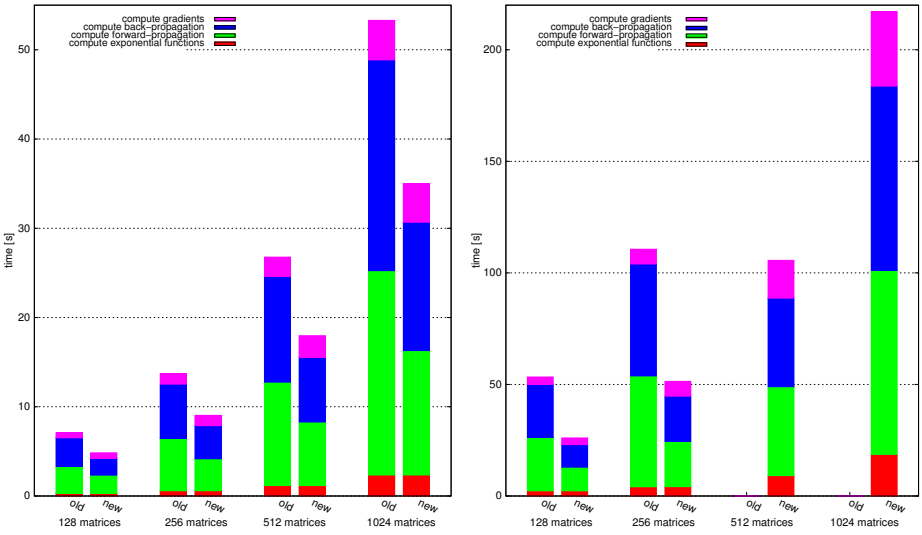
Despite the less-than-optimal speedups, the 3D block multiplication outperforms the parallel prefix scheme, which can be seen from Table 1. There, the runtimes to execute the prefix multiplications of the forward propagation are given for using parallel matrix multiplication on $p = 8$ up to $p = 128$ processors. For comparison, the rightmost column gives the runtime of the parallel prefix scheme (using the optimal number of processors, respectively). Using parallel matrix multiplication to parallelise the prefix computations gives a performance advantage of factor 2–3. As expected, the performance advantage grows for larger matrix sizes.

4. Performance Results for the GRAPE Algorithm

In figure 8 we illustrate the runtimes for one complete iteration of the GRAPE algorithm on the Infiniband cluster (compare section 3.3). For each problem size, the total runtime is split up into four parts: computing the matrix exponentials, forward propagation, back-propagation, and computation of the gradients (steps 4 and 5 of algorithm 1). For the forward and backward propagation, we compare the runtimes when using the 3D parallel matrix multiplication instead of the parallel prefix scheme.

		parallel runtime [s]					
Matrix size	M	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	par. prefix
512×512	2048	41.99	35.53	20.52	11.48	11.35	23.00
1024×1024	256	28.94	21.38	12.12	6.50	5.31	13.49
2048×2048	512	–	–	128.06	67.51	51.59	152.04
4096×4096	64	–	–	93.13	49.16	34.91	229.73

Table 1: Parallel runtimes for the prefix problem with parallel matrix multiplication on p processors for different matrix sizes and number M of matrices. The fastest runtime of the parallel prefix scheme on the optimal number of processors is given for comparison.



9 spins (matrix size 512×512) 10 spins (matrix size 1024×1024)

Figure 8: Runtime of one iteration of the GRAPE algorithm for different problem sizes. We compare the runtimes for using the parallel prefix scheme ('old') vs. using the 3D parallel matrix multiplication ('new') for systems with 9 and 10 spins, where the number M of matrices varies between 128 and 1024.

Using the 3D matrix multiplication gives a substantial performance advantage already for the 9-spin problem, where the matrix sizes are still very small (512×512)). For the 10-spin problem (matrix size 1024×1024), the 3D matrix multiplication is already more than 2 times faster than the parallel prefix scheme. Note that for the 10-spin problem, we were able to compute problems with $M = 512$ and $M = 1024$, which was not possible with the parallel-prefix-scheme implementation due to lack of memory. More precisely, the inhomogeneous distribution of matrices to processors was responsible for this restriction (compare figure 4).

5. Conclusions

In the presented work, we substantially improved the two most time-consuming linear algebra tasks in the GRAPE algorithm for quantum control problems. For the computation of the exponentials of Hamiltonian matrices, we integrated a method based on Chebyshev series expansion. As this method is able to exploit the sparsity of the exponent matrices, we obtain a runtime improvement of more than factor 2 compared to the earlier adopted approach, which was based on the eigendecomposition of the exponents. In addition, the Chebyshev method was shown to be more accurate than the eigendecomposition method.

For parallelisation of the prefix multiplication in the forward and back propagation step, we introduced a fine-grain parallelisation that relies solely on the parallelisation of individual matrix multiplications. Compared to the earlier adopted, tree-oriented parallel prefix computation, the respective approach saves the additional logarithmic factor in the total computational work. We therefore obtain improved runtimes despite the non-optimal parallel efficiency of the parallel matrix multiplication for the small matrix sizes typical for our quantum control problem. Overall, we gain a factor 2–3 in runtime, and also reduce the memory requirements of the implementation.

Acknowledgements

This work has been performed in the framework of the *HLRB 2 Project h1051(see [10]), Advanced Quantum Control by Gradient-Flow Algorithms on Parallel Clusters* at *Leibniz Rechenzentrum* of the Bavarian Academy of Science.

References

[1] M. Bader, S. Hanigk, T. Huckle, Parallelisation of block recursive matrix multiplication in prefix computations, in: C. Bischof, et al. (eds.), Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the ParCo, Parallel Computing 2007, vol. 38 of NIC Series, 2008, pp. 175–184.

[2] J. Choi, J. Dongarra, D. Walker, PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers, Concurrency: Practice and Experience 6 (7) (1994) 543–570.

[3] J. I. Cirac, P. Zoller, Quantum-Computing with Cold Trapped Ions, Phys. Rev. Lett 74 (1995) 4091–4094.

[4] T. Gradl, A. Spörl, S. Glaser, T. Schulte-Herbrüggen, T. Huckle, Parallelising matrix operations on clusters for an optimal-controled quantum compiler, vol. 4128 of Lecture Notes in Computer Science, Springer, 2006.

[5] L. K. Grover, Quantum Mechanics Helps in Searching for a Needle in a Haystack, Phys. Rev. Lett. 79 (1997) 325–328.

[6] A. Gupta, V. Kumar, Scalability of parallel algorithms for matrix multiplication, in: International Conference on Parallel Processing – ICPP'93, vol. 3, 1993, pp. 115–123.

[7] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[8] N. J. Higham, The scaling and squaring method for the matrix exponential revisited, SIAM J. Matrix Anal. Appl. 26 (4) (2005) 1179–1193.

[9] N. J. Higham, Functions of Matrices: Theory and Computation, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[10] T. Huckle, T. Schulte-Herbrüggen, A. Spörl, K. Waldherr, S. Glaser, Using the HLRB-II cluster as a quantum CISC-compiler, in: High Performance Computing in Science and Engineering, Garching 2007, Springer, 2008.

[11] D. Irony, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, J. Parallel Distrib. Comput. 64 (9) (2004) 1017–1026.

[12] B. E. Kane, A Silicon-Based Nuclear Spin Quantum Computer, Nature (London) 393 (1998) 133–137.

[13] N. Khaneja, S. J. Glaser, Cartan Decomposition of $SU(2^n)$ and Control of Spin Systems, Chem. Phys. 267 (2001) 11–23.

[14] N. Khaneja, B. Heitmann, A. Spörl, H. Yuan, T. Schulte-Herbrggen, S. J. Glaser, Shortest Paths for Efficient Control of Indirectly Coupled Qubits, Phys. Rev. A 75 (2007) 012322.

[15] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, S. J. Glaser, Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms, J. Magn. Reson. 172 (2005) 296–305.

[16] M. Krishnan, J. Nieplocha, SRUMMA: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems, in: Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), 2004.

[17] R. E. Ladner, M. J. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838.

[18] Y. Makhlin, G. Schön, A. Shnirman, Quantum-State Engineering with Josephson-Junction Devices, Rev. Mod. Phys. 73 (2001) 357–400.

[19] C. Moler, C. van Loan, Nineteen dubious ways to compute the exponential of a matrix, SIAM Review 20 (4) (1978) 801–836.

[20] C. Moler, C. van Loan, Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later, SIAM Review 45 (1) (2003) 3–49.

[21] M. A. Nielsen, I. L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, Cambridge (UK), 2000.

[22] T. J. Rivlin, Chebyshev Polynomials, Wiley-Interscience, 1990.

[23] T. Schulte-Herbrüggen, A. K. Spörl, N. Khaneja, S. J. Glaser, Optimal Control-Based Efficient Synthesis of Building Blocks of Quantum Algorithms: A Perspective from Network Complexity towards Time Complexity, Phys. Rev. A 72 (2005) 042331.

[24] P. W. Shor, Polynomial-Time Algorithms for Prime Factorisation and Discrete Logarithm on a Quantum Computer, SIAM J. Comput. 26 (1997) 1484–1509.

[25] A. K. Spörl, T. Schulte-Herbrüggen, S. J. Glaser, V. Bergholm, M. J. Storcz, J. Ferber, F. K. Wilhelm, Optimal Control of Coupled Josephson Qubits, Phys. Rev. A 75 (2007) 012302, e-print: http://arXiv.org/pdf/quant-ph/0504202.

[26] R. van de Geijn, J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, Concurrency: Practice and Experience 9 (4) (1997) 255–274.

[27] H. Yuan, R. Zeier, N. Khaneja, Elliptic functions and efficient control of ising spin chains with unequal couplings, e-print: http://arXiv.org/pdf/0710.0075 (2007).