Fast 3D Block Parallelisation for the Matrix Multiplication Prefix Problem

Application in Quantum Control

K. Waldherr, T. Huckle, T. Auckenthaler, U. Sander, and T. Schulte-Herbrüggen

Abstract For exploiting the power of supercomputers like the HLRB-II cluster, developing parallel algorithms becomes increasingly important. The matrix prefix problem belongs to a class of issues lending themselves for parallelisation. We compare the tree-based parallel prefix scheme, which is adapted from 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 $2-3$ and also leads to less memory requirements. Unlike the tree-based scheme, the fine-grain approach enables many options in the choice of the number of parallel processors and shows a better speedup performance when increasing the matrix sizes. The usage of the fine-grain approach in a quantum control algorithm instead of the coarse-grain approach allows us both to deal with systems of higher dimensions and to choose a finer discretisation.

Introduction: The Prefix Problem

In general, the *prefix problem* is given as follows: Let $\circ$ be a binary operator and $\mathscr{A}$ a set, which is closed under the operator $\circ$. Furthermore, let $\circ$ be associative, i.e. the identity $(a \circ b) \circ c = a \circ (b \circ c)$ holds for all $a, b, c \in \mathscr{A}$. Then, for given elements $x_1, \ldots, x_M \in \mathscr{A}$, the prefix problem means the computation of all the products

$$y_i = x_1 \circ \cdots \circ x_i \quad . \tag{1}$$

Analogously, the *suffix problem* amounts to computing of all the products

K. Waldherr, T. Huckle and T. Auckenthaler
Dept. of Computer Science, TU Munich, D-85748 Garching, Germany, e-mail: huckle@in.tum.de

U. Sander and T. Schulte-Herbrüggen
Dept. of Chemistry, TU Munich, D-85747 Garching, Germany, e-mail: tosh@ch.tum.de

$$z_i = x_{i+1} \circ \cdots \circ x_M \quad . \tag{2}$$

In our applications, the elements x_i denote quadratic matrices and the operator '$\circ$' describes the multiplication of two matrices.

Scope and Organisation of the Paper

This account comprises two sections. First, we present two approaches for the parallelisation of the prefix problem, the coarse-grain tree-based and the fine-grain 3D block approach. The coarse-grain approach reorganises the numbering of the matrix multiplication in such a way that the products may be computed in parallel, whereas the fine-grain approach parallelises the individual matrix multiplications. In Sect. 1.1 we describe the two methods, demonstrate their pros and cons and underpin these statements with numerical results, i.e. speedup measurements taken on the ALTIX and computation time measurements on different architectures.

Second, we present applications showing how matrix methods improve the computation time of solutions to cutting-edge optimal quantum control problems. In turn, these control methods pave the way to finding optimised experimental steerings of quantum devices in realistic settings as they occur in a broad array of applications in quantum electronics, nanotechnology, spectroscopy, and quantum computation.

Hardware and Software Setup

The improvements in matrix multiplication methods directly translate into faster algorithms for the parallel prefix problem. As a main test bed, here we use high-dimensional matrices as occurring in large quantum systems. To this end, we extended our parallelised C++ code of the GRAPE package described in [5].

Computations were performed on the HLRB-II cluster currently providing an SGI Altix 4700 platform equipped with 9728 Intel Itanium2 Montecito Dual Core processors with a clock rate of 1.6 GHz, which give a total LINPACK performance of 63.3 TFlops/s. The sequential linear algebra tasks were executed by usage of the MATH KERNEL LIBRARY(MKL).

For comparing the performance of our program on different architectures, computations were also performed 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 linear algebra tasks were executed by the implementation of AMD'S CORE MATH LIBRARY (ACML).

1 Parallelising the Prefix Problem

With regard to parallelisation, the matrix prefix problem defined in Eq. 1 offers both a fine-grain and a coarse-grain approach.

1.1 Coarse-Grain versus Fine-Grain Approach

The idea of the fine-grain approach is to simply compute the matrix products $\Pi_{j=1}^{k} U_j$ sequentially for $k = 1,...,M$ and to parallelise the individual matrix multiplications. In contrast, the coarse-grain approach applies the following divide-and-conquer approach to compute a product $U_{k_1:k_2}$:

1. compute the products $U_{k_1:\kappa}$ for all $\kappa = k_1,...,\hat{k}-1$, with $\hat{k} = \lceil \frac{1}{2}(k_1+k_2) \rceil$.
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.

This coarse-grain approach can then be extended recursively to a tree-like multiplication scheme as given in Fig. 1, as was first presented in [9]. In order to compute all the interior products $U_{1:k}$ for $k = 1,...,M$, we may at most use $\frac{M}{2}$ processors in parallel. If a larger number of processors is available, the individual matrix multiplication itself would have to be parallelised as well. The tree-like coarse-grain approach offers some advantages: the communication pattern is simpler than in the fine-grain approach and we can expect good parallel speedups of the tree algorithm, because the individual matrix multiplications are strictly sequential. How-

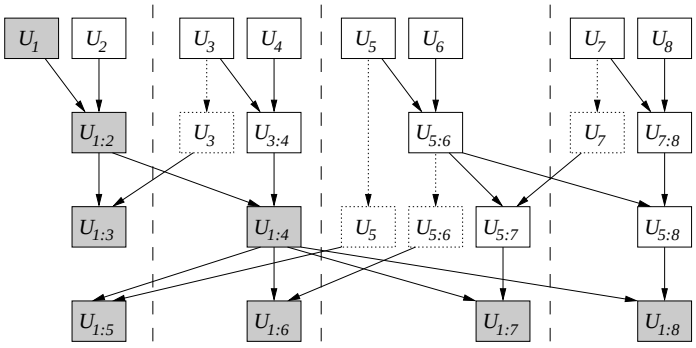


Figure 1 Divide-and-conquer scheme for the parallel prefix computation. The dashed lines define the processor scopes; solid lines denote communication between processors, and dotted lines and boxes indicate where matrices have to be retained for next-level computations. Note that the result matrices (grey boxes) are not balanced among the processes.

ever, the tree-like approach increases the total computational work by a logarithmic factor, because the parallel computation requires $\mathcal{O}(\log M)$ subsequent steps, which are given by the horizontal levels in Fig. 1. This additional overhead is saved in the fine-grain approach, where the individual matrix multiplications themselves are parallelised. Another disadvantage of the tree scheme is the fact, that the resulting matrices $U_{1:k}$ are not balanced among the processes: In the special case of $p = \frac{M}{2}$ parallel processes, the first process holds $\frac{M}{2}$ result matrices, whereas the last process holds only one of them. However, the balance missing here is recovered in the fine-grain approach. For a detailed analysis of the tree scheme, we refer the reader to our last report [11].

1.2 3D Block-Oriented Parallel Matrix Multiplication

In its most general form, the computation of the matrix product $C = AB$ 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**

(the matrix C is assumed to be initialised with zeros), where the C_{ik}, A_{ij}, and B_{jk} may be simple matrix elements or even matrix blocks. Note, that the body of the for-loop, i.e. all block operations of step 2, may be executed entirely in parallel. In Fig. 2, these block operations are visualised as a cube of size $n \times n \times n$, where the projections in the k-, i- and j- direction indicate which matrix blocks A_{ij}, B_{jk} and C_{ik} are to be accessed, respectively.

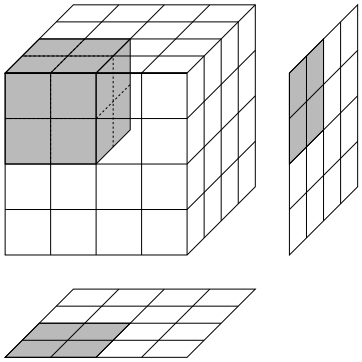


Figure 2 Block operations in the 3D block multiplication algorithm. The projections onto the planes below and right illustrate the accessed matrix blocks of the result matrix C and of one of the operand matrices, respectively.

Depending on the data distribution and also on the distribution of the block operations to the available processors, we may classify the algorithms for parallel matrix multiplication 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, these three classes of algorithms lead to different performance properties. Most importantly, 3D algorithms feature the lowest communication effort, namely $\mathcal{O}(n^2 p^{1/3})$ instead of $\mathcal{O}(n^2 p^{1/2})$ for $2D$, and $\mathcal{O}(n^2 p)$ for 1D algorithms (compare [6, 7]).

Under the influence of the performance study [2], where 3D algorithms showed the best runtime performance for our problem setting, we implemented a 3D block algorithm. The improved performance comes at the cost of a slightly higher memory requirement: $\mathcal{O}(n^2 p^{1/3})$ additional matrix elements have to be distributed on p processors, which is less than one additional matrix per processor. As we typically store several matrices (up to 256 in our examples in Sect. 1.3) on each CPU, this additional requirement is easily affordable. Details may be found in [1].

In the 3D block algorithm, the number D of blocks per index dimension is chosen such that D is the smallest power of 2, which fulfills $D^3 \geq p$ (p is 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 block 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).

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 Fig. 3.

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 such as 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 were stored on only one process each,

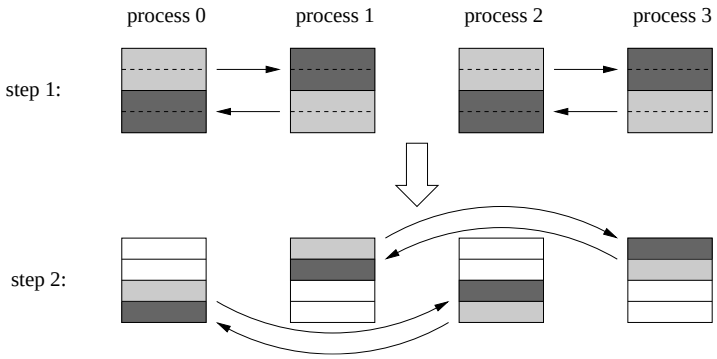


Figure 3 Communication pattern for the 3D block approach during the accumulation of results within a single matrix block C_{ik}. In each successive step, pairs of processes exchange and accumulate their partial results: the number of exchanged rows is halved and the distance between two communicating 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.

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 Fig. 3). 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 processor.

1.3 Numerical Results

In this section, we want to present some performance results of the 3D block approach. As already mentioned in the hardware setup at the beginning of our report, we evaluated these results both on the HLRB-II cluster and on an Infiniband cluster.

Figure 4 shows the speedups achieved when using 8, 16, 32, 64, 128 or 256 processors for $M = 2048$ matrices of size 512×512 and $M = 256$ matrices of size 1024×1024 respectively and when using 32, 64, 128, 256 or 512 processors for $M = 512$ matrices of size 2048×2048 and accordingly $M = 64$ matrices of size 4096×4096. We see that the speedups become increasingly better by augmenting the matrix size: for matrices of size 1024×1024, the achieved parallel efficiency on the HLRB-II is only about 30% for 256 processors, whereas for matrices of size 4096×4096, we get an efficiency of about 65% for 256 processors. This is both due to communication overhead and due to decreased performance of the sequential matrix multiplication for very small matrix blocks. The illustrated results also

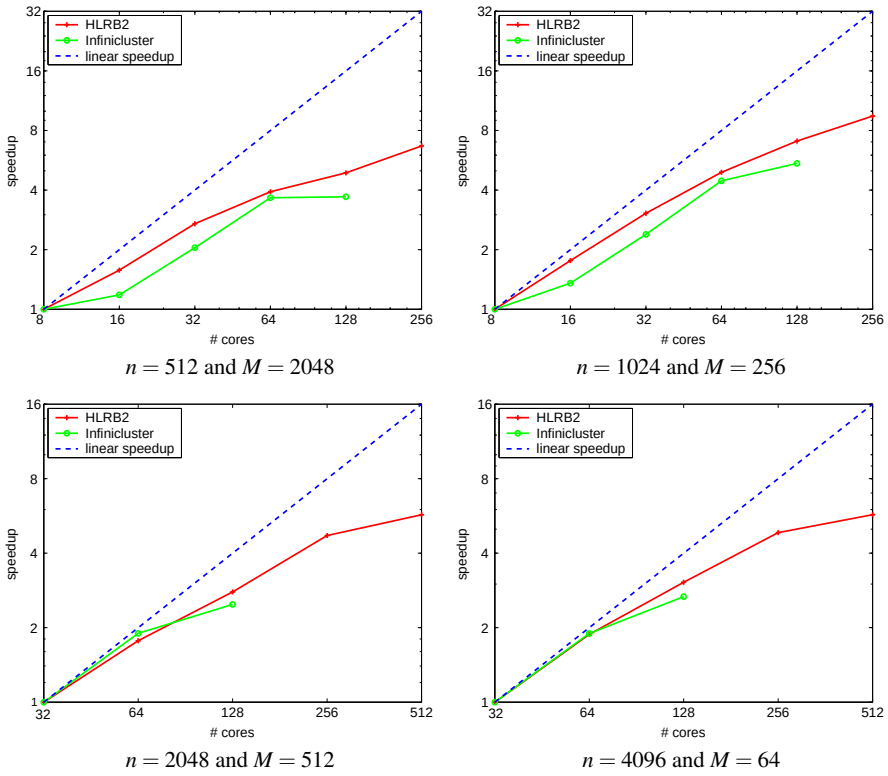
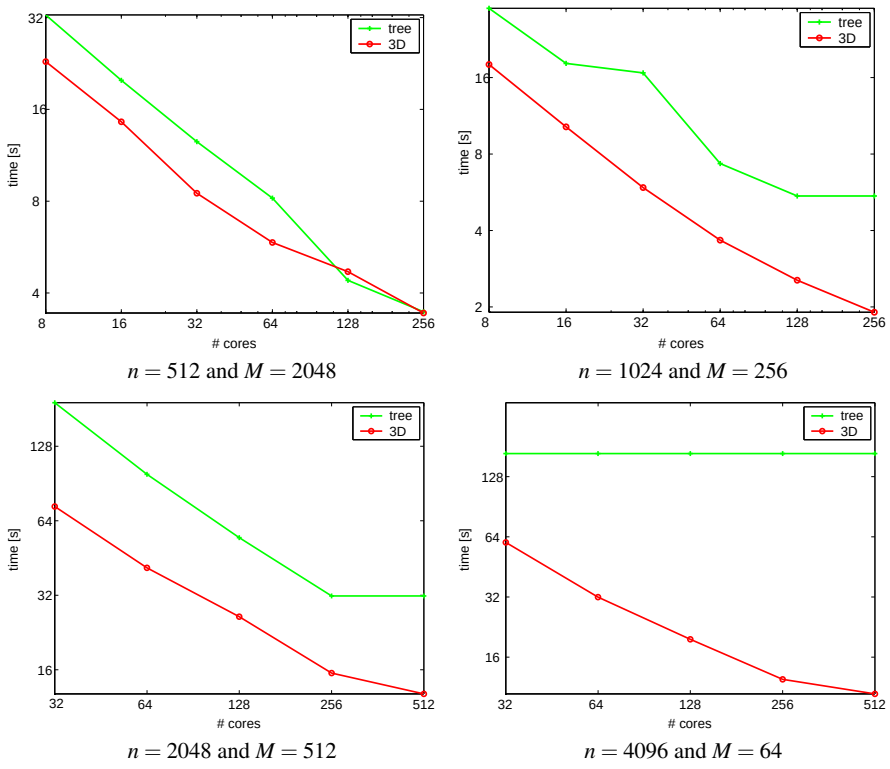


Figure 4 The speedups achieved on the HLRB-II cluster compared to the speedups on the Infiniband cluster. These speedups were measured for different configurations of the matrix size n and number M of matrices being multiplied. Please note, that the Infiniband cluster only offers 128 cores.

show that due to the better communication interconnect, the HLRB-II enables larger speedups than the Infiniband cluster.

For comparing the fine-grain to the coarse-grain approach in terms of runtime performance, we measured the computation times of both methods on the HLRB-II for the same problem settings and numbers of processes as in the previous speedup measurements. Despite the less-than-optimal speedups, the 3D block multiplication outperforms the parallel prefix scheme, which can be seen from Fig. 5. As expected, the performance advantage of the 3D block method grows for larger matrix sizes: for matrices of size 512×512, we get about the same computation times for 256 parallel processes, whereas for matrices of size 2048×2048, the 3D approach is more than 2 times faster than the tree-wise method. Recall that for the coarse-grain tree-like method, at most $\frac{M}{2}$ parallel processors may be used. This is why no further speedups are to be expected in the problem configurations of Fig. 5 c) and d).



$n = 512$ and $M = 2048$ $n = 1024$ and $M = 256$

$n = 2048$ and $M = 512$ $n = 4096$ and $M = 64$

Figure 5 Comparison of the tree-like and the 3D block approach due to computation time. These results were achieved on the HLRB-II cluster.

2 Application: Optimal Quantum Control

In view of novel quantum- and nano-technology, quantum control plays a key role for steering quantum hardware systems [4]. Moreover for exploiting the power of such quantum devices, one has to manipulate them by classical controls with the shapes of these controls critically determining the performance of the quantum system. Providing computational infrastructure for devising optimal shapes by using high-performance computer clusters is therefore tantamount to using the power of present and future quantum resources. To this end, numerical schemes like the GRAPE (GRadient Ascent Pulse Engineering) algorithm [8] have been introduced. — However, the task of finding optimised quantum controls is computationally highly demanding in the sense that the classical resources needed grow exponentially with the size of the quantum system.

The examples presented here refer to a particularly interesting class of quantum states known as *cluster states*. Their power lies in the fact that they are highly correlated (i.e. *entangled*). While preparing those states as initial states is challenging, it makes the actual quantum information processing step very efficient [3, 10], be-

cause the latter then reduces to local operations on each qubit. So minimising the time cost for this preparatory step is relegated to our numerical algorithm GRAPE, which is significantly improved by the fast matrix multiplication schemes presented.

2.1 Linear Algebra Tasks in the GRAPE *Algorithm*

The GRAPE algorithm provides a method for optimal quantum control based on gradient flows. In order to get a first impression about the complexity of the GRAPE

0: define as quality function $f\big(U(t_M)\big) := \mathrm{Re\ tr}\{U_G^\dagger U(t_M)\}$;
1: set initial control amplitudes $u_j^{(0)}(t_k)$ 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, take uniform $\Delta t := t_{k+1} - t_k$:

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

where $H_k := H_0 + \sum_j u_j(t_k) H_j$ comprises a non-switchable drift term H_0, the control amplitudes $u_j(t_k)$ and control Hamiltonians H_j.
3: likewise, starting with $T = t_M$ and $W(T) = e^{i\phi} \cdot U_G$ compute the back-propagation for all $t_M, t_{M-1}, \ldots, t_{k+1}$ (which we naturally label as $W^\dagger(t_k)$, because ideally $W^\dagger(t_k)U(t_k) = e^{-i\phi}\,\mathrm{I}$)

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

4: calculate $\frac{\partial f(U(t_k))}{\partial u_j} = \mathrm{Re\ tr}\left\{W^\dagger(t_k)\,(-i\Delta t\, H_j)U(t_k)\right\}$
5: with $u_j^{(1)}(t_k) = u_j^{(0)}(t_k) + \varepsilon \frac{\partial f}{\partial u_j}\,|_{t=t_k}$ update all the piece-wise constant Hamiltonians H_k and continue with step 2.

Algorithm 1: Gradient Flow Algorithm GRAPE for Optimal Quantum Control [8]

algorithm, we consider the pseudo code given in algorithm 1. It describes a step-wise optimisation in conjugate gradients: In each iteration step, we compute the Hamiltonian quantum evolution given by the matrix exponentials at every time step k. Then forward (step 2) and backward propagation (step 3) are given by a sequence of evolutions of the quantum system under M piecewise constant Hamiltonians H_k. The gradient $\frac{\partial f(U(t_k))}{\partial u_j}$ of the performance function f (i.e. the projection of the time evolution $U(T)$ onto the desired target $W(T)$) is given by the trace (calculated in step 4) and needed for the update of the control amplitudes in step 5. The control amplitudes may be initialised by a suitable guess for the initial values $u_j^{(0)}(t_k)$ in step 1 or by setting them to some constant.

From a computational point of view, the GRAPE algorithm makes heavy use of the following linear algebra tasks:

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).

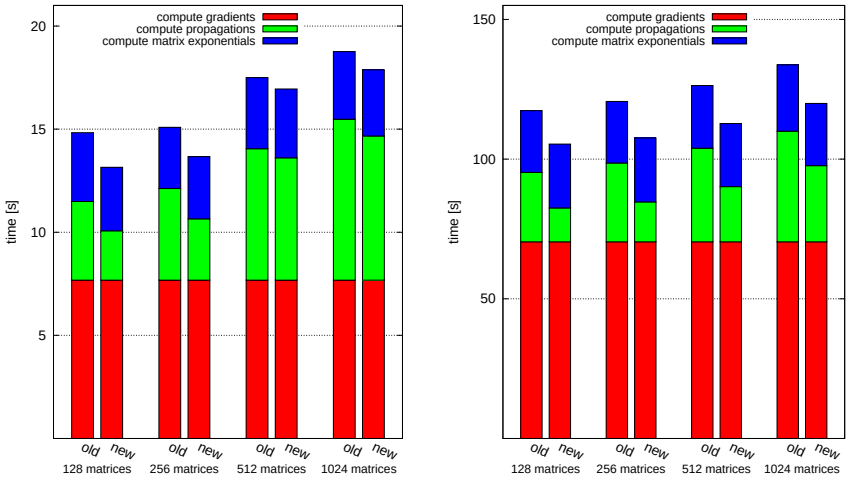


Figure 6 Runtime of one iteration of the GRAPE algorithm for different problem sizes. We compare the runtimes for using the parallel prefix scheme ('old') vs. using the 3D parallel matrix multiplication ('new') for systems with 9 and 10 spins, where the number M of matrices varies between 128 and 1024. The number of parallel processes is chosen as $p = \frac{M}{2}$.

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

The computation of the matrix exponentials was already an issue in the report [11] of 2008. By using a Chebyshev series approach instead of dealing with the eigendecomposition of the Hamiltonians, we could save a factor of 30 % in computation time without loss of accuracy. For details, see [1, 11].

Therefore, only the computation of the forward and backward propagation, which is an instance of the prefix problem Eq. 1 or the suffix problem Eq. 2, respectively, is left for computational improvement.

2.2 Numerical Results

For the computation of the forward and backward propagation, we may now use and compare the methods considered in Sect. 1. A 1D algorithm (see Sect. 1.2) had already been considered in the original work by Gradl et al. [5]. However, it was found to be inferior to the coarse-grain parallel prefix scheme, both with respect to runtime as well as due to increased memory and communication requirements. Therefore, the additional computation costs of $\log(M)$ matrix multiplications involved by the tree-like propagation were accepted in order to be able to deal with systems of larger size (see [11]). With the 3D block approach presented in Sect. 1.2,

we now have a second algorithm beside the tree-like propagation for the computation of the forward- and backward-propagation (steps 2 and 3 of algorithm 1).

Fig. 6 gives the runtimes for one complete iteration of the GRAPE algorithm on the HLRB-II cluster using the tree-like ('old') or the 3D block ('new') approach. For either setting, the total runtime is split up into three parts: computing the matrix exponentials, forward- and backward-propagation, and computation of the gradients (steps 4 and 5 of algorithm 1). The number p of processors was chosen as $p = \frac{M}{2}$, which means best performance for the tree-like method (cp. Sect. 1.1).

Using the 3D matrix multiplication gives a substantially better performance even for the 9-spin problem, where the matrix sizes are comparably 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 compared to the tree-like approach any other number of parallel processes would lead to even better results for the block approach and that due to Sect. 1.3, the speedup behaviour of the 3D method will become increasingly better when raising the number of spins (i.e. the matrix size).

After demonstrating benchmark results, we finally present a practical application. Table 1 summarises the computation times for the calculation of 100 iterations for two real problem settings, namely an 8-spin and a-10 spin instance of finding optimal controls for a quantum dynamic state-to-state transfer preparing cluster states.

Table 1 Parallel runtimes for the computation of the first 100 GRAPE iterations on p processors for different matrix sizes $2^n \times 2^n$ and number M of matrices. Note that n directly corresponds to the number of qubits (spins).

Problem Size			Parallel Runtime $[s]$	
matrix dimension $2^n \times 2^n$	# matrices M	# processes	tree	3D block
$n = 8$	64	32	324	362
$n = 8$	64	128	324	191
$n = 8$	128	64	388	391
$n = 8$	128	128	388	240
$n = 8$	256	128	836	427
$n = 10$	128	64	36842	21260
$n = 10$	128	128	36842	10609

3 Conclusions

Summing up, in order to fully exploit the power of the HLRB-II cluster, we devised a 3D approach for the parallel prefix multiplication of matrices. It efficiently parallelises the three loops occurring in matrix multiplication while keeping memory demands as well as communication costs low. Compared to the tree-like 'divide and conquer' matrix multiplication scheme (reported previously [11]), the 3D approach

presented here saves the additional logarithmic factor in the total computational work, provides more opportunities for parallelisation and promises good parallel speedups for large problem sizes. With matrix multiplication being in the core of many algorithms, we anticipate that the general scheme shown here awaits broad application for improving parallel prefix algorithms on high-speed clusters.

As an instance, we demonstrated the gain in CPU time in an algorithm optimising the preparation of quantum states in an important class of problems.

Acknowledgements

This work was supported in part by the integrated EU project QAP, by the Bavarian excellence initiative ENB in the PhD programme QCCC, and by *Deutsche Forschungsgemeinschaft*, DFG, within the collaborative research centre SFB-631.

References

1. Auckenthaler, T., Bader, M., Huckle, T., Spörl, A., Waldherr, K.: Matrix Exponentials and Parallel Prefix Computation in a Quantum Control Problem. Parallel Computing: Architectures, Algorithms and Applications, Special issue of the PMAA 2008 (submitted 2008)
2. Bader, M., Hanigk, S., Huckle, T.: 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, *NIC Series*, vol. 38, pp. 175–184 (2008)
3. Briegel, H.J., Raussendorf, R.: Persistent Entanglement in Arrays of Interacting Particles. Phys. Rev. Lett. **86**, 910–913 (2001)
4. Dowling, J., Milburn, G.: Quantum Technology: The Second Quantum Revolution. Phil. Trans. R. Soc. Lond. A **361**, 1655–1674 (2003)
5. Gradl, T., Spörl, A.K., Huckle, T., Glaser, S.J., Schulte-Herbrüggen, T.: Parallelising Matrix Operations on Clusters for an Optimal-Control-Based Quantum Compiler. Lect. Notes Comput. Sci. **4128**, 751–762 (2006). Proceedings of the EURO-PAR 2006
6. Gupta, A., Kumar, V.: Scalability of Parallel Algorithms for Matrix Multiplication. In: International Conference on Parallel Processing – ICPP'93, vol. 3, pp. 115–123 (1993)
7. Irony, D., Toledo, S., Tiskin, A.: Communication Lower Bounds for Distributed-Memory Matrix Multiplication. J. Parallel Distrib. Comput. **64**, 1017–1026 (2004)
8. Khaneja, N., Reiss, T., Kehlet, C., Schulte-Herbrüggen, T., Glaser, S.J.: Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms. J. Magn. Reson. **172**, 296–305 (2005)
9. Ladner, R.E., Fischer, M.J.: Parallel Prefix Computation. J. ACM **27**, 831–838 (1980)
10. Raussendorf, R., Briegel, H.J.: A One-Way Quantum Computer. Phys. Rev. Lett. **86**, 5188–5191 (2001)
11. Schulte-Herbrüggen, T., Spörl, A.K., Waldherr, K., Gradl, T., Glaser, S.J., Huckle, T.: in: High-Performance Computing in Science and Engineering, Garching 2007, chap. Using the HLRB Cluster as Quantum CISC Compiler: Matrix Methods and Applications for Advanced Quantum Control by Gradient-Flow Algorithms on Parallel Clusters, pp. 517–533. Springer, Berlin (2008)