UPPSALA
UNIVERSITET

IT Licentiate theses
2019-002

Inverse Factorization in Electronic Structure Theory: Analysis and Parallelization

ANTON G. ARTEMOV

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Inverse Factorization in Electronic Structure Theory: Analysis and Parallelization

Anton G. Artemov
anton.artemov@it.uu.se

June 2019

*Division of **Scientific Computing***
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in **Scientific Computing**

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

This licentiate thesis is a part of an effort to run large electronic structure calculations in modern computational environments with distributed memory. The ultimate goal is to model materials consisting of millions of atoms at the level of quantum mechanics. In particular, the thesis focuses on different aspects of a computational problem of inverse factorization of Hermitian positive definite matrices. The considered aspects are numerical properties of the algorithms and parallelization. Not only is an efficient and scalable computation of inverse factors necessary in order to be able to run large scale electronic computations based on the Hartree–Fock or Kohn–Sham approaches with the self-consistent field procedure, but it can be applied more generally for preconditioner construction.

Parallelization of algorithms with unknown load and data distributions requires a paradigm shift in programming. In this thesis we also discuss a few parallel programming models with focus on task-based models, and, more specifically, the Chunks and Tasks model.

Acknowledgments

Special thanks to my supervisors Dr. Emanuel H. Rubensson and Dr. Elias Rudberg, who have guided and still guiding me through my studies. Thanks to Prof. Maya Neytcheva for valuable comments. Thanks to my colleague, Anastasia Kruchinina, who helped me a lot in the early phase. Thanks to all people at the Division of Scientific Computing for making my work and being here nice and smooth.

List of Papers

This thesis is based on the following papers

I Emanuel H. Rubensson, Anton G. Artemov, Anastasia Kruchinina, Elias Rudberg. "Localized inverse factorization." Submitted.

II Anton G. Artemov, Elias Rudberg, Emanuel H. Rubensson. "Parallelization and scalability analysis of inverse factorization using the Chunks and Tasks programming model." Submitted.

Reprints were made with permission from the publishers.

Contents

Chapter 1

Introduction

In the last few decades electronic structure calculations have started to play an important role in various fields of science including materials science, biology and chemistry. The ultimate goal is to simulate materials or tissues consisting of thousands of atoms at the level, where quantum mechanics rules. However, the governing equations are too complicated to be solved analytically, therefore some simplifications are needed. There are two main approaches: to solve the original equations with reduced complexity numerical methods or to simplify the underlying model. The famous Hartree–Fock [17] method and Kohn–Sham [26] density functional theory belong to the second category.

From a computational point of view, the two methods are relatively close to each other. The complexity of the numerical methods used traditionally to solve the Hartree-Fock or Kohn-Sham equations is cubical, which literally means that if the system size is doubled, then the time to compute the solution becomes eight times longer. In the last 25-30 years there has been a significant progress in this field, and the complexity of the methods in many cases has been reduced to linear, which means that the solution time grows proportionally to the system size [8, 39, 12, 48, 41].

The complexity reduction allowed to apply the methods for relatively large systems. However, often the codes are limited to shared memory machines and their adoption to distributed memory environments significantly increases the development time and complexity. In many cases this step is done using conventional MPI, but other parallel programming models can be used to simplify the process and minimizes the chances of very common parallel problems like deadlocks and race conditions.

For efficient utilization of supercomputers with thousands or millions of cores, the choice of algorithm also plays an important role. As will be demonstrated in this thesis, in a parallel environment to use well-known and

popular algorithms might not be the best idea.

1.1 Thesis outline

The thesis is organized as follows: Chapter 2 gives a brief introduction to the electronic structure theory and the Hartree–Fock method, Chapter 3 describes the hierarchical representation of matrices used, Chapter 4 focuses on the problem of inverse factorization, Chapter 5 gives an overview of different parallel programming issues, models and, in particular, the Chunks and Tasks model.

Chapter 2

Electronic structure theory in brief

Material covered in this chapter is a very brief summary. A detailed overview can be found in [51] and [47].

2.1 The Schrödinger equation

The underlying physical problem consists of N identical charged particles, which can move (electrons) and M point charges (nuclei), which have fixed positions. Solving this problem, i.e. finding the positions of the charged particles which minimize the total energy, is equivalent to solving the N-electron Schrödinger equation:

$$\hat{H}\psi = E\psi, \tag{2.1}$$

where $\hat{H}$ is a Hamiltonian operator, ψ is the wave function and E is the eigenvalue. One can see, that this equation is an eigenvalue problem. The eigenfunction with the lowest energy E is the objective. The wave function is a function of the electron spatial positions and their spins. According to the Pauli exclusion principle, the wave function ψ must be antisymmetric with respect to exchange of any two electrons, i.e.

$$\psi(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_N) = -\psi(x_1, x_2, \ldots, x_j, \ldots, x_i, \ldots, x_N), \tag{2.2}$$

where $x_i = (r_i, \omega_i)$ is the combination of the coordinates and spin for the i-th electron.

2.2 The Slater determinant

One way to construct a wave function which satisfies (2.1) is to build it from one-electron functions. For instance, if only two electrons are present, then the wave function can be constructed as follows:

$$\psi(x_1, x_2) = \phi_1(x_1)\phi_2(x_2) - \phi_1(x_2)\phi_2(x_1) = \begin{vmatrix} \phi_1(x_1) & \phi_2(x_1) \\ \phi_1(x_2) & \phi_2(x_2) \end{vmatrix}, \tag{2.3}$$

where $\phi_i(x)$ is a one electron function of its coordinates and spin. As one can see, $\psi(x_1, x_2) = -\psi(x_2, x_1)$. This approach can be extended to any number of electrons as follows:

$$\psi(x_1, x_2, \ldots, x_N) = \begin{vmatrix} \phi_1(x_1) & \phi_2(x_1) & \ldots & \phi_N(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \ldots & \phi_N(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \ldots & \phi_N(x_N) \end{vmatrix}, \tag{2.4}$$

which is known as a Slater determinant [50]. Any linear combination of those for any choice of one electron functions $\phi_i(x)$ also satisfies condition (2.2). The Hartree–Fock approximation is trying to find the solution to (2.1) by varying one electron functions $\phi_i(x)$ in a single Slater determinant.

2.3 Basis functions

The one electron functions used in (2.4) are usually written as a product of spatial and spin parts:

$$\phi_i(x) = \varphi_i(r)\gamma_i(\omega), \tag{2.5}$$

where the spin part $\gamma_i(\omega)$ is completely described by two functions $\alpha(\omega)$ and $\beta(\omega)$. The spatial part, also known as *orbital*, is represented as a linear combination of basis functions:

$$\varphi_i(r) = \sum_{j=1}^{n} c_{i,j} b_j(r), \tag{2.6}$$

where the set of basis functions $\{b_j(r)\}$ forms a basis set. Commonly, these functions are chosen to be Gaussian with centering at nuclei positions. The exponential character of Gaussian functions leads to the important property of localization: at a certain distance from the center, the function becomes negligibly small. Practically one can assume that each function is zero outside some radius from the center point.

2.4 The Hartree–Fock method

The Hartree–Fock approximation of a wave function using the Slater determinant (2.4) can be treated differently. If one assumes that the electrons are paired, or, in other words, the number of electrons is even, then the method which utilizes this assumption is called the restricted Hartree–Fock method, whereas the unrestricted one handles the cases where the electrons are not paired. We will limit ourselves to a brief description of the restricted version only. A detailed description of both methods can be found in [51].

2.4.1 The restricted Hartree–Fock method

Assuming that the system consists of N electrons and n basis functions are used, one handles the three important matrices: the overlap matrix S, the density matrix D and the Fock matrix F. The overlap matrix S is constructed as

$$S_{i,j} = \int b_i(r) b_j(r) dr. \tag{2.7}$$

The density matrix D is related to the coefficients $c_{i,j}$ from equation 2.6:

$$D_{i,j} = 2 \sum_{k=1}^{N/2} c_{k,i} c_{k,j}. \tag{2.8}$$

Note that there are only $N/2$ summands, since the electrons are paired and thus there are only $N/2$ spatial functions aka orbitals for N electrons. The total electron density $\rho(r)$ can then be written as

$$\rho(r) = \sum_{i,j} D_{i,j} b_i(r) b_j(r). \tag{2.9}$$

The third ingredient, the Fock matrix F is the sum of three terms $F = T + V + G$, where T is the kinetic energy terms, V is the electron-nuclear attraction term and G is the two-electron part consisting of the Coulomb matrix J and the exchange matrix K. For simplicity, we skip the formulas for those terms, but it is important to mention that terms T and V are independent of the density matrix D, whereas term G does depend on it.

The total energy of the system can be written as

$$E = Tr\left(D(T+V)\right) + \frac{1}{2} Tr(DG). \tag{2.10}$$

As one can see, the electron density and the total energy both depend on the density matrix D, and the aim is to find the minimal energy and the

corresponding electron density. If the set of matrices D to try was small, then it would be relatively inexpensive to compute the minimal energy. In reality this is not the case and some iterative optimization scheme is applied, which is referred to as the self-consistent field procedure.

2.4.2 The self-consistent field procedure

The procedure consists of two steps: construction of the Fock matrix F for a given density matrix D, often referred to as $(D \to F)$, and construction of the new density matrix D for the computed Fock matrix F, which is referred to as $(F \to D)$. Then these two steps are repeated until the density matrix D no longer changes. Step $(F \to D)$ requires the solution of the generalized eigenvalue problem

$$FC = SC\Lambda, \tag{2.11}$$

and once the matrix of eigenvectors C is known, it can be used in equation (2.8). However, this approach is expensive and often can't provide sparsity in D, which is necessary in linearly scaling calculations. Another way to compute the density matrix D for a given Fock matrix F is to use another definition of the density matrix

$$D = \theta(\mu I - F), \tag{2.12}$$

where $\theta(x)$ is the Heaviside step function and approximate the latter using some kind of polynomials. The two most popular approaches are the recursive application of low-order polynomials aka density matrix purification [37] and approximation using Chebyshev polynomials [20, 35]. In both cases, multiplication of sparse matrices will be the dominant operation.

The convergence of the self-consistent procedure depends on the properties of the system, in particular on the gap between the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO). If this gap is vanishingly small, then the convergence of the procedure is slow or even cannot be reached [47].

There are some techniques to accelerate convergence of the self-consistent procedure [18]. The most prominent methods construct the new density matrix as a linear combination of density matrices from previous steps. The methods which utilize this strategy include DIIS (direct inversion of the iterative subspace) [40] and energy-DIIS [31].

2.5 Density functional theory

The Kohn-Sham density functional theory [26] shows that it is sufficient to know the electron density to describe a system of N electrons. Similarly to the Hartree–Fock method, it assumes the wave function in the form of a single Slater determinant. Similarly to the Hartree–Fock method, there are restricted and unrestricted versions. We'll very briefly touch the restricted one.

The counterpart of the Fock matrix F is the Kohn–Sham matrix $F^{KS} = T + V + J + \rho K + F^{XC}$, where J is the Coulomb matrix, K is the exchange matrix, ρ is a parameter and F^{XC} is the exchange-correlation matrix. Recall that matrices J and K are the parts of the two-electron matrix G from the Hartree–Fock method described in Section 2.4.1.

Then, the energy of the system is given as

$$E = Tr(D(T + V)) + \frac{1}{2}Tr\left(D(J + \rho K)\right) + E_{XC}. \tag{2.13}$$

With $E_{XC} = 0$ and $\rho = 1$ the expression becomes equivalent to (2.10). Computationally, one has to perform steps in a self-consistent fashion in order to find the minimal energy: construct D for computed F^{KS} and construct F^{KS} for computed D and then repeat until convergence. The second step ($F^{KS} \to D$) is exactly the same as in the Hartree–Fock method, whereas the first step ($D \to F^{KS}$) requires to compute matrices J, K and F^{XC}.

Chapter 3

Hierarchical matrix representation

In electronic structure calculations matrices are not sparse in the common sense. The notion of a sparse matrix somehow implies that there are just a few elements per row. In our case, this is not true, since the number of elements per row can be up to several thousands. This implies that popular sparse matrix formats like compressed sparse rows (columns) are not that suitable, since their strong point, i.e. compression, is lost. Also, as one will see later, there are operations which require matrix transpose. If a common format is used, than it becomes difficult to perform operations with transposed matrices.

The natural choice, successfully used by other researchers as well [7, 44, 43], is the hierarchical matrix representation. This means that a matrix is viewed as a matrix of other matrices forming a hierarchical structure. In our work, we used the simplest representation, where each level, except the last one (the leaf) of the hierarchy contains 2×2 matrices. The leaf level can be kept completely dense or use another sparse representation. An example of such representation can be found in Figure 3.1.

One should distinguish between the hierarchical matrix representation used here and so-called hierarchical matrices ($\mathcal{H}$-matrices) by Hackbusch [23]. $\mathcal{H}$-matrices are data-sparse approximations of matrices, which are not necessarily sparse by construction. The data-sparsity is introduced by low-rank approximations of some matrix blocks (admissible blocks), whereas the rest of the matrix (inadmissible) is kept dense. The blocks are organized in a tree-like structure giving rise to hierarchies, thus the name. Low-rank approximations of blocks become possible after applying some approximation technique to the underlying mathematical problem. So, in this sense, this class of methods belongs to the same cohort as the Hartree–Fock and

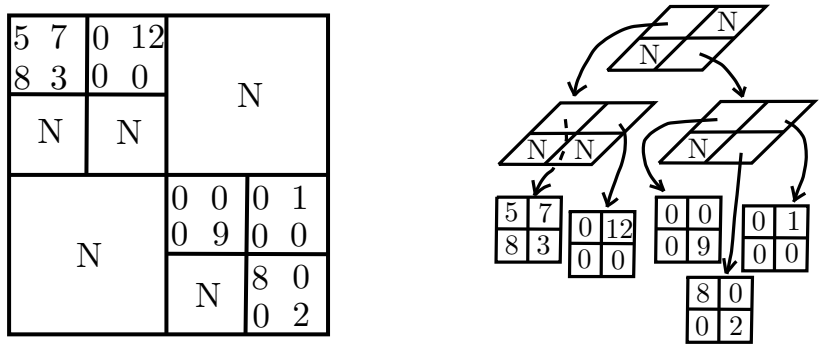

Figure 3.1: An example of hierarchical matrix representation. Leaf matrices have sizes 2×2, symbol N stands for null, depending on implementation, it might be a null pointer or some other empty object. Matrices marked with symbol N are not kept in the memory.

Kohn–Sham methods.

The key feature of $\mathcal{H}$-matrices is that the admissible blocks are kept and manipulated in factorized format. This makes it possible to perform matrix operations like addition, multiplication and even inversion at complexity, close to optimal. However, the construction of $\mathcal{H}$-matrices heavily relies on singular value decomposition, which is expensive. By choosing a different number of singular values, one can vary the accuracy of the matrix approximation by a $\mathcal{H}$-matrix.

The major difference of the hierarchical matrix representation from $\mathcal{H}$-matrices is that it is mostly a data format suitable for distributed environments and sparse matrices and not an approximation technique.

Chapter 4

Inverse factorization problem

4.1 Congruence transformation

Both Hartree–Fock and Kohn–Sham DFT result in a problem of the same form when computing the density matrix D from the Fock matrix F :

$$FC = C\Lambda \Longrightarrow D = CC^T \tag{4.1}$$

provided that the basis set used is orthonormal. However, in practice it is not always the case, and the basis set is not orthonormal. Then, the problem (4.1) becomes a generalized one:

$$FC = SC\Lambda, \tag{4.2}$$

where S is the basis set overlap matrix, which, in turn, is symmetric and positive definite. This generalized eigenvalue problem can be transformed to a standard form by a congruence transformation:

$$FC = SC\Lambda \Longrightarrow \left(Z^T F Z\right) C = C\Lambda, \tag{4.3}$$

where Z is such that $S^{-1} = ZZ^T$, i.e. some inverse factor of S.

The congruence transformation (4.3) is applied before every ($F \to D$) step of the self-consistent procedure and its inverse is then applied after a new density matrix D has been computed. The initial basis set is not necessarily orthonormal and the transformation changes it to an orthonormal one and thus allows to use well-established methods with controlled forward error [46], use precise stopping criteria [29], reduce the memory usage and decrease computational costs compared to non-orthonormal basis set. The calculation of an inverse factor is done only once before starting the self-consistent loop and it might seem that this procedure is not an important part of the whole machinery, since it usually occupies a small part of the total

computational time, but as it will be demonstrated in this thesis, popular algorithms are not always suitable for parallelization in distributed environments and become a bottleneck for the whole process for large enough systems.

4.2 Choice of the inverse factor

The inverse factor is not unique. In principle, any factor Z such that $S^{-1} = ZZ^T$ should work. Commonly, the inverse square root (Löwdin) [24, 33] or inverse Cholesky factor [11] are used. These factors have an important feature of decay of matrix elements with atomic separation [5]. However, classical methods to compute those factors have cubic complexity.

We would like to be able not only to compute a factor with decaying elements, but also to do that in time directly proportional to the number of basis functions and to get an inverse factor sufficiently sparse in order to perform computations efficiently.

4.3 Inverse Cholesky factorization

The computation of the inverse Cholesky factor using the AINV algorithm [3] has been commonly used in electronic structure calculations [11, 34]. There are several variants of the algorithm, including, for instance, a recursive one [44] and a blocked one [4].

For small and medium-size problems the algorithm is fast, but, as shown in Paper II, it has very limited parallelism and thus cannot be efficiently used in large scale calculations in a distributed environment.

4.4 Iterative refinement with a scaled identity

Iterative refinement with a scaled identity as a starting guess is another approach which has been popular over the past decades. It computes the inverse square root using the procedure of iterative refinement [38].

Assuming that some initial guess Z_0 is given such that

$$\|I - Z_0^T S Z_0\|_2 \leq 1, \tag{4.4}$$

the method constructs a sequence of factor Z_i such that the factorization error $\delta_i = I - Z_i^T S Z_i \longrightarrow 0$ as $i \longrightarrow \infty$. The method can be derived from the Newton-Schultz iteration for the sign matrix function.

In Paper I we are using a localized refinement approach, which can be used if it is known that only a small part of matrix elements has to be

updated, thus reducing computational costs. We show that the localized iterative refinement procedure produces an inverse factor, which, depending on the starting guess, has the property of decay of matrix elements with respect to distance between atoms.

The choice of Z_0 pays an important role. A good and reliable solution is to use a scaled identity matrix as a starting guess. This approach has been used by many researchers in the area [24, 55].

4.5 Localized inverse factorization

The localized inverse factorization proposed in Paper I is a divide-and-conquer method based on a combination of localized iterative refinement and a recursive decomposition of the matrix [45].

On each level of the hierarchy, the index set of the matrix is split into two parts, giving a binary principal submatrix decomposition. Then, inverse factors of the two submatrices lying on the main diagonal are computed and combined together as a starting guess for refinement. This procedure is applied recursively at all levels except the leaf one, where matrices are small enough to apply a direct method. The localized refinement procedure and the localized construction of starting guesses give the name for the method.

In Paper I we show that the cost of combining inverse factors of the submatrices is negligible compared to the overall cost for systems of sufficiently large sizes. It is also shown that the resulting inverse factor does have the decay property of the matrix elements, similarly to iterative refinement procedure.

One of the most important features of the algorithm (both the regular version in [45] and the localized version in Paper I) is that subproblems are absolutely independent on each level in the hierarchy and thus embarrassingly parallel. As demonstrated in Paper II, the localized inverse factorization algorithm has an advantage over the localized iterative refinement with a scaled identity as a starting guess in the amount of data sent during the computations. It is crucial since data movement is considered to be expensive, and thus its reduction can gain in performance.

4.6 A note on preconditioning

The main motivation for developing sparse inverse factorization methods is the need in efficient and scalable methods for electronic structure calculations. However, these methods can be easily applied to construct a preconditioner for iterative solutions methods with a symmetric positive definite matrix.

Given a linear system of equations

$$Ax = b, \tag{4.5}$$

the preconditioned system with a preconditioner C is

$$C^{-1}Ax = C^{-1}b. \tag{4.6}$$

In this case, the preconditioner C approximates A. The system (4.6) is never formed explicitly, instead, an iterative solution method of choice operates with matrices A and C^{-1} internally. The preconditioner C should satisfy certain requirements: the condition number of the matrix $C^{-1}A$ should be small, preferably $O(1)$, it should be cheap to construct C and the system with C should be easier to solve than the original one. An additional requirement is that the construction of the preconditioner must be parallelizable.

In case when A is symmetric positive definite, there are many techniques how to compute a preconditioner, the most prominent being the incomplete Cholesky factorization, multigrid methods and domain decomposition methods [21, 54].

Another choice of a preconditioner is a sparse approximate inverse, i.e. C approximates A^{-1}. The corresponding preconditioned system then looks like

$$CAx = Cb. \tag{4.7}$$

In this case, an iterative solution method performs matrix-vector multiplications instead of solving linear systems. One of the problems arising when constructing an approximate inverse is that in general the inverse of a sparse matrix is not sparse.

In [27, 28] a general framework is introduced that guarantees that the constructed approximate inverse of a symmetric positive definite matrix is also symmetric and positive definite. This is achieved by constructing C in a factorized form, minimizing the weighted Frobenius norm $F_W(C) = \|I - CA\|_W^2 = Tr(I - CA)W(I - CA)^T$, where W is a symmetric positive definite matrix and for $W = I$ the norm becomes the standard Frobenius norm. The sparsity pattern of C should be given, and then the entries of C are computed as parameters which minimize the weighted Frobenius norm $F_W(C)$. The drawback of this method is that one has to choose the sparsity pattern of the approximate inverse in advance, an it is not clear how to do that. Grote and Huckle [22] suggested a rather involved way to select sparsity patterns. Other methods consider sparsity patterns of powers of A [49]. The orthogonalization-based method [3] operates with columns separately and thus lacks parallelism, as it is demonstrated in Paper II.

The localized inverse factorization can be used to efficiently compute approximate inverse factors and thus the approximate inverse itself. There is no need to pre-determine the sparsity pattern or use a complicated strategy like in [22]. The procedure at least preserves symmetry.

Chapter 5

Parallelization aspects

We have already mentioned that methods used in electronic structure calculations have achieved linear scaling with respect to system size, i.e. the computational load increases in direct proportion to the size of considered chemical system. However, in modern conditions, methods also should be parallelizable in order to be able to explore large systems on supercomputers.

The development history of parallel computers first resulted in so-called shared memory machines, where a multi-core CPU or several CPUs have access to the same memory. Later, these machines became to be used as building blocks for larger machines as nodes. At this level, the global memory is no longer shared, but the information can be sent from one node and received at another. The trend of the last decade is to augment machines with accelerator boards such as Nvidia GPUs or Intel XeonPhi co-processors. These devices are also parallel machines, but with another paradigm: they use a bunch of relatively weak processing units to perform similar operations on data arrays. As the reader might guess, programming a modern parallel computer, which consists of nodes and accelerators, is an art.

Parallel programming has a long development history exactly like conventional programming, which started with machine languages, which have been replaced by assembly languages because of code complexity. Then, high-level languages started to play the dominant role. It is important to remember that high-level languages are anyway translated to assembly codes, and those are translated to machine codes. The reason is simple - machines understand only machine codes.

Somehow a similar process can be seen in the development of parallel programming and models used there.

5.1 Parallel programming models

First of all let us decide, why parallel programming models are necessary and then what is a parallel programming model, what is the purpose?

From our point of view, the necessity in models is not difficult to explain: they are required to simplify the creation of the code, increase performance of the programs and the programmer and make the code easier to understand. Then, a notion of a parallel programming model is a bit tricky. Are they models **of** parallel programming or models **for** it? We believe, that the latter is correct, and that a model should represent real-world concurrency and parallelism.

Historically, the first and most known concept of concurrency is a *thread*. A similar concept of a *process* is utilized in distributed programming. The well-known definition of a process is "a program when running", and a thread is "a lightweight process." Well, these concepts do not even vaguely resemble the real world concurrency and they are rich sources of nondeterminism [32]. When working with threads or processes, a variety of specialized tools like mutexes, semaphores etc. is used to reduce the amount of nondeterminism.

The need of better representation of concurrency and alleviating the programmer's responsibilities brings us to other concepts and models based on those concepts.

MPI. The Message Passing Interface [53] is not a parallel programming model or a language. It is just a standardized set of routines for passing messages between simultaneous processes. However, it is worth mentioning in this context, because is has became something similar to assembly languages in conventional programming. Most of implementation of other models rely on MPI internally.

Partitioned global address space models (PGAS). This approach tries to extend the concept of shared memory to distributed machines by creation of a global address space. Each process handles a part of that space, and communication between processes is made by updating the global memory. Examples of models which utilize this concept are Linda [10], Global Arrays [36], Hierarchical Place Trees [56], X10 [13] and Unified Parallel C [30]. To certain extent PGAS is supported in Charm++ [25] and OmpSs [15].

The idea behind the PGAS concept is simple: for the user the memory looks like a large continuous storage, but in fact, the runtime system decides where to place particular pieces of data and when and how to deliver it to the process which puts a request, and how to prevent race conditions. This definitely brings significant overhead. Moreover, there might be a question

if there is a central node who manages the global memory. If so, then it is the weak point of the design.

Domain-specific models. In TensorFlow [1] a computation is represented as a directed graph, where the nodes are the operations, whereas the information floating along the edges is tensors. Most tensors live only between two nodes, but the user can also utilize a mutable tensor, which has a longer lifetime. The main purpose of this model and its implementation is machine learning.

Another example of a domain specific model is the famous MapReduce [14] model, which was designed for processing large amounts of data. In this model, a set of worker machines perform tasks of two types, either map (compute intermediate key/value pair) or reduce (merge values with the same key to a smaller set). The master node keeps the statues of worker's tasks and re-distributes the work in case of failure.

As the reader might guess, TesorFlow and MapReduce are not generally applicable, but can be used in some applications.

Task-based models. A *task* can be defined in different ways. When talking about the real world, it can be defined as a piece of work. In programming, it can be defined as a large enough computation.

Tasks seem to represent the real world concurrency better than processes or threads, because tasks in programming do have a real-life counterpart, whereas processes and threads do not.

Historically, one of the first task-based models was Cilk [6], although it did not use the notion of a task and did use threads, conceptually it was a task-based model. It is worth mentioning that Cilk was developed for shared memory machines. The computation was represented as a dynamically unfolding directed acyclic graph. The authors of the Cilk model also suggested a model for execution time in such environments, which is directly related to the concept of the critical path used in Paper II. Another milestone is that the load balancing mechanism based on work stealing became very popular after it had been used in Cilk.

Later, other task-based models were developed. Examples are Concurrent Collections [9] (to some extent), DuctTeip [57], Hierarchical Place Trees [56], StarPU [2], Sequoia [16], OmpSs [15], XKaapi [19] and SuperGlue [52]. Not all of these models are for distributed memory, for instance SuperGlue, XKaapi, Cilk, and not all of them support dynamic parallelism (i.e. running tasks inside tasks). But all these models do represent the concurrency of the real world better than threads or processes.

5.2 Some issues of parallel programming

Apart from the problem with choosing the right abstractions, there are other issues in parallel programming, which are not always obvious to anyone familiar only with sequential programming.

Load balancing. Load balancing is an important problem to solve when running the code on supercomputers. Not only is it important for saving the time of the user, but also economically, by minimizing the idle time of available machines and thus costs.

In PGAS models the load balancing implicitly follows from the data distribution, which should be done by the runtime system. In most task-based models the concept of work-stealing is utilized. Basically it means that an idle worker tries to steal a task from another worker's queue of tasks. Most of the models do not consider locality issues in this approach, however, XKaapi [19] does so. In most domain-specific models a central node is responsible for load balancing.

Determinism. The notion of *determinism* means that a program always gives the same output provided that the input does not change. Lee [32] identified nondeterminism (i.e. the lack of determinism) as the main problem of concurrent programming.

Nondeterminism usually comes from a race condition, i.e. simultaneous modification of shared memory. The cure is to use a mutex mechanism. In distributed memory it is usually caused by the messages arriving in a wrong order. It is clear that with conventional tools like OpenMP and MPI it is very easy to end up with a nondeterministic code.

The models discussed above handle this issue differently. Some of them are provably deterministic like Concurrent Collections [9], some are deterministic if tasks are deterministic, see for instance StarPU [2] or MapReduce [14], and the rest have at least one source of non-determinism like user-defined access types, overlapping memory regions and so on. So, it would be a good idea to have a model like the Concurrent Collections, which by design is absolutely deterministic.

Deadlocks. A *deadlock* can be defined as a state of a program such that no process or worker can proceed further because of blocking by others. A rich source of deadlocks is global synchronisation.

Only a few of the models mentioned above are deadlock-free, for instance MapReduce [14], Sequoia [16] and Concurrent Collections [9]. The rest are not deadlock-free due to global synchronisation and possible out-of-order message handling.

Fault tolerance. By this one should understand the following: a failure of a single or multiple processes or nodes should not result in a failure of the whole program. Literally this means that the process or node which failed should be re-started on the fly and continue without any implications on others.

Explicitly this issue is addressed by a few models. In the MapReduce model [14], if a worker fails, its task is completely re-executed. The master node saves own state at checkpoints. In TensorFlow [1], the complete graph of computation is re-executed in case of failure. Charm++ has a built-in checkpoint mechanism. Note that the first two models were developed by the Google corporation and the Charm++ model has a long history (almost 25 years) and a large community.

Scalability. Obviously, models for shared memory are not scalable. PGAS models are more scalable, but the overhead of managing the global memory limits the scalability. Models for distributed memory should, in theory, scale very well. Some of those do have a master node, like the MapReduce model [14], but, due to the nature of the model, its influence is somehow minimized.

Easiness to use. Last, but not the least issue, is how easy it is to write programs using a particular model. Programmers tend to use the same tools they have become used to and most of the models require a paradigm shift. However, if this shift somehow is not far from a well-known tool or language, then the model can be successfully used.

From this perspective, MPI is an ultimate tool similar to assembly languages. It is powerful and almost everybody uses it. Then, PGAS models are close to popular OpenMP, thus the shift is not a problem. The task-based models are more complicated, since they do require another way of thinking.

5.3 The Chunks and Tasks programming model

The Chunks and Tasks model, suggested by Rubensson and Rudberg [42], is defined by a C++ application programming interface.

The two basic abstractions, which gave the name for the model are chunks and tasks. A piece of data is encapsulated as a chunk and a piece of work - as a task. It makes it possible to take advantage of parallelism in two dimensions, both in data and in work.

The user is responsible for preparing the chunk and registering it to the library. Once it is ready, the user obtains an identifier of it and the chunk becomes immutable, like in Concurrent Collections [9]. The chunk

identifiers can be part of other chunks creating a hierarchical structure or can be provided as input to tasks.

The work is split into small tasks. A task type is defined by a number of input chunk types, the work to be performed and a single output chunk type. All tasks eventually are converted into chunks. Chunk content can only be accessed when the corresponding chunk identifier is provided as an input to a task, except in the main program. It is not possible to run a task without having all the input chunks computed. Dynamic parallelism is supported.

The CHT-MPI library, which is the pilot implementation of the model, addresses the load balancing issue in the following way: when running, the program is executed by a group of worker processes, which attempt to steal tasks from colleagues when becoming idle, similarly to Cilk [6], XKaapi [19] and some other models. The model is fully deterministic due to immutability of data. Deadlocks are not an issue, since there is no global synchronisation involved. Unfortunately, The CHT-MPI implementation is not fault tolerant right now, but, in principle, this issue can be also addressed. The model is scalable, because it works in distributed environments and there is no master node. Our experience suggests that it takes significant effort for a programmer to switch from conventional programming to Chunks and Tasks.

The model is used to parallelize algorithms in Paper II.

Comments on author's contributions to the papers

1. I participated in discussions which led to Paper I, developed and tested numerical code in MATLAB, participated in writing and editing of the manuscript.

2. I developed the code, contributed to the Chunk and Tasks matrix library (CHTML), performed the tests on the supercomputer, processed the data and wrote a significant part of Paper II.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comp.-Pract. E.*, 23 (2):187–198, 2011.

[3] Michele Benzi, Carl D Meyer, and Miroslav Tuma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Sci. Comput.*, 17(5):1135–1149, 1996.

[4] Michele Benzi, Reijo Kouhia, and Miroslav Tuma. Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics. *Comput. Method. Appl. M.*, 190(49-50):6533–6554, 2001.

[5] Michele Benzi, Paola Boito, and Nader Razouk. Decay properties of spectral projectors with applications to electronic structure. *SIAM Rev.*, 55(1):3–64, 2013.

[6] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *J. Pallel Distr. Comput.*, 37(1): 55–69, 1996.

[7] Nicolas Bock and Matt Challacombe. An optimized sparse approximate matrix multiply for matrices with decay. *SIAM J. Sci. Comput.*, 35(1): C72–C98, 2013.

[8] DR Bowler and Tsuyoshi Miyazaki. Methods in electronic structure calculations. *Rep. Prog. Phys.*, 75(3):036503, 2012.

[9] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent Collections. *Sci. Programming-Neth*, 18(3-4):203–217, 2010.

[10] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. The Linda alternative to message-passing systems. *Parallel Comput.*, 20(4):633–655, 1994.

[11] Matt Challacombe. A simplified density matrix minimization for linear scaling self-consistent field theory. *J. Chem. Phys.*, 110(5):2332–2342, 1999.

[12] Matt Challacombe, Eric Schwegler, and Jan Almlöf. Fast assembly of the Coulomb matrix: A quantum chemical tree code. *J. Chem. Phys.*, 104(12):4685–4698, 1996.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.*, 21(02):173–193, 2011.

[16] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[17] Vladimir Aleksandrovich Fock. Fundamentals of quantum mechanics. *Mir Publishers, Moscow, 1983. Translation from the 2-nd edition.*, 1978.

[18] Alejandro J Garza and Gustavo E Scuseria. Comparison of self-consistent field convergence acceleration techniques. *J. Chem. Phys.*, 137(5):054110, 2012.

[19] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS),*

2013 IEEE 27th International Symposium on, pages 1299–1308. IEEE, 2013.

[20] Stefan Goedecker and Luciano Colombo. Efficient linear scaling algorithm for tight-binding molecular dynamics. *Phys. Rev. Lett.*, 73(1): 122, 1994.

[21] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

[22] Marcus J Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comput.*, 18(3):838–853, 1997.

[23] Wolfgang Hackbusch. *Hierarchical matrices: algorithms and analysis*, volume 49. Springer, 2015.

[24] Branislav Jansík, Stinne Høst, Poul Jørgensen, Jeppe Olsen, and Trygve Helgaker. Linear-scaling symmetric square-root decomposition of the overlap matrix. *J. Chem. Phys.*, 126(12):124104, 2007.

[25] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[26] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1133, 1965.

[27] L Yu Kolotilina and A Yu Yeremin. On a family of two-level preconditionings of the incomplete block factorization type. *Russ. J. Numer. Anal. M.*, 1(4):293–320, 1986.

[28] L Yu Kolotilina and A Yu Yeremin. Factorized sparse approximate inverse preconditionings I. Theory. *SIAM J. Matrix Anal. A.*, 14(1): 45–58, 1993.

[29] Anastasia Kruchinina, Elias Rudberg, and Emanuel H Rubensson. Parameterless stopping criteria for recursive density matrix expansions. *J. Chem. Theory Comput.*, 12(12):5788–5802, 2016.

[30] William Kuchera and Charles Wallace. The upc memory model: Problems and prospects. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 16. IEEE, 2004.

[31] Konstantin N Kudin and Gustavo E Scuseria. Converging self-consistent field equations in quantum chemistry–recent achievements

and remaining challenges. *ESAIM-Math. Model. Num.*, 41(2):281–296, 2007.

[32] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[33] Per-Olov Löwdin. Quantum theory of cohesive properties of solids. *Advances in Physics*, 5(17):1–171, 1956.

[34] John M Millam and Gustavo E Scuseria. Linear scaling conjugate gradient density matrix search as an alternative to diagonalization for first principles electronic structure calculations. *J. Chem. Phys.*, 106(13): 5569–5577, 1997.

[35] Stephan Mohr, William Dawson, Michael Wagner, Damien Caliste, Takahito Nakajima, and Luigi Genovese. Efficient computation of sparse matrix functions for large-scale electronic structure calculations: the CheSS library. *J. Chem. Theory Comput.*, 13(10):4684–4698, 2017.

[36] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, 10(2):169–189, 1996.

[37] Anders MN Niklasson. Expansion algorithm for the density matrix. *Phys. Rev. B*, 66(15):155115, 2002.

[38] Anders MN Niklasson. Iterative refinement method for the approximate factorization of a matrix inverse. *Phys. Rev. B*, 70(19):193102, 2004.

[39] Itai Panas and Jan Almlöf. A fragment multipole approach to long-range Coulomb interactions in Hartree–Fock calculations on large systems. *Int. J. Quantum Chem.*, 42(4):1073–1089, 1992.

[40] Péter Pulay. Convergence acceleration of iterative sequences. the case of scf iteration. *Chem. Phys. Lett.*, 73(2):393–398, 1980.

[41] Emanuel H. Rubensson. *Matrix Algebra for Quantum Chemistry.* PhD thesis, KTH Royal Institute of Technology, 2008.

[42] Emanuel H. Rubensson and Elias Rudberg. Chunks and Tasks: A programming model for parallelization of dynamic algorithms. *Parallel Comput.*, 40:328–343, 2014. doi: 10.1016/j.parco.2013.09.006.

[43] Emanuel H Rubensson and Elias Rudberg. Locality-aware parallel block-sparse matrix-matrix multiplication using the Chunks and Tasks programming model. *Parallel Comput.*, 57:87–106, 2016.

[44] Emanuel H Rubensson, Elias Rudberg, and Paweł Sałek. A hierarchic sparse matrix data structure for large-scale Hartree-Fock/Kohn-Sham calculations. *J. Comput. Chem.*, 28(16):2531–2537, 2007.

[45] Emanuel H Rubensson, Nicolas Bock, Erik Holmström, and Anders MN Niklasson. Recursive inverse factorization. *J. Chem. Phys.*, 128(10): 104105, 2008.

[46] Emanuel H Rubensson, Elias Rudberg, and Paweł Sałek. Density matrix purification with rigorous error control. *J. Chem. Phys.s*, 128(7): 074106, 2008.

[47] Elias Rudberg. *Quantum Chemistry for Large Systems.* PhD thesis, KTH Royal Institute of Technology, 2007.

[48] Elias Rudberg and Paweł Sałek. Efficient implementation of the fast multipole method. *J. Chem. Phys.*, 125(8):084106, 2006.

[49] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[50] John C Slater. The theory of complex spectra. *Phys. Rev.*, 34(10):1293, 1929.

[51] Attila Szabo and Neil S Ostlund. *Modern quantum chemistry: introduction to advanced electronic structure theory.* Courier Corporation, 2012.

[52] Martin Tillenius. SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM J. Sci. Comput.*, 37(6):C617–C642, 2015.

[53] David W Walker and Jack J Dongarra. MPI: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[54] Andy J Wathen. Preconditioning. *Acta Numerica*, 24:329–376, 2015.

[55] HJ Xiang, Jinlong Yang, JG Hou, and Qingshi Zhu. Linear scaling calculation of band edge states and doped semiconductors. *J. Chem. Phys.*, 126(24):244707, 2007.

[56] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A portable abstraction for task parallelism and data movement. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 172–187. Springer, 2009.

[57] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. DuctTeip: A task-based parallel programming framework for distributed memory architectures. Technical report, Technical Report 2016-010, Uppsala University, Division of Scientific Computing, 2016.