Algorithmic Rethinking and Code Reengineering
for Truly Massively Parallel \textit{ab initio} Molecular Dynamics Simulations

Costas Bekas and Alessandro Curioni
IBM Research - Zurich
Säumerstrasse 4, Rüschlikon, 8803, Switzerland
E-mail: \{bek,cur\}@zurich.ibm.com

\textit{Ab initio} molecular dynamics simulations are indispensable tools in the hands of researchers and practitioners, that have pushed the frontiers of knowledge to a large extent. They have been very successful in diverse fields ranging from solid-state physics, to amorphous materials and liquids to biophysics and biochemistry. This power though comes at a staggering computational cost, thus \textit{ab initio} molecular dynamics codes were always at the forefront of High Performance Computing. Indeed, simulations of this kind have benefitted tremendously from the advent of massively parallel machines. In this work we provide concrete examples of the systematic work in algorithmic rethinking and code re-engineering that is required to bring these codes to the next level and render them ready to use machines with millions of threads. We believe the lessons learned to serve as examples for future significant improvements.

1 Introduction

\textit{Ab initio} molecular dynamics is the combination of first-principles electronic structure methods with molecular dynamics based on Newton’s equation of motion. The use of electronic structure methods to calculate the interaction potential between atoms overcomes the main shortcomings of the otherwise highly successful pair potential approach. In particular, with \textit{ab initio} methods many-body effects are included, they are parameter-free, and are able to adjust to new chemical situations that may be encountered during a simulation, for example when chemical reactions or structural phase transitions occur. In their seminal paper\textsuperscript{1}, Car and Parrinello introduced a new method that allows the efficient propagation of the electronic wave function together with the atomic cores. Although the method is very general, it is primarily used together with the Kohn-Sham approach to density-functional theory. The method has proved to be valuable in many fields. Recent applications include topics in traditional solid-state physics, surface science, interfaces, glasses and amorphous systems, liquids and solutions, catalysis and other chemical reactions, as well as problems from biophysics and biochemistry. For overviews of applications, see recent review papers\textsuperscript{2-4}

The combination of a computationally demanding electronic structure method with molecular dynamics requiring thousands of force evaluations render \textit{ab initio} molecular dynamics simulations highly dependent on high-performance computing resources. Many parallel implementations, following various strategies, have been reported in the literature\textsuperscript{2,6-12} over the past decade. To be able to push the applications from originally a few atoms to now routinely several hundreds of atoms, it was instrumental to adapt algorithms and implementations to modern massively parallel architectures\textsuperscript{2}. 

235
Achieving extreme scaleout to massively parallel architectures requires a systematic approach and work. The goal is to eliminate all bottlenecks that appear when we require to scale to millions of threads. It is crucial to say that often, for even moderate number of processors, these parts of the code may account for only a small fraction of the overall runtime. However, as Amdahl’s law predicts, they tend to dominate the overall run time at massively parallel runs. On the other hand, it is equally important to design new methods that push the frontiers of scalability for the key kernels of the code as well. We describe perfect examples of this systematic approach in this lecture. In Sec. 2 we analyze a hierarchical parallelization approach for 3D Fast Fourier Transforms that is able to push scalability almost two orders of magnitude further than traditional parallel 3D FFT approaches. Sec. 3 illustrates how it is possible to completely rethink and re-engineer well known but not high performance, algorithms for wavefunction orthogonalization and render them into massively scalable and high performance kernels. In particular, we describe block Gram-Schmidt schemes that BLAS3 based and are particularly suited for the interconnects of modern supercomputers. Finally, in Sec. 4 we show how we can render fully scalable initialization from atomic wavefunctions. This is an example of non-scaling original kernel, to which little (if any) attention was paid to, which however becomes the bottleneck for large scale simulations with thousands of atoms.

A word about the computational platforms. Clearly, large scale simulations require massively parallel machines. We focused on the IBM BlueGene supercomputer series, that with its fast and multimodal interconnects and the overall balanced (and thus scalable) design allows for very fast large scale simulations to become possible. However, we stress that the lessons learned are not limited to this architecture only, but we claim them to be widely applicable. The main reason stems from the strong trends in modern supercomputers for localized networks of higher dimensions (i.e. toruses), multicore nodes and limited memory/network bandwidth per core.

2 Task Groups Strategy for 3D Parallel FFTs

A significant number of popular and highly successful electronic structure codes use a plane wave basis to discretize the Schrödinger equation and thus rely on heavy use of 3D Fast Fourier transforms. Thus, it is natural to spent a lot of efforts in developing highly efficient parallelization strategies for 3D FFTs. We describe here a scheme that exploits opportunities for hierarchical parallelism. In particular, the scheme is based on a Task Groups parallelization strategy that concurrently performs several parallel 3D FFTs, one per each group of processors. The approach was first implemented in CPMD\(^5\) and was later incorporated in Quantum-Espresso (starting from the CPV subtree\(^5\)) (several recent similar implementations in other codes exist as well).

In plane wave codes, wavefunctions \(\Psi_p = [\psi_1, \ldots, \psi_{\text{occ}}]\) are expanded in Fourier space, where \(\text{occ}\) is the total number of valence electrons, which in turn depends on the type and number of the involved atoms. In medium size simulations \(\text{occ}\) is in the order of several hundreds, while large simulations will push \(\text{occ}\) to thousands or even tens of

\(^5\)http://www.cpmd.org
Figure 1. Structure of the standard parallel 3D FFT.

thousands. The charge density $\rho(r)$ at position $r$ in real space is given as

$$\rho(r) = \sum_{i=1}^{occ} |\psi_i(r)|^2. \quad (1)$$

Observe that since the wavefunctions are expanded in Fourier space, computation of charge density in Fourier space would entail doubly nested summations. Instead, it is performed in real space. To this end, the wavefunctions are transformed back to real space by means of inverse 3D FFTs. In the case of $P$ available processors, all of them can be devoted to a single parallel 3D FFT. On the other hand, we can do $G, (G \ll occ)$ parallel 3D FFTs concurrently. We define $G$ groups of processors, each of which works on a single parallel 3D FFT. Thus, the number of loops in computing the charge density is $\lceil occ/G \rceil$ (special handling of the last loop takes care of the case that $occ$ is not divided exactly by $G$).

Performing one parallel 3D FFT at a time, thus using all available processors, limits scalability. Here is why: The Fourier coefficients of the wavefunctions are organized in a $x - y - z$ 3D mesh in Fourier space. For all wavefunctions, each processor is assigned a number of pencils across the $z$ (vertical) direction. Fig. 1 (left cube) illustrates the case for two processors and one wavefunction. The 3D inverse FFT is performed as follows:

1. 1D inverse FFTs across the $z$ (vertical) direction are computed independently.

2. An all-to-all global communication distributes the results to all processors, so that each processor ends up with a number of complete $x - y$ planes (see right cube of Fig. 1).
3. Then, 2D inverse FFTs are performed independently by each processor without the need for further communication.

It is clear that if the number $P$ of available processors is larger than the number of $x-y$ planes, which is the mesh dimension across the $z$ direction, some processors will get no planes at all. In general, the scalability of this scheme is limited by the largest dimension of the FFT mesh. For parallel architectures with a moderate number of available processors this limitation is not severe as practical runs of ab initio codes use hundreds of $x-y$ planes. However, on massively parallel architectures we need to utilize thousands of processors and thus we need a different parallel 3D FFT scheme. Our solution is to exploit opportunities for hierarchical parallelism.

Observe that in order to calculate the charge density $\rho(r)$ by means of (1) we need to iterate through a loop of 3D FFTs, equal to the number $occ$ of valence electrons. The Task Groups (TG) strategy will assign different groups of processors to different wavefunctions. Suppose that a processor $p_e$ is empty, in the sense that no $x-y$ planes would be assigned to it if all $P$ processors were to participate in an inverse 3D FFT. Then, since the number of its peers in the group will be $P/G$ we can choose the number of groups $G$ so that a processor will be never empty. We outline the TG scheme in Tab. 1.

The concurrent implementation of $G$ 3D FFTs is organized on a 2D mesh of processors. Each processor belongs to its row group as well as to its column group. Global communications are restrained within these groups. Iteration $k$ performs the 3D FFTs needed for wavefunctions $(k-1)G+i, i=1, \ldots, G$. Remember that each processor holds only part of the Fourier coefficients for each wavefunction. Thus, the all-to-all within the row group (line 2) brings to each column group all the Fourier coefficients for the wavefunction assigned to it. For example, at iteration $k$ the $j$th processor of the first column group will send its parts of the $(k-1)G+i, i=2, \ldots, G$ wavefunctions, to its row group peers $i=2, \ldots, G$, respectively, while it will receive from them all needed parts for the $(k-1)G+1$ wavefunction. Then, all processors in each column group can perform a parallel 3D FFT (line 3). Finally, the charge density $\rho$ can be accumulated by means of a global reduction across processors in each row group (line 4).

The Task Groups scheme requires additional memory. Remember that each processor holds a part of the wavefunctions coefficients for all eigenvalues. Thus, in order for a column group to work exclusively on a single eigenvalue each processor needs to receive additional wavefunction coefficients from its row group peers. The amount of the extra memory depends upon the number $G$ of Task Groups. There is a tradeoff between the number of available processors $P$ and the number of Task Groups. In order to exploit a large number of available processors we need many Task Groups. On the other hand, this will increase the amount of additional local memory as well as the traffic on the interconnect for the initial all-to-all. However, the 3D FFTs within each column group will also require less communication, since only $P/G$ processors are involved in each column group.

Similar to the calculation of charge density, forces contribution to the orthogonality constraints for the wavefunctions requires a loop of forward 3D FFTs across the occupied states. A parallel 3D FFT is implemented following the same steps as in the inverse transformation in exactly the opposite order: i) Each processor holds a number of complete $x-y$ planes on which it performs 2D FFTs, ii) a global all-to-all assigns to each processor a number of $z$ sticks on which independent 1D FFTs are performed. The
Define a 2D processor array
The number of columns is equal to the number $G$ of Task Groups
The number of rows is equal to the number of processors in each Task Group

1. \textbf{DO} $k = 1$, \textit{occ}/$G$
   2. \textit{all-to-all} communication in row group: brings all needed Fourier coefficients for 3D FFT
   3. parallel 3D FFT within column group
   4. \texttt{allreduce} to accumulate charge density within row group
   5. \textbf{ENDDO}

Table 1. The TG parallel 3D FFT scheme for the calculation of charge density $\rho(r)$.

Task Groups strategy is analogously adopted, so that each column group works on different wavefunctions.

We note that very good scaling for parallel 3D FFT\texttext{n}s has been achieved by means of the Volumetric FFT algorithm\texttext{\textsuperscript{14}}, which employs distribution of the FFT mesh across all three $x - y - z$ directions. However, employing this scheme in our testbed codes would require a major redesigning of the data organization and the corresponding data structures. The Task Groups scheme allows for very good scalability while requiring only minimal changes to the underlying electronic structures code.

Customizing for machines with localized interconnects (i.e. toruses) The decisive parameters in order to select the optimal $G$ involve i) the amount of memory available to each processor core ii) the latency and bandwidth of the dedicated collective communication tree interconnect. For example, on the BlueGene /L machine (which was the first to test the Task Groups strategy), latency of tree traversal was 2.5 $\mu$s with 2.5 GB/s bandwidth per link, thus leading to a 23TB/s total binary tree bandwidth (64k machine). It is typical in our practical applications to use 8-32 Task Groups.

2.1 Scalability Experiments

We experimented with a molecular system comprised of 80 water molecules, that represents a problem of intermediate size (240 atoms). The size of the FFT mesh used was 128$^3$. The number of occupied electrons of the system is $\text{occ} = 320$.

The left plot of Fig. 2 illustrates scalability results (total run time) for the calculation of the charge density and the forces contribution to the orthogonality constraints with and without the TG strategy. There are 128 $x - y$ planes across the $z$ direction. Thus, the standard parallel 3D FFT implementation scales only up to 128 computing nodes. On the other hand, the TG implementation continues to scale, where we have used 2 Task Groups in the case of 256 computing nodes and 4 Task Groups in the case of 512 computing nodes.

The right plot of Fig. 2 illustrates the percentage, in terms of run time, of the FFT related computation (with TG) compared with the percentage of the orthogonalization related computations. We stress that the latter is dominated by the diagonalization of a dense
matrix of \( \text{occ} \times \text{occ} \) at each Molecular dynamics step. These constitute the main computational kernels of the application. What remains involves input-output operations and other tasks whose relative load reduces drastically as the size of the simulation increases. It is evident that the improved scaling of the 3D \text{FFTs} causes the diagonalization to become
dominant in terms of cost: 40% for the orthogonalization while 25% for the 3D FFTs in the case of 512 computing nodes. It is important to note that diagonalization is based on a BLAS 3 implementation that uses a high performance DGEMM library available for the compute nodes.

3 Large Scale Wavefunction Orthogonalization

In large scale electronic structure calculations, that involve thousands of valence electrons, keeping an orthogonal set of wavefunctions starts to dominate the overall cost. In this section we describe recent developments in high performance orthogonalization methods, that allow extreme scalability.

3.1 Orthogonalization by Means of the Cholesky Factorization

Consider the matrix $Q = [q_1, q_2, \ldots, q_k] \in \mathbb{R}^{n \times k}$ the columns $q_i$ of which we wish to orthonormalize. Thus, consider matrix $X \in \mathbb{R}^{k \times k}$ such that:

$$Y = QX,$$

and

$$Y^\top Y = I_k,$$

where $I_k$ is the identity matrix of dimension $k \times k$. From now on we will omit the subscript $k$ and simply use $I$ when the dimensions are clear from the context. Observe that the columns of matrix $Y$ will span the same linear subspace as the columns of $X$, since the latter are linear combinations of the columns of matrix $X$. Substituting Eq. 2 into Eq. 3 leads to

$$X^\top Q^\top QX = I.$$

Thus, if we set $S = Q^\top Q$ to be the “overlap” matrix it is straightforward to see that

$$S = X^{-\top}X^{-1}.$$

Since the overlap matrix $S$ is symmetric positive definite (SPD) we can choose matrix $X$ to be the inverse of the Cholesky factor of $S$. In other words, let the upper triangular matrix $R \in \mathbb{R}^{k \times k}$ be the Cholesky factor of matrix $S$

$$S = R^\top R \quad \text{and set}$$

$$X = R^{-1}. \hspace{1cm} (5)$$

Then, we have

$$X^{-\top}X^{-1} = (R^{-1})^{-\top}(R^{-1})^{-1} = R^\top R = S. \hspace{1cm} (7)$$

The overall cost of the scheme is straightforward to analyze. The computation of the overlap matrix $S$ induces a cost of $O(nk^2)$, since symmetry allows us to calculate only the upper (or lower) triangular part and every entry requires a dot product calculation that costs $O(2n)$. The Cholesky factorization of matrix $S$ induces a cost $O(k^3/3)$. In order to calculate the final orthonormal matrix $Y$ we need to perform the computation $Y = XR^{-1}$. Inverting matrix $R$ will induce a cubic cost $O(k^3)$ and the final matrix-matrix multiplication $XR^{-1}$ will require $O(2nk^2)$. The total computational cost sums to $O(3nk^2 + k^3)$.
3.1.1 Practical Implementation

In the practical situation of Density Functional Theory (DFT) electronic structure calculations, the columns of matrix $Q = [q_1, q_2, \ldots, q_k]$ hold the coefficients of the expansion of the $k$ occupied wavefunctions on a suitable basis. In typical parallel implementations of DFT codes, such as CPMD which is our target code platform, matrix $Q$ is distributed row-wise to the available processors. We denote this by writing

$$Q = \begin{bmatrix} Q_0 \\ Q_1 \\ \vdots \\ Q_{P-1} \end{bmatrix},$$

where $P$ is the number of the available processors and $Q_i$ are $\lceil n/P \rceil \times k$ blocks. That is, every processor holds a number of consecutive rows of matrix $Q$. Thus, the computation of the overlap matrix $S = Q^T Q$ is accomplished as

$$S = \sum_{i=0}^{P-1} Q_i^T Q_i. \quad (9)$$

Observe that the matrix-matrix $Q_i^T Q_i$ is local to each processor, and that the summation of the overlap matrix $S$ requires a global reduction operation ($\text{MPI}_\text{ALLREDUCE}$). Furthermore, the local matrix-matrix multiplications (in particular Rank-k updates) are of BLAS 3 type (routine $\text{xSYRK}$), which insures close to peak processor performance.

In the case that the number of occupied wavefunctions $k$ is small, then the Cholesky factorization $S = R^T R$ of the overlap matrix $S$ can be performed on each processor independently. In the sequel, the calculation of the orthonormalized wavefunctions $Y$ by means of solving the linear system $R^T Y = Q^T$ can also be performed completely independently,

$$R^T [Y_1^T, Y_2^T, \ldots, Y_{P-1}^T] = [Q_1^T, Q_2^T, \ldots, Q_{P-1}^T], \quad (10)$$

where each processor solves its local linear system $R^T Y^T_i = Q_i^T$. The result is the orthonormal wavefunction matrix $Y$ which is distributed row-wise to the available processors.

However, in the case of large systems that involve thousands of electrons, the number of occupied states $k$ becomes so large that the Cholesky factorization $S = R^T R$ has to be done in parallel. In fact, the overlap matrix $S = Q^T Q$ cannot be replicated to all processors, but rather matrix $S$ has to be distributed as well. This is especially crucial in massively parallel platforms, that are equipped with tens of thousands of processing elements, each of which has limited physical memory available. For example, for a system with $k = 10000$, the overlap matrix $S$ alone will easily consume at least 800 MBytes of main memory. Furthermore, the computational cost of independently calculating the Cholesky factorization will rise to a staggering 0.33 TFlop$^b$ at each processor.

The choice of distribution of the overlap matrix $S$ to the available processors will be affected by a number of software design and parallel platform parameters.

---

$^b$TFlop = $10^{12}$ floating point operations.
Massively parallel deployment. In the present work we are interested in massively parallel computational platforms, such as the IBM Blue Gene supercomputer series\(^5\). In this case, the number of available processors \(P\) is directly comparable to the number \(k\) of occupied states or even significantly larger. Let us proceed with an example from planewave codes such as our target software platform CPMD. The number of planewaves \(n\) will typically reach several tens of millions for large systems, while the number of occupied states \(k\) will be in the order of some thousands. Thus, while the planewave matrices \(Q\) and \(Y\) can be efficiently distributed to tens of thousands of processors, the same is not meaningful for the overlap matrix \(S\), since we will end up with a distribution where each processor will hold a rather small number of elements of the matrix, or none whatsoever. It is well known that dense linear algebra kernels, such as the Cholesky factorization, are practically impossible to efficiently scale to thousands of processors without the involved matrices becoming adequately large. Thus, one is led to choose a distribution scheme where only a subset of processors will actually hold a part of the overlap matrix \(S\). The parallel Cholesky factorization will have to be computed on this subset of processors.

Software design. In implementing a parallel Cholesky factorization one may utilize several different approaches. In the popular SCALAPACK library\(^1^5\), a 2D cyclic block distribution is used. In particular, the active subset of processors is organized into a two dimensional grid and the overlap matrix \(S\) is mapped on this grid. This means that each processor in the grid will take a number of consecutive rows and columns. The block size has to be carefully selected so as to maximize both processor performance as well as maximum utilization of the interconnection network of the parallel machine. In all our implementation decisions we have to keep in mind the existing data distribution on all processors. Remember that the wavefunctions (columns of matrix \(Q\)) are distributed row-wise. This directly implies that a block column distribution of the overlap matrix \(S\) is natural and can be easily implemented. Indeed, the summation (9) can be implemented by blocks. Consider the block partitions

\[
S = [S_1, S_2, \ldots, S_p]
\]

(11)

\[
Q_i = [Q_{i,1}, Q_{i,2}, \ldots, Q_{i,p}],
\]

(12)

where each of the blocks \(S_j, Q_{i,j}\) has \(k/p\) columns and \(p < P\) is the number of active processors in the subset that will take part in the Cholesky factorization. Then, each of the blocks \(S_j\) is calculated as

\[
S_j = \sum_{i=0}^{p-1} Q_{i,j}^T Q_{i,j}, \quad j = 1, \ldots, p
\]

and stored only by the \(j\)-th processor of the active set of \(p\) processors. Typically the number of messages required in parallel Cholesky implementations grows as \(O(p^2)\), which is one of the main problems in achieving high scalability. Once the Cholesky factorization is computed we then need to solve the linear system \(R^T Y = X\) and since the Cholesky factor \(R\) is distributed as well, this will require additional communication.

We have shown that the simple and effective scheme of orthonormalization by means of the Cholesky factorization needs special attention when we consider the study of very

\(^5\)http://www.research.ibm.com/bluegene
large systems, with thousands of occupied states, on massively parallel computing platforms. In the sections that follow we describe a method that is based on the well known Gram-Schmidt orthonormalization method that aims to combine the appealing BLAS 3 characteristics of the Cholesky approach with the simplicity and scalability of the Gram-Schmidt algorithm.

### 3.2 The Scalar Gram-Schmidt Algorithm

The Gram-Schmidt method is one of the oldest and most popular methods to orthonormalize a set of vectors. The principle behind the method is simply explained in geometrical terms. Consider the vectors $q_1, q_2$ in $\mathbb{R}^2$ (standard Euclidean 2 dimensional space, see Fig. 3). Let us assume that $q_1$ is normalized such that $\|q_1\|_2 = 1$. Then, we calculate the projection of vector $q_2$ on vector $q_1$. It is clear that the vector $(q_2^T q_1) q_1$ is the coefficient of vector $q_2$ towards the direction (or span) of vector $q_1$. Thus, if we subtract this contribution from vector $q_2$ we will get a new vector that is clear of any directions towards the direction of vector $q_1$, i.e. it will be orthogonal to it. The argument readily generalizes for more vectors in higher dimensions. It is also not difficult to see that the new set of vectors spans the same linear subspace, as these new vectors are just linear combinations of the original vectors, with the first one ($q_1$) being the same. The upper part of Tab. 2 illustrates an algorithmic description of the standard Gram-Schmidt algorithm. Note that the orthonormalization is performed in place, meaning that no additional memory space is required to store the resulting orthonormal vectors.
The Scalar Gram-Schmidt Algorithm

(* Input *) Matrix $Q = [q_1, q_2, \ldots, q_k] \in \mathbb{R}^{n \times k}$ the columns $q_j$ of which we would like to orthonormalize

(* Output *) Matrix $Q \in \mathbb{R}^{n \times k}$ such that $Q^T Q = I$

**Standard Gram-Schmidt**

1. Set $r_{11} = \|q_1\|_2$, $q_1 = \frac{q_1}{r_{11}}$
2. for $j=2, \ldots, k$
3. Calculate $r_{jj} = \langle q_j, q_i \rangle$ for $i = 1, 2, \ldots, j - 1$
4. $\hat{q}_j = q_j - \sum_{i=1}^{j-1} r_{ij} q_i$
5. $r_{jj} = \|\hat{q}_j\|_2$
6. if $r_{jj} == 0$ then stop, else $q_j = \hat{q}_j / r_{jj}$
7. end

**Modified Gram-Schmidt**

1. Set $r_{11} = \|q_1\|_2$, $q_1 = \frac{q_1}{r_{11}}$
2. for $j=2, \ldots, k$
3. Set $\hat{q} = q_j$
4. for $i=1 \ldots j-1$
5. $r_{ij} = \langle \hat{q}, q_i \rangle$
6. $\hat{q} = \hat{q} - r_{ij} q_i$
7. end
8. $r_{jj} = \|\hat{q}\|_2$
9. if $r_{jj} == 0$ then stop, else $q_j = \hat{q} / r_{jj}$
10. end

Table 2. The Gram-Schmidt orthogonalization algorithm. Top: the standard version. Bottom: the modified version. With $\langle, \rangle$ we denote the vector scalar product.

*The Modified Gram-Schmidt algorithm* Note that in the standard Gram-Schmidt algorithm, the order of calculations in lines 3-4 (Tab. 2) can be interchanged. In particular, line 3 calculates the projection coefficients of the current vector to be orthogonalized against all previous (already orthogonal vectors) and line 4 performs the subtractions of these projections. In the modified Gram-Schmidt algorithm (bottom of Tab. 2) the order of operations just described is changed. In particular, when one projection coefficient has been calculated (i.e. $r_{ij} = \langle \hat{q}, q_i \rangle$), then the corresponding projected vector is immediately subtracted from the current approximation $\hat{q} = \hat{q} - r_{ij} q_i$. It is not difficult to see that this is mathematically completely equivalent to what is done in standard Gram-Schmidt. However, it is well known than in the environment of floating point calculations modified Gram-Schmidt can yield better numerical accuracy, especially in the situation that two vectors are almost parallel with each other (see for example Ref. 16).

Observe that the computational cost of both variants of Gram-Schmidt is in the order of $O(2nk^2)$ since,
The cost of computing the projection coefficients is

\[ \sum_{j=2}^{k} 2n(j - 1) = O(nk^2) \]  \hspace{1cm} (13)

The cost of subtracting the projections is

\[ \sum_{j=2}^{k} (n(j - 1) + n) = O(nk^2) \]  \hspace{1cm} (14)

A careful look in the algorithmic description of Gram-Schmidt (see Tab. 2) shows that if we denote by \( R = \{r_{i,j}\} \) the matrix projection coefficients, then matrix \( R \) is upper triangular. Furthermore, the matrix product \( QR \), where \( Q \) is the orthonormalized set of the original vectors, yields the original set of non-orthonormal columns. This is the well known QR factorization and Gram-Schmidt is one of the several possible ways to obtain it (see Ref. 16).

### 3.2.1 Practical Implementation Issues

In standard Gram-Schmidt the calculation of the projection coefficients \( r_{ij} \) (line 3 of Standard Gram-Schmidt in Tab. 2) can be organized as a BLAS 2 matrix-vector product. Indeed, we can write

\[
\text{Calculate } r_{ij} = \langle q_j, q_i \rangle, \ i = 1, \ldots, j - 1
\]
equivalently as

\[
r_{1:j-1,j} = Q_{1:j-1}^T q_j,
\]  \hspace{1cm} (15)

where we have adopted MATLAB notation in which, \( : \) denotes either a complete row or column (depending on its position at the subscript). On the other hand, observe that for modified Gram-Schmidt we are not able to organize calculations as matrix-vector products, since the projection coefficient \( r_{i,j} \) depends on the value of the immediately previous projection coefficient \( r_{i-1,j} \). As we will see in the sequel, this property has more perplexed implications when we consider the parallel implementation of the Gram-Schmidt algorithm.

### 3.2.2 Parallel Implementation and Scalability

Remember that the wavefunction matrix \( Q = [q_1, q_2, \ldots, q_k] \) is distributed row-wise to the \( P \) available processors. This has an immediate implication on the parallelization scheme of choice. In particular, for standard Gram-Schmidt observe that the matrix-vector product for the calculation of the projection coefficients \( r_{i,j} \) can be performed by means of a global reduction. In particular, let \( Q_{i,j}, i = 0, \ldots, P-1 \) be the part of the \( j \)-th column of matrix \( Q \) that resides on processor \( i \). Then, the calculation of the projection coefficients according to (8) is

\[
r_{1:j-1,j} = \sum_{i=0}^{P-1} Q_{i}^T Q_{i,j}.
\]  \hspace{1cm} (16)
Table 3. Computation/communication characteristics of standard, modified and block Gram-Schmidt.

<table>
<thead>
<tr>
<th>Method</th>
<th>Comp. per Proc.</th>
<th>Messages</th>
<th>max. size of message</th>
</tr>
</thead>
<tbody>
<tr>
<td>St. G-S</td>
<td>(O(2nk^2/P))</td>
<td>(O(2k))</td>
<td>(k - 1)</td>
</tr>
<tr>
<td>Mod. G-S</td>
<td>(O(2nk^2/P))</td>
<td>(O(k^2))</td>
<td>1</td>
</tr>
<tr>
<td>Bl. St. G-S</td>
<td>(O(2nk^2/P))</td>
<td>(O(2k/b))</td>
<td>(O(b(k - b)))</td>
</tr>
<tr>
<td>Bl. Mod. G-S</td>
<td>(O(2nk^2/P))</td>
<td>(O((k/b)^2))</td>
<td>(O(b^2))</td>
</tr>
</tbody>
</table>

Observe that the matrix vector product \(Q_i^T Q_{i,j}\) is local to each processor. Thus a global reduction (i.e. MPI\_ALLREDUCE) will ensure that all processors have the final projection coefficients. Then, the subtraction of the projection vectors (line 4 of standard Gram-Schmidt, Tab. 2) can proceed without any communication. The remaining required communication is needed for the calculation of the normalization factor \(r_{j,j}\) (line 5), for which a global reduction can be utilized again. Thus, the total number of global reductions required to orthonormalize the \(k\) columns of matrix \(Q\) is \(2k - 1\) and the largest size of the message required will be \(k - 1\) floating point numbers, which corresponds to the vector \(r_k\) of projection coefficients at the last step of the algorithm. Line 2 of Tab. 3 summarizes the computation and communication requirements of parallel standard Gram-Schmidt.

The case of modified Gram-Schmidt is quite different in terms of the communication pattern. The projection coefficient \(r_{i,j}\) requires again a global summation since vectors \(\hat{q}, q\) are distributed row-wise. However, since the current vector \(\hat{q}\) is updated at each step of the inner loop (lines 4-7, bottom of Tab. 2), the total number of global reductions is quadratic \(O(k^2)\) in terms of the number of vectors to orthonormalize. On the other hand, the size of the messages is minimal, namely 1 floating point number. The third line in Tab. 3 summarizes the computation/communication characteristics of parallel modified Gram-Schmidt.

In contrast to the parallel implementation of the Cholesky based orthogonalization, both variants of parallel Gram-Schmidt, are able to utilize all of the available processors at all stages of the algorithms. On the other hand, in contrast to Gram-Schmidt, as we saw in Sec. 3.1, the Cholesky approach in orthogonalization, can be implemented entirely in highly optimized BLAS 3 matrix operations. This has a profound effect in overall processor performance in favor of the Cholesky approach against the Gram-Schmidt algorithm (see Sec. 3.6).

### 3.3 Block Gram-Schmidt

A natural question arises whether a BLAS 3 variant of Gram-Schmidt is possible. Consider again the matrix \(Q = [q_1, q_2, \ldots, q_k]\) that we wish to orthonormalize. Now, let us partition this matrix column-wise defining a set of block submatrices \(B_l \in \mathbb{R}^{n \times b}\), where \(l = 1, \ldots, k/b\) and \(b\) is the block size, such that

\[
Q = [B_1, B_2, \ldots, B_{k/b}].
\]  

For simplicity of the discussion we have asserted that the block size \(b\) exactly divides the number \(k\) of columns of matrix \(Q\). Let us temporarily assume that the columns of the first block \(B_1\) are mutually orthonormal, i.e. \(B_1^T B_1 = I_b\). Then, we seek to orthogonalize all vectors of the next block \(B_2\) against the columns of the block \(B_1\).
The Block Gram-Schmidt Algorithm

(* Input *) Matrix \( Q = [B_1, B_2, \ldots, B_{k/b}] \in \mathbb{R}^{n \times k} \) that we would like to orthonormalize, block size \( b \)

(* Output *) Matrix \( Q \in \mathbb{R}^{n \times k} \) such that \( Q^T Q = I \)

Standard Gram-Schmidt
1. Compute Cholesky factorization \( B_1^T B_1 = R^T R \)
   if \( R \) is singular then \( \text{stop} \), else orthonormalize \( B_1 \equiv B_1 R^{-1} \)
2. \( \text{for} \ j=2, \ldots, k/b \)
3. Calculate \( R_{i,j} = B_1^T B_j \) for \( i = 1, 2, \ldots, j-1 \)
4. \( B_j = B_j - \sum_{i=1}^{j-1} B_i R_{i,j} \)
5. Compute Cholesky factorization \( \hat{B}_j^T \hat{B}_j = R^T R \)
   if \( R \) is singular then \( \text{stop} \), else orthonormalize \( B_j \equiv \hat{B}_j R^{-1} \)
6. \( \text{end} \)

Modified Gram-Schmidt
1. Compute Cholesky factorization \( B_1^T B_1 = R^T R \)
   if \( R \) is singular then \( \text{stop} \), else orthonormalize \( B_1 \equiv B_1 R^{-1} \)
2. \( \text{for} \ j=2, \ldots, k/b \)
3. Set \( \hat{B} = B_j \)
4. \( \text{for} \ i=1 \ldots j-1 \)
5. \( R_{i,j} = B_i^T \hat{B} \)
6. \( \hat{B} = \hat{B} - B_j R_{i,j} \)
7. \( \text{end} \)
8. Compute Cholesky factorization \( \hat{B}^T \hat{B} = R^T R \)
   if \( R \) is singular then \( \text{stop} \), else orthonormalize \( B_j \equiv \hat{B} R^{-1} \)
9. \( \text{end} \)


Consider the following block generalization of the scalar Gram-Schmidt projection process

\[
\hat{B}_2 = B_2 - B_1 (B_1^T B_2),
\]

(18)

where the projection matrix \( B_1^T B_2 \) is of dimension \( b \times b \). Then, it is not difficult to see that
the columns of matrix \( \hat{B}_2 \) are all orthogonal to the columns of the first block \( B_1 \). Indeed,
\[
B_1^\top \hat{B}_2 = B_1^\top (B_2 - B_1(B_1^\top B_2)) = B_1^\top B_2 - (B_1^\top B_1)(B_1^\top B_2) = B_1^\top B_2 - B_1^\top B_2 = 0
\]
where we have exploited the assumption that the first block \( B_1 \) is orthonormal.

In order to orthonormalize this first block we can either use the scalar Gram-Schmidt algorithm, or we can exploit the Cholesky approach as we saw in Sec. 3.1. The same applies for matrix \( \hat{B}_2 \) that will yield a new orthonormal block \( B_2 \). Note that after the orthonormalization of block \( B_2 \) against itself, the block remains orthonormal to the previous block \( B_1 \), since the new block \( B_2 \) spans the same subspace \( \hat{B}_2 \) which is by construction orthogonal to the subspace \( B_1 \) spanned by the columns of block \( B_1 \). The process is repeated until the last block \( B_{k/b} \) is orthonormalized against all previous blocks \( B_1, B_2, \ldots \).

In order to obtain a fully BLAS 3 Gram-Schmidt variant we opt to use the Cholesky approach in orthogonalizing the blocks \( B_l, l = 1, \ldots, k/b \). Tab. 4 contains an algorithmic description of the new block Gram-Schmidt algorithm. The last two rows of Tab. 3 summarize the computation and communication characteristics of block Gram-Schmidt.

The cost of the block Gram-Schmidt scheme is summarized as follows.

- The cost of calculating the projection coefficients and subtracting the projections (lines 3-4, upper part of Tab. 4) runs in the order \( O(2nk^2 - 2nk) \).

- The cost of orthogonalizing the latest block using the Cholesky approach is \( O(3nk - \frac{nk}{b^2}) \).

Thus the total cost is \( O(2nk^2 + nk) \). It is interesting to note that this cost approaches the cost of the scalar Gram-Schmidt algorithms for small and medium block sizes \( b \), while it converges to the cost of the Cholesky approach at the extreme case that \( b = k \). A similar cost analysis holds for the modified block Gram-Schmidt algorithm.

3.3.1 Practical Implementation Issues

It is clear that we designed both block Gram-Schmidt variants in order to allow their efficient implementation using BLAS 3 matrix-matrix operations. In particular, following the practice for the scalar standard Gram-Schmidt, the computation of the projection matrices \( R_{i,j} \) (see line 3, upper part of Tab. 4), is more efficiently implemented by grouping them in a coarser matrix-matrix multiplication

\[
R = B_{i,j-1}^\top B_j.
\]

In this case the resulting projection matrix \( R \) is of size \((j-1)b \times b\), attaining a maximum size of \((k-b) \times b\) when \( j = k/b \). Thus, the significant factors in selecting a proper blocksizes are the following two

- BLAS 3 performance. The complexity of the matrix-matrix multiplication for the calculation of the projection matrix \( R \) (see 20) ranges from the minimum \( O(nb^2) \) to
the maximum $O(nkb)$. The respective volume of data traffic to main memory would be $O(nb)$ and $O(kn)$, which leads to a constant $O(1/b)$ ratio of memory traffic over computation. Thus, it is clear that as the block size $b$ increases we are getting a better ratio which translates into increased processor performance. Observe that the same analysis holds in the parallel implementation of block standard Gram-Schmidt, since as we will see both the totality of computations and data traffic/storage is equally distributed among all available processors.

- **Message size.** We analyzed that increasing the block size $b$ benefits processor performance. On the other hand, in a parallel implementation the calculation of the projection matrix $R$ will require a global summation of $O(kb)$ floating point numbers in the worst case. Thus, increasing the block size $b$ too much can potentially over-stress the interconnection network of the machine and thus potentially cause a significant loss of the maximum communication performance. Notice however, that the dependence on the block size $b$ is linear. A well designed balance is required.

For the case of block modified Gram-Schmidt the same line of analysis holds. The main difference is that the projection blocks $R_{i,j}$ (see line 5, bottom of Tab. 4) computed and communicated are of size $b \times b$. Although the dependence of communication load depends quadratically on the block size $b$, this load is still smaller than the respective on the block standard Gram-Schmidt ($O(kb)$ and typically $k \gg b$). On the other hand, as we will see in the following section, introducing a block in modified Gram-Schmidt is beneficial in terms of number of required messages.

### 3.4 Parallel Implementation

The parallelization strategy follows the same design lines of the scalar case (see Sec. 3.2.1). Tab. 5 provides an algorithmic description for the parallel block standard Gram-Schmidt algorithm. In particular, the columns of matrix $Q$ are distributed row-wise to the available processors. Thus, the computation of the projection matrices local projection matrices $R_{i,j}$ (line 5) can be performed local to each processor, using a high performance $\text{xGEMM}$ matrix-matrix multiplication routine, followed by a global reduction ($\text{MPI\_ALLREDUCE}$) to accumulate the projection coefficients to all processors (line 6). Note that we use a single reduction operation (one call to $\text{MPI\_ALLREDUCE}$) in order to minimize the number of messages and thus the latency. Then, the subtraction of the projected vectors (line 7, Tab. 5) is performed completely locally, requiring no communication whatsoever. We perform this by using a single call to highly optimized matrix-matrix multiplication routine ($\text{xGEMM}$) in order to maximize performance.

Concerning the number of messages (global reductions in this case), it is not difficult to see that since the outer loop does not have $k$ but rather $k/b$ iterations, the total number of messages is reduced accordingly. Tab. 3 depicts the computation/communication profile of all proposed variants.

For the orthonormalization of the current block $B_j$ we opt to employ the $\text{BLAS 3}$ scheme that is based on the Cholesky factorization (see Sec. 3.1). First the overlap matrix $S_j = B^T \hat{B}$ is computed in parallel (lines 9-10) Observe that in contrast with the original Cholesky based scheme, the size of the overlap matrix $S_j$ is $b \times b$ and does not grow with the number $k$ of the columns to be orthogonalized. Thus, since $b$ will be typically much
The Parallel Block Gram-Schmidt Algorithm

(* Input *) Matrix \( Q = [B_1, B_2, \ldots, B_{k/b}] \in \mathbb{R}^{n/P \times k} \) that we would like to orthonormalize (distributed to \( P \) processors), block size \( b \)

(* Output *) Matrix \( Q \in \mathbb{R}^{n/P \times k} \) such that \( Q^T Q = I \) (distributed to \( P \) processors)

1. Compute local overlap matrix \( S_l = B_1^T B_1 \)
2. Compute global overlap matrix \( S_1 \) by means of \( MPI_{\text{ALLREDUCE}} \) on local matrices \( S_l \)
3. Compute Cholesky factorization \( S_1 = R^T R \) (local comp.)
   \( \text{if} \ R \text{ is singular then stop, else orthonormalize} \)
   \( B_1 \equiv B_1 R^{-1} \) (local comp.)
4. \( \text{for} \ j=2, \ldots, k/b \)
5. Calculate local projections \( R_{i,j}^l = B_i^T B_j \) for \( i = 1, 2, \ldots, j - 1 \)
   as \( [B_1, \ldots, B_{j-1}]^T B_j \) by means of \( \text{xGEMM} \)
6. Calculate global projection matrix \( R^{(g)} = [R_{1,j}^l, \ldots, R_{j-1,j}^l]^T \) by means of a global reduction \( MPI_{\text{ALLREDUCE}} \) on \( [R_{1,j}^l, \ldots, R_{j-1,j}^l]^T \)
7. Compute \( \hat{B}_j = B_j - [B_1, \ldots, B_{j-1}] R_{i,j} \) by means of \( \text{xGEMM} \)
8. Compute local overlap matrix \( S_j^l = B_j^T \hat{B}_j \)
9. Compute global overlap matrix \( S_j \) be means of \( MPI_{\text{ALLREDUCE}} \) on local matrices \( S_j^l \)
10. Compute Cholesky factorization \( S_j = R^T R \) (local comp.)
11. \( \text{if} \ R \text{ is singular then stop, else orthonormalize} \)
    \( B_j \equiv B_j R^{-1} \) (local comp.)
12. \( \text{end} \)

Table 5. Parallel Block Gram-Schmidt orthogonalization algorithm.

smaller than \( k \) (i.e. a few hundreds at most) there is no need to distribute it to the processors, but rather the global overlap matrix \( S_j \) is replicated. Then, the Cholesky factorization is computed locally inducing a negligible cost \( O(b^3/3) \). Finally, the local part of the final orthonormal block \( B_j \) is again performed completely locally at each processor, without any required communication (line 11).

We note at this point that the check for the overlap matrix \( S_j \) being singular (i.e. lines 3 and 11) is essentially performed by means of the Cholesky factorization. In particular, the Cholesky factorization (for example routine \( \text{xPOTRF} \) from \( \text{LAPACK} \)) will return with an error message, since the Cholesky factor \( R \) will have a zero (or very small) entry on its main diagonal.

The parallelization of the block modified Gram-Schmidt method follows the same lines as described above, and we omit its detailed description here for the economy of the paper.

It is interesting to point that unlike the scalar versions of the Gram-Schmidt methods which can be performed almost entirely in place, the block variants require a modest ad-
ditional memory space. Indeed, observe that scalar standard Gram-Schmidt requires $O(k)$ memory to store the projection coefficients $r_{i,j}$ (see Tab. 2) while the modified version requires $O(1)$ additional memory. On the other hand, the block versions of Gram-Schmidt require $O(kb)$ additional memory space ($O(b^2)$ for the modified version) for the projection coefficients, and an additional $O(nb)$ for the solution of the linear systems $\hat{B}_j R^{-1}$ (and the $\hat{B}_j R^{-1}$ for the modified version). Observe that the space for the projection matrices can be reused to store the local Cholesky factors of the overlap matrix $\hat{B}_j^T \hat{B}_j$ (respectively $\hat{B}_j^T \hat{B}$ for the block modified Gram-Schmidt).

It is obvious that the parallelization of the new block Gram-Schmidt schemes relies heavily on the use of collective communication primitives. This is a crucially favorable property that can take great advantage of the high bandwidth, low latency TREE network that is available on the Blue Gene/P Supercomputer.

Finally, it is important to stress that, unlike the parallel Cholesky based approach, in the parallel block Gram-Schmidt algorithm we utilize all of the $P$ available processors. In fact, the the serial part of the new algorithm has complexity in the order of $1/3b^3$. Thus, its percentage over the total cost of the algorithm per processor is:

$$C_s = \frac{1}{6} \frac{P b^3}{n k^2}.$$  

Since the block size $b$ is kept constant and small ($O(100)$), this ratio is small and tends to be even smaller as the size of the problem $(n, k)$ increases, even when we utilize many thousands of processors. For example, setting $P = 10^5$, $b = 150$ and $k = 60 \times 10^6$, $k = 4 \times 10^3$ will give a ratio $C_s < 10^{-5}$. Thus, we can expect the algorithm to scale very well to massively parallel platforms.

### 3.5 Numerical Experiments

In this section we provide several numerical experiments that illustrate the performance of all pre-existing schemes as well as the new block schemes. We measured performance profiles both in serial as well as in massively parallel mode. The computational platform was a Blue Gene/P Supercomputer. Each compute node of this architecture is equipped with a quad core PPC 450 processor at 850 MHz, with 4 GBytes of main memory. The theoretical peak performance of each core reaches 3.4 GFLOPS. The largest configuration at our disposal consisted of 8 Blue Gene/P racks, with a total of 32768 compute cores\(^d\). The top performance we achieved on this system using our new block Gram-Schmidt schemes was 73 TFLOPS which corresponds to 67% of peak performance on 8 Blue Gene/P racks.

### 3.6 Comparison of Cholesky Orthogonalization with Scalar Gram-Schmidt

We start with a comparison of the scalar Gram-Schmidt schemes against the BLAS 3 Cholesky based orthogonalization method. Fig. 4 clearly illustrates the superiority of the Cholesky based scheme in terms of performance. We describe two experimental settings. The length of the vectors to be orthonormalized was set to $n = 10000$ and $20000$, while

\(^d\)WatsonShaheen system at IBM T. J. Watson Research Center.
the number of vectors to be orthonormalized covered the range $k = 400 : 100 : 1500$. The best performance achieved by the scalar Gram-Schmidt schemes was 0.4 GFLOPS while the Cholesky based scheme exceeded 2.2 GFLOPS. In Fig. 5 we analyze the run-time breakdown of the various stages of the Cholesky based method. In particular, we tested with $n = 20000$ and $n = 40000$ while $k = 500 : 100 : 4000$. The left column in Fig. 5 illustrates performance (GLOPFS) while the right column illustrates percentage of total run-time. It is evident that the Cholesky factorization part achieves the lowest performance, while it also is responsible for a small part of the overall run-time. It is exactly this behavior that significantly limits the massively parallel scalability of the Cholesky based scheme.

### 3.6.1 Serial Block Gram-Schmidt

We now compare the scalar (not parallel) performance of the new block Gram-Schmidt algorithms. In particular, we analyze in detail the performance profile of their various stages and we compare them with the original orthogonalization scheme based on the Cholesky factorization.

Since the block size $b$ is a parameter of crucial importance, we first illustrate its effect...
on the performance of the new scheme. Fig. 6 holds a performance breakup (in GLFOPS) for the various stages of standard block Gram-Schmidt, for $n = 20000$ (top 4) and $n = 40000$ (bottom 4), $k = 400, 1000, 2000$ and $4000$, with varying block sizes $b$. Our first observation is that the matrix-matrix multiplication ($\times \text{GEMM}$) dominates, as expected, the overall performance of the scheme. Furthermore, the performance profile (its saw like nature being typical in BLAS 3 performance studies), shows that a selection of block size between 110-160 brings the best performance. It is important to note that best block size $b$ does not depend upon the length $n$ of vectors or the number $k$ of vectors to be orthogonalized. Indeed, the optimal block size $b$ depends upon the underlying processor architecture as well as on the memory hierarchy characteristics, and it can safely be pre-computed before any useful computations take place.

The second important observation that we can draw out of these runs is that the best performance achieved by the block Gram-Schmidt scheme is directly comparable and even better than the performance achieved by the Cholesky based scheme. This is an important finding, that clearly indicates that the new scheme should achieve smaller run times than the Cholesky based scheme in order to orthonormalize the same set of vectors. This is so since the computational complexity of the latter is $O(3nk^2)$ while the complexity of the new scheme is $O(2nk^2)$. Indeed, Fig. 7 illustrates a direct comparison of the standard block Gram-Schmidt scheme against the Cholesky based scheme for $n = 20000$ (top-left) and $n = 40000$ (top-right), using two different block sizes $b = 120$ and $b = 160$. The bottom part illustrates a percentage of run times breakup for the same sizes $n$ and block
Figure 6. Performance breakup (in GFLOPS) for the various stages of standard block Gram-Schmidt, $k = 400, 1000, 2000, 4000$ and varying block sizes. Top 4: $n = 20000$. Bottom 4: $n = 40000$.

size $b = 120$. It is important to stress the marked difference on the size of the Cholesky part for the cases. In the block Gram-Schmidt scheme, the relevant matrix is always kept
small $b \times b$, while in the Cholesky based approach it increases with the number of vectors $k$ as $k \times k$.

### 3.6.2 Symmetric Multiprocessor Scaling on Each Node

The 4 cores of the PPC 450 microprocessor can be used in a symmetric multiprocessor mode. In fact, the Blue Gene/P supercomputer allows for three different modes of parallel execution. The SMP mode, where each compute node hosts an MPI process and each such process can spawn 4 threads of execution. The DUAL mode, where we have 2 MPI processes per node and each process can spawn 2 threads. The Virtual Node (VN) mode where each one of the 4 cores of the PPC 450 chip hosts one MPI process. The top of Fig. 8 illustrates the performance of the block Gram-Schmidt scheme, using the multithreaded version of the ESSL library for the Blue Gene/P. We give both the total performance as well as the performance of the DGEMM part. We note that the peak performance of each compute node is 13.6 GFLOPS. It is evident that the block Gram-Schmidt scheme exhibits excellent multithreaded parallel scaling on each node. On the other hand that Cholesky based approach does not achieve such a good scaling, mainly because the Cholesky decomposition is hard to parallelize. In particular, the bottom part of Fig. 8 shows the achieved speedup. Observe that while the scaling of the block Gram-Schmidt scheme is very good, the scaling of the Cholesky approach is poor.
3.6.3 Massively Parallel Runs

We now turn our attention to very large parallel runs with the new block Gram-Schmidt schemes. We have chosen to illustrate results representing a smaller \((n = 4 \times 10^6)\), an intermediate \((n = 10 \times 10^6)\) and a large \((n = 60 \times 10^6)\) example. In all cases we experimented with \(k = 2000\) and \(k = 4000\) number of vectors that we wish to orthonormalize. We utilized up to 8 Blue Gene/P racks which correspond to 111 TFLOPS of theoretical peak performance. All of our runs were performed in SMP mode, where we utilized the multithreaded version of the ESSL library for the Blue Gene/P compute nodes. Fig. 9 illustrates the run times, using logarithmic scales in both axes. Note that the horizontal axis corresponds to the total number of compute cores used. Thus 32768 cores correspond to 8 BG/P racks. We provide scaling results for both the projection phase of the block Gram-Schmidt algorithm as well as for the orthonormalization of the current block and the overall scaling.

Our first observation concerns the run-time breakdown of the various phases of the algorithm. As expected we verify in practice the minimal impact of the orthonormalization of the current block (lines 8-11, Tab. 5). It’s overall share is in general one order of mag-
Our main observation is that it is clear that the method exhibits very good scaling and performance even for the smaller case \((n = 4 \times 10^6)\), which improves with the size of the problem and becomes excellent for the largest case. Indeed, for the case of \(n = 60 \times 10^6\), \(k = 4000\) and using all 8 Blue Gene/P racks we achieved a performance of 73 TFLOPS which corresponds to 67% of peak performance.
4 Initialization from Atomic Orbitals

Consider a molecular system with \(N\) atoms and \(M\) valence electrons. Obviously, it is far easier to solve the Kohn-Sham equations for each atom separately, i.e. to define a separate Hamiltonian for each atom of the system and thus producing a set of “atomic” wavefunctions for each atom. Then, we can approximate the solution to the complete problem by superimposing these single atom wavefunctions. Indeed, electronic structure codes make use of precalculated “atomic” wavefunctions for different types of atoms. Thus, from a linear algebra point of view, atomic wavefunctions initialization consists of restricting the full system Hamiltonian operator on a large enough basis of atomic wavefunctions of dimension \(k > M\) and then solving for the \(M\) smallest eigenvectors of the restricted Hamiltonian.

Formally put, let matrix \(W_k \in \mathbb{C}^{n \times k}\) be the expansion of the atomic wave functions on the basis of \(n\) plane-waves, where each column of \(W_k\) corresponds to a single atom wavefunction. The standard methodology for initialization from atomic wavefunctions proceeds as follows:

1. Compute the restricted Hamiltonian: \(\tilde{H}_k = W_k^* H W_k\) and the overlap matrix \(O_k = W_k^* W_k\). Matrices \(\tilde{H}_k \in \mathbb{C}^{k \times k}\) and \(O_k \in \mathbb{C}^{k \times k}\) are both Hermitian.

2. Calculate the eigendecomposition of the restricted generalized Hermitian eigenproblem

\[
\tilde{H}_k x = \lambda O_k x. \tag{22}
\]

3. Approximate the \(M < k\) desired initial wavefunctions as \(U_m = W_k X_m\), where the columns of \(X_m\) hold eigenvectors that correspond to the \(m\) smallest eigenvalues of the restricted generalized eigenproblem (22).

The calculation of the restricted Hamiltonian \(\tilde{H}_k\) and of the overlap matrix \(O_k\) is performed in parallel since plane-waves are distributed among processors:

1. Each processor: Calculate the application of the Hamiltonian \(H\) to its set of plane-waves: \(H W_k\).

2. Each processor: Calculate the overlap \(W_k^* (H W_k)\).

3. All processors: Calculate matrix \(\tilde{H}_k\) using global summation of \(W^* (H W_k)\) among all processors.

In CPMD the solution of the generalized eigenproblem (22) was initially not distributed across available processors, but rather solved exclusively on a single processor. We next illustrate that this practice, although perfectly adequate for conventional simulations, is absolutely impractical for next generation target simulations that involve tens of thousands of atoms. Instead, we propose a fully parallel initialization from atomic wavefunctions that is based on the parallel Lanczos algorithm.
4.1 Parallel Initialization Using Lanczos

The dimension $k$ of the restricted Hamiltonian $\tilde{H}_k$ is immediately linked to the number of valence electrons $M$, and thus with the total number of atoms $N$. Thus, in the context of large simulations that involve several thousands of atoms, the dimension of the restricted Hamiltonian $\tilde{H}_k$, which is a dense matrix, will be in the order of tens of thousands. Clearly, i) storage requirements, in the order of $O(k^2)$ ii) as well as computational complexity of the generalized eigenproblem (22), in the order of $O(k^3)$, render the calculation intractable on a single processor. The following observations are key in the design characteristics of a fully parallel approach:

- Matrices $\tilde{H}_k, O_k$ are dense. Solution of the eigenproblem (22) will require the transformation of these matrices to simpler form. Namely, since this is a Hermitian generalized eigenproblem, it can be transformed to a simple eigenproblem, by means of a Cholesky factorization of the overlap matrix $O_k$, and then a reduction to tridiagonal form of matrix $O_k^\dagger \tilde{H}_k$ is needed, where $O_k^\dagger$ is the pseudoinverse of $O_k$, using the Cholesky factors.

- The calculation of the eigendecomposition of the resulting tridiagonal matrix will require $O(k^2)$ storage. Thus, it must be done in parallel.

- The new approach should exploit the current distribution of all involved matrices in terms of the distribution of plane-waves across processors.

In light of the above we propose to use the Lanczos algorithm for iterative partial tridiagonalization of a modified restricted Hamiltonian.

4.2 The Lanczos Algorithm

Consider a symmetric matrix $A$ and a starting vector $v_1$ such that $\|v_1\|_2 = 1$, where $\|\cdot\|_2$ denotes the standard Euclidian norm. The Lanczos algorithm computes an orthonormal basis for the Krylov subspace

$$K_l(A, v_1) = \text{span}\{v_1, Av_1, A^2v_1, \ldots, A^{l-1}v_1\}. \quad (23)$$

In particular, after $l$ steps of the Lanczos algorithm for matrix $A$ and starting vector $v_1$, the following Lanczos factorization holds:

$$AV_l = V_lT_l + \beta_{l+1}v_{l+1}v_{l+1}^\ast, \quad (24)$$

where $V_l$ is the orthonormal basis for $K_l(A, v_1)$ and $T_l$ is a symmetric tridiagonal matrix with structure

$$T_l = \begin{bmatrix}
\alpha_1 & \beta_2 \\
\beta_2 & \alpha_2 & \beta_3 \\
\vdots & \ddots & \ddots \\
\beta_{l+1} & \alpha_{l+1} & \beta_l \\
\beta_l & \alpha_l
\end{bmatrix}. \quad (25)$$

When $l$ is taken to be equal to the dimension $n$ of matrix $A$, then matrix $T_n$ will have the same eigenvalues as $A$. Thus, for $l < n$, the Lanczos algorithm can be viewed as a means for partial reduction to tridiagonal form for matrix $A$. 

260
4.3 A Preprocessing Step

A preprocessing step that will significantly simplify the whole process is orthogonalization of matrix $W_k$. Then, the overlap matrix $O_k$ reduces to the identity matrix $O_k = W_k^*W_k = I_k$ and the new restricted eigenproblem becomes a standard one: $H_kx = \lambda x$.

The cost of orthogonalizing $W_k$ is in the order of $O(nk^2)$. We have adopted the new approach used in CPMD (see previous section). For what follows we consider $W_k$ to have orthonormal columns.

4.4 The Distributed Initialization

The working matrix is the new restricted Hamiltonian $W_k^*\hat{H}_kW_k$ (see the previous section). Then, the proposed fully distributed initialization method proceeds as follows:

1. Calculate a Lanczos factorization for matrix $H_k = W_k^*\hat{H}_kW_k$ (see Fig. 6)

   $$(W_k^*\hat{H}_kW_k)V_l = V_lT_l + \beta_{l+1}V_{l+1}e_l^*, \quad (26)$$

   where $M < l \leq k$, matrix $T_l \in \mathbb{R}^{l \times l}$ is symmetric tridiagonal and matrix $V_l \in \mathbb{C}^{k \times l}$ has orthonormal columns. The eigenvalues of the restricted Hamiltonian $H_k = W_k^*\hat{H}_kW_k$ are approximated by the eigenvalues of matrix $T_l$. Factorization (26) is calculated by means of the Lanczos algorithm (see Fig. 6), which requires only a matrix-vector product with matrix $(W_k^*\hat{H}_kW_k)$. It is clear that this matrix need not be formed, but rather the product $(W_k^*\hat{H}_kW_k)y$ with a vector $y$ can be calculated as a series of matrix-vector products (with matrices that are already distributed across processors).

2. Notice however, that if we let the length $l$ of the Lanczos basis $V_l$ approach the size $k$ of the restricted Hamiltonian $H_k$, then it is preferable to form the restricted Hamiltonian $H_k$ explicitly and distribute it row-wise across the involved processors. This choice greatly simplifies the implementation of the parallel Lanczos algorithm, since the formation of $H_k$ can already be done in parallel in CPMD. Furthermore, note that in CPMD the application of the Hamiltonian on a vector is fully distributed among all available processors. In this case, the restricted Hamiltonian $H_k$ is distributed row-wise.

3. The basis $V_l$ can be easily distributed row-wise across the available processors, as is the standard approach followed in parallel implementations of the Lanczos algorithm. Note then that, the $xAXPY$ operations at lines 3 and 5 of the Lanczos algorithm (see Fig. 6) can be accomplished with no communication whatsoever. The only synchronization points are in line 4 and in line 6, which require global reduction operations.

4. Monitoring the convergence of eigenvalues can be cheaply calculated at every step of Lanczos. The Lanczos iteration is a variational process: approximations to eigenvalues at step $i+1$ will always be better than those of the previous step. Extremal eigenvalues tend to converge first: thus, one can monitor convergence of the smallest eigenvalue of the tridiagonal matrix $T_l$ (at every step $l$) and when this has converged
move to monitoring convergence of the next one. Cheap algorithms for the calculation of a few selected eigenvalues of symmetric tridiagonal matrices are available in LAPACK. The cost of computing only the eigenvalues (not the eigenvectors) will be at most a modest $O(l^2)$ with storage requirements $O(l)$, since $T_l$ is tridiagonal. Thus, it can be easily achieved on a single processor.

5. When convergence of all $M$ desired (leftmost) eigenvalues has been achieved, these are distributed to the available processors. Then, each processor will do a small number of inverse iteration steps with the exact eigenvalue as shift: $(T_l - \lambda I)^{-1} q_i$ on a random starting vector $q_i^{(0)}$. Two to three iterations should be enough to achieve very good convergence to targeted eigenvectors $q_i$ of $T_l$. Observe that $q_i \in \mathbb{R}^l$, $l \ll n$, thus storage requirements per processor are kept very small. However, a potential problem with this approach can arise when the converged eigenvalues are closely clustered. Alternatively, we can utilize “divide and conquer” techniques also available through LAPACK expert drivers such as xSYEVX, in which the user can choose to compute particular consecutive eigenvalues/eigenvectors. Thus, we can easily distribute the calculation of wanted eigenpairs on the involved processors. Each processor calculates only a number of consecutive eigenpairs by suitably calling routine xSYEVX. We adopted this latter approach in our current implementation the method in CPMD.

6. The approximate eigenvectors for the modified restricted Hamiltonian are computed as: $\tilde{q}_i = V_l q_i$. Since the basis $V_l$ is distributed row-wise this calculation is performed in parallel. Note that, because a processor holds complete eigenvectors $q_i$, the calculation of $\tilde{q}_i$ will require a loop of broadcast collectives. Each processor will again end up with complete (consecutive) eigenvectors $\tilde{q}_i$.

7. Finally, approximations to wave functions are similarly (see above) computed in parallel as $x_i = W_k \tilde{q}_i$. Note that matrix $W_k$ is distributed row-wise while processors hold complete consecutive eigenvectors $\tilde{q}_i$.

In Tab. 6 we give an algorithmic outline of the Lanczos method.

4.5 Practical Application of the Parallel Lanczos Algorithm

The plane-wave code CPMD, with single processor initialization, has been demonstrated to achieve excellent scalability on massively parallel systems, consisting of hundreds of thousands of cores. However, it is straightforward to see that in order for the new parallel initialization to scale analogously, a huge number atomic wavefunctions $k$ would be required (i.e. 1 million), which is far beyond our target. Thus, the implementation has to be able to utilize only a subset of the available processors. For instance, while all processors contribute in the calculation of the restricted Hamiltonian $H_k$, only a subset of them will actually be employed in the Lanczos iteration. To this end, matrix $H_k$ is distributed row-wise to these processors. This is facilitated by means of a new MPI communicator for these processors. An additional benefit is that the collective communications will be restricted to only a subset of the machine, thus reducing the overall communication latency.

The matrix-vector operation (line 3) as well as the DAXPY operations (line 3, 5) require no communication. On the other hand, the calculation of scalars $\alpha_i, \beta_i$ require global reductions (MPI_ALLREDUCE), for which very efficient implementations are available on modern supercomputer interconnection networks.
Lanczos

(*Input*) Hamiltonian $\tilde{H}$, orthonormal matrix $W_k$, starting vector $v_1$, $\|v_1\|_2 = 1$, scalar $l \leq k$

(*Output*) Orthogonal basis $V_l \in \mathbb{R}^{k \times l}$ unit norm vector $v_{l+1}$ such that $V_l^T v_{l+1} = 0$

1. Set $\beta_1 = 0, v_0 = 0$
2. for $i = 1, \ldots, l$
3. \hspace{1em} $r_i = W_k^T (\tilde{H} (W_k v_i)) - \beta_i v_{i-1}$
4. \hspace{1em} $\alpha_i = \langle r_i, v_i \rangle$
5. \hspace{1em} $r_i = r_i - \alpha_i v_i$
6. \hspace{1em} $\beta_{i+1} = \|r_i\|_2$
7. \hspace{1em} if ($\beta_{i+1} == 0$) then stop
8. \hspace{1em} $v_{i+1} = r_i / \beta_{i+1}$
9. end

Table 6. The Lanczos algorithm for matrix $W_k^T \tilde{H} W_k$. The inner product for vectors is denoted by $\langle \ldots, \ldots \rangle$.

Reorthogonalization of Lanczos vectors It is well known that although the Lanczos iteration theoretically ensures orthogonality among the basis vectors, in practice the basis vectors quickly loose orthogonality due to roundoff. To remedy this we employ reorthogonalization at each step. We stress though that future versions of CPMD will utilize techniques for partial reorthogonalization\textsuperscript{17–19}, that perform orthogonalization only when it is deemed necessary. Reorthogonalization is performed by means of standard Gram-Schmidt\textsuperscript{16}. At each step $i$, and before calculation of scalar $\beta_{i+1}$ (see also previous section):

- Compute the local projection coefficient $w_{i-1}^l = V_{i-1}^* r_i$. No communication required.
- Compute the global projection coefficients $w_j^l$ by global reduction on the local projection coefficients $w_{i-1}^l$ (MPI ALLREDUCE).
- Compute the reorthogonalized $r_i^* = r_i - \sum_{j=1}^{i-1} w_j^l V_j$. No communication required.

Observe that standard Gram-Schmidt (GS) reorthogonalization induces only an additional collective operation per Lanczos step. We note that although modified GS is known to be more stable than standard GS (see Ref. 16), it requires $i - 1$ additional collectives at each step $i$, which includes $O(l^2)$ additional collectives for a total of $l$ Lanczos steps. Since we are only interested in initial guesses for the wavefunctions we opt to use standard GS. In practical applications so far we have not encountered a problem of unrecoverable severe loss of orthogonality. However, if this happens, we can always temporarily switch to modified GS.

4.6 Numerical Examples

We now illustrate the scalability performance of the parallel Lanczos algorithm for distributed atomic wavefunctions initialization. We experimented with a family of super...
cells of silicon bulk, ranging $N = 512, 1024$ and $2048$ atoms (see Fig. 10). For the first case ($N=512$) we used a cutoff energy of 20 Rydbergs while for the larger cases of $N = 1024, 2048$ atoms the cutoff energy was set to 12 and 8 Rydbergs respectively. The cutoff energy controls the dimension of the plane-wave basis on which the Hamiltonian and the wavefunctions are expanded (a larger value for the cutoff energy results into more plane-waves).

The smallest case ($N=512$) is a cubic mesh (with cube edge equal to 41.0449) and the larger two cells were generated by replication of the smallest cell along the $X$ axis. The sizes of the restricted Hamiltonians were $4 \times N = 2048, 4096, 8192$. CPMD utilized in all runs 128 compute nodes of a BG/L system, while for the atomic wavefunctions initialization we utilized a subset of $2^k, k = 1, \ldots, 7$ nodes. We stress that the ability to utilize only a subset of the available compute nodes is crucial in achieving good scaling for the initialization, while other parts of CPMD can take advantage of the full set of available processors.
Table 7. Run times for distributed initialization using the parallel Lanczos algorithm.

<table>
<thead>
<tr>
<th>#CPUs</th>
<th>N=512 time (secs)</th>
<th>N=1024 time (secs)</th>
<th>N=2048 time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>90</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>46</td>
<td>381</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>191</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>13</td>
<td>99</td>
<td>738</td>
</tr>
<tr>
<td>32</td>
<td>9</td>
<td>54</td>
<td>382</td>
</tr>
<tr>
<td>64</td>
<td>8.6</td>
<td>36</td>
<td>211</td>
</tr>
<tr>
<td>128</td>
<td>8.7</td>
<td>30</td>
<td>114</td>
</tr>
</tbody>
</table>

Tab. 7 illustrates run times for the distributed initialization. The dashes in the columns of Tab. 7 indicate that no run was possible for the corresponding number of CPUs because of node memory was not enough to hold the data. Remember that we have implemented the parallel Lanczos algorithm in CPMD in such a way that only a subset of the available processors are actually employed for the initialization. This versatility is crucial when we utilize thousands of processors with CPMD. It is clear that as the dimension of the restricted Hamiltonian increases, the distributed Lanczos algorithm scales better. For example, while for \( N = 512 \) scaling stops at 32 processors, for \( N = 1024 \) it stops at 64 and for \( N = 2048 \) scaling continues up to 128 processors.

In Tab. 8 we provide detailed run times (in seconds) for the various computational stages of the distributed initialization. \texttt{MATVEC} is the time spent multiplying the restricted Hamiltonian by the current Lanczos vector (line 3). \texttt{REORTH} is the time spent for re-orthogonalizing the new Lanczos vector. \texttt{COLL} is the time for collective communications in the Lanczos loop (excluding the \texttt{REORT}). \texttt{DAXPY} is the time spent in BLAS daxpy operations within the Lanczos loop. \texttt{L. EIGS} is the time spent to calculate the Lanczos eigenvectors and eigenvectors (immediately after the Lanczos loop has run). \texttt{VECS} is the time spent to multiply the Lanczos eigenvalues with the atomic wavefunctions \( W_k \) to get the final estimation of the initial wavefunctions. Remember that matrix \( W_k \) is row-wise distributed across all available processors, while the eigenvectors of the restricted Hamiltonian are distributed column-wise across the group of processors that participate in the Lanczos run.

Clearly, the matrix-vector operation (\texttt{MATVEC}) scales very well in all cases and the \texttt{DAXPY} operations contribute only minimally to the overall cost. The computation of the eigenvalues and eigenvectors of the Lanczos matrix (\texttt{L. EIGS}) also scales very well, especially for the larger problems. Reorthogonalization exhibits satisfactory scaling which is attributed to the modest additional communication cost of standard GS and to the very fast collective communication available on the BlueGene machines. Finally, we see that the part that does not scale is the final calculation of the approximate initial wavefunctions. This is expected, since the total number of messages in this part increases as we increase the number of processors in the Lanczos group. To see this, remember that the Lanczos eigenvectors are distributed column-wise to the group of processors that takes part in the Lanczos loop, i.e. each processor in this group holds a number of consecutive Lanczos eigenvectors. On the other hand, remember that matrix \( W_k \) is distributed row-wise across
Table 8. Detailed run times, in seconds, for the various stages of distributed initialization using the parallel Lanczos algorithm.

<table>
<thead>
<tr>
<th>N=512</th>
<th>#CPUs</th>
<th>MATVEC</th>
<th>REORT</th>
<th>COLL</th>
<th>DAXPY</th>
<th>L. EIGS</th>
<th>VECS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>36</td>
<td>49.2</td>
<td>1.2</td>
<td>0.07</td>
<td>4.9</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>18</td>
<td>24.3</td>
<td>1.4</td>
<td>0.05</td>
<td>2.6</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.2</td>
<td>11.9</td>
<td>1.3</td>
<td>0.03</td>
<td>1.4</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3</td>
<td>6.2</td>
<td>1.6</td>
<td>0.03</td>
<td>0.87</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.5</td>
<td>3.5</td>
<td>1.8</td>
<td>0.02</td>
<td>0.51</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.8</td>
<td>2.2</td>
<td>1.7</td>
<td>0.02</td>
<td>0.41</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.4</td>
<td>1.3</td>
<td>0.3</td>
<td>0.03</td>
<td>0.5</td>
<td>6.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N=1024</th>
<th>#CPUs</th>
<th>MATVEC</th>
<th>REORT</th>
<th>COLL</th>
<th>DAXPY</th>
<th>L. EIGS</th>
<th>VECS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>141</td>
<td>201</td>
<td>20</td>
<td>0.12</td>
<td>15.1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.0</td>
<td>97</td>
<td>9</td>
<td>0.09</td>
<td>9.7</td>
<td>4.4</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>35</td>
<td>48</td>
<td>6.5</td>
<td>0.07</td>
<td>4.8</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>14</td>
<td>24</td>
<td>5</td>
<td>0.06</td>
<td>2.7</td>
<td>6.9</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>6.5</td>
<td>13</td>
<td>4.8</td>
<td>0.06</td>
<td>1.7</td>
<td>9.6</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>3.2</td>
<td>7.7</td>
<td>1</td>
<td>0.05</td>
<td>1.2</td>
<td>16.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N=2048</th>
<th>#CPUs</th>
<th>MATVEC</th>
<th>REORT</th>
<th>COLL</th>
<th>DAXPY</th>
<th>L. EIGS</th>
<th>VECS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>277</td>
<td>384</td>
<td>33</td>
<td>0.16</td>
<td>35</td>
<td>11.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>137</td>
<td>191</td>
<td>25</td>
<td>0.15</td>
<td>16</td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>69</td>
<td>97</td>
<td>19</td>
<td>0.12</td>
<td>9</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>29</td>
<td>53</td>
<td>5</td>
<td>0.12</td>
<td>5</td>
<td>21.6</td>
</tr>
</tbody>
</table>

all of the available processors. Thus, this part of the calculation involves collectives that span the full breadth of the machine. Indeed, the number of these broadcast collective operations increases as the number of processors in the Lanczos group increases. However, we also note that for the larger case (N=2048) the veCS part is only a fraction of the total cost (roughly %20), which explains the very good scaling of the scheme for this case.

4.7 Discussion

Initialization from atomic wavefunctions in ab initio molecular dynamics codes is crucial in order to facilitate large scale next generation simulations with thousands of atoms. This initialization leads to very large dense eigenproblems that are impossible to solve on a single processor, both in terms of computational complexity as well as of memory requirements. In this paper we are reporting a new scheme for distributed initialization that is based on a distributed version of the Lanczos algorithm. Our decision to use Lanczos instead of parallel dense methods such as the ones in SCALAPACK20 is based on the following key observations:
• We needed to respect as much as possible the existing data structures in CPMD, which is the host molecular dynamics code. The distribution of the matrices involved, in terms of plane-waves, significantly favors matrix vector operations, rather than matrix transformations-factorizations that are inherit in parallel dense linear algebra.

• We can safely use in practice standard Gram-Schmidt reorthogonalization, instead of modified Gram-Schmidt, which induces only one additional collective operation at each step of the parallel Lanczos algorithm.

• Our target computational platform is the BlueGene supercomputer series which is equipped with an excellent separate network for collective communications.

We point out that although good scalability of the new scheme is of course a desired property it is not of crucial importance. This is due to the fact that the total run time of large simulations is by far dominated by the minimization of the Kohn-Sham equations (after atomic wavefunction initialization has run) and the subsequent molecular dynamics simulation thereafter. For example, for the case of 1024 Silicon atoms (see previous section) the new distributed initialization scheme on 8 BlueGene /L nodes required 193 seconds while the total run time for minimization was 2400 seconds. Then, each step, out of the thousands, of the molecular dynamics run costs itself roughly the same as the distributed initialization. However, if it was not for this successful initialization from atomic wavefunctions, the first minimization would require an enormous number of iterations in order to converge.

5 Discussion

We have described our efforts towards massively parallel electronic structure calculations. We have focused in efficient parallel 3D FFTs, wavefunction orthogonalization and initialization from atomic wavefunctions. We demonstrate that extreme scaleout is indeed possible, allowing for routine simulations with tens of thousands of atoms in very reasonable time frames. We followed an approach of algorithmic redesign and extensive software reengineering. This means that we adopted algorithms that, although at first looked unsuitable, in fact allowed extreme scaleout. This is proof that we have much to gain by re-thinking and adopting basic computational kernels, especially in view of many core nodes, with lower memory and bandwidth per core, that Exascale machines are projected to be composed of.

References