Computer Physics Communications 258 (2021) 107521

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Efficient update of determinants for many-electron wave function overlaps☆

Pedro Alonso-Jordá [a,*], Davor Davidović [b], Marin Sapunar [c], José R. Herrero [d], Enrique S. Quintana-Ortí [e]

[a] Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
[b] Centre for Informatics and Computing, Ruđer Bošković Institute, Croatia
[c] Division of Physical Chemistry, Group for Computational Life Sciences, Ruđer Bošković Institute, Croatia
[d] Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain
[e] Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Spain

ARTICLE INFO

Article history:
Received 9 June 2020
Accepted 24 July 2020
Available online 6 August 2020

Keywords:
Nonadiabatic dynamics
Surface hopping
Determinant
LU factorization
High performance computing

ABSTRACT

The calculation of overlaps between many-electron wave functions at different nuclear geometries during nonadiabatic dynamics simulations requires the evaluation of a large number of determinants of matrices that differ only in a few rows/columns. While this calculation is fast for small systems, its cost grows faster than the alternative electronic structure calculation used to obtain the wave functions. For wave functions that can be written as a CIS expansion, all determinants can be computed using the set of level-2 minors of the reference matrix. However, this is still a costly computation for large systems.

In this paper, we provide an algorithm for efficiently calculating all level-2 minors of a matrix by re-utilizing and updating the LU factorization for the determinants of the minors. This approach results in a parallel version of the algorithm that is up to an order of magnitude faster then the current best parallel implementation. The algorithm thus allows the computation of exact wave function overlaps for relatively large systems, with a high density of states, at virtually no cost compared with the electronic structure calculations. Furthermore, the new algorithm opens the path to further investigations in efficient computing of the exact wave function overlaps for complex wave functions such as MR-CIS and MR-CISD.

Program summary

Program Title: CIS Overlap
Licensing provisions: MIT
Programming language: FORTRAN 2008, C
Nature of problem: Calculation of overlaps between CIS type wave functions at different nuclear geometries during nonadiabatic dynamics simulation requires calculating a large number of connected determinants and scales with the seventh power of the size of the system being studied. Without additional approximations, for large systems this computation becomes more costly than the electronic structure calculation used to obtain the wave functions.
Solution method: All of the determinants required for a CIS wave function overlap calculation can be derived from the set of all level-2 minors of the matrix of overlaps between the reference Slater determinants. We developed an algorithm for efficiently computing all level-2 minors of a matrix. This part of the computational process was the bottleneck in previous solutions to the problem and is now an order of magnitude faster resulting in significantly faster overall calculation of the wave function overlaps.

1. Introduction

Modeling photochemical and photophysical phenomena in molecules requires analyzing the evolution of the nuclear and electronic degrees of freedom while taking into account a

☆ The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author.
E-mail addresses: palonso@upv.es (P. Alonso-Jordá), davor.davidovic@irb.hr (D. Davidović), marin.sapunar@irb.hr (M. Sapunar), josepr@ac.upc.edu (J.R. Herrero), quintana@disca.upv.es (E.S. Quintana-Ortí).

https://doi.org/10.1016/j.cpc.2020.107521
0010-4655/

manifold of electronic states. These problems are often tackled using mixed quantum–classical nonadiabatic dynamics [1,2], where only the electronic degrees of freedom are treated fully quantum mechanically and the electronic states are coupled through time-derivative couplings (TDCs). The matrix of overlaps between many-electron wave functions (MEWF) at successive time steps was introduced by Hammes-Schiffer and Tully [3], in the context of fewest-switches surface hopping calculations (FSSH) [4], to numerically compute TDCs. Initially, this was considered an approximation to the analytic computation of TDCs using nonadiabatic coupling vectors, which are more expensive to obtain and more complex to implement. Over time it was proved to be a necessary tool for dealing with trivial or near trivial crossings in nonadiabatic dynamics simulations. At these crossings, nonadiabatic couplings are highly peaked and cannot be assumed to change linearly between two time steps unless prohibitively small time steps are used. This problem is an active area of research in the field of nonadiabatic dynamics, and a number of solutions, all based on overlaps, have been proposed [5–10].

The wave functions (WFs) are typically expanded into Slater determinants (SDs), and the elements of the overlap matrix S are defined as

$$S_{AB} = \left\langle \Psi_A \middle| \Psi'_B \right\rangle = \sum_a^{n_{SD}} \sum_b^{n'_{SD}} d_{Aa} d_{Bb} \left\langle \Theta_a \middle| \Theta'_b \right\rangle , \tag{1}$$

where $|\Psi_A\rangle$ and $|\Psi'_B\rangle$ are the wave functions at subsequent steps in the dynamics; and $\{|\Theta_a\rangle\}$ and $\{|\Theta'_b\rangle\}$ are two distinct sets of SDs. Orbital basis sets to build the SDs are then defined at specific nuclear coordinates, which change with the time step in the dynamics. This means that the basis sets for $|\Psi_A\rangle$ and $|\Psi'_B\rangle$ are not orthogonal, and the formulation requires explicitly calculating each $\left\langle \Theta_a \middle| \Theta'_b \right\rangle$ term in (1). This requires the computation of the determinant of an $n \times n$ matrix [11] (where n is the number of occupied orbitals) for each term, so that the cost of this calculation grows very quickly with the dimension of the system and complexity of the wave functions.

Several methods have been developed to obtain these overlaps for various wave function types [12–14]. In a previous paper [14], two algorithms were developed to efficiently compute overlaps for configuration interaction singles (CIS) wave functions. These algorithms can be applied to obtain wave function overlaps for the single-reference electronic structure methods most commonly used for nonadiabatic dynamics simulations: (1) time-dependent density functional theory (TDDFT) [15,16]; and (2) with some approximations, algebraic diagrammatic construction scheme to second order (ADC(2)) [17,18]. The latter is usually leveraged to tackle relatively small systems, and the overlap calculation scheme presented in [14] is significantly faster than the electronic structure calculation for any system studied using this method. However, TDDFT scales more favorably with the system size and can be used to treat much larger systems at a low cost, especially with the use of graphics processing units (GPUs) [19,20]. One of the developed algorithms, named ONTO, was based on transforming the WFs into a compact natural transition orbital (NTO) basis, significantly reducing the number of SDs in the expansion. An alternative algorithm, named OL2M, exploits that all SDs in a CIS expansion differ by at most two orbitals, and therefore can be computed using the set of all level-2 minors of the reference SD. While the ONTO algorithm is significantly faster when calculating a single overlap matrix element, the OL2M algorithm does not depend on the number of electronic states, yielding a faster method for large overlap matrices. This is especially important for large systems with a high density of excited states. The idea of using an expansion into the minors to calculate the overlaps between many SDs is also more generally applicable to different types of wave functions, whereas the compactness of the NTO basis is specific to CIS wave functions. However, calculating all of the minors for the OL2M algorithm is still a computationally expensive task.

In this work, we significantly improve the OL2M algorithm, accelerating it by an order of magnitude compared with the previous version [14], which allows calculating the overlap matrices for medium size systems (≈100 atoms) with a high density of states at a fraction of the cost of TDDFT calculations for such systems. In addition to this, reducing the cost of the minor computations paves the road to obtaining exact overlap calculations for more complex wave function types, such as CI singles and doubles (CISD) or multi-reference (MR) versions of CIS and CISD.

The rest of the paper is organized as follows: In Section 2. we briefly review the Mathematical-Physics problem. Next, in Section 3, we introduce a straight-forward algorithm for computing the determinants of the minors and the basic columnwise re-utilization algorithm [14,21]. In Section 4, we describe in detail several algorithmic (and implementation) optimization strategies that can be applied to the columnwise re-utilization algorithms. The sequential and parallel realizations of all these algorithms are then evaluated in Section 5; and the results of applying the algorithm for computing full overlap matrix on several molecular systems as well as a comparison with existing solutions are presented in Section 6. Finally, a few concluding remarks and a discussion of the application of the algorithm to compute the overlaps of more complex wave functions close the paper in Section 7.

2. Mathematical-physics problem

Inserting a CIS wave function ansatz into (1), we obtain that

$$S_{AB} \propto \sum_a^n \sum_b^n \sum_i^p \sum_j^p d_{ai}^A d_{bj}^B \left\langle \Theta_a^i \middle| \Theta_b^{\prime j} \right\rangle , \tag{2}$$

where n and p respectively denote the number of occupied and virtual orbitals; $|\Theta_a^i\rangle$ are SDs with the occupied orbital a replaced with virtual orbital i; and d_{ai}^A are the coefficients of these SDs in wave function A. For simplicity, we do not consider electron spin in this expression and only include the term that dominates the computational cost. For a detailed derivation, see [14].

In this work, the key problem from the computational point of view is the efficient calculation of all the $\left\langle \Theta_a^i \middle| \Theta_b^{\prime j} \right\rangle$ elements. Each such element corresponds to the determinant of the matrix of overlaps between the orbitals that make up the SD $|\Theta_a^i\rangle$ and the SD $|\Theta_b^{\prime j}\rangle$. Fig. 1 depicts the representation of the matrix of orbital overlaps split into four blocks: A_{ref} (occupied orbital overlaps), VV (virtual orbital overlaps), and OV/VO (occupied/virtual orbital overlaps). Then, $\left\langle \Theta_a^i \middle| \Theta_b^{\prime j} \right\rangle$ is equal to the determinant of A_{ref} with its row a and column b replaced by row i and column j from the VO and OV blocks, respectively; and the element (a, b) of A_{ref} replaced with entry (i, j) taken from the VV block. The main computational problem thus consists in obtaining these determinants for all possible combinations of a, b, i, j.

The total number of determinants (combinations) is n^2p^2. Therefore, provided the LU factorization [22] is leveraged to compute these determinants, (which requires a cubic number of floating-point operations, or flops, on n), this requires a total of $O(n^5p^2)$ flops, and this approach quickly becomes even more expensive than the corresponding electronic structure calculations.

A relevant property of the overlap calculation is that, for a given fixed row/column pair (a, b) of the referent matrix A_{ref}, all rows from VO and columns from OV are to be tested, while all other rows and columns of A_{ref} remain unchanged. An initial

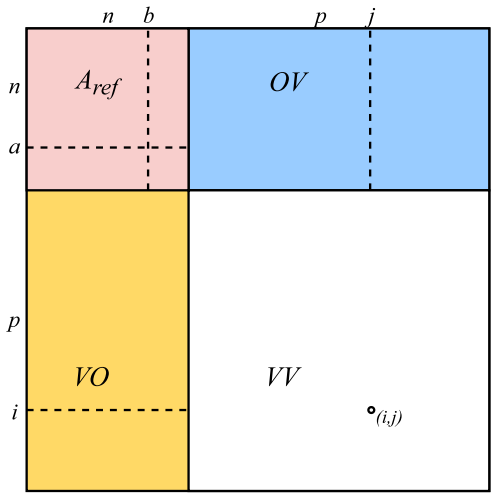

Fig. 1. Matrix representation of the computational problem. The referent matrix A_{ref} is square of order nn. Rows/columns a/b of A_{ref} are replaced with rows/columns i/j from blocks VO/OV, respectively; and element (a, b) of the same matrix is replaced with entry (i, j) of block VV.

approach to exploit this property was presented in [13]. There, the matrix A_{ref} was expanded by columns using the Laplace transformation, and the determinants of the minors were then stored and reused to obtain the determinants of those matrices which differed only in one column from the baseline determinant. This idea was later enhanced in [14], where the determinants of the minors were obtained from the second-level recursive Laplace expansion. The minors were constructed by expanding the referent matrix along both rows and columns; next, the LU factorization was repeatedly applied to obtain the determinants; and finally these results were reused to compute the overlap of the corresponding Slater determinants. In the remainder of this paper, we will refer to this approach as DL2M.

Obtaining the determinants of the minors poses a major computational bottleneck unless very large basis sets are used ($p \gg n$). As these minors only differ in a row/column pair, this opens the door to leverage some updating/downdating techniques for the LU factorization [22] in order to obtain the sought-after determinants with a moderate cost. In the next section we describe our efforts towards reducing the computational costs of this process taking advantage of those techniques.

3. Algorithms for DL2M calculations

3.1. Original formulation

Rename $A = A_{ref} \in \mathbb{R}^{n\times n}$, and let us introduce the notation $M_{r\|c}$ to refer to the submatrix that results from eliminating from M the rows and columns with the indices included into the sets $\{r\}$ and $\{c\}$, respectively. For example, $A_{r_1,r_2\|c_1,c_2}$ is the submatrix that results from eliminating rows r_1, r_2 and columns c_1, c_2 from A. Furthermore, the dash in $M_{-\|c}$ ($M_{r\|-}$) denotes that all rows (columns) of M are kept.

The calculation of the overlap between any two MEWFs described in [13] requires the computation of the determinants for all possible submatrices that result from eliminating any two rows/columns of A. Therefore, a straightforward yet costly solution to obtain the determinants consists of explicitly constructing all possible square submatrices of A, of order $m = n - 2$, and then computing the LU factorization (with partial pivoting) of each submatrix. The `naive` algorithm that realizes this approach is shown in Listing 1. For brevity, hereafter we employ a pseudo-Matlab notation in the presentation of the algorithms. Moreover, for simplicity, we omit the effect of the row permutations performed during the LU factorization on the sign of the determinant. Nevertheless, all our actual realizations of the Gauss-based algorithms include partial pivoting to ensure numerical stability in practice.

Table 1
Computational costs in flops for the triangularization of $\bar{m} \times \bar{n}$ dense and upper quasi-triangular matrices, with $\bar{m} \leq \bar{n}$ and real entries, using Gauss transforms and Givens rotations; see [22] for details.

Matrix	Gauss	Givens
Dense	$\bar{m}^2(\bar{n} - \bar{m}/3)$	$3\bar{m}^2(\bar{n} - \bar{m}/3)$
Hessenberg with s nonzero subdiag.	$2s(\bar{m}\bar{n} - \bar{m}^2/2)$	$6s(\bar{m}\bar{n} - \bar{m}^2/2)$

The computational cost of the `naive` realization is, approximately,

$$\sum_{r_1=1}^{n-1}\sum_{r_2=r_1+1}^{n}\sum_{c_1=1}^{n-1}\sum_{c_2=c_1+1}^{n} \underbrace{\frac{2}{3}m^3 \text{ flops}}_{\text{LU of } A_{r_1,r_2\|c_1,c_2}} \approx \frac{n^7}{6} \text{ flops}.$$

In this expression and other costs presented later, we assume that the LU factorization of a square matrix of order n costs $\frac{2}{3}n^3$ flops. We perform the same approximation for matrices with $m \approx n$ rows/columns, such as those that are actually involved in the factorizations in the `naive` algorithm.

Table 1 offers a few relevant computational costs for specialized matrix triangularization routines employed in our work. The minor order terms are neglected in the cost expressions hereafter.

```
1 function [A] = naive(A)
2 % Input dense matrix
3 n = size(A,2);
4 for r1 = 1:n-1
5     for r2 = r1+1:n
6         for c1 = 1:n-1
7             for c2 = c1+1:n
8                 [L,U,P] = lu(A_{r1,r2||c1,c2});
9                 det[r1][r2][c1][c2] = prod(diag(U));
```

Listing 1: Algorithm `naive` for DL2M calculations.

3.2. Columwise re-utilization

The `naive` algorithm in Listing 1 exposes that, between any two iterations of the two innermost loops (that is, those indexed by c_1, c_2), the matrices that are factorized only differ in two columns. In addition, the computational cost of the algorithm is dominated by the operation(s) performed in the innermost loop. In [21], the authors introduced a variant of the `naive` algorithm that exploited these two observations in order to update an initial factorization, computed at the beginning of each iteration of the loop indexed by r_2, and obtain from that all the determinants of the two innermost loops with a reduced cost.

In some detail, assume that

$$L^{-1}PA_{r_1,r_2\|-} = U, \tag{3}$$

is an LU factorization of $A_{r_1,r_2\|-}$, where $L \in \mathbb{R}^{m\times m}$ is a unit lower triangular matrix that accumulates the Gauss transforms that were applied to annihilate the subdiagonal entries of the initial matrix; $P \in \mathbb{R}^{m\times m}$ is the permutation matrix due to the application of partial pivoting during the factorization; and $U \in \mathbb{R}^{m\times n}$ is the resulting upper triangular factor [22]. Then, the

application of the same Gauss transforms and permutations of (3) to matrix $A_{r_1,r_2\|c_1,c_2}$, results in

$$L^{-1}PA_{r_1,r_2\|c_1,c_2} \;=\; U_{-\|c_1,c_2} \in \mathbb{R}^{m\times m}, \tag{4}$$

which corresponds to the matrix that is obtained by eliminating columns c_1, c_2 from U.

The key to the cost savings for the algorithm described in [21] lies in exploiting the "close-to-upper triangular" structure of $U_{-\|c_1,c_2}$ when reducing this matrix to upper triangular form. Concretely, consider the partitioning

$$U = \left[\; U_1 \;\middle|\; u_1 \;\middle|\; U_2 \;\middle|\; u_2 \;\middle|\; U_3 \;\right] \quad \text{and} \quad U_{-\|c_1,c_2} = \left[\; U_1 \;\middle|\; U_2 \;\middle|\; U_3 \;\right], \tag{5}$$

where u_1 and u_2 denote, respectively, the c_1th and c_2th columns of U. Then, $U_1 \in \mathbb{R}^{m\times c_1-1}$ is already upper triangular. Moreover, $U_2 \in \mathbb{R}^{m\times c_2-c_1-1}$ and $U_3 \in \mathbb{R}^{m\times n-c_2}$ respectively contain only one and two nonzero subdiagonals, which need to be annihilated in order to obtain the desired upper triangular factor yielding the sought-after determinant; see Fig. 2.

In order to compute the determinant of $U_{-\|c_1,c_2}$, it is then possible to exploit this special structure to significantly reduce the cost of the global algorithm. Specifically, consider the partitioning $U_{23} = \left[\frac{U_T}{u_B}\right]$, where u_B corresponds to the bottom row of U_{23}. The `base` algorithm that exploits the relationship between the matrices factorized in the inner two loops proposed in [21] is given in Listing 2. Note that the LU factorizations involving blocks $[U_{22}, U_{23}]$ and U_{33} leverage the special upper quasi-triangular structure of these submatrices to reduce the cost of this method.

In [21], the reduction of the appropriate blocks of U_2, U_3 to upper triangular form was done via Gauss transforms inside the innermost loop (indexed by c_2). Taking into account the costs in Table 1, the global cost of the algorithm in [21] is given by

$$\sum_{r_1=1}^{n-1}\sum_{r_2=r_1+1}^{n}\left(\underbrace{\frac{2}{3}n^3}_{\text{LU of } A_{r_1,r_2\|c_1,c_2}} + \cdot \sum_{c_1=1}^{n-1}\sum_{c_2=c_1+1}^{n}\right.$$
$$\left.\times\left(\underbrace{2((c_2-c_1)(n-c_1)-(c_2-c_1)^2/2)}_{\text{LU of } [U_{22},U_{23}]} + \underbrace{2(n-c_2)^2}_{\text{LU of } [u_B;U_{33}]}\right)\right)$$
$$\approx \;\tfrac{3}{16}n^6 \text{ flops.}$$

Unfortunately, the "reversed" formulation of the algorithm presented in [21], with the third loop specified as $c_1 = 1 : n$ and the fourth loop as $c_2 = 1 : c_1 - 1$, prevented the authors from detecting an additional re-utilization, which is implicit in the formulation in the `base` algorithm in Listing 2.

4. Improved column re-utilization for DL2M calculations

Consider again the partitioning of U in (5) and the matrix that results by eliminating only column c_1 from that:

$$U_{-\|c_1} = \left[\; U_1 \;\middle\|\; U_2 \;\middle|\; u_2 \;\middle|\; U_3 \;\right]. \tag{6}$$

By re-inserting c_2 into the diagram on the left-hand side of Fig. 2, we observe that this matrix only differs from the desired upper triangular matrix in that $\left[\; U_2 \;\middle|\; u_2 \;\middle|\; U_3 \;\right]$ presents an extra subdiagonal. Consider next the factorization

$$\bar{L}^{-1}\bar{P}U_{-\|c_1} = \left[\; U_1 \;\middle\|\; \bar{L}^{-1}\bar{P}\left(\left[\; U_2 \;\middle|\; u_2 \;\middle|\; U_3 \;\right]\right) \;\right] \tag{7}$$
$$= \left[\; U_1 \;\middle\|\; \bar{U}_2 \;\middle|\; \bar{u}_2 \;\middle|\; \bar{U}_3 \;\right] = \bar{U}_{-\|c_1} \in \mathbb{R}^{m\times(n-1)},$$

where $P \in \mathbb{R}^{m\times m}$ is a permutation matrix, and $\bar{L} \in \mathbb{R}^{m\times m}$ represents an aggregation of Gauss transforms into the unit lower triangular matrix which yields the desired upper triangularization. Note that these permutations/transforms do not modify U_1 as this block is already upper triangular.

The key to the improved algorithm proposed in our work lies in recognizing that the factorization in (7) can be again re-utilized when any column $\bar{u}_2$ (indexed by c_2) is eliminated from the factorization, since the effect of permutations and Gauss transforms, when applied to a matrix from the left, are "independent" for each matrix column.

In particular, from (7), we observe that the application of the permutations and transforms yields

$$\bar{L}^{-1}\bar{P}\left(\left[\; U_1 \;\middle\|\; U_2 \;\middle|\; U_3 \;\right]\right) = \left[\; U_1 \;\middle\|\; \bar{U}_2 \;\middle|\; \bar{U}_3 \;\right], \tag{8}$$

which only differs from the desired upper triangular factor in that $\bar{U}_3$ has a *single* nonzero subdiagonal, as u_2 has been eliminated from the expression. Restoring the upper triangular form can then be obtained from a new sequence of permutations and Gauss transforms that annihilate the nonzero entries in the subdiagonal of block $\bar{U}_3$.

The algorithm with improved re-utilization is given in Listing 3. The cost of the method is dominated by that of the factorization in the innermost loop, which results in

$$\sum_{r_1=1}^{n-1}\sum_{r_2=r_1+1}^{n}\left(\underbrace{2n^3/3}_{\text{LU of } A_{r_1,r_2\|c_1,c_2}} + \sum_{c_1=1}^{n-1}\right.$$
$$\left.\times\left(\underbrace{2(n-c_1)^2}_{\text{LU of } [U_2,u_2,U_3]} + \sum_{c_2=c_1+1}^{n}\underbrace{(n-c_2)^2}_{\text{LU of } \bar{U}_3}\right)\right) \;\approx\; \tfrac{n^6}{24} \text{ flops.}$$

This represents a reduction in the cost of the algorithm with column re-utilization from [21], described in the previous section, in a factor of 4.5×.

4.1. Partial pivoting with implicit permutations

The `reduceH_bs` routine in Listing 4 computes the LU factorization of an (upper) Hessenberg matrix H (that is, an upper quasi-triangular matrix with a single nonzero subdiagonal), yielding an (upper) triangular factor that overwrites the original matrix. The procedure there includes partial (row) pivoting for numerical stability and takes into consideration the nonzero entries below the first subdiagonal to reduce its cost to $2(\bar{m}\bar{n}-\bar{m}^2/2)$ flops, for a matrix $H \in \mathbb{R}^{\bar{m}\times\bar{n}}$, with $\bar{m} \le \bar{n}$; see Table 1.

For large matrices, the factor that dictates the performance of the reduction procedure in Listing 4 on a current computer architecture is not the amount of flops but the number of memory accesses (or memory operations, memops). For the type of matrices involved in MEWF overlap computations, with a dimension that varies between a few dozens and a couple of hundreds, the data easily fits into the cache memory of the processor so that, from the performance point of view, flops and memops are both relevant.

A close inspection of the routine in Listing 4 shows that the application of a Gauss transform requires 1.5 memory accesses per flop (see Line 13, where we assume that *pivot* remains in a processor register so that two elements are loaded and one is stored for each addition and multiplication). The row permutations due to pivoting (Lines 7–9) will require, on average, 4 additional memory accesses during half of the loop iterations (the auxiliary vector *vtmp* is assumed to remain in a processor register).

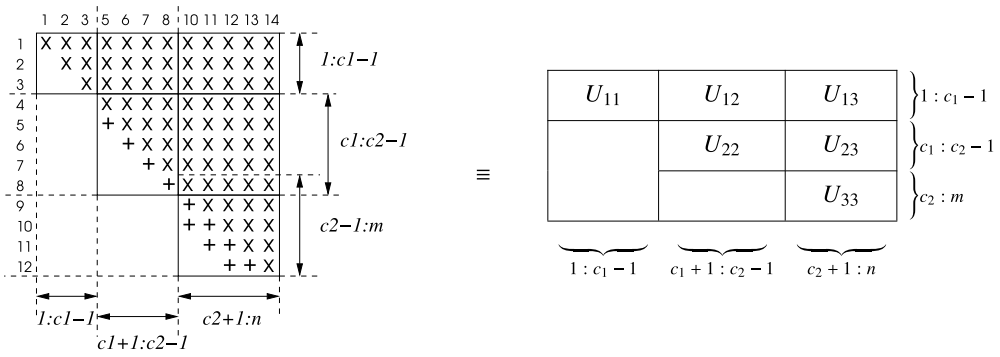

Fig. 2. Structure of the upper quasi-triangular matrix $U_{-\|c_1,c_2}$, with $n = 14$, $m = n - 2 = 12$, $c_1 = 4$ and $c_2 = 9$. Nonzero entries below the main diagonal are identified with the symbol '+'. The columns are numbered taking into account that the c_1th and c_2th were eliminated from U.

```
function [A] = base(A)
% Input dense matrix
n = size(A,2);
for r1 = 1:n-1
    for r2 = r1+1:n
        [L,U,P] = lu(A_{r1,r2||-});
        for c1 = 1:n-1
            for c2 = c1+1:n
                [L2,U2,P2] = reduceH([U22, U23], 1); % Exploit zeros below first subdiagonal
                [L3,U3,P3] = reduceH2([uB; U33], 1); % Exploit zeros below second subdiagonal
                det[r1][r2][c1][c2] = prod(diag(U11)) * prod(diag(U2)) * prod(diag(U3));
```

Listing 2: Algorithm `base` for DL2M calculations with column re-utilization. The partitionings of U that define the blocks $U_{11}, U_{22}, U_{23}, u_B, U_{33}$ for this algorithm are given in (5) and Fig. 2. Routines `reduceH`, `reduceH2` denote specialized versions of the LU factorization that respectively annihilate the first and two first subdiagonals of the first input parameter, starting at the row with the index specified by the second input parameter.

```
function [A] = opt(A)
% Input dense matrix
n = size(A,2);
for r1 = 1:n-1
    for r2 = r1+1:n
        [L,U,P] = lu(A_{r1,r2||-});
        for c1 = 1:n-1
            [L̄,Ū,P̄] = reduceH([ U2 | u2 | U3 ], c1); % Exploit zeros below first subdiagonal
            for c2 = c1+1:n
                [L̂3,Û3,P̂3] = reduceH(Ū3, c2);      % Exploit zeros below first subdiagonal
                det[r1][r2][c1][c2] = prod(diag(U1)) * prod(diag(Ū32)) * prod(diag(Û33));
```

Listing 3: Algorithm `opt` for DL2M calculations with improved column re-utilization. The partitionings of U that define blocks U_2, u_2, U_3, U_1 for this algorithm are given in (5). The blocks for $\bar{U}_{22}, \hat{U}_{33}$, respectively, correspond to the same entries of blocks U_{22}, U_{33} of U in Fig. 2. Routine `reduceH` denotes a specialized version of the LU factorization that annihilates the first subdiagonal of the first input parameter, starting at the row with the index specified by the second input parameter.

```
function [H] = reduceH_bs(H)
% Input Hessenberg matrix overwritten with upper triangular factor from the factorization
[p,q] = size(H);

for k = 1:min(p,q)-1
  if abs(H(k+1,k))>abs(H(k,k))
    vtmp       = H(k+1,k:q);
    H(k+1,k:q) = H(k,k:q);
    H(k,k:q)   = vtmp;
  end
  pivot        = H(k+1,k)/H(k,k); % Diagonal entry --> Update determinant
  H(k+1,k)     = 0.0;
  H(k+1,k+1:q) = H(k+1,k+1:q) - pivot*H(k,k+1:q);
end
```

Listing 4: Routine `reduceH_bs` for the reduction of a Hessenberg matrix to triangular form via Gauss transforms with partial (row) pivoting.

A technique to reduce the amount of memory accesses in this factorization is to perform the permutations implicitly, without doing any actual row interchange. This implies keeping track of the row pivot index during the factorization, and performing the update on the nonpivot row, as shown in the `reduceH_ip` routine in Listing 5. This effectively reduces the amount of memops to those necessary to apply the Gauss transform on the matrix rows, yielding an average of 1.5 memory accesses per flop (see Line 15).

4.2. Givens rotations with reduced cost

Givens rotations [22] provide an appealing approach to triangularize a matrix when the number of elements to annihilate is small, as in the case of the reduction of the Hessenberg matrices. From the computational point of view (flops), applying a Givens rotation to a pair of rows is 3× more expensive than doing the same for a Gauss transform; see Table 1. The reason for this higher cost is that the application of a Givens rotation is equivalent to multiplying the two rows by a dense 2×2 matrix while, in the case of the Gauss transform, the 2×2 matrix that participates in the multiplication presents a unit lower triangular structure.

Nevertheless, in the case that we target, with small matrices, it is interesting to analyze whether a triangularization via Givens rotations can offer a more efficient alternative than one based on Gauss transforms. On the one hand, as discussed earlier, the flop ratio between Givens and Gauss is 3× in favor of Gauss, but the ratio in memops of Gauss versus Givens is (4/3)× only. On the other hand, given that we are only interested in computing the determinant of the matrix, not the triangular factor itself, we only need to update the bottom row during the application of a Givens rotation, reducing the existing difference in favor of Gauss transforms as explained next.

Consider the `reduceH_Gfast` routine in Listing 6 that reduces a Hessenberg matrix to upper triangular form via Givens rotations. Line 8 (commented out) shows the standard application of a Givens rotation to a pair of rows, which requires 6 flops (four multiplications and two additions) and 4 memory accesses (two loads and two stores) per row entry of the two vectors (that is, one element from each vector). Hereafter we assume that the parameters which define the Givens rotation, G, are maintained in the processor registers. Lines 10 and 11 illustrate the alternative which only updates the diagonal entry and the bottom row vector, reducing the cost to 3 flops and 3 memops per row entry of the two vectors. The key to this alternative is that, in order to obtain the determinant of the matrix being reduced, we only need to update the diagonal entries (as in Line 10 of the routine) and the subdiagonal row.

4.3. Reduced workspace

The routines exposed so far operate with "virtual" matrices. In an actual implementation, these data structures are mapped into physical memory and special care is needed to avoid overwriting any information that will be necessary later. While it is possible to operate on a duplicate of the data, copies also take time and consume memory. For the reduction to Hessenberg form, given that we only need the determinant of the resulting factor, we can avoid the use of excessive workspace (of the size of the matrix being factorized) by performing the operations on a couple of auxiliary row vectors. This is exposed in the `reduceH_wk_Gfast` routine for the reduction of a Hessenberg matrix to triangular form via Givens rotations in Listing 7. The same idea applies to the reduction via Gauss transforms.

The problem with the `reduceH_wk_Gfast` routine is that it incurs in a considerable amount of data copies as, for each iteration of the loop, there are two vectors being copied (Lines 7 and 12). Both copies can be avoided if we maintain an index that keeps track of where the diagonal/subdiagonal rows lie into a circular buffer that acts as workspace, and we obtain one of the copies as an implicit result of the update. This is illustrated in the `reduceH_rc_Gfast` routine in Listing 8, which also performs the reduction of a Hessenberg matrix to triangular form via Givens rotations. The resulting routine performs 3 flops and 3 memops per row entry of the pair of buffers, and does not modify the contents of the original matrix. Again, the same idea applies to the factorization via Gauss transforms with implicit pivoting. With this scheme, in addition, the permutations due to the application of pivoting do not imply any extra data movement.

4.4. Row re-utilization

A natural question that arises after the analysis in the previous discussion is whether it is possible to realize any additional savings by exploiting also the row relationships between the matrices factorized in the two outermost loops. There are two arguments against this idea, though. First, updating/downdating a factorization, when the transforms (either Gauss or Givens) are applied from the left, is considerably more involved [22]. Of course, one can compute the determinants by applying only the transforms from the right, and reversing the order of the loops indexed by r_1, r_2 with those indexed by c_1, c_2. However, the routines that result are simply transposed variants of those presented earlier, which operate by rows instead of by columns. Second, any savings that can be realized in the "row-wise factorizations" will impact only the cost of the factorizations for $A_{r1,r2\|-}$. As this accounts for a cost of $O(n^5)$ flops only (loops for r_1, r_2 iterating over a cost of $O(n^3)$ flops), the effect on the execution time can be expected to be negligible (except for very small values of n).

4.5. Multi-threaded parallelization

The three algorithms discussed earlier in this paper, `naive`, `base` and `opt`, all present two outer loops (indexed by r_1, r_2) that involve independent iterations. Inside these loops, there are two extra loops (indexed by c_1, c_2) and a series of computations. Given the relatively small values of the iteration count for all loops, the reduced dimensions of the matrices involved in the DL2M computations (n is around a couple of hundreds, at most), and the independence between the iterations of the outer loops, the best parallelization approach in principle consist in extracting concurrency from these outer loops. This can be realized via OpenMP by including a `#pragma omp parallel for` directive around the outermost loop.

In case n is not an integer multiple of the number of threads though, we can expect higher performance if we parallelize both outer loops, for example, by collapsing them into a single one. This can be done manually or via the OpenMP `collapse` clause. Unfortunately, the latter option requires that the ranges for both loops are completely independent, which is not the case since the iteration space of the loop indexed by r_2 depends on the index r_1. (Concretely, the second loop iterates for $r_2 = r_1 + 1$ to n.) A simple solution to tackle this is to force the second loop to iterate for $r_2 = 1$ to n, but do nothing in those iterations for which $r_2 < r_1 + 1$. This is conveniently combined with a dynamic distribution of the iteration space, via a `schedule(dynamic)` clause, given that the workload is then irregularly concentrated in some of the loop iterations. The resulting loop-parallel scheme is illustrated in Listing 9.

A different option to extract parallelism from the DL2M operations is to declare the computations inside the loop body as tasks,

```
function [H] = reduceH_ip(H)
% Input Hessenberg matrix overwritten with upper triangular factor from the factorization
[p, q] = size(H);

Hid = 1;
for k = 1 : min(p, q) − 1
  pivoting = abs(H(k + 1, k)) > abs(H(Hid, k))
  if (pivoting)
    His = Hid; Hid = k + 1;
  else
    His = k + 1;
  end
  pivot        = H(His, k)/H(Hid, k); % Diagonal entry --> Update determinant
  H(His, k)    = 0.0;
  H(His, k + 1 : q) = H(His, k + 1 : q) − pivot ∗ H(Hid, k + 1 : q);

  if (pivoting)
    Hid = His;
  else
    Hid = k + 1;
  end
end
```

Listing 5: Routine `reduceH_ip` for the reduction of a Hessenberg matrix to triangular form via Gauss transforms with implicit partial (row) pivoting. *Hid* and *His* contain indices (pointers) to the rows that store the diagonal and subdiagonal rows of H, respectively.

```
function [H] = reduceH_Gfast(H)
% Input Hessenberg matrix overwritten with upper triangular factor from the factorization
[p, q] = size(H);

for k = 1 : min(p, q)−1
  G = givens(H(k, k), H(k + 1, k));

  % H(k : k + 1, k : q) = G ∗ H(k : k + 1, k : q);        % Standard Givens rotation updating both rows

  H(k : k + 1, k)     = G ∗ H(k : k + 1, k);              % Diagonal entry --> Update determinant
  H(k + 1, k + 1 : q) = G(2, 1 : 2) ∗ H(k : k + 1, k + 1 : q); % Fast alternative updating only the bottom row
end
```

Listing 6: Routine `reduceH_Gfast` for the reduction of a Hessenberg matrix to triangular form via Givens rotations.

```
function [H] = reduceH_wk_Gfast(H)
% Input Hessenberg matrix, not modified as a result of the factorization
[p, q] = size(H);

Hd = H(1, 1 : q);
for k = 1 : min(p, q)−1
  Hs(k : q) = H(k + 1, k : q);
  G = givens(Hd(k), Hs(k));

  [Hd(k); Hs(k)] = G ∗ [Hd(k); Hs(k)];                  % Diagonal entry --> Update determinant
  Hs(k + 1 : q)  = G(2, 1 : 2) ∗ [Hd(k + 1 : q); Hs(k + 1 : q)]; % Fast alternative updating only the bottom row
  Hd(k + 1 : q)  = Hs(k + 1 : q);
end
```

Listing 7: Routine `reduceH_wk_Gfast` for the reduction of a Hessenberg matrix to triangular form via Givens rotations. At each iteration, *Hd* and *Hs* store copies of the diagonal and subdiagonal rows of H.

```
function [H] = reduceH_rc_Gfast(H)
% Input Hessenberg matrix, not modified as a result of the factorization
[p, q] = size(H);

Hid = 1;
His = 2;
v(Hid, 1 : q) = H(1, 1 : q);
for k = 1 : min(p, q)−1
  G = givens(v(Hid, k), H(k + 1, k));

  [v(Hid, k); v(His, k)] = G ∗ [v(Hid, k); H(k + 1, k)];          % Diagonal entry --> Update determinant
  v(His, k + 1 : q) = G(2, 1 : 2) ∗ [v(Hid, k + 1 : q); H(k + 1, k + 1 : q)]; % Fast alternative updating only the bottom row

  tmp = Hid; Hid = His; His = tmp; % Prepare the circular buffer for the next iteration
end
```

Listing 8: Routine `reduceH_rc_Gfast` for the reduction of a Hessenberg matrix to triangular form via Givens rotations. *Hid* and *His* contain indices (pointers) to the rows of the circular buffer v that store the diagonal and subdiagonal rows of H, respectively.

```
function [A] = naive/base/opt(A)
% Loop-parallel version
#pragma omp parallel for collapse(2) schedule(dynamic) private (r2)
for r1 = 1 : n - 1
    for r2 = 1 : n % Actual iterations a from r2 = r1 + 1 : n
        if (r2 >= r1 + 1)
           % Remaining operations in loop body of the corresponding algorithm
        else
           ; % Do nothing
```

Listing 9: Loop-parallel multi-threaded realization of the `naive`, `base` and `opt` algorithms for DL2M calculations.

and exploit task parallelism. This is shown in the task-parallel algorithm of Listing 10.

5. Numerical experiments with the DL2M algorithms

All the experiments in the paper were performed using IEEE double-precision arithmetic on a compute platform equipped with two Intel® Xeon® Gold 6126 processors (12 cores per socket, running at a nominal frequency of 2.60 GHz). The Linux governor for all cores was set to `performance`, unless otherwise explicitly stated. This has some implications in the evaluation of the parallel algorithms that we will discuss in detail [23]. Each routine was repeatedly run until a minimum execution time was reached. The results correspond to the average time taking into account the number of repetitions.

The algorithms and testing routines for the DL2M calculations were developed in C, processed with the Intel `icc` compiler available in Intel Composer XE 2019, and rely on calls to the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) interfaces [24,25] whenever possible. Moreover, the codes were compiled with the optimization flags `-O3 -axCORE-AVX512`; and vectorization was enforced for some of the loops by annotating them with the Intel `#pragma ivdep` directive. Optimized implementations of the BLAS and LAPACK were those provided by the sequential version of these libraries in the Intel MKL Composer XE 2019. As described later in this section, parallelism was extracted at a coarse-grain level, using Intel OpenMP enabled via the `-qopenmp` compiler flag and the appropriate OpenMP pragmas inserted into the codes; see Section 4. The same experiments were conducted on an AMD EPYC 7401 using the GNU `gcc` compiler and OpenBLAS. As the results on the AMD architecture were similar, they have been omitted in the following experimentation.

5.1. Reference DL2M algorithms

The starting point for our experimental evaluation are the `naive` algorithm in Listing 1 plus the `base` alternative with column re-utilization introduced in [21] and presented in Listing 2. For the `naive` algorithm, the LU factorizations (Line 8 there) are performed by invoking routine `dgetrf` from LAPACK. For the baseline version enhanced with column re-utilization, `base`, the initial factorization of the dense matrix (Line 6 in the algorithm in Listing 2) is also performed via a call to `dgetrf`. Furthermore, the reduction of the quasi-triangular blocks in the trailing submatrix, with one and two nonzero subdiagonals (Lines 9 and 10 in the same listing), are done via specialized routines that take into account the special structure of these blocks.

As described in Section 3, the computational costs of the `naive` and `base` algorithms are, respectively, $\frac{1}{6}n^7$ and $\frac{3}{16}n^6$ flops (see Section 3) so that, in principle, we should observe a reduction of the execution time that is close to one order of magnitude by performing the DL2M calculations using the latter.

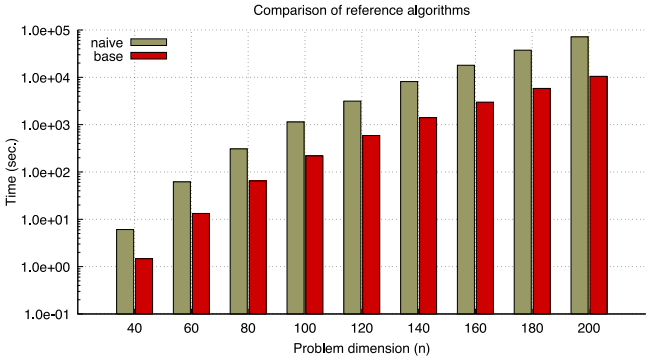

Fig. 3. Comparison of the reference algorithms: `naive` and `base`.

Fig. 3 compares the execution times of these two algorithms using a single core (sequential execution) of the target computer platform. These results show that, in practice, the reduction attained by the `base` algorithm is lower than expected. For example, for the problem dimensions $n = 40$ (small), 120 (intermediate), and 200 (large), the algorithm with column re-utilization delivers acceleration factors with respect to `naive` that are around 4.1, 5.3, 6.7, respectively. These values are notable, yet below the expected order of magnitude given by n. The reason for this is that the first algorithm encodes a compute-bound operation while the second one is memory-bound. As a result, the `naive` algorithm partially compensates its much higher computational cost by taking advantage of the significant gap between floating-point arithmetic performance and memory bandwidth on current computer architectures. In any case, due to its much higher execution time, experimentally demonstrated in Fig. 3, we drop the `naive` algorithm from the following discussions and experimentation.

5.2. Gauss-based DL2M algorithms

We next compare the algorithms for DL2M that employ Gauss transforms in the routines for the reduction to upper triangular form.

Our optimized solution for this process, algorithm `opt` in Listing 3, includes four variants, all with the same computation cost: $\frac{1}{24}n^6$ flops, which represents a theoretical reduction with respect to the `base` algorithm in a factor of 4.5; see Section 4. These variants compute the initial factorization of the dense matrix (Line 6 in the algorithm in Listing 3) using the same routine employed in the `base` algorithm (i.e., `dgetrf`). Moreover, the procedure that performs the reduction of the trailing submatrix, with a single nonzero subdiagonal, to triangular form (Line 8 in the algorithm in Listing 3) is the same used by the `base` algorithm, and basically implements the routine in Listing 4. The variants thus only differ in which routine is used to perform the second reduction (Line 10 in the `opt` algorithm in Listing 3). Concretely:

```
function [A] = naive/base/opt(A)
% Task-parallel version
#pragma omp parallel
#pragma omp single
for r1 = 1 : n − 1
    for r2 = r1 + 1 : n
        #pragma omp task firstprivate (r1, r2)
            % Remaining operations in loop body of the corresponding algorithm
```

Listing 10: Task-parallel multi-threaded realization of the `naive`, `base` and `opt` algorithms for DL2M calculations.

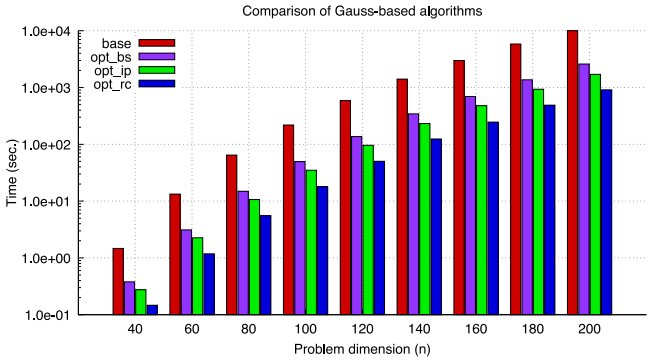

Fig. 4. Comparison of the Gauss-based algorithms.

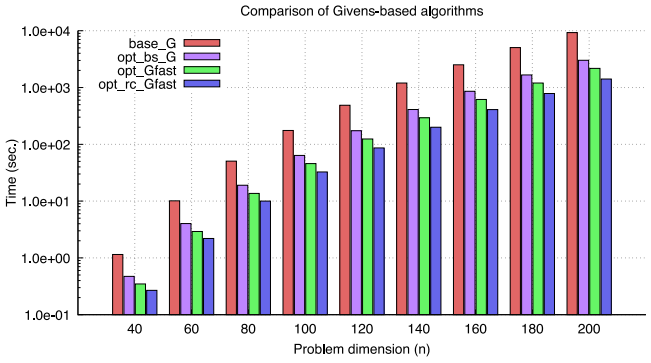

Fig. 5. Comparison of the Givens-based algorithms.

- `opt_bs` employs the baseline routine `reduceH_bs` in Listing 4;
- `opt_ip` includes implicit pivoting (as in routine `reduceH_ip` in Listing 5); and
- `opt_rc` combines implicit pivoting with the technique to reduce the use of workspace and data copies analogous to that in routine `reduceH_rc(_Gfast)` in Listing 8.

A fourth variant of the Gauss-based procedures will be introduced later, during the global comparison.

Fig. 4 reports the results of the experiment comparing the optimized versions and the `base` algorithm. A direct comparison of the `base` algorithm and the `opt_bs` variant shows a speed-up in favor of the latter that varies between 3.9 for the smallest problem ($n = 40$) and 4.1 for the largest one ($n = 200$), which is consistent with the acceleration that could be expected due to the theoretical cost reduction. The two additional enhancements, namely implicit pivoting and reduced data copies, further accelerate the factors to 10.04 ($n = 40$) and 11.52 ($n = 200$) in case we combine both as in the `opt_rc` variant. Finally, the `opt_ip` variant only integrates part of these enhancements and, as could be expected, delivers an acceleration that is in between those of `opt_bs` and `opt_rc`.

5.3. Givens-based DL2M routines

We have also developed realizations of all previous algorithms and routines (except for the `naive` case) that employ Givens rotations instead of Gauss transforms in those parts that are relevant for the execution time. As a result, we have obtained an implementation of the baseline algorithm in [21], named `base_G`, that combines the use of Gauss transforms for the initial transformation from dense to triangular with the use of Givens rotations for the specialized reductions from Hessenberg to triangular. In addition, we have obtained four variants of the improved solution:

- `opt_bs_G` replaces the use of Gauss transforms in the second reduction by Givens rotations (Line 10 in algorithm `opt` in Listing 3);
- `opt_Gfast` combines the former with the cheaper update (only bottom row) described in routine `reduceH_Gfast` in Listing 6; and
- `opt_rc_Gfast` is the most elaborate version, which combines the cheaper update with reduced workspace and data copies, as described in routine `reduceH_rc_Gfast` in Listing 8.

The introduction of the fourth variant is again delayed till the global comparison.

Fig. 5 compares the execution of the Givens variants and the realization of the `base` algorithm that employs Givens rotations for the reduction of the upper quasi-triangular matrices. The speed-up in this case is lower than the acceleration observed when using Gauss transforms. Concretely, the `opt_bs_G` variant delivers a speed-up that varies between 2.41 and 3.05, for the smallest and the largest problem size respectively ($n = 40$ and 200). Moreover, these accelerations grow to factors that range between 5.48 and 7.43 when introducing the cheaper update and reduced copies, as in the `reduceH_rc_Gfast` variant.

5.4. Best DL2M options

The experiment in this subsection compares the best variants of the `opt` algorithm based on Gauss and Givens rotations; that is, `opt_rc` and `opt_rc_Gfast`. In addition, we include two new variants that store the matrices by rows (as in C) instead of the standard column-wise storage that is dictated by the use of BLAS/LAPACK (and Fortran). Note that, as the goal of the algorithm is to compute the determinants, the fact that we operate on a matrix or on its transposed (which is analogous to the data being stored by rows or columns) is indifferent.

Fig. 6 offers the results from this comparison, showing the execution time for the best Gauss- and Givens-based variants as well as with their row-wise counterparts. Table 2 reports the speed-ups of these variants with respect to the `base` algorithm. The Gauss-based variant with row-wise storage, `opt_rc_row`, shows impressive acceleration factors, between 12.11 (lowest, for $n = 40$) and 19.21 (highest, for $n = 160$) with respect to `base`.

Table 2
Execution time (in s) of the best routines and speed-ups with respect to the base algorithm.

n	base	opt_rc		opt_rc_row		opt_rc_Gfast		opt_rc_Gfast_row	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up	Time	Speed-up
40	1.47E+00	1.46E−01	10.04	1.21E−01	12.11	2.68E−01	5.48	2.47E−01	5.96
60	1.33E+01	1.18E+00	11.27	9.29E−01	14.34	2.20E+00	6.06	1.89E+00	7.06
80	6.47E+01	5.54E+00	11.68	4.36E+00	14.84	1.00E+01	6.45	8.31E+00	7.78
100	2.19E+02	1.79E+01	12.22	1.24E+01	17.69	3.26E+01	6.71	2.54E+01	8.61
120	5.90E+02	5.03E+01	11.73	3.50E+01	16.88	8.63E+01	6.83	6.51E+01	9.06
140	1.41E+03	1.24E+02	11.35	8.73E+01	16.13	2.00E+02	7.03	1.44E+02	9.77
160	2.98E+03	2.46E+02	12.14	1.55E+02	19.21	4.09E+02	7.30	2.87E+02	10.39
180	5.83E+03	4.88E+02	11.94	3.11E+02	18.73	7.86E+02	7.41	5.45E+02	10.69
200	1.05E+04	9.11E+02	11.52	5.70E+02	18.41	1.41E+03	7.43	9.51E+02	11.04

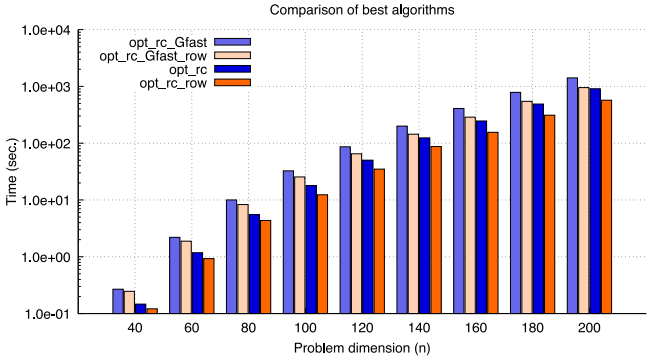

Fig. 6. Comparison of the best algorithms.

Again, the speed-ups for the Givens-based routines are lower than those observed for their Gauss counterparts, yet they are still remarkable.

5.5. Multi-threaded parallelization

In the following experiment, we evaluate the scalability of our parallelization approaches. For this evaluation, we consider the two best Gauss-based implementations determined in the previous experiment, `opt_rc` and `opt_rc_row`. For reference, we also include the parallel implementation of the `base` algorithm. The parallelization schemes run on 4, 8, 16, 20 and 24 physical cores of the target platform, using one thread per core, and exploiting either loop- or task-parallelism, as discussed in the final part of Section 4. However, given that both strategies deliver similar values, we only report next the results obtained with the former. In the actual implementation, we restrict thread migration by fixing the execution of each thread to a particular core via Linux's `sched_setaffinity` utility.

The initial experiment compares the parallel performance of `opt_rc` and `opt_rc_row` (and `base`) using all resources of both processor sockets (that is, up to 24 cores/threads). Fig. 7 reports the results of this evaluation for all problem dimensions and number of cores. Table 3 quantifies the differences for three specific problems, of small, moderate and large dimensions, exposing an interesting effect that depends on the number of cores employed in the evaluation. In particular, the plots in the figure show that the variant with row storage always outperforms its column-wise counterpart, and they both deliver execution times which are much lower than those of `base`, by about an order of magnitude. Interestingly, the numbers in the table show that the advantage of `opt_rc_row` over `opt_rc` is consistently higher when using 4 cores. We monitored the system during these experiments and discovered that, for these particular variants of the optimized algorithm, the Intel Xeon cores run at

Table 3
Speed-ups of the `opt_rc_row` variant with respect to `opt_rc`.

n	#Cores					
	4	8	12	16	20	24
40	1.44	1.32	1.31	1.33	1.32	1.31
120	1.50	1.30	1.30	1.35	1.30	1.30
200	1.64	1.42	1.43	1.49	1.42	1.43

2.6 GHz for the `opt_rc` instance and any number of cores. In contrast, the `opt_rc_row` instance operates at 2.6 GHz when running on 4 cores, but reduces this frequency to 2.3 GHz for 8, 12, . . . , 24 cores. This is due to the higher arithmetic intensity of the row-wise oriented variant, which makes a better use of the cache memories (due to the row-wise access to the matrix rows) but at the same time exerts a higher pressure on the arithmetic units, resulting in the Linux governor automatically decreasing the operating frequency. The frequency difference explains almost perfectly the reduction of the speed-up in a factor $f = 2.6/2.3$ that is observed when using more than 4 cores with respect to the results for 4 cores. For example, for the largest problem size and 24 cores, the speed-up is 1.43 which, multiplied by f is basically equivalent to the speed-up observed with 4 cores: $1.43 \cdot f = 1.62 \approx 1.64$.

The next experiment analyzes the scalability of the algorithms, further exposing the aggressive utilization of the power modes (and associated frequencies) featured by the Intel® Xeon® Gold 6126 processor [23] by the Linux governor. Fig. 8 reports the speed-up of the two variants, `opt_rc` and `opt_rc_row`, with respect to the sequential execution of the corresponding algorithm. The results in those plots may seem a bit disappointing at first. For an algorithm that is supposedly embarrassingly parallel, except for the execution with 4 cores, the speed-ups are significantly lower than could be expected for `opt_rc` and this is even more the case for `opt_rc_row`. This is due to the operation of the processor at much higher frequencies in the sequential execution, of up to 3.5 GHz, than in the parallel runs. The Linux governor applies Dynamic Voltage Frequency Scaling (DVFS) autonomously, by means of the Hardware-Controlled Performance States (HWP) (see section 14 in [26]), to fulfill power and thermal budget constraints [27]. Even when the higher arithmetic intensity of the row-wise version could potentially get more benefits from increased operating frequencies, the thermal design power (TDP) of the chip cannot be exceeded [28]. That justifies our observation that, as we increase the number of cores, the frequency is automatically decreased for the row-wise version and, consequently, scalability suffers (Fig. 8 right).

Our final experiment illustrates the parallel scalability of the algorithm by emulating a more conventional processor architecture, without an aggressive frequency scaling scheme. For this purpose, we maintain the Linux governor in `performance` mode, but we fix the maximum frequency for all the cores to 2.3 GHz.

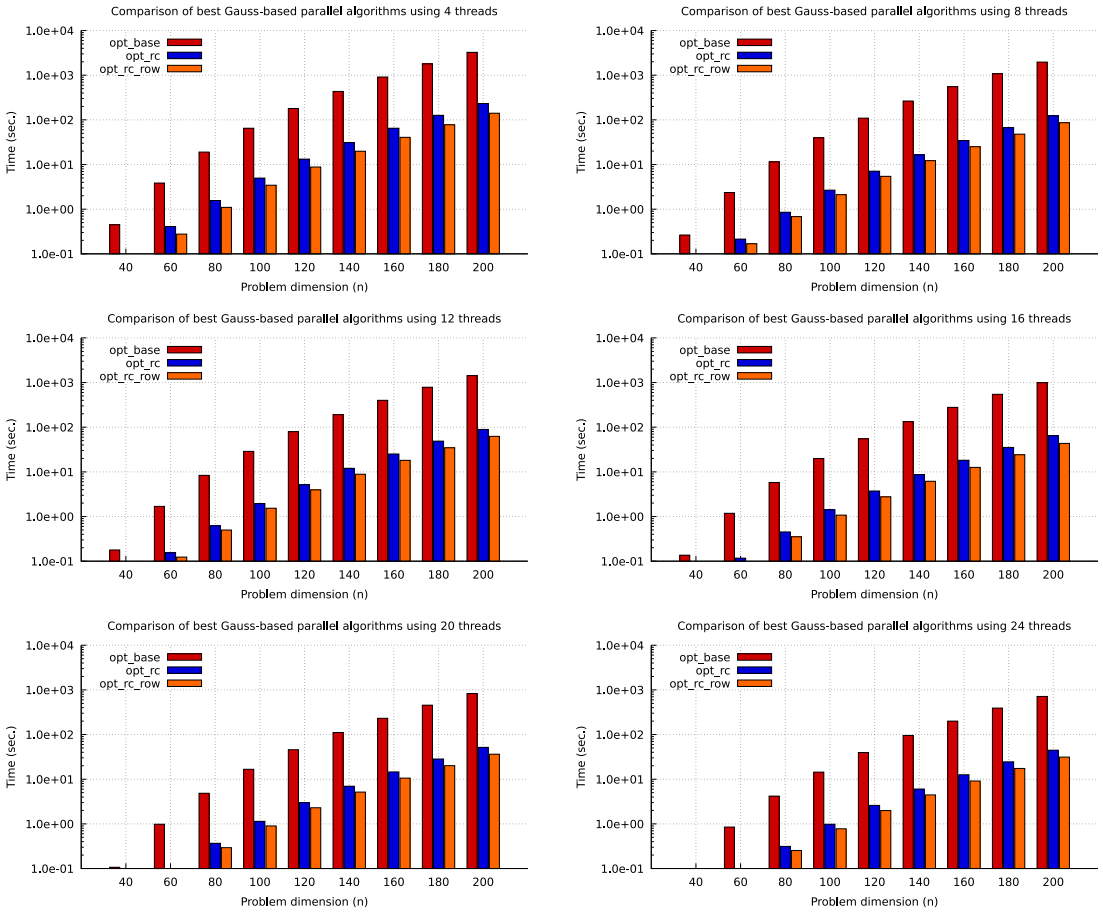

Fig. 7. Comparison of the best Gauss-based algorithms executed in parallel.

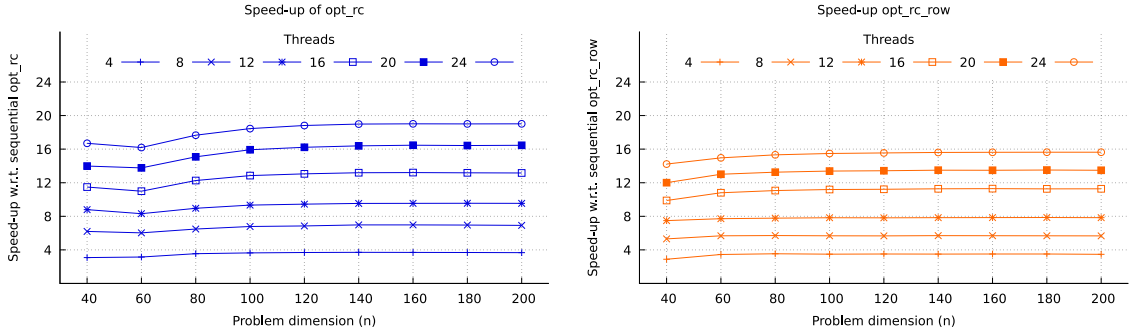

Fig. 8. Speed-up of the best Gauss-based algorithms executed in parallel.

This is below the specification of the nominal frequency for this processor (2.6 GHz) yet, as part of an independent experiment, we determined that it is the highest frequency in which we can execute all the variants of our algorithms, without the Linux governor lowering the value independently of the number of active cores.

Fig. 9 reports the speed-ups of the two variants, with respect to the sequential execution of the corresponding algorithm, when running in this frequency-constrained operating mode. The plots in the figure show that the speed-ups are quite aligned with the theoretical values for an embarrassingly parallel algorithm. The superlinear speed-ups observed for some of the cases can be explained by the limited control we can exert on fluctuations of the frequency setting below the fixed upper bound.

6. Application in full overlap matrix calculations for molecular systems

In this section we present several experiments to illustrate the impact that the enhancements to the DL2M algorithms for the minor computations have on the calculations of the full wave-function overlap matrix using Eq. (2) for a series of molecular systems. In this evaluation, `OL2M` refers to the algorithm from [14], which employs the `base` approach to compute the determinants of the matrix of overlaps; and `OL2M-Gauss` and `OL2M-Givens` denote two algorithmic variants that compute the determinants via algorithms `opt_rc_row` and `opt_rc_Gfast_row`, respectively. The rest of the MEWF overlap code is the same in all three cases.

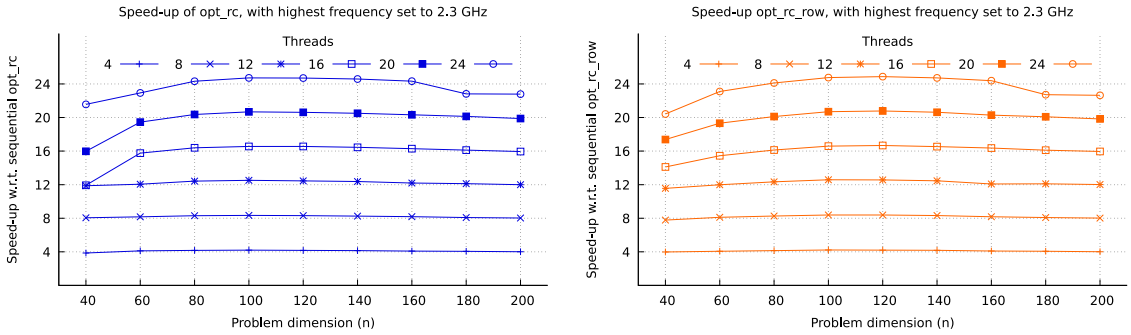

Fig. 9. Speed-up of the best Gauss-based algorithms executed in parallel, with highest frequency constrained to 2.3 GHz.

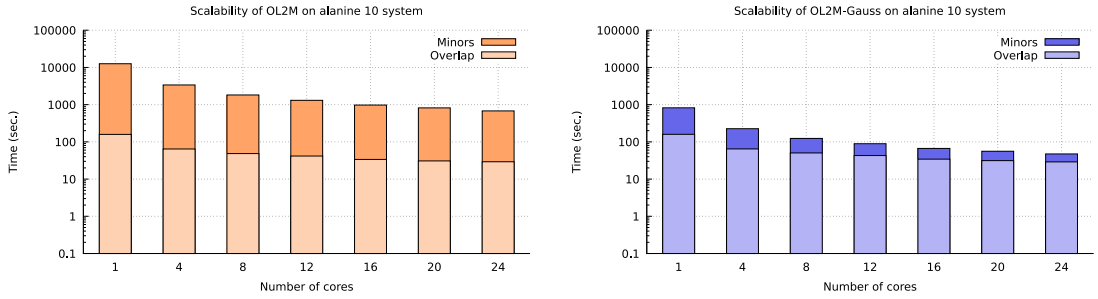

Fig. 10. Total execution time (log scale) of the `OL2M` (left) and `OL2M-Gauss` (right). Time to compute the determinants of the minors (dark colors) compared to time spent on the rest of the code (light colors). The tests are for Alanine 10 system (195 occupied orbitals). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

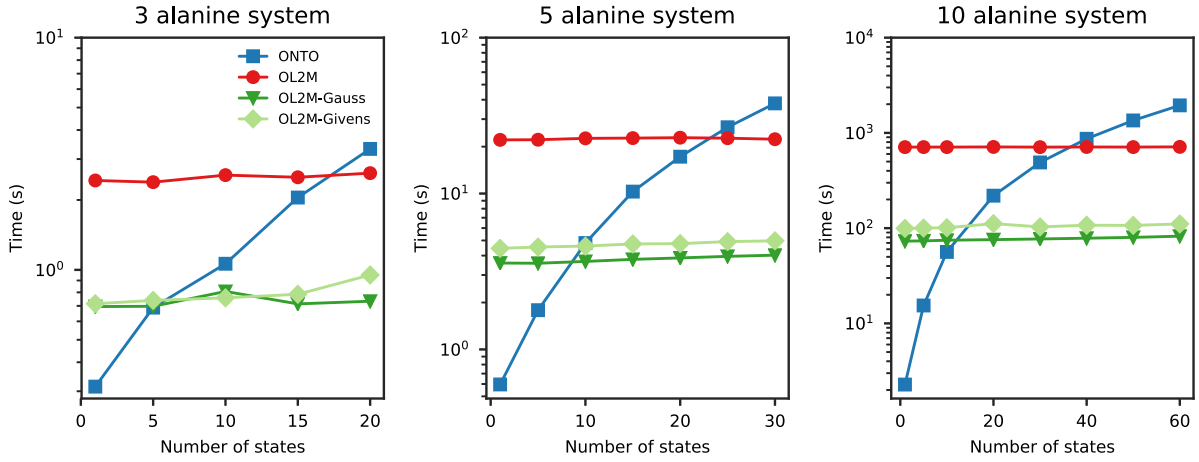

Fig. 11. Total execution time (log scale) of the `ONTO`, `OL2M`, `OL2M-Gauss` and `OL2M-Givens` algorithms for computing the full overlap matrix consisting of different numbers of excited electronic states for the three model polyalanine systems containing 3, 5 and 10 alanine residues.

For the test system we employ two model peptide systems. The first is a series of polyalanine chains consisting of three, five, or ten alanine residues. These correspond, respectively, to 33, 53, or 103 atoms; and 62, 100, or 195 occupied orbitals. The basis set used was def2-SV(P) [29]. For this system, this corresponds to 196, 318 and 623 virtual orbitals. Up to 60 excited states were considered for the largest system.

The first experiment, performed on the alanine 10 system, evaluates the parallel scalability of the algorithms. The parameters for this particular polyalanine system are set to 195 occupied orbitals, 608 virtual orbitals, and 20 excited states. The vertical bars in Fig. 10 present the total time required to compute the overlap matrix of wave functions, decomposed into time to obtain determinants of the minors (labeled as "Minors") and the time to calculate the coefficients of the overlap matrix (labeled as "Overlap"). The latter part of the code is out-of-scope for this research but is an integral part of wave-function overlap code and, therefore, its cost included in the tests. We observe that, regardless of the number of cores, more than 94% of the total execution time for `OL2M` is dedicated to computing the determinants of the minors. (Note the logarithmic scale in the y-axis.) In comparison, for `OL2M-Gauss` the ratio of computing the minors to the total time varies between 74% when using a single core to 54% using all 20 cores. Note that the execution time for the `Overlap` component is the same for both algorithmic variants since it does not depend on the algorithm used to calculate the determinants. The figure also demonstrates that the scalability of the entire algorithm for computing the wave-function overlaps is significantly better for the `OL2M-Gauss`. The reason is that, in this variant, computing the determinants is no longer a prevailing part, especially in the case when more cores are used.

We next compare the improved OL2M algorithms with the ONTO algorithm, which was previously shown to significantly outperform OL2M for large systems due to its better scaling

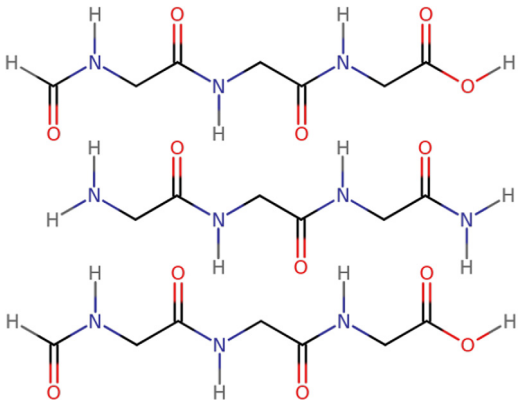

Fig. 12. Structure of the amyloid model system containing three short β-strands cross-linked with hydrogen bonds.

properties with system size. The drawback of the ONTO algorithm is that each overlap matrix element has to be calculated independently, whereas the most expensive part of the OL2M algorithm is shared for the entire overlap matrix. In Fig. 11, we compare the total execution time for the full overlap matrix calculation of the polyalanine systems while varying the number of states. Although the ONTO algorithm is faster for a single state, the improved OL2M-based algorithms start outperforming ONTO when the ratio between the number of occupied orbitals and the number of states is approximately 13:1 for the OL2M-Gauss algorithm. For the three studied systems this occurs, respectively, at 6, 9 and 13 states, which is lower than the number of states that one would typically include for systems of this dimension.

The second model system, recently introduced to study the photophysics of amyloid fibrils, was built to reproduce the hydrogen bonding pattern of parallel β-strands characteristic of amyloid proteins and consists of 77 atoms; Fig. 12. This system comprises a group of states of $n\pi^*$ character (each localized at a different amide group), all found in a narrow energy window, and a second group of states of $\pi\pi^*$ character at slightly higher energies. In order to capture the entire manifold of $n\pi^*$ and part of the $\pi\pi^*$ manifold, a total of 16 excited states were included in the calculation. Short nonadiabatic dynamics simulations were performed using ADC(2) [17] as the electronic structure method, and employing the cc-pVDZ basis set [30]. The core orbitals were frozen so that the calculation consisted of 121 occupied and 608 virtual orbitals. The execution times for the overlap calculation for this system are given in Table 4. `L1minors` is the stage that computes the determinants of the first level Laplace minors; the `RS/SR` stage computes the overlap of the bra/ket reference determinants with singly excited ket/bra determinants, respectively; `L2minors` is our new algorithm for the level-2 determinants; and `SS` denotes the part of the code where the final overlap matrix is computed. While the calculations of the minors was by far the dominant contribution to the overlap matrix calculation (`Total WF`) in the `OL2M` code, with the optimized algorithms this is now faster than the step of reconstructing the overlaps using the calculated minors. We can observe that the turn-point between `ONTO` and `OL2M-Gauss/Givens` is achieved for a slightly smaller ratio between the number of occupied orbitals and the number of states than for polyalanine systems (see Fig. 11) and is approximately 10:1.

7. Conclusion

We have presented accelerated algorithms for calculating the determinants of the level-2 minors during calculations of CIS type wave function overlaps for nonadiabatic dynamics simulations. Two new algorithms were developed based on Gauss transformations and Givens rotations, both an order of magnitude faster than the best previous column-wise implementation of level-2 minors calculations (a variant named `base` in the paper). The significant time savings are achieved by decreasing the number of operations and memory requirements, by applying various optimization techniques in both algorithmic variants, reducing the workspace required, partial pivoting with implicit permutations in Gauss transformations and reducing the cost of Givens rotations, resulting in significant time savings. The experimental results pointed that the variant using Gauss rotation achieves slightly better performance over the variant based on Givens rotations.

Table 4
Execution times (in s) of the OL2M, OL2M-Gauss, OL2M-Givens and ONTO algorithms for the amyloid model system.

Compute stage	OL2M	OL2M-Gauss	OL2M-Givens	ONTO
L1minors	2.07	2.10	2.10	–
RS/SR	0.01	0.01	0.01	–
L2minors	51.94	4.09	5.94	–
SS	3.50	4.12	3.38	–
Total WF	57.44	10.35	11.46	21.48
Total	59.19	11.97	13.17	23.13

The experimental evaluation considers full overlap calculations for systems between 33 and 103 atoms. For ADC(2), these are very large systems, with electronic structure calculations taking an average of 18 h on 20 cores for the 77 atom amyloid model system. This means that this type of overlap calculation can be performed for ADC(2) excited states at virtually no cost compared with the electronic structure calculation. However, the same calculation using TDDFT can be performed in a few minutes (depending on the implementation and choice of functional). This makes the overlap calculation a non-negligible part of the overall cost of a dynamics simulation. With the current implementation of the OL2M algorithm, systems with up to 100 atoms/200 occupied orbitals can be treated at a lower cost than that observed when using the TDDFT for electronic structure calculations. For larger systems, the previously developed ONTO algorithm (which is more amenable to approximations) can be applied along with a threshold for the wave function to calculate approximate overlaps for arbitrarily large systems at low cost.

The improved algorithm for calculating the minors also clears the path to perform exact overlap calculations for more complex wave function types. The overlaps between wave functions of CISD type require the calculation of all $\left\langle \Theta^{i,j}_{a,b} \middle| \Theta'^{k,l}_{c,d} \right\rangle$ elements for pairs of SDs with two occupied orbitals replaced by virtual orbitals. A naive implementation of this calculation is not feasible even for very small systems. The present algorithm can also be extended to tackle MR-CIS and MR-CISD wave functions by combining it with the algorithm of Plasser et al. [13] for the reuse of intermediates in MR expansions, where a large number of simultaneous excitations of both spin α and β are often present.

The new Gauss and Givens based versions of the level-2 minors calculations are integrated in the existing code for computing the overlaps for CIS type wave functions (CIS Overlap) and are available on the GitHub [31].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Enrique S. Quintana-Ortí was supported by project TIN2017-82972-R, Pedro Alonso-Jordá by project TIN2017-89314-P-AR, and José R. Herrero by project TIN2015-65316-P, all of the MINECO and FEDER. The latter was also funded by the Generalitat de Catalunya (contract 2017-SGR-1414). Marin Sapunar and Davor Davidović were supported by Croatian Science Foundation under grant HRZZ IP-2016-06-1142.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2020.107521.

References

[1] B.F.E. Curchod, T.J. Martínez, Chem. Rev. 118 (7) (2018) 3305–3336, http://dx.doi.org/10.1021/acs.chemrev.7b00423.
[2] R. Crespo-Otero, M. Barbatti, Chem. Rev. 118 (15) (2018) 7026–7068, http://dx.doi.org/10.1021/acs.chemrev.7b00577.
[3] S. Hammes-Schiffer, J.C. Tully, J. Chem. Phys. 101 (6) (1994) 4657–4667, http://dx.doi.org/10.1063/1.467455.
[4] J.C. Tully, J. Chem. Phys. 93 (2) (1990) 1061–1071, http://dx.doi.org/10.1063/1.459170.
[5] G. Granucci, M. Persico, A. Toniolo, J. Chem. Phys. 114 (24) (2001) 10608–10615, http://dx.doi.org/10.1063/1.1376633.
[6] S. Fernandez-Alberti, A.E. Roitberg, T. Nelson, S. Tretiak, J. Chem. Phys. 137 (1) (2012) 014512, http://dx.doi.org/10.1063/1.4732536.
[7] G.A. Meek, B.G. Levine, J. Phys. Chem. Lett. 5 (13) (2014) 2351–2356, http://dx.doi.org/10.1021/jz5009449.
[8] L. Wang, O.V. Prezhdo, J. Phys. Chem. Lett. 5 (4) (2014) 713–719, http://dx.doi.org/10.1021/jz500025c.
[9] J. Qiu, X. Bai, L. Wang, J. Phys. Chem. Lett. 9 (15) (2018) 4319–4325, http://dx.doi.org/10.1021/acs.jpclett.8b01902.
[10] Z. Zhou, Z. Jin, T. Qiu, A.M. Rappe, J.E. Subotnik, J. Chem. Theory Comput. (2019) http://dx.doi.org/10.1021/acs.jctc.9b00952, arXiv:1909.11157, acs.jctc.9b00952.
[11] P.O. Löwdin, Phys. Rev. 97 (6) (1955) 1474–1489, http://dx.doi.org/10.1103/PhysRev.97.1474.
[12] J. Pittner, H. Lischka, M. Barbatti, Chem. Phys. 356 (1–3) (2009) 147–152, http://dx.doi.org/10.1016/j.chemphys.2008.10.013.
[13] F. Plasser, M. Ruckenbauer, S. Mai, M. Oppel, P. Marquetand, L. González, J. Chem. Theory Comput. 12 (3) (2016) 1207–1219.
[14] M. Sapunar, T. Piteša, D. Davidović, N. Došlić, J. Chem. Theory Comput. 15 (6) (2019) 3461–3469.
[15] M.E. Casida, in: D.P. Chong (Ed.), Recent Advances in Density Functional Methods, World Scientific, Singapore, 1995, pp. 155–192, http://dx.doi.org/10.1142/9789812830586_0005.
[16] E. Tapavicza, I. Tavernelli, U. Rothlisberger, Phys. Rev. Lett. 98 (2) (2007) 023001, http://dx.doi.org/10.1103/PhysRevLett.98.023001.
[17] J. Schirmer, Phys. Rev. A 26 (5) (1982) 2395–2416, http://dx.doi.org/10.1103/PhysRevA.26.2395.
[18] F. Plasser, R. Crespo-Otero, M. Pederzoli, J. Pittner, H. Lischka, M. Barbatti, J. Chem. Theory Comput. 10 (4) (2014) 1395–1405, http://dx.doi.org/10.1021/ct4011079.
[19] C.M. Isborn, N. Luehr, I.S. Ufimtsev, T.J. Martínez, J. Chem. Theory Comput. 7 (6) (2011) 1814–1823, http://dx.doi.org/10.1021/ct200030k, PMID: 21687784, arXiv:https://doi.org/10.1021/ct200030.
[20] L.D. Peters, J. Kussmann, C. Ochsenfeld, J. Chem. Theory Comput. 15 (12) (2019) 6647–6659, http://dx.doi.org/10.1021/acs.jctc.9b00859.
[21] D. Davidović, E.S. Quintana-Ortí, in: R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski (Eds.), Parallel Processing and Applied Mathematics, PPAM 2019, in: Lecture Notes in Computer Science, vol. 12043, Springer, Cham, 2020 http://dx.doi.org/10.1007/978-3-030-43229-4_2.
[22] G.H. Golub, C.F.V. Loan, Matrix Computations, third ed., The Johns Hopkins University Press, Baltimore, 1996.
[23] Intel, Intel® Xeon® Processor Scalable Family. Specification udpate, Tech. Rep., 2019, https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf.
[24] J.J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, ACM Trans. Math. Software 16 (1) (1990) 1–17.
[25] E. Anderson, Z. Bai, L.S. Blackford, J. Demmel, J.J. Dongarra, J.D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, D.C. Sorensen, LAPACK Users' Guide, third ed., SIAM, 1999.
[26] Intel, Intel 64 and IA-32 architectures software developer's manual volume 3A, 3B, and 3C: System programming guide, 2016, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.
[27] C. Bruns, S. Touati, Empirical Study of Amdahl's Law on Multicore Processors, Research Report RR-9311, INRIA Sophia-Antipolis Méditerranée ; Université Côte d'Azur, CNRS, I3S, France, 2019, URL https://hal.inria.fr/hal-02404346.
[28] J.W. Choi, D. Bedard, R. Fowler, R. Vuduc, 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 661–672, http://dx.doi.org/10.1109/IPDPS.2013.77.
[29] A. Schäfer, H. Horn, R. Ahlrichs, J. Chem. Phys. 97 (4) (1992) 2571–2577, http://dx.doi.org/10.1063/1.463096, arXiv:1101.3067.
[30] T.H. Dunning, J. Chem. Phys. 90 (2) (1989) 1007–1023, http://dx.doi.org/10.1063/1.456153.
[31] M. Sapunar, Natural transition orbitals for CIS type wave functions, 2020, URL https://github.com/marin-sapunar/cis_nto. (Accessed 20 May 2020).