Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–15

DOI: 10.1177/1094342017727060
journals.sagepub.com/home/hpc
SAGE

Optimized parallel simulations of analytic bond-order potentials on hybrid shared/distributed memory with MPI and OpenMP

Carlos Teijeiro[1], Thomas Hammerschmidt[1], Ralf Drautz[1] and Godehard Sutmann[1,2]

Abstract

Analytic bond-order potentials (BOPs) allow to obtain a highly accurate description of interatomic interactions at a reasonable computational cost. However, for simulations with very large systems, the high memory demands require the use of a parallel implementation, which at the same time also optimizes the use of computational resources. The calculations of analytic BOPs are performed for a restricted volume around every atom and therefore have shown to be well suited for a message passing interface (MPI)-based parallelization based on a domain decomposition scheme, in which one process manages one big domain using the entire memory of a compute node. On the basis of this approach, the present work focuses on the analysis and enhancement of its performance on shared memory by using OpenMP threads on each MPI process, in order to use many cores per node to speed up computations and minimize memory bottlenecks. Different algorithms are described and their corresponding performance results are presented, showing significant performance gains for highly parallel systems with hybrid MPI/OpenMP simulations up to several thousands of threads.

Keywords

Bond-order potentials, large-scale simulations, hybrid implementation, MPI, OpenMP

1. Introduction

Atomistic simulations on different levels of approximation have contributed a tremendous insight in a number of disciplines, such as biology, chemistry, physics and materials science (Frauenheim et al., 2002; Konstanze et al., 2016; Qi et al., 2014). The key component in these simulations is the interatomic potential, which determines the potential energy of each atom in the system according to their environment. Therefore, the development of accurate interatomic potentials is a very important area of research nowadays, and many different potentials have been obtained for a large amount of molecules and systems (Mishin, 2005; Tersoff, 1989; The Interatomic Potentials Repository Project, 2017).

Typical molecular dynamics codes in materials science implement interatomic interactions using pair potentials, which allow very large system sizes on parallel simulations (Diemand et al., 2013; Nakano et al., 2008; Pei et al., 2007) but do not provide sufficient information about several materials' properties, such as magnetic or electric behaviour. When accuracy plays an important role, one of the most popular simulation approaches derived from quantum mechanics is density functional theory (DFT; Jones and Gunnarsson 1989). However, DFT calculations require extensive computation that usually restricts the applicability to systems with around 1000 atoms. Considering the limitations of empirical/pair potentials on one end and DFT calculations on the other end, the usage of atomistic simulations typically requires a trade-off between accuracy and speed of the computations.

The analytic bond-order potentials (BOPs) are an intermediate method that are based on the tight-binding

[1]Interdisciplinary Centre for Advanced Materials Simulation (ICAMS), Ruhr-University Bochum, Bochum, Germany
[2]Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation (IAS), Forschungszentrum Jülich, Jülich, Germany

Corresponding author:
Carlos Teijeiro, Interdisciplinary Centre for Advanced Materials Simulation (ICAMS), Ruhr-University Bochum, Universitätstr. 150, D-44801 Bochum, Germany.
Email: carlos.teijeirobarjas@rub.de

bond model (Sutton et al., 1988) and provide a reliable but computationally efficient description of the interatomic interactions. Analytic BOPs are based on the computation of the local density of states $n_{i\alpha}$ for an atom i and valence orbital α as a moments expansion (Drautz and Pettifor, 2006, 2011; Drautz et al., 2015). The binding energy E of the system can, therefore, be defined as an integral of the local density of states, in which the moment $\mu_{i\alpha}$ for a given expansion N is expressed as

$$\begin{aligned}\mu_{i\alpha}^{(N)} &= \int E^N n_{i\alpha}(E)dE \\ &= \sum_{j\beta k\gamma \ldots q\kappa} \langle i\alpha|\hat{H}|j\beta\rangle\langle j\beta|\hat{H}|k\gamma\rangle \ldots \langle q\kappa|\hat{H}|i\alpha\rangle,\end{aligned} \tag{1}$$

where each term $\hat{H}$ in the product (e.g. $\langle i\alpha|\hat{H}|j\beta\rangle$) represents the interaction between two neighbour atoms (e.g. i and j), including the information about their corresponding valence orbitals (e.g. α and β), as a small Hamiltonian matrix $H_{i\alpha j\beta}$ (where $H_{i\alpha j\beta} \in \mathbb{R}^{m\times n}$ with variable m and n values depending on the number of valence orbitals of atoms i and j, respectively). Therefore, these terms build a chain of multiplications of pairwise bond information (defined as self-returning paths from an origin atom i) in a volume around i, such that they provide a contribution to the final energy associated with i. Moreover, a path between i and one of its neighbour atoms j also represents a partial calculation of the forces associated with that bond, and they are calculated simultaneously as a subset of the previous paths. The value of the moment expansion determines the maximum length of the given self-returning paths and thus also determines the sampled volume around every atom. The final energy and force values for a given atom i are obtained by summing up all possible combinations of paths for all given lengths from $N = 1$ to the selected moment expansion $N = m$, so that every path contribution is obtained by multiplying the Hamiltonian matrices $H_{i\alpha j\beta}$ associated with the bonds that define the given path. The larger the moment expansion m is, the longer the computed self-returning paths are, which implies a higher number of pairwise Hamiltonian matrix multiplications per path for every atom i in equation (1) (thus more computational demand) but also higher accuracy in the approximation to the results of the tight-binding solution. More detailed information on the analytic BOPs is available in, for example, Hammerschmidt et al. (2009) and Teijeiro et al. (2016b).

The computation of self-returning paths for analytic BOPs can be very demanding in terms of memory, since the number of matrix multiplications for path calculations increases as $\mathcal{O}(m^{4.5})$ as function of the moment expansion m (Teijeiro et al., 2016b). Consequently, a parallel implementation is necessary to meet the memory requirements of the code and to obtain sufficient efficiency. As the use of a limited moment expansion naturally restricts the possible range of influence of an atom, the problem is of local nature, and the methods and algorithms known from short-range interacting systems apply. Therefore, a possible parallelization can be conveniently based on domain decomposition. According to this, previous works (Teijeiro et al., 2015, 2016a) have presented and analysed several parallel implementations for BOPs with Message Passing Interface (MPI) Forum (2017). However, a pure MPI approach can cause a non-optimal use of computational resources for the case of highly parallel simulations, in which the full memory of a computing node is exhausted.

As a result, this work focuses on the combination of the existing MPI parallelization with the use of The OpenMP® API specification for parallel programming (OpenMP; 2017) threads for every MPI process, in order to maximize the use of computational resources for CPU-based system architectures. In this aspect, the main contributions are (1) the definition of thread-safe computations of paths inside a domain, identifying the core data dependencies, (2) the comparison of different OpenMP algorithms, from simple to more elaborate approaches, in order to obtain the highest speed-up for any thread number and (3) the performance analysis of very large system sizes on highly parallel supercomputers, so that the full potential of the many-core architectures is exploited.

The rest of the article is organized as follows. Section 2 summarizes the previous parallel solutions for BOPs, discussing their limitations and introducing the main motivation for the present work. Section 3 describes the data dependencies in simulations based on analytic BOPs and the OpenMP solutions that can be implemented to solve them. Section 4 shows the performance comparison of different algorithms, focusing on the analysis of scalability of the best solution for large simulations. Finally, Section 5 presents the conclusions from the work.

2. Previous work and motivation

The code used for the simulation of analytic BOPs is BOPfox (Hammerschmidt et al., 2017), which is written in Fortran. The MPI parallelization of BOPfox has been implemented following a domain decomposition scheme, in which the system is subdivided into connected rectangular spatial regions (domains; Teijeiro et al., 2015). The domains together with all atoms are managed by an MPI process. Additionally, every domain defines an overlap zone with its neighbour domains, so that every process stores this redundant information in order to perform a local computation of paths. The analysis of different algorithms has proven the possibility of avoiding redundant path

computations and minimizing inter-process communications at the same time (Teijeiro et al., 2016a). However, this pure MPI approach exhibits some limitations in flexibility for an efficient exploitation of many-core systems.

The choice of resource allocation has a strong influence on the efficiency at runtime. For example, given a physical problem size, the use of a large number of processors reduces the size of the individual domains. Nevertheless, since the range of interactions is prescribed by the moment expansion, the width of the overlap zone around each domain is fixed. Therefore, the effectiveness of the parallelization will depend on the ratio between the width of the overlap zone and the size of the domain, which is directly related to the ratio between communications and computations. Still the memory demand of the simulation becomes significantly larger in the cases where at least one of these factors is required: (1) high accuracy and (2) magnetic information. The accuracy of the system is related to the number of moments m, so that a large moment expansion implies gathering information from a larger volume around every atom: the increase of m makes the allocated memory space grow in a polynomial way (Teijeiro et al., 2016b). Additionally, the use of magnetic information increases the number of computations because of the consideration of different spin values (e.g. for collinear magnetism, Ford et al., 2014, the number of computations and the memory allocated to store moment contributions for each bond is doubled because of the inclusion of spin-up and spin-down values). In the case of non-collinear magnetism, the spin is directional, and therefore complex numbers are used to represent its modulus and angle, which adds a further level of complexity to the path computations (Ford et al., 2015). Considering these factors, the whole memory space of a compute node can easily get filled with the data associated with a single domain, and the pure MPI approach can only be executed with one process per node. This implies that many cores are idle, because no further processes can allocate the necessary memory for another domain on the same node to preserve data locality. Even when using relatively small domains that allow to simulate several domains per node, the increment in the number of moments causes a significant growth of the overlap area, which implies a large amount of complementary bond information for every process. Figure 1 presents a graphical description of these situations in a many-core environment.

In order to solve analogous scientific problems, from different kernel algorithms to large molecular dynamics codes, hybrid approaches with MPI and OpenMP have been previously implemented with positive results (Jung et al., 2014; Pal et al., 2014; Procacci, 2016). Consequently, the present work focuses on applying this hybrid scheme to the BOPfox code by extending

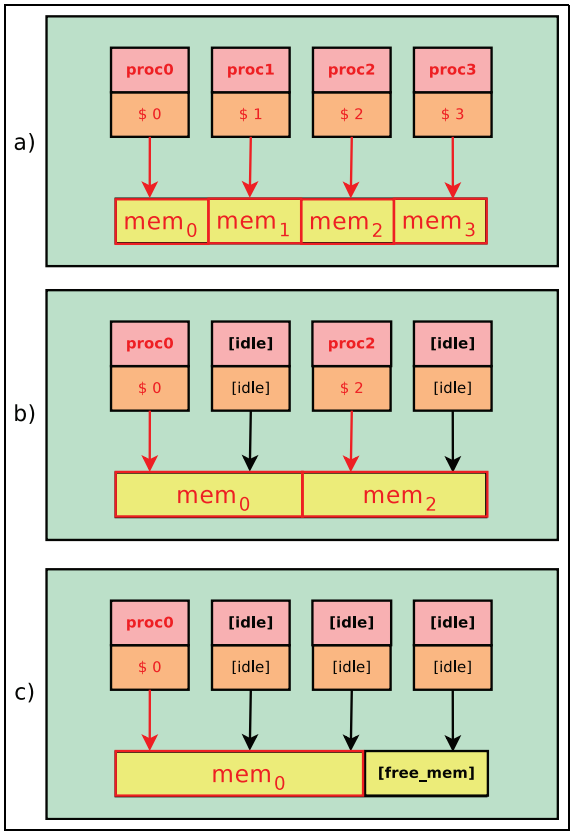

Figure 1. Example of problematic domain decomposition with pure MPI on a single node of many-core system with four processors. The first case (a) is the ideal situation, in which four domains fit correctly in memory, and therefore all processors are used. The second case (b) presents domains that occupied twice the memory space of (a), and therefore only two domains fit in memory and two processors in the node are idle. Finally, the third case (c) presents a worst-case scenario, in which the memory of the domain is occupying more than a half of the memory of the node, so that only one processor is working and there is a significant amount of memory space left unused.

the existing parallelization with pure MPI with the definition of OpenMP threads inside each MPI process. Following this method, the computations associated with a domain can be distributed into threads, and thereby two main goals are accomplished: (1) the use of many cores on a compute node to speed up the execution time and (2) the use of shared memory to minimize the replication of information in the overlap area. First, the analysis of different OpenMP approaches is presented in order to select the most effective parallel algorithm on shared memory, so that in the next step it can be combined with the efficient internode parallelization to provide an optimal and flexible parallel implementation of analytic BOPs. Finally, the performance of the parallel code is analysed up to large system sizes of several million atoms on highly parallel environments using two representative supercomputing

systems, so that the effective use of analytic BOPs for optimized large-scale simulations with high accuracy (up to 10 moments) is demonstrated.

3. OpenMP algorithms

The OpenMP parallelization is targeting the computation of self-returning paths in the schedule of analytic BOP computations. This task generally shows to consume more than 90% of the total execution time of the simulation code, and it is performed by two main loops, which compute the so-called interference and transfer paths, respectively (Teijeiro et al., 2016b). Both loops are very similar in structure, so that the differences between them do not imply new dependencies or structural modifications. Thus, the pseudo-codes included in the following sections are always presenting a generic structure of a path computation loop, on top of which the OpenMP directives and clauses are introduced.

The path computations are structured in two steps: (1) computation of half-length paths (from lengths 1 to $\lceil m/2 \rceil$, with m being the maximum moment expansion) and (2) merging of the previously computed paths to obtain longer paths up to length m. According to this, each iteration i of the loops is first computing the sets of half-length paths that start on an atom i and end on a set of endpoint atoms $j_1, j_2, \ldots, j_n$, which is a task that is guided using a to-do list (Teijeiro et al., 2016b) and therefore can be safely executed independently of other iterations. Some partial force contributions associated with the half-length paths are already obtained at this point. Later in the same iteration, these half-length paths are merged in order to obtain the complete force contribution between pairs of neighbour atoms up to the given moment expansion, so that the resulting longer paths start on an atom j_A and end on an atom j_B (both j_A and j_B being neighbour atoms and at the same time endpoints of the half-length paths from i). Unlike the first part, the storage of these force contributions is a source of conflicts for OpenMP, because different iterations may be updating the force contributions of a given bond. Each force contribution is referenced using a unique bond identifier for the same moment expansion between 1 and m, and therefore the same values may need to be updated in the iterations for atoms i_1, i_2 and i_3, as long as it is possible to merge paths on them that start and end on the same atoms j_A and j_B and have the same length: as a result, if different OpenMP threads are executing the loop iterations associated with i_1, i_2 and i_3, a race condition arises. Figure 2 illustrates this situation.

The following subsections show different OpenMP algorithms for the work distribution of path computations inside an MPI domain, and/or the full system in the case of pure shared-memory simulation (i.e. using only one process with MPI effectively deactivated).

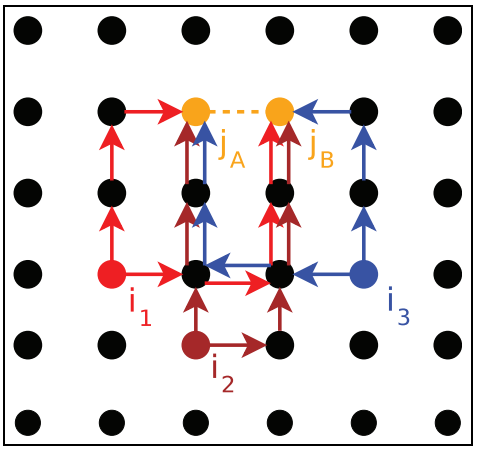

Figure 2. Example of conflict for the computation of force contributions for the bond between atoms j_A and j_B. After merging the marked paths that start on i_1, and i_3, all of them provide partial force computations associated with the constructed paths length of 7: consequently, they all update the same force values for the given bond, and a race condition occurs.

Therefore, in the next explanations, the expression *domain/system* is used to refer to both cases in a general way, whenever an algorithm can be applied to both. The algorithms are presented in an increasing order of complexity in their implementation: from the simplest approach with the minimum required changes to build a correct OpenMP code to more elaborated implementations that provide a finer tuning of the work distribution.

3.1. Base algorithm

The base algorithm introduces only the corresponding loop parallelization directive in the code for each path computation. Therefore, no further relevant change in the source code has been performed in order to obtain a correct OpenMP parallelization. Listing 1 presents a snippet of the structure of the path computation loops, which focuses on the most relevant elements of the implementation. The loop iterates over each atom i in a system with natom atoms and computes its corresponding path matrices, which are added conveniently in the array of force contributions force_contrib. Each force contribution is stored according to its associated bond identifier nbond (which does not depend on i, hence the race condition in Figure 2 occurs) and its path length nmom (between 1 and $m-1$, where m is the total moment expansion defined for the simulation).

As shown in Listing 1, the PARALLEL DO directive adds several clauses that describe the private or shared status of the variable, as well as the scheduling scheme for the iterations. The workload associated with every iteration will generally be very similar, because

```
!$OMP PARALLEL DO DEFAULT(NONE)
!$OMP& PRIVATE(i, nmom, nbond, path, ...)
!$OMP& FIRSTPRIVATE(...)
!$OMP& REDUCTION(+:force_contrib, ...)
!$OMP& SHARED(natom, ...)
!$OMP& SCHEDULE(DYNAMIC)
      do i = 1,natom
        ...
!       Forces updated for some half-length paths
        force_contrib(:,:,nmom,nbond) =
     &    force_contrib(:,:,nmom,nbond) + path(:,:)
        ...
!       Forces updated again for merged paths
        force_contrib(:,:,nmom,nbond) =
     &    force_contrib(:,:,nmom,nbond) + path(:,:)
        ...
      enddo
!$OMP END PARALLEL DO
```

Listing 1. Code snippet for the implementation of the base algorithm with OpenMP.

the path construction depends on the number of endpoint atoms for each path length, which can even be considered equal for a general case (Teijeiro et al., 2016b). However, the DYNAMIC scheduler is used here to avoid any possible imbalance.

The particular feature of this algorithm is the use of a REDUCTION clause to solve the race condition created by the storage of force contributions. As the target variable force_contrib is replicated for every OpenMP thread, the values can be updated locally and then reduced using an addition operation to get the final result. This is possible because the size of the force_contrib array is allocated outside the loop for the necessary number of bonds in the domain and overlap area, and therefore there are no size incompatibilities between threads.

The simplicity of this approach has one drawback: the memory allocation is increased because of the local copies of the force_contrib array. The size of this array is not as large as the information about every path stored during the computation of half-length paths, but it still represents a significant amount of information that grows polynomially with the number of moments (Teijeiro et al., 2016b).

3.2. Atomic algorithm

This approach, presented in Listing 2, is similar to the previous base algorithm in Section 3.1, but here the reduction operation over the force contribution array is removed, so that all OpenMP threads have access to it as a shared variable. Therefore, in order to solve the problem of concurrent accesses, an ATOMIC directive is introduced for all update operations performed over this array. As each path contribution is the result of multiplying the small matrices (i.e. pairwise Hamiltonians) associated with the bonds that form the path, the resulting path matrix is defined in terms of

```
!$OMP PARALLEL DO DEFAULT(NONE)
!$OMP& PRIVATE(i, nmom, nbond, path)
!$OMP& PRIVATE(j, norbj, k, norbk, ...)
!$OMP& FIRSTPRIVATE(...)
!$OMP& REDUCTION(...)
!$OMP& SHARED(natom, force_contrib, ...)
!$OMP& SCHEDULE(DYNAMIC)
      do i = 1,natom
        ...
!       Forces updated for some half-length paths
        do j = 1,norbj
          do k = 1,norbk
!$OMP ATOMIC
            force_contrib(k,j,nmom,nbond) =
     &        force_contrib(k,j,nmom,nbond) +
     &        path(k,j)
          enddo
        enddo
        ...
!       Forces updated again for merged paths
        do j = 1,norbj
          do k = 1,norbk
!$OMP ATOMIC
            force_contrib(k,j,nmom,nbond) =
     &        force_contrib(k,j,nmom,nbond) +
     &        path(k,j)
          enddo
        enddo
        ...
      enddo
!$OMP END PARALLEL DO
```

Listing 2. Code snippet for the implementation of the atomic algorithm with OpenMP. The atomic update of the force_contrib array is performed on its individual values using a loop over the number of orbitals norbj and norbk, associated with each of the atoms that define the target bond nbond.

the number of orbitals of the initial and final atoms (the maximum number of orbitals for all atoms in the system are used for the allocation of force_contrib). Consequently, every access to a matrix element is defined inside a loop and conveniently synchronized. Considering that the operations over the force_contrib array are simple additions and that the number of operations is restricted because of the small size of the path matrix, the atomic construct is chosen because it has shown to be more efficient than a critical section in these cases, even though the latter also provides a correct implementation. The number of orbitals norbj and norbk, which define the size of path, are used to implement the atomic updates of force contributions.

3.3. Overlap algorithm

The main idea behind this algorithm is to combine the work distribution between threads with the possibility of overlapping computations and communications specifically for the case of hybrid MPI/OpenMP simulations. Therefore, this implementation is explicitly based on the *BONDCOMMS* algorithm (Teijeiro et al., 2016a) for MPI parallelization, which defines the outer part of a domain as subject to communications of force contributions with the corresponding neighbour domains. In order to provide the desired functionality, there is an explicit classification of the atoms in every domain in two groups: (1) bulk-domain atoms, which

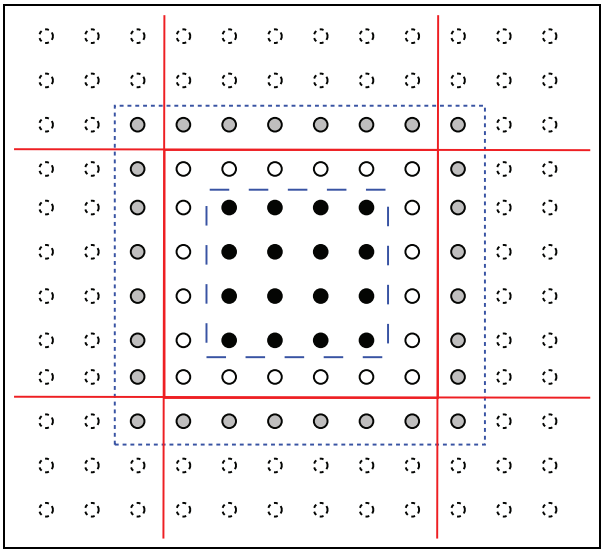

Figure 3. Simple example of an MPI domain in a system and additional overlap zones for path computations using two moments. The solid lines represent the domain boundaries. The dotted line represents the limit of the overlap zone atoms, which are the additional atoms outside the domain boundaries that are used to compute all possible paths locally inside a domain (for two moments only the nearest neighbours are necessary). The dashed line classifies the atoms inside the domain in bulk-domain atoms (black) and near-overlap atoms (white). As a consequence of its definition, the near-overlap atoms in a domain are actually the atoms in the overlap zone of the corresponding neighbour domain.

are located in the core of the domain and whose force contributions are fully computed by the domain without any communication, and (2) near-overlap atoms, which are located next to the overlap zone between domains, and whose force contributions should be considered for MPI communications. These basic concepts of the algorithm are explained in Figure 3, where the subdivision between bulk-domain, near-overlap and overlap zones between domains is re-marked for a simple two-dimensional system.

For a correct setup of this algorithm in the code, the atoms are initially classified as bulk-domain or near-overlap atoms using an additional flag, and then each atom in these groups is selected using an index that refers to it. This operation is performed by adding some minimal indexing information in the main initialization routine of the code, which is outside the core algorithm, and therefore only some negligible overhead is introduced with respect to the original code. Afterwards, both loops for path computations define an OpenMP parallel region, but without an explicit work distribution directive: this is explicitly done by taking the total number of threads and subdividing them into two groups, so that each group will process one task:

```
1  !$OMP PARALLEL DEFAULT(NONE)
2  !$OMP& PRIVATE(thread_id, nthreads)
3  !$OMP& PRIVATE(nthreads_overlap, nthreads_bulk)
4  !$OMP& PRIVATE(near_overlap_count, bulk_count)
5  !$OMP& PRIVATE(disp_bulk, disp_overlap)
6  !$OMP& PRIVATE(total_atoms_bulk, disp_bulk)
7  !$OMP& PRIVATE(total_atoms_overlap, disp_overlap)
8  !$OMP& PRIVATE(i, iatom, ...)
9  !$OMP& FIRSTPRIVATE(...)
10 !$OMP& SHARED(natom, semaphore, force_contrib)
11 !$OMP& SHARED(index_bulk, index_overlap, ...)
12 !      Define the number of threads for each task
13       nthreads = omp_get_num_threads()
14       thread_id = omp_get_thread_num()
15       nthreads_overlap = ceiling(
16      &  DBLE(near_overlap_count*nthreads)/
17      &  DBLE(natom))
18       if (nthreads_overlap < 1) nthreads_overlap = 1
19       nthreads_bulk = nthreads - nthreads_overlap
20
21       if (thread_id < nthreads_bulk) then
22 !       Process the bulk zone of the domain
23         total_atoms_bulk = bulk_count/nthreads_bulk
24         ...
25 !       Adjust total_atoms_bulk if needed and create
26 !         local indexes to process the correct amount
27 !         of atoms in each thread
28         ...
29 !       Main loop for path computations
30         do i = disp_bulk, disp_bulk +
31      &                   total_atoms_bulk-1
32 !         iatom = index_bulk(i)
33           ...
34 !         Path computations for near-overlap atoms:
35 !         ATOMIC is used to synchronize the access to
36 !           the shared force_contrib array
37           ...
38         enddo
39       else
40 !       Process the near-overlap zone of the domain
41         ...
42 !       Distribute the near-overlap atoms between
43 !         threads similarly to the bulk atoms
44         ...
45 !       Initialize semaphore for communications
46         semaphore = 0
47
48 !       Main loop for path computations: different
49 !         displacements are considered to match the
50 !         correct indexes for near-overlap atoms
51         do i = disp_overlap, disp_overlap +
52                              total_atoms_overlap-1
53 !         Get index of the selected atom
54           iatom = index_overlap(i)
55           ...
56 !         Path computations for near-overlap atoms:
57 !         ATOMIC is used to synchronize the access to
58 !           the shared force_contrib array
59           ...
60         enddo
61
62 !       When a thread finishes its chunk, the value
63 !         of the shared semaphore is increased
64 !$OMP ATOMIC
65         semaphore = semaphore + 1
66
67 !       When all threads have correctly increased
68 !         the semaphore, the last one performs the
69 !         MPI communication routine
70         if (semaphore == nthreads_overlap) then
71           call comms_inter_process
72         endif
73       endif
74 !$OMP END PARALLEL
```

Listing 3. Code snippet for the implementation of the overlap algorithm with OpenMP.

- One group of threads will compute paths and force contributions for the bulk-domain atoms, so that the processing will be very similar to the previous atomic algorithm in Section 3.2.
- The rest of the threads will process the near-overlap atoms, and right afterwards one of these threads will perform the necessary communications at MPI level with the neighbour domains, so that the force contributions for the target atoms are complete.

Listing 3 sketches the implementation of the basic functionality of this overlap algorithm. At least two threads are necessary to execute the code, so that at least one can execute the bulk-domain part and another the near-overlap part. The OpenMP threads are split into two groups with nthreads_bulk and nthreads_overlap threads that will process the bulk-domain and near-overlap zones, respectively, according to the thread identifier: the bulk-domain part is processed by threads with identifiers between 0 and nthreads_bulk-1, and the near-overlap part with the rest of threads up to nthreads-1. The subdivision of the work between both partitions is performed using two main indexes, index_bulk and index_overlap, that reference the identifiers of the bulk-domain and near-overlap atoms, respectively, inside the list of atoms in the system (from 1 to natom, as shown in Listings 1 and 2). According to these indexes, the correct chunk of atoms that each OpenMP thread should process is set using the private displacement variables disp_bulk and disp_overlap. The activation of communications in the near-overlap part is performed using a simple shared integer as a semaphore: every thread increases the semaphore after finishing its corresponding work. When the last thread finishes, it increases the semaphore to the number of threads in the near-overlap zone and thereby detects that the whole zone has already been processed, so only then the communications will be performed by that last thread. At that instant, the rest of the threads associated with the near-overlap part will already be performing an active wait at the end of the parallel region, and the threads associated with the bulk-domain zone can continue working independently.

3.4. Subdomain algorithm

In general, the need for ATOMIC operations for path computations is due to the transparent distribution of the iterations when using the DO directive, because two or many threads may be processing two neighbour atoms at the same time and try to update a force for the same bond and path length. Considering this fact, the concept of subdomains for OpenMP threads is introduced here to have a spatial subdivision of a domain, in order to facilitate the coordination of concurrent accesses to the force contribution array and minimize the use of ATOMIC updates. Essentially, every subdomain is associated with a single OpenMP thread, and two main zones are considered: (1) the subdomain-bulk zone, which includes all atoms and associated bonds whose force contributions are only computed by the corresponding thread, and therefore no ATOMIC accesses are required, and (2) the subdomain-overlap zone, where partial force contributions may be computed by another subdomain, and therefore synchronized accesses to update force contributions are necessary.

The implementation of this routine is presenting a significant amount of complexity, because it effectively performs an analogous domain decomposition to the MPI implementation: here each MPI domain is treated as the origin system, and communications between subdomains are substituted by ATOMIC writes to shared memory. As a result, it is necessary to perform significant changes in the main initialization routines to provide the spatial subdivision of the array of atoms for each domain. Considering the use of equal orthogonal domains and the use of Cartesian processor topology given by the MPI implementation (Teijeiro et al., 2016a), the creation of subdomains follows the same approach: every MPI process defines the same internal three-dimensional Cartesian topology, so that every domain is subdivided into equal orthogonal subdomains. In order to provide an efficient processing of path computations, the main atom information associated with each subdomain is stored consecutively and is marked with a start and a stop index. Moreover, the atoms are organized in order to have a first chunk with the subdomain-bulk atoms and another chunk with subdomain-overlap atoms.

Figure 4 provides a graphical explanation of the algorithm, where the different zones and arrays used here are presented for a two-dimensional case. Here the synchronization is not necessarily used for bonds associated with atoms in the border of the domain, because the full computation will be obtained with the MPI communications, without the interference of other threads in the same process.

Despite the introduction of the previous modifications, the performance of the code is never affected in a significant way, because only the main initialization routines of the code present some overhead. The most relevant modifications in the path computations part are shown next in Listing 4. The indexes subdomainstart and subdomainstop indicate the first and last element of each subdomain, respectively, and bulk_overlap_sep marks the last element of the subdomain-bulk part. In general lines, the changes in the main initialization and the use of these additional indexes are greatly simplifying the adaptation of the main algorithm to the use of subdomains. The base data structures for the path computations are not changing, so it is only necessary to indicate the initial and final atom identifiers for the chunk

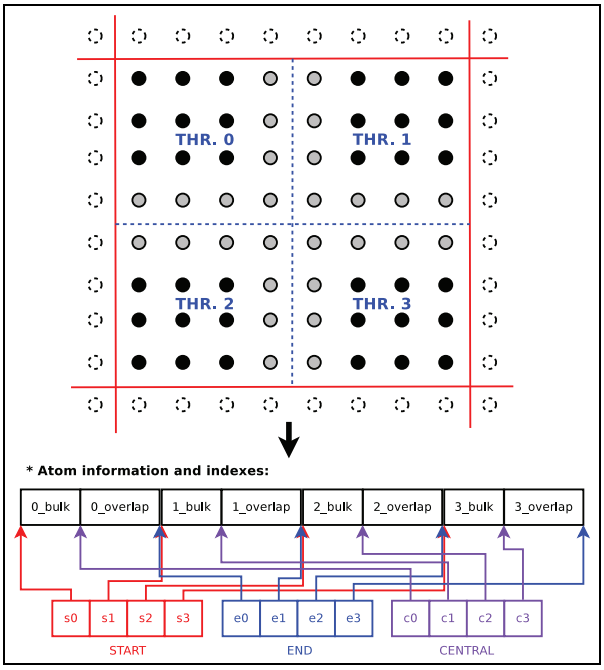

Figure 4. Simple example of the construction of subdomains by defining an inner Cartesian topology for OpenMP threads: here, four threads are defined, and each of them is associated with one subdomain. The organization of the arrays of atom information (e.g. positions, species and orbitals) for the whole domain is adapted to the inner topology for efficiency reasons, and it is described by using additional indexes: two indexes reference the starting and end positions of the information for each subdomain, respectively, and another index determines the central position that separates the subdomain-bulk and subdomain-overlap zones for the given subdomain. All these indexes are integer arrays with as many elements as the number of OpenMP threads used per MPI process. Thus, for example, `s1`, `e1` and `c1` represent the start, end and central indexes associated with thread 1, respectively.

```
1  !$OMP PARALLEL DEFAULT(NONE)
2  !$OMP& PRIVATE(i, j, k, thread_id, path, nmom)
3  !$OMP& PRIVATE(nbond, norbj, norbk, ...)
4  !$OMP& FIRSTPRIVATE(...)
5  !$OMP& SHARED(force_contrib, subdomainstart)
6  !$OMP& SHARED(subdomainstop, bulk_overlap_sep, ...)
7        thread_id = OMP_GET_THREAD_NUM()
8  !     Process the atoms in the subdomain-bulk zone
9        do i = subdomainstart(thread_id+1),
10    &         subdomainstart(thread_id+1) +
11    &           bulk_overlap_sep(thread_id+1)
12          ...
13 !        Forces updated for some half-length paths
14          force_contrib(:,:,nmom,nbond) =
15    &       force_contrib(:,:,nmom,nbond) + path(:,:)
16          ...
17 !        Forces updated again for merged paths
18          force_contrib(:,:,nmom,nbond) =
19    &       force_contrib(:,:,nmom,nbond) + path(:,:)
20          ...
21       enddo
22 !     Process the atoms in the subdomain-overlap zone
23       do iatom = subdomainstart(thread_id+1) +
24    &               bulk_overlap_sep(thread_id+1)+1,
25    &              subdomainstop(thread_id+1)
26          ...
27 !        Forces updated for some half-length paths
28          do j = 1,norbj
29            do k = 1,norbk
30 !$OMP ATOMIC
31                force_contrib(k,j,nmom,nbond) =
32    &             force_contrib(k,j,nmom,nbond) +
33    &             path(k,j)
34            enddo
35          enddo
36          ...
37 !        Forces updated again for merged paths
38          do j = 1,norbj
39            do k = 1,norbk
40 !$OMP ATOMIC
41                force_contrib(k,j,nmom,nbond) =
42    &             force_contrib(k,j,nmom,nbond) +
43    &             path(k,j)
44            enddo
45          enddo
46          ...
47       enddo
48 !$OMP END PARALLEL DO
```

Listing 4. Code snippet for the implementation of the subdomain algorithm with OpenMP.

associated with each thread. As shown in Listing 4, the ATOMIC directive is only required for the processing of the subdomain-overlap atoms: the subdomain-bulk parts are safely accessed by the corresponding thread without synchronization.

4. Performance evaluation

Two representative supercomputing systems from the Jülich Supercomputing Centre (JSC) have been used for the analysis of all codes. For general performance tests and comparison between algorithms, the first system used is JURECA Supercomputer – JSC (2017; ranked 57th in the TOP500 List of June 2016), which consists of 1872 compute nodes with two Intel Xeon E5-2680 Haswell CPUs (2 × 12 cores) and DDR4 memory with different sizes per node (for the present tests, nodes with 128 GB of memory have been used), using Mellanox EDR InfiniBand with non-blocking flat tree topology. For high scalability analysis, the test bed system has been JUQUEEN Supercomputer – JSC (2017; ranked 13th in the TOP500 List of June 2016), with 28,672 IBM BlueGene/Q nodes that make a total of 458,752 PowerPC A2 cores (16 cores per node with full crossbar switch), with a main memory of 16 GB SDRAM-DDR3 per node and a low-latency 5D Torus network at 40 GBps. The performance results obtained on JURECA have used the Intel Fortran compiler 16.0.2 (compliant with OpenMP 4), using the Intel MPI compiler 5.1.3.181 for the hybrid compilation with flags -xHOST -O3. The compiler used for JUQUEEN has been the IBM XL Fortran compiler 14.1 (compliant with OpenMP 3.1) with IBM XL MPI wrapper for BlueGene/Q, using -O3 -qstrict -qarch=qp -qtune=qp as optimization flags for the hybrid code. Regarding OpenMP thread pinning, all tests in JURECA present

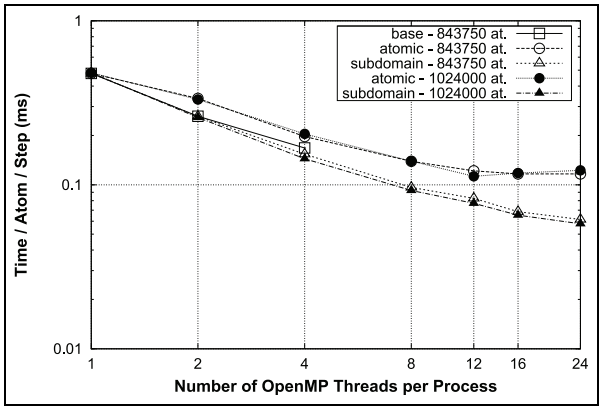

Figure 5. Performance comparison of base, atomic and subdomain algorithms on pure shared memory (one process, deactivated MPI) for the full computation of interference paths.

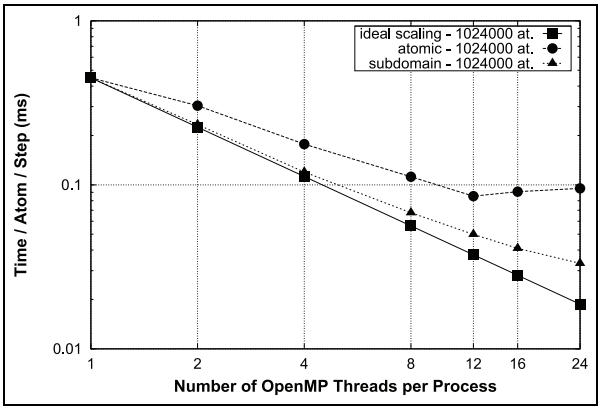

Figure 6. Algorithm comparison against the ideal speed-up on pure shared memory (one process, deactivated MPI) only for the parallelized loop that computes interference paths.

socket-level pinning, whereas the tests on JUQUEEN directly rely on the intra-node crossbar switch.

The simulated systems have been similar to previous works (Teijeiro et al., 2016a, 2016b): 3D cubic systems with regular bcc-W crystals (i.e. 5 valence orbitals per atom, which defines 5×5 bond matrices for path computations) and periodic boundary conditions on all dimensions. The BOP interaction scheme includes up to the second nearest neighbour in bcc lattices, like typical BOP parameterizations for bcc transition metals (see, e.g., Cak et al., 2014; Mrovec et al., 2004). The system sizes are always chosen in terms of the number of bcc cells per dimension, so that every tested size has a multiple of five cells per dimension. The 3D domain decomposition at MPI level subdivides the system into equal Cartesian domains, which are again decomposed in a similar way for the OpenMP subdomain algorithm in Section 3.4. The number of OpenMP threads varies from 1 (sequential or pure MPI) to the maximum number of physical processors in a node (24) for JURECA, whereas for JUQUEEN, the physical number of cores (16) is exceeded for some tests in order to present some results using the four-way multithreading features of the PowerPC cores (A2 Processor Users Manual for Blue Gene/Q, 2017). In general, the execution times have been obtained for BOP computations using a moment expansion $m = 6$ (i.e. with an overlap zone of $3 * R_{cut}$ around for each MPI domain, where R_{cut} is the cut-off radius for the applied W model), unless otherwise stated, and are represented in the graphs using the measure 'time per atom and step' (i.e. average time to obtain the energy and the force for one atom using one computation of paths).

Figure 5 compares the performance results of the more simple OpenMP algorithms described in Sections 3.1 and 3.2 with the subdomain algorithm from Section 3.4 in a pure shared memory scenario (i.e. with deactivated MPI and therefore only one process with different number of OpenMP threads) using a node of JURECA. The system sizes shown here are 843,750 and 1,024,000 atoms (i.e. 75 and 80 bcc cells per dimension, respectively), so that most of the memory of the compute node is used in both cases. These results correspond to the execution of the routine that computes the interference paths, which is taken as comparison benchmark for the core computations for BOPs: this routine includes a short initialization part (mostly consisting of array allocations for path matrices and force contributions), the core loop with the OpenMP parallelization and the MPI communications to exchange force contributions between domains (only for more than one process, when the MPI support is active).

Each of the algorithms shows that the use of OpenMP threads helps to obtain a significant speed-up in the computation of interference paths, but there are important differences in the computations. The basic algorithm obtains good results for two and four threads and 843,750 atoms, but it is not possible to increase the number of atoms because of the memory demand of the algorithm. The reason is the use of private copies of the array of force contributions for each thread, which is exhausting the memory of the node when many threads are used. In fact, in the case of tests with one million atoms, only one replication of this array is sufficient to saturate the memory, and therefore no results could be added to the graph. Nevertheless, when all relevant arrays in the simulation are defined as shared, such as for the atomic and the subdomain algorithms, it is possible to create many threads for every test case. The results of the atomic algorithm show good scalability, even though this tendency is broken when the memory space gets more filled and many OpenMP threads are used. This is due to (1) the accumulative overhead of atomic operations over every access to the force contributions array and (2) the irregular accesses to atom-

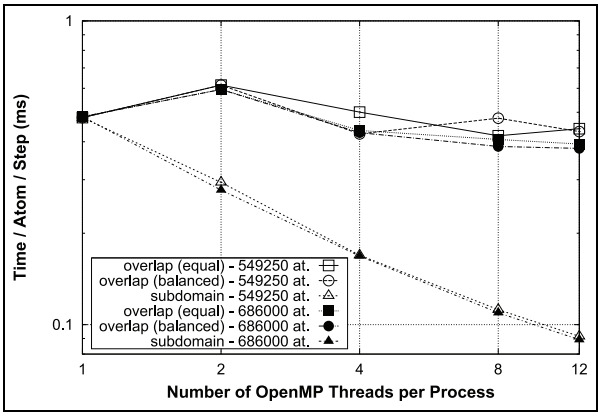

Figure 7. Performance comparison of overlap and subdomain algorithms on pure shared memory (one process, deactivated MPI) for the full computation of interference paths.

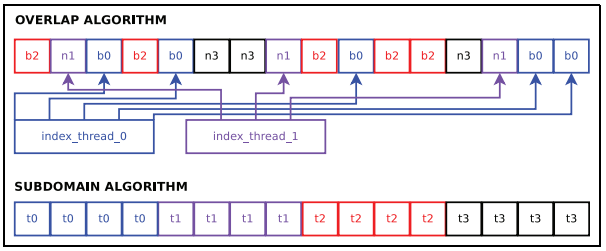

Figure 8. Comparison of the memory storage of atomic information for the overlap and the subdomain algorithms. Each element for the overlap algorithm is tagged according to its type, either as *b* for bulk-domain elements or *n* for near-overlap elements, with an additional number that indicates the thread that processes each element. For this algorithm, a thread will process only one type of elements. In contrast, the subdivision for the subdomain algorithm is done in a spatial way, and the elements associated with each thread (here tagged as *t* plus the thread number) can be accessed consecutively in memory using simpler indexes (cf. Figure 4).

related variables in small chunks, because of the internal implementation of the work-distribution routine of the DO directive.

This behaviour contrasts with the subdomain algorithm, which obtains very good speed-up for every execution up to the maximum number of physical cores on JURECA. In fact, these benefits are even clearer when looking at the results with only the core algorithm, that is, subtracting the execution time of the initialization routine with array allocations for interference paths, which is not parallelized. These results for the atomic and subdomain algorithms are compared in Figure 6 to the ideal speed-up, which is computed directly by dividing the execution time of the sequential code by the number of threads. Here, it is clearly shown how the subdomain algorithm always keeps close to the ideal for any number of threads, with significantly better results than the atomic algorithm. The main reason for this is the minimization of the ATOMIC updates on the force contribution array: for the case of large domains with few OpenMP threads and a restricted moment expansion, the number of updates tends to be negligible compared to the computational demand, and therefore OpenMP threads are almost obtaining a perfect speed-up. It is also important to note that here the OpenMP subdomains are based on the subdivision of 3D domains made of perfect crystals, and therefore the load balance between subdomains is optimal. An increment in the number of OpenMP threads for the same system size also increases the number of subdomain-overlap atoms that require ATOMIC operations, and performance is accordingly affected, as shown in the figure 6. In summary, the spatial decomposition with minimal ATOMIC operations and simple processing of consecutive chunks of atomic information per thread shows to be satisfactorily close to the optimal performance.

In particular, the influence in performance of these factors is clearly shown in Figure 7, where some results for the full computation of interference paths with the overlap algorithm in Section 3.3 are compared with the subdomain approach, using 549,250 and 686,000 atoms (65 and 70 bcc cells per dimension, respectively) in a node of JURECA. The overlap algorithm presents two variants depending on the number of threads that execute the bulk-domain and near-overlap zones: (1) *equal* indicates that both zones are processed with the same amount of threads and (2) *balanced* distributes threads among the two zones according to the number of atoms in each one. As Figure 7 shows, the differences between both algorithms are considerable, being the performance of the overlap algorithm very irregular and even worse for two threads than the sequential version. This behaviour is mainly due to the use of indirect index references on an unordered array implementation and the associated cache effects on JURECA. The overlap algorithm classifies atoms in bulk-domain and near-overlap, but the atomic information is not stored according to this subdivision: Only an additional index identifies the set of atoms that belong to each group.

Consequently, the comparison between the storage in memory of atomic information between these two algorithms can be illustrated as in Figure 8 for a shared array with 16 elements, where each element represents some information associated with one atom and two OpenMP threads. The subdivision of the processing for each set of atoms in the overlap algorithm implies that the atom information is loaded in memory in a very irregular way, so that first the index of the next atom in each set has to be obtained, and eventually there will be many times where a new chunk of memory will be loaded. Considering that the present tests are practically filling the memory space of the node, the cache

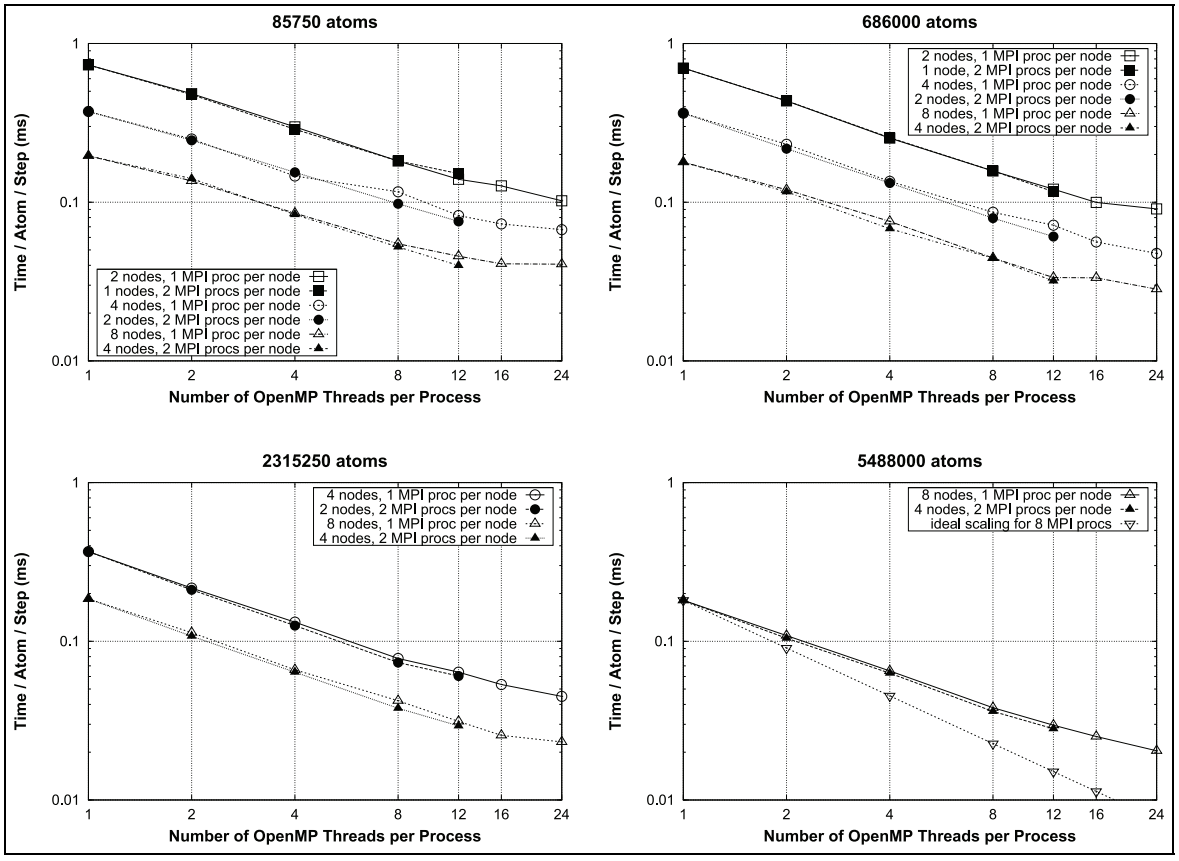

Figure 9. Performance results of the subdomain algorithm for interference and transfer path computations on the JURECA supercomputer, using different system sizes, number of MPI processes and OpenMP threads per process.

efficiency is significantly affected by the irregular access pattern. A second problem of the overlap algorithm is the number of synchronized accesses to the force contribution array: when there is no structured spatial processing of atomic bonds by each thread, it is not possible to minimize the effects of the race condition. As a result, the overlap algorithm is performing ATOMIC updates over the array of force contributions, which also implies a performance drawback with respect to the subdomain algorithm. Moreover, a third key factor is the difficult load balancing between threads for the partitioning: the *balanced* variant is, in general, better than the *equal*, but even the accuracy of a balanced threads distribution is very limited, and the benefits of this approach are still very much influenced by the irregular memory accesses. This third factor implies that, in general, the subdomain algorithm will be able to provide higher flexibility in the work distribution, always with equal or better load balancing between threads.

As a result of the previous comparison, the subdomain algorithm has obtained the best overall results, which have proven to be very satisfactory for pure shared memory tests with OpenMP. Therefore, this algorithm is used in the rest of this evaluation for the hybrid shared/distributed memory tests in combination with MPI. In Figure 9, a more extensive performance analysis for this algorithm is presented using two, four and eight MPI processes for up to eight compute nodes of JURECA for systems with 85,750, 686,000, 2,315,250 and 5,488,000 atoms (35, 70, 105 and 140 bcc cells per dimension, respectively), so that the results are presented whenever the system size can be allocated within the memory associated with the selected number of nodes (e.g. the largest system size can only be executed on eight nodes). Here, the results of the full computation of paths are considered, that is, both interference and transfer paths with their specific initialization routines (array allocations) and MPI communications. When using one process per node, each process generates up to 24 OpenMP threads per process, and only up to 12 threads when two processes per node are used. Considering the processor architecture of JURECA, all tests with up to 12 OpenMP threads are executed inside the same socket to maximize performance.

The study with different system sizes indicates several facts. The most relevant is that the use of more

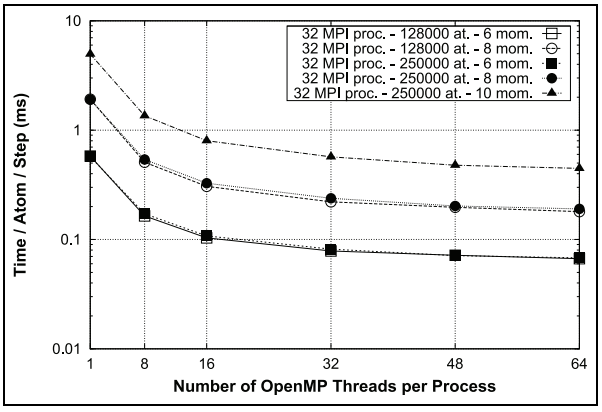

Figure 10. Performance results for the subdomain algorithm with different moment expansions using 32 nodes (1 MPI process per node) and up to 64 OpenMP threads per MPI process.

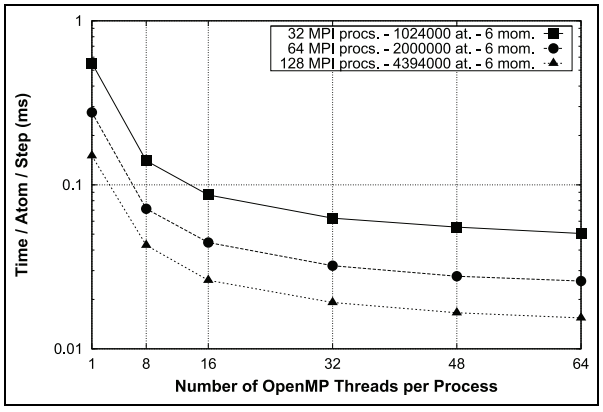

Figure 11. Performance results for the subdomain algorithm on weak scaling tests up to 8192 OpenMP threads (up to 64 threads per node on 128 nodes, with 1 MPI process per node).

OpenMP threads per process in JURECA is providing a performance benefit for each test case. For the small systems, the benefits are very restricted when all cores in a node are used, but they notably increase for larger systems: the slopes of the results get steeper and even the execution time per atom gets slightly lower as the system size grows. As here the full computation of paths is measured, the results are not as close to the ideal scaling behaviour as the pure shared memory tests in Figure 6, as shown in the graph with 5,488,000 atoms: when the number of threads increases, the non-OpenMP-parallelized initial array allocations and MPI communications associated with both interference and transfer path computations get to be more significant. This happens because the execution time of core computations is reduced by OpenMP threads, but the execution time of MPI communications only depends on the number of MPI processes (an estimate of the communication sizes can be found in Teijeiro et al., 2016a). Nevertheless, the fact that it is possible to create OpenMP threads per process without significant memory overhead (i.e. the memory requirements of the pure MPI code are not growing significantly), alongside with the positive impact in performance of the use of OpenMP threads for all the tests performed here, indicates that the present hybrid solution is definitely suitable for highly parallel simulations. It is also important to note that the use of highly regular test systems with an almost perfect load balancing in the MPI domains represents a strong benefit for a pure MPI implementation over the use of OpenMP threads, and consequently the graphs are generally showing slightly better results when two MPI processes per node are used.

Figure 10 shows the performance results of simulations for 32 MPI processes with many OpenMP threads on JUQUEEN (i.e. one MPI process per node, with many OpenMP threads per process) for different moment expansions using systems of 128,000 and 250,000 atoms (40 and 50 bcc cells per dimension, respectively) with the subdomain algorithm. Here, the execution times correspond to the full computation of paths (i.e. interference and transfer paths, with initial array allocations and MPI communication routine). The overall performance of the tests on the JUQUEEN system is significantly lower with respect to the results on JURECA, mainly because JUQUEEN processors are less powerful and the whole system is more focused on providing a very efficient communications system than in the optimization of intra-node computations. Therefore, JUQUEEN is particularly suitable to perform scalability tests, and in the results in this graph are showing two examples of worst-case scenarios, in which a minimum number of atoms per domain are simulated with a large number of OpenMP threads, and up to the maximum moment expansion allowed by the MPI domain decomposition with 32 processes (Teijeiro et al., 2016a). Even in these extreme cases, the hybrid MPI/OpenMP implementation of the subdomain algorithm is presenting always better results when all cores in the JUQUEEN nodes are used (16 cores per node), also including the cases when the full power of the four-way multithreading of all the cores in JUQUEEN (64 OpenMP threads), which makes a total of 512 threads on 32 MPI processes. The reduced size of these test cases with respect to the number of threads implies that there are no effective bulk-subdomain zones: therefore, all accesses to the shared force contributions array are controlled by the ATOMIC directive. Nevertheless, the presence of scalability for all test cases and the fact that the speed-up ratio is always kept with a growing number of computations (both in the case of larger system and longer moment expansion) are confirming the optimal behaviour of this algorithm in these worst-case scenarios.

Finally, Figure 11 presents weak scaling results for the subdomain algorithm on JUQUEEN for executions with 1,024,000, 2,000,000 and 4,394,000 atoms (80, 100 and 130 bcc cells per dimension) using 32, 64 and 128 compute nodes, respectively. These system sizes are chosen in order to obtain present similar MPI domains per process: more precisely, each node of JUQUEEN contains one MPI process with an associated domain of around 32,000 atoms, which is simulated with many OpenMP processes, therefore having a maximum number of 8192 threads for the largest test case (i.e. an MPI grid of $8 \times 4 \times 4$ processors, with an OpenMP subgrid of $4 \times 4 \times 4$ threads). These results show a very good scaling behaviour for the hybrid parallel algorithm up to 64 MPI processes, and only when using 128 processes the results present more restricted speed-up, which is due to the relative difference in the domain sizes with respect to the tests with 32 and 64 processes, and also to the less optimized intra-node processing combined with larger MPI communication sizes for 128 nodes. Nevertheless, the introduction of OpenMP threads shows to maintain an analogous positive trend for all test cases, obtaining similar performance benefits for the same amount of threads regardless of the number of MPI processes. Moreover, these results confirm that the use of large domain sizes is advisable whenever possible (Teijeiro et al., 2016a), so that the memory use of the compute nodes (and thereby the amount of local computations) is maximized with respect to communications.

5. Conclusions

This work has presented the design, implementation and performance analysis of the parallel simulation of analytic BOPs on hybrid shared/distributed memory architectures using the BOPfox code (Hammerschmidt et al., 2017). Taking previous works on MPI parallelizations as a basis, the main new contribution from the present article is the development of an extended implementation to exploit intra-node shared memory computations with OpenMP, in order to obtain high scalability on top of an optimal use of the resources of many-core supercomputers. After identifying a main race condition on the update of force contributions between neighbour atoms, different algorithms have been compared to show several OpenMP implementations that can manage the data dependencies in an effective way.

The performance results indicate that a fine-tuned control over the OpenMP threads leads to better performance. In particular, the main benefit is obtained by a subdomain decomposition of every MPI domain, because of the structured access to the shared variables and the minimization of synchronized ATOMIC operations with OpenMP. This subdomain implementation performs a tailored initialization routine that supports a highly scalable hybrid implementation, which is able to obtain remarkable performance benefits from the use of all physical cores on a general-purpose supercomputer. Moreover, this implementation is able to simulate system sizes of up to millions of atoms with several thousand threads on a BlueGene/Q supercomputer, showing that the code is able to make a full and efficient use of the allocated computational power, in terms of memory and number of cores, also exploiting the full multithreading capabilities of the system.

In summary, the present hybrid memory implementation shows to be optimal for the exploitation of highly parallel simulations of very large systems on many-core supercomputers. Consequently, this algorithm based on subdomains is able to overcome the main computational restrictions of highly accurate analytic BOPs, facilitating its application in future large-scale simulations.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work has been financially supported by ThyssenKrupp Steel Europe AG.

References

A2 Processor Users Manual for Blue Gene/Q (2017) Available at: http://wiki.alcf.anl.gov/parts/images/c/cf/A2.pdf (accessed May 2017).

Cak M, Hammerschmidt T, Rogal J, et al. (2014) Analytic bond-order potentials for the bcc refractory metals Nb, Ta, Mo and W. *Journal of Physics: Condensed Matter* 26: 195501.

Diemand J, Angélil R, Tanaka KK, et al. (2013) Large scale molecular dynamics simulations of homogeneous nucleation. *Journal of Chemical Physics* 139(7): 074309.

Drautz R and Pettifor DG (2006) Valence-dependent analytic bond-order potential for transition metals. *Physical Review B* 74: 174117.

Drautz R and Pettifor DG (2011) Valence-dependent analytic bond-order potential for magnetic transition metals. *Physical Review B* 84: 214114.

Drautz R, Hammerschmidt T, Cak M, et al. (2015) Bond-order potentials: derivation and parameterization for refractory elements. *Modelling and Simulation in Materials Science and Engineering* 23: 074004.

Ford ME, Drautz R, Hammerschmidt T, et al. (2014) Convergence of an analytic bond-order potential for collinear magnetism in Fe. *Modelling and Simulation in Materials Science and Engineering* 22: 034005.

Ford ME, Pettifor DG and Drautz R (2015) Non-collinear magnetism with analytic bond-order potentials. *Journal of Physics: Condensed Matter* 27(8): 086002.

Frauenheim T, Seifert G, Elstner M, et al. (2002) Atomistic simulations of complex materials: ground-state and excited-state properties. *Journal of Physics: Condensed Matter* 14(11): 3015.

Hammerschmidt T, Drautz R and Pettifor DG (2009) Atomistic modelling of materials with bond-order potentials. *International Journal of Materials Research* 100: 11.

Hammerschmidt T, Seiser B, Ford ME, et al. (2017) BOPfox program for tight-binding and analytic bond-order potential calculations (unsubmitted manuscript).

Jones RO and Gunnarsson O (1989) The density functional formalism, its applications and prospects. *Reviews of Modern Physics* 61: 689–746.

Jung J, Mori T and Sugita Y (2014) Midpoint cell method for hybrid (MPI + OpenMP) parallelization of molecular dynamics simulations. *Journal of Computational Chemistry* 35(14): 1064–1072.

JUQUEEN Supercomputer – JSC (2017) Available at: http://www.fz-juelich.de/ias/jsc/juqueen (accessed May 2017).

JURECA Supercomputer – JSC (2017) Available at: http://www.fz-juelich.de/ias/jsc/jureca (accessed May 2017).

Konstanze RH, Melis C and Colombo L (2016) Structural, vibrational, and thermal properties of nanocrystalline graphene in atomistic simulations. *Journal of Physical Chemistry C* 120(5): 3026–3035.

Message Passing Interface (MPI) Forum (2017) Available at: http://www.mpi-forum.org (accessed May 2017).

Mishin Y (2005) *Interatomic Potentials for Metals*. Netherlands: Springer, pp. 459–478.

Mrovec M, Nguyen-Manh D, Pettifor DG, et al. (2004) Bond-order potential for molybdenum: application to dislocation behaviour. *Physical Review B* 69: 094115.

Nakano A, Kalia RK, Nomura K, et al. (2008) De novo ultrascale atomistic simulations on high-end parallel supercomputers. *International Journal of High Performance Computing Applications* 22(1): 113–128.

Pal A, Agarwala A, Raha S, et al. (2014) Performance metrics in a hybrid MPI–OpenMP based molecular dynamics simulation with short-range interactions. *Journal of Parallel and Distributed Computing* 74(3): 2203–2214.

Pei QX, Lu C and Lee HP (2007) Large scale molecular dynamics study of nanometric machining of copper. *Computational Materials Science* 41(2): 177–185.

Procacci P (2016) Hybrid MPI/OpenMP implementation of the ORAC molecular dynamics program for generalized ensemble and fast switching alchemical simulations. *Journal of Chemical Information and Modeling* 56(6): 1117–1121.

Qi Z, Campbell DK and Park HS (2014) Atomistic simulations of tension-induced large deformation and stretchability in graphene kirigami. *Physical Review B* 90: 245437.

Sutton AP, Finnis MW, Pettifor DG, et al. (1988) The tight-binding bond model. *Journal of Physics C* 21: 35.

Teijeiro C, Hammerschmidt T, Drautz R, et al. (2015) Parallel bond order potentials for materials science simulations. In: *4th International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG'15)*, Dubrovnik, Croatia, 24–27 March 2015, Paper 4.

Teijeiro C, Hammerschmidt T, Drautz R, et al. (2016a) Efficient parallelization of analytic bond-order potentials for large-scale atomistic simulations. *Computer Physics Communications* 204: 64–73.

Teijeiro C, Hammerschmidt T, Seiser B, et al. (2016b) Complexity analysis of simulations with analytic bond-order potentials. *Modelling and Simulation in Materials Science and Engineering* 24: 025008.

Tersoff J (1989) Modeling solid-state chemistry: interatomic potentials for multicomponent systems. *Physical Review B* 39: 5566–5568.

The Interatomic Potentials Repository Project (2017) Available at: http://www.ctcms.nist.gov/potentials/ (accessed May 2017).

The OpenMP® API specification for parallel programming (2017) Available at: http://www.openmp.org (accessed May 2017).

Author biographies

Carlos Teijeiro received his BSc (2006), MSc (2008) and PhD (2013) degrees in Computer Science from the University of A Coruña, Spain. After working as predoctoral research fellow in collaboration with the Supercomputing Center of Galicia, Hewlett-Packard and the Forschungszentrum Jülich, currently he is a postdoctoral research assistant in the Interdisciplinary Centre for Advanced Materials Simulation (ICAMS) at the Ruhr-University Bochum, Germany. His main research interests are the development of high performance computing (HPC) solutions applied to large-scale simulations in materials science.

Thomas Hammerschmidt obtained his degree in Physics in the Technical University of Munich (TUM) in 2002 and his PhD at the Fritz Haber Institute of the Max Planck Society (FHI-MPG) in 2006. He worked at the Department of Materials in the University of Oxford as postdoctoral research assistant until 2008, when he took the position of group leader in the Interdisciplinary Centre for Advanced Materials Simulation (ICAMS) at the Ruhr-Universität Bochum. The main focus of Thomas Hammerschmidt's research work is the modelling of accurate interatomic interactions by means of analytic bond-order potentials (BOPs), and he also coordinates the development of different software simulation tools associated to BOPs, such as the BOPfox code.

Ralf Drautz holds the chair for Atomistic Modelling and Simulation at the Interdisciplinary Centre for Advanced Materials Simulation (ICAMS) at Ruhr-Universität Bochum, where he also is a professor in the Department of Physics and Astronomy. He conducted the work on his PhD at the Max-Planck-Institut für Metallforschung in Stuttgart and spent time as a visiting scholar at the University of Texas at Austin and as

a senior research fellow at the University of Oxford. In 2008, Ralf Drautz was the first managing director of ICAMS. Ralf Drautz's research focuses on the theory of interatomic interactions and its application to understanding, predicting and designing properties of materials, including long and large atomistic simulations.

Godehard Sutmann has studied Theoretical Physics in Göttingen, Montpellier and Heidelberg, and received his PhD in 1999 at the Technical University Munich. He accomplished several postdoctoral research at the University of Trento, with research stays at NIH Washington, Santa Barbara and Mexico City. He is working in the Jülich Supercomputing Centre at Forschungszentrum Jülich, where he is the leader of the Simulation Laboratory Molecular Systems and deputy head of the division Computational Science. He is the director of the CECAM Node Jülich. He is also a professor for High Performance Computing in the Interdisciplinary Centre for Advanced Materials Simulation (ICAMS) at the Ruhr-University Bochum. He works in the fields of high performance computing and applied mathematics, with main focus in the development of computational methods for complex many-particle physics and statistical physics.