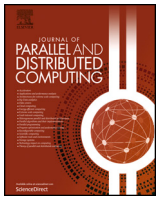
Journal of Parallel and Distributed Computing 193 (2024) 104945

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

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



3D DFT by block tensor-matrix multiplication via a modified Cannon's algorithm: Implementation and scaling on distributed-memory clusters with fat tree networks

Nitin Malapally [a,b], Viacheslav Bolnykh [a,1], Estela Suarez [b,c], Paolo Carloni [a,d], Thomas Lippert [b,e], Davide Mandelli [a,*]

[a] Computational Biomedicine (IAS-5/INM-9), Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428, Jülich, Germany
[b] Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428, Jülich, Germany
[c] Computer Science Department, University of Bonn, Bonn, Germany
[d] Molecular Neuroscience and Neuroimaging (INM-11), Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428, Jülich, Germany
[e] Frankfurt Institute for Advanced Studies, Goethe Universität, Frankfurt, Germany

ARTICLE INFO

Keywords:
3D Discrete Fourier Transform (3D DFT)
Block tensor matrix multiplication
Volumetric decomposition
Cannon's algorithm
Shared-memory parallelism
Distributed-memory parallelism
Overlapping communication and computation
MPI
Performance analysis
Roofline model

ABSTRACT

A known scalability bottleneck of the parallel 3D FFT is its use of all-to-all communications. Here, we present S3DFT, a library that circumvents this by using point-to-point communication – albeit at a higher arithmetic complexity. This approach exploits three variants of Cannon's algorithm with adaptations for block tensor-matrix multiplications. We demonstrate S3DFT's efficient use of hardware resources, and its scaling using up to 16,464 cores of the JUWELS Cluster. However, in a comparison with well-established 3D FFT libraries, its parallel efficiency and performance were found to fall behind. A detailed analysis identifies the cause in two of its component algorithms, which scale poorly owing to how their communication patterns are mapped in subsets of the fat tree topology. This result exposes a potential drawback of running block-wise parallel algorithms on systems with fat tree networks caused by increased communication latencies along specific directions of the mesh of processing elements.

1. Introduction and motivation

Many fields of numerical simulation such as astrophysics, plasma physics and molecular dynamics (MD) involve computing the pairwise long-range interactions between the physical system's constituents [1–3]. Some examples are gravitational forces, Van der Waals and electrostatic interactions. This computation is time-consuming and often restricts sizes and time scales. For example, in atomistic MD simulations of bio-physical systems, the sizes to be simulated can be very large – ranging up to 10^9 [4] particles – and computing the long-range interactions is typically responsible for 90% of the total run-time. This problem is particularly relevant in the field of quantum mechanics-based MD simulations [5]. To limit the computational costs and improve scaling, techniques derived from the Ewald summation method are extensively used, which utilize the three dimensional Discrete Fourier Transform (3D DFT) – both to ensure the convergence of the calculation and to gain speed-up [6–8]. Therefore, improving the scalability of the parallel 3D DFT is very desirable as it will extend the scope of these simulations – enabling the study of larger and more complex systems and for longer times – exploiting modern, massively parallel computer architectures [9].

The DFT operation is typically applied using any of the set of algorithms known collectively under the name of Fast Fourier Transform (FFT) [10]. Specifically, variants of the Cooley-Tukey FFT algorithm are the most commonly employed. They break the original DFT problem down into a tree of smaller DFT problems, which are sometimes solved recursively and more often non-recursively [11]. This results in a drastic reduction of arithmetic complexity from $O(N^2)$ of the naïve algorithm, down to $O(N \log_2 N)$. Similarly, the 3D FFT operation reduces the arithmetic complexity from $O(N^4)$ to $O(N^3 \log_2 N)$. Depending on

* Corresponding author.
E-mail addresses: n.malapally@fz-juelich.de (N. Malapally), bolnykh@gmail.com (V. Bolnykh), e.suarez@fz-juelich.de (E. Suarez), p.carloni@fz-juelich.de (P. Carloni), th.lippert@fz-juelich.de (T. Lippert), d.mandelli@fz-juelich.de (D. Mandelli).
[1] Author was affiliated with the above institute until December 2021.

https://doi.org/10.1016/j.jpdc.2024.104945
Received 23 May 2023; Received in revised form 5 March 2024; Accepted 20 June 2024
Available online 28 June 2024

the size of the problem, this may considerably reduce the run-time of computer applications. However, in the context of distributed-memory computers, the run-time of the 3D FFT is dominated by communication, which can make up to 80–95% of it [12,13], and so there is still interest in improving the scaling performance of the 3D FFT algorithm [12–15].

Reports suggest that the high fraction of communication time of the 3D FFT is due to its unavoidable use of all-to-all communications [12,16]. Further, on the Fugaku supercomputer, it has been shown that 3D FFT algorithms that make use of point-to-point communication scale better than those that make use of all-to-all communication [13]. This indicates that one may profit from 3D DFT algorithms that achieve better scalability by means of alternative communication patterns.

In order to achieve better scalability by swapping all-to-all for point-to-point communication, in 2015, Sedukhin et al. studied the scalability of an alternative algorithm that makes use of point-to-point communication to compute the 3D DFT, albeit at the significantly higher arithmetic complexity of $O(N^4)$ as compared to that of $O(N^3 \log N)$ of the 3D FFT algorithm. Their work implemented a point-to-point orbital algorithm that targeted the IBM Blue Gene/Q computer, which is based on a 5D torus topology. Their benchmarking revealed worse overall performance of their implementation compared to that of the standard 3D FFT implementations of the time. However, the authors noted that, for a single node, their implementation of the core computational operation of the algorithm – the tensor-matrix multiplication – achieved 20% of the corresponding peak performance, and concluded with the speculation that a more efficient implementation could outperform the 3D FFT algorithm for very large node counts [17]. Since then, the computational power of CPUs has grown more than the speed of the interconnects between the nodes [18], a fact which leans in favour of this alternative approach.

Inspired by the work of Ref. [17], we have designed and implemented a new 3D DFT algorithm that makes use of point-to-point communication, and benchmarked it on the JUWELS Cluster [19] at the Jülich Supercomputing Center. This cluster is based on the fat tree network topology, which is one of the most commonly adopted network topologies nowadays. Our algorithm – which we refer to as 3D DFT by block tensor-matrix multiplication – is based on specially adapted variants of Cannon's algorithm, which, in its original form, is an efficient distributed-memory matrix-matrix multiplication algorithm suited for square matrices [20–22]. We chose this algorithm because of its scalability [21] and the simplicity of its implementation. Our adaptations not only make it possible to use tensor operands, but also enable the utilization of the well-known strategy of overlapping communication and computation – by dedicating a thread to communication with the help of a custom work-sharing function for OpenMP-based multithreading – in an effort to hide the latency of communication. Further, by carefully applying only static scheduling, which almost entirely eliminates the use of processor interconnects in a non-uniform memory access (NUMA) setup, our implementation enables the efficient use of multithreading across NUMA domains. This avoids the use of distributed-memory parallelism within a shared-memory space, thereby increasing the efficiency of hardware resource utilization on the single node level.

We have implemented our algorithm as a C++ library that we named S3DFT and compared it with two competitive FFT implementations, namely, the FFTW3 library [11] and the Intel Math Kernel Library (MKL). This choice is motivated by the fact that these libraries are well-established, high-performing and arguably the most used worldwide in various applications.[2] The results demonstrate strong scaling of S3DFT up to the maximum number of 16,464 cores considered. However, we found its overall performance to fall behind those of its competitors. A detailed analysis uncovered the source of the problem in two of its distributed-memory components that scale poorly owing to how their intrinsic communication patterns are mapped in subsets of the fat tree topology. Algorithmic variants of these components designed to overcome this problem significantly improved the strong-scaling performance of S3DFT, but did not lead to overall increase of its performance due to the overhead of the additional global data transpositions required.

Summarizing the main contributions of our work:

- We have designed and implemented a 3D DFT algorithm that uses only point-to-point communication.
- In this endeavour, we have designed, implemented and analyzed the performance of variants of Cannon's algorithm with adaptations that extend its use for block tensor-matrix multiplications on volumetrically decomposed domains.
- We have exposed a potential drawback when running block-wise parallel algorithms on machines with a fat tree network.
- We have compared the strong scaling performance of S3DFT with that of FFTW3 and Intel MKL on the pre-exascale JUWELS Cluster at the Jülich Supercomputing Center. This represents useful information for researchers who routinely use these popular 3D FFT libraries on similar machines.

The paper is organized as follows. Section 2 introduces the notation used throughout the paper, reviews key concepts needed to describe our 3D DFT algorithm, and describes the modeling of the overlapping of the communication and the computation used to select the combination of block size and node count in the performance analysis of the distributed-memory block tensor-matrix multiplication algorithm. Section 3 outlines the details of the actual implementation of S3DFT. Section 4 provides the specifications of the JUWELS Cluster that was used for the benchmarking. Section 5 presents a performance analysis of the core functions and provides insights into the overall performance of the algorithm. Finally, sections 6 and 7 discuss the results of comparisons of S3DFT with FFTW3 and Intel MKL and put forward our conclusions.

2. Theory

2.1. 3D DFT by block tensor-matrix multiplication

The DFT for a sequence of 3D operand data $x(l,m,n) \in \mathbb{C}$ can be written as [17][3]

$$y(i,j,k) = \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \sum_{l=0}^{N-1} x(l,m,n)\; c(l,i)\; c(m,j)\; c(n,k) \tag{1}$$

where the coefficients

$$c(n_1,n_2) = \exp(-i\,\frac{2\pi}{N} n_1 n_2) \quad \forall\; 0 \le n_1, n_2 < N$$

define the DFT matrix $C \in \mathbb{C}^{N\times N}$. Making use of the formalism of Kolda et al. [23], the operand data $x(l,m,n)$ can be viewed as an order-3 tensor[4] $X \in \mathbb{C}^{N\times N\times N}$ existing in a 3D space defined by orthogonal directions $1,2,3$, with the data $x(l,m,n)$ arranged as a cubic mesh. Now, we introduce a right product of X and a matrix $A \in \mathbb{C}^{N\times N}$ in terms of their mode-3 product as

$$Z_R = \rho_R(X,A) = X \times_3 A^T, \tag{2}$$

[2] The Intel Math Kernel Library is the fastest and most-used math library for Intel-based systems (data from Evans Data Software Developer survey, 2022). The reference publication discussing the implementation of FFTW3 [11] has more than 3,200 citations in papers and more than 40 citations in patents.

[3] In this work, we only consider the case in which the operand data $x(l,m,n)$ can be arranged as a cube, i.e., $0 \le l,m,n < N$. To extend the functionality to irregular cuboids, load balancing schemes must be additionally developed, which goes beyond the scope of this work.

[4] For brevity, henceforth, we shall take "tensor" to mean the order-3 tensor.

Algorithm 1 Procedure to perform the tensor-matrix multiplication ρ_R. Here, $Z_R^{(r)}$, $X^{(r)}$ and A are matrices. The tensors Z_R, X can be obtained by stacking their slices $Z_R^{(r)}$, $X^{(r)}$ along dimension 1 (see Fig. 1a).

1: **for** $r \leftarrow 0$ to N **do**
2: $\quad Z_R^{(r)} \leftarrow X^{(r)} A$
3: **end for**

Algorithm 2 Procedure to perform the tensor-matrix multiplication ρ_L. Here, $Z_L^{(r)}$, $X^{(r)}$ and A are matrices. The tensors Z_L, X are obtained by stacking the slices $Z_R^{(r)}$, $X^{(r)}$ along dimension 1 (see Fig. 1b).

1: **for** $r \leftarrow 0$ to N **do**
2: $\quad Z_L^{(r)} \leftarrow A^T X^{(r)}$
3: **end for**

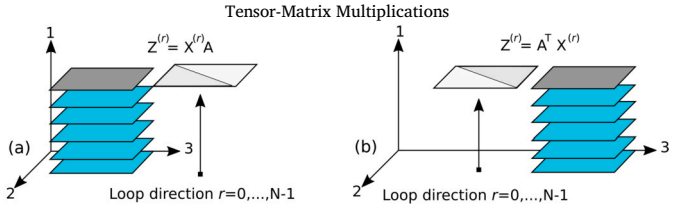


Fig. 1. Visualization of the procedures to compute the tensor-matrix multiplications (a) ρ_R and (b) ρ_L, as a set of independent matrix-matrix multiplications.

and a left product in terms of their mode-2 product as

$$Z_L = \rho_L(X, A) = X \times_2 A^T, \tag{3}$$

where $Z_R, Z_L \in \mathbb{C}^{N\times N\times N}$. These products can be conceived as a set of independent matrix-matrix multiplications as detailed in Algorithms 1 and 2. To visualize this, one can view X as a stack of N matrices $X^{(r)} \in \mathbb{C}^{N\times N}$ for $0 \le r < N$, which we shall call the slices of the tensor, piled up along any of the 3 orthogonal directions. In practice, to ensure a contiguous memory layout for optimal data access, we fix the piling direction to direction 1, as illustrated in Fig. 1.

Using the products defined by equations (2) and (3), it can be shown that the 3D DFT equation (1) can be rewritten in terms of tensor-matrix multiplications as

$$Y = \tau_2(\rho_R(\tau_2(\rho_L(\rho_R(X, C), C)), C)) \tag{4}$$

where $X, Y \in \mathbb{C}^{N\times N\times N}$ are the input and output tensors respectively, and $C \in \mathbb{C}^{N\times N}$ is the DFT matrix. The operation $\tau_k : \mathbb{C}^{N\times N\times N} \to \mathbb{C}^{N\times N\times N}$ indicates the transposition of the slices along piling direction $k \in \{1, 2, 3\}$ of the operand tensor. In this specific case, this is applied along direction $k = 2$.

In the implementation, the calculation of the transform via equation (4) can be performed in three stages [17]. In the first stage, the tensor-matrix multiplication of the input data with the DFT matrix is carried out as per Algorithm 1,

$$\dot{Y} = \rho_R(X, C). \tag{5}$$

In the second stage, a similar procedure is performed, this time using the output of the first stage, and as per Algorithm 2,

$$\ddot{Y} = \rho_L(\dot{Y}, C). \tag{6}$$

In the third stage, the piling direction for the tensor-matrix multiplication changes from 1 to 3. Consequently, in order to ensure a contiguous memory layout for the subsequent multiplication, a preliminary step must be performed in which $\ddot{Y}$ is subjected to the transpose operation represented by τ_2. After this, the final tensor-matrix multiplication is carried out as per Algorithm 1. At the end, the transpose operation τ_2 is applied once more to arrange the result in the same spatial layout

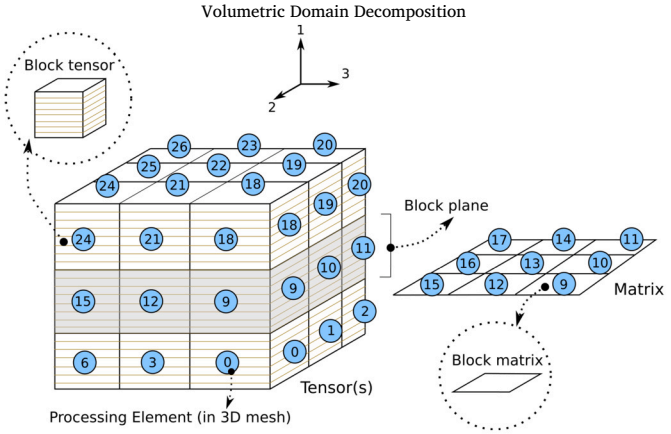


Fig. 2. A tensor and a matrix are broken down into p^3 and p^2 blocks (here, $p = 3$), respectively. Each block is located in the memory of its corresponding PE, as indicated by the number in the circles.

Table 1
Notation used in Algorithms 3, 4, 5, 6 and 7. See Fig. 2 for a schematic representation of the PE mesh and the corresponding locally allocated block tensors and block matrices. More details are provided in the text.

PE(i,j,k)	A PE located at indices (i,j,k) in the cubic mesh
$X^{(b)}_{i,j,k}$	Locally allocated operand block tensor
$Y^{(b)}_{i,j,k}$	Locally allocated result block tensor
$C^{(b)}_{i,j}$	Locally allocated DFT block matrix
$X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$	Tensor matrix multiplication (Algorithm 1)
$C^{(b)T}_{j,k} * X^{(b)}_{i,j,k}$	Tensor matrix multiplication (Algorithm 2)

as the input tensor X. Putting everything together, the third and final stage implements the following operations:

$$Y = \tau_2(\rho_R(\tau_2(\ddot{Y}), C)). \tag{7}$$

The tensor Y contains the result of the forward DFT operation of equation (4).

2.2. Adaptation of Cannon's algorithm for block tensor-matrix multiplication

In this subsection, we provide the designs of the procedures which make use of the basic idea of Cannon's algorithm [20] to perform the operations of equations (5), (6), (7), resulting in three unique distributed-memory block tensor-matrix multiplication algorithms.

In order to enable the use of as many Processing Elements (PEs) as possible, the tensors X and Y from equation (4) are subjected to the volumetric domain decomposition [13,16,17], resulting in a cubic mesh comprising p^3 PEs (see Fig. 2). Each PE(i,j,k) is accessible via indices $0 \le i, j, k < p$, and has locally allocated operand and result block tensors $X^{(b)}_{i,j,k}, Y^{(b)}_{i,j,k} \in \mathbb{C}^{b\times b\times b}$, and an operand block matrix $C^{(b)}_{j,k} \in \mathbb{C}^{b\times b}$, which is obtained by matrix decomposition. The block size is given by $b = N/p$.

Fig. 2 is a schematic representation of volumetrically decomposed data with $p = 3$. In this example, it is shown how a tensor can be decomposed into p^3 block tensors, each of which is associated with and contained in the memory of an independent PE. Moreover, it can be seen how an operand matrix can be distributed to PEs in each block plane by decomposing it into p^2 block matrices and assigning each block to a PE. A pictorial definition of the block plane is also provided. To help the reader, in Table 1 we report the notation used in the following discussion of the algorithms to indicate the key elements depicted in Fig. 2.

Algorithm 3 Procedure of the adapted Cannon's algorithm for the function $Y = \rho_R(X, C)$. See Table 1 for definitions.

▷ Alignment phase:
1: **if** $j \neq 0$ **then**
2: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-j) \bmod p)$
3: **end if**
4: **if** $k \neq 0$ **then**
5: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-k) \bmod p,k)$
6: **end if**
▷ Computation phase:
7: **for** $r \leftarrow 0$ to p **do**
8: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
9: **if** $r < p-1$ **then**
10: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-1) \bmod p)$
11: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
12: **end if**
13: **end for**

We will start by focusing on the block tensor-matrix multiplication involved in the first stage, given by equation (5). From the description of ρ_R in Algorithm 1 we recall that the slices of X along piling direction 1 are to be multiplied with C (see also Fig. 1a). Analogously, here, the block *tensors* in each plane orthogonal to piling direction 1 are to be multiplied with C. The multiplication of each such block plane with C can be executed independently, and hence, parallely. It is for this multiplication that we can utilize a scheme similar to that of the original Cannon's algorithm, with the essential deviation that one of the operand matrices and the result matrix are replaced by operand and result tensors, respectively. The adapted algorithm is composed of an alignment phase and a computation phase, with a total of p communication events. We define here a communication event as a set of multiple, parallel message passing calls.[5] The alignment phase consists of a single communication event involving $2p^2(p-1)$ parallel messages, whereas the computation phase consists of $p-1$ communication events, each involving $2p^3$ parallel messages respectively.

The algorithm for ρ_R is as outlined in Algorithm 3. In the alignment phase, the block tensors $X^b_{i,j,k}$ and block matrices $C^b_{i,j}$ are redistributed to different PEs following the rules detailed in lines 1–6 of the procedure. In the computation phase, the operation in line 8, which we refer to as the local update, can be performed slice-wise as per Algorithm 1, either sequentially, or, when additional computational resources are available to each PE, using a separate mode of parallelism, giving rise to multi-level parallelism. Additionally, the communication event (lines 10 and 11) and the local update can be executed in parallel. At the end of the computation phase, the block tensor $Y^{(b)}_{i,j,k}$ located in each PE contains the block result of the desired product. We observe here that the communication of block tensors (line 10) occurs between PEs that are neighbours along direction 3 of the PE mesh.

The algorithm for the multiplication of the second stage, given by equation (6), is outlined in Algorithm 4. It is very similar to that of the first stage, albeit with minor changes in the pattern of communication. Here, it is worth highlighting the fact that the communication of block tensors (line 8) occurs between PEs that are aligned along direction 2 of the PE mesh.

The algorithm for the third stage needs to incorporate the transpose operations represented by τ_2 in equation (7). Although these global transpositions are unavoidable, we can avoid communication events by transposing the PE mesh instead of the data itself. All the same, a local (i.e., not involving communication) data transpose function is still required to perform the transposition of the operand and result block tensors. The procedure of the adapted Cannon's algorithm for the third stage is provided in Algorithm 5. Note that, in this case, the communication of block tensors (line 9) occurs between PEs that are aligned along direction 1 of the PE mesh.

[5] Conceptually, parallel messages are passed independently and simultaneously by all PEs and thus the duration of a single communication event is decided by the most time-consuming message passing.

Algorithm 4 Procedure of the adapted Cannon's algorithm for the function $Y = \rho_L(X, C)$. See Table 1 for definitions.

▷ Alignment phase:
1: **if** $k \neq 0$ **then**
2: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-k) \bmod p,k)$
3: **end if**
4: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,k,(j+p-k) \bmod p)$
▷ Computation phase:
5: **for** $r \leftarrow 0$ to p **do**
6: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + C^{(b)T}_{j,k} * X^{(b)}_{i,j,k}$
7: **if** $r < p-1$ **then**
8: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
9: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-1) \bmod p)$
10: **end if**
11: **end for**

Algorithm 5 Procedure of the adapted Cannon's algorithm for the function $Y = \tau_2(\rho_L(\tau_2(X), C))$. See Table 1 for definitions.

▷ First local transposition:
1: $X^{(b)}_{i,j,k} \leftarrow \tau_2(X^{(b)}_{i,j,k})$
▷ Alignment phase:
2: **if** $j \neq 0$ **then**
3: Send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}((i+p-j) \bmod p,j,k)$
4: **end if**
5: Send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(k,(j+p-k) \bmod p,i)$
▷ Computation phase:
6: **for** $r \leftarrow 0$ to p **do**
7: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
8: **if** $r < p-1$ **then**
9: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}((i+p-1) \bmod p,j,k)$
10: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
11: **end if**
12: **end for**
▷ Final local transposition:
13: $Y^{(b)}_{i,j,k} \leftarrow \tau_2(Y^{(b)}_{i,j,k})$

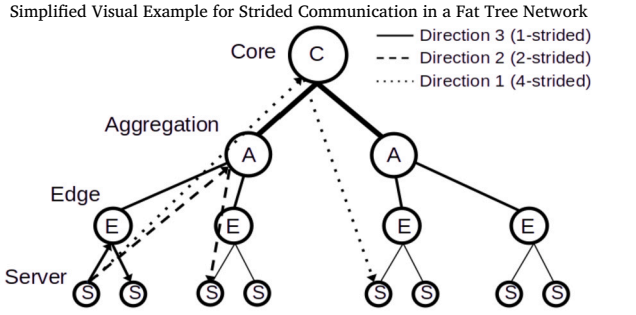


Fig. 3. Neighbours along directions 3, 2 and 1 in the PE mesh (not shown here) are 1-strided, 2-strided and 4-strided, respectively. This means that messages between neighbours along directions 3, 2 and 1 must cross 1, 3 and 5 switches, respectively. This leads to different communication rates.

2.3. Global transpose variants

Our performance analysis of the distributed-memory block tensor-matrix multiplication functions (see subsection 5.2) showed that the direction along the PE mesh in which point-to-point communication between neighbours takes place crucially decides the time of communication. This is due to how the communication patterns of the algorithms are mapped into subsets of the fat tree topology. To understand this, let us consider a simplified visual example with $p = 2$ and a fat tree network with two servers per edge, as shown in Fig. 3. In the PE mesh, neighbours along all three directions are equidistant pairs. However, in

Algorithm 6 Procedure of the global transpose variant of the adapted Cannon's algorithm for the function $Y = \rho_L(X, C)$. See Table 1 for definitions.

▷ First local transposition:
$X^{(b)}_{i,j,k} \leftarrow \tau_1(X^{(b)}_{i,j,k})$
▷ Alignment phase:
send $X^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,k,(j+p-k) \bmod p)$
if $k \neq 0$ **then**
  send $C^{(b)}_{j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,(j+p-k) \bmod p,k)$
end if
▷ Computation phase:
for $r \leftarrow 0$ to p **do**
  $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
  if $r < p-1$ **then**
    send $X^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,j,(k+p-1) \bmod p)$
    send $C^{(b)}_{j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,(j+p-1) \bmod p,k)$
  end if
end for
▷ Global transpose:
send $Y^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,k,j)$
▷ Final local transposition:
$Y^{(b)}_{i,j,k} \leftarrow \tau_1(Y^{(b)}_{i,j,k})$

Algorithm 7 Procedure of the global transpose variant of the adapted Cannon's algorithm for the function $Y = \tau_2(\rho_L(\tau_2(X), C))$. See Table 1 for definitions.

▷ First local transposition:
$X^{(b)}_{i,j,k} \leftarrow \tau_2(X^{(b)}_{i,j,k})$
▷ Alignment phase:
Send $X^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(k,j,(i+p-j) \bmod p)$
if $k \neq 0$ **then**
  Send $C^{(b)}_{j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,(j+p-k) \bmod p,k)$
end if
▷ Computation phase:
for $r \leftarrow 0$ to p **do**
  $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
  if $r < p-1$ **then**
    send $X^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,j,(k+p-1) \bmod p)$
    send $C^{(b)}_{j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(i,(j+p-1) \bmod p,k)$
  end if
end for
▷ Global transpose:
send $Y^{(b)}_{i,j,k}$: $\mathrm{PE}(i,j,k) \rightarrow \mathrm{PE}(k,j,i)$
▷ Final local transposition:
$Y^{(b)}_{i,j,k} \leftarrow \tau_2(Y^{(b)}_{i,j,k})$

the network, depending on the stride between the neighbours, messages may cross 1, 3 or 5 switches, which leads to three possible, distinct communication rates.

In practice, similarly, if p is large enough in relation to the number of nodes on the server level in the allocation, up to three distinct communication rates can be realized. In our tests, we observed that the communication rate is the highest for direction 3 in the PE mesh, followed by direction 2 and finally, direction 1.

From Algorithms 3, 4 and 5, it can be seen that the computation phase involves $p-1$ more communication events than the alignment phase, a number which scales linearly with the node count. To limit the communication to direction 3 as much as possible, we implemented variants of Algorithms 4 and 5, in which the data is globally transposed in the alignment phase such that, in the computation phase, communication only occurs between neighbours along direction 3. This strategy has additional costs: (i) both procedures include an additional communication event after the computation phase, in which the transposed data is transposed back; (ii) in both procedures, local data transpositions become necessary before the alignment phase and after the computation phase. The procedures for these variants are provided in Algorithms 6 and 7.

2.4. Modeling the overlapping of the communication and the computation

In the computation phase of all the distributed-memory algorithms discussed in the previous subsections, the communication event and the local update can be overlapped to hide the latency of communication. To identify configurations of block size and node count for which the highest efficiency of a block tensor-matrix multiplication algorithm can be attained, it is useful to separately model the time of the communication event and the time of the local update in the problem size range of interest, as a function of the block size. To this end, microbenchmark programs that exactly imitate the communication event and the local update can be used. The obtained times can then be fitted as polynomials. The points of intersection of the two fitted curves fulfill the condition of maximum efficiency of the overlapping under which the algorithm does not incur any communication overhead. This is also the case in block size intervals in which the time of the local update is always greater than that of communication. We used this approach to select the configurations of block size and node count in our performance analysis in subsection 5.2.

3. Details of implementation

The 3D DFT by block tensor-matrix multiplication algorithm was implemented as a C++ library named S3DFT, which has a distributed-memory Application Programming Interface (API – the set of functions which are to be used to compute the 3D DFT) in both single and double precision. The library was built using the Intel compiler and the Intel MKL and Intel MPI libraries, which are part of the Intel OneAPI v2021.4.0 toolkit suite. It is open-source software available for use under the GNU Lesser General Public License v3 (LGPL) [24].

S3DFT combines the use of shared- and distributed-memory parallelism by means of an OpenMP/MPI hybrid approach. On the shared-memory level, the computational work is identified by a set of slices of the operand tensor. In all shared-memory functions, we use a custom work-sharing function, which (provided the number of threads is constant) always returns the same set of indices of operand tensor slices for any given thread. This tactic ensures that the association between operand data and threads does not change. Combined with proper runtime thread pinning, it strongly reduces the number of non-local memory accesses in NUMA systems, which are significantly slower than local memory accesses. This design makes sure that S3DFT can be run with threads distributed across NUMA domains without a performance drop. The excellent scaling of S3DFT across NUMA domains is demonstrated by the performance analysis presented in subsection 5.1.

Initial profiling of the distributed-memory block tensor-matrix multiplication functions that implement Algorithms 3, 4 and 5 (see subsection 5.2) revealed that the time of communication *significantly exceeds that of the local update*, which confirms our initial speculation that the communication (and not the computation) would be the bottleneck. To hide the latency of communication, we decided to dedicate one thread to communication. Our work-sharing function fulfills this requirement by returning an empty set for the communicating thread and distributing the slices of the operand tensor amongst the remaining threads in a round-robin fashion.

The local update mentioned in subsection 2.2 has been realized by a shared-memory tensor-matrix multiplication function, which computes the product as a set of parallel matrix-matrix multiplications as shown in Algorithms 1 and 2. For this, we used the CBLAS implementation provided by the Intel MKL v2021.4.0. We see from the strong scaling of the function (Fig. 4b) that the difference in performance when all 48 cores are used and when 47 cores are used is small, with the performance decreasing only slightly from 88% to 85% of the single node peak performance. Hence, the use of a dedicated communication thread does not reduce the performance of the local update much.

Table 2
Specifications of the standard compute node of the JUWELS cluster.

Processor	Intel Xeon® Platinum 8168 (Skylake)
CPU count	2 (sockets)
Core count	48 cores (24 cores per CPU)
SMT/HT	Available, 96 threads (48 threads per CPU)
Clock frequency	[1.2–3.7 GHz], base @ 2.7 GHz
Cache	L1 - 32 kB, L2 - 1 MB, L3 - 33 MB
DRAM	96 GB DDR4 @ 2666 MHz

Table 3
Cache and memory bandwidths according to Intel Advisor.

	Bandwidth	
	1x NUMA	2x NUMA
L1	11.6 TB/s	23.2 TB/s
L2	5.5 TB/s	11.0 TB/s
L3	649 GB/s	1299 GB/s
DRAM	115 GB/s	230 GB/s

4. Specifications of the JUWELS cluster

All benchmarks and tests in this work have been executed on the JUWELS Cluster [19], which uses a fat tree network with InfiniBand interconnects. The standard compute node has two cache-coherent NUMA domains. The salient specifications are listed in Table 2, and the peak bandwidths as obtained with the Intel Advisor tool [25], in Table 3. These numbers were used as reference values while analysing the performance of the core functions of the implementation.

Since Intel did not explicitly provide information on peak performance in terms of FLOP/s at the time of writing of this article [26], the peak performance of the compute node had to be estimated using the published specifications. The base frequency of the Intel Xeon® Platinum 8168 processor is 2.7 GHz [27]. However, when all cores are active and the use of AVX-512 instructions is maximized, the clock frequency drops to 2.5 GHz [27]. The processor is equipped with 2 AVX-512 Fused Multiply-Add (FMA) units per core, which yields a theoretical peak performance of 3840 GFLOP/s. Corroborating this estimation, Intel Advisor's roofline chart includes information about the double-precision FMA peak performance [25], which in this case is 3812 GFLOP/s. Henceforth, we refer to this value when we speak about the peak performance of the node.

5. Performance analysis of components

In this section, we present the performance analysis of the core functions of the S3DFT implementation. For this analysis, the functions were microbenchmarked on the JUWELS Cluster in double precision. The open-source library TiXL, which is available under the LGPL v3, was used for this purpose [28].

The microbenchmark programs consist of (i) an initialization phase, in which operand/result data are allocated afresh, and each OpenMP thread (excepting the communication thread) accesses the first word of each memory page of its associated data – thereby ruling out the measuring of page-faults, (ii) an experiment phase, in which the function of interest is run, and (iii) a clean-up phase, in which all data is freed. Each benchmark test was concluded by performing 20 warm-up and 100 timed runs. The experiment phase of only the latter runs was used to measure the duration of the function of interest. The result was calculated as the geometric mean of these measurements.

Via mircobenchmarking, we present a breakdown of the run time of the main API function in Table 4 for node count 343 (i.e., $p = 7$) and problem size $N = 4200$. Since the shared-memory tensor zeroing and transpose functions constitute a negligible fraction of the total run time, they have been excluded from the section. For completeness, a performance analysis of the local transpose function is still reported in Appendix A.

Table 4
Breakdown of S3DFT's main API function for $p = 7$ and $N = 4200$.

Distributed-Memory Function	Run Time Percentage
Block Tensor-Matrix Multiplication (Algorithm 3)	21%
Transp. Block Matrix-Tensor Multiplication (Algorithm 4)	35%
Transp. Block Tensor-Matrix Multiplication (Algorithm 5)	44%
Shared-Memory Function	
Tensor-Matrix Multiplication, Transp. Matrix-Tensor Multiplication	9%
Set Zero Tensor	0.8%
Transpose Tensor	1%

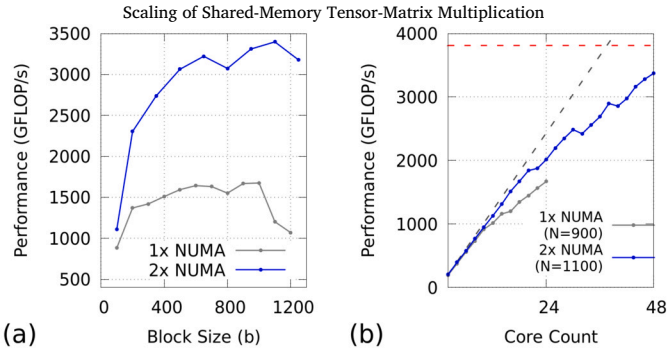


Fig. 4. Subfigures show (a) problem scaling as a function of block size and (b) strong scaling of the shared-memory tensor-matrix multiplication. Grey and blue curves are for single (24 cores) and dual (48 cores) NUMA configurations, with $N = 900$ and $N = 1100$, respectively. The red line in subfigure (b) indicates the peak performance of the single node, while the grey dashed line indicates ideal linear scaling.

5.1. Shared-memory tensor-matrix multiplication

This function performs the local update operation of the distributed-memory tensor-matrix multiplication as mentioned in subsection 2.2.

We began the analysis by running problem scaling tests using all available cores in the single and dual NUMA configurations to identify the problem sizes at which peak performances of the function can be expected. Next, we conducted strong scaling tests for these problem sizes. The results are reported in Fig. 4. We observed a peak performance of 1671 GFLOP/s at $N = 900$ and of 3373 GFLOP/s at $N = 1100$ for the single and dual NUMA configurations, respectively. This corresponds to 88% of the peak performance. As shown in the right panel of Fig. 4, the function scales well.

To estimate the corresponding effective bandwidth,[6] we must first model the traffic and computation requirements of Algorithm 1. For a tensor of side N, a computer can perform $8N^4$ floating point operations[7] after $2N^3(N+1)$ transfers. For double precision, we have the code balance given by $B_c = \frac{4(N+1)}{N} \approx 4$ B/FLOP. Using the roofline model, we can calculate the effective bandwidth as $b_s = B_c P$, where P is the attained performance [29, p. 66]. Following this, we can estimate peak effective bandwidths $b_s = 6.7$ TB/s and $b_s = 13.5$ TB/s for the single and dual NUMA configurations, respectively, which are greater than the corresponding L2-cache bandwidths listed in Table 3. This can be taken to conclude that the function makes excellent use of caching.

[6] The effective bandwidth is calculated using the run time and the *a priori* data traffic estimation.

[7] Each complex number addition and multiplication involves at least 2 and 6 FLOPs respectively.

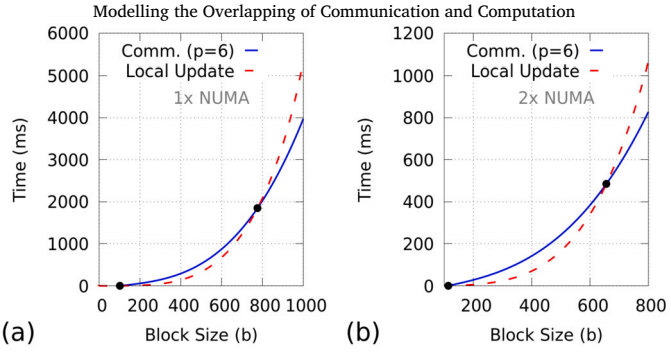


Fig. 5. Fitted curves model the time of the communication event (continuous lines) and that of the local update (dashed line) as functions of block size. The black dots indicate the intersections at which perfect overlapping can be expected. Subfigures (a) and (b) report results obtained in 2 MPI tasks/node and 1 MPI task/node configurations, respectively.

5.2. Distributed-memory block tensor-matrix multiplication

In this analysis, we consider the implementations of Algorithms 3, 4 and 5.

First, following the approach outlined in subsection 2.4, we identified the configurations of block size and node count for which the highest efficiency of the Algorithm 3 can be reached. For this purpose, we designed microbenchmark programs which exactly imitate the communication event and the local update, respectively, and ran them for $p = 2, 3, 4, 5, 6$ and $b \in [100, 1300]$. As shown in Fig. 5, we then fitted the obtained times with cubic polynomials to model the time of the communication event for the node count of interest (continuous curve), and a quartic polynomial for that of the local update (dotted curves). The optimal block sizes for the above-stated node counts were found to be $b > 750$ for the 2 MPI tasks/node (single NUMA) and $b > 600$ for the 1 MPI task/node (dual NUMA) configurations, respectively.

Next, we conducted strong scaling tests for the Algorithms 3, 4 and 5 in the 1 MPI task/node configuration for the problem size $N = 4200$ and $p = 4, 5, 6, 7$, corresponding to block sizes $b = 1050, 840, 700, 600$, at which the latency of the communication event is expected to be completely hidden by the overlapping (see Fig. 5b). The results are reported in Fig. 6, showing parallel efficiencies[8] in the ranges of 81%–95%, 58%–77% and 51%–68% for Algorithms 3, 4 and 5, respectively.

To understand the poor scaling of Algorithms 4 and 5, we microbenchmarked the code of the communication event in the computation phase of the three block tensor-matrix multiplication algorithms. As shown in Fig. 7, the results revealed that the communication rate varies considerably with the direction along which the neighbours are identified, as the node count is increased. Interestingly, the communication rate between neighbours along direction 3 in the PE mesh is the highest, followed by direction 2 and finally, direction 1. As noted in subsection 2.2, in the computation phase of the Algorithms 3, 4 and 5, most of the communication occurs between neighbours along directions 3, 2 and 1, respectively. This explains the scaling performance of each algorithm.

Finally, on another note, we observe that the performance of the local update reduces with reducing block size (see the left panel of Fig. 4), which warns us of a reduction in the shared-memory resource utilization as the node count scales up, which predicts an additional cause for the worsening of the strong-scaling performance of the above-mentioned algorithms.

[8] Here, the parallel efficiency has been evaluated relative to $p = 4$ case (i.e., with 64 nodes) which is the minimum number of nodes we could use for the given problem size due to memory limitations.

5.3. Global transpose variants

In an attempt to overcome the problem discussed in the previous section, we implemented variants of Algorithms 4 and 5 in which the transposition of the PE mesh is replaced by the global transposition of the data, as outlined by Algorithms 6 and 7. This resulted in improvements in the strong-scaling performance (see red curves in Figs. 6b and 6c), with parallel efficiencies of 65%–81% and 70%–95% for Algorithms 6 and 7, respectively. However, the overall performance was found to suffer considerably due to the additional communication events required.

6. Comparison with Intel MKL and FFTW3

Here, we present the results of the benchmarking of S3DFT against two competitive 3D FFT implementations: Intel MKL v2021.4.0[9] and FFTW3 v3.3.10. The benchmarking procedure is identical to that outlined in section 5, with the exception that 70 warm-up runs and 50 timed experiments were conducted. In the plots included in this subsection, we report the geometric mean of the run time. In the programs that recorded the performance of the cluster-based Intel MKL/FFTW3 libraries, multithreading was initialized as per the manual [30]. The FFTW-plan [11] was created in the initialization phase of the program using the flag FFTW_MEASURE. To be able to benchmark under reproducible conditions, contiguous node allocation was requested. Within a single compute node, a thread-placement policy of 1 thread/core was applied. Further, each thread was pinned to avoid thread migrations by the operating system during runtime.

Initial testing showed that both Intel MKL and FFTW3 performed much better when launched with 2 MPI tasks/node, which corresponds to 1 NUMA domain/MPI task. Therefore, their benchmarking programs were always run using this configuration. S3DFT was found to perform similarly in both the 1 MPI task/node and 2 MPI tasks/node configurations.

In what we call small problem scale, strong scaling comparisons were conducted for problem sizes $N = 120, 240, 480, 600, 840$. Similarly, in the large problem scale, we did the strong scaling comparisons for problem sizes $N = 2520, 3360, 4200$. Moreover, these comparisons were conducted two-fold, with S3DFT in the 1 MPI task/node and 2 MPI tasks/node configurations. In all these comparisons, we observed similar scaling behaviours and performances for all investigated problem sizes. The results for sizes $N = 840, 3360$ are provided in Figs. 8 and 9.

Results show that Intel MKL was consistently the fastest across all problem sizes and node counts, followed by FFTW3. On average, in the small problem scale and with S3DFT in the 2 MPI tasks/node configuration, Intel MKL was 2.5 times faster and FFTW3 was 1.8 times faster than S3DFT. With S3DFT in the 1 MPI task/node configuration, Intel MKL was 2.2 times faster and FFTW3 was 1.5 times faster than S3DFT. Frequently, for small node counts, S3DFT was found to be slightly faster than FFTW3 (see, e.g., Fig. 8a). In the large problem scale, Intel MKL was 3.7 times faster and FFTW3 was 2.3 times faster than S3DFT.

The benchmarking was repeated for the same problem scales, sizes, node counts and MPI tasks/node configurations with the implementations of Algorithms 4 and 5 being replaced by the global transpose variants (GTV) 6 and 7. As can be seen in Figs. 10 and 11, the result of the comparison with FFTW3 and Intel MKL remained essentially unchanged, even in the large problem scale, despite the significant improvement of S3DFT's scaling. Specifically, in the small problem scale and with S3DFT in the 2 MPI tasks/node configuration, Intel MKL was 3.0 times faster and FFTW3 was 2.2 times faster than S3DFT. With S3DFT in the 1 MPI task/node configuration, Intel MKL was 2.4 times faster and FFTW3 was 1.7 times faster than S3DFT. In the large problem

[9] We made use of the convenient FFTW3 wrapper interface provided by Intel MKL, which makes use of its implementation of cluster FFT functions.

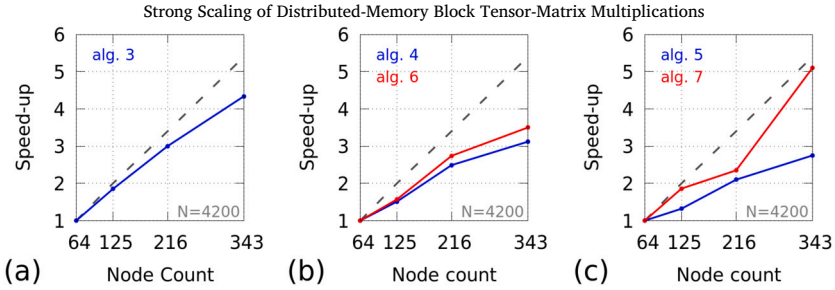


Fig. 6. The blue curves in subfigures (a), (b) and (c) show the strong scaling performance of the implementations of Algorithm 3, Algorithm 4 and Algorithm 5, respectively, at problem size $N = 4200$, in the 1 MPI task/node configuration. The grey dashed lines indicate ideal linear scaling. The red curves in subfigures (b) and (c) show the results obtained using the global transpose variants 6 and 7, respectively.

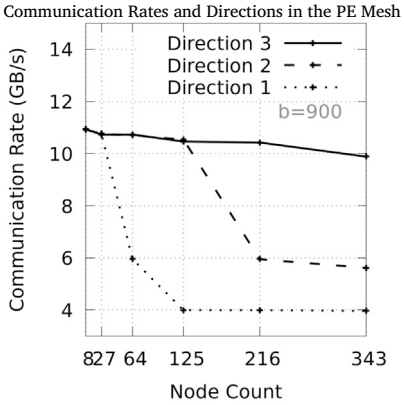


Fig. 7. Communication rate as a function of node count with $b = 900$. The continuous, dashed and dotted lines correspond to the communication rates of neighbours along directions 3, 2 and 1, respectively.

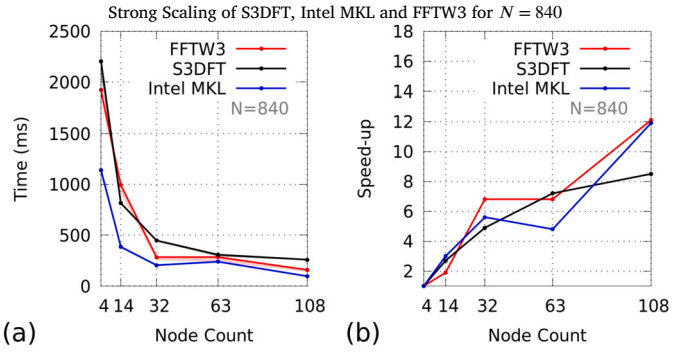


Fig. 8. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node.

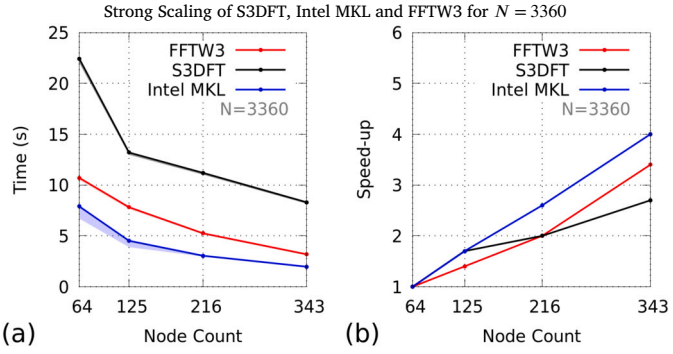


Fig. 9. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node.

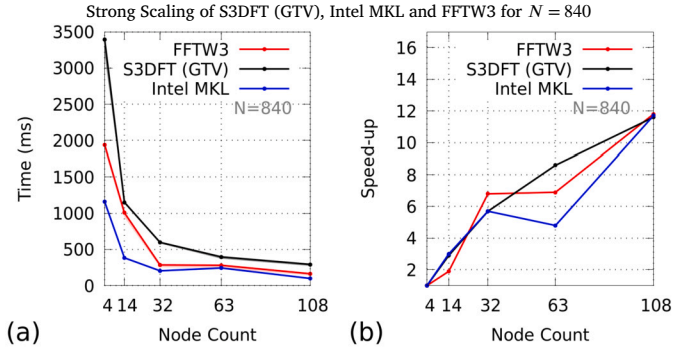


Fig. 10. In this comparison, S3DFT uses the global transpose variants as detailed in subsection 2.2. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node.

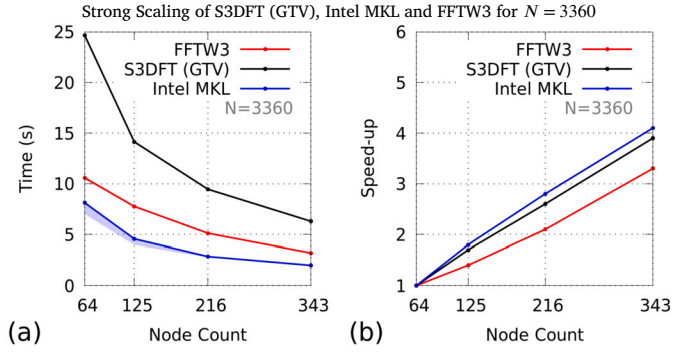


Fig. 11. In this comparison, S3DFT uses the global transpose variants as detailed in subsection 2.2. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node.

scale, Intel MKL was 3.6 times faster and FFTW3 was 2.2 times faster than S3DFT.

We note that other parallel 3D FFT implementations might have higher scaling limits than FFTW3 and Intel MKL. This is to be expected especially for libraries that make use of the pencil decomposition, such as FFTK [15], for which scaling up to 196,608 cores has been demonstrated. A comparison of S3DFT with FFTK is provided in Appendix B. Briefly, FFTK was found to be on average 4 times faster than S3DFT.

7. Conclusions

We have presented a new parallel algorithm that exploits block tensor-matrix multiplication to compute the 3D DFT of a volumetrically decomposed, cubic domain using point-to-point communication. The algorithm has been implemented as a C++ library called S3DFT capable of utilizing shared-memory parallelism across multiple NUMA domains

within a single node and distributed-memory parallelism across multiple nodes. In the process, we designed, developed and tested three adapted variants of Cannon's algorithm. These adaptations enable the use of tensor operands, realize multi-level parallelism and make efficient use of the technique of overlapping computation and communication with the help of a custom work-sharing function for OpenMP-based multithreading. S3DFT has been optimized for the JUWELS Cluster and its core functions have been analyzed to show their efficiency. Finally, the performance of S3DFT has been compared with those of well-established, widely-used libraries for a wide range of problem sizes.

Our analysis by microbenchmarking has shown that S3DFT has a highly efficient implementation on the shared-memory level. On the other hand, we found that while one of the three required distributed-memory block tensor-matrix multiplication algorithms scales excellently, the other two scale poorly. We identified the origin of this problem in the mapping of their communication patterns in subsets of the fat tree topology: this result exposes a potential drawback of running block-wise parallel algorithms on systems with fat tree networks, which is caused by increased communication latencies along specific Cartesian directions in the PE mesh.

In an effort to improve the scalability of S3DFT, we designed algorithmic variants of the poorly scaling components in which the majority of the communication occurs between PEs aligned along the Cartesian direction in which the communication was found to be the fastest. This was made possible by performing global data transpositions instead of transposing the PE mesh, which comes at the cost of additional communication events. Although these variants improved the strong-scaling performance of S3DFT, its overall performance did not change considerably.

Our results let us speculate that at the current stage, the 3D DFT by block tensor-matrix multiplication is not a viable alternative to modern FFT-based approaches on computer clusters using fat tree networks. Different results might be expected on different computer clusters. For example, it is possible that S3DFT performs and scales considerably better on a computer cluster based on a network topology which is isomorphic to the PE mesh. Unfortunately, we did not have access to such a system to verify this hypothesis.

CRediT authorship contribution statement

Nitin Malapally: Data curation, Formal analysis, Investigation, Conceptualization, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Viacheslav Bolnykh:** Supervision, Writing – review & editing, Conceptualization. **Estela Suarez:** Writing – review & editing, Supervision. **Paolo Carloni:** Funding acquisition, Supervision, Writing – review & editing, Resources. **Thomas Lippert:** Funding acquisition, Writing – review & editing. **Davide Mandelli:** Project administration, Supervision, Writing – original draft, Writing – review & editing.

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.

Data availability

All software is open source and links to public gitlab projetcs are provided in the manuscript. Information about the data presented will be gladly provided upon request.

Acknowledgments

PC and DM acknowledge funding from the Helmholtz European Partnership program "Innovative high-performance computing approaches for molecular neuro-medicine". PC acknowledges funding from the Human Brain Project (EU Horizon 2020). This research was supported by the Helmholtz Joint Lab "Supercomputing and Modeling for the Human Brain". The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS [19] at Jülich Supercomputing Centre (JSC). Open access publication fee funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) (491111487). NM thanks the support team at JSC, especially Ilya Zhukov, and Rolf Rabenseifner for all the helpful discussions and suggestions.

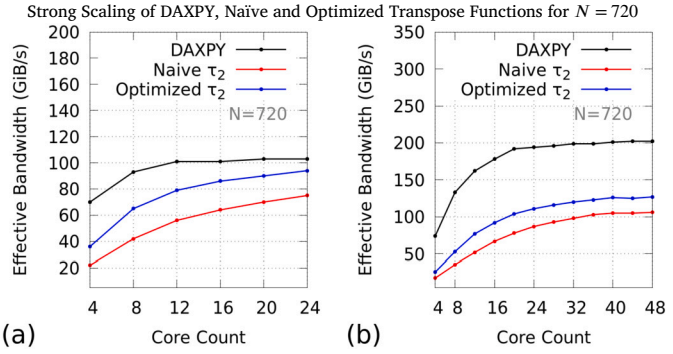


Fig. A.12. Subfigures show (a) strong scaling in single NUMA configuration and (b) strong scaling in dual NUMA configuration. In both subfigures, $N = 720$.

Appendix A. Performance analysis of the transpose function

Here, we take a closer look at the performance of our implementation of the local transpose function, τ of equation (7). The transpose function can be viewed as a streaming function because in a perfect implementation it would closely resemble a copy operation. Thus, one way to assess the performance of our implementation is to compare it to that of a suitably similar streaming function having an excellent memory bandwidth utilization. As a reference, we decided to use the DAXPY loop, by measuring the performance of the operation $Y = Y + aX$ on the target computer, where $Y, X \in \mathbb{R}^N$ are double-precision arrays and $a \in \mathbb{R}$ is a double-precision scalar. We performed the microbenchmarking with $N = 720$, at which size the performance of the loop was found to saturate. The black lines in Fig. A.12 illustrate the increase of the effective bandwidth of this reference loop as a function of the number of cores, for the single as well as dual NUMA configurations. The recorded corresponding peak performances are 103 GB/s and 202 GB/s, respectively. We used these values as reference to measure the efficiency of our implementation of the transpose function.

The performance of the naïve implementation of the transpose function is shown by the red curves in Fig. A.12 for the size $N = 720$. Building on it, we improved the cache utilization by applying loop-blocking with the help of an intermediate array so small as to fit into the cache. The optimal blocking size was experimentally found to be 36 kiB. Upon optimization, only a small improvement in performance could be observed, as shown by the blue curves in Fig. A.12. Indeed, we found the performance of the näive implementation to be quite high, which we attributed to the size of the processor's L3-cache, by virtue of which good cache-line reuse can be achieved even for relatively large matrix sizes.

In the single NUMA configuration, the optimized transpose function attained a peak efficiency of 91% as compared to that of 73% of the naive function. However, we note that the efficiencies drop to 63% and 53% respectively, when the dual NUMA configuration is applied. This can be attributed to unavoidable non-local memory accesses arising from the fact that the functions make use of a different multi-threading work-sharing plan as compared to the other kernels in the implementation. Although an adaptation of the algorithm to minimize

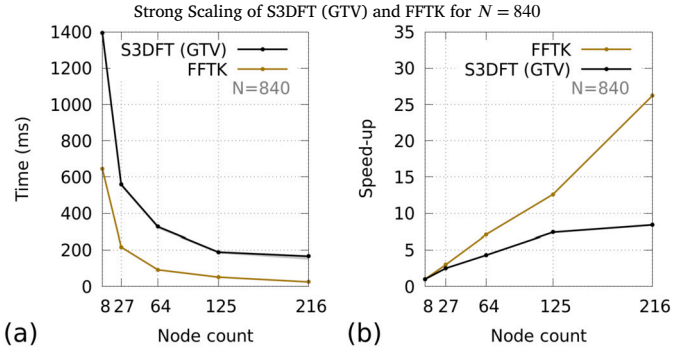


Fig. B.13. In this comparison, S3DFT uses the global transpose variants as detailed in subsection 2.2. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node.

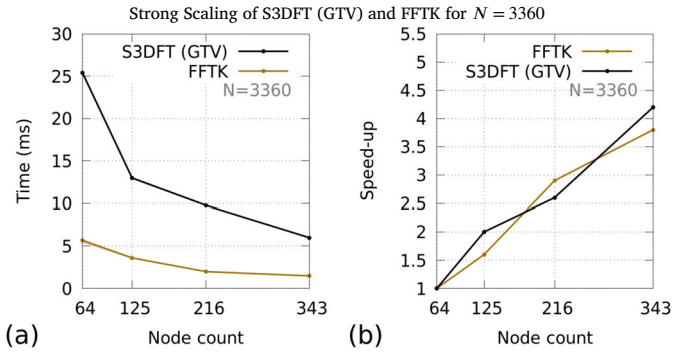


Fig. B.14. In this comparison, S3DFT uses the global transpose variants as detailed in subsection 2.2. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node.

these non-local memory accesses is conceivable, the expected performance gain did not justify its design and implementation within the scope of this study.

Appendix B. Comparison with FFTK

Figs. B.13 and B.14 show the benchmarking results of S3DFT (using the global transpose variants described in subsection 2.2) and FFTK. In the small problem scale, with S3DFT in the 2 MPI tasks/node configuration, on average, FFTK was 3.8 times faster than S3DFT. In the large problem scale, with S3DFT in the 1 MPI task/node configuration, on average, FFTK was 4.3 times faster than S3DFT.

References

[1] A. Campa, T. Dauxois, S. Ruffo, Statistical mechanics and dynamics of solvable models with long-range interactions, Phys. Rep. 480 (3) (2009) 57–159, https://doi.org/10.1016/j.physrep.2009.07.001.
[2] H. Yildirim, J. Matos, A. Kara, Role of long-range interactions for the structure and energetics of olympicene radical adsorbed on au(111) and pt(111) surfaces, J. Phys. Chem. C 119 (45) (2015) 25408–25419, https://doi.org/10.1021/acs.jpcc.5b08191.
[3] B. Kohnke, C. Kutzner, H. Grubmüller, A gpu-accelerated fast multipole method for gromacs: performance and accuracy, J. Chem. Theory Comput. 16 (11) (2020) 6938–6949, https://doi.org/10.1021/acs.jctc.0c00744, pMID: 33084336.
[4] M. Tuckerman, Statistical Mechanics: Theory and Molecular Simulation, Oxford Graduate Texts, OUP, Oxford, 2010, https://books.google.de/books?id=Lo3Jqc0pgrcC.
[5] B. Raghavan, M. Paulikat, K. Ahmad, L. Callea, A. Rizzi, E. Ippoliti, D. Mandelli, L. Bonati, M. De Vivo, P. Carloni, Drug design in the exascale era: a perspective from massively parallel QM/MM simulations, J. Chem. Inf. Model. 63 (12) (2023) 3647–3658, https://doi.org/10.1021/acs.jcim.3c00557.
[6] A.Y. Toukmaji, J.A. Board, Ewald summation techniques in perspective: a survey, Comput. Phys. Commun. 95 (2) (1996) 73–92, https://doi.org/10.1016/0010-4655(96)00016-1.
[7] M.J. Harvey, G. De Fabritiis, An implementation of the smooth particle mesh Ewald method on gpu hardware, J. Chem. Theory Comput. 5 (9) (2009) 2371–2377, https://doi.org/10.1021/ct900275y, pMID: 26616618.
[8] V. Weber, C. Bekas, T. Laino, A. Curioni, A. Bertsch, S. Futral, Shedding light on lithium/air batteries using millions of threads on the BG/Q supercomputer, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 735–744.
[9] Ariana Remmel, What exascale computing could mean for chemistry, Chem. Eng. News (2022) 29–33, https://doi.org/10.47287/cen-10031-cover.
[10] D. Rockmore, The fft: an algorithm the whole family can use, Comput. Sci. Eng. 2 (1) (2000) 60–64, https://doi.org/10.1109/5992.814659.
[11] M. Frigo, S. Johnson, The design and implementation of fftw3, Proc. IEEE 93 (2) (2005) 216–231, https://doi.org/10.1109/JPROC.2004.840301.
[12] D. Pekurovsky, Ultrascalable Fourier transforms in three dimensions, in: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, TG '11, Association for Computing Machinery, New York, NY, USA, 2011.
[13] A. Ayala, S. Tomov, M. Stoyanov, J. Dongarra, Scalability issues in fft computation, in: V. Malyshkin (Ed.), Parallel Computing Technologies, Springer International Publishing, Cham, 2021, pp. 279–287.
[14] T. Lippert, K. Schilling, S. Trentmann, F. Toschi, R. Tripiccione, Fft for the ape parallel computer, Int. J. Mod. Phys. C 8 (06) (1997) 1317–1334.
[15] A.G. Chatterjee, M.K. Verma, A. Kumar, R. Samtaney, B. Hadri, R. Khurram, Scaling of a fast Fourier transform and a pseudo-spectral fluid solver up to 196608 cores, J. Parallel Distrib. Comput. 113 (2018) 77–91, https://doi.org/10.1016/j.jpdc.2017.10.014, arXiv:1805.07801.
[16] J. Jung, C. Kobayashi, T. Imamura, Y. Sugita, Parallel implementation of 3d fft with volumetric decomposition schemes for efficient molecular dynamics simulations, Comput. Phys. Commun. 200 (2016) 57–65, https://doi.org/10.1016/j.cpc.2015.10.024.
[17] S. Sedukhin, T. Sakai, N. Nakasato, 3d discrete transforms with cubical data decomposition on the ibm blue gene/q, in: Proceedings of the 30th International Conference on Computers and Their Applications, CATA 2015, 2015, pp. 193–200.
[18] T. Klöffel, G. Mathias, B. Meyer, Integrating state of the art compute, communication, and autotuning strategies to multiply the performance of the application programm cpmd for ab initio molecular dynamics simulations, https://arxiv.org/abs/2003.08477, 2020.
[19] D. Alvarez, JUWELS cluster and booster: exascale pathfinder with modular supercomputing architecture at Juelich supercomputing centre, J. Large-Scale Res. Facil. 7 (10 2021), https://doi.org/10.17815/jlsrf-7-183.
[20] L.E. Cannon, A cellular computer to implement the Kalman filter algorithm, Ph.D. thesis, USA, aAI7010025, 1969.
[21] A. Gupta, V. Kumar, Scalability of parallel algorithms for matrix multiplication, in: 1993 International Conference on Parallel Processing, ICPP'93, vol. 3, 1993, pp. 115–123.
[22] J.-N. Quintin, K. Hasanov, A. Lastovetsky, Hierarchical parallel matrix multiplication on large-scale distributed memory platforms, in: 2013 42nd International Conference on Parallel Processing, 2013, pp. 754–762.
[23] T. Kolda, B. Bader, Tensor decompositions and applications, SIAM Rev. 51 (2009) 455–500, https://doi.org/10.1137/07070111X.
[24] N. Malapally, S3dft: scalable 3d-dft, Available on https://gitlab.com/anxiousprogrammer/s3dft, 2021. (Accessed 11 August 2022).
[25] Intel Corporation, Intel(R) advisor user guide, 2022nd edition, available at https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top.html, 2022.
[26] I. Corporation, Where can I find information about flops per cycle for intel(r) processors?, https://www.intel.com/content/www/us/en/support/articles/000057415/processors.html, 2021. (Accessed 11 August 2022).
[27] Intel Corporation, Intel(R) Xeon(R) processor scalable family specification update, 017th edition, available at https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf, October 2020.
[28] N. Malapally, Tixl: timed experiments in a loop, Available on https://gitlab.com/anxiousprogrammer/tixl, 2021. (Accessed 11 August 2022).
[29] G. Hager, G. Wellein, Introduction to high peformance computing for scientists and engineers, https://doi.org/10.1201/EBK1439811924, 2010.
[30] M. Frigo, G.S. Johnson, FFTW, Massachusetts Institute of Technology, available at http://www.fftw.org/fftw3.pdf, December 2020.

Nitin Malapally obtained his MSc in Computational Engineering in 2016 from the Friedrich-Alexander University of Erlangen and Nuremberg. He is currently a PhD student at the Institute of Computational Neuromedicine (IAS-5/INM-9) and the Jülich Supercomputing Center (JSC) at Forschungszentrum Jülich GmbH, Germany. His research project focuses on the design and implementation of massively parallel algorithms for atomistic molecular dynamics simulations.

Dr. Viacheslav Bolnykh obtained his Ph.D. in High Performance Computing and Theoretical Biophysics from the University of Cyprus and RWTH Aachen University, Germany. Since January 2022 he is a Senior Software Engineer at NVIDIA.

Prof. Dr. Estela Suarez is research group leader at the Jülich Supercomputing Centre at Forschungszentrum Jülich GmbH, Germany, which she joined in 2010. Since 2022 she is also Professor for High Performance Computing at the University of Bonn. Her research focuses on HPC system architectures and codesign. As leader of the DEEP project series she has driven the development of the Modular Supercomputing Architecture, including hardware, software and application implementation and validation. Additionally, since 2018 she leads the codesign efforts within the European Processor Initiative. She holds a PhD in Physics from the University of Geneva (Switzerland) and a Master degree in Astrophysics from the University Complutense of Madrid (Spain). Research interests: high performance computing (HPC), heterogeneous HPC system architectures, modular supercomputing architecture (MSA), hardware prototyping and evaluation, software environment development, co-design and application optimization.

Prof. Dr. Paolo Carloni received his diploma in Chemistry in 1990 and he completed his Ph.D. theses in Chemistry, majoring Computational Biophysics, in 1994 from the University of Florence, Italy. He is director of the Institute for Computational Biomedicine (INM-9), Institute for Neuroscience and Medicine, and of the Institute for Computational Biomedicine (IAS-5), Institute for Advanced Simulation, Forschungszentrum Jülich GmbH, Germany. He is co-director of the JARA Center for Simulation and Data Science (JARA CSD), and co-Director of the JARA-Institute for Molecular Neuroscience and Neuroimaging (INM-11) at Forschungszentrum Jülich GmbH and RWTH Aachen University. He is also co-director of Key Science Laboratory on Multiscale Simulations of Complex Systems, University of Hanoi, Vietnam and full Professor (W3) for Theoretical and Computational Biophysics at RWTH Aachen University. His research interests cover the field of Bioinformatics, Molecular Simulations, Free-Energy Calculations and Cellular Pathways Modeling. He has expertise in computer simulation of molecular processes involved in neuronal function and dysfunction and cancer progression and in the development and application of computational methods in molecular medicine and neurobiology, including bioinformatics, ab-initio, classical, coarse-grained and hybrid methods for molecular simulation.

Prof. Dr. Dr. Thomas Lippert received his diploma in Theoretical Physics in 1987 from the University of Würzburg. He completed his Ph.D. theses in theoretical physics at Wuppertal University on simulations of lattice quantum chromodynamics and at Groningen University in the field of parallel computing with systolic algorithms. He is director of the Jülich Supercomputing Centre at Forschungszentrum Jülich, member of the board of directors of the John von Neumann Institute for Computing (NIC) and the Gauss Centre for Supercomputing (GCS). He holds the chair for Modular Supercomputing and Quantum computing at Goethe University Frankfurt. His research interests cover the field of modular supercomputing, quantum computing, computational particle physics, parallel and numerical algorithms, and cluster computing.

Dr. Davide Mandelli received his diploma in Physics in 2011 from the University of Milano Bicocca. He completed his Ph.D. theses in Theory and Numerical Simulations of the Condensed Matter from the International School of Advanced Studies (SISSA) of Trieste, Italy, in 2015. Since 2022, he has been an independent researcher at the Institute of Computational Neuromedicine (IAS-5/INM-9) at Forschungszentrum Jülich GmbH, Germany. His research interest is in the design, implementation and application of HPC-oriented algorithms in the area of molecular dynamics simulations.