PLOS ONE

RESEARCH ARTICLE

Strategies of data layout and cache writing for input-output optimization in high performance scientific computing: Applications to the forward electrocardiographic problem

Louie Cardone-Noott, Blanca Rodriguez, Alfonso Bueno-Orovio*

Department of Computer Science, University of Oxford, Oxford, United Kingdom

* alfonso.bueno@cs.ox.ac.uk

OPEN ACCESS

Citation: Cardone-Noott L, Rodriguez B, Bueno-Orovio A (2018) Strategies of data layout and cache writing for input-output optimization in high performance scientific computing: Applications to the forward electrocardiographic problem. PLoS ONE 13(8): e0202410. https://doi.org/10.1371/journal.pone.0202410

Editor: Rafael Sachetto Oliveira, Universidade Federal de Sao Joao del-Rei, BRAZIL

Received: January 5, 2018

Accepted: August 2, 2018

Published: August 23, 2018

Data Availability Statement: All relevant data are within the paper and its Supporting Information files.

Funding: L. Cardone-Noott was supported by the Engineering and Physical Sciences Research Council (EP/G03706X/1; https://www.epsrc.ac.uk/). B. Rodriguez was supported by a Wellcome Trust Senior Research Fellowship in Basic Biomedical Science (100246/Z/12/Z; https://wellcome.ac.uk/). A. Bueno-Orovio is funded by a BHF Intermediate

Abstract

Input-output (I/O) optimization at the low-level design of data layout on disk drastically impacts the efficiency of high performance computing (HPC) applications. However, such a low-level optimization is in general challenging, especially when using popular scientific file formats designed with an emphasis on portability and flexibility. To reconcile these two aspects, we present a novel low-level data layout for HPC applications, fully independent of the number of dimensions in the dataset. The new data layout improves reading and writing efficiency in large HPC applications using many processors, and in particular during parallel post-processing. Furthermore, its combination with a cached write mode, in order to aggregate multiple writes into larger ones, substantially decreased the writing times of the proposed strategy. When applied to our simulation framework for the forward calculation of the human electrocardiogram, the combined strategy resulted in drastic improvements in I/O performance, of up to 40% in writing and 93–98% in reading for post-processing tasks. Given the generality of the proposed strategies and scientific file formats used, our results may represent significant improvements in I/O performance of HPC applications across multiple disciplines, reducing execution and post-processing times and leading to a more efficient use of HPC resource envelopes.

Introduction

The optimization of high performance computing (HPC) codes is an area of active research, underpinning a continuous and cost-effective development of both established and emergent industrial and scientific sectors. As a representative example, the progress we are experiencing in computational medicine based on HPC applications is allowing the translation of mathematical models of physiological systems such as the heart to biomedical research and clinical practice. Within the field of cardiac electrophysiology, these include investigations on multi-scale mechanisms of disease and lethal arrhythmias [1, 2], drug action [3, 4], electrical therapy

Basic Science Research Fellowship (FS/17/22/32644; https://www.bhf.org.uk/). B. Rodriguez and A. Bueno-Orovio also acknowledge additional support from an Impact for Infrastructure Award of the National Centre for the Replacement, Refinement & Reduction of Animals in Research (NC/P001076/1; https://www.nc3rs.org.uk/), and the CompBioMed Centre of Excellence in Computational Biomedicine (European Commission Horizon 2020 research and innovation programme, grant agreement No. 675451; https://ec.europa.eu/programmes/horizon2020/). We also acknowledge the support of the BHF Centre of Research Excellence (RE/13/1/30181). This work made use of the facilities of the UK National Supercomputing Service (Archer Leadership Award e462). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

[5, 6], causes of inter-patient variability in response to treatment [7, 8], the role of myocardial structure in modulating heart function [9, 10], identification of novel biomarkers for clinical diagnosis [11, 12], and the stratification of patients at high risk of sudden cardiac death [13], among others.

In scientific computing, substantial efforts to improve HPC performance are frequently placed on the optimization of the numerical solution of the underlying physical models. As in other disciplines, in cardiac electrophysiology this involves the development of strongly scalable solvers [14, 15], improved problem-specific preconditioners [16, 17], temporal and/or spatial adaptivity [18, 19], higher-order numerical schemes [20, 21], or the use of reduced models to alleviate model complexity [22, 23]. Additional areas of active HPC performance improvements include multithreading, load-balance, compiler optimization, or optimization at the application and operating system levels. Critically, scientific applications in large HPC systems often read and write vast amounts of data. However, much less attention is given in general to the input-output (I/O) optimization of these codes, frequently assumed as an inevitable burden with little scope for improvement, leaving I/O as a challenging factor in the overall performance of HPC applications [24].

Ideally, the first stage of I/O optimization in HPC applications should take place at the low-level design of the output structure, based on the most frequent access patterns to data. This is, however, a laborious task, in particular when using popular scientific file formats, designed with a focus on portability and flexibility (such as the HDF5 file format considered here [25]). To circumvent this complexity, the use of higher level analysis tools is usually preferred for HPC I/O optimization [26–29], commonly based on the profiling of communication, latencies and computation overheads in parallel applications. Other diagnostic tools also provide comprehensive summaries of data access patterns [30–32], which can then be used in later stages of I/O optimization. Additional middleware file formats with improved write and read performance have also been developed [33], although their acceptance in scientific applications still remains low.

To simplify such a delicate crafting process of low-level I/O optimization, in this work we present a general algorithm for the automatic design of data layout, solely based on the size of the dataset and one additional parameter, the target chunk size in bytes. When combined with a cached write mode, the new algorithm (applied to our simulation framework for the forward calculation of the human electrocardiogram) resulted in overall improvements in I/O performance of up to 40% in writing to disk, and between 93% to 98% in reading for different post-processing tasks. Given the generality of the proposed strategies and the scientific file formats used (HDF5 as a standard for portability and flexibility, and of widespread use among the scientific computing community), our methodology may be broadly applied to other scientific areas, yielding significant improvements in I/O performance of HPC applications and a more efficient use of HPC resources across multiple scientific disciplines.

Materials and methods

Bidomain equations in a bath

The bidomain equations [17] describe the evolution of the electrical activity in the heart (Ω_h), surrounded by a conductive passive medium (i.e. the bath, Ω_b). Two overlapping domains are assumed in the heart: the intracellular and extracellular domains, with respective potentials ϕ_i and ϕ_e, whose difference provide the transmembrane potential ($V_m = \phi_i - \phi_e$). The formulation of the problem is then given by the system of partial differential

equations (PDEs):

$$\chi(C_m \partial_t V_m + I_{ion}) - \nabla \cdot (\sigma_i \nabla \phi_i) = -I_i, \quad \text{in } \Omega_h \tag{1}$$

$$\nabla \cdot (\sigma_i \nabla \phi_i + \sigma_e \nabla \phi_e) = 0, \quad \text{in } \Omega_h \tag{2}$$

$$\nabla \cdot (\sigma_b \nabla \phi_e) = 0, \quad \text{in } \Omega_b \tag{3}$$

$$\partial_t \mathbf{u} = \mathbf{f}(\mathbf{u}, V_m), \quad \text{in } \Omega_h \tag{4}$$

together with boundary conditions $\mathbf{n} \cdot (\sigma_i \nabla \phi_i) = 0$ and $\mathbf{n} \cdot (\sigma_b \nabla \phi_e) = I_e$ on the heart and the external bath boundaries, respectively, where $\mathbf{n}$ represents the outward-facing unit normal. In the equations above, σ_i and σ_e are the intracellular and extracellular conductivity tensors, σ_b is the bath conductivity, χ is the surface-area-to-volume ratio, and C_m the membrane capacitance per unit area. The vector $\mathbf{u}$ contains cell-level variables (such as ionic concentrations and membrane gating variables), and $I_{ion}(\mathbf{u}, V_m)$ is the ionic current per unit surface area, as given by the cellular electrophysiological model $\mathbf{f}$. The source term I_i is the intracellular stimulus per unit volume, whereas I_e is a stimulus current per unit area at the external boundary of the bath (zero in the absence of an external electrical field).

Simulation environment

For the numerical solution of the bidomain equations we used Chaste (Cancer, Heart, and Soft Tissue Environment) [34, 35], an open-source electrophysiology solver package using the finite element method. Main dataset I/O used HDF5 version 1.8.14 on a Lustre filesystem. MPI was provided by the Cray MPT, based on MPICH 3. Simulations were performed on the ARCHER UK National Supercomputing service (http://www.archer.co.uk/). Our new HDF5 chunking algorithm with caching as described in this work is publicly available as part of the `/io/src/` subfolder of Chaste's open-source distribution (https://github.com/Chaste/).

Benchmark problem

The HPC scientific computing framework for which the I/O strategy is optimized in this work is illustrated in Fig 1. The bidomain with bath equations were solved in an anatomically realistic human ventricular and torso (i.e. the bath, also containing lungs and bones) mesh. The

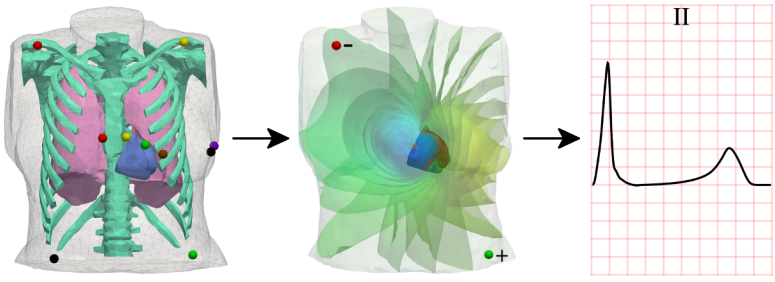

Fig 1. Illustration of the HPC scientific computing framework for which the I/O strategy was optimized in this work. From left to right: anatomically realistic human heart-torso mesh, also containing lungs and thoracic cage (left). The colored spheres indicate the location of the virtual electrodes for the calculation of the electrocardiogram, using standard (European) color-coding. Two double-precision floating-point numbers representing electric scalar fields in time and space are recorded at each node in the tetrahedral mesh (middle). These electric fields are finally post-processed to generate multiple time series (e.g. lead II as shown in the right panel), representing a simulated 12-lead electrocardiogram.

https://doi.org/10.1371/journal.pone.0202410.g001

combined heart-torso mesh has a total of about 3.25 million nodes and 19.4 million tetrahedra. A detailed description of model parameterization is provided in [36]. The ten Tusscher-Panfilov model [37] was used to describe human ventricular electrophysiology at the cellular level. For this mesh resolution with two double-precision outputs per printing time step (V_m and ϕ_e), each printing time step requires storage of 8B × 3253316 × 2 ≈ 52.1MB.

The PDE time step was 25 μs, with temporal adaptivity between consecutive PDE time steps for the numerical solution of the cellular electrophysiology model. For feasibility in generating the benchmark solutions, the printing time step was set to the PDE time step, and the total run time was 150 printing time steps. To include other types of data access, the post-processing of ventricular maps of activation times and peak transmembrane voltage were enabled, which involve reading from the main results dataset. Finally, all datasets were converted to VTK visualization files as an additional post-processing step.

Results

General considerations on HDF5 default data layout

At the hardware level, digital data storage is one-dimensional, so obviously there must be a mapping between the multidimensional datasets represented in HDF5 files and the disk. The default method simply serializes the in "row-major order", which might or might not be suitable depending on the access pattern (the order in which each process accesses values in the dataset).

Suppose we have a two-dimensional 10 × 10 dataset using this default layout (Fig 2). In row-major order the data are serialized by rows, e.g. elements 1 to 10 (labelled) are contiguous on disk, and the same for the following rows. Using conventional matrix notation indexed from 1, if a process wants to read entries (3,4) to (3,8) inclusive, i.e. the 24th to 27th elements (shaded, top panel), then it can do this very efficiently by reading a contiguous region on disk (solid arrows). On the contrary, if a process wants to read entries (4,3) to (8,3) inclusive, i.e. the 33rd, 43rd, 53rd, 63rd, and 73rd elements (shaded, bottom panel), then it must perform a number of relatively expensive disk seeks between each row (dotted arrows). Clearly, for good performance the data layout must be based on the access pattern.

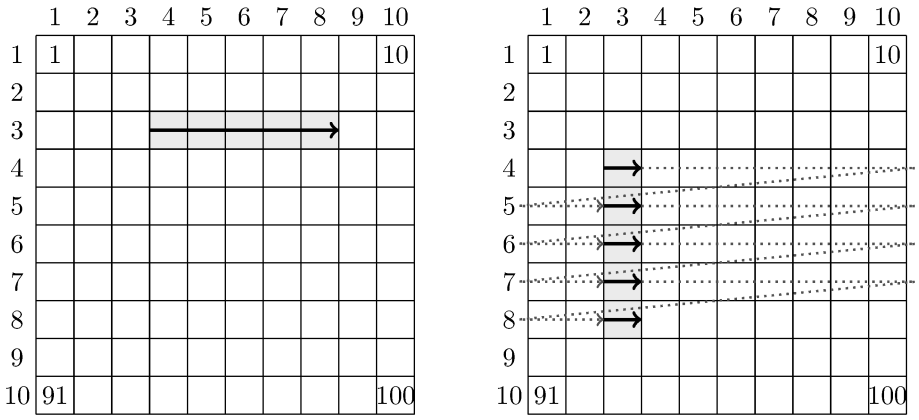

Fig 2. Default data layout in HDF5, and differences between reading a row and a column. A row (left) may be read efficiently because the elements are contiguous (solid arrow); reading a column (right) by contrast requires a disk seek (dotted arrows) before every read. The columns and rows of the dataset are labelled 1–10. Four data elements are labeled with their locations on disk (1,10, 91 and 100). Adapted from the HDF5 support webpage: https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/.

https://doi.org/10.1371/journal.pone.0202410.g002

Since data layout on disk has a significant influence on performance, HDF5 allows the specification of customized *chunk* shapes based on typical access patterns. A chunked dataset is divided into repeating units of the chunk dimensions, and space for each chunk is allocated contiguously on the disk. In the second example on column-reading above, we might utilize chunking by setting the chunk dimensions to the shape of a column in the dataset. With chunks that coincide with columns, the library would lay the data out on the disk column by column, so that any access from a column becomes a single, fast, contiguous read.

For cardiac applications, HDF5 datasets in our simulation package are three-dimensional objects, over time in the first dimension, nodes in the second, and variables in the third. A typical chunk has size $\{C_t, D_N, D_v\}$, where: C_t depends on the number of printing time steps, D_t; D_N is the total number of nodes; and D_v is the total number of variables. In other words, each chunk spans the dataset in the 'nodes' and 'variables' dimensions, with ceil(D_t/C_t) chunks in the time dimension. This strategy is fairly efficient for simulations with a small number of processors and nodes as each chunk will be relatively small and easy to cache. It is, however, poorly suited to large parallel applications as discussed below.

Considering parallel performance, using chunks that span the node dimension is suboptimal due to the way the problem is partitioned across parallel processes. At the start of a simulation, the mesh is partitioned and each process is assigned a subset of the nodes that remains unchanged for the duration of the simulation. For simplicity, the nodes are reordered so that the nodes owned by each process are indexed contiguously. Using the earlier notation, each process will access a contiguous block of approximately size $\{D_t, D_N/j, D_v\}$, where j is the number of processes (assuming an equal partition of nodes between processes). Recalling the chunk shape, we note that it is "orthogonal" to the regions owned by each process (Fig 3).

At this point it is necessary to briefly introduce two of the HDF5 drivers of our simulation package, and how they differ in reading and writing a chunked dataset. When writing, it uses the MPI-IO driver, which is specialized for parallel applications. The chunk shape has relatively little impact on writing because the MPI-IO driver uses direct access to the disk, and collective writes are used (designed to improve performance when writing from many processes to a single file by first concentrating data onto intermediate *aggregators*). When reading, however, (e.g. post-processing or generating visualization files) the default driver is used which uses standard POSIX operations. Unlike the MPI-IO driver, the default driver attempts to cache an entire chunk when data in it are accessed. Furthermore, the caching implementation in the default driver dictates that access can only be done on whole chunks. In other words, even if a process attempts to read just one entry from a chunk and it is possible for the chunk to be cached, then the process will read the entire chunk into the cache before continuing. If the chunk is too large to be cached then the cache is bypassed completely and direct access is used.

From the preceding paragraphs it becomes clear why using chunks that span the node dimension is suboptimal. First, consider an HDF5 reader being used to perform post-processing on a dataset in parallel. Each process is expected to access all the variables for all its nodes for all times. Depending on the size of each chunk compared to the size of the chunk cache there are two possibilities:

- If chunks are small relative to cache size, then when a process requests a value in its block it must first read the entire chunk into its cache, despite most of the nodes belonging to other processors and being therefore of no interest. This might yield acceptable performance if enough chunks can be cached and the access pattern is conducive, which is not the case in frequent post-processing tasks. For example, imagine the reader accesses all the time points for one node, then all the time points for the next node, etc. Unless all the chunks can be

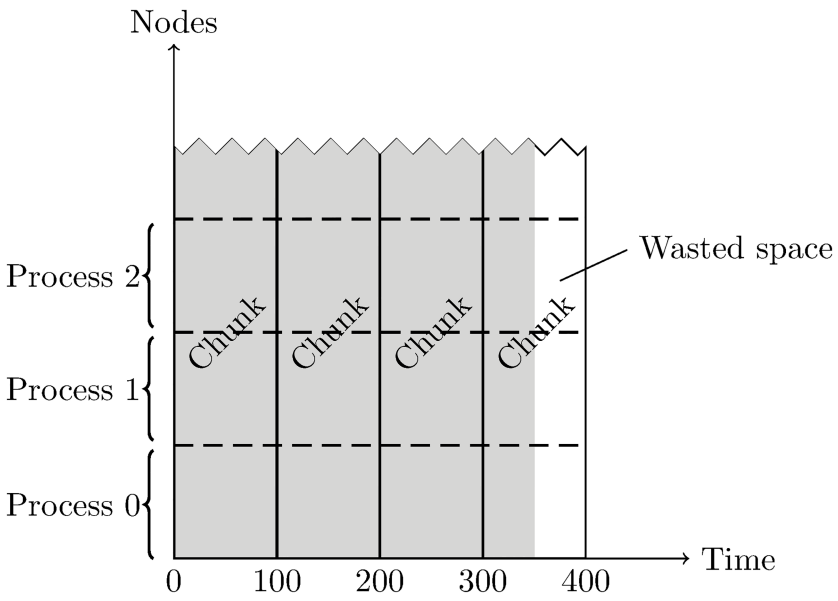

Fig 3. Typical chunk layout. This example depicts a dataset of size 350 in the time dimension and a large number in the nodes direction (grey). The chunks are of size $100 \times D_N$ (solid lines), so 50 elements (12.5% of the file) are wasted at the edge (white). Each process is concerned with a slice of the dataset shaped orthogonally to the chunks (dashed lines). The third dimension has been suppressed for clarity, since the chunks and process boundaries span it.

https://doi.org/10.1371/journal.pone.0202410.g003

cached at once, the reader will be forced to read the entire dataset on every iteration, just to get the values for each node.

- If chunks are larger than cache size, then each process will access the dataset independently and directly. In this case, the potential performance improvement from using the cache is lost, but the requirement to read in whole chunks is dropped, possibly resulting in less disk activity. Still, for optimal performance the use of caching should be favored.

As mentioned above, writing to the HDF5 file is expected to be less affected by chunk settings than reading, since the MPI-IO driver only uses direct access with collective writes. Nevertheless, the chunk shape results in every chunk being accessed simultaneously by all processes (Fig 3), possibly resulting in increased library overhead to track modifications and maintain consistency between processes. A chunk layout more closely resembling the process boundaries would alleviate this issue.

I/O optimization in large HPC systems

A new chunking algorithm. The most methodical way for optimal chunk design would be to set the chunk shape based on an analysis of the most frequent access patterns, within some chunk capacity limits. The chunk size is important because (1) disks are generally better at reading fewer, larger regions than more, smaller regions, and (2) it influences the size of chunk cache needed for good read performance. Unfortunately, two opposing modes of access coexist in our case at different stages of the simulation. When solving, the fastest varying dimension is the variable dimension, followed by nodes, and finally time. When performing post-processing, it is not uncommon to instead access the variables for each node over all time. An access-based approach might also result in highly problem- and/or machine-specific algorithms, that might show good performance in some applications at the expense of others.

Instead, a general algorithm was developed to set the chunk size based only on the size of the dataset and one parameter, the target chunk size in bytes (T_B). For maximum generality and in order to ensure its applicability to any number of dimensions, the new chunk algorithm is designed to treat all dimensions equally. Another design requirement is that the chunk shape should result in high storage efficiency. HDF5 allocates the minimum integer number of chunks required to contain the dataset (recall Fig 3). Most conceivable chunk shapes *a priori* might therefore result in unacceptable amounts of wasted space at the edges of the dataset, because the chunk size is unlikely to be (close to) a factor of the dataset size in every dimension.

Central to the proposed solution is the variable "target size", T (not to be confused with T_B). First, for each dimension, the rounded-up division of the dataset size by the target size gives the minimum number of target-sized chunks that would span the dataset. Second, the rounded-up division of the dataset size by this number of chunks gives the actual size of chunk that is closest to the target size while still being close to a multiple. The problem then reduces to finding the target size that best satisfies the chunk size requirement in bytes. The solution can be written concisely as follows. Let the chunk and dataset sizes be vectors denoted by $\vec{C}$ and $\vec{D}$, respectively:

$$\vec{C} = (C_1, C_2, \ldots, C_N) \tag{5}$$

$$\vec{D} = (D_1, D_2, \ldots, D_N) \tag{6}$$

where N is the number of dimensions in the dataset (usually 3 for time, nodes and variables). We therefore get $\vec{C}$ by finding the largest T such that

$$8 \prod_{i=1}^{N} C_i \leq T_B \tag{7}$$

$$C_i \leq D_i \qquad (1 \leq i \leq N) \tag{8}$$

$$C_i = \left\lceil \frac{D_i}{\lceil D_i/T \rceil} \right\rceil \quad (1 \leq i \leq N) \tag{9}$$

where in Eq (7) represents the chunk size constraint (each element is 8 B), in Eq (8) limits the chunk to the dataset size in each dimension, and Eq (9) defines the chunk size given the dataset size and target size in such a way as to minimize wasted space.

Algorithm 1 New HDF5 chunk size algorithm

```
1: D⃗                                  ▷ Dataset size in elements
2: C⃗                                  ▷ Chunk size in elements
3: C_B ← 0                                    ▷ Chunk size in B
4: T ← 0                               ▷ Target chunk size in elements
5: T_B ← 128 × 2^10                         ▷ Target chunk size in B
6: ϒ ← False                          ▷ Whether chunk spans dataset
7:
8: function SetChunkSize
9:   while (C_B < T_B) &!ϒ do ▷ While chunk is smaller than target
10:    Increment T
11:    (C⃗, C_B, ϒ) ← CalculateChunkDims(T)
12:  end while
13:  if C_B > T_B then                ▷ If chunk has exceeded target
14:    Decrement T
```

```
15:     (C⃗, C_B, ϒ) ← CalculateChunkDims(T)
16:   end if
17: end function
18:
19: function CalculateChunkDims(T)
20:   C_B ← 8                          ▷ 8 B per element
21:   ϒ ← True
22:   for i in dimensions do           ▷ For each dimension
23:     x ← ceil(D_i / T)
24:     C_i ← ceil(D_i / x)
25:     C_B ← C_B × C_i
26:     ϒ ← ϒ & (x ↔ 1)
27:   end for
28:   return C⃗, C_B, ϒ
29: end function
```

The method for solving the above is described in Algorithm 1. The dataset size ($\vec{D}$) and target chunk size (T_B) are assumed as predetermined. The first while loop (line 9) increases the target size (T) and calculates the resulting chunk dimensions ($\vec{C}$) until the target size in bytes (T_B) is reached, or the chunk spans the entire dataset (Υ is True). Note that a binary search for T between 1 and $\max(\vec{D})$ (for example) would be faster, but as invoked only once per dataset (resulting in a negligible overhead in overall wall times) this does not represent a significant increase in performance, and we chose to present the algorithm here in incremental form in the interest of clarity. After leaving the while loop, if T_B has been exceeded (line 13), the algorithm brings the size back below the target. Once given a target size in elements, the CalculateChunkDims function (line 19) calculates chunk dimensions that aim for the target size in each dimension while minimizing wasted space at the dataset edge as outlined above. First, it calculates the minimum number of chunks of size T that would be required to span the dataset (x, line 23). Then, given x chunks, it calculates the minimum number of elements required in each chunk to span the dataset (line 24). This function also calculates the actual chunk size in bytes (C_B) and determines Υ. The chunk size in bytes is the product of 8 B and all the elements of $\vec{C}$ (line 24). Finally, if $x = 1$ on every iteration, then one chunk spans the entire dataset and Υ is True (line 26).

The choice of T_B depends on the problem size and the computer. The default value was set to 128 kB as this resulted in consistent performance in the small profiling tests that are run regularly to monitor performance. For large problems it was increased to 1 MB, in agreement with the default stripe size in Archer as discussed next.

Data striping. At the filesystem level, *data striping* is a common technique in HPC systems (including the *Lustre* filesystem in Archer) to increase data throughput by splitting files into segments and dividing the segments amongst multiple physical storage targets (e.g. hard disk drives).

For performance improvements at this level, two parameters may be set on a file or directory basis: the stripe size (S) and stripe count (c). The former is the size (in bytes) of each stripe, whereas the latter is the number of Object Storage Targets (OSTs) over which to divide the stripes (see Fig 4). The system defaults on Archer are 1 MB and 4, respectively, and there are 48 OSTs at the time of writing this work, each capable of writing at roughly 500 MB/s.

Optimal values for S and c can only be found through experimentation. For reference, the Lustre documentation (see Section 18.2.1 in [38]) recommends a stripe size between 512 kB and 4 MB. Smaller sizes are not recommended 'because the Lustre file system sends 1 MB chunks over the network'; more is not recommended because 'stripe sizes larger than 4 MB may result in longer lock hold times and contention during shared file access'. Finally, the stripe size must

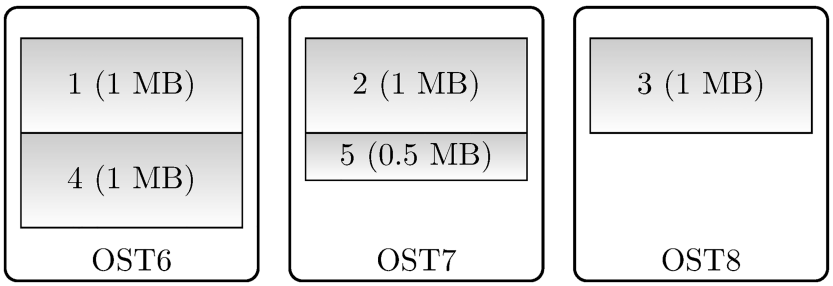

Fig 4. Data striping. This example depicts a 4.5 MB file striped across 3 OSTs with 1 MB stripe size. The first OST is chosen at random by the filesystem in order to load-balance, and in this example it is OST6. 1 MB stripes are then placed round-robin on each of the three OSTs, ending with a 0.5 MB stripe on OST7.

https://doi.org/10.1371/journal.pone.0202410.g004

be a multiple of the page size (enforced to a multiple of 64 kB for compatibility). The default stripe size on Archer (1 MB) is generally optimal and other values will not be considered here.

The stripe count c will be investigated below. For writes from many processors to a single file a large stripe count is recommended, but not too many counts as this results in extra overhead for diminishing returns. A starting point based on the Lustre documentation is to use approximately '1 stripe per GB' to '1 stripe per 4 GB' of file size. As an example, for 100 printing time steps of our benchmark problem (expected dataset size of $\sim$5.21 GB), c should probably be between 2 and 6. Another is to "load balance" by using an integer factor of the number of processors, such as one stripe per compute node so that each node gets one aggregator.

Cached writes. Regardless of the data layout used, the simulation results are written to the HDF5 file every print time step of simulation time. Recall from our benchmark description (see Methods) that one printing time step of data in our benchmark consumes about 50 MB on disk. Large HPC parallel file systems have good sustained throughput and are typically optimized for high bandwidth (such as the 500 MB/s per OST in Archer as mentioned above), but performance for small writes is much lower. They work best with a small number of large, contiguous I/O requests whereas small ones are generally discouraged. We should therefore expect several hundred 50 MB writes to show worse performance than, say, one 40 GB write.

The chunk cache provided by HDF5 might have been a viable answer, but it is currently not available when using the MPI-IO driver in write mode. The selected solution was to implement a memory cached mode in the HDF5 writer, whose constructor now takes an argument specifying if the cache will be enabled. This simplifies the implementation of our strategy over making the cache switchable, which would require extra logic like flushing the cache when switching. A new vector member with a reserved size of $C_t \times N_n \times N_v$ acts as the cache, where C_t is the size of a chunk in the time dimension (as calculated by the new chunking algorithm), N_n is the number of nodes owned by a process, and N_v is the number of variables. Once the number of elapsed print time steps equals C_t, each process writes the contents of its cache to the HDF5 file. As collective writes are still used, the library then takes care of consolidating the data onto aggregators as normal.

Input-Output efficiency

Performance testing. In this section, the described data layout and cache writing strategies will be evaluated to investigate I/O efficiency. Their performance will be measured in the benchmark problem detailed in Methods, for both a small (8) and a medium (20) number of compute nodes (i.e. 192 and 480 cores, respectively) to test parallel scaling, and for three values of stripe count c (4, 24 and 42). Specifically, we compare:

1. Default chunks of size 41 in the time dimension, i.e. chunk size {41, 3253316, 2} (∼ 2 GB, the maximum single write size in ROMIO/MPI-IO).
2. New-style chunks of target size 1 MB, specifically {151, 434, 2} (1048544 B) as a result of applying Algorithm 1 to the dimensions of our dataset.
3. As in (2), but with caching enabled.

Benchmark results are shown in Fig 5. The three stacked bars in each panel correspond to the three strategies detailed above. The bars display the time spent (in minutes) in each of the following I/O categories: Output (writing to disk), PostProc (performing post-processing), and DataConv (HDF5 conversion to VTK visualization files). The three rows from top to bottom show results for each considered stripe count on the HDF5 file (4, 24, and 42, respectively). Finally, the left and right columns show results from 8 and 20 compute nodes (192 and 480 cores, respectively). Each simulation was performed three times in isolation to account for machine load. The times are presented as means and standard error of the mean (S.E.M.) of

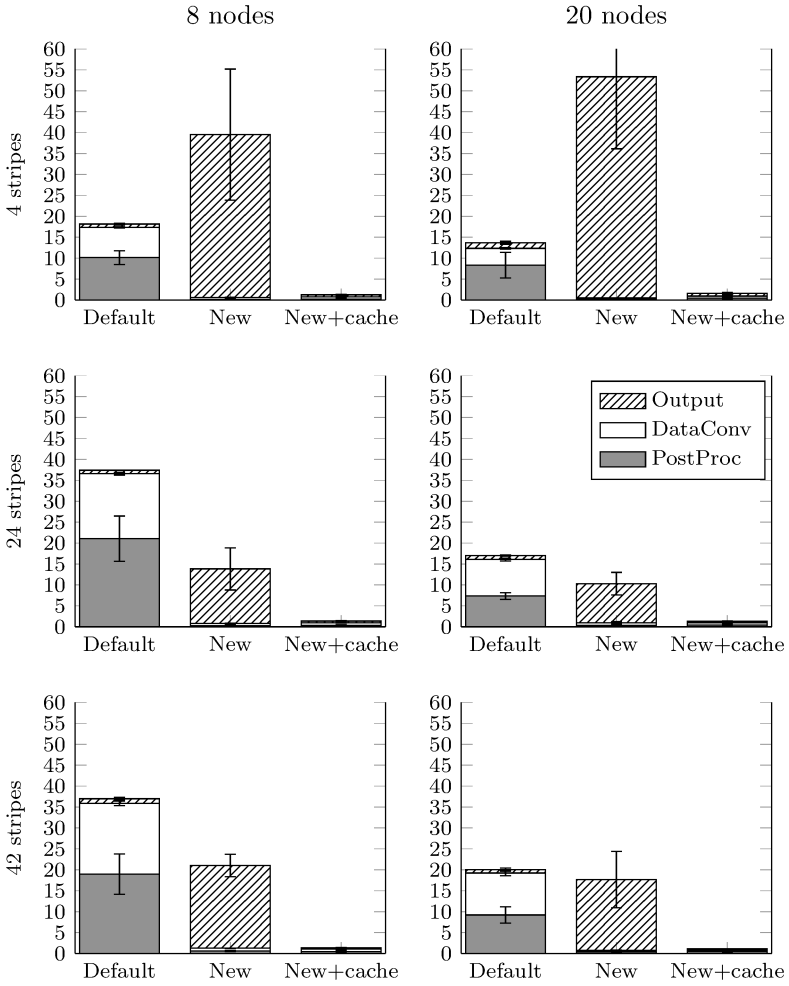

Fig 5. Summary of I/O results. Time spent writing to disk (Output), converting to VTK visualisation files (DataConv), and performing post-processing (PostProc) by each of the three methods: default-style chunks, new-style chunks, and new-style chunks with caching. The left and right columns represent results for simulations on 8 and 20 compute nodes, respectively. From top to bottom, the panels show results using stripe counts of 4, 24, and 42, respectively. Times are means from three repeats (in minutes), whereas error bars represent the S.E.M. Full data provided in S1 Table.

https://doi.org/10.1371/journal.pone.0202410.g005

the three repeats. Performing additional repeats was unfeasible due to time and resource requirements.

The principal results of this study are as follows. In all cases, the default chunk layout (1) spent little time in Output and substantial time in DataConv and PostProc. The former point is likely due to the small number of chunks (just three), resulting in very little overhead when coordinating collective writes. The latter point, however, clearly illustrates the high cost of reading results from a poorly laid out dataset for additional post-processing tasks. Conversely, the new layout (2) spent the vast majority of time in Output and little in DataConv or PostProc. The writing of the resulting 7497 new style chunks is evidently slow, but the smaller, squarer chunks allow the post-processing and conversion steps to happen extremely quickly by leveraging the built-in HDF5 chunk-caching functionality. Moreover, the effect of enabling the custom chunk-writing cache on the new style method is striking (3). In this case, the benefits of the new data layout to DataConv and PostProc are retained, whereas the time spent in Output is reduced to under 30 s in all cases. Clearly the new algorithm with cached writes offers superior performance on Archer compared to the considered alternatives.

Fig 5 further illustrates the results for each method with respect to stripe and node counts, of importance for performance optimization. First, comparison by rows (stripe count effects) for the default chunk (1) illustrates increased DataConv times with 24 or 42 stripes compared to 4 stripes, both with either 8 or 20 nodes. Clearly, the DataConv process is unable to leverage the extra bandwidth offered by the large stripe counts, probably due to either an overhead of communicating with many OSTs, or technical advantage from concentrating on a small number of OSTs (e.g. internal OST caching). PostProc showed the same trend. In contrast, the uncached new chunks (2) were faster with 24 or 42 stripes than 4, showing performance benefits in parallel I/O. The severe bottleneck in Output is alleviated with a larger number of stripes, suggesting that the performance with 4 stripes is either limited by the OSTs or due to overwhelming the 4 threads assigned to aggregators. If we interpret the large error bars on Output as a sign of sensitivity to the machine load then the answer is probably the former. Finally, this method performed better with 24 stripes than 4 or 42, supporting the rule of thumb that a single large file written to by many processors should be striped, but not excessively, to avoid incurring large overheads. Another well-suited characteristic of the new-chunk cached method (3) for large HPC systems is its insensitivity to stripe counts, which alleviates optimization needs at the file system level, in particular for novice users.

Second, by comparing columns (node count effects), performance is improved for the default chunks (1) going from 8 to 20 nodes, both in DataConv and PostProc. This suggests that in spite of the sub-optimal I/O performance of these chunk layouts, the post-processing stage is somewhat able to utilize additional cores. The un-cached new algorithm (2), however, scales poorly at best, showing no significant difference between 8 and 20 nodes. In the case of 4 stripes, Output is still slow, perhaps exacerbated by the larger number of nodes. Yet again, there were no significant differences in performance for the cached new method (3) with node counts, highlighting its robustness for scalable applications.

The previous results also indicate that a strip count c of 4 simultaneously yields the best studied I/O performance for both the default chunks and new cached algorithm. This agrees with the initial estimate on that c should be between 2 and 6. A comparative summary of benchmark times for these 4 stripes on 8 compute nodes is presented in Table 1 for all the considered I/O strategies (see S1 Table for rest of stripes and compute nodes). These represent relative improvements in I/O performance of the new strategy over the default layout method of 40% in HDF5 writing to disk, 93% in post-processing, and 98% in HDF5 conversion to visualization files.

Table 1. Benchmark times for 4 stripes on 8 compute nodes. Results correspond to those illustrated in the top-left panel of Fig 5. Times are given in seconds (mean ± S.E.M.; full data provided in S1 Table.). The default chunk layout is fast in Output but slow in the other two categories. The new chunk shape is the opposite. With caching of writes enabled on the new shape the time spent in all three areas is low.

Data Layout	Output (s)	DataConv (s)	PostProc (s)
1. Default	49.5 ± 13.0	431.2 ± 8.0	608.7 ± 98.5
2. New	2333.5 ± 940.4	27.5 ± 3.2	10.0 ± 2.9
3. New + cache	29.9 ± 6.5	30.6 ± 5.0	14.6 ± 7.3

https://doi.org/10.1371/journal.pone.0202410.t001

Alignment of new chunks with caching. As described in the presentation of our new chunking algorithm, any chunking algorithm is unlikely to produce chunks of exactly the target size. In addition, chunks are located by default at irregular locations within the file. Together, these two statements imply that a given chunk is unlikely to align perfectly with the stripe boundaries of the file. In such circumstances, accessing a chunk requires reading stripes from more than one OST. For example, if the stripe size and chunk target size are 1 MB and the true chunk size is slightly less than 1 MB, then reading a chunk is likely to involve requests to two OSTs. When the chunk and stripe size are similar, it might therefore be preferable for each chunk to be padded slightly with empty space so that each chunk starts on a stripe boundary. Note however that such an approach should be used with caution in order to avoid excessive wasted space.

The benchmark problem was used to test the performance with and without alignment (as implemented natively in the HDF5 library) in the new cached chunking algorithm. Results (mean ± S.E.M., in seconds) are shown in Table 2 for 110 and 113 repetitions of the unaligned and aligned cases, respectively.

Whereas HDF5 alignment did not substantially affect the overall performance of the new method, no improvements were attained in any of the considered I/O categories. A possible explanation for this is that the processes regularly read and write across chunk boundaries (i.e. the partition boundaries rarely fall exactly on chunk boundaries), so placing each chunk into its own stripe rarely results in a reduction in the number of OSTs accessed. Enabling alignment might also introduce small gaps into otherwise contiguous data, reducing performance slightly. The theoretical advantage of aligning chunks to stripes might however become apparent in larger problems.

Conclusion

In this work, we have presented significant improvements in the I/O performance of our electrocardiogram-simulation framework in large HPC infrastructures, particularly in the challenging and frequently neglected areas of data writing, post-processing, and data conversion. A general algorithm for the efficient design of data layouts in HDF5 files (as a leading scientific file format for data storage and portability) was developed, and further optimized using cached writes. The efficiency of the resulting I/O strategy with respect to native concurrent layouts has been shown, independently to stripe and node count effects in large HPC filesystems, as well as to data alignment within the resulting files. Furthermore, as a single parameter (the target

Table 2. Benchmark times for the new chunking algorithm. Cached writings are used (4 stripes on 8 compute nodes), with and without HDF5 alignment set to the stripe size. Times given in seconds (mean ± S.E.M.). Full data provided in S2 Table.

Method	Output (s)	DataConv (s)	PostProc (s)
Unaligned	20.7 ± 0.4	30.2 ± 1.3	7.5 ± 0.2
Aligned	23.8 ± 1.2	31.0 ± 1.3	8.4 ± 0.2

https://doi.org/10.1371/journal.pone.0202410.t002

chunk size in bytes) is responsible in our algorithm for the low-level design of the underlying datasets regardless their number of dimensions, this guarantees a maximum generality and its applicability to other scientific areas beyond the one considered in this work.

The most substantial contribution is the method in which HDF5 files are written to disk, including the design of a novel low-level data layout independent to the number of dimensions in the dataset. Two actions are central to this I/O strategy. Firstly, the data layout (the so-called chunk shape) was modified to improve efficiency when reading small amounts of data, which is common in large HPC applications using many processors. This yielded a significant reduction in times for post-processing of simulation results and their conversion to other visualization formats, which are common scientific requirements across disciplines. A side effect of the new data layout was an increase in the output times required for the writing of results to the HDF5 file, due to the nature of storage systems in large HPC systems. This was overcome by implementing a cached write mode which bundles multiple small writes into larger ones, substantially reducing the aggregate writing times. The overall result was a drastic reduction in the time spent in all I/O stages of our simulation framework, with relative improvements over default HDF5 layouts of 40% in writing, 93% in post-processing, and 98% in data conversion.

Previous efforts have also been deployed for the optimization of I/O performance in HPC applications. Of particular relevance for our work are write-optimized middleware systems, such as ADIOS (Adaptable IO System, [39]) or PLFS (Parallel Log-structured Filesystem, [40]). These high-level I/O Application Programming Interfaces (APIs) allow for a more aggressive writing and efficient reordering of data locations in the case of ADIOS, and for a decoupling of concurrent writes to improve the speed of checkpoints in the case of PFLS, resulting in up to 100 × improvements in writing in selected applications [40, 41]. Importantly, these write-optimized APIs have been also shown to not penalize read speeds [33]. However, they both introduce intermediate file formats that require conversion for analysis to standard scientific formats, or to be mounted as stackable filesystems on top of an existing parallel filesystem.

On the contrary, the simplicity of the I/O strategies presented in this work, solely based on the size of the dataset (independent to its number of dimensions) and the straightforward implementation of a cached write mode, could easily be incorporated into codes using popular scientific file formats like HDF5, which has a history of optimization on popular HPC platforms [42]. This would alleviate the need of using intermediate API layers and the associated additional complexity to end users, while resulting in important savings in writing, reading and post-processing times in scientific applications.

For applications involving mesh adaptivity, an inherent limitation of the HDF5 file format is that the chunk size is set at the dataset creation time and cannot be changed later, which forces the use of a fixed chunk size. Based on our results for fixed chunk sizes, we hence still expect an increased I/O efficiency for the new designed chunks compared to the default HDF5 layout in the presence of adaptivity. Such investigations (including the estimation of an optimal chunk size) fall however beyond the scope of our present work. In addition, our benchmark experiments were performed using a single (Lustre) parallel I/O environment. Although our cached results demonstrate almost complete independence to the number of stripe counts (see Fig 5), which in turn minimizes sensitivity to the choice of this parameter as a common technique to increase data throughput across multiple HPC filesystems, the evaluation of our methodology in other parallel environments also constitutes an interesting aspect for future research.

In conclusion, given the generality of our I/O strategies and file formats used, the improvements presented in this work might enable a more efficient use of HPC resources and accelerated progress in multiple areas of scientific research. This may allow researchers to achieve a

wider range of functionalities using standard scientific file formats, and therefore more complete simulation frameworks, within tolerable HPC resource envelopes.

Supporting information

S1 Table. Benchmark results for considered data layouts. I/O times for data writing to disk, data conversion, and post-processing in default-style chunks, new-style chunks, and new-style chunks with caching.
(XLSX)

S2 Table. Benchmark results under HDF5 alignment. I/O times for data writing to disk, data conversion, and post-processing in new-style chunks with caching, with and without HDF5 alignment.
(XLSX)

Acknowledgments

The authors thank the Chaste team and the Archer Help Desk for technical support, and Prof Kevin Burrage and Dr Ana Mincholé for valuable discussions.

Author Contributions

Conceptualization: Louie Cardone-Noott, Blanca Rodriguez, Alfonso Bueno-Orovio.

Data curation: Louie Cardone-Noott.

Formal analysis: Louie Cardone-Noott.

Funding acquisition: Louie Cardone-Noott, Blanca Rodriguez, Alfonso Bueno-Orovio.

Investigation: Louie Cardone-Noott.

Methodology: Louie Cardone-Noott.

Resources: Blanca Rodriguez, Alfonso Bueno-Orovio.

Software: Louie Cardone-Noott.

Supervision: Blanca Rodriguez, Alfonso Bueno-Orovio.

Validation: Louie Cardone-Noott.

Writing – original draft: Louie Cardone-Noott.

Writing – review & editing: Blanca Rodriguez, Alfonso Bueno-Orovio.

References

1. Behradfar E, Nygren A, Vigmond EJ. The role of Purkinje-myocardial coupling during ventricular arrhythmia: a modeling study. PLoS One. 2014; 9:e88000. https://doi.org/10.1371/journal.pone.0088000 PMID: 24516576
2. Dutta S, Mincholé A, Zacur E, Quinn TA, Taggart P, Rodriguez B. Early afterdepolarizations promote transmural reentry in ischemic human ventricles with reduced repolarization reserve. Prog Biophys Mol Biol. 2016; 120:236–248. https://doi.org/10.1016/j.pbiomolbio.2016.01.008 PMID: 26850675
3. Wilhelms M, Rombach C, Scholz EP, Dössel O, Seemann G. Impact of amiodarone and cisapride on simulated human ventricular electrophysiology and electrocardiograms. Europace. 2012; 14:v90–v96. https://doi.org/10.1093/europace/eus281 PMID: 23104920
4. Zemzemi N, Rodriguez B. Effects of L-type calcium current and human ether-a-go-go related gene blockers on the electrical activity of the human heart: a simulation study. Europace. 2015; 17:326–333. https://doi.org/10.1093/europace/euu122 PMID: 25228500

5. Rodríguez B, Li L, Eason JC, Efimov IR, Trayanova NA. Differences between left and right ventricular chamber geometry affect cardiac vulnerability to electric shocks. Circ Res. 2005; 97:168–175. https://doi.org/10.1161/01.RES.0000174429.00987.17 PMID: 15976315
6. Fenton FH, Luther S, Cherry EM, Otani NF, Krinsky V, Pumir A, et al. Termination of atrial fibrillation using pulsed low-energy far-field stimulation. Circulation. 2009; 120:467–476. https://doi.org/10.1161/CIRCULATIONAHA.108.825091 PMID: 19635972
7. Liberos A, Bueno-Orovio A, Rodrigo M, Ravens U, Hernandez-Romero I, Fernandez-Aviles F, et al. Balance between sodium and calcium currents underlying chronic atrial fibrillation termination: An in silico intersubject variability study. Heart Rhythm. 2016; 13:2358–2365. https://doi.org/10.1016/j.hrthm.2016.08.028 PMID: 27569443
8. Crozier A, Blazevic B, Lamata P, Plank G, Ginks M, Duckett S, et al. The relative role of patient physiology and device optimisation in cardiac resynchronisation therapy: A computational modelling study. J Mol Cell Cardiol. 2016; 96:93–100. https://doi.org/10.1016/j.yjmcc.2015.10.026 PMID: 26546827
9. Rutherford SL, Trew ML, Sands GB, LeGrice IJ, Smaill BH. High-resolution 3-dimensional reconstruction of the infarct border zone: impact of structural remodeling on electrical activation. Circ Res. 2012; 111:301–311. https://doi.org/10.1161/CIRCRESAHA.111.260943 PMID: 22715470
10. Zahid S, Cochet H, Boyle PM, Schwarz EL, Whyte KN, Vigmond EJ, et al. Patient-derived models link re-entrant driver localization in atrial fibrillation to fibrosis spatial pattern. Cardiovasc Res. 2016; 110:443–454. https://doi.org/10.1093/cvr/cvw073 PMID: 27056895
11. Mincholé A, Pueyo E, Rodríguez JF, Zacur E, Doblaré M, Laguna P. Quantification of restitution dispersion from the dynamic changes of the T-wave peak to end, measured at the surface ECG. IEEE Trans Biomed Eng. 2011; 58:1172–1182. https://doi.org/10.1109/TBME.2010.2097597 PMID: 21193372
12. Loewe A, Schulze WH, Jiang Y, Wilhelms M, Luik A, Dössel O, et al. ECG-based detection of early myocardial ischemia in a computational model: Impact of additional electrodes, optimal placement, and a new feature for ST deviation. Biomed Res Int. 2015; 2015:530352. https://doi.org/10.1155/2015/530352 PMID: 26587538
13. Arevalo HJ, Vadakklumpadan F, Guallar E, Jebb A, Malamas P, Wu KC, et al. Arrhythmia risk stratification of patients after myocardial infarction using personalized heart models. Nat Commun. 2016; 7:11437. https://doi.org/10.1038/ncomms11437 PMID: 27164184
14. Vázquez M, Arís R, Houzeaux G, Aubry R, Villar P, Garcia-Barnés J, et al. A massively parallel computational electrophysiology model of the heart. Int J Numer Meth Biomed Engng. 2011; 27:1911–1929. https://doi.org/10.1002/cnm.1443
15. Augustin CM, Neic A, Liebmann M, Prassi AJ, Niederer SA, Haase G, et al. Anatomically accurate high resolution modeling of human whole heart electromechanics: A strongly scalable algebraic multigrid solver method for nonlinear deformation. J Comput Phys. 2016; 305:622–646. https://doi.org/10.1016/j.jcp.2015.10.045 PMID: 26819483
16. Plank G, Liebmann M, dos Santos RW, Vigmond EJ, Haase G. Algebraic multigrid preconditioner for the cardiac bidomain model. IEEE Trans Biomed Eng. 2007; 54:585–596. https://doi.org/10.1109/TBME.2006.889181 PMID: 17405366
17. Bernabeu MO, Kay D. Scalable parallel preconditioners for an open source cardiac electrophysiology simulation package. Procedia Comput Sci. 2011; 4:821–830. https://doi.org/10.1016/j.procs.2011.04.087
18. Cherry EM, Greenside HS, Henriquez CS. Efficient simulation of three-dimensional anisotropic cardiac tissue using an adaptive mesh refinement method. Chaos. 2003; 13:853–865. https://doi.org/10.1063/1.1594685 PMID: 12946177
19. Franzone PC, Deuflhard P, Erdmann B, Lang L, Pavarino LF. Adaptivity in space and time for reaction-diffusion systems in electrocardiology. SIAM J Sci Computing. 2006; 28:942–962. https://doi.org/10.1137/050634785
20. Bueno-Orovio A, Pérez-García VM, Fenton FH. Spectral methods for partial differential equations in irregular domains: the spectral smoothed boundary method. SIAM J Sci Comput. 2006; 28:886–900. https://doi.org/10.1137/040607575
21. Arthurs CJ, Bishop MJ, Kay D. Efficient simulation of cardiac electrical propagation using high order finite elements. J Comput Phys. 2012; 231:3946–3962. https://doi.org/10.1016/j.jcp.2012.01.037 PMID: 24976644
22. Bueno-Orovio A, Cherry EM, Fenton FH. Minimal model for human ventricular action potentials in tissue. J Theor Biol. 2008; 253:544–560. https://doi.org/10.1016/j.jtbi.2008.03.029 PMID: 18495166
23. Wallman M, Smith NP, Rodriguez B. A comparative study of graph-based, eikonal, and monodomain simulations for the estimation of cardiac activation times. IEEE Trans Biomed Eng. 2012; 69:1739–1748. https://doi.org/10.1109/TBME.2012.2193398

24. Luu HVT. Optimizing I/O performance for high performance computing applications: from auto-tuning to a feedback-driven approach. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2015.
25. The HDF Group. Hierarchichal Data Format version 5, 2000-2010. [Online]. Available: http://www.hdfgroup.org/
26. Zaki O, Lusk E, Gropp W, Swider D. Toward scalable performance visualization with jumpshot. Int J High Perform Comput App. 1999; 13:277–288. https://doi.org/10.1177/109434209901300310
27. Mohr B, Wolf F. Kojak—a tool set for automatic performance analysis of parallel programs. Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference. 2003;1301–1304.
28. Shende S, Malony AD. The tau parallel performance system. Int J High Perform Comput Appl. 2006; 20:287–311. https://doi.org/10.1177/1094342006064482
29. Zaki O, Lusk E, Gropp W, Swider D. Automatic performance analysis with periscope. Concurr Comput Pract E. 2010; 22:736–748.
30. Seelam S, Chung IH, Hong DY, Wen HF, Yu H. Early experiences in application level I/O tracing on blue gene systems. IEEE International Symposium on Parallel and Distributed Processing. 2008;1–8.
31. Carns P, Latham R, Ross R, Iskra K, Lang S, Riley K. 24/7 characterization of petascale I/O workloads. 2009 IEEE International Conference on Cluster Computing and Workshops. 2009;1–10.
32. Yin Y, Byna S, Song H, Sun XH, Thakur R. Boosting application-specific parallel I/O optimization using IOSIG. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. 2012;196–203.
33. Polte M, Lofstead J, Bent J, Gibson G, Klasky SA, Liu Q, et al. . . . and eat it too: High read performance in write-optimized HPC I/O middleware file formats. Proceedings of the 4th Annual Workshop on Petascale Data Storage. 2009;21–25.
34. Pitt-Francis J, Pathmanathan P, Bernabeu MO, Bordas R, Cooper J, Fletcher AG, et al. Chaste: a test-driven approach to software development for biological modelling. Comput Phys Commun. 2009; 180:2452–2471. https://doi.org/10.1016/j.cpc.2009.07.019
35. Mirams GR, Arthurs CJ, Bernabeu MO, Bordas R, Cooper J, Corrias A, et al. Chaste: an open source C++ library for computational physiology and biology. PLoS Comput Biol. 2013; 9:e1002970.
36. Cardone-Noott L, Bueno-Orovio A, Mincholé A, Zemzemi N, Rodriguez B. Human ventricular activation sequence and the simulation of the electrocardiographic QRS complex and its variability in healthy and intraventricular block conditions. Europace. 2016; 18:iv4–iv15. https://doi.org/10.1093/europace/euw346 PMID: 28011826
37. ten Tusscher KHWJ, Panfilov AV. Alternans and spiral breakup in a human ventricular tissue model. Am J Physiol Heart Circ Physiol. 2006; 291:H1088–H1100. https://doi.org/10.1152/ajpheart.00109.2006 PMID: 16565318
38. Lustre Software Release 2.x Operations Manual. Oracle/Intel Corporation, 2013. [Online]. Available: http://lustre.org/documentation/
39. Lofstead J, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the Adaptable IO System (ADIOS). Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments. 2008;15–24.
40. Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, et al. PLFS: A checkpoint filesystem for parallel applications. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 2009;21:1–12.
41. Lofstead J, Zheng F, Klasky S, Schwan K. Adaptable, metadata rich IO methods for portable high performance IO. IEEE International Symposium on Parallel & Distributed Processing. 2009;1–10.
42. Howison M, Koziol Q, Knaak D, Mainzer J, Shalf J. Tuning HDF5 for Lustre file systems. Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10). 2012;1–10.