Hybrid Parallelization of a Large-Scale Heart Model

Dorian Krause[1], Mark Potse[2], Thomas Dickopf[1], Rolf Krause[1], Angelo Auricchio[3], and Frits Prinzen[2]

[1] Institute of Computational Science, University of Lugano, Switzerland
{dorian.krause,thomas.dickopf,rolf.krause}@usi.ch
[2] Cardiovascular Research Institute, Maastricht University, The Netherlands
mark@potse.nl, frits.prinzen@fys.unimaas.nl
[3] Fondazione Cardiocentro Ticino, Lugano, Switzerland
angelo.auricchio@cardiocentro.org

Abstract. The simulation of the electrophysiology of the heart is challenging due to its multiscale nature requiring the use of high spatial resolutions. Hence, it is important to efficiently utilize large parallel machines. In this article, we present a code designed to meet these scalability challenges on contemporary multicore-based massively parallel architectures. It is based on a well-established model originally designed for shared-memory systems. To improve scalability and extend support to distributed-memory architectures, we developed a hybrid OpenMP-MPI code. The new code shows excellent scalability up to 8448 cores with both explicit and implicit time discretizations. We present an in-depth analysis of the advantages of hybrid parallelization for this type of application.

1 Introduction

The contraction of the heart is a highly tuned mechanism that is organized by a complex electrical activation system. In each cardiac cell, approximately a million ion channels, pumps, and exchangers work together by allowing or forcing specific ions to cross the cell's inner and outer membranes [5]. They come in dozens of different types, encoded by different genes. Their permeability/activity depends on transmembrane voltage, ion concentrations, and time. Together, the ion channels in a cell membrane generate *action potentials*: temporary changes in transmembrane voltage that serve to open calcium channels, allowing a large amount of calcium to enter the cell, bind to the cell's contractile molecules, and initiate a contraction. Unlike skeletal muscle cells, cardiac muscle cells can trigger action potentials in their neighbors by passing current through the gap junctions that connect their interiors. By this mechanism, the entire cardiac muscle can be activated in less than 100 ms; a prerequisite for an ordered contraction.

Mathematical modeling is essential to understand the dynamics of the interactions between ion channels in the cell membrane [17]. The first numerical models of cardiac cells date from the 1960s. Since then, the models have grown

in complexity to capture newly discovered channel types as well as our evolving understanding of the known channels. In addition, it is now possible to couple many such models together to simulate entire hearts.

The high spatial and temporal gradients occurring in the propagation of the action potential require high spatial resolution. The size of whole-heart models therefore ranges from $O(10^6)$ nodes for small mammals [24] to $O(10^8)$ nodes for an adult human heart [6]. The required sizes could increase by more than an order of magnitude when muscle diseases are modeled. Consequently, much work is devoted to improving the performance and scalability of these simulations [1, 7, 15, 16, 25].

In this paper, we report on our work to develop a hybrid OpenMP-MPI parallelization for an existing heart model in order to optimize strong and weak scaling, improve performance, and advance the limit of achievable model size. The paper is organized as follows. In Section 2, we describe the mathematical models underlying the numerical simulation of the electrophysiology of the heart. In Section 3, we present the PROPAG code which is the basis of the work described in the article. In Section 4, we explain the new hybrid parallelization of PROPAG. Finally, in Section 5, we present and analyze our performance results.

2 Mathematical Model

The human heart contains a few billion muscle cells. Gap junctions allow action potentials to propagate from one cell to another [2, 12]. To model this electrophysiological system mathematically, it is customary to treat the intracellular environment with the gap junctions as a continuous domain. Likewise, the extracellular environment, which in reality consists of many different components, is treated as another continuous domain. These domains and the active membrane between them can then be discretized with a spatial step size that is much larger than a single cell. This leads to the *bidomain model* [4, 19]

$$\nabla \cdot (\sigma_\mathrm{i} \nabla \phi_\mathrm{i}) = \beta I_\mathrm{m} = -\nabla \cdot (\sigma_\mathrm{e} \nabla \phi_\mathrm{e}) \tag{1}$$

where ϕ_i and ϕ_e denote the intra- and extracellular potential fields, β is the membrane surface-to-volume ratio, σ_i and σ_e denote the conductivity tensors in the intra- and extracellular domain, and the transmembrane current density I_m equals

$$I_\mathrm{m} = C_\mathrm{m} \frac{\partial V_\mathrm{m}}{\partial t} + I_\mathrm{ion} + I_\mathrm{stim}\,, \tag{2}$$

with $V_\mathrm{m} = \phi_\mathrm{i} - \phi_\mathrm{e}$ and $C_\mathrm{m} = 1\mu\mathrm{F/cm}^2$ the membrane capacitance. Here, I_ion is the ionic current; I_stim denotes a stimulation current. In this study, we simulated I_ion with the Ten Tusscher-Panfilov 2006 model [23]. Free boundary conditions are imposed for ϕ_e, ϕ_i and V_m.

By inserting (2) into (1) and using an operator splitting approach (see also, for example, Vigmond et al. [25]), we obtain the following *bidomain reaction-diffusion model*

$$\frac{\partial V_\mathrm{m}}{\partial t} = \frac{1}{\beta C_\mathrm{m}} \Big[\nabla \cdot (\sigma_\mathrm{i} \nabla (V_\mathrm{m} + \phi_\mathrm{e})) - \beta \left(I_\mathrm{ion} + I_\mathrm{stim} \right) \Big] \tag{3}$$

$$\nabla \cdot ((\sigma_\mathrm{i} + \sigma_\mathrm{e})\nabla \phi_\mathrm{e}) = -\nabla \cdot (\sigma_\mathrm{i} \nabla V_\mathrm{m})\,. \tag{4}$$

Equation (3) is used to integrate V_m and (4) is used to compute ϕ_e from V_m at each time step.

An important simplification of the bidomain model is possible by assuming that σ_i is proportional to σ_e. This allows for lumping the two domains together in the integration. Introducing the monodomain conductivity tensor $\sigma'_{\mu\nu} = (\sigma_{\mathrm{i}\mu\nu}\, \sigma_{\mathrm{e}\mu\nu})/(\sigma_{\mathrm{i}\mu\nu} + \sigma_{\mathrm{e}\mu\nu})$ and eliminating ϕ_e from (3) and (4), we obtain the following *monodomain reaction-diffusion model* [19]:

$$\frac{\partial V_\mathrm{m}}{\partial t} = \frac{1}{\beta C_\mathrm{m}} \Big[\nabla \cdot (\sigma' \nabla V_\mathrm{m}) - \beta \left(I_\mathrm{ion} + I_\mathrm{stim} \right) \Big]\,. \tag{5}$$

Especially in whole-heart simulations, a monodomain model approximates a bidomain model very well [19]. By combining (5) with (4) it is still possible to compute ϕ_e, which is of special importance because, in contrast to V_m, it can be measured clinically (Figure 1). By solving (4) less frequently, this approach is much more efficient than a bidomain reaction-diffusion model. Solution techniques for these equations are a subject of continuing research [1, 7, 16, 25].

3 The Propag Code

The purpose of this work was to improve an existing cardiac simulation code, named Propag, [6, 19, 20], and to study and remove the bottlenecks that prevented it from running efficiently on contemporary massively-parallel computers.

The original code had been developed to solve both mono- and bidomain models on complicated geometries obtained from CT or MRI images of the heart. It was designed to run efficiently on shared-memory machines such as the SGI Altix family, using 16 to 128 cores. Parallelization had therefore been done with OpenMP directives in a NUMA-aware fashion (taking care of memory placement). In practice, the existing code could run heart models up to 100 million nodes in a reasonable amount of time and with good parallel performance. Strong scaling had a fixed limit of about $4 \cdot 10^5$ model nodes per core.

3.1 Characterization of the Code

Propag works with semi-structured finite-difference meshes, i.e., many of the possible node positions are not occupied. The heart or torso anatomy is input as a Cartesian array storing the cell types (tissue type, blood, or void). We refer to the elements of this Cartesian box as *voxels* whereas non-void voxels are called *cells*. Based on the cell types of surrounding voxels, the vertices of the mesh receive types as well. Vertices that are not completely surrounded by void are referred to as (mesh) *nodes*. In the original code, connectivity was computed on the fly. In the new code, the topology is stored explicitly since we cannot control the shapes of individual subdomains in the domain decomposition.

In this article we focus on the monodomain capabilities of the code. Originally, the code used an explicit Euler scheme to solve (5), see Algorithm 1.

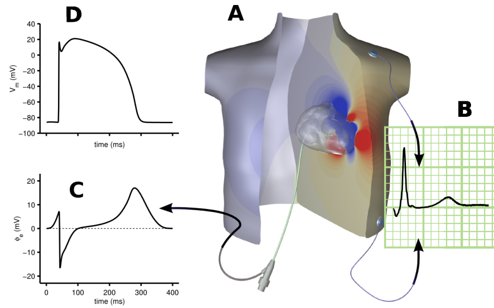

Fig. 1. Visualization of model results. The heart generates a potential field in the torso (A). Electrocardiograms (B) and catheter electrograms (C) can be derived and compared to measured data as well as to the underlying simulated action potentials (D) and dozens of other membrane-related variables.

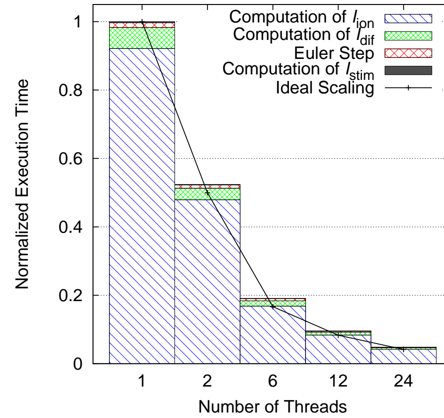

Fig. 2. Scaling of the original Propag code in a monodomain run with breakdown of runtime.

Algorithm 1 Monodomain Explicit Euler Time Integrator

1: Compute $I_{\text{dif}}^{n+1} = \beta^{-1}\nabla \cdot (\sigma' \nabla V_{\text{m}}^n)$ and I_{stim}^{n+1}
2: Evaluate $I_{\text{ion}}^{n+1} = \texttt{ION_STEP}(V_{\text{m}}^n, I_{\text{dif}}^{n+1}, I_{\text{stim}}^{n+1})$
3: Set $V_{\text{m}}^{n+1} = V_{\text{m}}^n + \tau \left[I_{\text{dif}}^{n+1} - I_{\text{stim}}^{n+1} - I_{\text{ion}}^{n+1} \right]$

In monodomain mode, the computation of I_{ion} in `ION_STEP` dominates the runtime (cf. Figure 2). It consists of a single loop over all mesh nodes and the approximate solution of a set of ordinary differential equations (about 40 in our model) at each node and hence is amenable to parallelization.

In Figure 2, an analysis of the runtime of the original Propag is shown. The graph shows a breakdown of the runtime of a monodomain simulation on one 24-core node of a Cray XE6 (equipped with two AMD Opteron 2.1 Ghz "Magny Cours" processors). Due to the NUMA-aware memory allocation and since runtime is distributed over only few scalable tasks of large granularity, the OpenMP parallelization is very efficient and OpenMP management overhead is negligible. The parallel efficiency on 24 cores is 86.9 % for this rather small example (422,091 mesh nodes).

The reaction-diffusion equation (5) contains a stiff diffusion term. With explicit time integration schemes, numerical stability requires a time step size that decreases quadratically with the spatial step size. To enable stable and accurate integration of very large models (with several billion degrees of freedom) we recently implemented an Implicit-Explicit (IMEX) Euler time discretization in Propag. Here, the linear diffusion term is treated implicitly while the non-linear ionic current is treated explicitly. In contrast to an explicit integration scheme, the IMEX Euler method requires the solution of a linear system in each time

step. In our experience, the matrix in this system is well-conditioned for practical time step sizes so that a few Bi-CGSTAB steps suffice to effectively reduce the (relative) residual norm to the tolerance $\varepsilon = 10^{-8}$.

4 Hybrid Parallelization

The currently largest shared-memory machines are limited to a few thousand cores per machine while the largest distributed-memory architectures scale to hundreds of thousands of cores. To efficiently utilize these resources, we ported Propag to an MPI code that can run on distributed-memory architectures. Such systems usually consist of a large number of multi-socket compute nodes connected by a high-speed interconnect. In recent years, the number of cores per socket has increased significantly. Within a compute node, memory is shared between cores, usually with NUMA architecture. Therefore, we retained the existing OpenMP parallelization, which is efficient for intra-node parallelization, and added an MPI layer for inter-node parallelism. Such a *hybrid* parallelization approach has been used for a variety of codes and has proven beneficial for several reasons:

1. It simplifies adding new levels of concurrency beyond what is easily accomplished with MPI and hence can be used to overcome algorithmic scaling limitations (e.g., GTC [3]).
2. It allows to mitigate efficiency loss in applications that are limited by the scaling of all-to-all communication (e.g., PARATEC [18] and CPMD [8]) or where communication time is a significant part of the runtime.
3. Since the shared memory often renders halo (or overlap) zones unnecessary, hybrid codes can use less memory. If additional work must be performed on the halo, scalability can be enhanced by increasing the number of threads per process (e.g., FISH [10]).
4. It simplifies the load balancing of applications with dynamic or complicated structure since intra-process load balancing is possible using `dynamic` or `guided` loop scheduling (e.g., NPB BT-MZ Benchmark [21]).

It is worth noting, though, that hybrid parallelization is not always beneficial. Mahinthakumar and Saied report no improvement in a hybrid implicit finite element (FE) solver [14]. In general, there are many factors contributing to the performance of hybrid execution and results can vary between simulation setups, cf. [13].

4.1 MPI Parallelization

For the MPI parallelization of the code, we exploited techniques that have proven to be very efficient for the parallelization of general (unstructured) FE applications. Hence, we use a cell-wise distribution of the geometry. The decomposition is computed through an interface to existing graph-partitioning libraries (e.g., ParMETIS [11]). Differently than previous versions of Propag, all arrays range

only over cells and nodes and connectivity information is stored explicitly. While this change has a negative impact on single-core performance and the OpenMP scalability of the code (due to additional indirect accessing), it is compensated for by better scalability of the MPI layer.

Since the mesh in PROPAG is distributed cell-wise, nodes are duplicated on multiple processes. One of these processes is distinguished as the *owner* of the node. For inter-process communication, we use the notion of *communication traces* introduced by Sahni et al. [22]. In PROPAG a communication trace consists of a set of nodes (located on an inter-process boundary) and the rank of a peer process. On the peer, a matching communication trace is built with a consistent ordering of the interface entities. Hence, by means of a communication trace, inter-process communication is possible without the need for a global numbering of mesh entities. All communication is based on two primitives: The function `SUMUP_AT_OWNER` gathers data on the owner and `COPY_TO_OTHERS` overwrites the data at each copy by the data at the owner (scatter). These communication steps are implemented on top of non-blocking MPI send/receive calls and an extended interface (`START`, `TEST`, `WAIT`) is provided to overlap these operations with computations.

Using these communication primitives, we can rewrite Algorithm 1 as shown in Algorithm 2. The algorithm is written in such a way that it allows for overlapping communication of the diffusion currents with the computation of I_{stim} (to hide the communication in `SUMUP_AT_OWNER`) and with the evaluation of I_{ion} for the interior nodes (to hide `COPY_TO_OTHERS`), assuming the necessary hardware capabilities. In our tests, we have not seen improvements in scalability or runtime due to overlap. Nevertheless, by construction, all receive calls are preposted timely before the `WAIT` call. This is important for good MPI performance on many systems including the targeted Cray XT5.

Algorithm 2 Parallel Monodomain Explicit Euler

1: Compute locally $I_{\mathrm{dif}}^{n+1} = \beta^{-1}\nabla \cdot (\sigma' \nabla V_{\mathrm{m}}^{n})$
2: Call `SUMUP_AT_OWNER_START`(I_{dif}^{n+1})
3: Compute I_{stim}^{n+1}
4: Call `SUMUP_AT_OWNER_WAIT`(I_{dif}^{n+1})
5: Call `COPY_TO_OTHERS_START`(I_{dif}^{n+1})
6: Evaluate $I_{\mathrm{ion}}^{n+1} = \mathtt{ION_STEP}(V_{\mathrm{m}}^{n}, I_{\mathrm{dif}}^{n+1}, I_{\mathrm{stim}}^{n+1})$ for all own nodes
7: Call `COPY_TO_OTHERS_WAIT`(I_{dif}^{n+1})
8: Evaluate $I_{\mathrm{ion}}^{n+1} = \mathtt{ION_STEP}(V_{\mathrm{m}}^{n}, I_{\mathrm{dif}}^{n+1}, I_{\mathrm{stim}}^{n+1})$ for all other nodes
9: Set $V_{\mathrm{m}}^{n+1} = V_{\mathrm{m}}^{n} + \tau \left[I_{\mathrm{dif}}^{n+1} - I_{\mathrm{stim}}^{n+1} - I_{\mathrm{ion}}^{n+1} \right]$

4.2 MPI Threading Support

The intra-process parallelization via OpenMP was retained and extended to new code segments. As in the original code, we mostly use `parallel for` worksharing

constructs. This approach (in comparison to the use of large parallel sections) incurs some overhead but simplifies the implementation. Experiments with the original code (Figure 2) show that OpenMP overhead does not significantly affect the scalability of the explicit solver.

All MPI calls in PROPAG are performed outside the parallel sections. Therefore, the minimal level of thread support an MPI implementation must provide is `MPI_THREAD_FUNNELED`. As defined by the standard, this level of thread support suits applications where it is ensured that only the main thread makes MPI calls. In comparison to higher levels of thread support, this does not incur overhead due to locks/mutexes in the MPI implementation.

We do not anticipate savings in communication time by having multiple threads performing communication since the code is limited by latency rather than bandwidth. Using multiple threads for communication can be advantageous if a single thread is incapable of saturating the network interface [21].

5 Performance Analysis

All experiments were performed on a Cray XT5 machine operated by the Swiss National Supercomputing Centre. The system consists of 1844 nodes with two 6-core AMD Opteron 2.6 Ghz "Istanbul" processors per node (22,128 cores in total[4]). The nodes are connected through a Seastar 2+ interconnect.

For our experiments, we consider approximations of a model anatomy (based on CT data of a human heart obtained at autopsy [19]) at different spatial resolutions. We summarize the description of the four considered problem sizes (small, medium, large and extra-large) in Table 1.

Table 1. Problem sizes for experiments.

Name	Resolution	#cubes	#nodes
S	0.5 mm	3,024,641	3,200,579
M	0.25 mm	24,197,121	24,900,671
L	0.125 mm	193,576,968	196,390,842
XL	0.0625 mm	1,548,615,744	1,559,870,636

We study strong scaling for the problem sizes **S**, **M**, **L** and **XL**, varying both the number of processes and the number of threads per process, the latter between 1 (one MPI process per core), 6 (one MPI process per socket), and 12 (one MPI process per node). For all setups we start with at least 12 threads. We measure the average time required to perform ten Explicit Euler or IMEX Euler steps, respectively. Every tenth step, an `MPI_ALLREDUCE` is performed to sum up some statistics that have been accumulated locally. For the purpose of our tests, we do not perform significant I/O. For the IMEX runs, we use the Bi-CGSTAB solver with a Jacobi preconditioner and a fixed time step size $\tau = 0.02$ ms.

[4] Due to an interconnect congestion problem, we could not yet perform tests on more than 8448 cores.

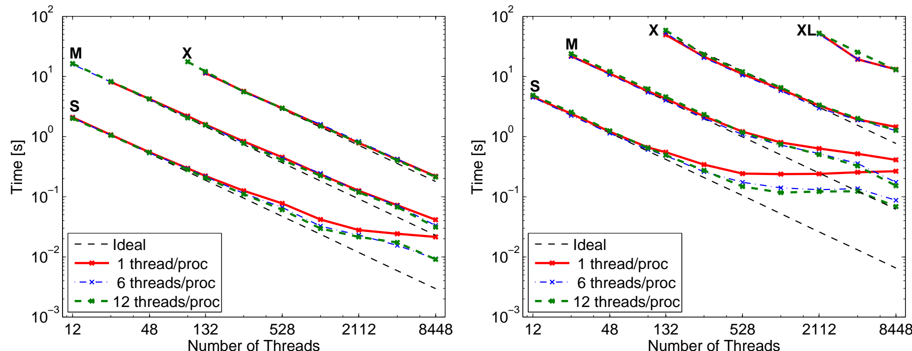

Fig. 3. Scaling of Explicit Euler (left) and IMEX Euler (right) on the Cray XT5. Problem **M** requires at least 24 cores for IMEX Euler or Explicit Euler with one thread per process. **X** requires at least 132 cores for execution (96 when using 12 threads per process). The starting point for the strong scaling study for problem **XL** is 2112 cores.

5.1 Performance of Single-Threaded Execution

In Figure 3, the time per run for the different problem sizes is plotted against the number of threads (i.e., number of processes times threads per process). The code scales well up to 8448 cores for the larger problem sizes. In general, the scaling of the Explicit Euler is much better than the IMEX Euler as the latter requires multiple `MPI_ALLREDUCE` calls per time step and additional point-to-point communication for sparse matrix-vector multiplication.

For **S** on 1056 cores (one thread per process), the IMEX Euler requires $\sim 169\times$ more `MPI_ALLREDUCE` calls than Explicit Euler. At this scale, the code spends 48.0 % of the compute time in the calls to `MPI_ALLREDUCE` (compared to 9.7 % for the Explicit Euler). Hybrid execution can improve this situation, see Section 5.2. Nevertheless, for this small problem size, the code still achieves an efficiency of 56.5 % and 21.9 % on 1056 cores using the Explicit Euler and IMEX Euler, respectively. For larger problems, such as **L**, the parallel efficiency on 8448 cores relative to 132 cores (the minimum required to run the problem) is 81.6 % and 53.2 % for Explicit Euler and IMEX Euler, respectively.

The limits in (strong) scalability of PROPAG can be linked to two major sources of inefficiency: A relative increase in communication time and a sub-optimal decrease in the degrees of freedom per process.

In Table 2, we report the relative percentage of the average walltime of communication in the main computational loop as reported by the Integrated Performance Monitor (IPM) [9]. The data show that there is an $\sim 4\times$ increase in the relative communication time (both point-to-point and collective) when increasing the number of cores by a factor of 8.

In Table 3 we show the increase in the total number of nodes due to the overlap between subdomains. Due to the cell-based decomposition, nodes on inter-process boundaries must be duplicated so that the total number of nodes (where copies are accounted for) grows with the number of processes. As can be

Table 2. Breakdown of communication time for **S** using Explicit and IMEX integration with one thread per process.

#cores	% of walltime in point-to-point communication	% of walltime in collective communication
Explicit Euler		
132	4.91 %	2.31 %
1056	20.10 %	10.07 %

#cores	% of walltime in point-to-point communication	% of walltime in collective communication
IMEX Euler		
132	13.04 %	12.31 %
1056	32.57 %	48.09 %

Table 3. Characteristics of the node distribution during scale-out of **M**.

#procs	12	24	528	1056	4224	8448
% Increase in #nodes	1.58	2.33	10.14	13.25	22.55	29.67

seen in Table 3, the number of nodes has grown by almost 30 % on 8448 cores. Using an argument similar to that of Amdahl's law, we can derive an upper bound for the parallel efficiency as the ratio between the total number of nodes in serial and parallel. In our example, the maximum attainable efficiency when scaling from 12 to 8448 cores is 78.3 %. A similar finding was reported by Sahni et al. [22] in the context of an unstructured FE solver.

5.2 Benefits of Hybrid Execution

In Section 5.1, we have identified two major sources of scalability loss in Propag. In this section, we will analyze how hybrid execution, using multiple threads per process, allows to mitigate these inefficiencies.

In Table 4, we present a breakdown of the communication time for the problem size **S**. The results for runs with one thread per process correspond to the results in Table 2. Unlike before, Table 4 contains absolute communication times (for 1010 time steps) to allow for comparing the results from different runs. Our results show that the use of multiple threads per process can significantly reduce the communication time. Using 6 or 12 threads per process reduces the time in `MPI_ALLREDUCE` by 22–52% or up to 61 %, respectively. Similarly, T_{Pt2Pt} is decreased by 22–64 % or 5–72 % for 6 or 12 threads. Interestingly though, a smaller number of processes does not always imply lower communication cost since the T_{Pt2Pt} for 11×12 threads is larger than for 22×6 threads. Using more threads per process leads to larger buffer sizes. This results in an improved bandwidth utilization but also increased latency.

In Section 5.1, we have noted that a strict upper limit for the parallel efficiency in Propag exists due to the growth of node copies on inter-process boundaries. For the intra-process parallelization based on OpenMP worksharing constructs, no overlap is required. When keeping the total number of threads

Table 4. Breakdown of communication time for **S** using Explicit and IMEX Euler. T_{Pt2Pt} and T_{Coll} denote point-to-point and collective communication time, respectively.

#cores	procs × threads/proc	T_{Pt2Pt}	T_{Coll}
Explicit Euler			
132	132×1	5.12 s	2.41 s
	22×6	3.99 s	1.37 s
	11×12	4.87 s	2.34 s
1056	1056×1	3.90 s	1.95 s
	176×6	2.43 s	0.95 s
	88×12	2.25 s	0.76 s

#cores	procs × threads/proc	T_{Pt2Pt}	T_{Coll}
IMEX Euler			
132	132×1	35.53 s	33.55 s
	22×6	20.86 s	25.89 s
	11×12	12.18 s	14.99 s
1056	1056×1	38.13 s	56.29 s
	176×6	13.73 s	39.81 s
	88×12	10.52 s	33.46 s

Table 5. Percentage increase in #nodes for **M** with 1, 6, and 12 threads per process.

#cores / threads	12	24	528	1056	4224	8448
1	1.58	2.33	10.14	13.25	22.55	29.67
6	0.40	0.84	4.82	6.55	11.31	14.87
12	0.00	0.40	3.33	4.82	8.79	11.31

constant, using more threads per process will result in fewer node copies. In Table 5, we show that this results in a strong reduction of the number of additional nodes. Consequently, the theoretical upper bound for the efficiency improves: When using 12 threads per process, efficiency when going from 12 to 8448 cores is bounded by 89.8 % (rather than 78.3 %, cf. Section 5.1). We measure an efficiency of 74% for the Explicit Euler solver which seems to be practically impossible to achieve with a pure MPI version.

The actual, measured improvement of the hybrid code (running with 6 or 12 threads per process, respectively) is shown in Figure 4. For the case of the Explicit Euler, threaded execution is beneficial starting at 96 cores. The code on 1056 cores with 6 threads per process shows an unexpectedly bad performance that we cannot explain yet. For the IMEX Euler, which is more strongly limited by communication time, execution with 6 threads per process is advantageous already at 24 cores; execution with 12 threads per process is advantageous for 528 cores or more. When 2112 cores or more are used, running with 12 threads per process is faster than running with 6 threads per process.

6 Conclusion

We have presented the successful hybrid parallelization of a large-scale heart model. Performance was measured in monodomain simulations with up to 1.5 billion nodes. These system sizes are among the largest reported in the literature for this scientific problem.

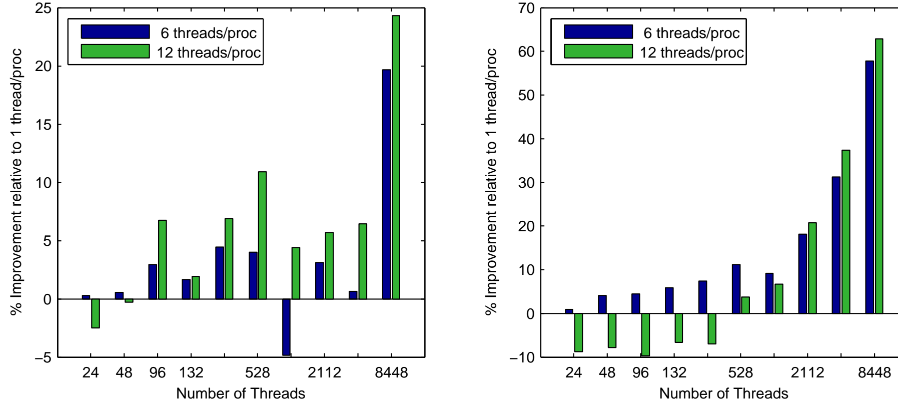

Fig. 4. Improvement through hybrid execution for Explicit (left) and IMEX Euler (right) relative to pure MPI for **M** on the Cray XT5.

We have shown that hybrid parallelization can improve scalability of this application as it 1) decreases the relative and absolute communication time and 2) reduces the size of the overlap between adjacent subdomains. We have analyzed both effects separately and have demonstrated runtime reductions up to 24 % for an Explicit Euler and up to 62 % for an IMEX Euler time discretization.

Acknowledgments

Computational resources for this work were provided by the Università della Svizzera italiana (USI), the Swiss National Supercomputing Centre (CSCS), and the Réseau québécois de calcul de haute performance (RQCHP). This work was supported by the project “A High Performance Approach to Cardiac Resynchronization Therapy” within the context of the “Iniziativa Ticino in Rete” and the “Swiss High Performance and Productivity Computing” (HP2C) Initiative.

References

1. R. Bordas, B. Carpentieri, G. Fotia, F. Maggio, R. Nobes, J. Pitt-Francis, and J. Southern. Simulation of cardiac electrophysiology on next-generation high-performance computers. *Phil. Trans. Roy. Soc. A.*, 367:1951–1969, 2009.
2. T. Desplantez, E. Dupont, N. J. Severs, and R. Weingart. Gap junction channels and cardiac impulse propagation. *J. Membrane Biol.*, 218:13–28, 2007. (review).
3. S. Ethier, W. M. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *J. Phys. Conf. Ser.*, 16(1):1–15, 2005.
4. C. S. Henriquez. Simulating the electrical behavior of cardiac tissue using the bidomain model. *CRC Crit. Rev. Biomed. Eng.*, 21:1–77, 1993.
5. B. Hille. *Ion Channels of Excitable Membranes.* Sinauer Associates, Inc, Sunderland, MA, USA, 2001.

6. M. G. Hoogendijk et al. Mechanism of right precordial ST-segment elevation in structural heart disease: Excitation failure by current-to-load mismatch. *Heart Rhythm*, 7:238–248, 2010.
7. N. Hooke, C. S. Henriquez, P. Lanzkron, and D. Rose. Linear algebraic transformations of the bidomain equations: Implications for numerical methods. *Math. Biosci.*, 120(2):127–145, 1994.
8. J. Hutter and A. Curioni. Dual-level parallelism for ab initio molecular dynamics: Reaching teraflop performance with the CPMD code. *Parallel Comput.*, 31(1):1–17, 2005.
9. IPM Homepage, 2009. http://ipm-hpc.sourceforge.net/.
10. R. Kaeppeli, S. C. Whitehouse, S. Scheidegger, U. L. Pen, and M. Liebendörfer. FISH: A 3D parallel MHD code for astrophysical applications. Technical Report arXiv:0910.2854, 2009.
11. G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Parallel Processing for Scientific Computing*. SIAM, 1997.
12. A. Kléber and Y. Rudy. Basic mechanisms of cardiac impulse propagation and associated arrhythmias. *Physiol. Rev.*, 84:431–488, 2004.
13. R. Loft, S. Thomas, and J. Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *Supercomputing, ACM/IEEE 2001 Conference*, 2001.
14. G. Mahinthakumar and F. Saied. A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. *Int. J. High Perform. C.*, 16(4):371–393, 2002.
15. L. Mitchell, M. Bishop, E. Hötzl, A. Neic, M. Liebmann, G. Haase, and G. Plank. Modeling cardiac electrophysiology at the organ level in the peta flops computing age. *AIP Conference Proceedings*, 1281(1):407–410, 2010.
16. S. Niederer, L. Mitchell, N. Smith, and G. Plank. Simulating a human heart beat with near-real time performance. *Front. Physio.*, 2:14, 2011.
17. D. Noble and Y. Rudy. Models of cardiac ventricular action potentials: Iterative interaction between experiment and simulation. *Phil. Trans. Roy. Soc. London; Phys. Sc.*, 359:1127–1142, 2001.
18. PARAllel Total Energy Code. http://www.nersc.gov/projects/paratec.
19. M. Potse, B. Dubé, J. Richer, A. Vinet, and R. M. Gulrajani. A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. *IEEE Trans. Biomed. Eng.*, 53:2425–2435, 2006.
20. M. Potse, B. Dubé, and A. Vinet. Cardiac anisotropy in boundary-element models for the electrocardiogram. *Med. Biol. Eng. Comput.*, 47:719–729, 2009.
21. R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.
22. O. Sahni, M. Zhou, M. S. Shephard, and K. E. Jansen. Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores. In *Supercomputing, ACM/IEEE 2009 Conference*, 2009.
23. K. H. ten Tusscher and A. V. Panfilov. Alternans and spiral breakup in a human ventricular tissue model. *Am. J. Physiol.*, 291(3):H1088–1100, 2006.
24. N. Trayanova and F. Aguel. Computer simulations of cardiac defibrillation: A look inside the heart. *Comput. Vis. Sci.*, 4:259–270, 2002.
25. E. J. Vigmond, F. Aguel, and N. A. Trayanova. Computational techniques for solving the bidomain equations in three dimensions. *IEEE Trans. Biomed. Eng.*, 49:1260–1269, 2002.