33rd Annual International Conference of the IEEE EMBS
Boston, Massachusetts USA, August 30 - September 3, 2011

# Petascale Computation Performance of Lightweight Multiscale Cardiac Models using Hybrid Programming Models

Bernard J Pope, Blake G Fitch, *Member IEEE*, Michael C Pitman, John J Rice, Matthias Reumann, *Member, IEEE*

*Abstract*—**Future multiscale and multiphysics models must use the power of high performance computing (HPC) systems to enable research into human disease, translational medical science, and treatment. Previously we showed that computationally efficient multiscale models will require the use of sophisticated hybrid programming models, mixing distributed message passing processes (e.g. the message passing interface (MPI)) with multithreading (e.g. OpenMP, POSIX pthreads). The objective of this work is to compare the performance of such hybrid programming models when applied to the simulation of a lightweight multiscale cardiac model. Our results show that the hybrid models do not perform favourably when compared to an implementation using only MPI which is in contrast to our results using complex physiological models. Thus, with regards to lightweight multiscale cardiac models, the user may not need to increase programming complexity by using a hybrid programming approach. However, considering that model complexity will increase as well as the HPC system size in both node count and number of cores per node, it is still foreseeable that we will achieve faster than real time multiscale cardiac simulations on these systems using hybrid programming models.**

## I. Introduction

MULTISCALE, multiphysics models require supercomputing resources to perform basic research into human diseases and disease progression [1-3]. Message passing has been a programming model that allows the design of parallel programs on massively parallel, distributed memory supercomputers like the IBM Blue Gene supercomputer. With the current trend of having multiple compute cores on a physical node, hybrid programming models can offer multithreading on the compute node while message passing is used for communication between nodes. In our previous work [4] we showed that hybrid programming models have an advantage over the pure message passing model. However, in that study we used a model that was computation bound. In this article we want to compare hybrid programming models of computationally lightweight multiscale cardiac models.

Manuscript received, March 24th 2011. This research was supported by a Victorian Life Sciences Computation Initiative (VLSCI) grant number VR0088 on its Peak Computing Facility at the University of Melbourne, an initiative of the Victorian Government.

B. J. Pope is with the Victorian Life Science Computation Initiative, 187 Grattan Street, Carlton, VIC 3010, Australia (e-mail: bjpope@unimelb.edu.au).

B. G. Fitch, M. C. Pitman and J. J. Rice are with the IBM T. J. Watson Research Center, 1101 Kitchawan Road, Yorktown Heights, NY, 10598 USA (e-mail: {bgf, pitman, johnrice}@us.ibm.com).

M. Reumann is with the IBM Research Collaboratory for Life Sciences-Melbourne, 187 Grattan Street, Carlton, VIC 3010, Australia and the Dept. Computer Science and Software Engineering, University of Melbourne (corresponding author phone: +61 3 9035 4432 e-mail: m reumann@ieee.org).

## II. Methods

The method used in this study has the same setup as described in [4] except for changing the cell model for the multiscale cardiac simulations. This is required to be able to perform comparisons between computationally heavy models and the lightweight model used in this study. We give an overview of our method in the following section.

### A. Architecture of the IBM Blue Gene/P supercomputer

All of our tests were performed on an IBM Blue Gene/P (BG/P). It is a massively parallel, distributed memory supercomputer. We use its high speed torus network for communication between compute nodes. Each compute node has four 850 MHz PowerPC cores connected to a local memory unit of 4GB. BG/P can be configured to run in one of three modes: *SMP* – one process per node, up to four threads per process; *Dual* – two processes per node, up to two threads per process; *Virtual Node (VN)* – four separate processes per node, one thread per process.

Threads share the address space of their containing process, whereas each process has a distinct address space. Thus, in the SMP model the process has a 4GB address space that all four threads can access while in VN mode each process has only 1GB address space. With regards to these particulars, a future looking programming model from multiscale, multiphysics biomedical simulations needs to make use of this particular system architecture by using hybrid programming models for computational efficiency.

### B. Data decomposition and communication framework

As previously described [3-6] we apply the orthogonal recursive bisection algorithm (ORB) for data decomposition. This allows us to create a number of subvolumes that corresponds to the number of MPI tasks. The communication framework ensures that the values of adjacent subvolumes are stored in a ghost layer around the subvolume associated with the respective MPI task so that the diffusion term can be computed without introducing errors. This communication framework is the same for all three programming models we investigated.

### C. Programming models

#### 1) MPI model

The simplest approach regarding programming complexity is a purely MPI implementation because it only involves a single parallel programming interface. We will call this the *MPI model* and it serves as baseline implementation for our comparisons. By using the VN mode of the BG/P each MPI process runs on a separate core, giving four processes per node.

*2) MPI and OpenMP Model*

Programming complexity increases by adding parallelism in shared memory on each compute node. We use the OpenMP application programmer interface to define regions in the source code that are to be parallelized with so called `#pragma` statements. OpenMP then handles thread creation and synchronization without the user requiring an understanding of threads. We parallelize the reaction-diffusion computation only and let the master thread carry out the communication. The `#pragma omp parallel default(shared)` directive declares the parallel region of the program which comprises the computation phase. The diffusion term is then parallelized using the `#pragma omp for nowait` directive which allows the threads to carry on the computation beyond the end of the loop. We enforce a synchronization step after the computation of the reaction part by using the same directive without the `nowait` option. The same holds for the synchronization step after the communication phase which is implicitly enforced by using the MPI's `wait` function. We automatically have two barrier functions for synchronization and thus, data coherency is ensured. We will call this implementation the *MPI+OpenMP* hybrid programming model that allows the user to semi-automatically add threads to the original MPI code.

*3) MPI and POSIX pthreads Model*

Our second hybrid programming model uses MPI for inter-node communication and POSIX pthreads for intra-node computation. The programming complexity increases because the programmer has to understand thread creation and synchronization. Also, data structures have to be created manually for the parallel regions of the code. We adopt the peer programming model where the master thread takes part in the computation. Thus, all threads carry out the same amount of computational work. However, our implementation, the master does communicate while the worker threads are paused. Synchronisation is carried out by the pthreads barrier primitive, which ensures data coherency. We call this approach the *MPI+pthreads* model.

Both hybrid programming models enable us to reduce the level of data decomposition by two bisections: instead of e.g. 16,384 MPI tasks we have 4,096 MPI tasks with four threads each. The reduced number of MPI tasks leads to a smaller number of subvolumes. Since the transmembrane voltage of tissue at the surface of the subvolumes needs to be communicated only, the global communication load on the network is reduced because the sum of surfaces of the smaller number of subvolumes is smaller than for the larger number of subvolumes. On each SMP node the computation load is then simply divided equally by four (the number of threads).

The MPI+OpenMP implementation has the same communication-computation framework as the MPI+pthreads implementation. The difference between the two is that using pthreads we have to explicitly define the synchronization of pthreads and carry out the data decomposition for each thread manually. Both are hidden from the programmer using OpenMP.

## D. Lightweight Multiscale Cardiac Model

To create a lightweight multiscale cardiac model for propagation studies, we use the monodomain reaction-diffusion model of cardiac excitation [7]. The model equation is given by

$$\nabla \cdot \left(\sigma \nabla V_m\right) = A_m \left( C_m \frac{\partial V_m}{\partial t} + I_{ion} \right) - I_s \qquad \text{(equation 1)}$$

The transmembrane potential $V_m$ describes the cellular excitation. The surface to volume ratio and the membrane capacitance is given by $A_m$ and $C_m$, respectively. $\sigma$ is the anisotropic conductivity tensor and $I_s$ the stimulus currents. The ionic current density $I_{ion}$ is computed by the electrophysiological cell models in the simulations. To have a computationally light weight model we choose the FitzHugh-Nagumo model [8, 9] to compute the reaction term. The ventricles of the Visible Female data set (National Library of Medicine, Bethesda, Maryland, USA) defines the anatomy with 0.2 mm cubic resolution resulting in 32.5 million active elements in a data set of 128.9 million elements. We use explicit Euler method to solve the reaction term and finite difference method for the diffusion part. Our program flow is given by

- Initialization
- Time step loop
    - Communication phase
    - Computation of diffusion term (loop)
    - Computation of reaction term (loop)

For benchmarking we carry out strong scaling simulations for all three programming models described in section II.D. We record overall maximum run time as well as computation and communication times. We simulate 1000 time steps with 0.01 ms per time step yielding a 10 ms simulation. The smallest computer partition size used is 128 SMP nodes equaling 512 CPUs. We increase the number of SMP nodes by powers of two. The largest computer partition available to us is a four rack IBM Blue Gene/P supercomputer with 16,384 cores. As a measure of load balance we also show the ratio of average vs. maximum time for both computation and communication. As this value reaches one, load balance becomes ideal. Finally, we will show and discuss the speedup factors with reference to the 128 SMP node partition.

## III. Results

The run times decrease linearly with the number of cores for all programming models (Fig. 1, Tab. I). The fastest run times are given by the baseline

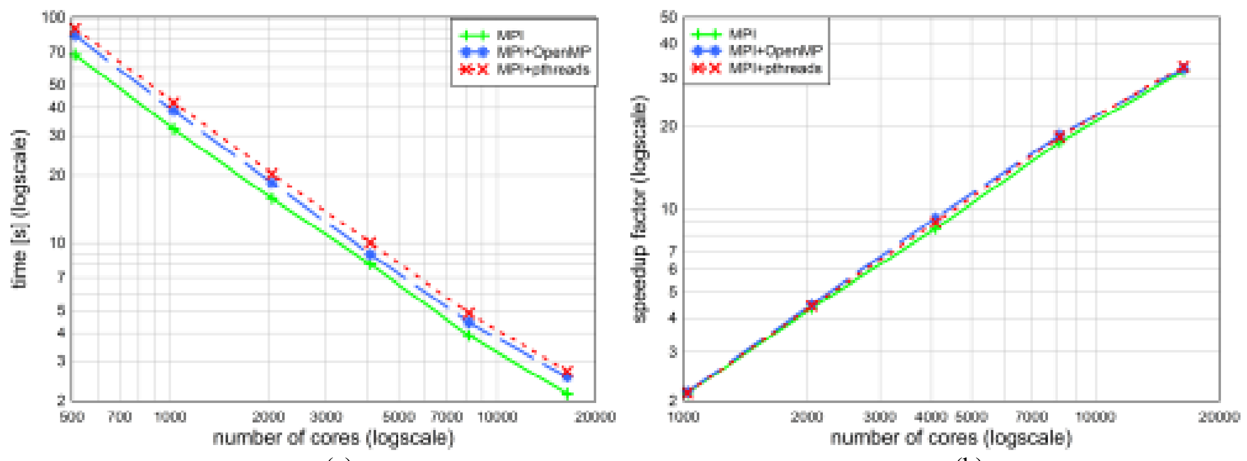

Fig. 1 Performance results: (a) maximum run times (b) overall scaling with respect to the smallest processor partition N = 512 cores

TABLE I
MAXIMUM RUN TIMES AND SPEEDUP FACTORS

| | Max. run time [s] | | | Speedup vs. next smaller N | | | Overall speedup | | |
|---|---|---|---|---|---|---|---|---|---|
| N cores | VN | OpenMP | pthreads | VN | OpenMP | pthreads | VN | OpenMP | pthreads |
| 512 | 68.28 | 82.83 | 89.67 | - | - | - | - | - | - |
| 1,024 | 32.01 | 38.51 | 42.04 | 2.13 | 2.15 | 2.13 | 2.13 | 2.15 | 2.13 |
| 2,048 | 15.72 | 18.46 | 20.27 | 2.04 | 2.09 | 2.07 | 4.34 | 4.49 | 4.42 |
| 4,096 | 8.06 | 8.91 | 10.01 | 1.95 | 2.07 | 2.02 | 8.48 | 9.30 | 8.96 |
| 8,192 | 3.89 | 4.45 | 4.88 | 2.07 | 2.00 | 2.05 | 17.56 | 18.62 | 19.36 |
| 16,384 | 2.14 | 2.54 | 2.70 | 1.81 | 1.75 | 1.80 | 31.87 | 32.59 | 33.18 |

TABLE II
TIMING FOR REACTION – DIFFUSION PHASE

| | MPI | | | MPI+OpenMP | | | MPI+pthreads | | |
|---|---|---|---|---|---|---|---|---|---|
| N cores | avg. [s] | max. [s] | avg/max | avg. [s] | max. [s] | avg/max | avg. [s] | max. [s] | avg/max |
| 512 | 59.11 | 67.32 | 0.88 | 65.93 | 80.81 | 0.82 | 73.70 | 87.85 | 0.84 |
| 1,024 | 26.31 | 31.37 | 0.84 | 31.66 | 37.30 | 0.85 | 35.65 | 40.78 | 0.87 |
| 2,048 | 12.42 | 15.35 | 0.81 | 15.19 | 17.44 | 0.87 | 17.20 | 19.26 | 0.89 |
| 4,096 | 5.98 | 7.61 | 0.79 | 6.80 | 8.28 | 0.82 | 7.80 | 9.41 | 0.83 |
| 8,192 | 2.92 | 3.62 | 0.81 | 3.26 | 4.01 | 0.81 | 3.73 | 4.55 | 0.82 |
| 16,384 | 1.43 | 1.93 | 0.74 | 1.62 | 1.98 | 0.82 | 1.83 | 2.27 | 0.81 |

TABLE III
TIMING FOR COMMUNICATION PHASE

| | MPI | | | MPI+OpenMP | | | MPI+pthreads | | |
|---|---|---|---|---|---|---|---|---|---|
| N cores | avg. [s] | max. [s] | avg/max | avg. [s] | max. [s] | avg/max | avg. [s] | max. [s] | avg/max |
| 512 | 9.12 | 36.01 | 0.25 | 16.80 | 39.16 | 0.43 | 15.87 | 39.95 | 0.40 |
| 1,024 | 5.68 | 31.99 | 0.18 | 6.82 | 33.86 | 0.20 | 6.35 | 23.74 | 0.27 |
| 2,048 | 3.27 | 15.71 | 0.21 | 3.25 | 10.01 | 0.32 | 3.05 | 10.86 | 0.28 |
| 4,096 | 2.06 | 8.05 | 0.26 | 2.09 | 8.91 | 0.23 | 2.19 | 10.01 | 0.22 |
| 8,192 | 0.96 | 3.88 | 0.25 | 1.17 | 4.45 | 0.26 | 1.14 | 4.88 | 0.23 |
| 16,384 | 0.70 | 2.14 | 0.33 | 0.91 | 2.54 | 0.36 | 0.86 | 2.70 | 0.32 |

MPI model in VN mode. Here, the run time on 512 cores is 68.28s which comes down to 2.14s on 16,384 cores. For the *MPI+OpenMP* and *MPI+pthread* models the times are 82.83s down to 2.54s and 89.67s to 2.70s, respectively. Thus, the *MPI+OpenMP* model performs better than the *MPI+pthreads* model. Looking at the speedup factors (Tab. I) the results demonstrate that both hybrid programming models show higher speedup factors than the non-hybrid model. The overall speedup factors for the baseline at 512 cores are 31.87, 32.59 and 33.18 for the *MPI*, *MPI+OpenMP* and *MPI+pthreads* model, respectively. Indeed, all speedup factors are above the theoretical value for all programming models and all partitions. The exception is the *MPI* model where the speedup factor of 31.87 from 512 to 16.384 cores is just below the theoretical value of 32.

The load balance measure for computation phase shows values above 0.8 for mostly all programming models (Tab. II) and partitions which demonstrates that the ORB algorithm for data decomposition of the computation phase is reasonable. In general, the load balance on partitions with up to 2,048 cores is better than for the larger partitions. The load balance for the communication phase (Tab. III) is not ideal with values between 0.18 to 0.33 for the *MPI* model, 0.20 to 0.43 and 0.22 – 0.40 for the *MPI+OpenMP* and *MPI+pthreads* model, respectively. While the MPI models load balance measure for the communication time is below the values of the hybrid programming models, it has the lower values for the maximum communication time for all partitions.

## IV. DISCUSSION

The speedup factors show that the simulations scale well for all programming models to 16,384 cores despite the poor load balance in the communication phase. The speedup is best for the *MPI+OpenMP* model up to 2,048 cores. Then the *MPI+pthread* model takes over to yield the highest speedup factor for 16,384 cores. Thus, with respect to speedup and efficiency, the *MPI+pthread* hybrid programming model with the highest programming complexity outperforms the others on the largest partition used in this study.

Having said that, the maximum run times show the opposite picture. The *MPI* model is 1.17±0.065 times faster than the *MPI+OpenMP* model and 1.28±0.036 times faster than the *MPI+pthread* model across all partitions. This is in contrast to the results in [4] where the hybrid programming models overall perform better than the *MPI* model with the *MPI+OpenMP* model showing the best performance. Since we use a very lightweight cardiac model that has a ten times faster execution time compared with the model in [4], the simulation is less compute bound. This leads to the communication overhead to have a greater effect on the simulation. The hybrid programming models have an additional synchronization step, in the form of a barrier primitive prior to the communication phase, which is not needed in the *MPI* model. We believe this leads to the slower run times for the hybrid programming models.

However, our implementation using hybrid programming models does not take advantage of all threads in the communication phase. Currently, the worker threads are waiting without doing work during that phase. Thus, there is potential to improve the performance. This can be achieved by either two strategies. Either all worker threads participate in the communication phase as well, or the worker threads start the computation of the inner tissue elements in the subvolume while the master carries out the communication of border values. Once this has been completed, the reaction-diffusion equations can be solved for those as well. Hence, hybrid programming models allow for more sophisticated program design with regards to interleaving computation and communication phases.

Also, our implementation the computational load only is considered during data decomposition. Thus, further performance improvement can be achieved by more sophisticated load balance strategies. Nevertheless, our current implementation of the lightweight multiscale cardiac model achieves a run time of just over 3.5 minutes for a 1000 ms simulation run.

## V. CONCLUSION

This work indicates that a model developer does not have to add programming complexity by using hybrid programming models for lightweight multiscale cardiac models. However, the communication and computation phases are kept separate, which is a simplification that can be improved upon. Making use of the computing power of the worker threads during the communication phase will certainly increase performance. We believe the performance advantage of hybrid models will be more apparent on future HPC systems because they will have many more hardware threads than current systems.

We acknowledge that more sophisticated and optimized multiscale cardiac models exist. Bearing that in mind, it is easily foreseeable that combining our methods and those models will yield faster than real time simulation run times in the near future.

## ACKNOWLEDGMENT

We would like to thank Fred Mintzer and David Singer for enabling us to use of the IBM Blue Gene/P supercomputer at the IBM T. J. Watson Research Laboratory.

## REFERENCES

[1] A. Peters, S. Melchionna, E. Kaxiras et al. Multiscale simulation of cardiovascular flows on the IBM Blue Gene/P: full heart-circulation system at near red-blood cell resolution, Supercomputing 2010

[2] A. Hosoi, T. Washio, J. Okada et al. A Multi-Scale Heart Simulation on Massively Parallel Computers, Supercomputing 2010

[3] M. Reumann, B. G. Fitch, A. Rayshubskiy, M. C. Pitman, J. J. Rice, Orthogonal Recursive Bisection as Data Decomposition Strategy for Massively Parallel Cardiac Simulations, Biomed Tech 2011; 56:129-145

[4] B. J. Pope, B. G. Fitch, J. J. Rice, M. Reumann. Performance of Hybrid Programming Models for Multiscale, Cardiac Simulations: Preparing for Petascale Computation. IEEE Trans Biomed Eng. (submitted March 2011)

[5] M. Reumann, B. G. Fitch, A. Rayshubskiy et al. Strong scaling and speedup to 16,384 processors in cardiac electro-mechanical simulations, 31st Annual International IEEE EMBS Conference Minneapolis, MN, USA, September 2-6, 2009

[6] M. Potse, B. Dube, 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. 2006;53(12):2425-2435

[7] R. FitzHugh: Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal* 1961, 1:445-466.

[8] J. Sundnes, G. T. Lines, X. Cai et al.. Computing the Electrical Activity in the Heart. Berlin Heidelberg: Springer Verlag; 2006.