Enabling computer models of the heart for high-performance computers and the grid

BY JOE PITT-FRANCIS, ALAN GARNY AND DAVID GAVAGHAN

1Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
2Department of Physiology, Anatomy and Genetics, University of Oxford, Parks Road, Oxford OX1 3PT, UK

Although it is now feasible to compute multi-cellular models of the heart on a personal desktop or laptop computer, it is not feasible to undertake the detailed sweeps of high-dimensional parameter spaces required if we are to undertake in silico experimentation of the complex processes that constitute heart disease. For this research, modelling requirements move rapidly beyond the limit of commodity computers’ resource both in terms of their memory footprint and the speed of calculation, so that multi-processor architectures must be considered. In addition, as such models have become more mature and have been validated against experimental data, there is increasing pressure for experimentalists to be able to make use of these models themselves as a key tool for hypothesis formulation and in planning future experimental studies to test those hypotheses.

This paper discusses our initial experiences in a large-scale project (the Integrative Biology (IB) e-Science project) aimed at meeting these dual aims. We begin by putting the research in context by describing in outline the overall aims of the IB project, in particular focusing on the challenge of enabling novice users to make full use of high-performance resources without the need to gain detailed technical expertise in computing. We then discuss our experience of adapting one particular heart modelling package, Cellular Open Resource, and show how the solving engine of this code was dissected from the rest of the package, ported to C++ and parallelized using the Message-Passing Interface. We show that good parallel efficiency and realistic memory reduction can be achieved on simple geometries. We conclude by discussing lessons learnt in this process.

Keywords: cardiac modelling; high-performance computing; grid computing

1. Introduction

Over the last decade, the breathtaking success of high-throughput approaches to reductive biological research has yielded a wealth of biological data. As a result, attention is now shifting to the next stage of this process where the research community is attempting to integrate this information in an attempt to
understand how biological function emerges. This research involves an iterative interplay between laboratory experiment, mathematical modelling and \textit{in silico} experimentation, and this new research paradigm is often referred to as integrative systems biology. As a result, there is a pressing need to create computational and IT infrastructures to support this activity, and to allow non-expert users to gain access to, and make best use of, state-of-the-art approaches to data management and data analysis tools, and to the high-performance computing (HPC) facilities required to undertake the \textit{in silico} experimentation.

This requirement for an infrastructure to support large-scale data and computing requirements is not limited to the biological sciences, and the necessary IT research activity has resulted in a global research effort to develop what is termed cyberinfrastructure in the USA, and e-Science in the UK (Hey & Trefethen 2002), built upon ‘grid’ technologies (Foster \textit{et al.} 2001). In this paper, we describe some of the initial research effort undertaken in one of the larger projects funded under the UK’s e-Science Programme, the Integrative Biology (IB) project.\textsuperscript{1} One of the key aims of the IB project as a whole is to build an infrastructure to enable medical scientists and non-expert computer users to make full use of modern computing resources. The project is focusing initially on two disease areas—whole heart modelling and cancer modelling. The former is perhaps the most mature area of physiological modelling, with the fundamental paper in the field having been written in 1960 by Denis Noble (1960), and current models incorporating detailed descriptions at cellular, tissue and organ levels. Cancer modelling is an area of much research activity within the UK, but to date very little work has been done in integrating cancer models across spatial and temporal scales.

In this paper, we focus on a preliminary study undertaken to determine the issues involved in whole heart modelling, and to inform the requirements gathering exercise for developing the IB Grid infrastructure. We therefore begin by giving a brief overview of the main goals of the IB project. We then go on to discuss the central role of HPC in systems biology research, before giving a detailed description of our experience in taking a desktop-bound code, the Cellular Open Resource (COR, Garny \textit{et al.} 2003\textsuperscript{b}) for modelling heart physiology, and migrating it to a computer cluster, as a typical case study in parallelizing codes of this type. We conclude by discussing the lessons learnt in this process.

2. The role of Grid technologies in supporting the systems biology research process

As discussed earlier, the current study was undertaken as part of the IB Project to gain insight into the research process underpinning \textit{in silico} experimentation in heart physiology as an exemplar of its role in systems biology more generally. The wider goals of the project are to build the IT infrastructure, based upon Grid technologies (Foster \textit{et al.} 2001), together with truly \textit{usable} interfaces to that infrastructure, to allow the routine use of such \textit{in silico} techniques by non-expert

\footnote{See Gavaghan \textit{et al.} (2005) for an overview of Grid technology, the aims and objectives of both the e-Science Programme in the UK, and of the Integrative Biology pilot project.}
laboratory scientists to support, interpret and guide their wet-lab experiments. In brief, the key features that the system must possess to meet these goals are:

(i) Implementation of the necessary simulation codes on appropriate platforms, including national HPC facilities such as HPCx, CSAR, the data and compute clusters of the National Grid Service, and the Atlas Data Store at the Rutherford Appleton Laboratory.

(ii) Integration of the different model components, and development of tools to manage execution of distributed simulations.

(iii) Development of mechanisms to assimilate experimental and clinical data with simulation results.

(iv) Development of mathematical modelling tools and services.

(v) Provision of fault-tolerance by regularly checkpointing the simulations during execution.

(vi) Ensuring efficient and secure data transfer, storage and management of the individual datasets produced in each simulation together with a metadata catalogue containing information about all simulations.

(vii) Analysis and visualization of the results of the simulations, both individually and comparatively, including exploration of very high dimensional parameter spaces.

(viii) Provision of real-time interaction with running simulations (‘computational steering’) so that the user can monitor their progress, observe their developing state and steer the subsequent course of the simulation.

(ix) Ability to allocate and co-allocate one or several resources, including preemptive access to resources in the second case.

(x) Management of accounts and authentication across all resources in the testbed, to enable monitoring and migration across the heterogeneous network (networks of workstations, clusters, CSAR and HPCx).

(xi) Provision of performance control by optimizing the deployment of resources in a dynamically varying Grid environment (requiring the monitoring of progress and performance-based decision algorithms to use efficient checkpoint/restart and the ability to migrate across architectures), and to monitor operational parameters to enable optimization of performance.

(xii) Provision of a simple-to-use interface which focuses on the physiological processes involved and can either expose or hide as many of the implementation features as is appropriate for the user’s task.

(xiii) Provision of a comprehensive set of tools to support workflow.

(xiv) Provision of tools to support sharing of the interactive features of the environment and audio/visual communication (e.g. AccessGrid\(^3\)) between researchers at different locations working either concurrently or at different times.

From a software engineering perspective, the IB infrastructure is based on a Service Oriented Architecture, and is focused on sharing resources or access to shared resources. The project is utilising an object-oriented component-based

\(^2\) Further details of the project can be found in Gavaghan et al. (2005) and on the project website at www.integrativebiology.ox.ac.uk.

\(^3\) See www.accessgrid.org.
toolkit to provide a layer of abstraction above our application codes and the middleware (web services) that directs the resource management software. This toolkit will be incorporated into a Virtual Research Environment (VRE) based on the OGCE platform, by which we mean a single desktop environment with a single sign-on that will support all aspects of the researcher’s work. The project builds upon several existing e-Science projects, and in particular will build upon, extend and combine the workflow support tools being developed in the myGrid project (Stevens et al. 2003), and the HPC simulation support tools being developed by the RealityGrid (Harting et al. 2003), and gViz projects (Brodlie et al. 2004).

Workflow capture is particularly important when attempting to provide easy access to high-performance facilities to novice users. Our goal here is to capture entire simulation workflows of experienced in silico modellers in editable form so that they can be re-enacted by novice users (via a workflow engine).

(a) Example of user interaction with the VRE

This brief technical overview masks the way in which such an infrastructure might actually support researchers in practice. An example of a possible user interaction with the VRE therefore follows.

(i) An expert user logs into the VRE through a single, secure sign-on, gaining access to all tools and resources to which he/she is authorized.
(ii) The user checks e-mail and diary.
(iii) The user sets up a new in silico experiment to look at a possible mechanism for the initiation of arrhythmias in a three-dimensional model of the whole heart by editing an existing workflow (from a library of workflows) with the aid of intelligent workflow support tools.
(iv) The user submits the workflow, and the IB infrastructure then initiates a job on a remote HPC facility, together with a steering, visualization and data management client.
(v) The user, waiting for results to be generated, accesses and analyses (within the VRE) the relevant experimental data for comparison with the simulation data that were previously stored (2 days earlier by an experimental colleague in the United States) within the IB system.
(vi) The user is notified that interim results are ready for visualization, and contacts the colleague in the United States via instant messaging to initiate collaborative visualization and steering session making use of the Grid-enabled video conferencing support middleware, AccessGrid.
(vii) By direct comparison of experimental and simulation results, these transatlantic collaborators are able to determine the point of initiation of arrhythmia, and automatically generate additional in silico experiments that will guide the next round of the experimental programme.
(viii) This pattern of working continues, alternating between interactive monitoring and control of simulation, and other tasks, all supported by VRE tools.

---

4 See www.jisc.ac.uk.
5 See www.ogce.org.
6 See www.mygrid.org.uk.
(ix) The job is complete and all relevant simulated data, provenance data, metadata, visualizations and workflows are securely stored at a remote storage facility.

In this way we aim to ensure that not only the simulation and related data are captured and securely stored for future analysis, but also that the entire process, utilising the advanced knowledge and expertise of the user, is also reclaimed for future direct use and in training by novice users.

3. Central role of high-performance computing

The key requirement in facilitating this iterative approach between experiment, modelling and in silico simulation in systems biology research is that simulation codes run sufficiently quickly. As we have discussed earlier, the computing requirements of systems level simulations are such that this requires the use of multi-processor and distributed memory computer clusters. The IB project has available to it several computer packages for simulating cardiac activity, some of which run only on the desktop but some can be run on parallel shared memory machines or on computer clusters. In order to migrate novice users from their own desktop computers to HPC resources it is necessary to first migrate the code, and then to give the users the tools to facilitate easy access to remote resources. In this paper, we describe our experiences in taking a desktop-bound code and migrating it to a computer cluster, as a typical case study in parallelizing codes of this type. While some of the features of the COR package may be unique to it, others are generic, and many of the decisions that we have made and some of the problems that we have encountered are applicable to the parallelization of any such cardiac modelling or indeed physiological systems-modelling package.

(a) Cellular Open Resource

The desktop system that we used in this study is the Cellular Open Resource (COR) developed by Garny et al. (2003b), which is a modelling environment used for both research and teaching. It was written in Borland Delphi. It runs under Microsoft Windows and uses CellML 1.0 (Hedley et al. 2001), a markup language for describing biological models. COR can be used to solve cardiac electrophysiological problems. Such problems can consist of a cell (zero-dimensional) or a strand (one-dimensional), plane (two-dimensional) or volume (three-dimensional) of cells. Independent of the dimensionality of the problem, each virtual cell requires its transmembrane potential ($V_m$) to be computed through time ($t$):

$$\frac{\partial V_m}{\partial t} = -\frac{1}{C_m} \sum I_{ion}, \quad (3.1)$$

where $C_m$ is the membrane capacitance, and $I_{ion}$ the current associated to a given ion, whose formulation is as follows:

$$I_{ion} = g_{ion} (V_m - E_{ion}), \quad (3.2)$$
where $g_{\text{ion}}$ is the ion conductance and $E_{\text{ion}}$ the ion equilibrium potential. The formulation for $g_{\text{ion}}$ is of the Hodgkin–Huxley type (Hodgkin & Huxley 1952), which relies on a set of ordinary differential equations (ODEs), while $E_{\text{ion}}$ is determined using the Nernst equation. At the multi-cellular level, in addition to the above set of ODEs, either one (monodomain) or two (bidomain) partial differential equations must be solved. The bidomain approach is based on cable theory and takes into account both the intracellular and extracellular space. COR, however, uses a monodomain approach where only the intracellular space is considered. The formulation for this approach is

$$ A_m \left( C_m \frac{\partial V_m}{\partial t} + \sum I_{\text{ion}} \right) = \nabla \cdot (D \nabla V_m), \quad (3.3) $$

where $A_m$ is the surface-to-volume ratio of the cell, $I_m$ the transmembrane current, $\nabla \cdot$ the divergence operator, $D$ the conductivity tensor, and $\nabla$ the gradient operator.

Garny et al. have chosen the simplest numerical approach to solving these equations—explicit finite difference methods. These were chosen since they are intuitive and therefore easily understood by non-specialists, and are very straightforward to code allowing progress to be made rapidly in simulating realistic problems. This choice also means that the parallelization of the code to a distributed memory environment is also comparatively straightforward, allowing the IB project to gain valuable insight into the aims and objectives of the modelling and visualization aspects of the problem, without becoming unduly bogged down in complex parallelization issues that would arise with more complex numerical approaches (e.g. Gavaghan et al. 1995). The reaction part of the problem (equation (3.1)) is solved using an explicit Euler integrator, i.e.

$$ y_{n+1} = y_n + hf(t, y_n) + O(h^2), \quad (3.4) $$

while a finite difference technique is used to solve the diffusion part (right-hand side of equation (3.3)). Equation (3.3) can be expanded as follows:

$$ A_m \left( C_m \frac{\partial V_m}{\partial t} + \sum I_{\text{ion}} \right) = D_{11} \frac{\partial^2 V_m}{\partial x^2} + D_{22} \frac{\partial^2 V_m}{\partial y^2} + D_{33} \frac{\partial^2 V_m}{\partial z^2} $$

$$ + 2D_{12} \frac{\partial^2 V_m}{\partial x \partial y} + 2D_{13} \frac{\partial^2 V_m}{\partial x \partial z} + 2D_{23} \frac{\partial^2 V_m}{\partial y \partial z} $$

$$ + \frac{\partial D_{11}}{\partial x} \frac{\partial V_m}{\partial x} + \frac{\partial D_{12}}{\partial x} \frac{\partial V_m}{\partial y} + \frac{\partial D_{13}}{\partial x} \frac{\partial V_m}{\partial z} $$

$$ + \frac{\partial D_{21}}{\partial y} \frac{\partial V_m}{\partial x} + \frac{\partial D_{22}}{\partial y} \frac{\partial V_m}{\partial y} + \frac{\partial D_{23}}{\partial y} \frac{\partial V_m}{\partial z} $$

$$ + \frac{\partial D_{31}}{\partial z} \frac{\partial V_m}{\partial x} + \frac{\partial D_{32}}{\partial z} \frac{\partial V_m}{\partial y} + \frac{\partial D_{33}}{\partial z} \frac{\partial V_m}{\partial z}. \quad (3.5) $$

Numerical approximations can then be derived for first-order derivatives (equation (3.6)), second-order derivatives (equation (3.7)) and mixed second-order
Figure 1 illustrates the importance of fibre orientation through a simple model of rabbit atrial cells (Hilgemann & Noble 1987). The mesh is that of an annulus with radii of 0.6 and 1 cm. A grid spacing of 0.1 mm is used, while diffusion in the direction of the fibre (approx. 44.5 \( \mu \)S mm\(^{-1} \)) is five times that of the cross-fibre direction (approx. 8.9 \( \mu \)S mm\(^{-1} \)), so that the effect of anisotropic diffusion on electrical propagation can be observed. In the first case (figure 1a), all the cells are oriented vertically. An S1–S2 protocol is used in an attempt to initiate re-entry. It consists of two electrical stimuli that are applied at a different time and location within the mesh. The first stimulus (157.5 nA mm\(^{-2} \)) is applied for 5 ms at the top of the annulus, over a width of 6 mm (frame 1), yielding two wavefronts that progress into the annulus (frames 2–5). The second stimulus, which is similar to the first one, is applied to the right of the mesh (frame 6) after 130 ms, over a height of 6 mm. It has, however, no effect, for the cells located in that region are still in their plateau phase and are, therefore, not excitable. Consequently, the two wavefronts follow their course (frames 7–9) and eventually meet (frame 10), before dying out (frames 11 and 12).

Illustrated in figure 1b is the case where the cells are oriented circumferentially. The same S1–S2 protocol described earlier is used. Following the first stimulus (frame 1), the two wavefronts start developing (frame 2), but propagation appears faster on the inside of the annulus compared to the outside (frame 3). It is in contrast with the previous simulation where the cells are aligned vertically (frame 3 and figure 1a) and, therefore, reflects the difference in fibre orientation between the two simulations. In figure 1a, the wavefront remains globally vertical throughout, while it is much more complex in figure 1b.
This difference in the shape of the wavefront and, therefore, tail proves critical when it comes to the second stimulus. The wave propagates more efficiently (frames 4 and 5) in figure 1b and, by the time the second stimulus is applied, it has reached the other side of the annulus (frame 6). At this point, cells located on the inside of the stimulus region have become re-excitabile. The injection of current into these cells leads, therefore, to the generation of an ectopic beat (frames 7–9) and, subsequently, to a rotor responsible for the successive waves of excitation (frames 10–18).

(b) Computational requirements

The two simulations in §3a each took about an hour to run on a commodity desktop machine and used about 1 GB of memory. Other COR applications with realistic three-dimensional geometry have required significantly more storage and CPU time. Hunter et al. (2003) estimate that a realistic bidomain simulation

Phil. Trans. R. Soc. A (2006)
of a whole heart requires about 30 million grid points. If the current implementation of COR were to simulate a whole heart at this level of detail we would require around a terabyte of storage.

It is acknowledged that the computational technique used in COR (a monodomain explicit finite difference approach) is not suitable for some cardiac problems where a bidomain finite element approach would be more appropriate. The example discussed here nonetheless serves as a place-holder. Research within the IB project (and in other research groups) includes three-dimensional bidomain finite element modelling, and coupling between mechanical and electrical models. Researchers on the cutting edge of cardiac research have a real need for high-performance grid-based computing.

The bottom line is that current simulations of cardiac electrical activity are pushing the limits both in terms of the raw cost of the CPU time, and the amount of memory required. The limit on memory footprint available to a single program with 32-bit addresses is 2 or 4 GB (depending on the underlying operating system). Since there are no immediate plans to have 64-bit addressing in Delphi (or indeed to have a parallel implementation of Delphi) it seems obvious that large calculations should be handled in a different language.

The languages favoured by HPC programmers are C/C++ and Fortran 77/90. Both families of languages have compilers, which can exploit the full capabilities of 64-bit processors. Also both families have bindings to the popular parallel programming libraries OpenMP (for shared memory machines) and the Message-Passing Interface (MPI, for generic distributed or shared memory parallel computing). Our aim in this project was to build a straw-man version of COR in a HPC language. The straw-man code does not have to include all the functionality of the original COR (such as a CellML parser, an equation viewer or a movie maker). However, it should contain the main solver part of the code. It should produce output, which can be read by the original Windows Delphi COR (for post-processing), and it should be portable across architectures and operating systems. Above all, it should be able to run on parallel architectures and divide the data space in such a way that large memory use is less of an issue.

4. Design choices

As discussed in §3, the migration of COR from a desktop program to a parallel code is to be seen as a case-study in parallelizing codes of this type. While some of the features of the COR package may be unique to it, others will be generic. Some of the decisions that we made and some of the problems that we encountered may prove to be applicable to the parallelization of a generic cardiac modelling package.

It was thought best to develop the straw-man version of COR (which we entitled COR//) in C on a Windows machine. This is because the dual tasks of code translation and debugging are complicated when the reference program and the development program exist on separate machines, architectures or operating systems. If the two programs are both on the same platform then there are fewer variables to rule out when unexpected behaviour occurs.

Our first goal was to get COR// running as a sequential code alongside the original COR. Once the new code was validated against the original, then it
could be deployed on other platforms and subsequently parallelized. The main solver for COR was found to be contained in a separate Delphi unit, from which it was relatively easy to work out the prerequisites for starting a computation and explore the structure of the solver. A new method was added to the main solver unit, which allowed all the relevant parameters to be stored in a file, then a C program was spawned, which was able to parse those parameters.

The idea was to present the user of the graphical user interface with a radio toggle box, which allowed them to select either the original Delphi code or another program. When the user clicks the ‘Compute’ button and the solve method is entered, the value of the radio toggle is read, and the solution either proceeds as normal or dumps its parameters to a file before spawning an external program. In a future version of COR it is to be imagined that users are given the option of running the simulation on the local machine, or of submitting it to one of a selection of Grid resources.

The individual cell models in COR are built in an object-oriented way using Delphi units. There is a generic CardiacCell class, which contains various parameters common to all heart cells, such as junctional current and fibre orientations and contains generic methods for initialization and for forward integration in time. Each specific cardiac cell model (such as Garny et al. 2003a or Noble et al. 1998) inherits the generic information from the CardiacCell class but extends the class with its own parameters and its own methods for computing derivatives. The initialization method of each of these inherited cells calls the initializer of the parent class before performing its own initializations.

Since the structure of this inheritance is easy to duplicate in C++ it was thought best to move from C. It must be admitted that there are disadvantages to choosing C++. First, there is likely to be a drop in performance, as is often the case in numerical codes. Second, parallel implementations of common parallel libraries have to be used cautiously with C++. For instance, the OpenMP binding to C++ will not allow modifications to objects in a parallel region since they may have unknown side-effects. Also, the MPI binding to C++ expects the programmer to serialize objects before they are sent as messages. However, despite these disadvantages it should be noted that the choice of C++ allows for a line-by-line audit of code: the code for the implementation of the same object in COR (Delphi) and in COR// (C++) can be viewed alongside each other. If necessary, two debuggers can step through the same object looking for points of divergence between the implementations.

An immediate problem with the translation of the original Delphi code was in the use of floating point numbers. Delphi allows for the use of an Extended type, which is a 10-byte floating point number. The Extended type is used throughout the original COR. This is reasonable because it adds a great deal of confidence in the accuracy of computation with little impact on the speed. This is because the native type of the Intel floating point unit is the 10-byte extended double type. When a double precision computation is made the arguments are first padded by an extra 2 bytes.

Unfortunately, not all compilers are able to make use of the 10-byte type. We want COR// to be as portable as possible, so it may be detrimental to rely on the use of a non-standard type. Some C++ compilers do have an extended double, for example, the Intel based GNU C++ compiler has a ‘long double’, but the MPI library does not have a native way of transporting such a type. In the light
of this it was thought best to parameterize all floating point numbers in COR/\ in such a way that the Windows version uses 10-byte arithmetic but the generic version (and parallel versions) use 8-byte double precision. In this way, it ought to be possible to validate the COR/\ running under Windows before dropping precision on more generic hardware.

The COR/\ solver proved to be more than three times faster than the original COR solver. That is, the sequential version of COR/\ with no processor specific optimizations (just \( -03 \)) outperformed COR on the same processor by a factor of 3.4. In this comparison, the two solvers were run without producing any output files and without updating any graphical user interface.

5. Parallel implementation

The most used parallel programming library is MPI (Gropp et al. 1999). Using MPI, it is possible to distribute the computation over a number of processors, which communicate via messages. Since, in general, the memory spaces of the processors are disjoint, the memory footprint problem is immediately reduced.

The input geometry to COR currently consists of a regular grid in a three-dimensional Cartesian cuboid (figure 1 is actually a one-layer thick three-dimensional calculation). Grid points are either unoccupied or they contain a cardiac cell object of a particular type. The best division of a particular problem will depend on the exact geometry of the occupied grid point and on the exact nature of the cardiac cells used. The ideal division strategy also varies from time to time, since it is more work to integrate cells which are undergoing excitation.

A means of dividing the problem space, which is adequate for our straw-man code and that works in two or three dimensions and for any number of processes is to divide the computational grid up in layers of one Cartesian direction. For example, in a grid \((x, y)\) where \(0 \leq x, y < 200\) the points \((0...99, 0...199)\) could be handled by one process and the points \((100...199, 0...199)\) by another. Since, in finite difference, the integration method of each cell needs to know certain parameters of its neighbours in the grid, it is necessary to duplicate information about the layer \(x=100\) on the first process and \(x=99\) on the second. We introduced a halo cell object, which inherited information from the CardiacCell type and was able to store variables, such as the junction current and the fibre orientation.

The first version of parallel COR/\ had a master process reading in all the parameters and the computation grid from a file. An even division of the computational grid is made in the \(x\)-direction, and cell information is fed to each of the other processes. Each process is responsible for constructing its own cell objects and one or two layers of halo cells. After each time-step, information is exchanged from the boundary cells of each process into the halo cells of its neighbour.

(a) Output

Output from the parallel code proved to be a little tricky. For every few time-steps a new output file is produced, which contains a time-stamp and a list of cell voltages in traversal order. A naive way to produce such an output file is for each process to write its own output file and then, after synchronization, for one
process to collect and concatenate all the results into a single file. As can be seen from figure 2, this output style has a dramatic effect on the parallel efficiency of the overall running of the code. Presumably this is because of a limit on the number of multiple writes to the disc.

Better output strategies include using the MPIIO file interface for multiple writes to a single virtual file, or a master/servant routine, in which the master process produces a single file by polling each of the other processes in turn. Since our experiments were carried out on clusters with low-latency high-bandwidth networks, we used the master/servant option in the final version of COR//.

(b) Load balancing

Figure 2 shows that even when no output is being produced, the parallel efficiency is still variable. This is because there are certain even splits of Cartesian space where some processes get an unreasonably high number of active cells. The problem is simplified because we are using a simplified geometry: one cell type in a two-dimensional annulus, where symmetry may be exploited. In a real world simulation, such as a three-dimensional whole heart model embedded in an axis-aligned cuboid, the number and type of active cells will vary widely between layers in the $x$-direction.

We therefore turned our attention to the division of cells between processes. A splitting strategy was written, in which the master process counts the number of active cells in each $x$-layer. Then, regions of $x$-space are assigned to each process in a manner likely to even up the load balance. However, only complete layers are assigned in this way and no attention is paid to underlying complexity of the cell models.

As can be seen from figure 3, this new splitting strategy leads to better parallel speedup. In fact, the parallel efficiency for the load-balanced code with no output is above 90% with up to 16 processes.
Memory footprint

In the straw-man COR\textvisiblespace// code, memory was allocated statically, so there was no opportunity for investigating the exact memory usage of any of the processes. However, by counting the number of active cells and extrapolating, we are able to gauge the approximate memory usage of various runs. Since an active cardiac cell object takes up approximately 800 bytes (and a halo cell needs considerably less) it is possible to plot the memory usage of the cell objects in each of the runs from figure 3.

Figure 4 shows the maximum and minimum memory use of each individual process for varying number of processes. It can be seen that the load imbalances in the even Cartesian split strategy are mirrored by an imbalance in memory needs. This is only a problem when a few processes have significantly more than the average amount of work to do. This can be seen more clearly in figure 5a.

(c) Memory footprint

Phil. Trans. R. Soc. A (2006)
where it is apparent that processes 13 and 2 are likely to be the last to finish any particular time-step. The other processes will need to wait before exchanging halo information.

6. Conclusion

This research has been a case-study in migrating desktop-bound simulation codes to parallel computer resources. It can be seen that with a little work and with careful testing it is possible to reproduce the functionality of an existing code in a new code which is both parallel and portable.

The new program COR// exists as a straw-man code. That is, it is an exercise in the process involved and in the problems encountered. A production level version of the code would do many things differently. First, in COR//, memory is allocated statically which means that the overall memory footprint of each process cannot be altered. It also means that all processes carry the extra memory needed for book-keeping by the master process during initialization. Second, the Delphi source for each CardiacCell type was translated into COR// on a line-by-line basis and no attempt was made to optimize it (with code transformations or with lookup tables) for a particular architecture. Third, the halo data exchange between neighbouring cells uses the full slice for a particular x-coordinate, and therefore includes exchanging data for empty grid points. A production version of COR// would only exchange active cell data between neighbouring processes, and would therefore need to balance speed of calculation with cost of communication in an intelligent way.

In undertaking the necessary detailed analysis of the code to allow porting to a parallel architecture, IB project researchers have gained invaluable insights into the process of code development and use. These insights are informing the entire iterative requirements exercise within the project, including performance, visualization, data storage, ontologies for meta-data and provenance data, and requirements for a usable VRE interface. Further details of the requirements process have been described elsewhere (Gavaghan et al. 2004).

Figure 5. A comparison of the relative memory use of the two splitting strategies when there are 16 MPI processes (using the geometry shown in figure 1).
Editors’ note

Please see also related communications in this focussed issue by Bassingthwaighte et al. (2006) and Seemann et al. (2006).

This research was supported by the EPSRC Integrative Biology project GR/S72023. The authors would like to acknowledge the use of the UK National Grid Service and of the Oxford Supercomputing Centre in carrying out this work. The distributed debugging tool (DDT) from Allinea Software Ltd was used in the development of the MPI code.

References


