Parallelisation of Heart Practical 6

Joe Pitt-Francis

May 4th, 2005

This revision May 5th, 2005

1 Introduction

This document concerns the parallelisation of heart practical 6. As of this date the code which runs the practical is contained in `Chaste/heart/tests/TestMonodomainDg0Assembler.hpp` and consists of tests of the form `testMonodomainDg0?D` (where ? is replaced by the dimension of the problem). All these tests have the same basic structure:

1. A mesh is constructed from a mesh reader.
2. An `EulerIvpOdeSolver` is constructed.
3. A `MonodomainPde` object is constructed using the number of nodes in the mesh, the Euler solver and some time parameters. This object will contain the ODE parameters, ODE solutions and stimulus functions at each of the node locations.
4. The initial conditions at each location are set by preparing a std::vector and using `SetUniversalInitialConditions`.
5. A stimulus function is set on (at least) one of the nodes.
6. A boundary condition container is constructed to contain Neumann boundary at all boundaries.
7. A `SimpleLinearSolver` (which is a wrapper to a KSP) is constructed.
8. A `MonoDomainDg0Assemble` object is constructed.
9. A vector of voltages (-84.5 for each node) is built.
10. A data writer is initialised.
11. In a loop over time
    - The voltages are used to set the initial condition of the assembler.

- The assembler is used to solve the PDE (by repeatedly assembling and solving a linear system). Note that the `MonodomainDg0Assembler` has over-ridden the `AssembleOnElement` method of its superclass. This method calls the `ComputeNonlinearSourceTermAtNode` method of the `MonodomainPde` class (which is itself an over-riding class). The upshot of this is that `ComputeNonlinearSourceTermAtNode` now computes the solution to the ODE system at each individual node. This returns a new vector of voltages which is used in the next iteration.
- The data writers write some data.

12. The final solution may be tested against some known solution.

2 What can be parallelised?

2.1 Linear algebra

The linear algebra components of the code are already written in parallel since they use PETSc vectors and matrices. The core linear algebra solver (KSP) is aware of the parallel structure of the matrices and vectors and is written in parallel. There are many efficiency savings to be made in terms of sparse matrix storage and packing the matrix block diagonal, but these won't effect the correctness of the solver.

A potential problem is in interrogating PETSc vectors in order to test solutions when the code is run in parallel. Consider this code (which is correct when run sequentially:

```
PetscScalar *solution_elements;
VecGetArray(solution_vector, &solution_elements);
TS_ASSERT_DELTA(solution_elements[0], 1.0, 0.000001);
TS_ASSERT_DELTA(solution_elements[1], 2.0, 0.000001);
TS_ASSERT_DELTA(solution_elements[2], 3.0, 0.000001);
```

The problem with this test is that `solution_vector` will be split between processors. When run on two processors, process 0 owns the first 2 components of the vector and process 1 owns the final component. The pointer `solution_elements` points to the portion of the vector which is locally owned. If the above code is run on two processors, process 0 will fail the final assertion (since it doesn't own the final element), and process 1 will fail all the assertions. The correct code is to only tests the locally owned portions of the vector:

```
PetscScalar *solution_elements;
VecGetArray(solution_vector, &solution_elements);
int lo, hi;
VecGetOwnershipRange(solution_vector,&lo,&hi);
double real_solution[3]={1.0,2.0,3.0};
for (int i=0;i<3;i++){
```

```
  if (lo<=i && i<hi){
    TS_ASSERT_DELTA(solution_elements[i-lo], real_solution[i], 0.000001);
  }
}
```

2.2 Linear system assembly

Since the matrix and right-hand side of the linear algebra system are already distributed, there are 3 immediate solutions to the problem of assembling the linear system in parallel.

The cleanest solution is to produce a **master assembly**. That is, one process read the mesh files and constructs the finite-element mesh. It then assembles the entire linear system while the other processes spin. All the data is pushed out to the remote processes using the normal PETSc synchronisation techniques. The disadvantages of this method are that it require lots of communication, the processes need call MPI in order to know their rank and there is a memory in-balance between the master process and the slaves.

A compromise solution is to do a **fully distributed assembly**. In this method each process loops through a subset of the elements and a subset of the boundary conditions. Care would need to be taken to ensure that the nodes owned by the subset of elements matched the nodes owned by the processor in the global solution. It may be possible for the processes to work on a reduced mesh in order to save memory, but this would be a lot of careful re-programming. This is the most efficient method, but it still suffers from a communication overhead (since processes will need to set values owned by remote processes).

The least efficient way to code the assembly is to do **full local assembly**. Here every process reads the entire mesh and attempts to construct the entire system. We need to guard against the same value in a matrix being set multiple times (since this is inefficient) or being added to multiple times (since this would be incorrect). If the guard makes sure that only locally owned values are inserted or added to, then we can cut the communication down to a negligible amount. Since coding this amounts to a small re-factoring of the `LinearSystem` class, we opt for this approach.

2.3 PDE to ODE coupling

Currently, the `MonodomainPde` class contains the input and output variables for the solution of the system of ODEs at each node position. The ODE solver solves a system of ODEs at each node independently and therefore its work may be trivially parallelised. It is our opinion that the PDE class ought to be distributed in the same manner that PETSc vectors are distributed. When a PDE class is constructed, each processor ought to reserve the amount of space for ODE inputs and outputs that corresponds to the number of nodes which it owns in the global solution. Simple guards can be implemented such that all operations on the PDE object are only actioned locally. For example, the method `SetStimulusFunctionAtNode(0, ...)` should only be actioned on

process 0 and ignored by all other processes (because it is likely that process 0 will own node 0).

The advantage of distributing the PDE object in the same manner as the PETSc vector is that all exchange of data between the PDE solver and the ODE solver stays local to the process.

2.4 Data writers

The data which needs to be output from a parallel simulation is distributed across all the processes. We need to rewrite the data writers in order to ensure that data from all processes is collated into a single file in a consistent manner. This problem is similar to that in Parallel Programming Exercise 7.

3 Fine-grained tasks

3 Make separate directory for parallel tests which are always run on more than one virtual processor. (Done Tues AM).

1 Fix some linear algebra tests so that they pass when run in parallel. (Done Tues AM).

3 Fix linear system class so that assemblers can only add or insert to values that are held locally. (Done Weds AM).

3 Make a simple PDE on a mesh work (Done Tues PM for heat equation on a disk).

10 Make `MonodomainPDE` into a distributed object. The distribution should mirror the distribution of a PETSc vector.

5 Distribute other PDE classes.

15 Make data writers work in parallel.

2 Copy and fix the three Heart Practical 6 tests.

42 (Total estimate in pair hours. 32 remaining)