*Communicating Process Architectures 2000*
*P. H. Welch and A. W. P. Bakkers (Eds.)*
*IOS Press, 2000*



# Steering High-Performance Parallel Programs: A Case Study

Peter J. LOVE
*Department of Theoretical Physics, 1, Keble Road, Oxford 0X1 3NP, UK*

Jeremy M. R. MARTIN
*Oxford Supercomputing Centre, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

**Abstract.** Computational steering is the ability to visualise the data from a computation in progress and to modify the future behaviour of the computation in response to this. It is often perceived as being something very difficult to implement, especially for parallel computations. However, given a good visualisation environment, we have found that this is not necessarily the case. We have sought to dispel this myth using a very simple model which makes it easy to 'wire-up' an existing MPI parallel program for steering. New insights may quickly be gained by continually monitoring and guiding the progress of computational simulations, that were perhaps previously analysed only in their final state.

## 1 Introduction

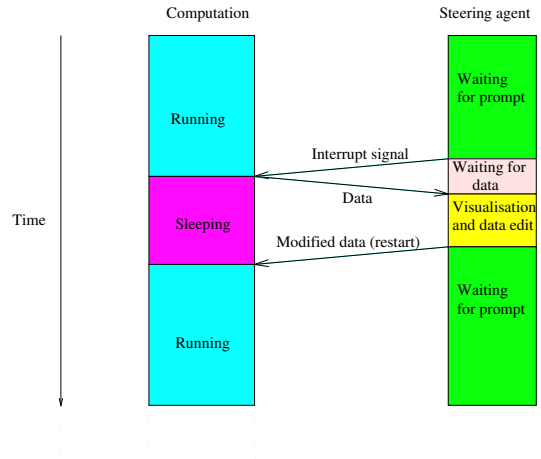
Technological advances are making parallel computers ever more powerful, and the range of problems that they may tackle is continually expanding. However a 'mainframe mentality' persists in the manner of their use which dates from the nineteen-sixties. They are typically operated as batch-processors sealed off from their users in some way – usually system time must be reserved days in advance [1]. The purpose of this method of operation is to try to maximise the flow of work through a heavily-loaded shared resource. However this approach overlooks the fact that many jobs ultimately fail to yield useful results.

Were there a facility to visualise the state of running programs many of these failures might be detected at an early stage, freeing up the resources for another purpose. In some cases failure might be averted simply by modifying certain parameters and feeding them back into the application in progress. Allowing scientists the freedom to exploit their intuition interactively can greatly reduce the computation time required to get results.

Much research effort has been applied to developing computational steering environments for parallel programming (e.g. [2–4]). Typically these involve a visualisation component, a communication component, and some sort of model for parallel computation. There are usually tens of thousands of lines of source code that need to be ported to one's own system. This can be intimidating to a potential user, giving the impression that much work will have to be done. Our approach is bare-bones. We do not provide any parallel programming model, but we support the industry-standard MPI library [5] (and also OpenMP). We do not specifiy any particular visualisation system, but have found that our code fits easily into existing graphics systems, such as AVS [6]. All we provide is the 'glue' for connecting the parallel program to visualisation code using three simple subroutines. The advantage of this minimalistic approach is ease of implementation. The effort required to make an existing parallel program steerable is very small, assuming that the operation of the program is well-understood.

Our model for computational steering consists of two elements: a parallel application and a steering agent. The steering agent provides a user interface for visualisation and command entry, and may run on a separate machine from the application. When prompted by the user, the agent sends a signal to the application causing it to pause, at a suitable point in its execution, and then send back a snapshot of its data. The data is then visualised and explored by the user and in due course a restart signal is sent back to the application, possibly accompanied by some modified parameters (figure 1).

Figure 1: A basic steering model




This paper describes a FORTRAN subroutine library which is designed to facilitate computational steering of MPI jobs, according to this simple model. We illustrate use of the library by means of a significant example: a program for mesoscale modelling of complex fluids. A more detailed technical tutorial on implementation ofsteering is contained in appendix A. Visualisation is performed using AVS [6]. Although all routines are implemented in FORTRAN a port to another high level language such as C would be trivial.

## 2 The MPI parallel programming model

MPI [5] is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. It is now accepted as the industry standard for programming distributed memory parallel computers using conventional high level languages: C, C++, and Fortran.

MPI is a *single-program-multiple-data* paradigm (SPMD), which means that the same code is executed on each processor. However, processors may still be made to perform different tasks by first invoking the inquiry function `MPI_COMM_RANK` to discover the processor identity, and thence to determine the subsequent behaviour at each node. The total number of processors used is fixed throughout the execution of an MPI job, and may be inquired using the `MPI_COMM_SIZE` function.

Communication of a message between two processors requires that one processor issue an `MPI_SEND` instruction and that the other processor issue a corresponding `MPI_RECIEVE` command. Communication may be either buffered or unbuffered and, independently of that, it may also be either blocking or non-blocking. Global communication facilities, such as all-to-all exchange (`MPI_ALLTOALL`) and barrier synchronisation (`MPI_BARRIER`), are also provided.

Due to the flexibility of message-passing available in MPI there is much that can go wrong. Deadlocks, livelocks and buffer overflows may crop up and it can be very difficult to debug MPI codes. Hence it is a good idea to program according to tried and tested design patterns which guarantee avoidance of these demons [7, 8].

## 3 The STR library

The main function of this code is to provide a communication interface between an MPI application program and an interactive 'controller', (which may be a visualisation tool). Synchronisation between the two agents is achieved primarily by repeatedly creating and deleting a lock-file.

### *3.1* SUBROUTINE STRSETUP

```
SUBROUTINE STRSETUP(I)
INTEGER I
```

This routine is called by the application to set up computational steering. The argument `I` specifies a flag number which is to be used for controlling the application henceforth. The assignment of a unique number to each parallel application means that a single steering agent can interact with multiple parallel applications independently.

### *3.2* SUBROUTINE STRBREAKPOINT

```
SUBROUTINE STRBREAKPOINT(DUMP,GET)
EXTERNAL DUMP, GET
```

This routine is called by the application at points in its execution where it would be appropriate to halt and be examined. It has the effect of globally synchronising all the MPI threads, and so should be used discriminately so as not to reduce significantly the overall efficiency of the code.

It works as follows. Firstly the root MPI process checks whether the steering flag has been set by an external agent, i.e. the controller, and then broadcasts the result of this check to all the other processes. If the flag has been set each process calls a user-supplied subroutine `DUMP` that must be supplied as an argument by the calling program. The root process then unsets the flag and goes to sleep, periodically checking to see whether the flag has been reset. Once the flag has been reset a global barrier synchronisation is applied between the MPI processes and then they all call subroutine `GET`, to take on board any information that has been introduced by the external agent. The root process finally unsets the flag once again and execution of the code resumes at the point where it was interrupted (figure 1).

### *3.3* SUBROUTINE STRKICK

```
SUBROUTINE STRKICK(ID)
INTEGER ID
```

This routine is the steering agent interface. It is used both for intercepting and restarting the application. When performing an intercept it blocks until the application has completed its call to the `DUMP` subroutine. When performing a restart it blocks until the application has completed its call to the `GET` subroutine.

This synchronisation is achieved using a lock-file semaphore and the precise mode of operation of this routine is as follows. First it waits for the flag specified by the argument `ID` to be unset (by the application), if necessary, and then resets it. It then waits again for the flag to be unset by the application.

## 4 A case study: Mesoscale modelling of complex fluids

In this section we present an example of the use of computational steering for the mesoscale simulation of binary immiscible fluids. The simulation of such fluids remains a very serious challenge. At the very finest level, molecular dynamics is often used to study small systems, but is too computationally demanding to reach the hydrodynamic timescales of interest. At the very coarsest level, one would like to obtain hydrodynamic equations for such fluids, similar to the Navier-Stokes equations for simple viscous fluids, but such equations are often unknown or ill-posed for complex fluids. For this reason, attention has turned in recent years to *mesoscale* descriptions of complex fluids, such as lattice-gas automata models. Such descriptions aim to simulate the complex fluid on length and time scales that are much larger than those of the molecular level, but still much smaller than those of the bulk hydrodynamics. The algorithm is based on a hydrodynamic lattice gas technique originally obtained by Rothman and Keller [9]. The details of the particular implementation are described in a recent paper [10].

The code is implemented in F90 and parallelised using MPI. It has been used extensively on the T3D formerly at EPCC in Edinburgh, on Origin 2000 machines both at the Oxford Supercomputing Centre and at the Boston University Center for Computational Science, and on T3E machines at SGI and at the CSAR service in Manchester. Prior to the implementation of steering the code produced AVS viewable files at a user specified interval. A suitable `DUMP` routine, as referred to above, was therefore already available. The `GET` routine is simply the subroutine which reads the initial input file. All the essential ingredients required by the steering library were therefore already implemented in the code. In order to implement steering, we simply added a call to `STRSETUP`, using the random seed as the job id and a call to `STRBREAKPOINT` wherever the output was written. The *control* module was added to the AVS visualisation network, and the implementation was complete.

An initially homogeneous mixture of oil and water below a critical temperature referred to as the *spinodal point* will spontaneously separate. Our model is capable of simulating this behaviour and computational steering was used to locate the spinodal point. Previously such points in the phase diagram of the model have been found using large task-farm parameter searches. Such searches are computationally expensive. Additionally, one is limited to a system which will fit on a single processor and so finite size effects may distort the results obtained. In the course of a single simulation lasting under an hour the systems temperature was repeatedly raised above and lowered below the spinodal point and the behaviour observed until the desired accuracy was obtained. The sequence of events observed in a cycle of simulation is shown in figure 2. The use of steering meant that the simulation could be stopped when the desired accuracy for the result had been obtained, and represented an enormous saving in both wallclock and CPU time.

For examples such as the above, search algorithms based on some metric measure are the alternative to the more 'brute force' taskfarm. For one-dimensional parameter spaces such search algorithms are relatively straightforward. However, the extension of such algorithms to multidimensional parameter spaces is highly non-trivial. Steering enables the use of intuition to guide such parameter space searches.

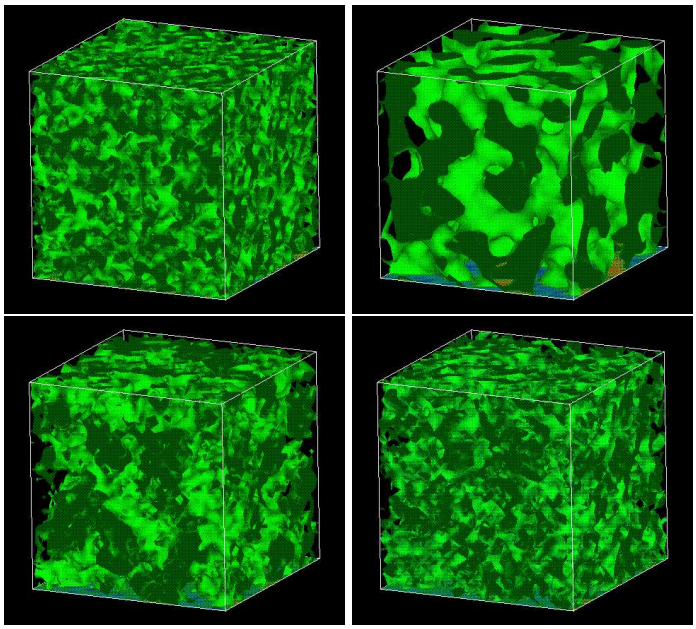


Figure 2: The cycle of observations in a steered simulation to determine the spinodal point of an immiscible fluid. Top left: initial homogenous mixture below spinodal point. Top right: Clear phase separation indicating system is below spinodal point. Bottom right: System remixing after temperature is shifted above spinodal point. Bottom left: System remixed ready for the next temperature change.

## 5 Conclusions and future work

We have shown that computational steering can be very easy to implement as an add-on to existing MPI programs. What little effort is required to get started is quickly repaid by getting faster research results. Despite its immense simplicity, we have found our approach to be surprisingly useful. The case study described would have been hugely expensive in cpu-time had traditional batch task-farm analysis been applied.

We feel that computational steering is mainly an *educational* issue rather than a technical one. The main creative challenge is to think of a way to apply a suitable interactive feedback loop to an existing simulation program so as to get better results. Once this is done the implementation should be easy.

There are many other libraries in existence that offer computational steering environments. The CUMULVS [4] library is similar to ours in that it is intended as an aid to steering existing parallel programs, although these must be written using the PVM parallel library rather than MPI. It is higher level than our library in that it offers additional features such as checkpoint/restart and fault tolerance. It also provides special parallel routines for data extraction, whereas we leave this task to the user (by way of the `DUMP` and `GET` routines). On the downside CUMULVS is much more complicated for a user to learn than our library amd also has additional system overheads: requiring certain daemons to be run. It also ties the user into using the CUMULVS visualisation software, whereas our software can be used with any graphics package. Other approaches such as VIPAR [3] and SPaSM [2], are more intrusive - taking control of the interprocess communications as well as the steering. Whilst they may be attractive to developers of new parallel applications they do not offer the ability to 'retrofit' a steering interface to legacy code.

There is much benefit in designing simple portable protocols upon which higher level facilities may be constructed, and this has been the philosophy of our approach. There is much evidence for the validity of this in the world of networking. Consider the universally

accepted SNMP protocol for network management. This has but two basic commands: one to write a variable and one to read a variable, and yet it is the linchpin of much complex software for maintaining the health of the internet.

In the future we shall be collaborating in setting up a national computational steering service, which will run on a front-end machine to a 512 processor Cray T3E. This will provide us with the opportunity to work with many varied applications and to consider enhancements to our model that might be necessary. We also plan to blend the computational steering facilities described here into the Oxford BSP visual programming development system [11].

The STR computational steering library may be freely downloaded from URL
`http://www.osc.ox.ac.uk/vsig.html`

## References

[1] Kathryn M. Measures, Jeremy M.R. Martin and Robert C.F. McLatchie. *Supercomputing Resource Management – Experience with the SGI Cray Origin 2000.* in B.M. Cook ed. Architectures, Languages and Techniques for Concurrent Systems. IOS Press (1999).

[2] David M. Bezley and Peter S. Lomdahl. *Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations.* Proceedings of Supercomputing '96.

[3] Steve Larkin, Andrew J. Grant, W. T. Hewitt. *Libraries to Support Distribution and Processing of Visualization Datasets.* Future Generation Computer Systems (Special Issue HPCN '96) Vol. 12, no 5, Springer pp431-440.

[4] G. A. Geist II, James Arthur Kohl and Philip M Papadopolous. *CUMULVS: Providing Fault-Tolerance, Visualisation and steering of Parallel Applications.* International Journal of High Performance Computing Applications, Volume 11, Number 3, August 1997, pp. 224-236.

[5] *MPI: A Message Passing Interface.* Proc Supercomputing '93, IEEE Computer Society, Message Passing Interface Forum series, pp878-883. (1993).

[6] Advanced Visual Systems Inc. *AVS User's Guide, Release 4* (1992).

[7] J. M. R. Martin, I. East, and S. Jassim. *Design Rules for Deadlock Freedom*. Transputer Communications Vol 2, No. 3 (1994)

[8] J. M. R. Martin and P. Welch. *A design strategy for Deadlock-Free Concurrent Systems*. Transputer Communications, Vol. 3 Number 4. (1996).

[9] D. Rothman and J. Keller. *Immiscible Cellular-Automaton Fluids.* J. Stat. Phys. **52**, 1119 (1988).

[10] B. Boghosian, P. Coveney, and P. Love. *A three-dimensional lattice gas model for amphiphilic fluid dynamics.* Proc R Soc London A, Vol. 456, pp 1431-1454.

[11] Jeremy Martin and Alex Wilson. *A Visual BSP Programming Environment for Distributed Computing.* Proceedings of Fourth Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, LNCS Springer-Verlag, (2000)

## A An Example of Steering Implementation.

### *A.1 Making an application steerable.*

For the purpose of illustration we shall consider a rather contrived example. We shall start with an MPI parallel program, written in FORTRAN, which calculates temperature distribution over a metal plate with a heating element attached to one side. This is achieved by a numerical solution to Laplace's equation over a rectangular grid using Dirichlet boundary conditions. The data domain is partioned into strips. Data are passed backwards and forwards between two rectangular arrays, U and UNEW.

```
      PROGRAM MPIDIRICHLET
      INCLUDE 'mpif.h'
      INTEGER PWIDTH, PHEIGHT, STRIPSIZE
      PARAMETER (PWIDTH = 400, PHEIGHT = 498, NPROCS = 8)
      PARAMETER (STRIPSIZE = (PHEIGHT - 2)/NPROCS+2)
      REAL U(PWIDTH,STRIPSIZE), UNEW(PWIDTH,STRIPSIZE)
      CALL MPI_INIT (ERRCODE)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, PROC, ERRCODE)
      CALL INITIALISE(U,UNEW,PWIDTH,PHEIGHT,STRIPSIZE,PROC)
      DO N = 1, 100000
        CALL ITERATE(U,UNEW,PWIDTH,STRIPSIZE,PROC,NPROCS)
        CALL ITERATE(UNEW,U,PWIDTH,STRIPSIZE,PROC,NPROCS)
      END DO
      CALL MPI_FINALIZE (ERRCODE)
      STOP
      END
```

Now in order to make this code 'steerable' we need to add a call to STRSETUP early on to set an identifier that may be used to control the program from outside. We also need to place calls to STRBREAKPOINT at suitable locations. Whenever the program arrives at such a point in its execution it will check whether an interrupt signal has been sent, and, if so, it will call the DUMP subroutine and then go to sleep until it is reactivated. Upon reactivation it will call the GET subroutine prior to restart.

This transformation is very simple to apply. The main program simply needs to have a few lines added as follows.

```
      PROGRAM MPIDIRICHLET
      INCLUDE 'mpif.h'
      INTEGER PWIDTH, PHEIGHT, STRIPSIZE
      PARAMETER (PWIDTH = 400, PHEIGHT = 498, NPROCS = 8)
      PARAMETER (STRIPSIZE = (PHEIGHT - 2)/NPROCS+2)
      REAL U(PWIDTH,STRIPSIZE), UNEW(PWIDTH,STRIPSIZE)
C
C Create COMMON block for data and declare communication routines
C
      COMMON/ONE/U, N
      EXTERNAL DUMP, GET
      CALL MPI_INIT (ERRCODE)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, PROC, ERRCODE)
C
C Set Job id for computational steering
C
      CALL STRSETUP(12345)
      CALL INITIALISE(U,UNEW,PWIDTH,PHEIGHT,STRIPSIZE,PROC)
```

```
      DO N = 1, 100000
C
C Set break point
C
        CALL STRBREAKPOINT(DUMP,GET)
        CALL ITERATE(U,UNEW,PWIDTH,STRIPSIZE,PROC,NPROCS)
        CALL ITERATE(UNEW,U,PWIDTH,STRIPSIZE,PROC,NPROCS)
      END DO
      CALL MPI_FINALIZE (ERRCODE)
      STOP
      END
```

It is also necessary for the programmer to supply suitable DUMP and GET subroutines, which must have no arguments. The purpose of these routines is to send data to the external agent and also to retrieve modified parameters before restart. (Note that routines DUMP and GET are called by all processes, and so may communicate with each other using MPI routines.)

In this case we shall make DUMP output the entire data array U to a file **dirichlet.dta**. Subroutine GET will read in a new temperature for the heating element, resulting in resetting of some boundary values for the array. Code for DUMP is as follows. We do not include code for GET as that is very similar.

```
      SUBROUTINE DUMP
      INCLUDE 'mpif.h'
      ... data and parameter declarations omitted
      COMMON/ONE/U, N
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, PROC, ERRCODE)
      IF (PROC .EQ. 0) THEN
        OPEN(UNIT=23,FILE=DUMPFILE,STATUS='NEW')
C
C Now we have to get the information from each process and
C dump it out
C
        WRITE(23,'(f4.2)'),((U(I,J),I=1,PWIDTH),J=1,STRIPSIZE-1)
        DO P2 = 1, NPROCS - 1
          DO K = 2, STRIPSIZE - 1
            CALL MPI_RECV(UBUFF,PWIDTH,MPI_REAL,P2,1,
     :      MPI_COMM_WORLD,STATUS,IERR)
            WRITE(23,'(f4.2)') UBUFF
          END DO
        END DO
        CALL MPI_RECV(UBUFF,PWIDTH,MPI_REAL,NPROCS-1,1,
     :  MPI_COMM_WORLD,STATUS,IERR)
        WRITE(23,'(f4.2)'),UBUFF
        CALL FLUSH(23)
        CLOSE(23)
      ELSE
        DO K = 2, STRIPSIZE - 1
          CALL MPI_SSEND(U(1,K),PWIDTH,MPI_REAL,0,1,
     :    MPI_COMM_WORLD,IERR)
        END DO
        IF (PROC .EQ. NPROCS-1) CALL MPI_SSEND(U(1,STRIPSIZE),PWIDTH,
     :  MPI_REAL,0,1, MPI_COMM_WORLD,IERR)
      END IF
      RETURN
      END
```

Now all the work has been done to make the application steerable: we have marked a suitable breakpoint in the code, and have also provided subroutines for writing out the current state of the simulation and reading in new parameters.

### A.2 Steering the Application

Once a code has been made steerable it is controlled by a sequence of calls to subroutine `STRKICK`, which successively cause it to pause and 'dump', and then to 'get' and restart.

The following code skeleton could be used to control a steerable application interactively.

```
PROGRAM CONTROLLER
CHARACTER R
PRINT'(''Please type job ID:'')'
ACCEPT*, ID
R = ''
DO WHILE (.TRUE.)
  PRINT'(''Press return to interrupt or S to STOP:'')'
  ACCEPT'(A)', R
  IF (R .EQ. 'S' .OR. R .EQ. 's') STOP
  CALL STRKICK(ID)
  ... insert code for visualisation and parameter modification
  PRINT'(''Press return to resume:'')'
  ACCEPT'(A)', R
  CALL STRKICK(ID)
END DO
END
```

This code could run on a dedicated visualisation machine, assuming that the filestore for the application is remotely accessible (e.g. using NFS or AFS). It is often possible to integrate this steering code directly into the visualisation system being used, as will be described in the next section.

### A.3 Visualisation using AVS

The application code, once made steerable, is ready to be coupled to a visualisation system such as AVS [6]. AVS is a high-powered commercial tool, with a visual programming environment. AVS applications are constructed by connecting together 'modules' on a graphical display. Each component has a computational function, together with a collection of typed input and output ports for data communication. An easy programming interface is provided for constructing new modules in C or Fortran.

Figure 3 displays a simple AVS network compiled to visualise the results from the above application program. The *control* module has a instrumentation panel with 'stop' and 'start' buttons to control a steerable application. It has been built using the AVS module API and interacts with the application by calling the `STRKICK` subroutine described above.

The standard *readfield* module reads in the data that the application writes with its `DUMP` subroutine. The *temperature* module provides a slider bar for resetting the temperature of the heating element. When the value of the slider is changed a parameters file is written out by AVS which will subsequently be read in by the `GET` subroutine upon restart of the application.

Figure 3: AVS Network Editor and Image Display