# A Project Based Approach to Teaching Parallel Systems

Alistair P. Rendell

Department of Computer Science, Australian National University
Canberra ACT0200, Australia
alistair.rendell@anu.edu.au

**Abstract.** For several years we have delivered advanced undergraduate courses related to computational science using a traditional approach of lectures, laboratory exercises and assignments. In recent years, however, we have moved away from this towards project based approaches. In this paper we discuss our attempts to structure a course in parallel systems around a group project that required the students design, build and evaluate their own message passing environment.

## 1 Introduction

In 2001 under a joint initiative between the Departments of Mathematics and Computer Science, and with funding from the Australian Partnership in Advanced Computing (APAC) [1], an undergraduate degree in computational science was established at the Australian National University. This degree involves students taking roughly a quarter of their courses in computer science, another quarter in mathematics and the remaining half in their chosen field of specialization [2]. Two courses that are offered by the Dept. of Computer Science as part of this degree are a third year course in "High Performance Scientific Computing" [3] and a fourth year course in "Parallel Systems" [4]. In the last two years, and for both these courses, we have adopted a project centered approach to course delivery. The aim is to give the students a better understanding of the hardware and software that they are using, a sense of achieving something significant, and to provide a natural springboard for in depth discussions and subsequent research projects. In the high performance scientific computing course this has involved getting the students to develop a simple parallel molecular dynamics application code that they then run on a cluster that they have built. Some details of the molecular dynamics case study, and how it relates to modern concepts in software design, are given in reference 5. In this paper we focus instead on the parallel systems course, and our efforts to build this course around a group project in which the students write their own basic message passing environment.

## 2 COMP4300: A Course in Parallel Systems

Within the Australian system a science, IT or computational science degree can be completed in three years full-time study. Students who achieve a mid-credit or above

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 155 – 160, 2006.

average (≥65%) are then permitted to study for one extra year and gain an honours degree. In addition to this path some other degrees, such as software engineering, are four year honours programs from the outset. The parallel systems course targets undergraduate students who are in their fourth year, so by in large these are science, IT or computational science honours students or software engineers in their final year of study. Technically it is possible for a standard 3 year degree student to enroll in the course, but this requires them to have undertaken a non-standard enrolment pattern, or be a part-time student. Suffice it to say that the general standard of the students embarking on this course is quite high, so the course is designed to be challenging.

The course is offered in alternate years and aims to cover several aspects of parallel systems, including hardware architecture, programming paradigms, parallel algorithms and sample applications of parallel computers. It is based loosely on the text book "*Parallel Programming: techniques and applications using networked workstations and parallel computers"*, by Barry Wilkinson and Michael Allen [6], augmented with additional material covering, for example, one-sided communications in MPI-2 [7] and Global Arrays [8]. It also usually includes one or two guest lectures and a tour of the APAC National Facility [1]. The course has roughly 30 hours of lectures distributed over a 13 week semester and includes 12 hours of supervised laboratory work. Course prerequisites include a second year course in "Concurrent and Distributed Systems" and a third year course in either "High Performance Scientific Computing" (the course mentioned above) or "Algorithms".

In 2004 after having delivered the parallel systems course 2-3 times in a fairly standard lecture format, the opportunity arose to trial a radical new format. Part of the motivation for this was the fact that in this year the course would only be open to a small group of 5 honours students; making it much easier to map out a new course structure on the fly. And so it was decided to drive much of the course around a group project where the students developed their own message passing environment based solely on the use of simple UNIX utilities; essentially the students were required to write their own limited version of the Message Passing Interface (MPI) [9].

The project, codename `mympi`, began after 2 weeks of introductory lectures and one laboratory session. It was broken down into five 2-week segments that were each assigned to one of the students. Logistically the main lecture time was a block of 2 hours each Friday afternoon. In the first week of each 2-week segment the relevant student was required to come and discuss their part of the project in private a few days before the Friday lecture. At this meeting they were asked to outline what they thought was required, and how they proposed to tackle it. After clarifying any misconceptions, and ensuring that the student was on the right track, they made a formal 10 minute presentation of their proposed work to the rest of the class during the Friday lecture. This would invariably evolve into a class discussion and further refinement of their ideas. In the second week the student would discuss their progress in private before giving a formal presentation and demonstration of their code to the class during the Friday lecture. The student was required to handover their work to the next student before the following Monday. Both presentations were peer marked, with more detailed marking and feedback given after the student had submitted a formal write-up.

The five project stages, their requirements and some comments on the objectives and outcomes are given below:

### 2.1 Stage 1 – Basic Process Creation and Communication

***Requirements:*** Develop an elementary message passing capability using UNIX processes created using the `fork` and `exec` system calls with inter-process communication performed using TCP/IP sockets. Demonstrate operation on a multiprocessor shared memory system with simple byte stream data transfers. The environment developed should be similar to other MPI implementation with program initiation taking place via the following command:

```
mympirun -n n_proc a.out
```

where `n_proc` is the number of copies of the executable (`a.out`) that will be created. The program that gave rise to executable `a.out` should include calls to functions `mympi_init` and `mympi_finalize` that are part of the `mympi` library and are responsible for initializing and terminating the message passing environment.

***Comments:*** As mentioned above a second year course in concurrent and distributed systems is a prerequisite for enrolment. This provided the students with a basic understanding of process creation using `fork` and `exec`, and some exposure to buffered asynchronous message passing using pipes (not sockets). Fundamental design decisions relating to performance, extensibility and understandability were discussed, with particular attention given to the topology of the connections between the processes. In lectures the latter was related back to the more general issue of network topology on parallel computers. For simplicity a completely connected network was chosen, although this was highlighted as a potential bottleneck for very large process count. How to communicate the various socket port numbers between the different processes was discussed, as was ensuring that the network of processes was established in a deadlock free manner. These issues were solved by augmenting the command line arguments that were passed to the user program in the `exec` call and by imposing a specific ordering when the connections were established. From this work the roles of `MPI_Initialize` and `MPI_Finalize` in a real MPI implementation were immediately apparent to the students. The final demonstration involved sending a simple byte stream around a ring of processes – a so called "communication snake" or "com-snake" program.

### 2.2 Stage 2 – Rank, Size, Typed and Tagged Communication with Multihosts

***Requirements:*** Write the equivalent of `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send`, and `MPI_Recv`, but without the use of communicators and requiring the send and receive calls only to support `int`, `double`, and `byte` data types. Specifically, the send and receive calls should be tagged, and there should be wild cards that permit receive calls to match messages from any sending process or any incoming tag. Extend the original implementation to run on multiple platforms of the same type.

***Comments:*** Inclusion of message types and tags requires some additional information beyond the message content to be transferred between processes. The concept of a message header naturally follows from this. The ability to receive a message from any process prompts discussion of non-determinism, busy waiting, and use of the

`select` system call. How to match a message with a specific tag requires the receiver to interrogate an incoming message, read its header, and then potentially receive the message. This clearly shows the need for buffers to store header information (and maybe more) for messages that have been interrogated but found not to match the required message tag. The transition from asynchronous to synchronous message passing that occurs in most MPI implementations as the message size increases (and can no longer be held in the intermediate buffer) is now very obvious. Expanding `mympi` to involve multiple hosts requires thought as to how the hosts will be specified (e.g. via command line list, environment variable, or other means) and how to create processes on remote machines. In particular the option of having daemons on each host responsible for creation of application (`a.out`) processes, versus direct remote initiation was discussed. Different policies for mapping `a.out` processes to available hosts were considered, introducing concepts like round-robin and blocked allocation. Some security issues associated with the creation of processes on remote machines was also discussed. The final demonstration was a modification of the com-snake demo, but with typed data and across multiple hosts.

### 2.3 Stage 3 – Heterogeneous Hosts and Global Operations

***Requirements:*** Use XDR (external data representation) to extend the above code to run between heterogeneous UNIX computers (specifically between UltraSPARC/ Solaris and x86/Linux systems). Using the `mympi` rank, size, send and recv routines developed above construct higher level functions for performing collective operations equivalent of `MPI_Barrier`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Gather` and `MPI_Scatter` (again without communicators and only for reduction calls involving summation). Provide theoretical and observed performance characteristics for all of these functions.

***Comments:*** Moving to a heterogeneous environment requires different binaries to be run on each system type, and considering how these locations should be specified. The difference between big and little endian, learnt during earlier courses, was now very obvious. While use of XDR was mandated, its impact on performance was discussed, as was the use of lighter weight alternatives. Various alternative approaches to constructing barriers and other collective operations were discussed and the cost analyzed as a function of number of processes and length of message. (For the student the primary objective was to implement correct collective operations and understand their performance characteristics. This invariably resulted in binary tree type implementations, although more elaborate schemes such as those outlined by Rabenseifner [10] were discussed in class.) The demonstration involved running the com-snake program over a heterogeneous environment, and then using other programs to perform a variety of collective operations.

### 2.4 Stage 4 – Shared Memory

***Requirements:*** Modify the code to use UNIX shared-memory segments and semaphores for message transfers that occur between processes on the same shared-memory node. Compare performance using shared-memory transfers with socket-based transfers.

***Comments:*** The students had at this stage completed a laboratory class that covered basic use of shared-memory segments. They were also familiar with the concept of a semaphore through the second year course in concurrent and distributed systems, although this had not specifically covered UNIX semaphore arrays. To enable easy switching between use of shared-memory and socket based intra-node communications, a command line option was added to `mympirun`. The concept of a group of processes, defined as all processes running on the same host, comes naturally when using clusters of shared-memory processors. The number of shared-memory segments and semaphore arrays to be used was discussed in the context of contention (for shared resources) and possible system wide limits. In the end a model that used one shared-memory segment divided up to provide unique read and write buffers for each pair of processes on the same host was used. How to handle wild card options that may involve data being received either in the shared memory segment or on a socket was solved, rather inefficiently, using a busy wait loop that monitored all possible sockets and semaphores in a round robin fashion. Superior performance was demonstrated by running a simple ping-pong benchmark with and without shared-memory transfers.

### 2.4 Stage 5 – Performance Evaluation and General Critique

***Requirements:*** Perform extensive performance evaluation for all functions in `mympi`. Consider possible TCP/IP tuning options [11]. Download and install at least one other version of MPI (eg LAM-MPI or MPI-CH [12]) and compare its performance with that of `mympi`. Give consideration to issues raised by your fellow developers during the earlier stages of this project and comment on where `mympi` might go from here.

***Comments:*** Since the performance evaluation was undertaken on a rather noisy student computing environment no performance data will be given here, just details of what was evaluated. The Nagle algorithm, used to concatenate small messages on a TCP/IP network, was identified as a possible performance bottleneck raising latencies for small message transfers. Some tests were run to compare transfers with and without the Nagle algorithm invoked. The effect of changing the size of the shared-memory buffers, identified in stage 4 as a performance issue, was investigated. MPI-CH was installed and its performance compared to `mympi` for simple ping-pong transfers and for collective operations. Results both within a shared-memory node and between nodes were obtained. While MPI-CH was found to be slightly faster, the students were pleased to find that it was not hugely faster (noting that this conclusion was drawn from experiments run on a non-dedicated environment). Error handling and error detection (e.g. when a remote process dies) were identified as the two areas that most urgently required further work.

## 3 Discussion

The group project as outlined above constituted 25% of the students final course mark. Another 25% was associated with a more conventional assignment requiring parallelization of a 1-D fast Fourier transform using MPI on a cluster and pthreads or OpenMP (the student could chose) on a shared memory system. The final exam was worth 50%. With just 5 students in the course feedback was informal, and along the

lines of the course being hard and requiring considerable work, but that they all greatly enjoyed the challenge and the effort made to run a very different style of course. Of the five students who completed the course, 3 went on to obtain first class honours, while 2 ended up leaving university to take up full-time employment before they had completed their honours year. Of the students who obtained first class honours one is now pursuing a PhD in work related to cluster computing.

As with any group project that comprises an incremental set of steps the danger is that someone in the chain fails to deliver. Fortunately this did not occur, and what the students actually achieved was quite impressive. Likewise for any software development project it is also important that the first student makes wise design decisions, as this will intricately affect all later students.

Having trialed this project once with a small group of able students it would be relatively easy to adapt it to a much larger diverse class by, e.g. dividing the students into groups of mixed ability and having each group work on their own version of `mympi`. The exercise could then be run as a challenge between groups to produce the best performing message passing software.

Perhaps the biggest limitation in the project as carried out was the lack of a dedicated teaching cluster so that the students could obtain reliable performance data. In future, however, this will not be a problem, since due to the success of this course and related project work in the high performance scientific computing course we were awarded in mid 2005 a teaching grant that has enabled us to purchase an 8 node dual core Athlon 64 cluster.

**Acknowledgements.** The author gratefully acknowledge support from the Computational Science Education Program of the Australian Partnership in Advanced Computing.

## References

1. The Australian Partnership in Advanced Computing, see http://www.apac.edu.au
2. ANU Bachelor of Computational Science degree, see http://bcomptlsci.anu.edu.au/
3. COMP3320: *High Performance Scientific Computing*, see http://cs.anu.edu.au/student/comp3320
4. COMP4300: *Parallel Systems*, see http://cs.anu.edu.au/student/comp4300
5. J. Roper and A.P. Rendell, *Introducing Design Patterns, Graphical User Interfaces and Threads within the Context of a High Performance Computing Application*, LNCS **3515**, 18 (2005).
6. *Parallel Programming: techniques and applications using networked workstations and parallel computers*, Barry Wilkinson and Michael Allen. Prentice Hall 1999. ISBN 0-13-671710-1.
7. *Using MPI-2: Advanced Features of the Message-Passing Interface*, William Gropp, Ewing Lusk and Anthony Skjellum, MIT Press, 1999; ISBN 0-262-57132-3.
8. Global Arrays, www.emsl.pnl.gov/docs/global
9. Message Passing Forum, see http://www.mpi-forum.org
10. R. Rabenseifner, *Optimization of Collective Reduction Operations*, LNCS **3036**, 1 (2004)
11. See for example: *Enabling High Performance Data Transfers* http://www.psc.edu/networking/projects/tcptune/
12. LAM-MPI, see http://www.lam-mpi.org, MPI-CH, see http://www-unix.mcs.anl.gov/mpi/mpich