Chapter 2

Experimental Approach

To evaluate the efficacy of the design techniques, a suite of parallel problems was chosen and coded in MPI. The suite of problems is described in section 2.1. Other MPI benchmarks are now available, organised by the PARKBENCH committee [2].

In his paper on performance prediction of data parallel programs, Toledo [5] stated that:

> *"We believe that the most important open question in performance prediction today is how to assess and verify the accuracy of performance models. Without the means to assure the accuracy of models it is difficult to put them in production use."*

Section 2.3 describes the profiling technique used to extract actual timings from runs on parallel machines. Section 2.4 describes how detailed comparisons of predicted and actual trace files are performed. Section 2.5 describes the techniques used for comparing actual and predicted scalability.

Assessing how easy the design techniques are to use is more subjective than assessing accuracy. A subjective comparison based on experiences using the design techniques is therefore given in the final chapter.

2.1 The test suite: the Cowichan problems

The Cowichan problems [1] were set by Wilson [6] to assess the usability of parallel programming systems. The suite consists of fourteen problems intended to be implemented in parallel. Some of the problems such as the vector difference routine `vecdiff` are simple to code for a parallel machine. Others such as invasion percolation `invperc` present more difficulties.

[1] "Cowichan" is a place name on the NW coast of N. America

The problems were set to provide an objective basis for comparing how easy different parallel programming systems are to use; the intention was that different groups would code the problems using their preferred tools (shared memory, threads, HPF, MPI etc.) and record the time required to write the parallel versions and the problems faced. A comparison of the problems encountered in porting the problems would enable the usability of MPI, threads, shared memory, HPF etc. to be compared.

A sequential version of all the problems was coded by Wilson in ANSI C, and this was used as the reference implementation for checking that the parallel versions produced the correct results.

The MPI version of the problems was written by Howell and Marr using C++.

The Cowichan problems are specified fully in [6], but brief descriptions of the problems are included below, along with notes on the MPI implementation. Appendix A includes a brief summary of MPI.

2.1.1 `mandel`: Mandelbrot Set

This module computes the mandelbrot set as a matrix of integers (figure 2.1). Each point of the set may be computed independently so it is a simple routine to parallelise. In the MPI implementation, each process is allocated an equal slice of the set to work on.

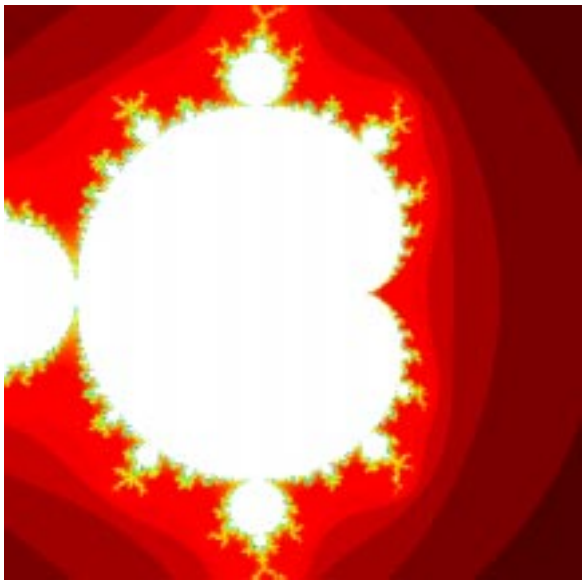

Figure 2.1: The output from `mandel`.

2.1.2 `randmat`: Random matrix generation

This module generates a matrix of pseudo random integers, with a given random number seed (figure 2.2). The aspect of this problem which complicates the parallel implementation is that each successive point in the matrix is computed from the previous one;

$$r_{i+1} = (r_i * a + b) \text{modulo} 2^{32}$$

i.e. there is a sequential dependency. The output must be repeatable and independent of the number of processes; starting each process with the same seed leads to a distinctly non-random striped effect.

The parallel version solves this problem by computing the initial seeds for all processes using an order log(N) algorithm. After this, all may continue independently to compute their area of the matrix.

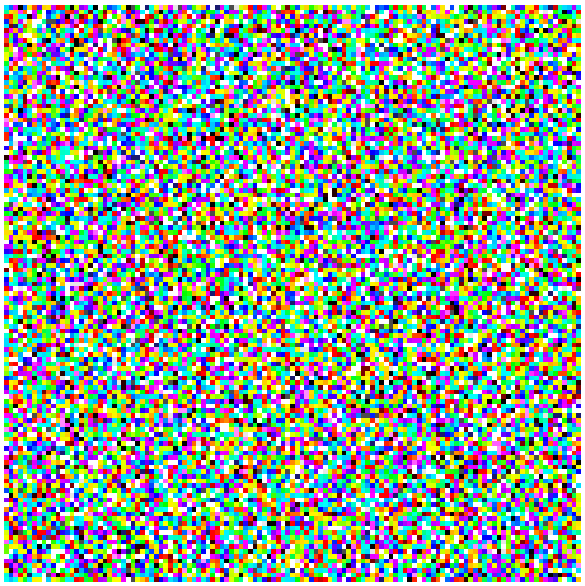

Figure 2.2: The output from `randmat`.

2.1.3 `half`: Two-dimensional shuffle

This module shuffles the values of a matrix along both the rows and the columns. Figure 2.3 shows the algorithm applied to the Mandelbrot set. It exercises the communications facilities of MPI.

2.1.4 `life`: The game of life

This module simulates the evolution of Conway's game of life, a 2D cellular automaton. Figure 2.4 shows the algorithm applied to boolean matrix generated by

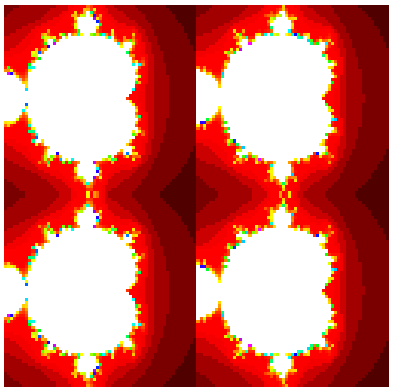

Figure 2.3: The output from `half`.

thresholding a random matrix. The routine uses nearest neighbour communications to calculate the number of neighbours each point has. Global synchronisation is also necessary to keep all processes in step.

Figure 2.4: The output from `life`.

2.1.5 `thresh`: Histogram thresholding

This module performs histogram thresholding on an image. It constructs a binary image from all the pixels of an integer image with an intensity in the top $p\%$ of all pixels. This requires a global reduction to find the highest valued pixel, followed by a local histogram computation. The histograms are then merged (by summing across all processors), the threshold is computed and applied to the image in parallel. Figure 2.5 shows the results of thresh applied to a random matrix (the output of 2.1.2).

2.1.6 `outer`: Outer product

This module takes a vector of point coordinates and forms a dense symmetric diagonally dominant matrix of the distance of each point from every other point,

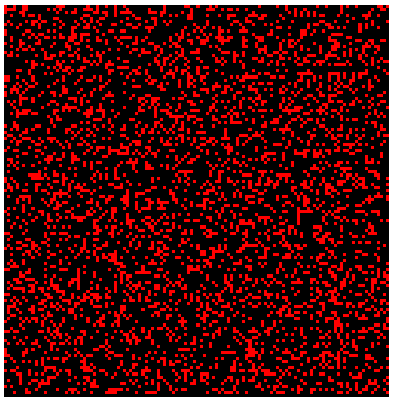

Figure 2.5: The output from histogram thresholding a random matrix.

and a vector of the distance from each point to the origin. Figure 2.6 shows a sample output matrix.

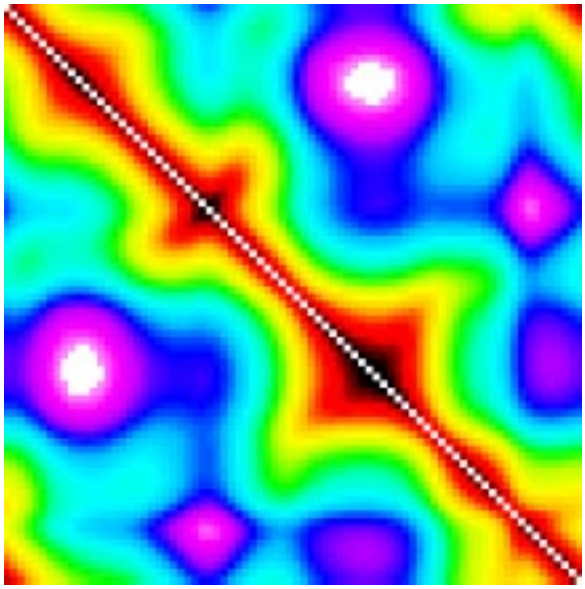

Figure 2.6: The output from the outer product module.

The vector generation is simple; each process has a copy of all points and works on its local section of the distributed vector. The matrix generation is nastier. The simplest solution has each process computing the sections of both the upper and lower triangles with a global reduction (implemented with `MPI_Allreduce`) to get the diagonal. This does twice as much calculation as necessary (as the upper triangle is a mirror copy of the lower), so an alternative would be to load balance one of the triangles then copy that triangle onto the other. Both were implemented for comparison.

The decision of whether to load balance and copy the triangles or to do twice the computation requires a performance specification of the MPI routines. The

triangle copy is difficult to implement in MPI, requiring complex data types and a new operation: `MPI_Alltoallvi`.

2.1.7 `elastic`: Elastic net simulation

This module solves the travelling salesman problem using the elastic net algorithm. The algorithm deforms an elastic circular loop towards the city locations, using an algorithm for the "force" exerted on the elastic by each city and for the elastic force keeping the loop together. Multiple iterations are run, deforming the loop until all cities are connected. Figure 2.7 shows the first iterations of the algorithm, with the circular ring being stretched towards the cities.

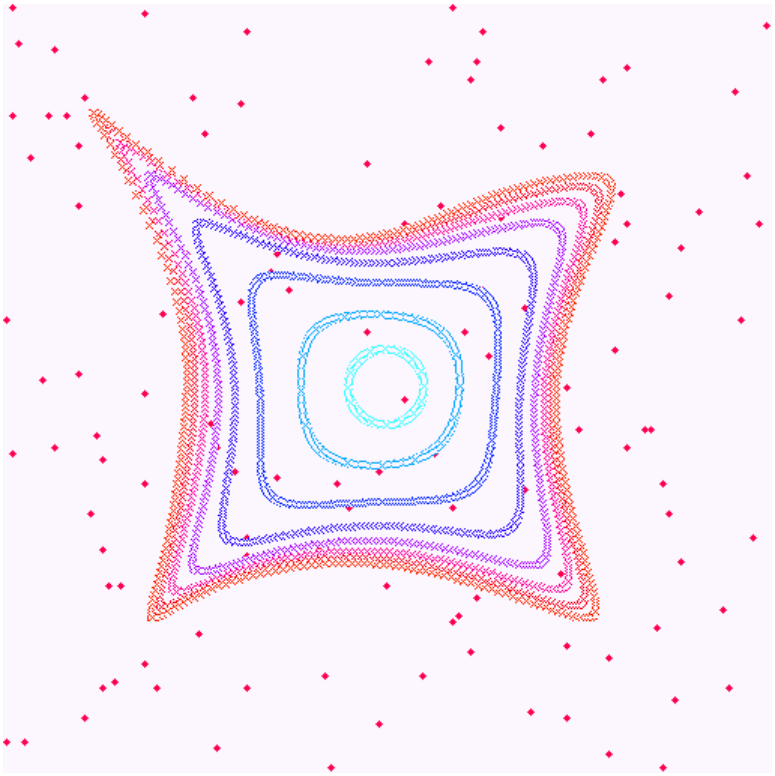

Figure 2.7: In the first steps of the elastic net simulation a circular ring is stretched towards the cities.

The vector of city locations is fixed so it is copied to all processes. The vector of points on the loop is distributed evenly, with neighbour communications. In MPI this boundary communications step is not trivial as there is a special case when fewer than three points of the elastic loop are stored on a process. The boundary exchange was implemented as the sequence `send, send, recv, recv` which may deadlock given limited buffer space but is simple. This is reasonable as the buffer space requirement is only for four real numbers. The alternative

would be for the odd numbered processes to send and the even numbered ones to receive, then vice-versa.

2.1.8 `invperc`: Invasion percolation

Invasion percolation simulates the displacement of one fluid by another in fractured rock. The input is a matrix of integers representing rock densities. In each iteration, all neighbours of all filled cells are examined, and the one with least resistance (i.e. lowest density) is filled. The output is a fractal shape such as that shown in figure 2.8, where the white dots show the filled locations.

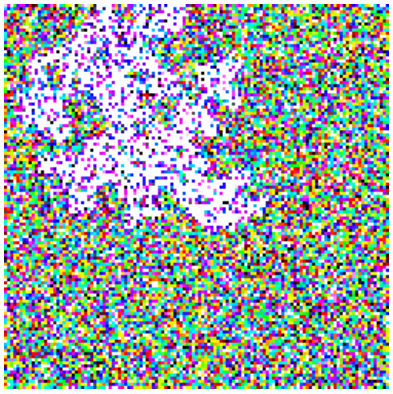

Figure 2.8: The output from invasion percolation run on the random matrix produced by `randmat`.

This is an awkward problem to parallelise. The initial implementation is *not expected to give a speedup*. It uses a distributed queue, where each process stores elements of the queue which lie on the local slice of the matrix. The `enqueue` and `dequeue` operations are called by all processes but operate on only one at a time. The algorithm is inherently sequential and the only possible speedup would be from the shorter queue insert times.

2.1.9 `product`: Vector/matrix product

This module performs the product of a real matrix with a real vector, returning a real vector. The parallel version distributes the rows of the matrix and copies the entire vector to all processes. Each process then computes its local section of the results vector.

2.1.10 `sor`: Successive over-relaxation

This module solves a system of linear equations using the successive over-relaxation iterative technique. The parallel version evenly distributes the rows of the mat-

rix and result vector. At each iteration step each element of the result vector is "relaxed" towards the correct solution by applying a factor computed from the error in the current value. Each process then obtains a local copy of the entire updated result vector for the next iteration. This continues until the solution is within a defined tolerance, or the maximum number of iterations is reached.

2.1.11 `gauss`: Gaussian elimination

This module performs Gaussian elimination to solve a set of linear equations. The output is a real vector containing the solution. In the parallel version each process computes its local pivot. A global communication step then selects which pivot element to use.

2.1.12 `norm`: Point normalisation

The problem is to normalise a vector of point coordinates to lie in the unit square. Figure 2.9 shows example output.

Figure 2.9: The output from `norm`.

The point vector is distributed evenly. A global reduce is required to determine extremities, followed by a scaling of the local data.

The global reduction needs to find the maximum and minimum point locations for the scaling. There is a choice of four `MPI_Allreduce`s (minx, maxx, miny, maxy), or a single `MPI_Allreduce` with a user defined reduction operation. The former was selected for simplicity although the latter is likely to be faster.

2.1.13 `winnow`: Weighted point selection

This module converts a matrix of integer values into a vector of points. A boolean mask matrix is used to select values from the integer matrix. These values are sorted, and the row/column coordinate of every Nth is added to a point vector which is returned. The sequential implementation is straightforward, using loop indices to select particular items of data. With distributed data structures in MPI however it is extremely awkward, requiring extensive use of user constructed datatypes to perform the strided redistribution of data. Each process sends and receives different amounts and types of data to/from all the others.

2.1.14 `vecdiff`: Vector difference

This module returns the maximum difference between corresponding elements of two real vectors. Local differences are computed first, and a global operation returns the overall maximum.

2.2 Techniques used in the MPI implementation

The following notes describe the fundamental parallel data structures and I/O techniques used in the MPI implementation of the Cowichan routines.

2.2.1 Distributed data structures

Matrices and vectors are often distributed evenly across the processes to balance the workload. The standard method of distributing and gathering data in MPI uses the collective communications functions. The example below shows how a matrix distributed across processes may be gathered so that each process has a copy of the entire matrix.

```
int local_matrix[localnrows][ncols];
int global_matrix[nrows][ncols];

int  *counts = /* number of elements on each process */;
int  *displs = /* global offset of 1st element on each process */;

MPI_Allgatherv(
  local_matrix, localnrows*ncols, MPI_INT,
  global_matrix, counts, displs, MPI_INT,
  MPI_COMM_WORLD );
```

To allow such redistribution of data, each process must maintain the arrays `counts` and `displs` to store the number of elements on each process and the displacement from the start of an array. The arrays are combined into a class to simplify use of collective operations:–

```
class distribution {
  int *counts;
  int *displs;
public:
  // Methods to generate useful distributions.
}
```

The inheritance facility of C++ allows a "distributed vector" or "distributed matrix" to be defined by deriving from both `class distribution` and `class vector<Type>`:–

```
class vector_d<Type> : public vector_t<Type>, distribution {
public:
  // Extra methods peculiar to distributed vectors
}
```

Extra methods can then be added to allow global access to distributed data structures, hiding the local offset calculations. These distributed structures may be passed as function arguments and the `counts` and `displs` arrays are available for calling the MPI collective routines.

2.2.2 File I/O

Parallel file I/O is not a standard part of MPI, although all parallel programs will require some I/O. The solution adopted for the Cowichan problems was to perform standard I/O from the root process alone, and to scatter or gather the data to all processes as appropriate. This leads to a heavy I/O cost for each module, as there is a sequential phase before and after the computation which is not shortened as more processes are added. This I/O phase was quantified for the performance evaluations, but the main focus of comparisons was the parallel sections of code, since it is anticipated that truly parallel I/O routines will become the norm eventually.

2.2.3 Graphics

An X windows graphics display was written using the facilities of the MPI implementation on workstations (LAM). The facilities are primitive, offering basic

pixel and area routines but suffice for the production of some interesting pictures such as those illustrating this chapter.

2.2.4 Problems with using MPI

The most painful aspect of using MPI is the datatype definition. The facilities provided for constructing user defined datatypes are powerful but awkward to use. To define a simple `struct` of a `float` and an `int` takes about 10 lines of code and provides plenty of scope for mistakes.

The least pleasant aspect of message passing is the extra code required for computing offsets into distributed data structures. More code is also needed to handle special cases such as how a neighbour exchange should work with less than three elements on a process.

2.3 Measuring performance

To compare predicted with actual timings, a reliable method for obtaining the actual timings was needed. The approach involved a mixture of automatic profiling (for the times of the MPI functions) and user profiling (for the times of the different application phases).

The automatic profiling was accomplished using the MPI profiling library. The user profiling was done using some simple macros:

```
time_set(MPI_Wtime());
// Input
time_mark(‘‘input’’);
// Broadcast arguments
time_mark(‘‘arg broadcast’’);
// Compute results
time_mark(‘‘compute’’);
// Write output
time_mark(‘‘output’’);
time_total(MPI_Wtime());
time_trace(‘‘test’’);
```

These produce a trace file for each run. A separate tool was written to perform successive runs with a range of data sizes and number of processes and to collect the results.

Timing results from a single run may be displayed using the timing diagram tool from the HASE simulation environment [3] described in chapter 6. This shows a zoomable timing diagram with bars for each of the process states.

2.4 Comparing single run predictions with measurements

The obvious way to check the accuracy of the prediction is to time an actual run and compare it with the predicted total time, e.g. (the numbers in the following tables are for illustration only).

Predicted time	Measured time	Ratio
1.23	2.46	2.0

It is possible that this method could indicate that a prediction is perfect whereas in fact a gross overestimate for one part of the time may be serendipitously compensated for by an underestimate for another part. To gain a deeper perspective into the accuracy thus requires looking at more detail than the total run time.

At the next level of detail, the measured/estimated times for *phases* of the application may be compared.

Phase	Predicted time	Measured time	Ratio
Load	1	3	3.0
Bcast	0.3	0.3	1.0
Compute1	3	1	0.3
Compute2	2	2.4	1.2
Gather	7	3.5	0.5
Wr Results	0.32	0.35	1.1
Total	14.6	10.2	0.7

This gives more detail on where the technique is over or underestimating the time. However, even this amount of detail is not sufficient to evaluate whether the technique could be gainfully applied to developing a new application, and it is necessary to look at a finer level of detail to check that the models of the building blocks of an application are applicable.

At this level the possibility of measurements interfering with the system emerges and the quantity of data starts to explode as the comparison is between *tracefiles* of predicted and actual execution. Each iteration of each loop is included in the trace. There is too much data to display in a table and it is difficult to

provide any meaningful comparison. Displaying both traces as timing diagrams gives a visual comparison but extracting numbers is more difficult.

To illustrate the problem, part of a sample trace file from Upshot's ALOG format [1] is given below. On each line, the first number is the event type (e.g. 1 = start broadcast); this is followed by the process number and three zeros (left for expansion). The last number on a line is the time stamp, and this is followed by the name of the event.

```
1 1 0 0 0 680407 start broadcast
2 1 0 0 0 682514 end broadcast
7 1 0 0 0 682693 Start Sync
1 3 0 0 0 682774 start broadcast
2 3 0 0 0 683227 end broadcast
7 3 0 0 0 683322 Start Sync
2 0 0 0 0 687328 end broadcast
7 0 0 0 0 687417 Start Sync
8 0 0 0 0 690727 End Sync
3 0 0 0 0 690796 start compute
8 2 0 0 0 692187 End Sync
3 2 0 0 0 692361 start compute
8 1 0 0 0 695886 End Sync
```

The problem would be somewhat simplified if the predicted and measured tracefiles differed only in the values of their timestamps, but the asynchronous nature of parallel systems means that the ordering of tracefiles often varies. All the Cowichan routines were written to use deterministic patterns of communications which enables traces to be compared using the utility described below.

2.4.1 A trace comparison utility

To address the problem of comparing predicted and measured executions, a *trace comparison utility* was written in C++. Figure 2.10 illustrates the technique.

The utility is used from the Unix command line:

```
cmptrace <infileA> <infileB> <outfile>
```

The input format is trace files in SIM++ format [4]:-

```
$types
State COMPUTE SEND RECV REDUCE BARRIER BCAST GATHER \
```

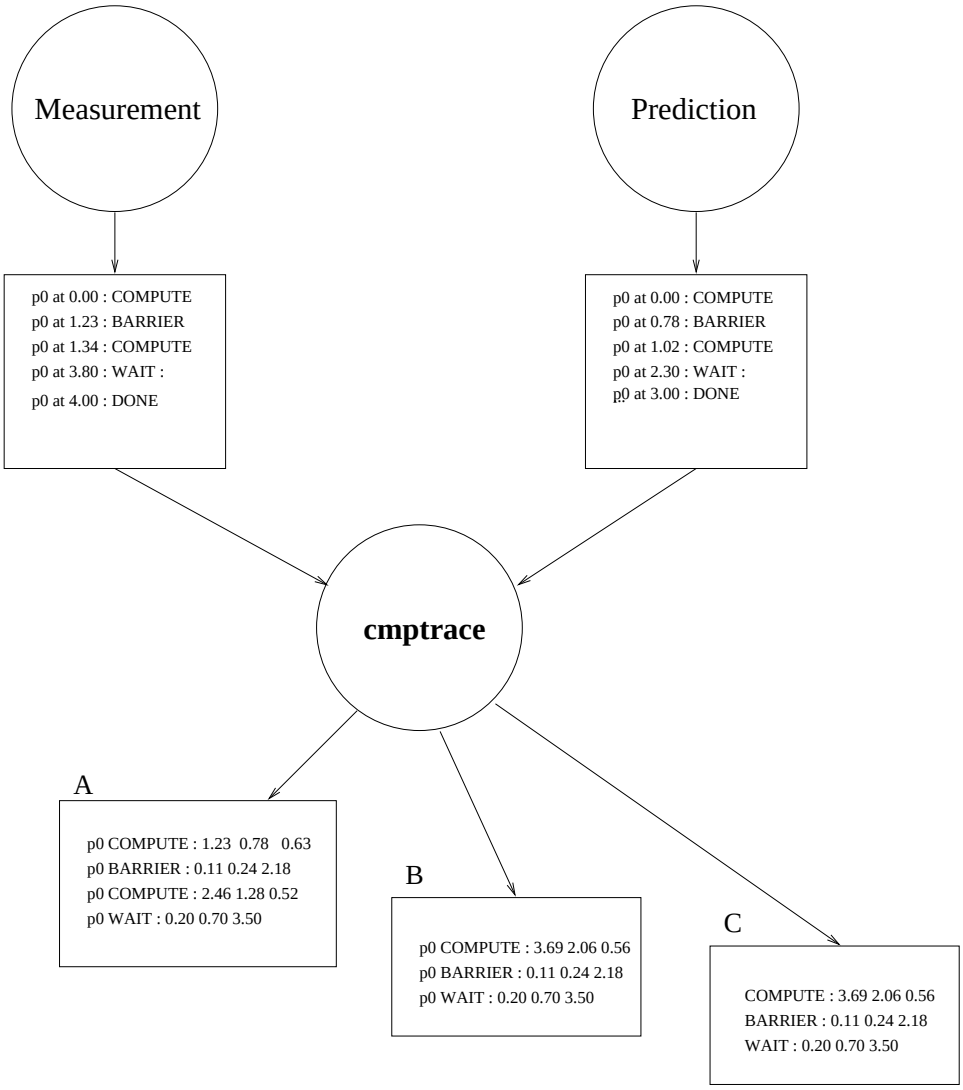

Figure 2.10: The trace comparison utility `cmptrace`.

```
      ALLGATHER ALLREDUCE COMMSPLIT DONE
$bars
p[0] State
p[1] State
$events
u:p[0] at 0.0:        P BARRIER
u:p[0] at 1.0:        P REDUCE
u:p[0] at 2.0:        P COMPUTE
u:p[0] at 3.0:        P BARRIER
u:p[0] at 4.0:        P REDUCE
u:p[0] at 5.0:        P COMPUTE

u:p[1] at 0.0:        P BARRIER
u:p[1] at 1.0:        P REDUCE
u:p[1] at 2.0:        P COMPUTE
u:p[1] at 3.0:        P BARRIER
u:p[1] at 4.0:        P REDUCE
u:p[1] at 5.0:        P COMPUTE
// ...
```

There are several output formats, with varying levels of detail. The first (**A** in the diagram) is a line by line comparison of *all* the events in the trace file. It has one line for each line in the input trace files, so may be very large.

```
p[0] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[0] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[0] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
p[0] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[0] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[0] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
// ...
p[1] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[1] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[1] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
p[1] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[1] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[1] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
// ...
```

The next output format (**B**) collates totals for each process in each state:

```
p[0] BARRIER       <t1> <t2> <t2/t1>
p[0] REDUCE        <t1> <t2> <t2/t1>
p[0] COMPUTE       <t1> <t2> <t2/t1>
p[1] BARRIER       <t1> <t2> <t2/t1>
p[1] REDUCE        <t1> <t2> <t2/t1>
p[1] COMPUTE       <t1> <t2> <t2/t1>
```

and the last summary format (**C**) gives totals for the states across all processes:

```
BARRIER   <t1> <t2> <t2/t1>
REDUCE    <t1> <t2> <t2/t1>
COMPUTE   <t1> <t2> <t2/t1>
```

The trace comparison utility uses the SIM++ trace format but could be extended to use other formats such as ALOG or Pablo.

The utility enables detailed comparisons of prediction techniques with actual measurements, and helps to pinpoint the failings (and successes) of the prediction techniques. It may also be used to compare the detailed performance of runs of the same program on different machines. Another use is determining the repeatability of measurements by comparing successive runs on the same machine.

Only one timing (predicted or measured) is given for each phase even though in a MIMD system each process will finish a phase at a different time. The time for a phase is taken to be the maximum time taken by all processors in the group.

2.5 Multiple runs

The designer of a parallel program may be designing for a fixed machine and problem size, in which case a performance prediction technique which provides a single number for the run time would suffice. However it is more likely that the design will have to encompass a range of machine and/or problem sizes leading to 2D graphs or 3D surfaces.

In addition, each sample point on the surface will be taken from a distribution (since delays will vary statistically), so the comparison must be between two 3D probability distributions.

Experiments to measure how performance varies with different data and machine sizes were controlled using an `experimentor` routine written using `Perl`.

Comparison routines were written to compare two graphs, returning a third graph giving the ratio of the first two.

Another approach would be to fit curves to both sets of data and compare the coefficients. However this would involve guessing the form of the equations, which may well be very complicated (and possibly non-linear).

The following subsections describe two utilities developed for displaying and comparing 3D surfaces obtained from multiple run experiments.

2.5.1 `mkgraph`: a utility to generate graphs

`mkgraph` is a utility for generating 3D surfaces. It is used from the Unix command line:

```
mkgraph <infile> <outfile>
```

The input format is:

```
# <nprocs1> <ndata1>
<phase1name> <time>
<phase2name> <time>
...
<totalname> <time>
# <nprocs2> <ndata2>
<phase1name> <time>
<phase2name> <time>
...
<totalname> <time>
... etc
```

As output it generates a separate data file for each phase which includes the speedup from the single processor time. It also produces GNUplot script files for displaying the data as a 3D surface.

2.5.2 `cmpgraph`: a utility to compare graphs

A utility for comparing two 3D surfaces was developed, `cmpgraph`. The utility is used from the Unix command line:

```
cmpgraph <infileA> <infileB> <outfile>
```

The format of the two inputs is the same as for `mkgraph`. The utility computes the ratio of the two input surfaces, and generates the data files and 3D GNUplot scripts for displaying them.

2.6 Conclusion

This chapter has described the suite of real MPI programs used for evaluating the design techiques described in later chapters. It is difficult to assess the accuracy of a prediction. The ultimate comparison is between predicted and measured trace files, so a tool was developed to perform this detailed line by line comparison (section 2.4.1). Scalability comparisons are also important, and tools were developed for controlling multiple runs (section 2.5) and for comparing the timing results (section 2.5.2).

Bibliography

[1] G.D. Burns, R.B. Daoud, and J.R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[2] T. Hey, J. Dongarra, and R. Hockney. PARKBENCH: Parallel kernels and benchmarks. available at `http://www.netlib.org/parkbench/html/`, 1996.

[3] R.N. Ibbett, P.E. Heywood, and F.W. Howell. HASE: A Flexible Toolset for Computer Architects. *The Computer Journal*, 38(10):755–764, 1995.

[4] *SIM++ v3.8 Reference Manual*, 1991.

[5] Sivan Toledo. Performance prediction with benchmaps. In *Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii*, pages 479–484, Los Alamitos, California, April 1996. IEEE, IEEE Computer Society Press.

[6] Gregory V. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser Verlag AG, April 1994.