Chapter 5

Reverse Profiling

Simple performance estimates such as those in the previous chapter are good for giving an overall picture of expected speedup for simple algorithms. However, the estimates are less valuable when processes are not all doing the same thing at the same time and when there are data dependent communications. A method for including these complexities is presented in this chapter. It is based on profiling, the well established technique for measuring how much time is spent in each part of a program.

5.1 Introduction

Reverse profiling [2] applies the MPI performance model for an architecture to a user's program to generate an estimate of the run time on that architecture. The model is generated automatically by the routines described in chapter 3. It uses the MPI profiling interface to intercept the user's calls to MPI functions and to calculate the expected delay before returning control to the MPI routine to do the actual work. In this way it is possible to estimate the expected run time of a program on any architecture using a workstation as the development platform. For example, a Cray T3D model may be used on MPI on a single workstation (or vice-versa). The technique is a simplified form of discrete event simulation. It provides quick results in the majority of cases (and will perform most of the calculations for the more complex ones).

Reverse profiling is easier to use than simulation techniques (to which it bears a resemblance), but it may only be applied to a subset of all parallel programs. The big advantage is that it may be applied simply by linking with the standard profiling library of the message passing interface. It involves each process keeping track of its own simulation time and updating it whenever an MPI function is called. This means a normal trace file can be generated. A model of any machine may be used, and any MPI implementation can be used as the development environment.

Because it does not involve full simulation, it may not be gainfully applied to non-deterministic routines, for example those employing dynamic load balancing. However, the performance model will provide the key design data for such routines (such as the minimum and maximum message times). For non-deterministic programs the method must be combined with pencil and paper calculations, or with times measured from the target machine. Non-deterministic programs are likely to strain simulators and profilers too, since a minor miscalculation of delay may affect the outcome. A large proportion of useful parallel programs are deterministic. Reverse profiling is a simple usable technique aimed at the majority of programs.

Running a reverse profiled MPI program produces a trace file which may be displayed as a timing diagram. Repeated runs may be used to produce graphs showing how performance varies with the problem size and number of processors in the machine. The machine model is supplied at run time as an environment variable pointing to a file produced by the MPI characterisation routines.

Section 5.2 describes the technique in detail; section 5.3 describes various ways of estimating computation delays, section 5.4 presents results obtained from applying reverse profiling to some of the Cowichan problems and section 5.5 concludes.

5.2 The technique in detail

The MPI interface provides a simple profiling interface; all the `MPI_` functions are also accessable with the prefix `PMPI_`. Profiling (or reverse profiling) code may be added by writing substitute `MPI_` functions which perform the necessary (reverse) profiling task and call the `PMPI_` function to do the actual work. The linker ensures that the appropriate functions are called. The compilation commands to compile a normal MPI program, to compile with a profiler and to compile with the reverse profiler are:-

```
cc prog.c -lmpi
cc prog.c -lprof -lpmpi -lmpi
cc prog.c -lrevprof -lpmpi -lmpi
```

Each process has a variable (named `the_time`) to store its current simulation time. The profiled versions of the MPI functions update `the_time` according to the performance equation for that function and write lines to the trace file.

For point-to-point communications the receiver needs to know the time the sender started sending the message in order to work out when it should arrive (figure 5.1). The minimum delay at the receiving end occurs when the message has been posted by the sender well in advance and the message has only to be copied from a system buffer. If the `send` starts at the same time as the `recv`, the receiver will suffer an additional wait time for the message to arrive. This will be worse if the sender starts after the receive does.

These delays may be estimated from several measured parameters (determined by the routines described in chapter 3):

T_{send} Time for `MPI_Send` to complete.

T_{recv} Time for `MPI_Recv` to complete if started at the same time as the corresponding `MPI_Send`.

$T_{recvmin}$ Minimum time for `MPI_Recv` if the message has already arrived.

The total delay at the receiver is given by:

$$T_{recvmin} + T_{wait}$$

$$T_{recvend} = max(T_{recvstart} + T_{recvmin}, T_{sendstart} + T_{recv})$$

so

$$T_{wait} = T_{recvend} - T_{recvstart} - T_{recvmin}$$

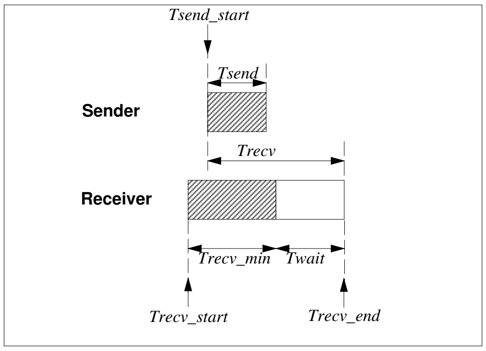

Figure 5.1: Times involved in point-point delay calculation.

For collective operations involving synchronisation (i.e. the majority of them), each process must know the start time of every other.

Thus a point-point reverse profile function looks like:

```
int MPI_Send( data, dest, ...)
{
  // Send the_time to the destination
  PMPI_Send(the_time, dest, ...);
  the_time += /* computed delay for the message */;

  // Perform the actual send
  PMPI_Send( data, dest, ... );
}

int MPI_Recv( ... )
{
  // Recv the sender's start time
  // Compute the recv delay the_time
  // function of ( the_time, sender_start, message size )
}
```

and a collective operation:–

```
int MPI_Barrier()
{
  // MPI_Allgather to get each process's the_time
  // Set local the_time to the latest of all the_times
  // Plus the computed delay for the barrier.
}
```

This works as long as two conditions are met:

1. MPI_Recv is not allowed wildcarded receives. This is because there are two receives (one for the sender time, one for the actual data) which couldn't be guaranteed to come from the same source. This problem is related to the non-determinism issue raised earlier.

2. Collective operations imply synchronisation.

A trace file is generated which may be displayed with a timing diagram tool. Each process generates a separate trace file (`p0.trace, p1.trace, etc`). Repeated runs may be combined to produce scalability graphs.

5.3 Estimating the computation delays

The reverse profiling technique has so far accounted for the communication costs quite happily, but the times for user code have not been accounted for. Even without considering compute times, useful results may be obtained since the amount of time spent in idle "wait" states can be measured from the timing diagram and the communications structure of the code is clearly visible. None of the techniques thus far encountered for estimating computation times are entirely satisfactory.

Several options for including computation are:

1. Fix it at 0. This is the dual of the PRAM model which sets the computation cost at 1 and makes communication cost 0.

2. Let the user estimate it (in units of seconds, or number of memory accesses, arithmetic operations, etc.)

3. Cycle count the assembly code.

4. Measure the times on the fly.

5. Measure the important times with a profiler off line.

Arguments may be made for all the above approaches. They make different tradeoffs between accuracy and ease of use.

Option 1, ignoring computation altogether, yields graphs showing the total communication time for an algorithm on a machine, which may be useful in itself as it shows how computation time must scale in order to make use of the machine. Option 2 is surprisingly useful. The programmer adds calls to a "`compute(N)`" macro which adds N "time steps" to the local simulated time, where a "time step" is the time taken to perform an arithmetic operation. This time is highly variable because of the influence of the memory hierarchy, but may be bracketed between likely limits (e.g. between 1 and 10 microseconds). This time step can be given as a parameter to the reverse profiler, so one may check how a design fares when given minimum expected compute step time and maximum expected communications time (the worst case for parallel algorithm scalability). Saavedra-Barrera [3] describes characterisation routines for measuring the performance of different classes of operations in Fortran and if such figures were generally available for sequential code it would make parallel design easier.

Cycle counting of assembler code (option 3) is the preferred choice of parallel machine simulators. This technique has been shown to yield very accurate time estimates [1]. It involves an extra compilation stage, with the assembly code for the application being interpreted and augmented by a routine which inserts instructions to update a global cycle count after each basic block. Since the number of cache misses may lead to an order of magnitude variation in the execution time, a cache model is required for such simulators. This technique also requires augmented versions of all libraries used.

Experience using the Proteus `augment` tool indicated that though the technique works, it is too time consuming and awkward for quick estimates of compute time. It is also a "black box" approach and it it hard to know how reliable the estimates will be.

Option 4, measuring the compute times on the fly, is tricky on a multi-tasking system. Some multi-threading libraries provide "virtual timers" which only measure compute time consumed by the current thread, but these are not generally available. In any case, the compute times would have to be scaled for the target architecture.

The final option, profiling important subroutines on the target system and feeding the numbers back into the reverse profiler yields the most believable numbers, but requires the most effort on the part of the parallel program developer.

5.4 Results

Reverse profiling scores over the simpler technique presented in chapter 4 wherever there are data dependencies, since these are hard to incorporate into a purely analytical model. To test the suitability of reverse profiling for developing real programs, the technique was applied to a selection of the Cowichan problems. The results are presented below.

The trace output from a single run is a timing diagram showing how much time each process spends communicating and computing. This diagram is compared directly with a measured timing diagram; this is the ultimate test of a performance prediction. If a technique can generate an accurate timing diagram then it has solved the prediction problem. In practice there will be discrepencies between predicted and measured diagrams.

Comparing speedup curves is a more forgiving method of testing a prediction - even if the prediction is several orders of magnitude out in absolute terms it may produce a similar speedup curve to that measured.

The comparisons below focus on timing diagrams since they provide the ultimate check. The problem with timing diagrams is that they include a very large amount of information, so sample extracts are presented to illustrate specific aspects of the predictions.

In the diagrams the light shade represents computation and the dark stripes are communication steps. A bar is given for each processor (`p[0], p[1],...`). The bottom bar ("`All`") shows the overall phase. On screen the different communication operations are distinguishable by different colours. The two vertical measuring lines **O** and **X** allow time intervals to be measured and displayed.

The sections below illustrate the reverse profiling process using examples from the Cowichan problems described in chapter 2.

5.4.1 Mandelbrot set generation (`mandel`)

This is an example with data dependent amounts of work to perform. In the simple technique of the previous chapter, an upper bound on the quantity of work had to be estimated (the maximum number of iterations). Reverse profiling bases timing calculations on the actual data used. This means that detailed timing diagrams can be generated, showing the expected behaviour of each processor individually, rather than the broad brush single equation for overall time used previously.

Figure 5.2 shows the predicted timing diagram produced using reverse profiling for the Mandelbrot set problem on eight SPARCstation 5 workstations connected using Ethernet, with a matrix size of $160x160$. The computation times are largest for processors 3 and 4 which compute the points inside the set, and all processors have to wait for these to complete before the problem is done. The overall time is estimated at $0.71s$ of which $71ms$ is spent in the initial barrier synchronisation, 10% of the total. Figure 5.4 shows the measured timing diagram. The measured diagram is less regular than the prediction as processors are imperfectly synchronised, leading to the staggered effect of the diagram. The total measured time is $0.23s$, a factor of 3 better than predicted, because the computation was overestimated. The initial barrier takes $78ms$, close to the expected $71ms$.

The predicted diagram for 8 processors of the Cray T3D is shown in figure 5.3. The time is expected to be determined by the maximum computation time; the initial barrier time is now insignificantly small, much less than the smallest compute time. This was confirmed by the measurement (figure 5.5), but again the absolute values of the computation time were overestimated, in this case by a factor of five.

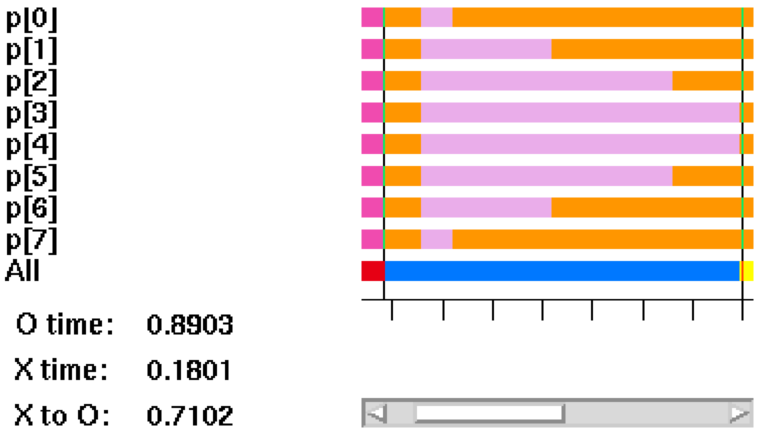

Figure 5.2: The `mandel` routine: predicted timing diagram for 8 workstations.

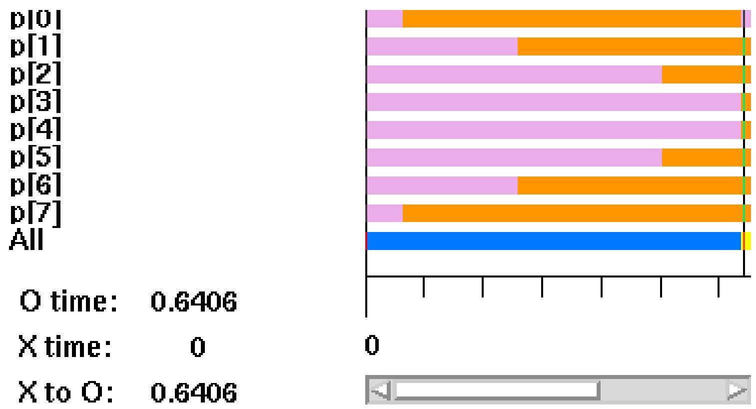

Figure 5.3: The `mandel` routine: predicted timing diagram for 8 processors on the Cray T3D.

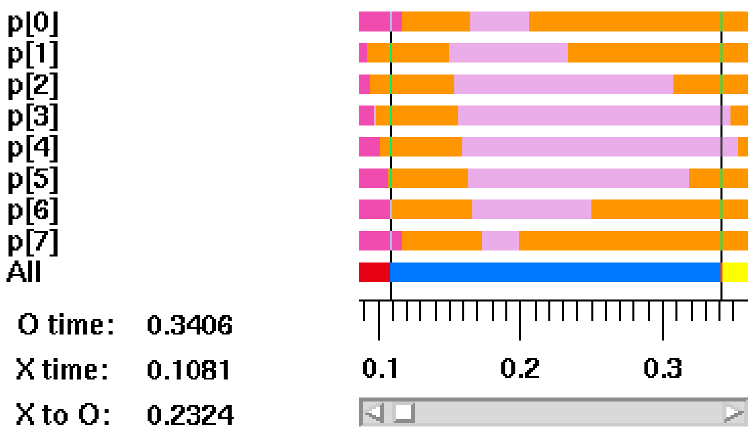

Figure 5.4: The `mandel` routine: measured timing diagram for 8 workstations.

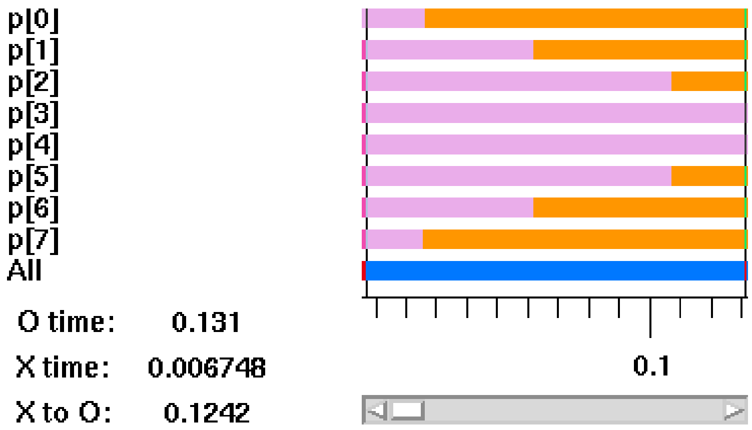

Figure 5.5: The `mandel` routine: measured timing diagram for 8 processors on the Cray T3D.

5.4.2 Elastic net simulation (`elastic`)

This example presents a quantitative comparison between timing diagrams produced by successive profiling runs, and between forward and reverse profiles. It also illustrates how I/O models may be incorporated into the reverse profile.

No two runs on a machine will take *exactly* the same time. The variation between runs on a machine provides a bound on the accuracy which can be expected from a prediction.

Table 5.1 compares the results of two runs of the `elastic` routine. The measurements were taken on 2 processors of the T3D, and the input parameter size was 128. The ratio of times varied from 0.53 for **BUSY** to 1.25 for **ALLGATHER**, a factor of 2 variation in measured times.

Phase	Prof1 (s)	Prof2 (s)	Ratio (prof2/prof1)
SEND	0.004571	0.004605	1.007365
RECV	0.367272	0.357369	0.973037
BARRIER	0.242121	0.183040	0.755984
BUSY	3.949576	2.121767	0.537214
BCAST	0.002806	0.003097	1.103726
ALLGATHER	0.004120	0.005161	1.252807
GATHER	0.249811	0.218897	0.876252
ALLREDUCE	22.954994	18.272695	0.796023

Table 5.1: Two profiles of `elastic` compared.

Table 5.2 shows the results of comparing forward and reverse profiles of `elastic`.

- SEND, BUSY, ALLREDUCE and GATHER are within a factor of 3.

- BARRIER and BCAST are out by 1000.
- ALLGATHER and RECV are out by 10.

The big errors occur where there is synchronisation. The times shown are the total times spent in each phase across all processors - so if one processor is delayed by t seconds the time for the next synchronisation operation will be extended by Nt seconds. This causes major proportional errors, particularly for synchronisation operations which execute quickly (such as the barrier). The effect of these errors in absolute terms is less significant, since the program is delayed by the t rather than the Nt.

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.018175	0.031034	1.707511
RECV	0.165123	0.020715	0.125453
BARRIER	0.262404	0.000517	0.001971
BUSY	23.825751	41.324430	1.734444
BCAST	0.225873	0.000402	0.001780
ALLGATHER	0.003093	0.000290	0.093749
GATHER	0.000258	0.000293	1.133884
ALLREDUCE	3.274713	1.311991	0.400643

Table 5.2: Forward vs Reverse profiles of `elastic`.

To determine the causes of these errors in more detail, it is necessary to examine the timing diagrams.

Figure 5.6 shows the measured and predicted timing diagrams alongside. For this experiment, there is an overall error of 50%. Figure 5.7 shows a detail from the inner loop, which consists of long **BUSY** phases ended with an **ALLREDUCE**. The **BUSY** phases are overestimated by about 60%, and the **ALLREDUCE** is taking longer on process 1 than process 0 (whereas the prediction is for them to take the same time). Table 5.3 shows the overall times for allreduce for each processor. The explanation for this apparently major discrepancy in the allreduce prediction (taking 4 times longer than predicted on processor 1) is the variation in compute times between the two processors. Figure 5.8 shows one iteration of the inner loop. The busy time of processor 0 is 1900us, 300us greater than the busy time of processor 1 (1600us). Because the subsequent allreduce implies a synchronisation, and because this discrepancy between the compute times of 0 and 1 was not predicted, the 300us is added onto processor 1's allreduce time. There is no easy way to measure the proportion of the allreduce time spent waiting for synchronisation and the proportion actually performing the allreduce (since the synchronisation is integral to the algorithm used to implement the allreduce and not a separate operation).

Figure 5.9 shows the allreduce discrepancy in more detail. The measured time of $116us$ for the allreduce on processor 0 is close to the $102us$ predicted time; the large discrepancy is caused by processor 1 finishing its computation phase before processor 0.

Processor	Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
p[0]	ALLREDUCE	0.763569	0.655996	0.859118
p[1]	ALLREDUCE	2.511144	0.655996	0.261234

Table 5.3: Total times for `ALLREDUCE` for each processor.

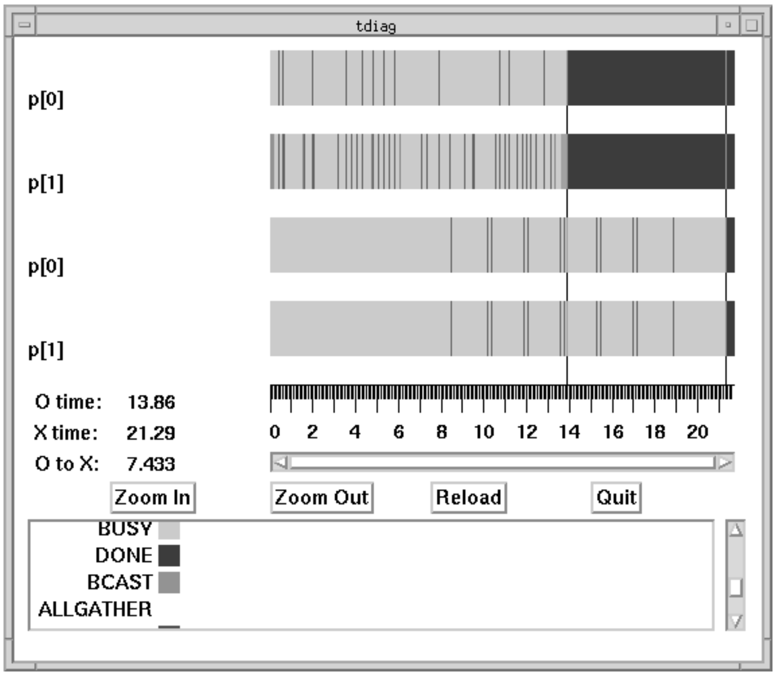

Figure 5.6: Measured (top) and predicted (bottom) timing diagrams for `elastic` - top level.

Figure 5.10 illustrates the measured time taken for the following line:

```
if (rank==0)
  printf("[%d] elastic_t iteration %d\n", net.rank, iter);
```

Output is costly, taking $3ms$ in this case. Only processor 0 performs the output, but both processors are delayed as there is a subsequent send/recv exchange of data. This appears on the diagram as an extended recv period.

This can be incorporated into the reverse profile as well - even a small amount of I/O has an effect. This may be done by changing the above line to:

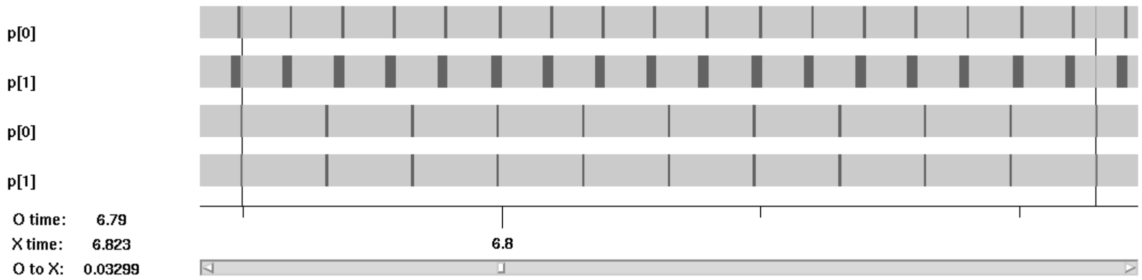

Figure 5.7: Measured (top) and predicted (bottom) timing diagrams for `elastic` - detail inside inner loop.

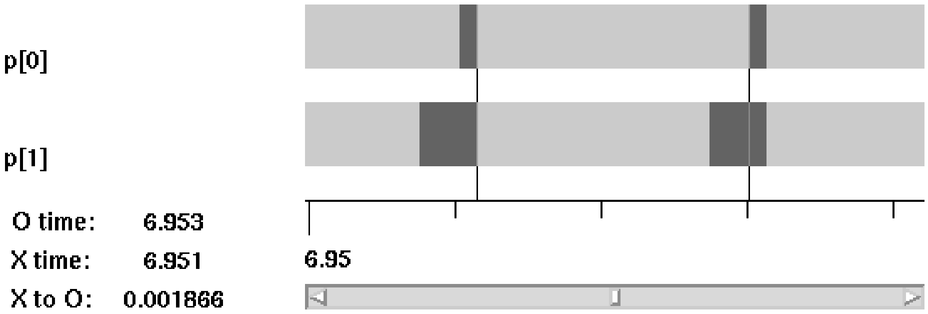

Figure 5.8: Measured timing diagram for `elastic` - one inner loop iteration.

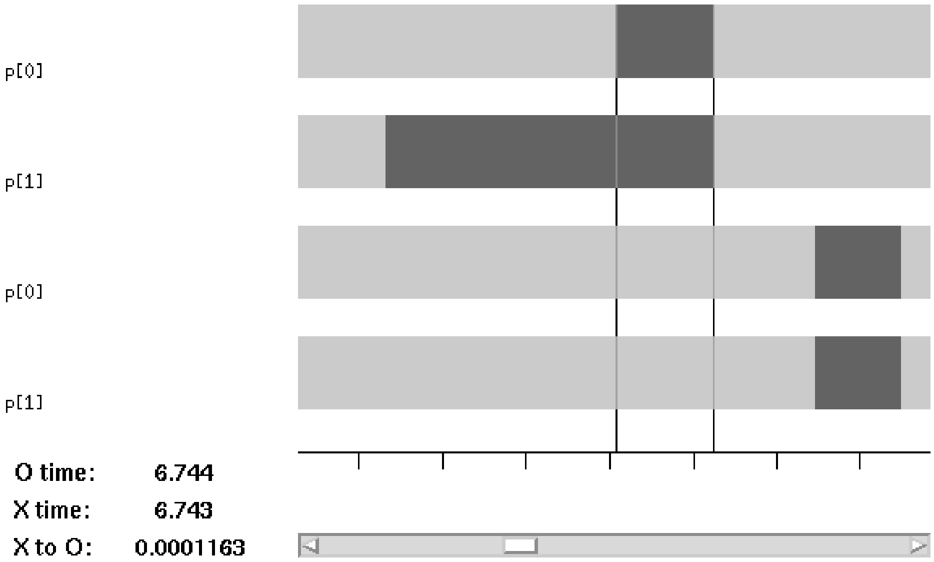

Figure 5.9: Measured (top) and predicted (bottom) timing diagrams for `elastic` - the **allreduce**.

```
if (rank==0) {
  printf("[%d] elastic_t iteration %d\n", net.rank, iter);
  compute(3000);
}
```

The equivalent extract from the timing diagram is shown in figure 5.11.

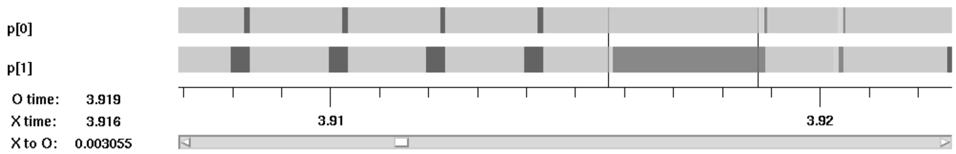

Figure 5.10: Measured time for `printf()` in each outer loop.

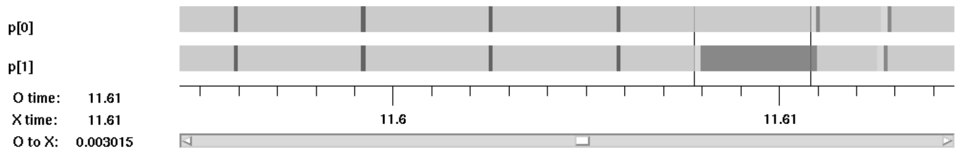

Figure 5.11: Predicted time for `printf()` in each outer loop.

Figure 5.12 shows the initial measured and predicted timing diagram for 4 processors.

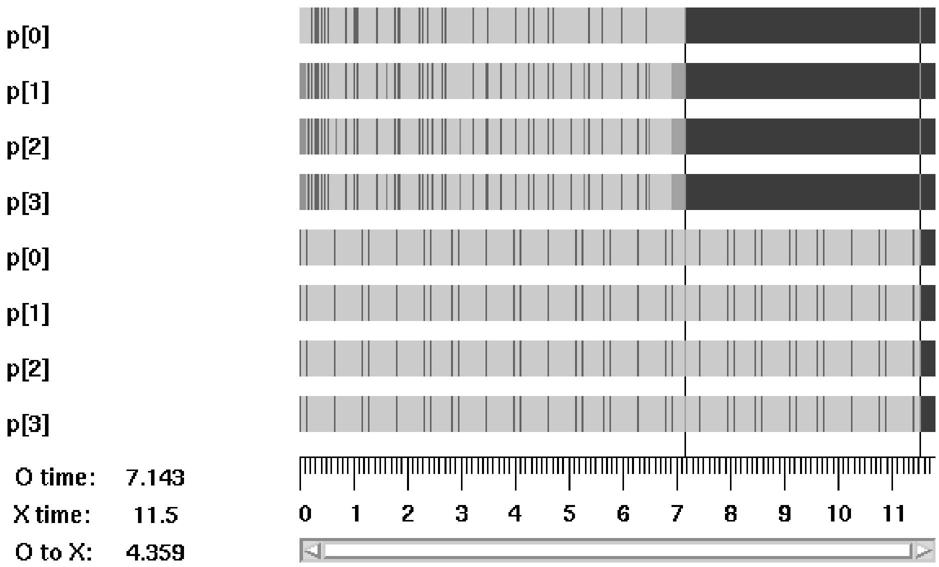

Figure 5.12: Measured and predicted times for elastic with 4 processors.

Figure 5.13 shows that the reason the predicted time is about 50% too long is that the compute phases have been overestimated; the shorter `allreduce` communications phases have been accurately predicted. The distribution of the times actually taken for allreduce is shown in figure 5.14. This is concentrated at $150us$ with outliers scattered at higher delays. Figure 5.15 shows the distribution for busy times, with the peak at $800us$. Figure 5.16 shows the distribution of the *ratios* of predicted to measured performance; the bulk of data is clustered from 1.8 to 2 indicating that compute times are being overestimated by a factor of 2.

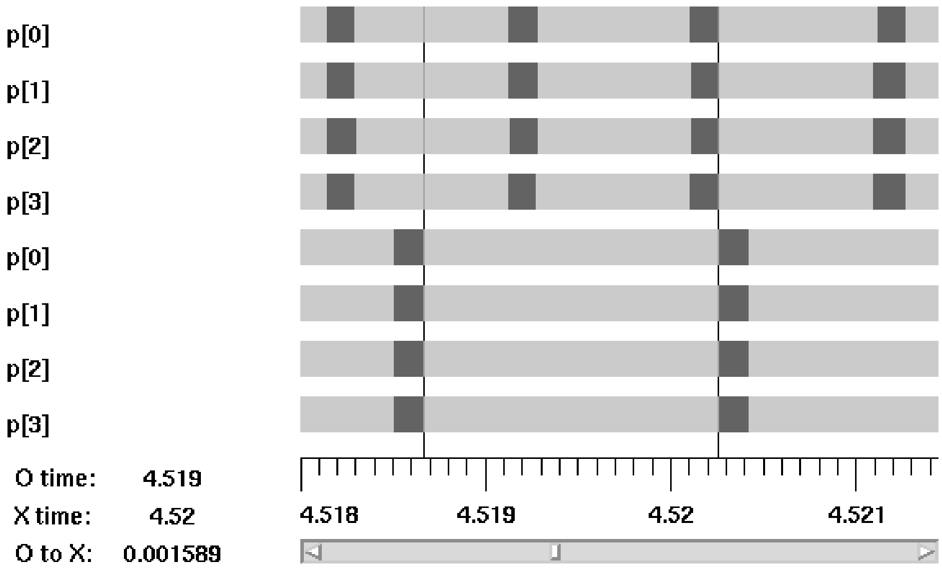

Figure 5.13: Measured and predicted times for elastic with 4 processors.

This example has shown that a prediction cannot be expected to produce a result within a factor of two, since there is a factor of two variation in successive runs. Predictions of synchronisation times are likely to be over optimistic, since prior variations across processors build up and are incorporated into the total measured synchronisation time. I/O has to be incorporated into models; even a simple `printf` reporting status takes milliseconds and could decimate available speedups.

5.4.3 Outer product (`outer`)

The example illustrated below is part of the `outer` routine. It illustrates the effect of varying the compute step time on the performance predictions and shows speedup graphs generated using reverse profiling.

`outer` is given a set of N points and computes the distance of each point from every other point. These distances are stored in a $N \times N$ matrix. Since the distance from point A to point B is the same as from B to A, the matrix is symmetric about its diagonal. For N points, $N^2/2$ distance computations are needed.

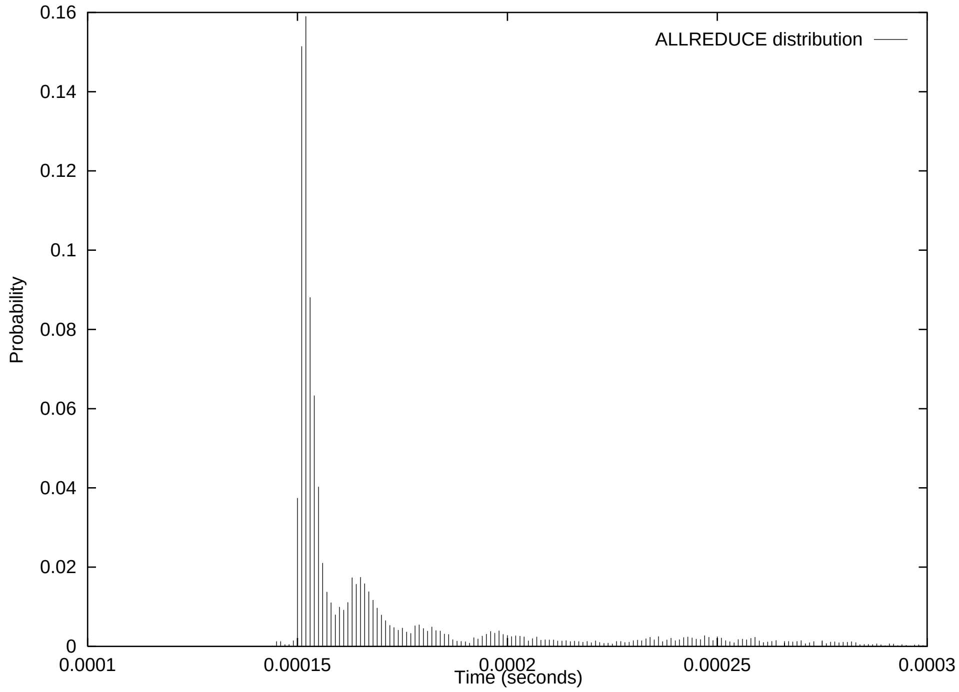

Figure 5.14: The measured distribution of `allreduce` times during elastic with 4 processors.

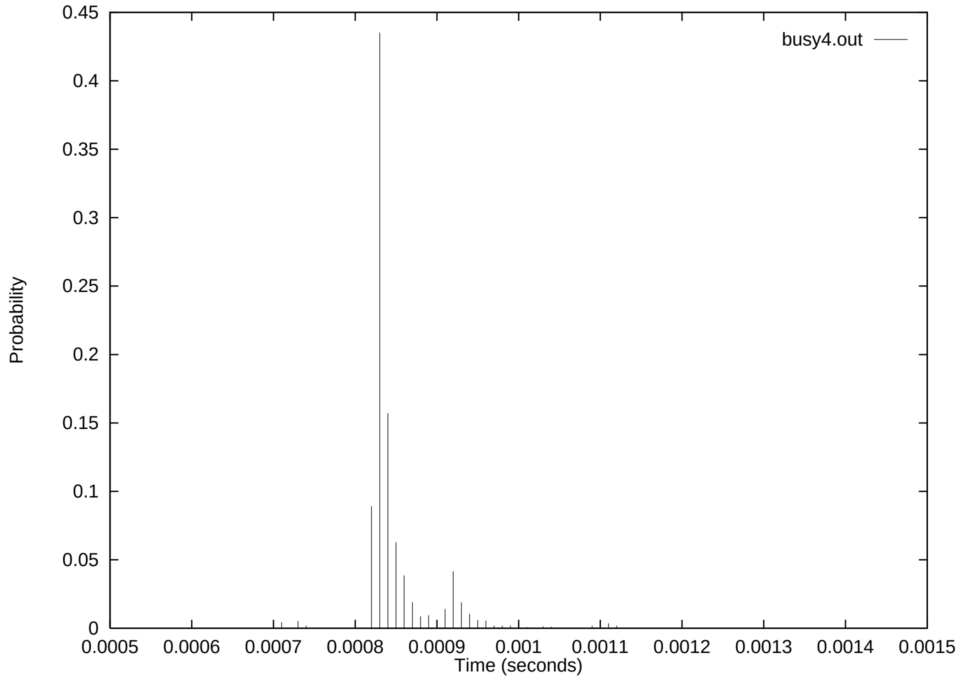

Figure 5.15: The measured distribution of compute times during elastic with 4 processors.

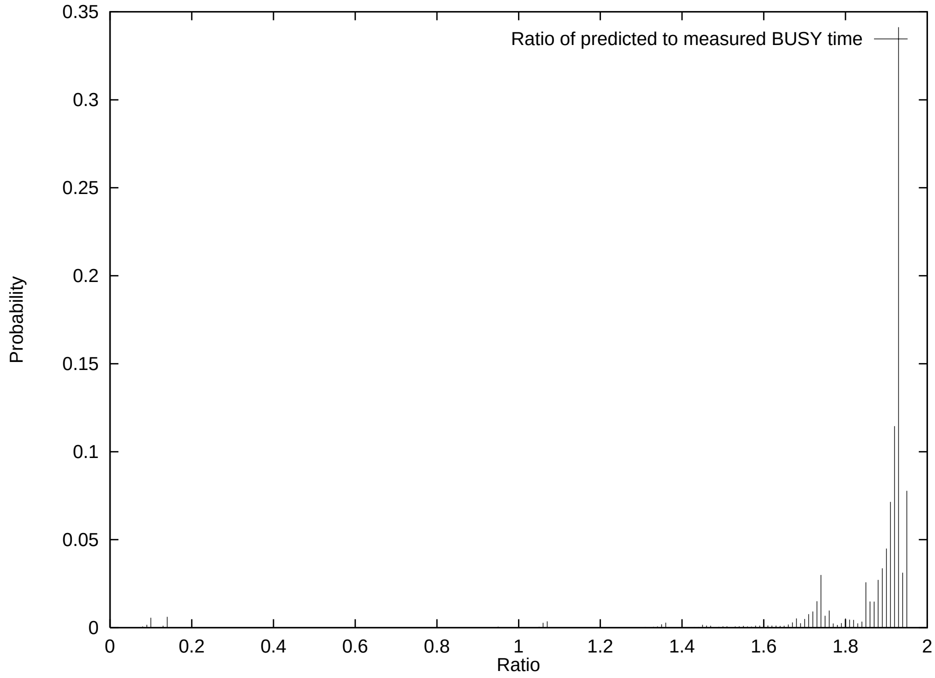

Figure 5.16: Distribution of the ratio of predicted to measured compute times.

In practice it is easier to perform N^2 distance computations than to perform half the computations and redistribute the matrix. Figure 5.17 shows the two possible distributions and figure 5.18 illustrates what the triangular redistribution would involve. Performing a triangular copy for a distributed matrix is not trivial using MPI. The MPI standard gives an elegant example of the use of datatypes to perform such copies where an entire matrix is stored on a single processor. However, coding such a copy using `MPI_Alltoallv` is not possible since `MPI_Alltoallv` requires all elements to have the same datatype. The elements in this case are irregularly shaped areas on the matrix. What is required is `MPI_Alltoallvi`, with a separate datatype constructed for each of the irregularly shaped areas. This is not part of the standard, so this approach was not taken.

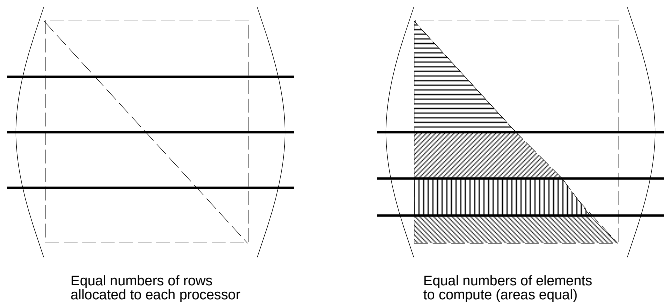

Figure 5.17: `outer` : possible distributions

Figure 5.19 shows the measured and predicted speedups, which correspond

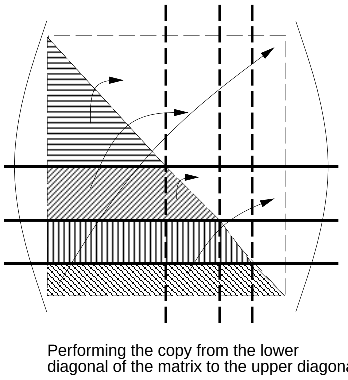

Figure 5.18: `outer` : performing the triangular matrix copy

reasonably with a compute step set between $0.1us$ and $1us$. Figure 5.20 shows that with a compute step as fast as $10ns$ the algorithm would yield minimal speedup as communication time would dominate.

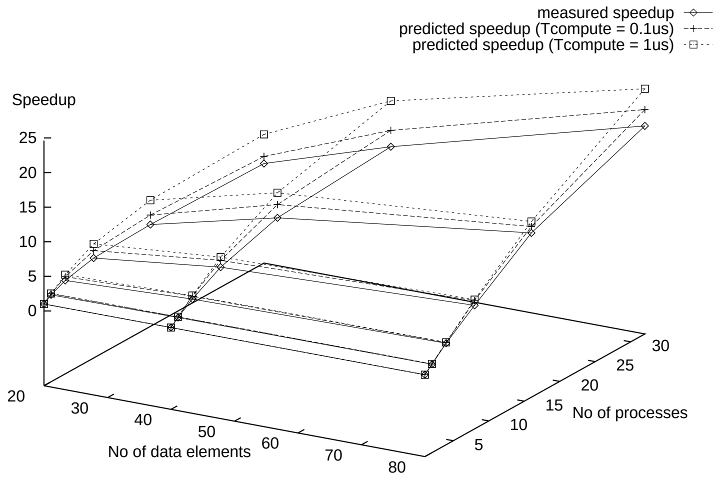

Figure 5.19: `outer` : predicted and measured speedups on the Cray T3D

5.4.4 Image threshholding (`thresh`)

Figures 5.21 and 5.22 show the predictions and measurements of the computation phase with 32 processors. The prediction is overestimated by a factor of 2 (73ms predicted, 36ms actual). It is also interesting to note that the durations of operations vary more in the measurements than the predictions. This is because collective operations are predicted to complete when all processors have completed them - but the measurements determine the exact return time (which will be different for each processor).

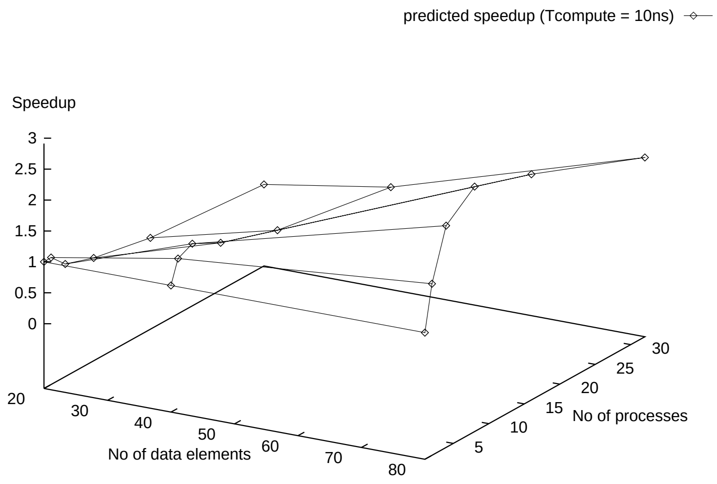

Figure 5.20: `outer` : predicted speedup with Tcompute=10ns

This simplification was made to make predictions tractable. The performance equation for `allreduce` (the collective operation which is causing the variation in this case) returns a single expected time for the allreduce across all processors. In practice, because of the algorithm used to implement the allreduce, some processors return more quickly than others, and this results in the ragged edge on the timing diagram.

The effect of this on the accuracy of predictions is not major in practice, since the discrepencies are compensated for on the next synchronisation. Those processors finishing first just have longer to wait at the next barrier.

5.4.5 The game of life (`life`)

The examples above use the *average* values to predict communications times. However the communications models of chapter 3 include more detail. This example shows how the expected minimum and maximum communications times may be used. It also illustrates the consequences of including network contention in the point to point communications models.

Figure 5.23 shows the predicted timing diagram for the game of life on two processors. Computation dwarfs the communication steps (the vertical lines on the diagram).

On 32 processors, the communication has started to make an impact (figure 5.24), taking $255us$ out of a total of $1300us$ for each life iteration (20% of the total; figures were measured from the timing diagram.) For 2 processors,

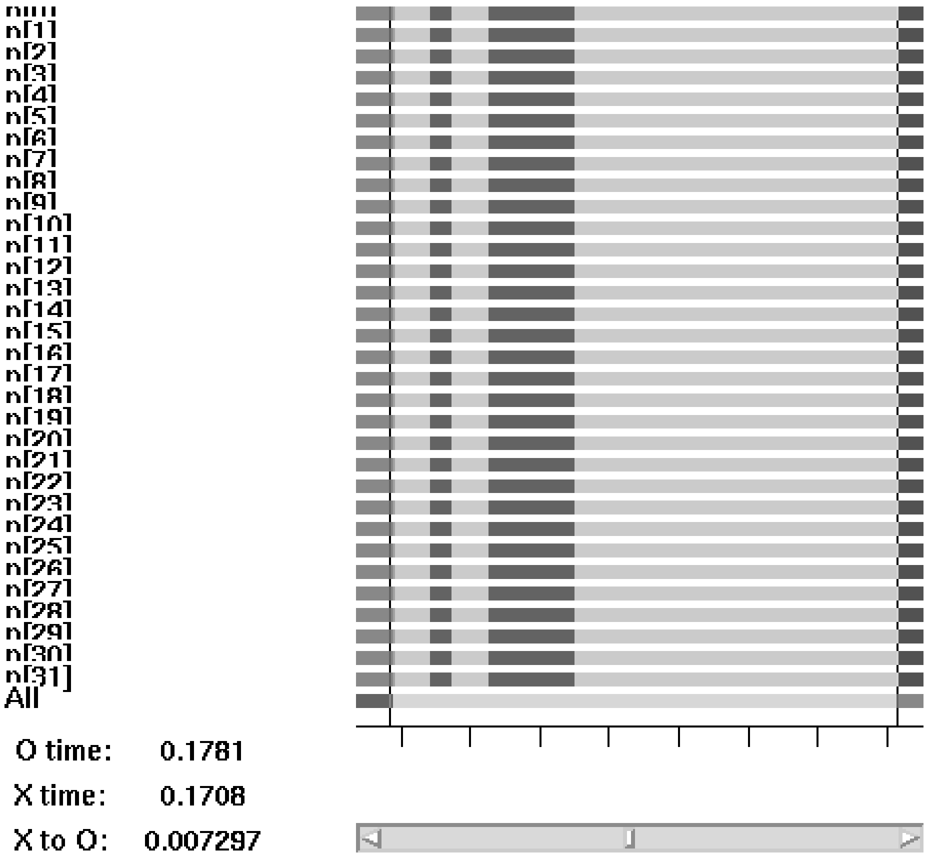

Figure 5.21: The `thresh` routine; predicted timing diagram for 32 processors.

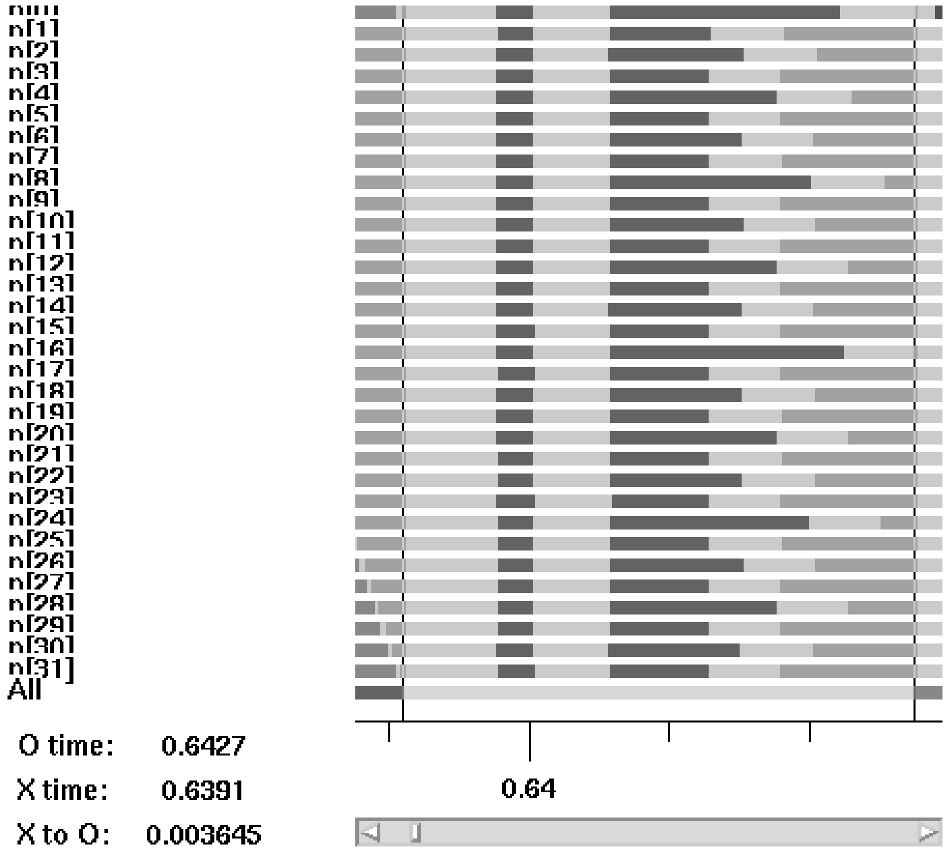

Figure 5.22: The `thresh` routine; measured timing diagram for 32 processors.

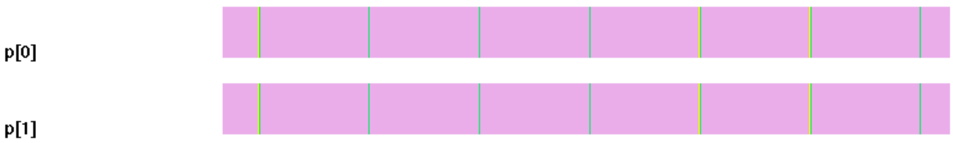

Figure 5.23: The `life` routine; predicted timing diagram for 2 processors.

communications was taking 255us out of a total of 16900us for each life iteration (1.5%).

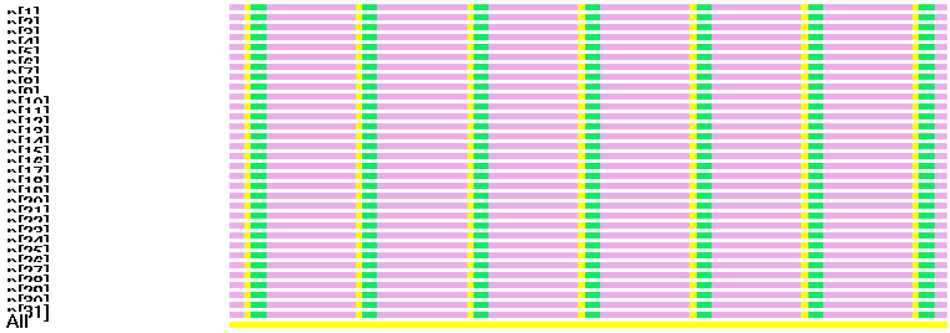

Figure 5.24: The `life` routine; predicted timing diagram for 32 processors.

On 32 processors however, the reading and writing of files is expected to dominate, with the computation burst of activity taking only 7% of the time. Figure 5.25 shows the central computation band.

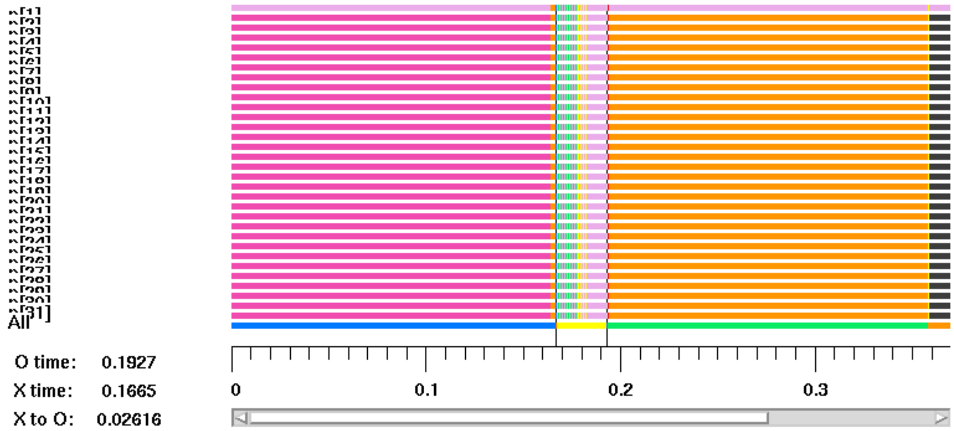

Figure 5.25: The `life` routine; predicted timing diagram for 32 processors (top level).

Comparing with measurements, figures 5.26 and 5.27 show the expected and actual times for the computation phase of life with 32 processors. Visually they are similar; the actual total time for the phase is 19.5ms and it was predicted at 26ms.

Table 5.4 compares the total send and receive times spent by all processors. Send is underestimated by a factor of 2.4 and Recv is overestimated by 35%. This send error is a concern and is a result of characterising the performance of a send on an uncongested network.

Replacing the send predictions with a characterisation based on a congested network (timing the send when one half of the processors are sending to the other

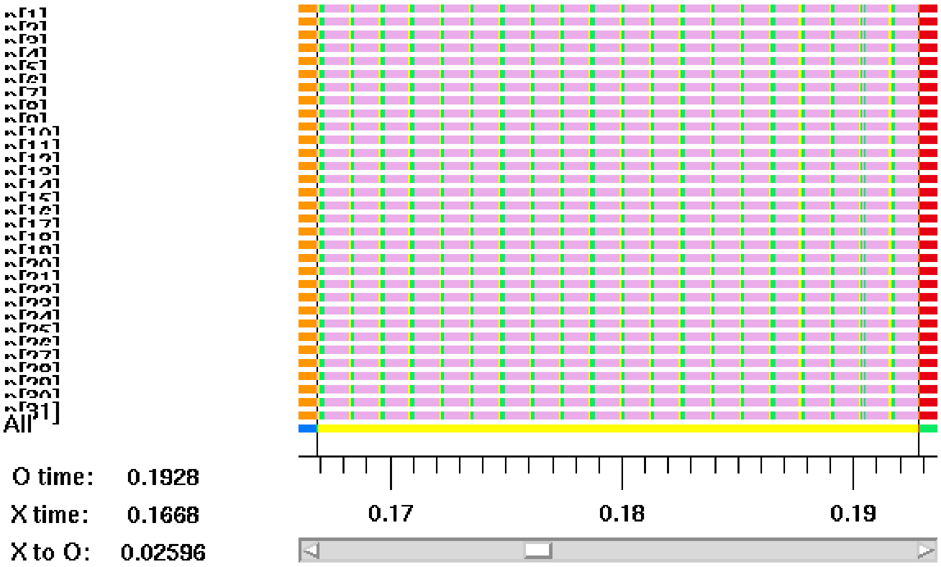

Figure 5.26: The `life` routine; predicted timing diagram for 32 processors (computation phase).

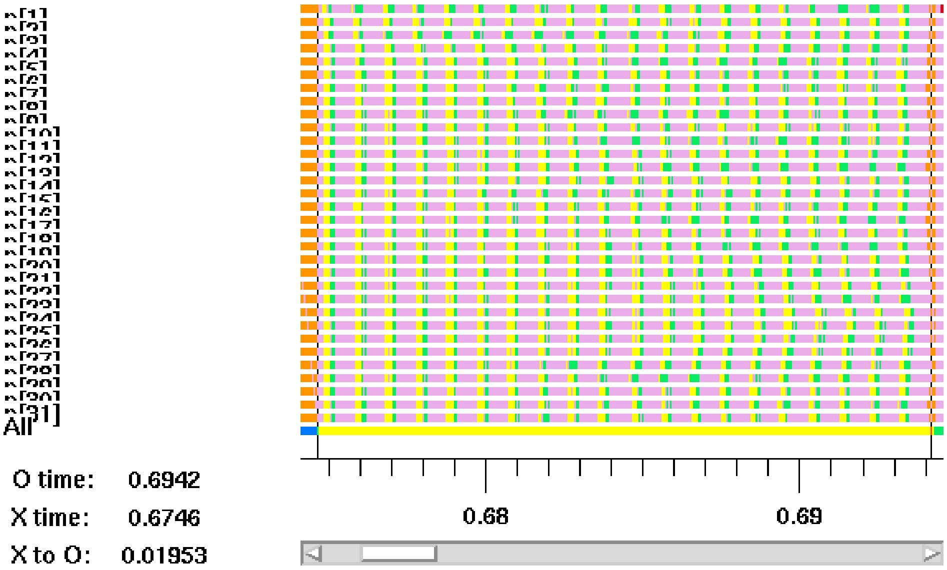

Figure 5.27: The `life` routine; measured timing diagram for 32 processors (computation phase).

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.118	0.049	0.41
RECV	0.085	0.114	1.34

Table 5.4: Forward vs Reverse profiles of `life` : using simple send prediction.

half) yields more accurate times (table 5.5):

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.118	0.084	0.71
RECV	0.085	0.109	1.29

Table 5.5: Forward vs Reverse profiles of `life` : using congested send prediction.

The equations for MPI performance give confidence intervals on the parameters, so predictions are possible based on both minimum and maximum expected communications times. Figure 5.28 shows the computation phase of `life` using the maximum and minimum expected communications delays. The maximum total time for the phase is $28.1ms$, the minimum $25.7ms$. Each communication step in the inner loop has a maximum expected time of $363us$ and a minimum of $244us$. Figure 5.29 shows the variation in the time for a single neighbour exchange communications step. The actual measured time of each communications step is $360us$, within the predicted range.

These results were obtained by setting the environment variable:

```
export REVPROFMODE=MIN
export REVPROFMODE=MAX
or export REVPROFMODE=AVG
```

before running the program.

Bracketing the computation time is also important; this may be done by using different values of the `tcompute` computation step time (set within the MPI characterisation file).

For example, improving computation performance by a factor of 10 leads to the profile in figure 5.30. The computation phase time is now $8.2ms$ (down from $26.9ms$ before), i.e. a factor of 3 improvement in algorithm performance can be expected from a factor of 10 improvement in processor performance. At this computation rate, communications takes 75% of the time of each life iteration.

Going the other way, slowing the computation time down by a factor of 10 yields a total computation phase time of $213ms$. Communications only occupies 3% of the time.

5.4.6 Weighted point selection (`winnow`)

The input for the test of `winnow` (weighted point selection) was the Mandelbrot set. The predicted timing diagram for the first phase of computation is shown in figure 5.31. The load is not balanced; the input Mandelbrot set has high values

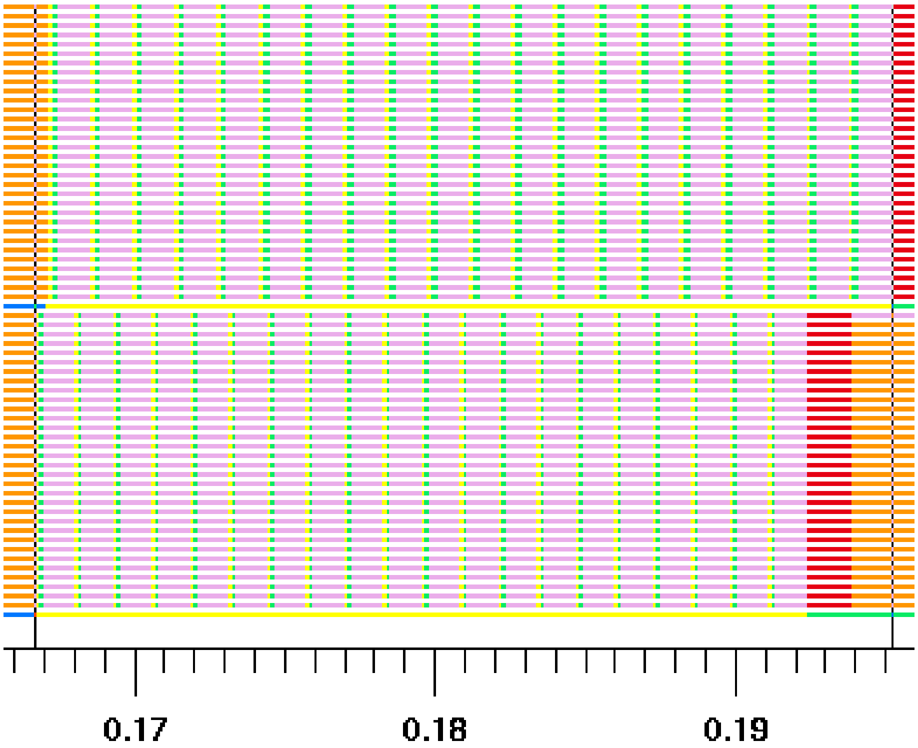

Figure 5.28: The `life` routine; predicted timing diagram for 32 processors using maximum (top) and minimum (bottom) expected communications times.

concentrated in the centre, so a disproportionate amount of computation is done by the central processors.

Figures 5.32 and 5.33 show the predicted and measured times for the top level of winnow computation. The measurement is twice as fast as the prediction. This error is due to the computation running approximately twice the expected speed; the predictions of communication times are within expected limits. The extreme imbalance of the load, with processor zero executing a very long computation step and holding up the rest of the processors is highlighted by the prediction.

5.5 Conclusions

This chapter has illustrated the application of reverse profiling with examples drawn from the Cowichan problems. Communications dominated programs are predictable to within a factor of two, and programs dominated by computation to a factor of ten. This is not as accurate as simulation techniques presented in the literature, but is sufficient to answer the question: "will this application run faster or slower on a parallel machine?".

The important features of the technique are:

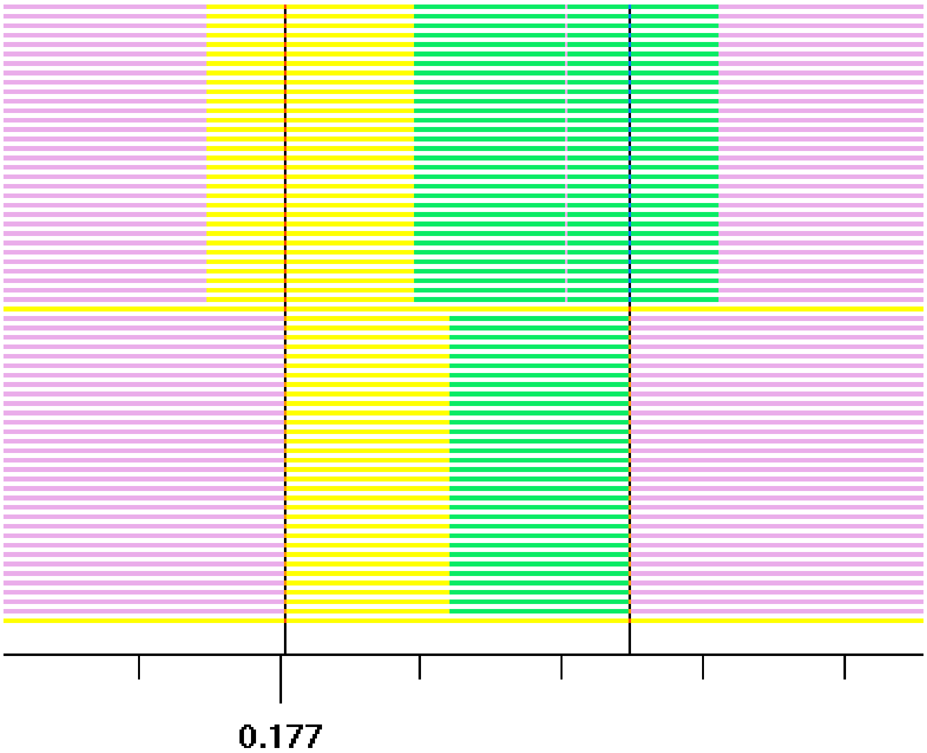

Figure 5.29: The `life` routine; predicted timing diagram for 32 processors (detail). The top half shows the maximum expected time for a communication step, the bottom half shows the minimum expected time.

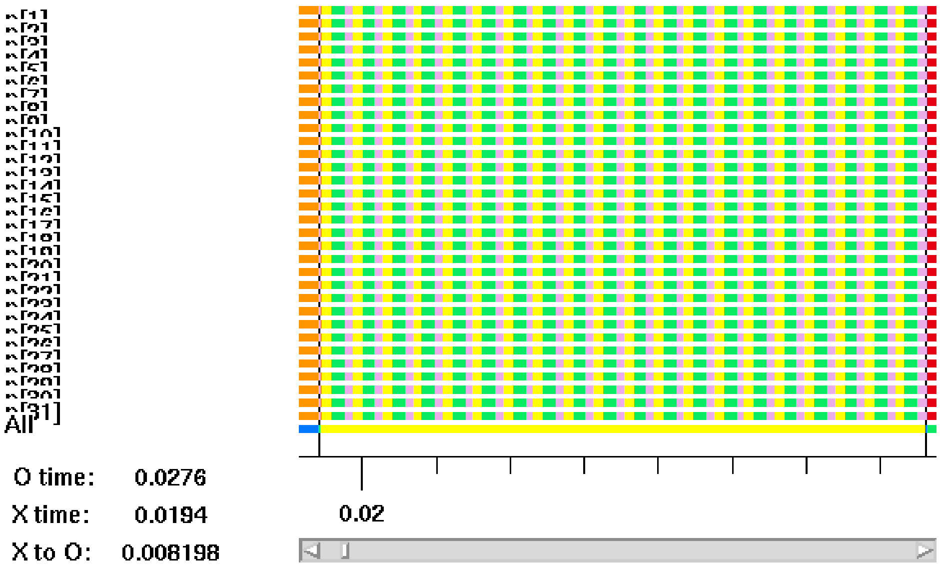

Figure 5.30: The `life` routine; predicted timing diagram for 32 processors, with computation speeded up by a factor of 10, communications left the same speed.

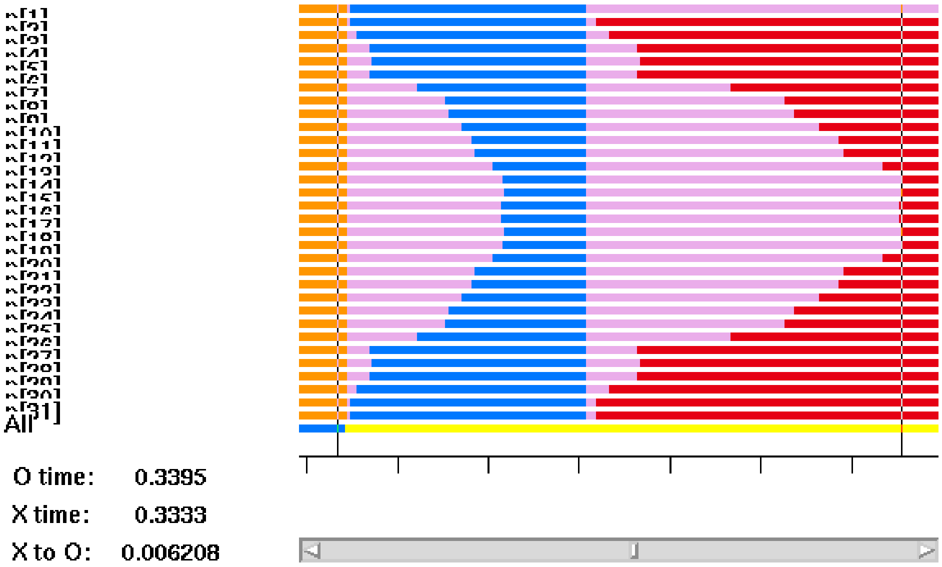

Figure 5.31: The `winnow` routine: predicted timing diagram for 32 processors. The first stages of computation have the appearance of the edge of the Mandelbrot set.

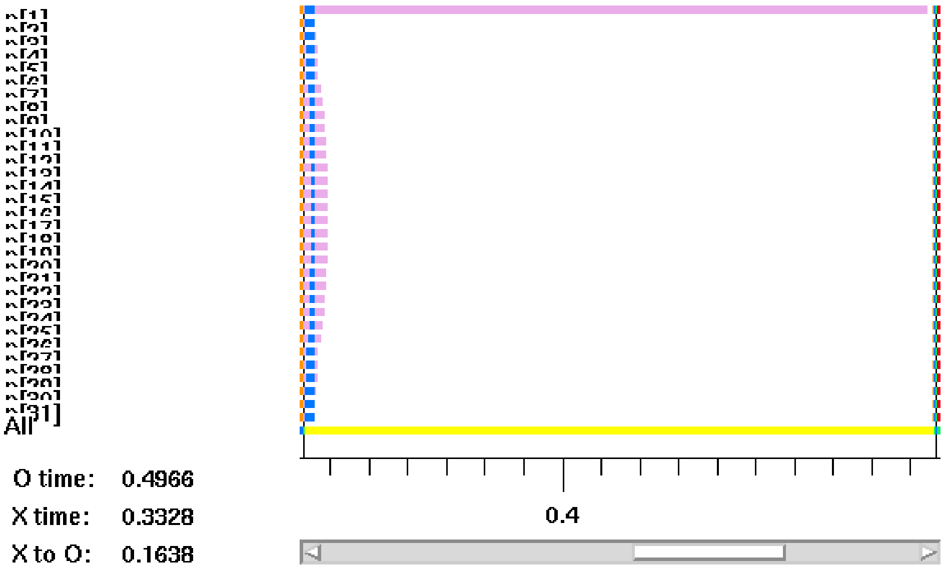

Figure 5.32: The `winnow` routine: predicted top level timing diagram for 32 processors.

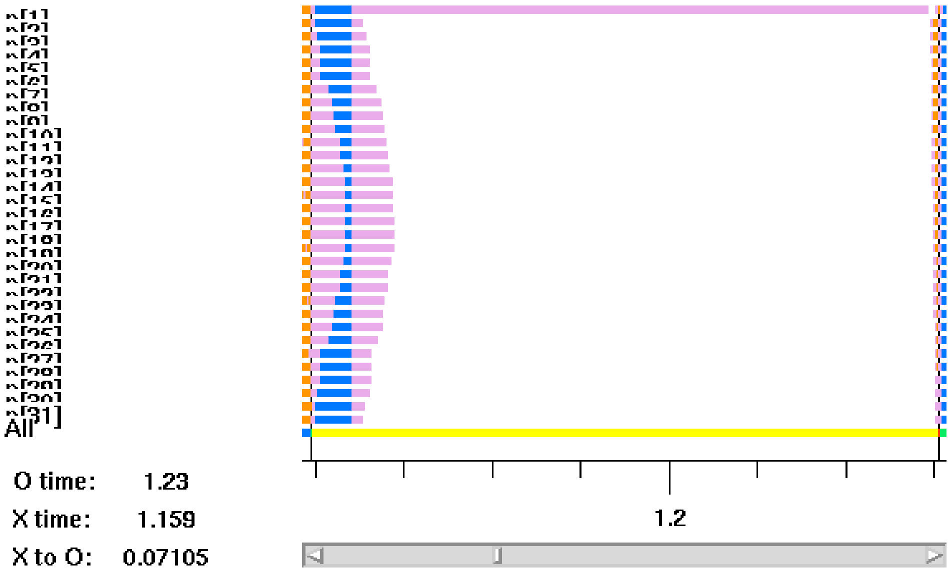

Figure 5.33: The `winnow` routine: measured top level timing diagram for 32 processors.

- It is as easy to use as normal profiling
- Predictions for any target architecture may be made on the development platform
- Predictions are based on actual data distributions
- Data dependent communications may be included
- Expected timing diagrams are produced, showing graphically where most time is spent
- Non deterministic communications are not supported

Reverse profiling offers a very quick and easy method of performance prediction for MPI programs. Unlike simulation techniques it builds directly upon the full and complete MPI libraries available now. It does not attempt to handle non-determinism but this is the area in which existing profilers and simulators produce the least believable results. It works with any MPI implementation which provides the standard profiling interface, so predictions may be performed in parallel.

The next chapter investigates the use of a discrete event simulation tool for performance prediction, to plug the non deterministic gap.

Bibliography

[1] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[2] F.W. Howell. Reverse profiling. In Innes Jelly and Ian Gorton, editors, *Software Engineering for Parallel and Distributed Systems : Proceedings of the First IFIP TC10 International Workshop on Parallel and Distributed Software Engineering*, pages 244–255. IFIP, Chapman and Hall, March 1996.

[3] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya. Machine characterisation based on an abstract high-level language machine. *IEEE Trans. on Comp.*, 38(12):1659–1679, December 1989.