Chapter 4

Simple Performance Estimates of the Cowichan Problems

The data sheets described in the previous chapter may be used as they are for manual estimates of performance and scalability. However this becomes impractical for all but the smallest programs, so this chapter looks at a way of feeding the datasheets into an equation and graph plotting program. This study was performed to investigate when such a minimalist approach to modelling may be applied.

Section 4.1 describes the basics of the approach; section 4.2 shows how data sheet results are integrated; section 4.3 describes the models of the Cowichan problems and section 4.4 discusses the results.

4.1 The models

Simple performance models of the MPI Cowichan suite were constructed using the GNUplot package. GNUplot turned out to be a powerful tool, providing functional composition to express hierarchy, as well as providing graphs.

The aim of this work was extremely rapid construction of models with gross assumptions to see how valuable such an approach can be, contrasted to more accurate (and time consuming) modelling.

In the simplest version, computation and communications costs were denoted by parameters $T_{compute}$ and T_{comms} (in section 4.2 the single communications parameter is replaced by the set of equations from a data sheet). Collective operations were modelled using simple combinations of the parameters, e.g.

$$log(N_{procs}) * T_{comms} * N_{data}$$

The computation and communication performance parameters for all models are specified in a GNUplot script file:

```
tcomms           = 10.0
tcompute         = 1.0
tbcast(p,d)      = tcomms*d*log(p)
tallreduce(p,d)  = tcomms*d*log(p)
talltoall(p,d)   = tcomms*d*log(p)
talltoallv(p,d)  = tcomms*d*log(p)
talltoallvi(p,d) = tcomms*d*p
tallgather(p,d)  = tcomms*d*log(p)
tallgatherv(p,d) = tcomms*d*log(p)
```

This gives the expected time for the communications functions `MPI_Bcast`, `MPI_Alltoall` etc. in terms of the data size **d** and the number of processors **p**. The parameters `tcomms` and `tcompute` give the expected times in microseconds to send an integer or perform a computation step.

Individual models include these top level parameters.

4.1.1 An example: the mandelbrot set

The model for the mandelbrot set example appears as:

```
# model of mandel performance
tmandelcalc      = tcompute * 8 * maxmandeliter
tmandel(p)       = ncols * (nrows/p) * tmandelcalc
```

`tmandelcalc` is the time required to compute a single pixel of the set, given here as eight compute steps per iteration multiplied by the maximum number of iterations. `tmandel(p)` is the estimated time to compute the set on **p** processors where each processor has an equal slice of the matrix to work on (i.e. a slice of size $ncols * (nrows/p)$).

Because these equations are included in a script file for GNUplot, a graph (figure 4.1) may be produced with the line:

```
plot tmandel(x)
```

This just models the computation to be done, but the communication must also be accounted for. This leads to a model such as:

```
# model of mandel performance (2)
tbroadcast(p)     = tbcast(p,8)
tgatherresults(p) = tgather(p,ncols*nrows/p)
tmandelcalc       = tcompute * 8 * maxmandeliter
```

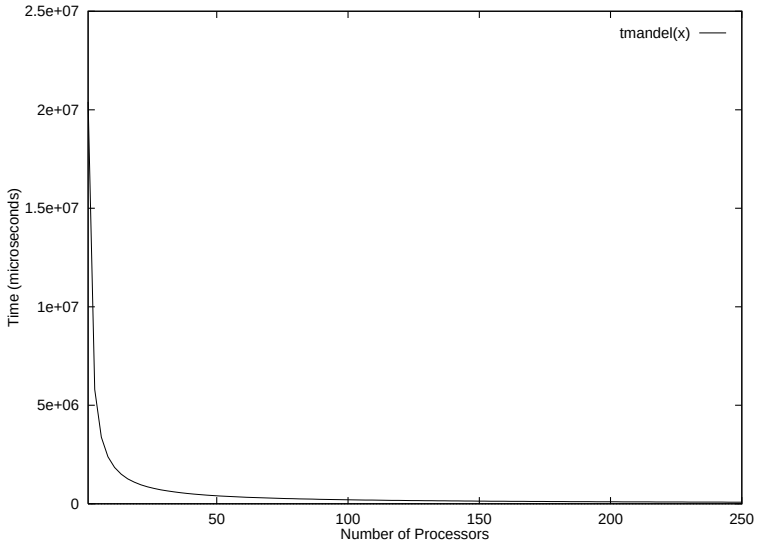

Figure 4.1: Simple plot of expected mandelbrot performance.

```
tmandelcomp(p)    = ncols * (nrows/p) * tmandelcalc
tmandel(p)        = tbroadcast(p) + tmandelcomp(p) + tgatherresults(p)
```

The components are shown in figure 4.2, with `tcomm` set to 1000.

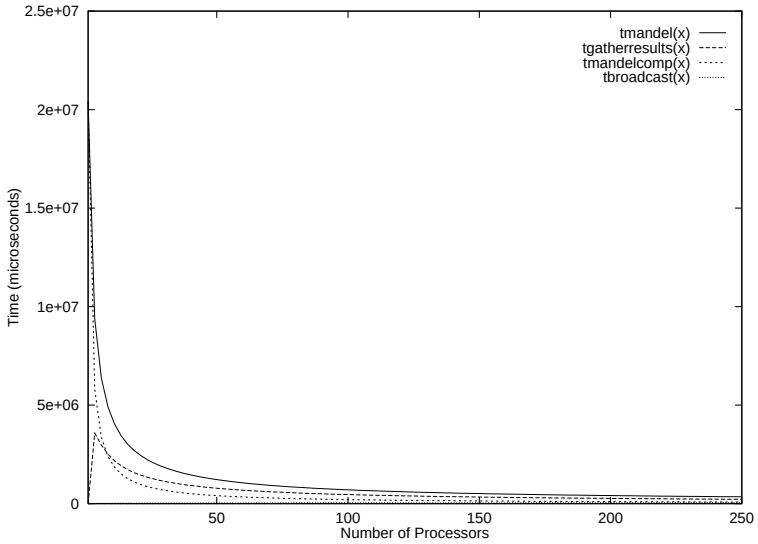

Figure 4.2: Time plot of expected mandelbrot performance with communications added.

Speedup curves may also be plotted (figure 4.3):

```
plot tmandel(1) / tmandel(x)
```

Figure 4.4 shows the speedup curve with the compute times stepped from 1.0 to 100.0 and the communications time left at 1000.0.

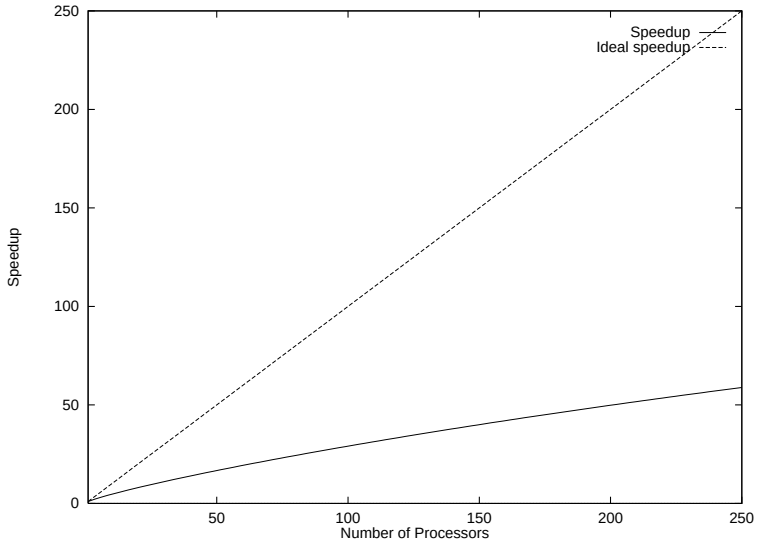

Figure 4.3: Speedup plot of expected mandelbrot performance with communications.

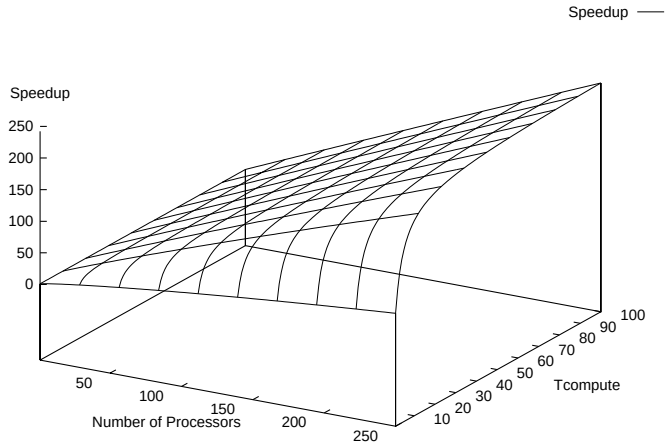

Figure 4.4: Speedup plot: `tcompute` varied from 1 to 100.

All of these graphs may be produced extremely rapidly from a basic model of a parallel program. Given particular values of `tcompute` and `tcomms` it is straightforward to determine whether a particular algorithm will scale well on a parallel machine. The difficulty is knowing what values to use for `tcompute` and `tcomms` for an architecture. The simplistic estimates of collective performance used above are not very believable, so a method using models derived from data sheets is described in the next section.

4.2 Using data sheet models

The MPI datasheet generator described in chapter 3 produces a set of equations characterising the MPI communications performance.

The measured model for the Cray T3D is shown below (times are in microseconds; `p` is the number of processors in the group and `d` is the message size).

```
# Cray T3D model
talltoall(p,d)  = 40*p + 0.3*p*d
tallsend(p,d)   = 40 + 0.1*p + 0.1*d
tgather(p,d)    = 200*log(p) + 0.0009*p*p*d
tallgather(p,d) = 40*p + 0.3*p*d
treduce(p,d)    = 300 + 2*p + 0.6*log(p)*d
tallreduce(p,d) = 300 + 6*p + 1*log(p)*d
tbcast(p,d)     = 100 + 2*p + 0.2*log(p) *d
tbarrier(p)     = 40
```

For comparison, the model for a network of 8 Sun SPARCstation 5 workstations connected using ethernet is:

```
# Network of Workstations model
talltoall(p,d)  = 2000 + 4000*p*p + 2*p*p*d
tallsend(d)     = 2000 + 1*d
tgather(p,d)    = 30000*log(p) + 4*log(p)*d
tallgather(p,d) = 8000*p + 4*p*p*d
treduce(p,d)    = 20000*log(p) + 6*log(p)*d
tallreduce(p,d) = 10000 + 500*p*p + 2*d*p*p
tbcast(p,d)     = 2000 + 700*p*p + 1*p*d
tbarrier(p)     = 9000*p
```

This may be incorporated into the performance models for programs, effectively instantiating measured values for `tcomm`. Figure 4.5 shows the expected

speedups using the Cray model for communications and varying `tcompute` from 0.01 to 1.0. Note that a *slowdown* is expected at low values of `tcompute`; figure 4.6 shows that the minimum time occurs at 70 processors. Figure 4.7 shows the expected speedup using the network of workstations performance model.

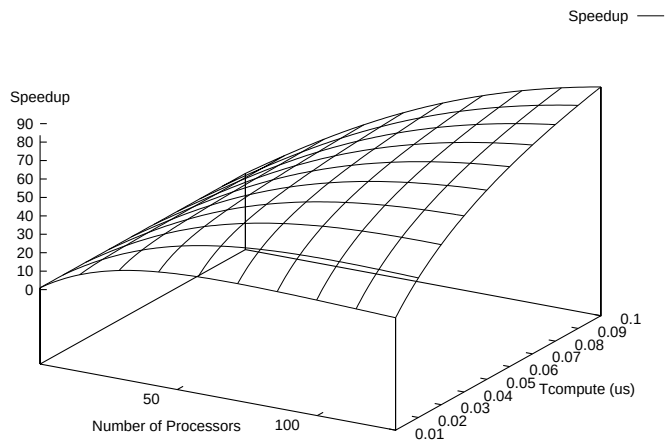

Figure 4.5: Speedup plot generated using datasheet model for Cray T3D performance.

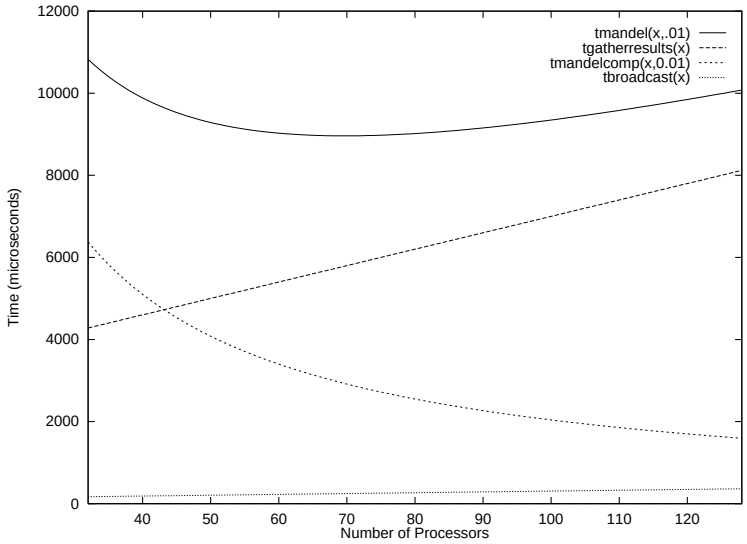

Figure 4.6: Closeup of figure 4.5 at `tcompute=0.01us`.

4.3 Modelling the Cowichan problems

This section describes models of all the Cowichan problems detailing what design information may be gleaned from using this design technique. The models are

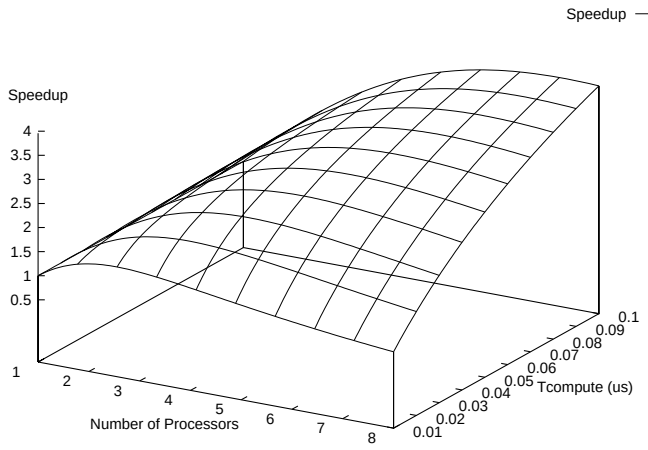

Figure 4.7: Speedup plot generated using datasheet model for Network of Workstations performance.

presented in GNUplot equation format, with an equation for the time of each phase of the program in terms of the number of processors **p** and the data size **d**.

4.3.1 Mandelbrot set generation (`mandel`)

Each of the points in the mandelbrot set may be computed independently, which makes this an "embarrassingly parallel" problem. The only difficulty for performance prediction is that the computation at any point is unpredictable, with any number of iterations in the central computation loop between 1 and the maximum number of iterations `MAX_MANDEL_ITERS`. The black regions inside the set require large amounts of computation, those far from the set require very little. The model given below includes the parameters **p** (the number of processors), **d** (number of rows in the matrix) and **c** (the compute step time, set at $1us$).

```
# model of mandel performance
tmandelcalc(c)     = c * 8 * maxmandeliter
tmandelcomp(p,d,c) = d * (d/p) * tmandelcalc(c)
tmandel(p,d,c)     = tmandelcomp(p,d,c) + tbarrier(p)
```

Figure 4.8 shows the measured and predicted speedup on a network of workstations. The shapes of the curves match very closely; the predicted slowdown with small data sizes does actually occur, and the best speedup occurs at eight processors. Using sixteen processors yields no additional speedup as expected. Figure 4.9 shows the measured and predicted times which also match up well.

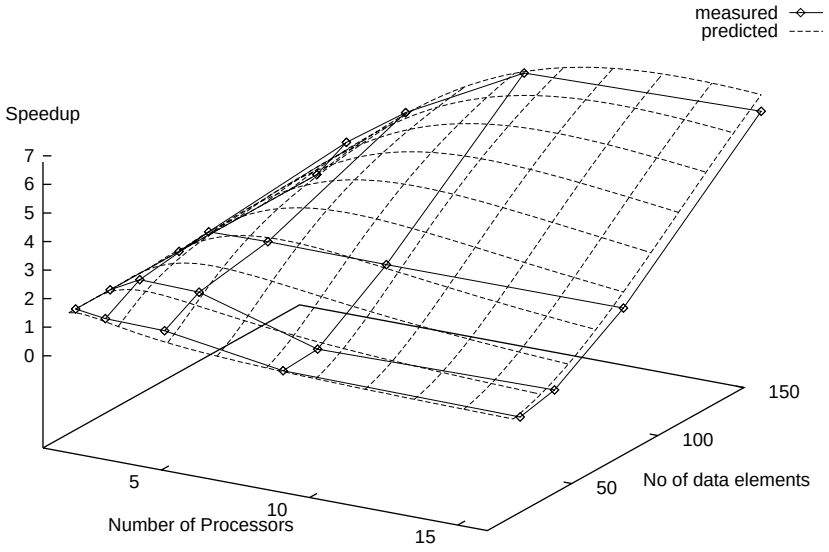

Figure 4.8: Mandel measured and predicted speedup on a network of workstations.

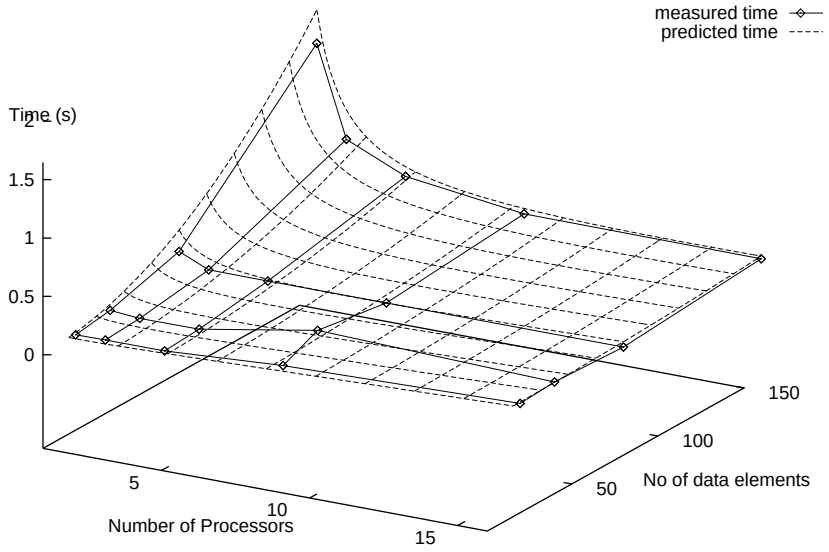

Figure 4.9: Mandel measured and predicted times on a network of workstations.

On the Cray T3D, with the compute time step left at $1us$ the measured and predicted times are shown in figure 4.10, with the speedup in figure 4.11. The speedup prediction is overly optimistic at low data sizes and overly pessimistic for high data sizes. The reason for this was determined by looking at the detailed timing diagrams. The matrix is distributed by entire *rows*. A 20×20 matrix size only makes use of 20 out of the available 32 processors, leaving the other 12 idle. This quantisation effect was not included into the simple model, resulting in the overly optimistic prediction. The second discrepancy was caused by missing an initial startup computation cost from the model - the time to allocate the memory for the matrix. This data dependent startup cost actually *improves* speedup with larger data sizes, since it has more impact on the single processor timing than on the multiple processor timing. This discrepancy was not apparent for the network of workstations comparison because communications time was dominant in that case.

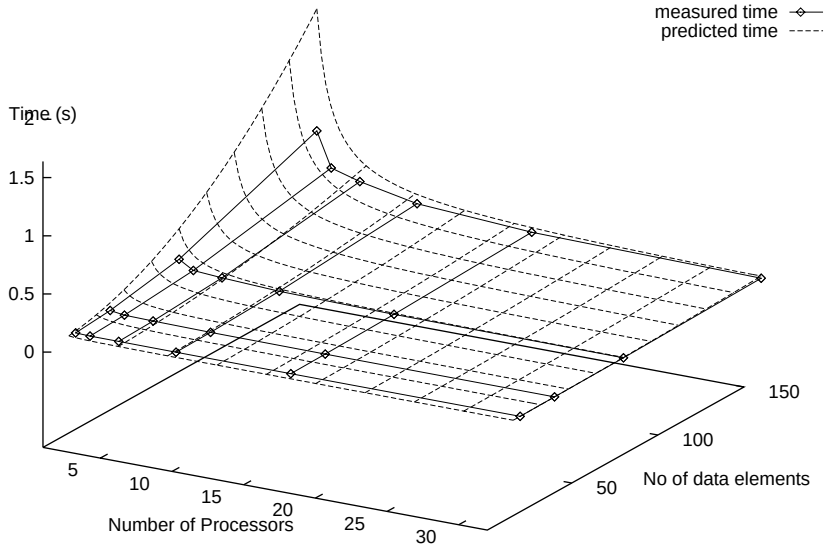

Figure 4.10: Mandel measured and predicted times on the Cray T3D.

4.3.2 Random matrix generation (`randmat`)

The matrix is divided equally among the processors and each computes the random numbers within its section. The overhead with respect to the sequential algorithm is that the initial seed must be computed for each process before it can start generating. This initial seed calculation requires a time proportional

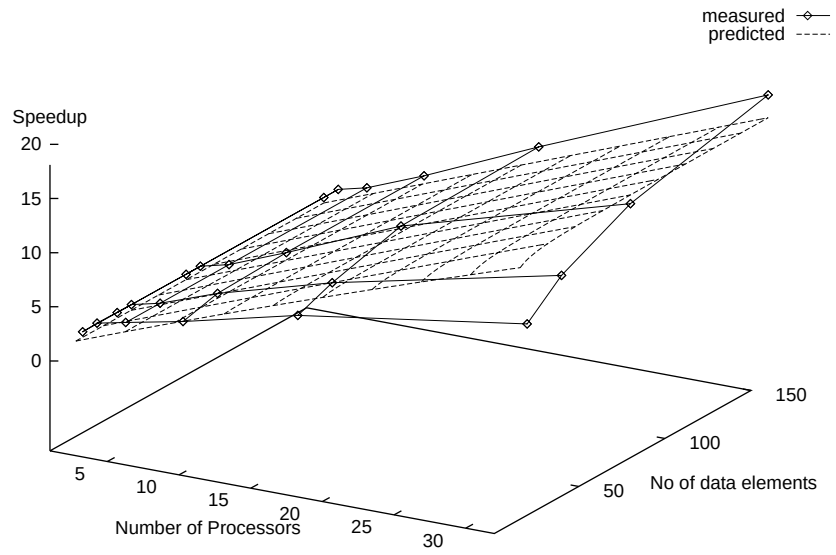

Figure 4.11: Mandel measured and predicted speedup on the Cray T3D.

to the logarithm of the number of matrix elements, so takes slightly longer for processors at the bottom of the matrix than for those at the top.

```
# model of randmat performance
tjrandom(i,p,d,c) = log(i*d*d/p) * c * 8
trandmat(p,d,c)   = c * d * d/p * 10
ttotal(p,d,c)     = tjrandom(p,p,d,c) + trandmat(p,d,c) + tbarrier(p)
```

Figure 4.12 shows the measured and predicted speedup on a network of workstations. All the important aspects of the performance are predicted correctly; the slowdown above four processors with the maximum data size, and the slowdown with more than one processor at the minimum data size. Figure 4.13 shows the measured and predicted times. For small data sizes, the time is dominated by the barrier time. The computation time has been consistently underestimated by a factor of two.

On the Cray, figure 4.14 shows the times (on a logarithmic axis). The prediction is an underestimate for small numbers of data elements but converges for larger matrix sizes, indicating that a constant overhead of the order of $500us$ has been left out of the model.

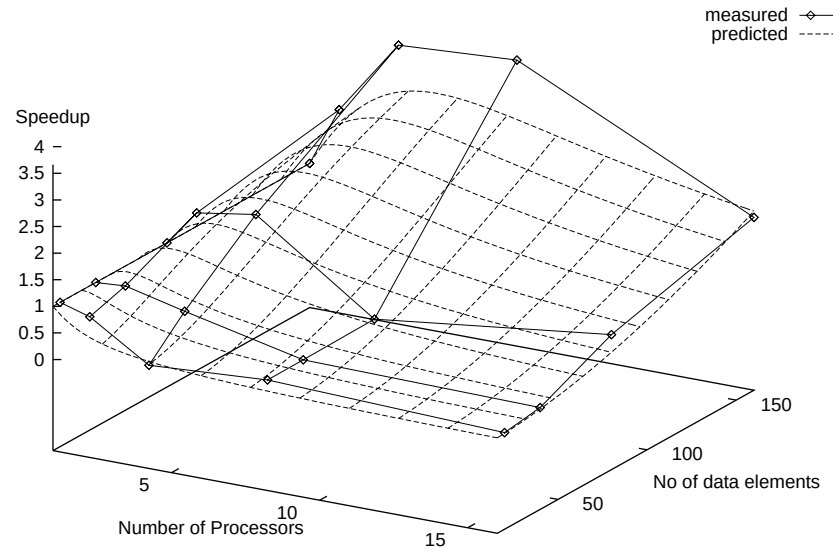

Figure 4.12: Randmat measured and predicted speedup on a network of workstations.

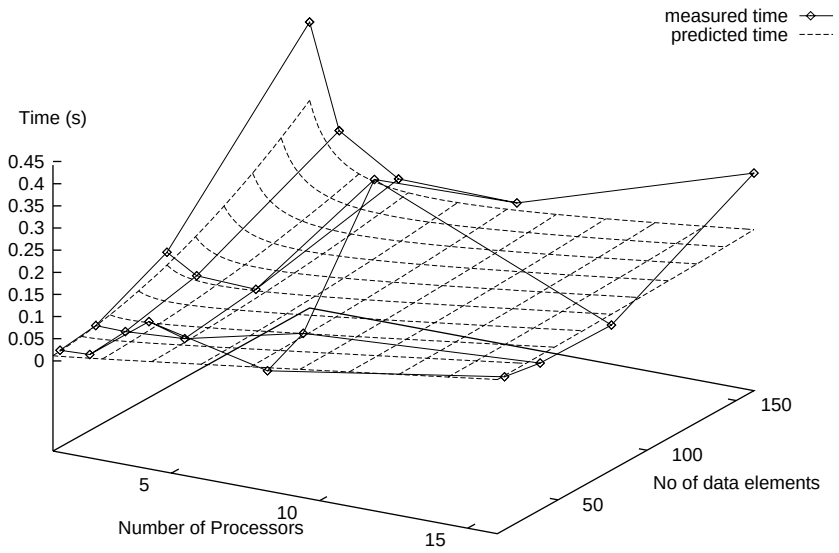

Figure 4.13: Randmat measured and predicted times on a network of workstations.

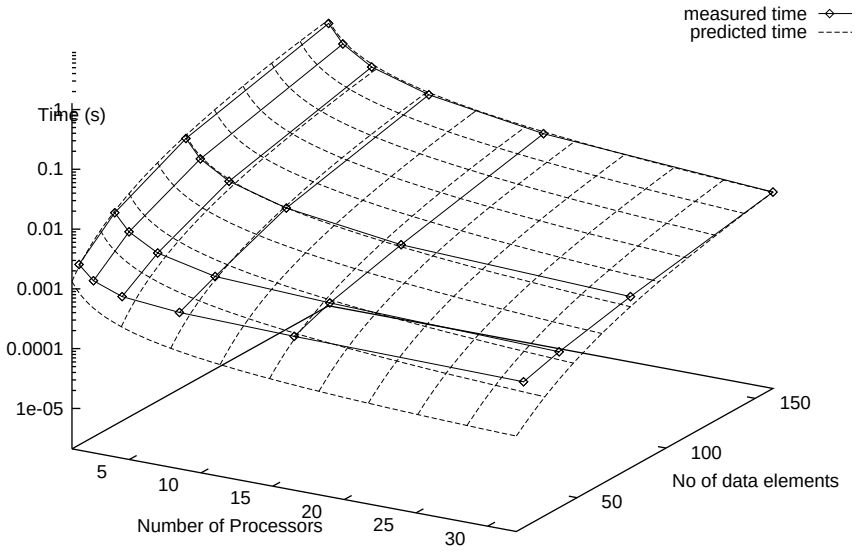

Figure 4.14: Randmat measured and predicted times on the T3D.

4.3.3 Perfect shuffle (`half`)

This module is heavy on communications as half of the matrix must be sent at each shuffle step. The model takes into account the time to build the complex MPI message data type required to complete the shuffle with one communications call.

```
# model of half performance
tcomputedtypes(p,d,c) = d * (d/p) * c * 2
tlocalshuffle(p,d,c)  = d * (d/p) * c * 3
ttotal(p,d,c)         = tcomputedtypes(p,d,c) + \
                        talltoallvi(p, (d/p)/p) + \
                        tlocalshuffle(p,d,c)
```

Figure 4.15 shows the measured and predicted speedup on a network of workstations. The prediction has yielded the useful information that a maximum speedup of two can be expected across eight processors. Figure 4.16 shows the measured and predicted times. These agree for all but the one point at the maximum number of processors and data.

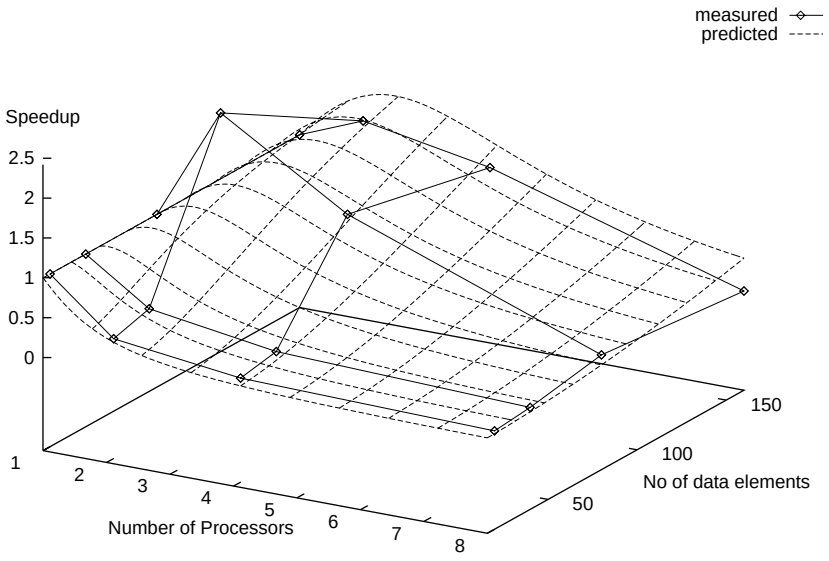

Figure 4.15: Shuffle (Half) measured and predicted speedup on a network of workstations.

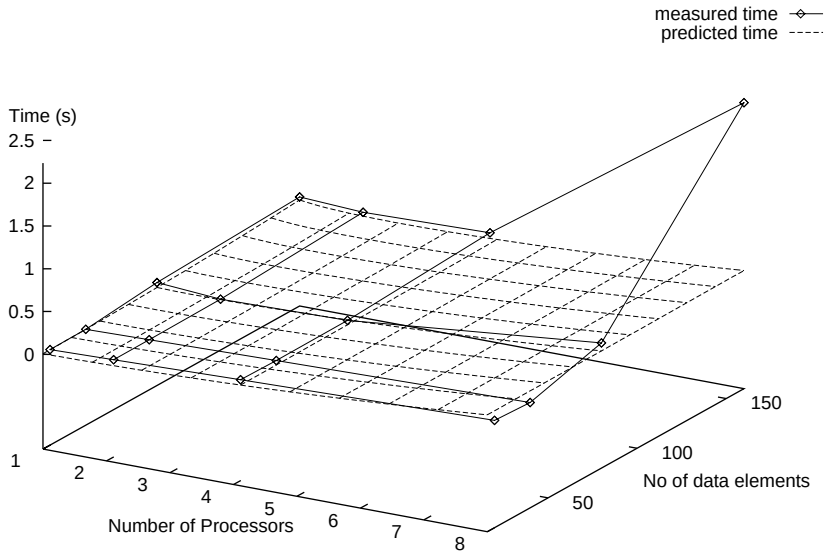

Figure 4.16: Shuffle (Half) measured and predicted times on a network of workstations.

4.3.4 The game of life (`life`)

This has a boundary swap followed by the local computation. The boundary swap consists of two sends followed by two receives.

```
# model of life performance
tboundaryswap(d)    = 2*tsend(d) + 2*trecv(d)
tliferow(d,c)       = c * d
tsublife(p,d,c)     = (d/p) * tliferow(d,c) + \
                      d * (d/p) * c
titer(p,d,c)        = tboundaryswap(d) + tsublife(p,d,c)
ttotal(p,d,c)       = nlifeiters * titer(p,d,c) + tbarrier(p)
```

Figure 4.17 shows the measured and predicted times on a network of workstations. As predicted, the times are dominated by communications, so no speedup is obtained. Figure 4.18 shows the measured and predicted speedups.

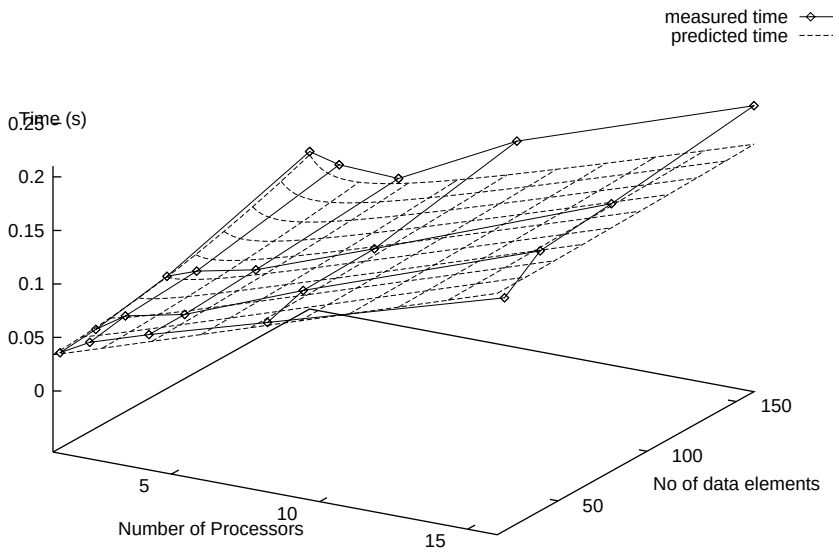

Figure 4.17: Life measured and predicted times on a network of workstations.

On the Cray, the measured and predicted times are shown in figure 4.19. The predictions are good for small numbers of processors, but out by a factor of three for 32 processors, because of quantization effects.

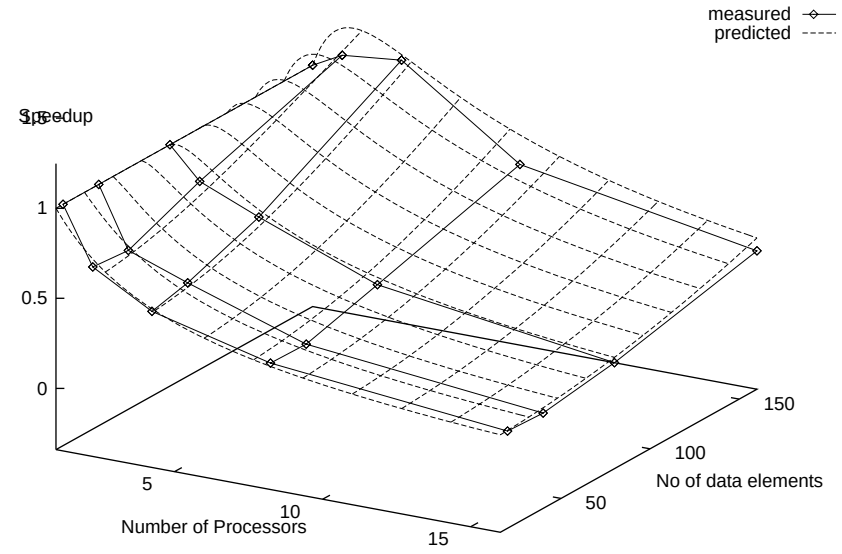

Figure 4.18: Life measured and predicted speedup on a network of workstations.

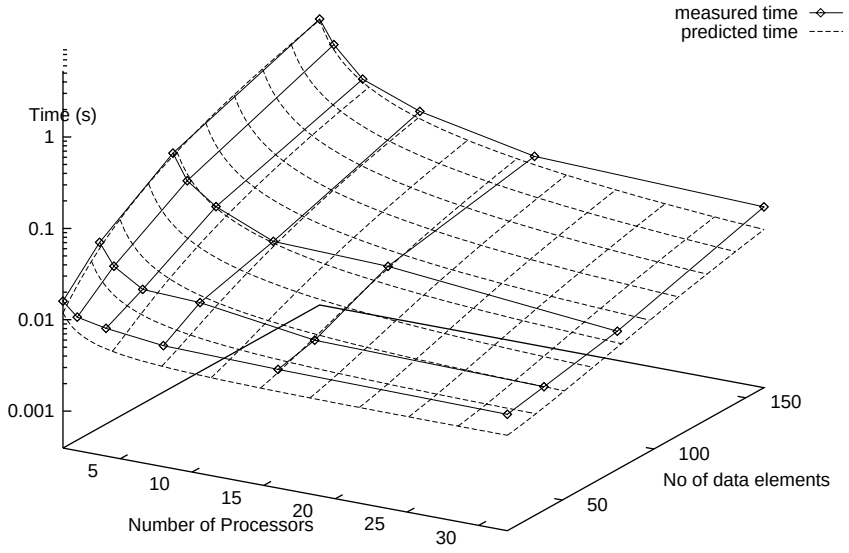

Figure 4.19: Life measured and predicted times on the T3D.

4.3.5 Image thresholding (`thresh`)

This model includes the local and global histogram computations time, the time to compute the threshold value and the time to generate the mask image.

```
# model of thresh performance
tglobalminmax(p,d,c) = d * (d/p) * c + tallreduce(p,1)
tlocalhist(p,d,c)    = c * d * d / p
tglobalhist(p)       = tallreduce( p, maxval )
tthresh(p,c)         = c * fraction * maxval
tmask(p,d,c)         = c * d * d/p

ttotal(p,d,c)        = tglobalminmax(p,d,c) + tlocalhist(p,d,c) +\
                       tglobalhist(p) + tthresh(p,c) + tmask(p,d,c) +\
                       tbarrier(p)
```

Figure 4.20 shows the predicted and measured times for a network of workstations, and figure 4.21 shows the predicted and measured speedups. This application shows an expected and measured slowdown on networks of workstations.

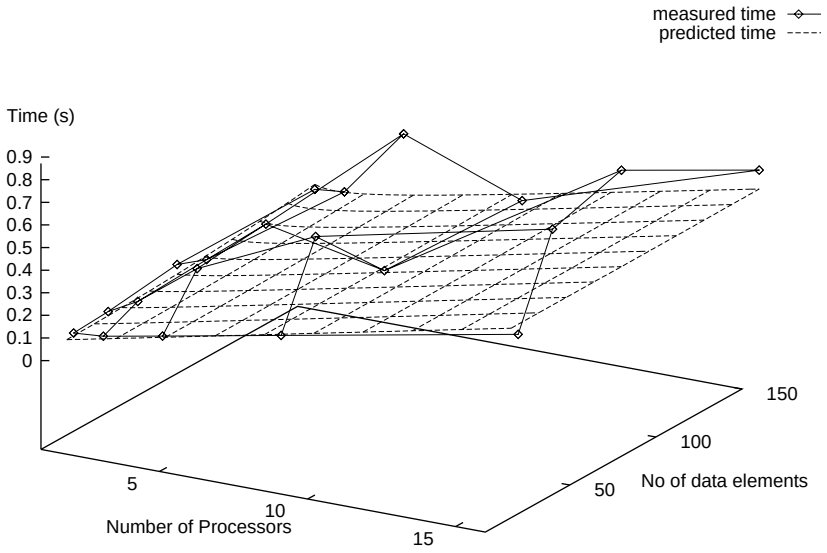

Figure 4.20: Thresh measured and predicted times on a network of workstations.

Figure 4.22 shows the measured and predicted times on the T3D. The times are underestimated by a factor of two; figure 4.23 shows that the speedup has been reasonably estimated.

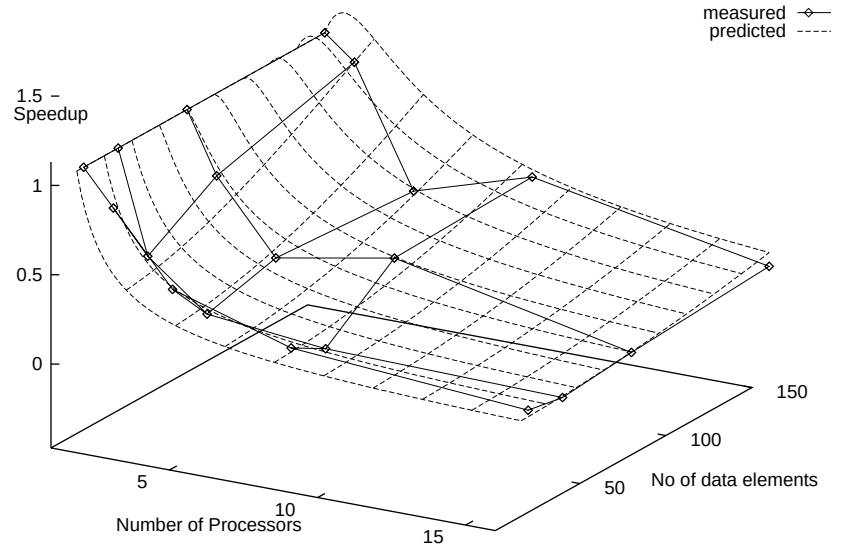

Figure 4.21: Thresh measured and predicted speedups on a network of workstations.

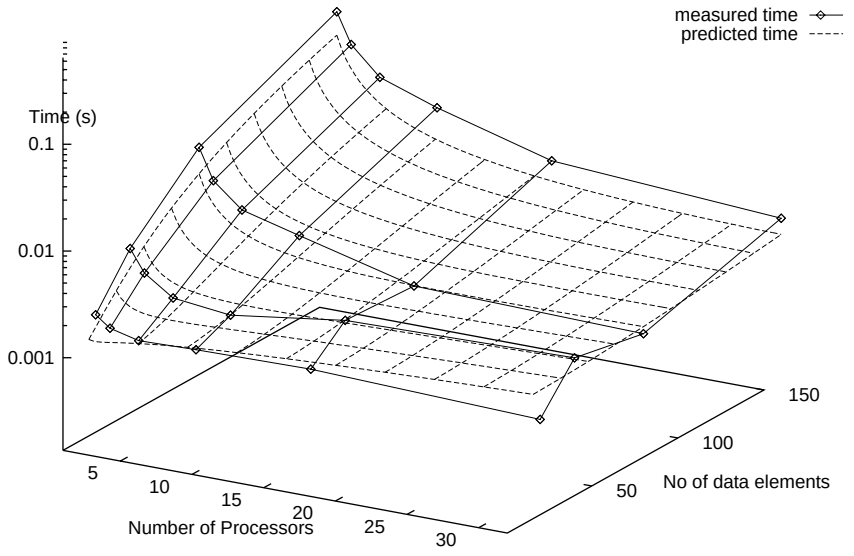

Figure 4.22: Thresh measured and predicted times on the T3D.

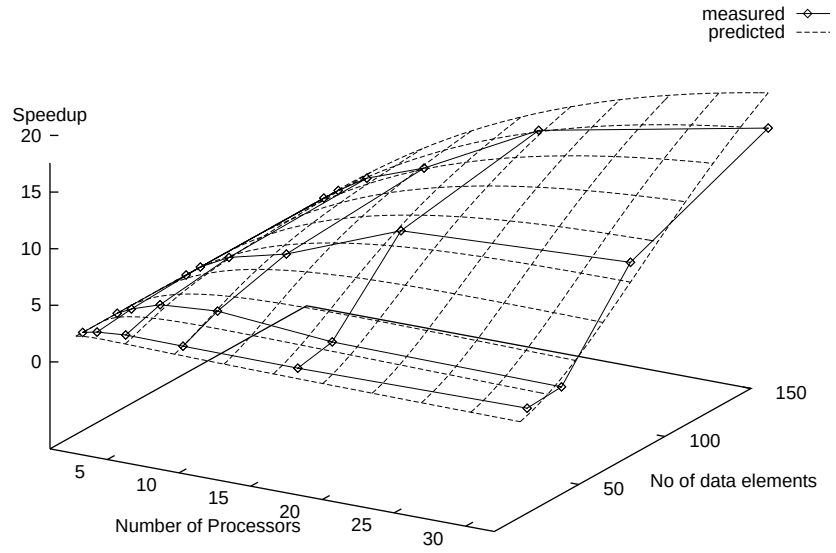

Figure 4.23: Thresh measured and predicted speedups on the T3D.

4.3.6 Outer product (`outer`)

This is interesting because of the possibility of cutting down the amount of computation by communicating results from the lower diagonal of the matrix to the upper diagonal.

```
tdist(c)              = 50*c
tmklocal(p,d)         = tallgatherv(p,d)
tcreate(p,d,c)        = d * tdist(c)
tfillmatrix(p,d,c)    = d * (d / p) * tdist(c)
tdiagfill(p,d,c)      = d * tdist(c)
ttotal(p,d,c) = tmklocal(p,npoints/p) + tcreate(p,d,c) +\
                tfillmatrix(p,d,c) + tallreduce(p,1) +\
                tdiagfill(p,d,c) + tbarrier(p)
```

Figure 4.24 compares predicted and measured speedups on the T3D, with the compute time parameter varied from $0.05us$ to $0.5us$. The general shape of the speedup curve is not changed by the order of magnitude change in compute time, indicating that on the T3D the algorithm is not dominated by communications.

On the network of workstations, the times are shown in figure 4.25. The time is out by at most a factor of three. Figure 4.26 shows how sensitive this algorithm is to the compute step value; at $0.1us$ the speedup prediction is accurate; at $1us$ the prediction is optimistic.

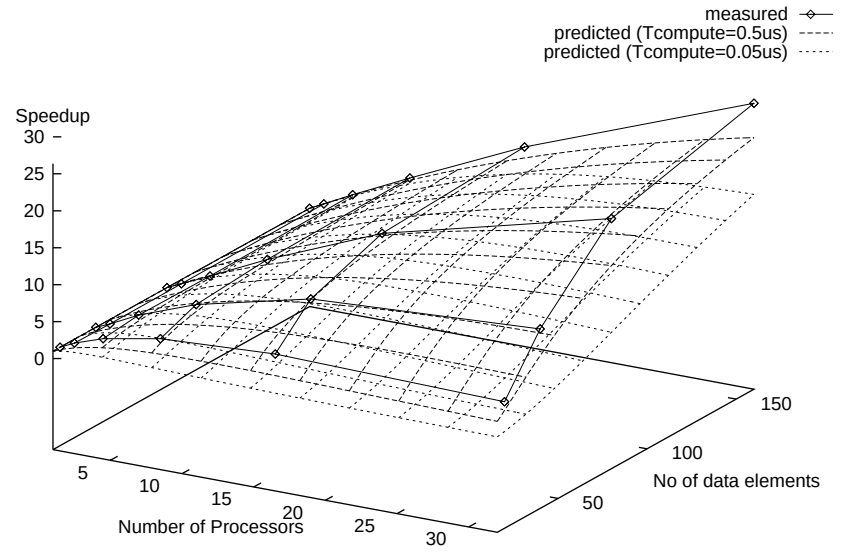

Figure 4.24: Outer measured and predicted speedups on the T3D.

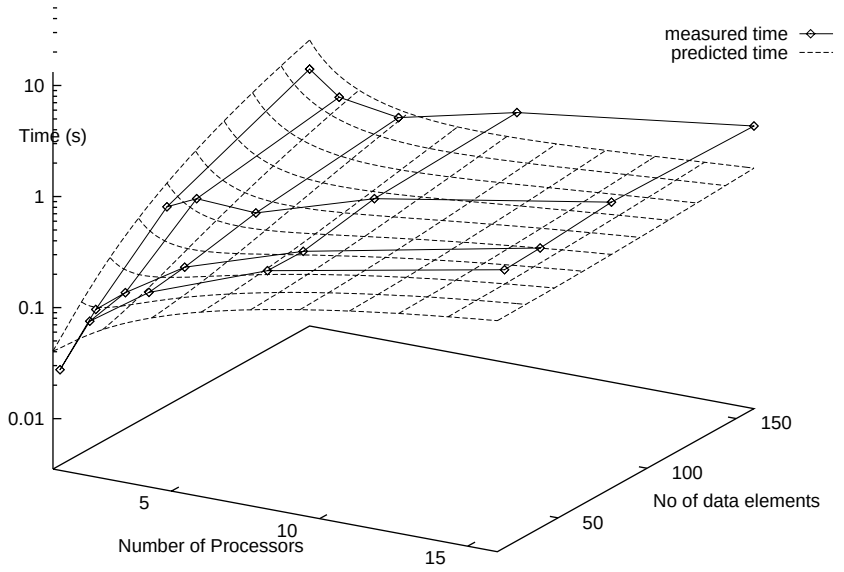

Figure 4.25: Outer measured and predicted times on a network of workstations.

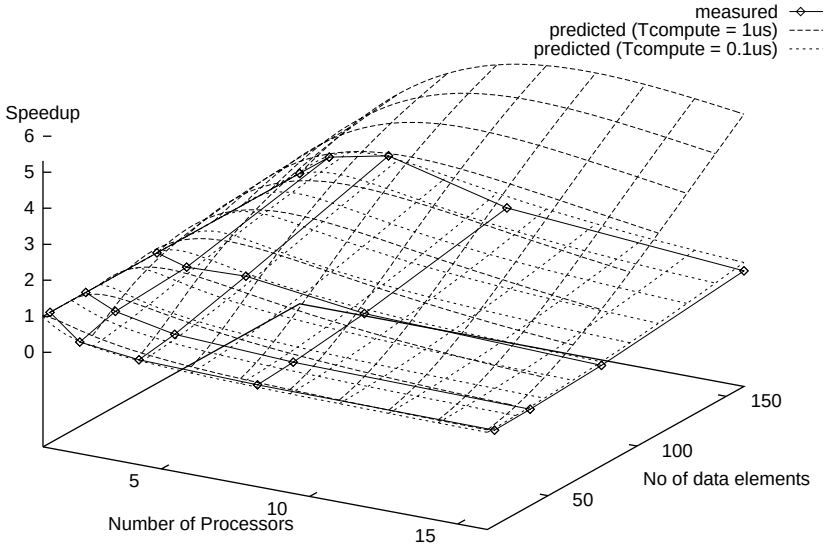

Figure 4.26: Outer measured and predicted speedups on a network of workstations.

4.3.7 Elastic net simulation (`elastic`)

This is a complex algorithm and the model reflects this. The parameters are the number of iterations and the number of cities. The total time is made up of the time required to copy all the city locations to all processes (`tmklocal`), an initialisation step `tinit` and the time for all the iterations. Each iteration involves a nearest neighbour exchange (`tneigh`), the time to compute the influence of all the cities on the net `tcity` and the time to apply the resultant force to update the net `tmove`.

```
nnet(d)          = d * 2.5
ncities(d)       = d
tmklocal(p,d)    = tallgather(p,d / p)
tinit(p,d,c)     = c*(d + nnet(d)/p)
tneigh(p,d,c)    = (nnet(d)/p)*c + 2*tsend(1) + 2*trecv(1)
tmove(p,d,c)     = c * (nnet(d) / p)
tcity(p,d,c)     = nnet(d) * c + \
        ncities(d) * ( \
                c * (d * nnet(d)/p) +\
                tallreduce(p,1) +\
                c * nnet(d)/p \
```

```
        )
titer(p,d,c)     = tneigh(p,d) + tcity(p,d) + tmove(p,d)
ttotal(p,d,c,i)  = tmklocal(p,d) + tinit(p,d) + i*titer(p,d) +\
                   tbarrier(p)
```

On the network of workstations, the times are shown in figure 4.27. The prediction is very good apart from the figures for two workstations. The measured figures for these were investigated by examining the timing diagrams. The culprit was found to be a random network delay which was delaying the inner loop communication by over 2 seconds, an occasional hazard of using a shared ethernet. The prediction is so good for all other points because the time is totally dominated by communications which are well predicted.

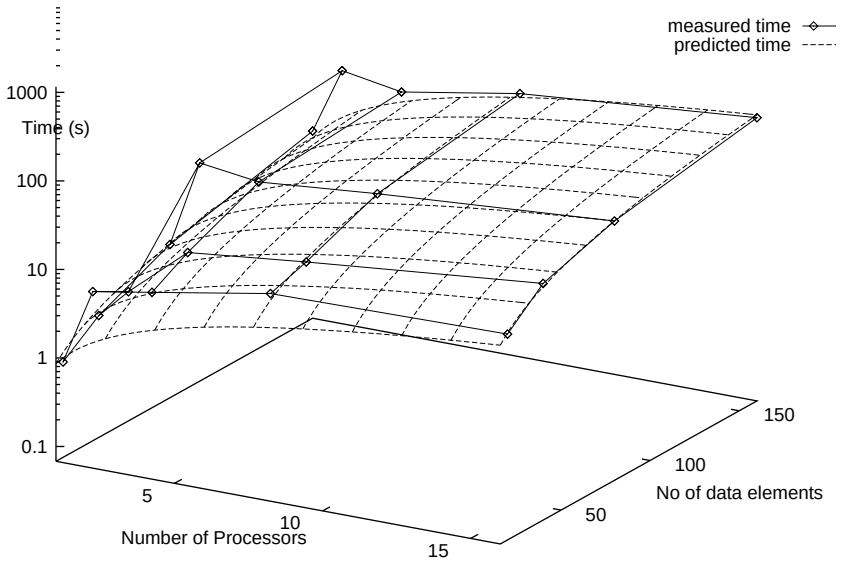

Figure 4.27: Elastic measured and predicted times on a network of workstations.

On the Cray, the initial prediction was not so good. Figure 4.28 shows that the prediction using the standard value of $0.5us$ for the compute step value led to a factor of 10 over-estimate in overall times. A value of $0.05us$ produced an accurate fit, indicating that this problem uses small enough data sizes to fit into the cache, leading to the order of magnitude better than expected performance.

4.3.8 Invasion percolation (`invperc`)

In the model, `titer` is the individual iteration time, `tenqueue` is the time needed to add a point to the local queue, `tglobalhead` is the time to determine the

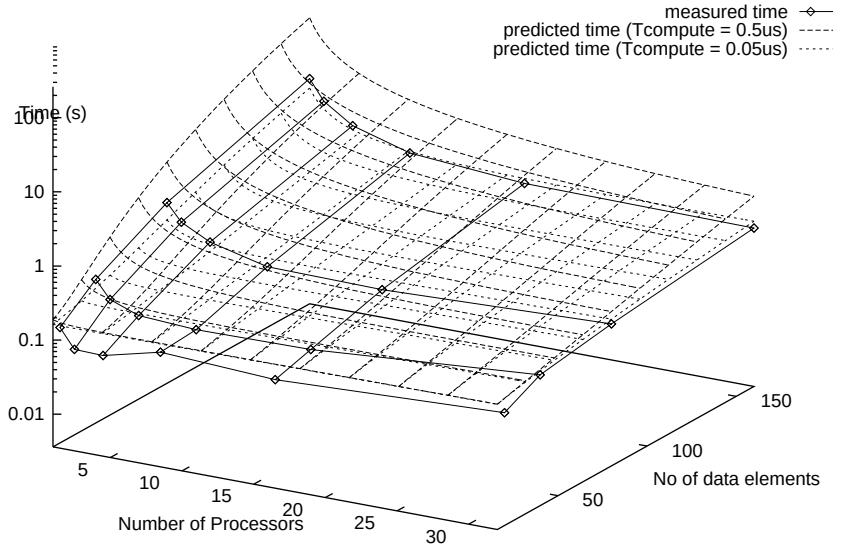

Figure 4.28: Elastic measured and predicted times on the Cray T3D.

overall head of the queue. The possible speedup comes from having a smaller queue insertion time on a parallel machine.

```
# model of invperc performance
maxqlen(d)        = d*d * fraction;
qlen(p,d)         = maxqlen(d) / (p*10);
tglobalhead(p)    = tallreduce(p,4)
tenqueue(p,d,c)   = c * qlen(p,d) * 4;
titer(p,d,c)      = tglobalhead(p) + tenqueue(p,d,c);
ttotal(p,d,c)     = titer(p,d,c) * maxqlen(d);
```

Figure 4.29 shows the measured and predicted speedups on the Cray T3D. The amount of speedup available is very sensitive to the compute step time; the initial estimate was a factor of four overly optimistic for large data sizes. This error was caused by having to guess the average queue length in the model; the times for this program are highly data dependent. Using a lower value for the compute time ($0.05us$) produced the lower bound curve in the figure.

Figure 4.30 shows the corresponding speedup curves for a network of workstations. The severe slowdown for this application is correctly predicted.

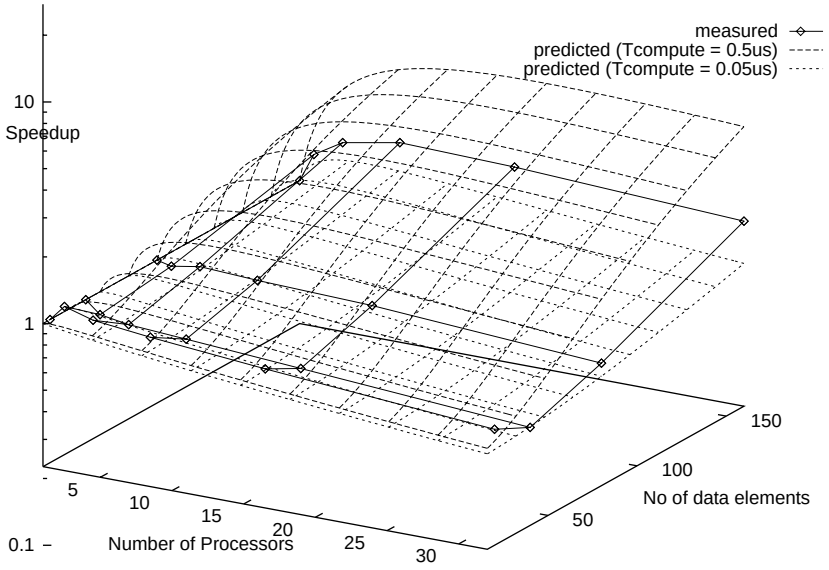

Figure 4.29: Invasion percolation measured and predicted speedups on the Cray T3D.

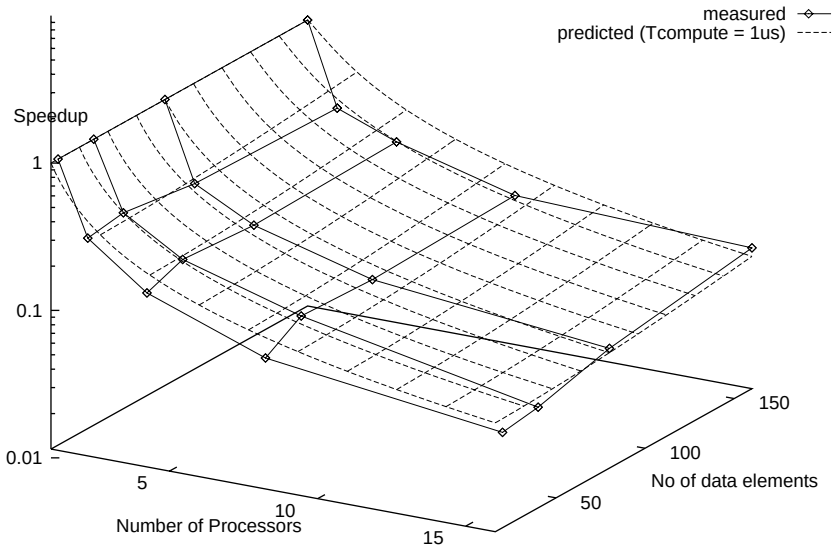

Figure 4.30: Invasion percolation measured and predicted speedups on a network of workstations.

4.3.9 Vector product (`product`)

The algorithm for the vector-matrix product first ensures that each process has its own copy of the entire vector, then each process may continue independently.

```
# model of product performance
tmklocal(p,d) = tallgatherv(p,(d/p))
tcalc(p,d,c)  = d * (d/p) * 2 * c
ttotal(p,d,c) = tmklocal(p,d*2.5) + tcalc(p,d*2.5,c) + tbarrier(p)
```

Figure 4.31 shows the measured and predicted times on a network of workstations. The times are dominated by communications for this problem, so adding more processors slows things down. The predictions are within a factor of two throughout; this discrepancy was tracked down to an underestimate in the `allgather` times caused by the curve fit in the MPI model. The corresponding times for the Cray T3D are shown in figure 4.32. This is a very good fit, with times dominated by computation.

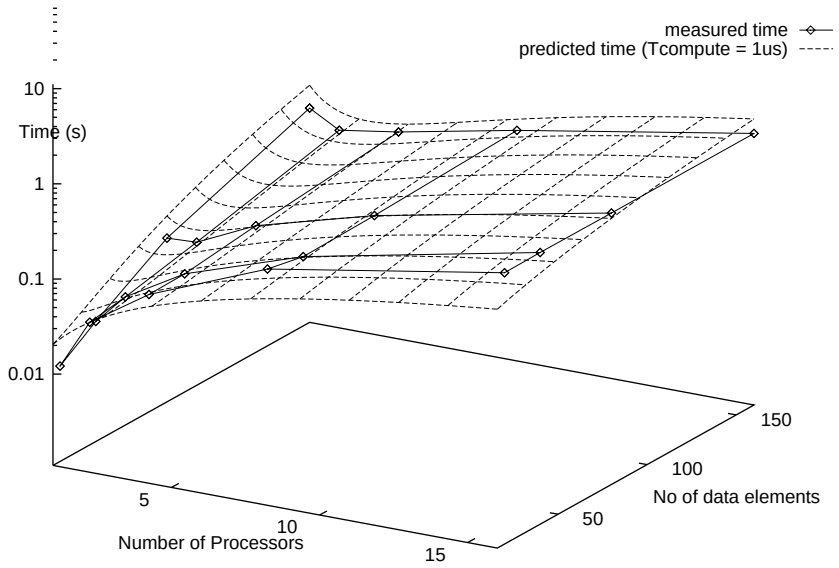

Figure 4.31: Vector product measured and predicted times on a network of workstations.

4.3.10 Successive over-relaxation (`sor`)

This is an iterative algorithm, so it is not possible to predict the convergence rate. It is however possible to put upper bounds on the convergence (i.e. `sormaxiters`).

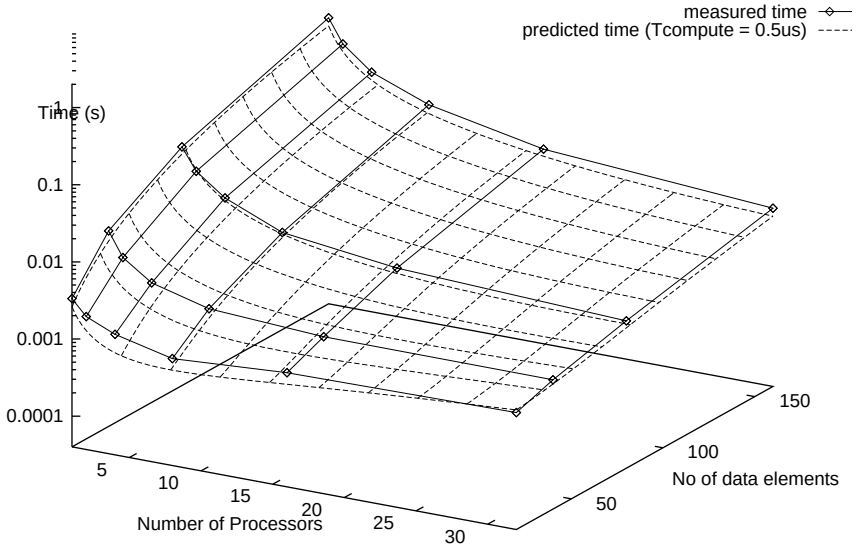

Figure 4.32: Vector product measured and predicted times on the Cray T3D.

```
# model of SOR performance
tmklocal(p,d) = tallgatherv(p,d)
titer(p,d,c)  = (d/p) * (
                        d * c * 2 +
                        c * 8) +
                tmklocal(p,d/p) +
                tallreduce(p,1)
ttotal(p,d,c) = tmklocal(p,d*2.5/p) * 2 +
                sormaxiters * titer(p,d*2.5,c)
```

Figure 4.33 shows the measured and predicted times on the Cray T3D, a very good fit. The fit for network of workstations is less good (figure 4.34) as the underlying model for collective performance is less predictable. However it does provide the useful design information that no speedup is expected.

4.3.11 Gaussian elimination (gauss)

The matrix to solve is distributed blockwise by row. The algorithm used to implement Gaussian elimination involves a single pass through the rows of the matrix in which all processes participate. The process which stores the current row then computes a pivot and broadcasts it. All processes apply this pivot row to their local section of the matrix. Figure 4.37 illustrates the performance model.

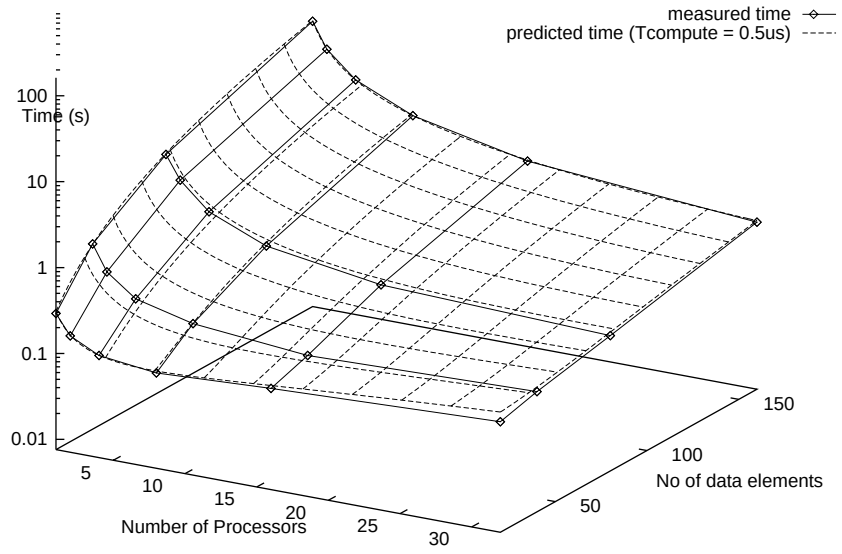

Figure 4.33: SOR measured and predicted times on the Cray T3D.

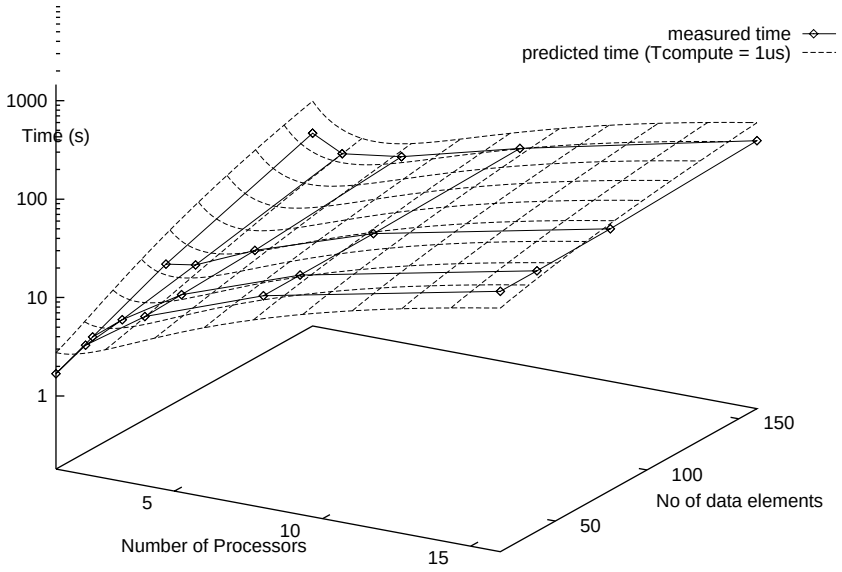

Figure 4.34: SOR measured and predicted times on a network of workstations.

```
# model of gauss performance
tcopy(d,c)           = d*c
tpivotcompute(d,c,i) = (d-i) * 2 * c + 4 * c
ttransform(p,d,c)    = d * (d/p) * c * 5
tgauss(p,d,c)        = d * (tpivotcompute(d,c,d/2) +
                            tbcast(p,d) +
                            ttransform(p,d,c) )
ttotal(p,d,c)        = tcopy(d*2.5,c) + tgauss(p,d*2.5,c)
```

Figure 4.35 shows the measured and predicted speedups on the Cray T3D. The actual speedup is worse than expected because the compute times were overestimated by a factor of five. The compute times were also overestimated on the network of workstations (figure 4.36).

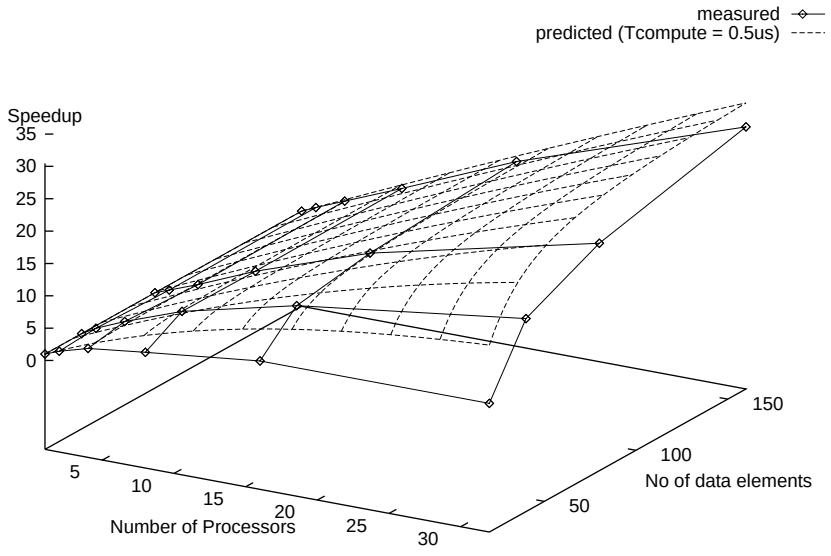

Figure 4.35: Gauss measured and predicted speedups on the Cray T3D.

4.3.12 Point normalisation (`norm`)

The parallel implementation of point normalisation involves a global reduction followed by independent computation phases. For design purposes it is necessary to know if the time for the global reduction swamps the total time for computation.

```
# model of norm performance
```

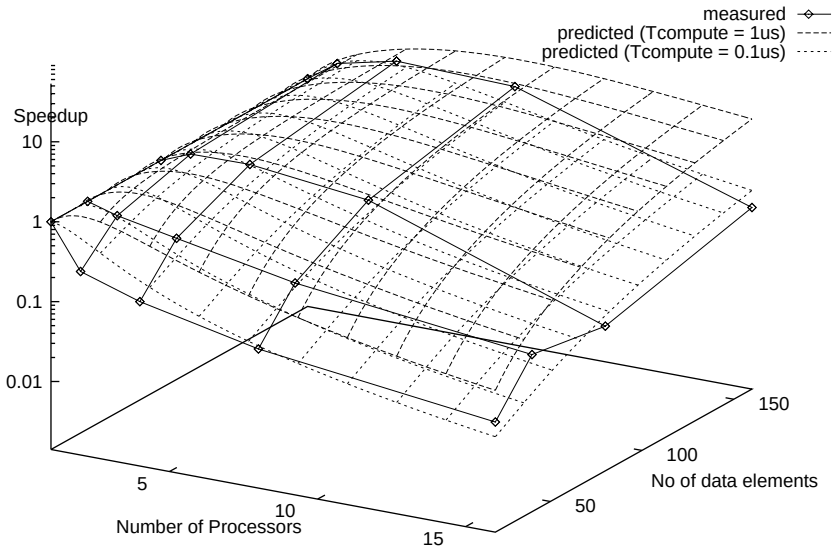

Figure 4.36: Gauss measured and predicted speedups on a network of workstations.

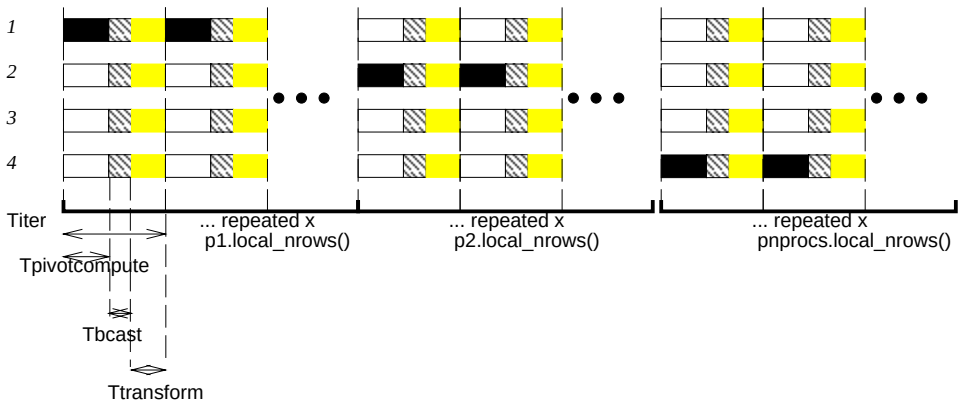

Figure 4.37: The Gauss performance model.

```
tlocalredptvector(d,c)  = d * c
tglobredptvector(p,d,c) = tlocalredptvector(d/p,c) + \
                          4*tallreduce(p,1)
tnorm(p,d,c)            = (d/p) * 8 * c
ttotal(p,d,c)           = tglobredptvector(p,d,c) +
                          tnorm(p,d,c) + tbarrier(p)
```

Figure 4.38 shows the predicted and measured times on a network of workstations. The algorithm runs more slowly as more processors are added, as predicted. However the actual times for the single processor runs are faster than predicted. This is because no communication calls are actually made, but the simple model doesn't include this special case. The prediction for the Cray T3D is a consistent factor of two pessimistic (figure 4.39), apart from the single processor case. This is because the model for the `allreduce` time (which dominates) is not accurate for the very small amounts of data used here.

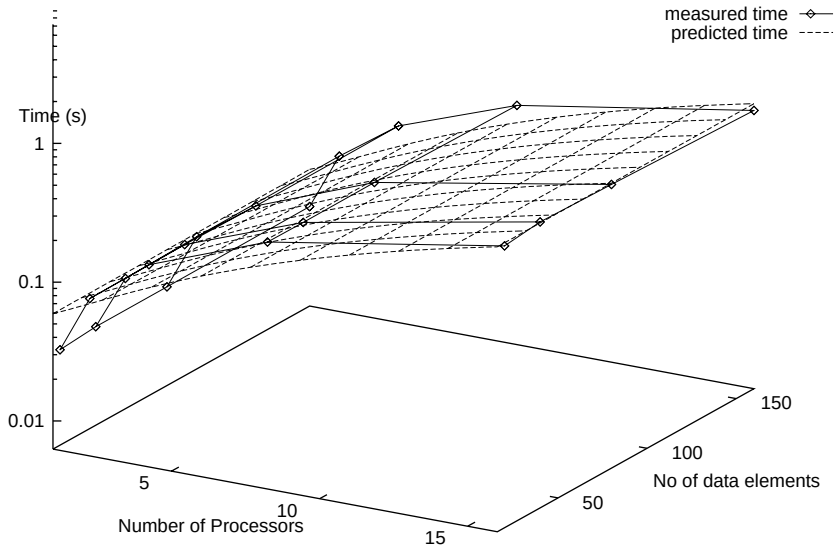

Figure 4.38: Point normalisation predicted and measured times on a network of workstations.

4.3.13 Weighted point selection (`winnow`)

```
# model of winnow performance
maxpts(p,d)          = d * d * maskpercent
minpts(p,d)          = maxpts(p,d) / p
```

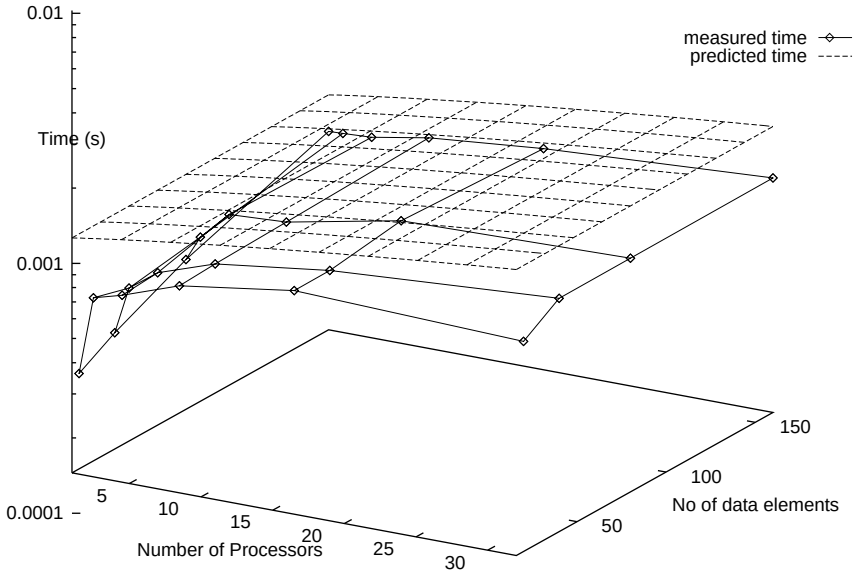

Figure 4.39: Point normalisation predicted and measured times on the Cray T3D.

```
localpoints(p,d,unevenness) = unevenness    * maxpts(p,d) +
                              (1-unevenness)* minpts(p,d)
stride = 4
tbuildlocal(p,d,c)   = c * (d/p) * maskpercent * 5
tminmaxtot(p)        = 3 * tallreduce( p, 1 )
tpivotcompute(p,d,c) = 3 * p * c +
                       localpoints(p,d,c) * 3 * tcompute
tbuffercompute(p,c)  = 2 * p * c
tlocalsort(d,c)      = d * log(d) * c *4
tpackstrided(p,d,c)  = c * 2 * d
ttotal(p,d,c,e) = tbuildlocal(p,d,c) +
                  tminmaxtot(p) +
                  tpivotcompute(p,d,c) +
                  talltoall(p,1) +
                  tbuffercompute(p,d,c) +
                  talltoallv( p, localpoints(p,d,e) )+
                  tlocalsort( localpoints(p,d,e),c ) +
                  tallgather( p,1 ) +
                  tpackstrided( p, localpoints(p) / stride,c )
```

This routine includes many phases and is too complex to make computing the time by hand a sensible proposition, so GNUplot is useful as a quick tool for "what if" calculations. The function `MPI_Alltoallv` performs the redistribution in which each process sends and receives a different amount of data to/from all the others. A parameter of "unevenness" would ideally be used to work out how long the collective redistribution operation will take. The extremes of this parameter are 0.0 (equal distribution) and 1.0 (all in one processor). A mathematical definition is given below:- We have N elements distributed amongst P processors. Ideally there would be N/P in each. Actually there are $N_i (i : 0..P-1)$.

$$\begin{aligned} \text{Deviation } D_i &= \text{Abs}(N_i - \frac{N}{P})) \\ \sum_{i=0}^{P-1} D_i &= 0 \textit{ (for equal distribution)} \\ &= 2N(1-1/P) \textit{ (for all elements in one processor)} \\ \text{Unevenness} &= \frac{\sum_{i=0}^{P-1} D_i}{2N(1-1/P)} \end{aligned}$$

As a simplification, the *maximum* number of elements stored in any one processor was taken as the parameter to determine the expected time of `talltoallv`.

Figure 4.40 shows the measured and predicted times on the Cray T3D. The unevenness parameter was varied from 0 to 0.1; perfect data distribution led to an over optimistic prediction and 10% unevenness was about right. 100% unevenness creates no speedup. This illustrates the sensitivity of this algorithm to the data distribution.

On the network of workstations, the algorithm produces a slowdown (figure 4.41), and this is expected even with perfect distribution.

4.3.14 Vector difference (`vecdiff`)

This program computes the maximum element by element difference between two vectors. The vectors are distributed evenly across the processors; each computes the maximum difference between the local elements, and then there is a global reduction to determine the overall maximum difference.

Thus this problem is a balance between the computation time and the reduction time. The crossover point is located where the time for the local maximum computation is equal to the time for the allreduce operation.

```
# model of vecdiff performance
```

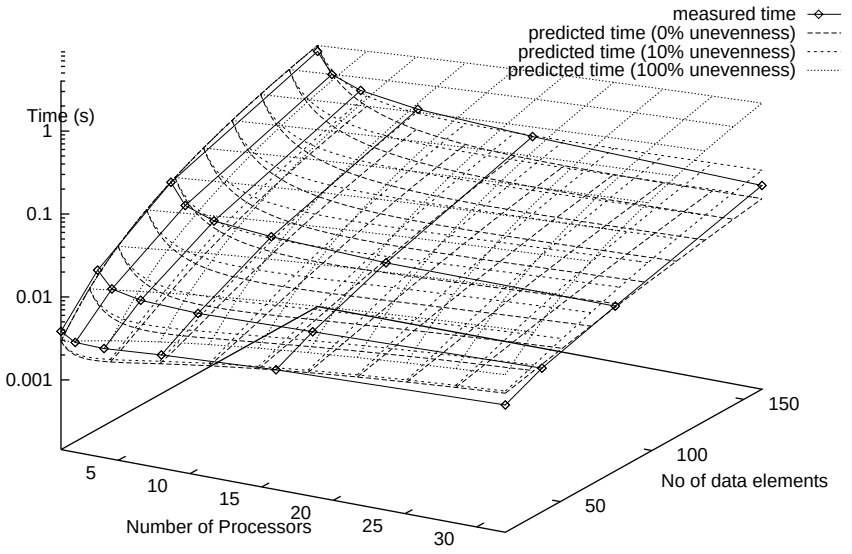

Figure 4.40: Winnow measured and predicted times on the Cray T3D.

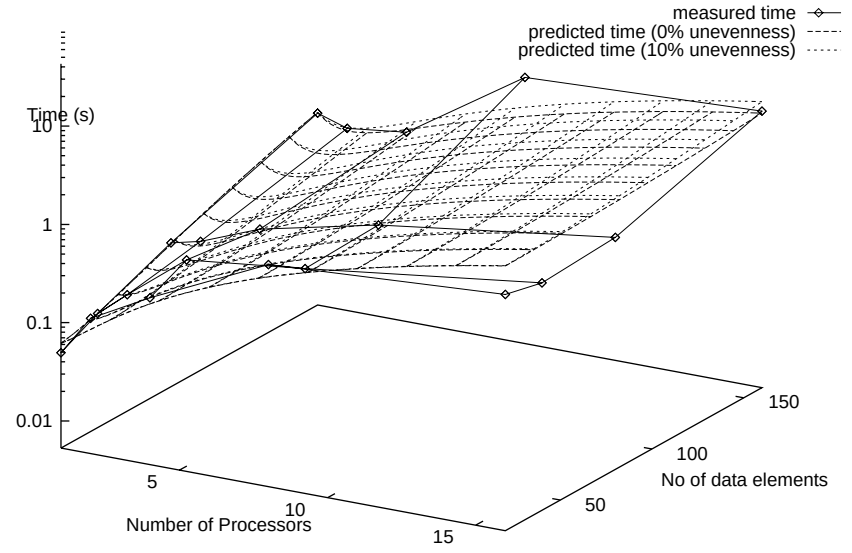

Figure 4.41: Winnow measured and predicted times on a network of workstations.

```
tlocalmax(p,d,c) = c * (d*2.5 / p)
tglobalmax(p)    = tallreduce( p, 2 )
ttotal(p,d,c)    = tlocalmax(p,d,c) + tglobalmax(p) + tbarrier(p)
```

Figure 4.42 compares measured and predicted times on a network of workstations. The times are accurately predicted for all but the single processor case, for which the communications operations take less time than the models predict. On the Cray (figure 4.43), the prediction is a factor of two pessimistic. This is because the `tallreduce` operation is a factor of two faster than the model predicts for small message sizes.

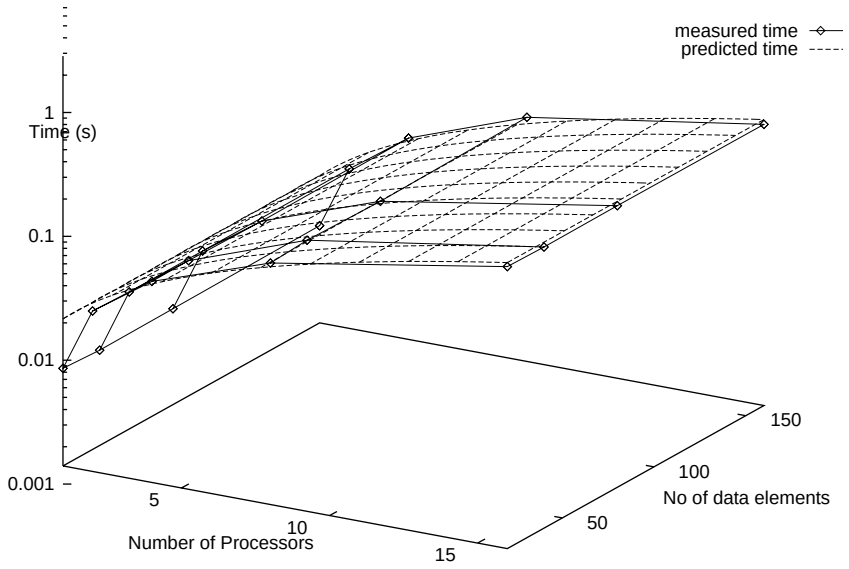

Figure 4.42: Vecdiff measured and predicted times on a network of workstations.

4.4 Conclusion

A graphing package is a very powerful and simple tool for performance prediction when used with data sheet performance models. The importance of performance modelling was highlighted by the number of slowdowns obtained with the Cowichan problems on networks of workstations.

Scalability plots are straightforward to generate, as are plots varying parameters across a wide range to check that the design performs as intended.

The best predictions were for the programs dominated either by the communications or by the computation. Programs spending equal times computing and communicating are very sensitive to minor changes, so are hard to predict.

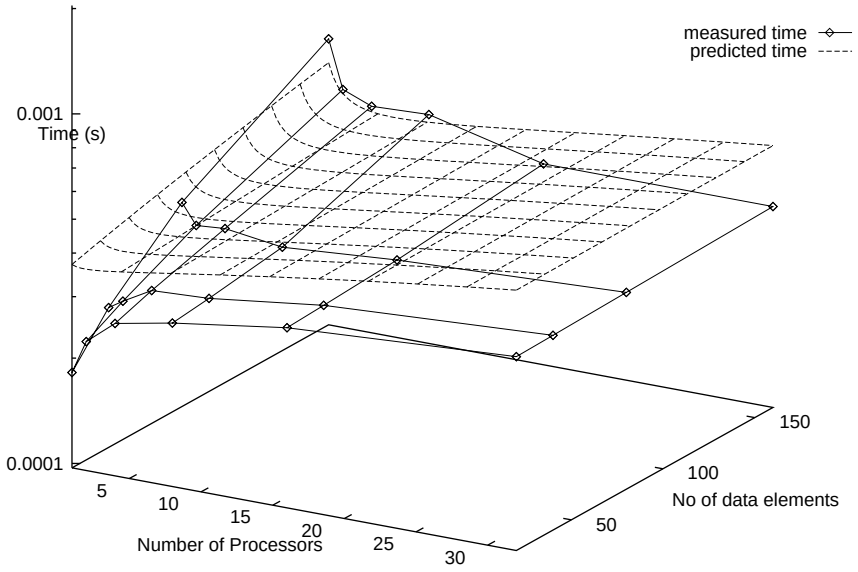

Figure 4.43: Vecdiff measured and predicted times on the Cray T3D.

The restrictions of the technique are that it requires the bounds on loop iterations to be fixed at design time and the fact that data dependencies are difficult to include. As the models for programs are generated by hand, there is a danger that seemingly unimportant phases of the algorithm will be left out, leading to overly optimistic predictions.

These restrictions are removed by the reverse profiling technique described in the next chapter.