Chapter 3

The Performance Characterisation of MPI Functions

It is unusual for detailed performance information to be available to a parallel programmer setting out to design a program. This is rather a serious omission, since the point of using a parallel machine is to obtain better performance, and without performance information it is hard to make design tradeoffs. This chapter presents a method for characterising an MPI implementation which produces approximate equations for the behaviour of each function. The result is an automatically generated LaTeX datasheet for the MPI implementation. A summary file suitable for computer aided performance prediction is also produced by the routines.

3.1 Introduction

A programmer sitting down to design a program for a parallel machine with (say) an MPI manual will encounter design choices with precious few design guidelines. For example, if the program makes extensive use of triangular matrices, is it worth redistributing the data to balance the computation load? Or would the cost of redistribution outweigh the benefits of balanced load and make it faster to suffer the imbalance? The answers to such questions, of course, depend on the relative costs of computation and communication which in turn depend on the architecture. Many attempts at characterising architectures have been made and abstract "bridging models" have been proposed to simplify life for programmers. Examples include PRAM (shared memory, communication is free) and LogP. As Foster notes [1], they are not especially applicable to program writing. None of the models will tell a programmer that (say) an `MPI_Barrier()` executes instantaneously while an `MPI_Bcast()` requires **O(N*P)** time on the chosen machine.

What would be useful in practice is a table giving the performance of each `MPI_` function as an equation with the amount of data and number of processes as parameters. Any such equation is bound to be a simplification, so information about the accuracy of such estimates should also be included. It is not obvious what form the equations should take. There is a compromise between accuracy and usability; an equation such as:

$$t = 0.15644 + (N_{procs})^{2.2345} * 1.23456 + (N_{data})^{1.0017}$$

may describe the behaviour of a function accurately but would be cumbersome to use in practice.

The best set of design rules would assign simple costs (like **1**, **N**, **0**) to operations enabling rapid paper calculations to resolve design choices like the triangular matrix dilemma above. The most familiar model following this path is the PRAM which assigns zero to the communications cost. Unfortunately such simple design rules are unlikely to describe reality so something more complex is needed. It is interesting to reflect that such informal design rules govern programmers at present, as in "I'll use an asynchronous send here since it will be 'much quicker' than a standard send"; "better not put a barrier in this loop as it will dominate the time".

Simple latency and bandwidth measurements are frequently quoted for machines, for example Nog and Kotz [5]. Ciula [3] compared latency and bandwith for workstations connected by ATM, FDDI, FCS and ALLNODE switches. These were point to point measurements between two machines:

	Latency (us)		Bandwidth (MB/s)	
Network	PVM	MPI	PVM	MPI
ATM	677	702	9.2	9.2
FDDI	913	1115	6.4	7.8
FCS	944	1253	6.4	6.9
ALLNODE	546	546	3.7	4.3

Actually making use of this information is difficult. It shows the cost of bouncing a message from point to point, but not the delay the sender undergoes before the next instruction. It also provides no clue as to the performance of collective communications or to the performance of a set of point to point operations performed concurrently (as in for example a boundary exchange).

The rest of this chapter describes MPI characterisation routines aimed at providing more concrete information to guide designers. Section 3.2 discusses alternative ways of presenting performance information. Section 3.3 describes how the communications functions are timed. Section 3.4 discusses some of the problems inherent in timing communications on parallel machines. Section 3.5 explains how the data sheets are generated, with some sample extracts given in section 3.6; finally section 3.7 concludes.

3.2 Presenting performance information

Three ways of presenting the performance information for MPI functions were considered; interpolating directly from measurements, using a curve fitting technique, and categorising functions into a few simple buckets (such as “quick” and “slow”). A curve fitting technique was chosen as it provides a balance between simplicity and accuracy.

3.2.1 Interpolation from measurements

If enough data values were to be measured for the required architecture, then the performance for a particular function could be read from a table (or extrapolated from adjacent timings). This would be cumbersome without a computer tool to help, but could be approximated by referring to a graph of measurements.

3.2.2 Best fit equations

Curve fitting techniques exist to fit an equation to a set of measurements. The best form of the equation is not obvious; the aim is a simple and accurate equation. A simple form was tried initially, with three coefficients:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff}$$

This plane didn’t fit the curved surface of many of the collective routines (see figure 3.1) so an extra term pd_coeff was introduced:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff} + \text{pd\_coeff} * N_{procs} * N_{data}$$

This improved many of the fits (at the cost of making the equation more complex), but was not sufficient for functions whose time grew with $log(N_{procs})$, so an extra term was introduced:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff} + \text{pd\_coeff} * N_{procs} * N_{data} + \text{logp\_coeff} * \log(N_{procs})$$

This made the fits better (by eye), but some functions had a dependency on $\log(N_{procs}) * N_{data}$, so yet another term was added.

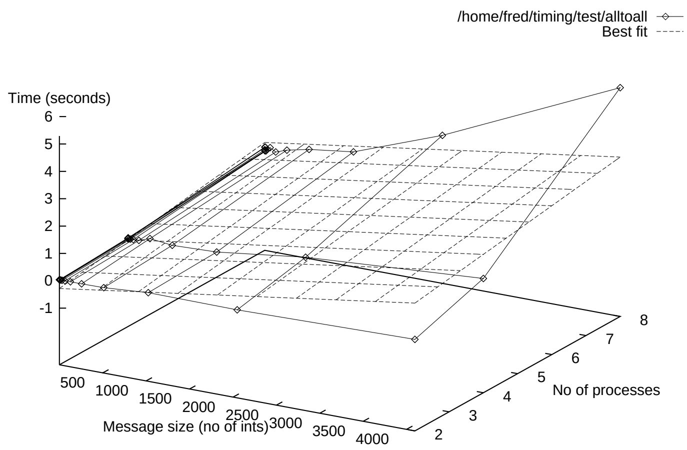


Figure 3.1: Example measured times for `MPI_Alltoall()` with fitted *plane*.

By this stage the equations gave reasonable fits in all cases measured (on Sun networks and the T3D) but the equation was too complex to allow quick estimates.

Another approach, suggested by Marr [4], was that rather than have one form of equation for all functions, it might be better to try many different, simpler, equations and see which fits best. The equations for the time of an operation in terms of the number of processes in the group p and the message size d take the form of a constant factor, a "startup parameter" dependent on the number of processors, and a "data dependent" factor dependent on the message size and the number of processors:-

$$t(p,d) = \mathrm{c\_coeff} + \mathrm{s\_coeff} * \mathbf{startupfn(p)} + \mathrm{d\_coeff} * \mathbf{datafn(p,d)}$$

$$\mathbf{startupfn(p)} = \text{one of} \begin{cases} p \\ \log(p) \\ p^2 \end{cases}$$

$$\mathbf{datafn(p,d)} = \text{one of} \begin{cases} d \\ pd \\ \log(p)d \\ p^2 d \end{cases}$$

Thus a total of 12 curve fits are performed using every combination of the startup and data functions and the best fit is selected. These functions were chosen as they provide reasonable fits for all cases thus far encountered. This approach gives simple functions with three parameters, e.g.

$$t = 14 + 123 * \log(N_{procs}) + 1.2 * N_{data} * \log(N_{procs})$$

There is an issue of the number of significant figures which should be quoted. An estimate of the standard error of the parameters may be made if the error of the measurements is known, and this can be used to give expected minimum and maximum times (*t_min* and *t_max* equations). This standard error may be used to guide the accuracy of quoted figures. An alternative, justified because it makes quick rough-and-ready calculations easier, is to round the coefficients to the nearest order of magnitude, so the above equation becomes:

$$t = 10 + 100 * \log(N_{procs}) + 1 * N_{data} * \log(N_{procs})$$

An attractive way to do this is to use the logarithm of the time (in μs) rounded to the nearest integer. This gives buckets at $1\mu s$, $10\mu s$, $100\mu s$, $1ms$, $10ms$ etc. which allows quick and simple performance estimates to be done, accurate to within an order of magnitude.

3.2.3 Categorisation of functions into "buckets"

Functions could be placed into one of two categories - "Quick" ones and "Slow" ones, with some threshold used to differentate between the two (say $100\mu s$). This has the advantage of being very simple to use but lacks the subtlety required.

3.3 Measuring MPI performance

Characterising the behaviour of the MPI functions is straightforward in principle; measure the time to complete N calls and take the average. The parameters of interest are the number of processors and the size of the messages.

To time an operation (e.g. `MPI_Bcast()`), a short function is written:-

```
void time_Bcast(int numelems, int iter, double &time)
{
  int *buffer = new int[numelems];
  MPI_Barrier( comm );
  double e1 = MPI_Wtime();

  // Operation to time
  MPI_Bcast( buffer, numelems, MPI_INT, 0, comm );

  time = MPI_Wtime() - e1;
```

```
    time = getmax( time );
    delete buffer;
  }
```

The MPI_Wtime() function is used to time the operation. The processes are synchronised beforehand using an MPI_Barrier. This is not perfect, as some processes may return from the barrier before others, so an alternative synchronisation technique has also been used which first determines the clock skew between different processes' MPI_Wtime() values, then busy waits until the timer reaches an agreed value. This provides synchronisation to a resolution of the short time required to read the timer, but just using MPI_Barrier is more convenient in practice.

The time is measured from this synchronisation point until the last process has returned. The getmax() function uses an MPI_Reduce across all processes to determine this maximum delay and the resolution of the timer is reported from the MPI_Wtick() function.

The parameters are the size of the message and the number of processes in the current communication group comm. These are varied across the range of values of interest on the machine, and each timing is repeated to produce a 3D set of measured times of the operation on the machine.

A separate time_... routine is needed for each MPI function. The routine:-

```
void time_routine(
  op_fn op,
  char *fname,
  int vary_nprocs,
  int vary_nelems
);
```

calls the basic operation timing function, varying the number of processors and data size. Timings are repeated niters times to obtain the minimum, maximum and median values at each data point. (The median is used in place of the mean as it is less susceptible to outlying points).

Operations fall into four classes -

- point to point routines (vary data size),
- collective routines (vary data size and number of processors),
- barrier-type routines (vary number of processors) and

• local operations (vary nothing).

For blocking point to point operations, the time for the send or receive to complete is taken, where both sender and receiver have been synchronised with a barrier beforehand. It is also necessary to measure the receive time of a message which has already arrived. This is obtained by delaying the call to `MPI_Recv` by more than the expected time to transmit the message. In figure 3.3 **Tsend** is the time the sender is delayed; **Trecv** is the time the receiver is delayed when the send and receive start at the same time. **Trecvmin** is the best possible receive time, where the message send was deliberately started in advance in order to hide the network latency. With these values, it is possible to calculate the expected delays of any send/receive combination.

For the non-blocking operations, two times are recorded; the time to post the send or receive and the time for the subsequent `MPI_Wait()` to complete. Another figure which is useful is the time for which useful work may be performed before calling `MPI_Wait` without imposing any extra delay. This is measured by using `MPI_Test` to poll the request. Figure 3.4 shows the times measured for non blocking point to point routines. **Tisend1** is the time to post an asynchronous send. **Tisend2** is the time for the asynchronous send to complete. **Tisendoverlap** is the amount of time available for hiding computation between posting the send and it completing. Similarly **Tirecv1** is the time to post a receive, **Tirecv2** is the time for it to complete and **Tirecvoverlap** is the amount of useful work which can be hidden.

These point to point measurements are taken using just two processes on a quiet machine. They do not take message contention into account. To incorporate message contention, variants of the point to point measurements are also taken where half of the processes send to the other half. This leads to more pessimistic estimates of point to point communication times. A random permutation is used to select which pairs of processes communicate. It would also be possible to take measurements of particular point to point patterns such as neighbour exchanges on meshes. The random pattern was chosen in preference to this as it acts as a likely average time for such communications.

Collective operations will take different times for the different processes involved (figure 3.2), but to make the problem tractable the time measured is the time between the first process starting and the last process finishing. Some operations have more parameters than just the number of processes in the group and the data size. For `MPI_Comm_split`, there is a choice of parameters to measure, since a group may be split in any fashion according to the key. The one measured here is the time to split a group of size N into two. For `MPI_Reduce`, the time for the reduction function ought to be part of the equation but initially only the standard addition operation is used.

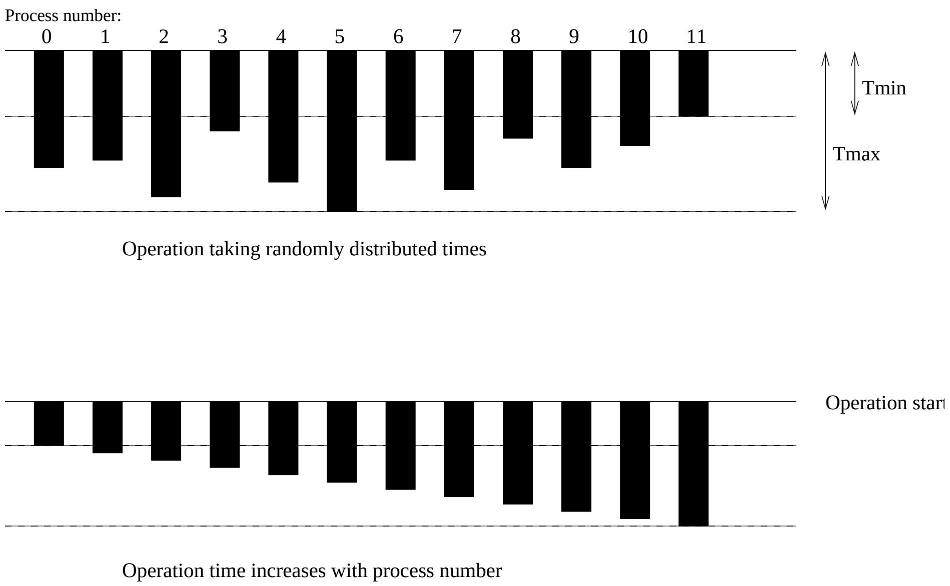


Figure 3.2: Possible discrepencies between collective timings made on different processes.

Detailed characterisation of the performance of *subgroups* was not carried out, as all of the test routines just made use of the global group of all processes `MPI_COMM_WORLD`. However, characterising the performance of these subgroups would be an interesting extension as it would quantify the benefits which could be expected from exploiting the (hopefully) faster performance of smaller, logically independent groups of processes.

One simple way of incorporating this into the measurements would be to repeat each collective measurement for a group of size P with an equivalent measurement using two groups each of size $P/2$ concurrently. The way in which the group is split is likely to have an effect on the performance, so it would be necessary to perform the obvious splits (randomly selected, first and second half, alternate). This would only give information for splitting a group into two. Repeat measurements would be needed for four, eight etc., giving a large amount of information in the datasheet.

Characterising the general case, where different subgroups perform entirely different operations, is difficult and in practice it would probably be simplest to use the models of the supergroup instead. Further work would be required to devise a usable general model for independent subgroups.

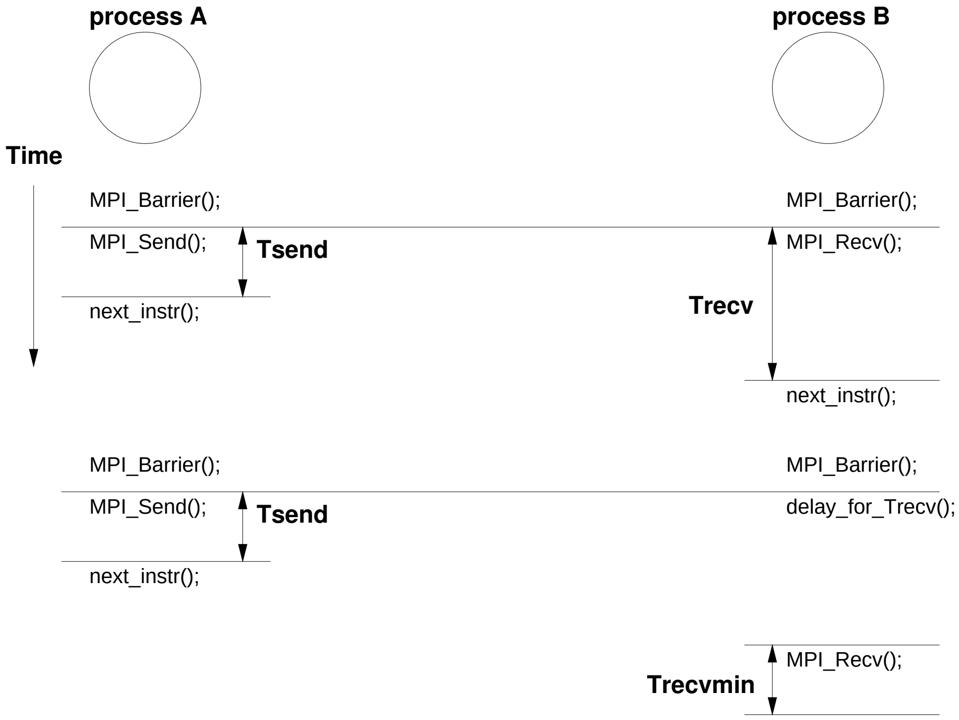


Figure 3.3: Times measured for blocking point-point operations.

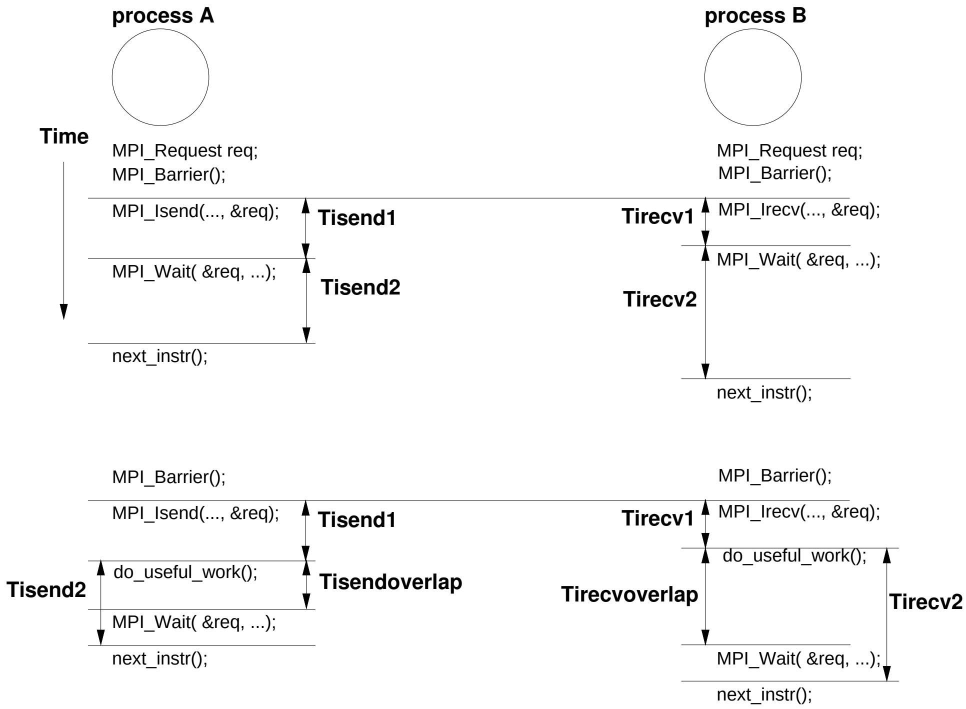


Figure 3.4: Times measured for non-blocking point-point operations.

3.4 Pitfalls

The timings have to be run when no other users are loading the system. These can seriously affect the results, by several orders of magnitude. This also affects the validity of predictions made using the performance models of the building blocks, since they will be unreliable on a loaded system.

There will also be variability on an unloaded system caused by interrupts, disk accesses not taking a uniform time and virtual memory paging. This may be incorporated into the measurements by repeating each timing and noting the spread.

On some machines, the time for message passing operations may vary by orders of magnitude depending on where the message happens to be in the memory hierarchy. This is a very hard variation to estimate in practice, since it depends on where the compiler puts data, the exact sizes of the various levels of caches, the buses between caches, the main memory system organisation and access time, and whether the data in the message has been recently accessed.

Rather than attempt to solve the Sisyphan problem, a decision was made to use freshly allocated memory for all timings of the routines. Since newly allocated memory will not have been read from by the timing program, it will reside in main memory rather than cache. Thus all MPI time measurements given are for the case where the data has not been cached, and may be more pessimistic than is observed in actual programs.

It would be possible to repeat all measurements for the case where the messages are cached, increasing the complexity of the data sheets. The question then arises: how will the user know which model to use? Without performing a full system simulation, it is very difficult to tell.

3.5 Data sheet generation

The generated data sheets give a page for each MPI function measured. Examples are given in section 3.6 below. Each page has the sections:

- Operation
- Full timing graph
- Fitted curve

In addition, there are summary tables at the start. The source code for the routines is given in [2]; the sections below describe how data sheets are generated.

3.5.1 Curve fitting

To obtain an equation summarising the performance of each function from the measured data, a linear least squares technique is used. The result is a curve for point to point operations and a surface for collective routines. As explained above, the form of the equation to fit is:

$$t = \text{c\_coeff} + \text{s\_coeff} * [\text{start\_mode}] + \text{d\_coeff} * [\text{data\_mode}]$$

where start_mode is one of (nprocs, log(nprocs), nprocs^2) and data_mode is one of (ndata, ndata $*$ nprocs, ndata $*$ log(nprocs), ndata $*$ nprocs^2).

The least squares curve fit is run using all combinations of start_mode and data_mode to see which fits best (i.e. has the smallest chi-squared value).

The "goodness of fit" parameter Q is also recorded to give an indication of how accurately the equation fits the measured data. The standard error of each parameter yields equations giving the maximum and minimum expected times. This should only be used as a rough guide, as there is no guarantee (or even likelihood) that the measured data conforms to a normal distribution. However, it is useful to have at least some indication of expected confidence intervals.

3.5.2 Implementation of the surface fit routine

The input to the curve fitting routine is a set of measurements giving the group size, the data size, the average time and an estimate of the error of each measurement. The results are the coefficients of the equation, with estimates of the errors for each coefficient, e.g.

$$T_{bcast}(\mu s) = (100 \pm 10) + (6 \pm 0.8) \times nprocs + (0.04 \pm 0.002) \times nprocs \times ndata$$

To perform the curve fit, it is convenient to write the equations in matrix form and then apply a matrix solver to obtain the coefficients:

$$y(\mathbf{x}) = \sum_{k=1}^{M} a_k X_k(\mathbf{x})$$

where $\mathbf{x}$ is the vector of input parameters $(nprocs, ndata)$ and $X_1(\mathbf{x}), ..., X_M(\mathbf{x})$ are the functions of $\mathbf{x}$.

The merit function (χ^2) is

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2$$

From the design matrix the *normal equations* may be derived. These are solved using Gauss-Jordan elimination to give the vector of coefficients, the chi-squared value and the vector of coefficient variances.

This process is repeated for all possible combinations of the X_k functions and the combination which yields the lowest chi-squared value is chosen.

3.5.3 Units and Significant Figures

The simplest unit of time is absolute time for a specific architecture. Unless there are real time constraints, however, it is more useful to know a time relative to the time to perform arithmetic (or memory access) operations. The times could be given in processor cycles, but this wouldn't mean *that* much because of the memory hierarchy. Hence times are given in milliseconds or microseconds (depending on a compile time flag).

The appropriate number of significant figures to quote in the datasheet is debatable. One extreme is to round all coefficients to the nearest order of magnitude. This makes rough and ready estimates of a function's speed very straightforward. The other extreme is to use as many figures as are justifiable given the standard errors of the coefficients. The solution chosen was to let the user choose the number of significant figures at compile time.

3.5.4 Output formats

The datasheet generation is performed by two routines;

- `rawtiming` is an MPI program which gathers the data, and
- `dsheet` performs the curve fitting and generates the latex document.

The split into a data gathering program and a data analysis routine makes performing several analyses on the same data more convenient than if the two routines were combined. Figure 3.5 shows the connections between the routines.

3.5.4.1 Output from `rawtiming`

For each operation measured, two data files are produced; one for large message sizes and one for small message sizes. The reason for separating small and large message sizes is that there is often a nonlinearity in the performance curve. Performing separate fits for small and large messages makes for more accurate equations. The threshold between small and large messages may be varied; by default it is set at 32 integers, or what may reasonably be regarded as parameters

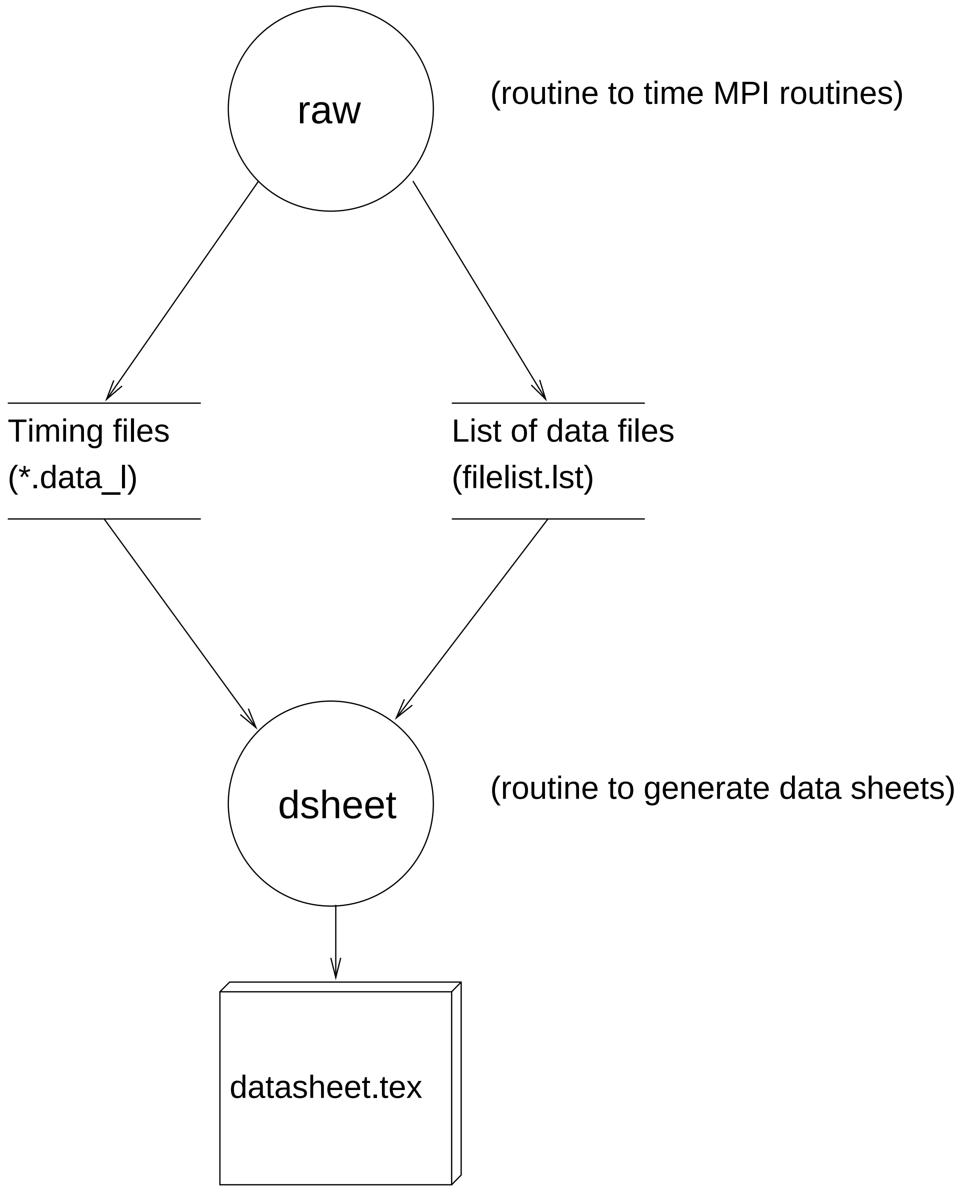


Figure 3.5: Datasheet generation routines.

rather than a block of data. For example, the two data files for the `allgather` operation are called:

```
allgather.data_l
allgather.data_s
...
```

The format of the data files is

```
2 32 0.000138214 0.0000045
2 64 0.000146659 0.0000043
2 128 0.00016432 0.0000041
2 256 0.000201307 0.0000035
2 512 0.000270488 0.0000065

4 32 .... etc
```

with the columns storing the number of processes, the message size, the median time taken and an estimate of the measurement error.

The file `filelist.lst` is also generated; it stores the parameters, date on which the measurements were taken, and a list of the routines which were measured. It is used by the `dsheet` program to generate the datasheet.

3.5.4.2 Output from `dsheet`

The `dsheet` program turns the raw data into a latex file `datasheet.tex` along with a set of GNUplot scripts for displaying the measured data and the approximated curve fits. It expects the file `filelist.lst` and all the raw data files to be in the current working directory.

3.5.5 A simple calculating utility

As an alternative to using the datasheets by hand, a simple tool was developed to evaluate the equations for a given operation, data and group size.

The routine reads in the results of `dsheet` and presents a simple command line interface. A sample session is:

```
[balnagowan]fwh: timecalc summary.lst

Simple utility for estimating the time of MPI
functions based on the summary.lst file produced by
```

```
rawtiming.
Enter <name> <groupsize> <datasize>

>
> bcast 16 1000
# t = (0.000106549 +/- 1.23071e-05) +
#     (6.35065e-06 +/- 7.83058e-07)*nprocs +
#     (4.39693e-08 +/- 1.75882e-09)*nprocs*ndata
Times (s): min=0.00086141 avg=0.000911668 max=0.000961926
>
> alltoall 16 1000
# t = (1.41845e-05 +/- 4.80534e-07) +
#     (4.61065e-05 +/- 2.9256e-06)*nprocs +
#     (2.44134e-07 +/- 8.85081e-09)*nprocs*ndata
Times (s): min=0.00444749 avg=0.00465803 max=0.00486857
```

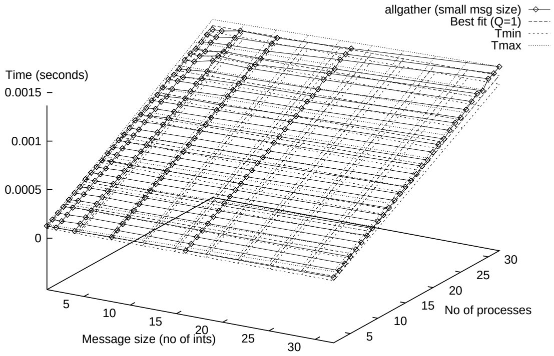
3.6 Example data sheets

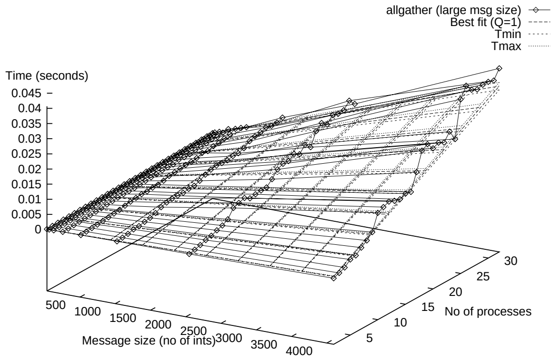
Table 3.1 shows an excerpt from the summary part of a data sheet (for the Cray T3D); a detailed page is shown in figure 3.6. A complete data sheet document is included in appendix B.

3.7 Conclusion

This chapter has presented a routine for characterising the performance of MPI functions. The approach is a compromise between usability and accuracy. The aim has been to allow design for specific architectures based on measured data rather than guesswork. Computer based design tools using this data, as an alternative to pencil and paper, are the subject of the next three chapters.

allgather





$$T_{allgather}(\mu s) = \begin{cases} (50 \pm 20) + (40 \pm 1) \times nprocs + (1 \pm 0.9) \times ndata & \text{if } ndata <= 32 \\ (4 \pm 20) + (40 \pm 3) \times nprocs + (0.3 \pm 0.009) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

19

Figure 3.6: A page from an automatically generated MPI data sheet.

MPI Function	Expected time (μs)			Goodness of fit (Q)
send	$70 +$	$3 \times d$		0.61
ssend	$100 +$	$2 \times d$		0.84
rsend	$70 +$	$3 \times d$		0.6
isend1	$70 +$	$3 \times d$		0.47
isend2	$20 +$	$0 \times d$		0.93
isendoverlap	$3 +$	$0 \times d$		0.97
recv	$70 +$	$5 \times d$		0.46
recvmin	$50 +$	$3 \times d$		0.43
irecv1	$30 +$	$0.4 \times d$		0.96
irecv2	$60 +$	$3 \times d$		0.57
irecvoverlap	$0.5 +$	$1 \times d$		0.043
sendrecv	$100 +$	$5 \times d$		0.57
pingpong	$200 +$	$8 \times d$		0.52
alltoall	$10 +$	$50 \times p +$	$0.2 \times p \times d$	1
gather	$80 +$	$8 \times p +$	$0.07 \times p \times d$	1
allgather	$4 +$	$40 \times p +$	$0.3 \times p \times d$	1
reduce	$200 +$	$10 \times p +$	$0.6 \times log(p) \times d$	1
allreduce	$300 +$	$20 \times p +$	$0.9 \times log(p) \times d$	1
bcast	$100 +$	$6 \times p +$	$0.04 \times p \times d$	1

Table 3.1: Summary table for Small messages (32 integers or less). p is the number of processors in the group and d is the message size (number of integers)

Bibliography

[1] I. Foster. *Designing and Building Parallel Programs*, chapter 3. Addison-Wesley, 1994. Available online at http://www.mcs.anl.gov/dbpp/.

[2] F.W. Howell. MPI Data Sheet Generation Routines. `http://www.dcs.ed.ac.uk/home/fwh/perfchar`, 1996.

[3] K. Ciula. PVM and MPI benchmarks on the LACE cluster. `http://www.lerc.nasa.gov/WWW/ACCL/PARALLEL/benchmark.html`, 1995.

[4] M. Marr. Equations for describing MPI run times. (private communication), April 1995.

[5] S. Nog and D. Kotz. Performance Comparison of TCP/IP and MPI on FDDI, Fast Ethernet and Ethernet. Technical Report PCS-TR95-273, Department of Computer Science, Dartmouth College, Hanover, NH, November 1995. Available at `http://www.cs.dartmouth.edu/reports/abstracts/PCS-TR95-273.html`.