Chapter 7

Conclusion

Several methods for attacking the central problem of designing explicitly parallel programs have been presented.

The techniques have focussed on solving the *low level* aspects of parallel program design rather than in creating higher level abstractions. This is because the low level problems have not been solved adequately and higher level programming models are all built on the low primitives. The message passing model of MPI was used.

The main difficulty is developing a technique which is simple enough to use at the initial stages of design yet is accurate enough to provide meaningful guidance. The main competition for any tool for parallel program design is not so much an alternative tool, rather the current situation where performance is left as a "tuning" task to be done after the event. Is it so bad that this aspect is left to tuning? Is design important? In some ways the answer is no. Since software is (superficially) easy to change, why not just build a program one way, test it then make design changes afterwards? In other ways, the fact that the performance characteristics of the primitives are not given means that the program designer is forced to make decisions which affect the performance with nothing other than guesswork and intuition for guidance. It is like designing a circuit with no data sheets.

So the initial phase of work was to provide "data sheets" for programmers (chapter 3). These may be used to provide concrete data to help with pencil and paper calculations at the initial stages of design. Alone, these may be sufficient for many people. A characterisation program generates the sheets automatically for an MPI implementation. It times all the MPI functions using a range of data and machine sizes, then fits a curve to the data. The aim of the data sheets is to describe the delays as seen by the programmer and not to characterise the hardware performance. Thus the time for an `MPI_Send` is quoted as the time

a process is delayed by calling `MPI_Send` (and not the time for the message to arrive). Sheets have been generated for a network of workstations, the Cray T3D and the IBM SP2.

How can the information in the data sheets be best used? This was addressed in chapter 4 which used the raw data from such simple models along with a graphing package to produce scalability plots from equations. This is a very quick method for obtaining rough estimates. The method produces useful graphs showing how much (if any) speedup is expected. The shapes of the expected speedup curves are very similar to those measured on the Cray T3D and a network of workstations. It is easy to see the effects of varying input parameters on a program's overall performance; for example computation time is only predictable to an order of magnitude so speedup curves at both ends of the compute time range can be produced. The restrictions of the technique are that the models are generated by hand, and it is hard to incorporate data dependent communications.

For more complicated patterns of communication, or where more detail is needed, the reverse profiling technique of chapter 5 provides performance prediction using the MPI profiling interface. This applies the data sheet model to programs in the development stage to produce timing diagrams for a single run or scalability graphs for multiple runs. The attraction of this technique is its ease of use. Predictions may be obtained as part of normal development. It is most appropriate for producing timing diagrams showing the detailed behaviour of a single run. Cacheing effects mean that compute time may only be estimated to a factor of ten, but communications time is predicted to a factor of two. The program's exact data dependent communications patterns are incorporated into the expected timing diagram, as long as there are no non-deterministic receives.

Non-deterministic programs may be handled using discrete event simulation. Chapter 6 described a version of this approach. It is a direct execution simulator which uses the running application to drive the simulator kernel. It generates predicted timing diagrams, and because it maintains strict ordering of simulation events it is able to handle non-determinism correctly. The simulator implements low level message passing two to three times faster than implementations of MPI on a single workstation. Because it is a sequential simulator, however, the time to simulate a program running on a parallel machine grows with the number of processors simulated. The MPI data sheets provide the communications model used by the simulator. In addition to these models, simulation allows more detailed models of network architectures to be specified, and some experiments were conducted using graphical techniques to keep the models visible. The cycle count-

ing technique was also used to obtain more accurate estimates of compute times. However it was found to be too cumbersome for widespread use. The simulation approach provides the most detailed results and similar techniques have been suggested by others for parallel program development. However it is too detailed for most developers and it requires a re-implementation of the message passing interface rather than simply building on top of an existing one.

7.1 Prediction as part of design?

In the introduction, it was stated that the ideal was to move away from post-mortem techniques for performance analysis towards incorporating performance into the design stage. From a design point of view, it is better to obtain evaluations of proposed solutions at an early stage of development rather then when coding is completed. The lightweight pencil and paper and graphing techniques may be applied without having to realise the design as a concrete implementation, so fit naturally into the early stages of design to help choose between alternative strategies. The more sophisticated techniques of reverse profiling and simulation both rely on complete programs, or sections of programs, in order to generate more accurate predictions. Thus they are appropriate later in the design cycle for selecting between different key algorithms or determining whether how a program will run on a possibly unavailable machine.

The increase in level of detail of the approaches ties in naturally with top down design, since an appropriate prediction technique may be used at each stage of refinement. At the simplest level, overall estimated timings for application phases may be used. The few phases expected to take the majority of the time may be analysed using a more detailed method. For all the MPI programs developed, the application phases were separated with some form of global communication or synchronisation, so the total time could be calculated by summing the component phase times. This separation of phases (into input, compute and output stages for example) was done in order to obtain correct behaviour of the programs, but also made modular prediction of performance simpler. The BSP model uses the same approach throughout to simplify predictions.

7.2 Further work

7.2.1 Data sheets

Parallel programmers are not given sufficient information at design time to design effective parallel programs. The MPI data sheets presented in chapter 3 go some way towards rectifying this situation for message passing, but similar measurements should be available for other programming models.

The design of the MPI data sheets themselves could be improved, possibly expanding the summary section at the start to include sample times for "common" data and machine sizes in order to save having to plug values into an equation. The current data sheet generator could be expanded to characterise I/O times in addition to the communications functions. It could also characterise a range of computation operations to improve estimates of computation times.

Such data sheets should be a standard part of parallel library documentation.

7.2.2 Combining reverse profiling and simulation

The reverse profiler could be extended by including a parallel simulation engine such as that used in Lapse [1]. This would combine the ease of use of reverse profiling with the ability to handle non-deterministic routines.

7.2.3 Improving compute time prediction

The compiler, processor pipelining and memory hierarchy all conspire to make compute time unpredictable at design time. The only foolproof methods are measurement and full simulation but neither is convenient to do at design time. Intermediate techniques based on cycle counting of assembler code or interpretation of compiler parse trees are too tied to particular implementations to be generally applicable, and in any case are prone to order of magnitude errors.

So it is only practical to predict compute times to within an order of magnitude. The techniques of this thesis left the basic compute time step as a parameter to allow early experimentation to check how sensitive an algorithm is to such compute time variations. In practice, many of the algorithms run on the Cray and the network of workstations produced remarkably little change in expected speedup. They were either communications dominated to an extent that only minimal speedups were available, or computation dominated, giving reasonable speedups across the compute time range. It was only for algorithms with roughly equal computation and communications times that getting the computation step right was essential.

7.3 Overall conclusion

This thesis has presented three approaches to performance prediction; each has its merits. The best technique to use is the simplest one possible. The information in the data sheets along with a calculator (or pen) may well be enough for simple programs. The graphing package is not much more difficult to use for estimates of speedups. For producing timing diagrams showing the way in which complex data dependent communications will work in practice, reverse profiling is as simple to use as standard profiling. Simulation is overkill at the early stages of design, but is appropriate for non-deterministic applications, or for investigating the effects of a program on a network.

Bibliography

[1] P.M. Dickens, P. Heidelberger, and D.M. Nicol. A distributed memory LAPSE: Parallel simulation of message-passing programs. Technical Report NAS1-19480, NASA Langley Research Center, Hampton, VA 23681, December 1993.