

Performance Prediction for Portable Parallel Execution on MIMD Architectures*

Gopal Chillariga

Balkrishna Ramkumar

Department of Electrical and Computer Engineering,
The University of Iowa, Iowa City, Iowa 52242

Abstract

Performance debugging and prediction for parallel systems is a difficult problem. The difficulties in identifying performance bottlenecks stem from the need for an intimate understanding of the underlying architecture. It has been recognized that portability is an important requirement for parallel program development. However, this makes the task of performance debugging even more difficult. In this paper, we present a simulation based approach for performance prediction of portable parallel programs. We demonstrate it using Charm: a message driven programming environment, which provides program portability across a variety of shared and distributed memory MIMD parallel systems. The proposed approach makes it possible to use a single debugging environment for the development of portable parallel software. This environment can provide correctness and performance debugging support that provides the developer with valuable feedback for improving program performance.

1 Introduction

Debugging is a major concern for all parallel programs. The foremost objective of debugging tools is to help ensure program correctness in a functional sense and to this end provide the most useful information in a meaningful way. However, once program correctness issues are resolved, the objective of extracting performance from parallel machines comes to the fore and program performance debugging becomes important. This involves finding bottlenecks causing degraded performance and is usually related to synchronisation overheads, communication latencies, load balancing and computational grain size. Detecting and eliminating performance bugs is considerably more difficult than other bugs encountered with parallel programs particularly since few debugging tools exist for parallel and concurrent program performance debugging.

A lot of work has been done in the area of performance monitoring and the associated intrusion and perturbation issues [8, 11, 9]. Performance visualization has been addressed by tools like *Chitra* [1] which depend on captured

program execution sequences to predict program performance with different input parameter sets. Tools such as Hence [2] for the Parallel Virtual Machine and Projections [12] for *Charm* provide post-mortem performance analysis by collecting information during execution. However, they suffer inherently from the intrusive nature of the process of collecting the execution sequences and/or lack adequate prediction capabilities and are usually application specific. In addition, they do not address the problem of predicting performance across a range of target machines for portable parallel programs.

Program portability has been widely recognized as an important requirement in parallel software development [3, 5, 7]. This makes the task of debugging portable programs becomes even more difficult since development now requires intimate knowledge of *all* target architectures and their debugging tools (assuming they exist). During development of portable programs, an environment that permits performance tuning of a program to yield good performance in a machine independent manner is clearly desirable.

The need for tools which provide adequate performance metrics to permit effective and easy identification of problem areas in a parallel program is being addressed as part of the *Intrepid* project at the University of Iowa. *Intrepid* (*I*ntegrated environment for performance prediction, instrumentation and debugging) is a debugging and instrumentation environment for *Charm*, a language and runtime system for portable parallel programming.

To aid performance debugging and prediction within the *Intrepid* framework, we have created a simulation environment designed to reliably predict the performance of *Charm* applications on several target machines supported by *Charm*. The primary objective is to identify potential performance bottlenecks in the execution of *Charm* programs by estimating various performance penalties including communication latencies, load balancing, information sharing and other system overheads. The system described here permits the development and performance tuning of applications in a single development environment without having the user actually run them on the target machine. Additionally, this relieves the user of the burden of learn-

* This research was supported in part by the National Science Foundation grant CCR-9308108.

ing the use of target specific performance measurement and analysis tools.

The rest of this paper is organised as follows. Section 2 provides an overview of *Charm*. In Section 3, we describe our solution to the performance debugging and prediction problem. Experimental results and the effectiveness of this method are discussed in Section 4 and Section 5 concludes the paper.

2 Charm

Charm [4, 6] is a language for message driven parallel programming and provides runtime support for load balancing, memory management, quiescence detection and message queuing and scheduling. It has been ported to several distributed and shared memory machines including the Encore Multimax, Sequent Symmetry, Alliant FX/2800, nCube/2, Intel iPSC/2, i860, Paragon, CM-5 and a network of homogeneous UNIX workstations. For each architecture, the run time support system is tuned to extract the best performance. A detailed description of *Charm*, its design philosophy and associated issues can be found elsewhere [4, 7]. A brief overview of *Charm* relevant to the rest of this paper is provided here.

Charm applications are expressed as a collection of small grained tasks called *chares*. New instances of a *chare* can be created dynamically. Messages are always sent to an *entry point* which is a block of sequential C-code that determines the action taken to process the message. A *Charm* program consists of one or more *chares* each consisting of definitions of entry points and local variables. Execution at an entry point proceeds uninterrupted to completion till the end of that entry point. The system is then free to pick a new message from the message pool for processing. Activating a *chare* at an entry point involves retrieving the *chare's* data area and executing the entry point code.

The system supports quiescence detection and quiescence can be reported at a user-specified entry point. The system manages the message pool by maintaining *queues* of messages awaiting scheduling. The pool is examined repeatedly (in a *base loop*) and messages are picked (and processed) from this pool as dictated by the queuing strategy used. Several queuing (first-in-first-out, stack, and prioritized) and load balancing strategies are available and the user may choose any of them.

3 Performance Prediction in *Intrepid*

We describe our approach to address the performance debugging problem in the context of portable *Charm* programs. For a given application we desire performance information for varying number of processors (how well does the application scale?) and on different machines (how effective is the application on a different architecture?). We demonstrate that these questions can be answered without detailed knowledge of the desired target architecture. The objective of the simulation environment created within *Intrepid* is to permit users to extract such information while working on a single familiar development environment, for

instance, a Unix workstation, irrespective of the intended target architecture. The approach we take is as follows:

1. Instrument *Charm* and measure various system overheads associated with processing on machines supported by *Intrepid*,
2. Measure various user program execution times associated with the entry point code (discussed in greater detail later in this section), and
3. Simulate the application behaviour on the development platform.

To achieve this, we create a simulation environment for *Charm* applications which is capable of reproducing the application execution behaviour on the machine of choice. The intention is to be able to arrive at estimates of various system overheads and computation efficiency on any machine supported by the *Charm* system. To this end, we first estimate various system overheads by *direct* measurements on the target machines. These measured overheads are then used as parameters in the course of simulation to derive *estimates* of actual system overheads for the application execution being simulated. The emphasis is on relative performance characteristics (e.g., idle time vs. busy time) and not on absolute execution times.

A *Charm* program in execution performs user specified computation and has overheads associated with (i) message pool maintenance, (ii) load balancing, (iii) quiescence detection and (iv) general message processing overhead (such as retrieving a *chare's* data area). The execution can be viewed as consisting of four different states with the system moving from state to state depending on the kind of processing being performed. These states are - (a) *idle*, which is entered whenever the system runs out of messages to process, (b) *queuing* which involves operations on the message queues including system send and receives, (c) *computation* during which user code is executed and (d) *memory management* which involves allocating and freeing memory resources managed under *Charm*.

The total execution time for any application may be represented as

$$T = T_{MsgPool} + T_{MsgProc} + T_{Idle} + T_{LoadBal} + T_{Comptn}$$

For an n processor system, the execution time is

$$T_{total} = \text{Max}(T_1, \dots, T_{i-1}, T_i, T_{i+1}, \dots, T_n)$$

where T_i represents the execution time on processor i .

For purposes of performance prediction, we are interested in information such as the effectiveness of the queuing strategies and the system overhead both in terms of the communication overheads and the message processing overhead (which is also related to the grain size of the computation). The rest of this section examines these issues in relation to various *Charm* system activities mentioned above.

3.1 Message Pool Management

Charm uses a message pool as a source of work for the processing nodes. This pool is maintained as a queue of messages and two different operations are of interest — entering a message into the pool and deleting or extracting a message from the pool.

3.1.1 Shared Memory Systems

For shared memory machines, a shared global pool exists. Locks are used to control access and the waiting time at a lock depends on the number of requests for the lock at a given time. Assuming that the number of requests (n) for a lock during the course of processing a message is distributed uniformly over $[0, N_p - 1]$, the waiting time when N_p processors are being used is given by

$$W = 1/2 N_p (t/T) t$$

where T is the average time spent in a single iteration of the *base loop* and t represents the average time that the lock is held by a processor.

In addition to grain size, several other factors affect the overhead — these include the specific implementation of the locks, and other cache and operating system costs. The influence of these factors on the queuing overhead is captured by considering the queuing overhead as an explicit function of the number of processors being used :

$$t_Q^{ohd} = f(N_p) + W$$

where f is a function of the form

$$f(x) = \sum_{i=0}^n A_i x^i$$

with A_i and n being empirically determined machine dependent constants.

3.1.2 Distributed Memory Systems

In the case of distributed memory machines, a separate pool exists on each node and a load balancing algorithm distributes work among the processors. This means that the overhead associated with message pool maintenance is local and restricted to the node itself except in cases where load balancing dictates redistribution. The additional load balancing overhead is discussed later in this paper. Messages destined for existing chares located on other processors are sent immediately — consequently the message send overhead also becomes part of the queuing overhead.

It was found sufficient to treat the queuing cost as fixed for each node with the implicit assumption that inserting a message into the queue is done in constant time. This is indeed true since queue maintenance strategies like FIFO and LIFO maintain pointers to both the head and the tail of the queue allowing one step insertion and deletion.

3.2 Message Processing Overhead

Some amount of system overhead is associated with message processing in the *base loop*. This is associated with system book keeping for each message. This overhead is independent of the number of processors used but is dependent on the *type* of message being processed. It can be modeled as :

$$T_{MsgProcessing} = f(msg_type)$$

and can be regarded as a fixed constant for distributed memory systems. For shared memory systems, however, this is true only for inter-charge messages. For new chare creation, memory allocation is required and this again involves accessing locks. Consequently, we model the overhead in the same manner as described in section 3.1.1 using a polynomial function of the number of processors.

3.3 Idle and Quiescence Detection

The Charm system initiates quiescence detection only when a processor becomes idle. In other words, the overhead of quiescence is almost entirely absorbed in idle time.

For shared memory systems, quiescence detection involves only the overhead of exchanging information in shared memory and is therefore expected to show a change (albeit minor) with increasing number of processors. Since this occurs only during idle time, the cost is expected to be minimal. An idle processor repeatedly searches for work by peeping into the message pool. However, this does not significantly impact other processors since they access the global pool only in order to place work into it if they are not idle. Consequently, modeling idle time (in a single *base loop* iteration) as being approximately constant (independent of the number of processors) is sufficient since a processor exits the idle state only when a non-idle processor places work into the message pool.

For distributed memory systems, quiescence detection is implemented using messages and the overhead depends on the number of processors being sent messages and the associated latencies.

3.4 Load Balancing

Overheads related to load balancing apply only to distributed memory systems since the nature of the shared pool ensures fairly even load balancing in a shared memory system. For distributed memory systems, the overhead depends largely on the type of load balancing algorithm used. For a given load balancing strategy this overhead appears primarily as the additional operations performed by each processor when messages are exchanged. These operations concern the piggy-backed load balancing information at the sending end and stripping and analysis of this information at the receiving end. These overheads are constant for each message since the amount of data transferred and the computation required to effect the transfer are both constant.

Depending on the load balancing strategy used, additional messages expressly for load balancing may also be exchanged. The associated processing time depends on the

actual implementation of the load balancing strategy. The system provides a suite of useful strategies but the user could potentially choose to use his own strategy. Consequently the actual costs involved with load balancing (as with other chores) are treated similar to the user computation time estimation which is described below.

3.5 Useful Computation

Estimating the time spent in useful computation is difficult since in most cases it depends on the parameters used by the entry point code. The size of the problem (determined by these parameters) and the nature of the entry point code determine the execution time. Since the objective is to come up with a generalised performance model, application specific analytic execution models besides being difficult to determine are not very useful.

Our approach is to use *linear scaling* and estimate the execution time on the target machine by scaling the measured execution time on the simulation platform while maintaining the computation to communication ratio. In order to determine the scaling factor, we run the application on *one* processor of the target machine and on the simulator for a fixed problem size. This permits us to accurately capture the behavior of the target machine including the effects of cache size, virtual memory and other operating system overheads. In our experiments, we chose small problem sizes to determine these scaling factors. This is described in Section 4. Although this approach is heuristic in nature, our studies show that it works quite well. In the simulator being described here, we measure and estimate the execution time for *each entry point* individually and as indicated in the earlier sections, also estimate each of the overheads separately.

4 Experimental Results

The model described in Section 3 was used for purposes of performance prediction on some benchmark programs. In this section, we summarize the results of applying this model to some sample applications for validation.

Intrepid's simulator was used to study the predicted behaviour of two sample programs - *PatternMatch* and *TestGen* (a sequential circuit test pattern generator [10]) - on shared and distributed memory systems. We then compared these predictions to data obtained by instrumenting Charm and running the same programs on the actual machines.

The first sample program, *PatternMatch*, is an implementation of string pattern matching. This involves matching and finding all occurrences of a set of specified strings in some text. The second sample program, *TestGen*, implements a portable sequential test pattern generator for sequential circuits. The performance predictions of the simulator are studied here for some standard benchmark circuits using a limit on the amount of CPU time per fault.

We measure three separate indices to draw conclusions concerning performance — the *computation efficiency* or

the fraction of time spent in useful computation, the fraction of time spent *idling* (grain size and load distribution determine this) and the fraction of time in *system overheads*.

4.1 Shared Memory System – Encore Multimax

Table 1 shows the results of *PatternMatch* on a text containing 18200 characters compared against the actual data for the Encore Multimax on four processors. The predicted performance data for a buffer size of 4000 clearly showed a significant load imbalance. This was solved by reducing the grain size of the computation (increasing the number of chares). The result of successively reducing the grain size are not shown here - instead the prediction error is reflected in Table 1. As the grain size decreases, the absolute system overhead tends to increase (not shown here) but the fraction of total time lost in overheads decreases. This can be attributed to the increased message traffic and the corresponding system message processing overhead and is predicted accurately in the simulation data. Note that the predictions are accurate to within 5% in this case and to under 1% for *TestGen* (Table 2).

4.2 Distributed Memory System – nCUBE/2

Table 1(b) shows the behaviour of *PatternMatch* on the actual machine compared against the simulator predictions. A pseudo random load balancing algorithm is used to distribute load across processors. Scaling factors required (as described in Section 3.5) were obtained using a buffer size of 6500 bytes. The imbalance in load is reflected accurately in the predicted data to within about 1% of the actual performance.

Table 2 shows the performance of the simulator for *TestGen* with some standard benchmark programs. In this case, scaling factors were obtained using the ISCAS circuit *s27*. The data shown here is for two cases - (a) a time limit of 200 milliseconds and (b) a time limit of 1 sec per fault. Note that all figures reported are normalized against the total execution time. Again, the simulation is able to predict the actual data to within 1.5%.

5 Conclusions

Performance prediction and debugging is a major issue in parallel systems and requires considerable attention and expertise on the part of the application/system programmer. Machine specific performance tools aid the process of performance tuning. However, when portable parallel programs are needed, it is unreasonable to expect the user to acquire expertise on a variety of machines and architectures.

In this paper, we have presented a simulation based technique which provides performance related information to users on the development platform itself freeing developers of the burden of learning the use of performance tools on various machines. The results of our experiments with two applications show a mean prediction error of less

BufferSize	μ_C	σ_C	μ_Q	σ_Q	μ_I	σ_I
500	1.8	1.3	1.3	0.9	1.0	0.5
2000	2.3	1.8	1.8	1.8	2.0	1.2
4000	5.1	3.1	9.0	5.0	4.4	2.5
Overall	3.2	3.7	4.1	5.1	2.5	2.7

(a) Encore MultiMax

Circuit	μ_C	σ_C	μ_Q	σ_Q	μ_I	σ_I
500	0.7	0.3	0.9	0.4	0.5	0.3
2000	0.2	0.2	0.2	0.2	0.0	0.2
4000	0.2	0.1	0.2	0.2	0.2	0.2
Overall	0.4	0.3	0.4	0.3	0.2	0.3

(b) nCUBE/2

Table 1: Mean (μ) and standard deviation (σ) of prediction errors in percentages for computation (C), queuing (Q) and idle (I) for *PatternMatch*.

Circuit	μ_C	σ_C	μ_Q	σ_Q	μ_I	σ_I
s208	0.7	0.3	0.9	0.4	0.5	0.3
s820	0.2	0.2	0.2	0.2	0.0	0.2
s1494	0.2	0.1	0.2	0.2	0.2	0.2
Overall	0.4	0.3	0.4	0.4	0.3	0.3

(a) Encore MultiMax

Circuit	μ_C	σ_C	μ_Q	σ_Q	μ_I	σ_I
s208	3.1	2.3	0.0	0.0	3.1	1.3
s820	1.1	0.5	0.0	0.0	1.1	0.5
s1494	0.8	0.1	0.0	0.0	0.8	0.2
Overall	1.6	2.3	0.0	0.0	1.6	2.3

(b) nCUBE/2

Table 2: Mean (μ) and standard deviation (σ) of prediction errors in percentages for computation (C), queuing (Q) and idle (I) for *TestGen*.

than 5% which suggests that the models used were quite accurate. We have shown that simple models for various carefully identified system overheads together with limited information concerning the application performance on a uniprocessor is sufficient to yield highly accurate performance predictions on both shared as well as distributed memory machines.

Acknowledgements

We would like to thank Sandia National Laboratories and the Weeg Computing Centre, University of Iowa for access to their parallel computers.

References

- [1] Abrams M., Doraswamy N., and Mathur A. Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event and Frequency Domains. *IEEE Trans. Par. Dist. Syst.*, vol. 3, no. 6:672-685, Nov. 1992.
- [2] Beguelin A., Dongarra J., Geist A. and Sunderam V. Visualization and Debugging in a Heterogeneous Environment. *Computer*, June 1993.
- [3] Carriero N., and Gelernter D. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Comp. Surv.*, vol. 21, no. 2:323-357, Sept. 1989.
- [4] Fenton W., Ramkumar B., Saletore V.A., Sinha A.B., Kalé L.V. Supporting Machine Independent Programming on Diverse Parallel Architectures. In *Intl. Conf. Par. Proc.*, pages II:193-201, August 1991.
- [5] Gabber E. VMMP: A Practical Tool for the Development for Portable and Efficient Programs for Multiprocessors. *IEEE Trans. Par. Dist. Syst.*, pages 304-317, July 1990.
- [6] Kalé L.V. The Chare Kernel Parallel Programming System. In *Intl. Conf. Par. Proc.*, pages II: 17-25, August 1990.
- [7] Kale L.V., Ramkumar B., and Sinha A.B. The CHARM Parallel Programming Language and System: Part I - Description of Language Features. *IEEE Trans. Par. Dist. Syst.*, 1994. (submitted).
- [8] Lange F., Kroeger R., and Gergeleit M. JEWEL: Design and Implementation of a Distributed Measurement System. *IEEE Trans. Par. Dist. Syst.*, vol. 3, no. 6, Nov. 1992.
- [9] Malony A.D., Reed D.A., and Wijshoff A.G. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Trans. Par. Dist. Syst.*, vol. 3, no. 4, July 1992.
- [10] Ramkumar, B., Banerjee P. Portable Parallel Test Generation for Sequential Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 220-223, Nov. 1992.
- [11] Rinard M. C., Scales D. J., and Lam M. S. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *Computer*, June 1993.
- [12] Sinha. A. B., and Kalé L. V. A Framework for Intelligent Performance Feedback. In *Intl. Conf. Par. Proc.*, Aug. 1994.