Implementation Lessons of Performance Prediction Tool for Parallel Conservative Simulation

Chu-Cheow Lim[1], Yoke-Hean Low[2], Boon-Ping Gan[3], and Wentong Cai[4]

[1] Intel Corporation
SC12-305, 2000 Mission College Blvd, Santa Clara, CA 95052-8119, USA
chu-cheow.lim@intel.com

[2] Programming Research Group, Oxford University Computing Laboratory
University of Oxford, Oxford OX1 3QD, UK
mlow@comlab.ox.ac.uk

[3] Gintic Institute of Manufacturing Technology
71 Nanyang Drive, Singapore 638075, Singapore
bpooi@gintic.gov.sg

[4] Center for Advanced Information Systems, School of Applied Science
Nanyang Technological University, Singapore 639798, Singapore
aswtcai@ntu.edu.sg

Abstract. Performance prediction is useful in helping parallel programmers answer questions such as speedup scalability. Performance prediction for parallel simulation requires first working out the performance analyzer algorithms for specific simulation protocols. In order for the prediction results to be close to the results from actual parallel executions, there are further considerations when implementing the analyzer. These include (a) equivalence of code between the sequential program and parallel program, and (b) system effects (e.g. cache miss rates and synchronization overheads in an actual parallel execution). This paper describes our investigations into these issues on a performance analyzer for a conservative, "super-step" (*synchronous*) simulation protocol.

1 Introduction

Parallel programmers often use performance predictions to better understand the behavior of their applications. In this paper, we discuss the main implementation issues to consider in order to obtain accurate and reliable prediction results. The performance prediction study[1] is carried out on a parallel discrete-event simulation program for wafer fabrication models. Specifically, we designed a performance analyzer algorithm (also referred to as a parallelism analyzer algorithm), and implemented it as a module that can be "plugged" into a sequential simulation to predict performance of a parallel run.

[1] The work was carried out when the first two authors were with Gintic. The project is an on-going research effort between Gintic Institute of Manufacturing Technology, Singapore, and School of Applied Science, Nanyang Technological University, Singapore, to explore the use of parallel simulation in manufacturing industry.

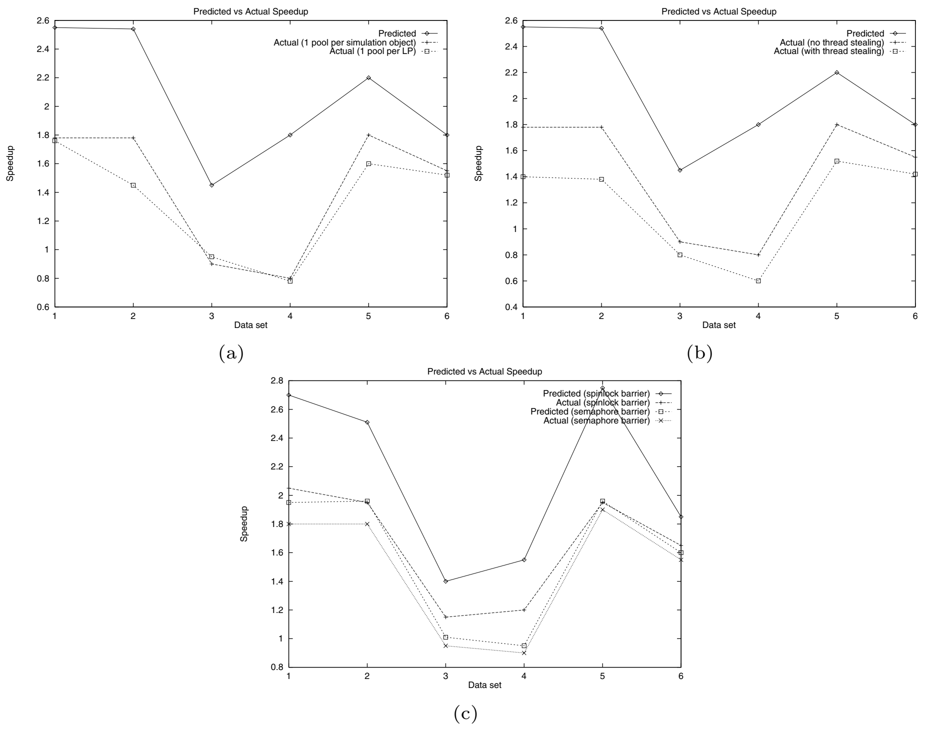

Fig. 1. Predicted and actual speedups for 4 processors using conservative synchronous protocol. (a) one event pool per simulation object and per LP. (b) thread-stealing turned on and turned off in the Active Threads library. (c) spinlock barrier and semaphore barrier.

2 Analyzer for Conservative Simulation Protocol

The parallel conservative super-step synchronization protocol and the performance analyzer for the protocol are described in [1]. The simulation model is partitioned into entities which we refer to as *logical processes* (LPs), such that LPs can only affect one another's state via event messages. In our protocol, the LPs execute in *super-steps*, alternating between execution (during which events are simulated) and barrier synchronization (at which point information is exchanged among the LPs).

The performance analyzer is implemented as a "plug-in" module to a sequential simulator. The events simulated in the sequential simulator are fed into the performance analyzer to predict the performance of the parallel conservative super-step protocol as the sequential simulation progresses.

Our simulation model is for wafer fabrication plants and is based on the Sematech datasets [3]. The parallel simulator uses the Active Threads library [4] as part of its runtime system. A thread is created to run each LP. Multiple simulation objects (e.g. machine sets) in the model are grouped into one LP. Both the sequential and parallel simulators are written in C++ using GNU g++ version 2.7.2.1. Our timings were measured on a four-processor Sun Enterprise 3000, 250 MHz Ultrasparc 2.

3 Issues for Accurate Predictions

Code equivalence and system effects are two implementation issues to be considered in order to obtain accurate results from the performance analyzer. To achieve code equivalence between the parallel simulator and the sequential simulator, we had to make the following changes:

(1) To get the same set of events as in a parallel implementation, the sequential random number generator in the sequential implementation is replaced by the parallel random number generator used in the parallel implementation.
(2) A technique known as *pre-sending* [2] is used in the parallel simulation to allow for more parallelism in the model. If the sequence of events in the sequential simulator is such that event E_1 generates $E2$ which in turn generates $E3$, pre-sending may allow event E_1 to generate E_2 and E_3 simultaneously. The event execution time of E_1 with pre-sending may be different from the non-presend version. We modified the sequential simulator to use the pre-sending technique.
(3) Our sequential code has a global event pool manager managing event objects allocation and deallocation. The parallel code initially has one local event pool manager associated with each simulation object. We modified the parallel code to have one event pool for each LP. (A global event pool would have introduced additional synchronization in the parallel code.) Table 1a shows that external cache miss rates for datasets 2 and 4 are reduced. The corresponding speedups are also improved (Figure 1a). The actual speedup with one event pool per LP is now closer to the predicted trend.

There are two sources of system effects that affect the performance of a parallel execution:(a) cache effects (b) synchronization mechanisms used.

Cache effects Our parallel implementation uses the Active Threads library [4] facilities for multi-threading and synchronization. The library has a self load-balancing mechanism in which an idle processor will look into another processor's thread queue and, if possible, try to "steal" a thread (and its LP) to run. Disabling the thread-stealing improves the cache miss rates (Table 1b) and brings the actual speedup curve closer to the predicted one (Figure 1b).

Program synchronization At the end of each super-step in the simulation protocol, all the LPs are required to synchronize at a barrier. We experimented with two barrier implementations: (a) using semaphore and (b) using spinlock.

Data set	1	2	3	4	5	6
Parallel (One event pool per simulation object)	6.4%	8.4%	10.9%	12.2%	7.2%	7.1%
Parallel (One event pool per LP)	6.0%	6.3%	11.9%	12.2%	5.9%	7.3%
Sequential	0.36%	0.29%	0.18%	0.12%	0.50%	0.61%

(a)

Data set	1	2	3	4	5	6
Parallel (1 event pool per LP, thread stealing on)						
References	373.8×10^6	820.7×10^6	1627.1×10^6	83.9×10^6	368.4×10^6	344.4×10^6
Hits	332.5×10^6	735.4×10^6	1378.1×10^6	69.5×10^6	329.2×10^6	305.9×10^6
Miss %	11.1	10.4	15.3	17.2	10.6	11.2
Parallel (1 event pool per LP, thread stealing off)						
References	379.3×10^6	730.2×10^6	1736.3×10^6	79.3×10^6	328.5×10^6	382.0×10^6
Hits	356.6×10^6	684.1×10^6	1529.3×10^6	69.7×10^6	309.0×10^6	354.0×10^6
Miss %	6.0	6.3	11.9	12.2	5.9	7.3

(b)

Table 1. External cache miss rates for (a) parallel implementation using one event pool per simulation object, one event pool per LP, and sequential execution; (b) parallel implementation when thread-stealing mechanism in the runtime system is turned on or off.

We estimated the time of each barrier from separate template programs. The estimate is 35 μs for a "semaphore barrier" and 6 μs for a "spinlock barrier". The total synchronization cost is obtained from multiplying the per-barrier time by the number of super-steps that each dataset uses. Figure 1c shows the predicted and actual speedup curves (with thread-stealing disabled) for both barriers. The predicted and actual speedup curves for "semaphore barrier" match quite closely. There is however still a discrepancy for the corresponding curves for "spinlock barrier". Our guess is that the template program under-estimates the time for a "spinlock barrier" in a real parallel program.

4 Conclusion

This paper describes the implementation lessons learnt when we tried to match the trends in a predicted and an actual speedup curve on a 4-processor Sun shared-memory multiprocessor. The main implementation issue to take note of is that the code actions in a sequential program should be comparable to what would actually happen in the corresponding parallel program. The lessons learnt was put to good use in implementing a new performance analyzer for a conservative, *asynchronous* protocol. Also, in trying to get the predicted and actual curves to match, the parallel simulation program was further optimized.

References

[1] C.-C. Lim, Y.-H. Low, and W. Cai. A parallelism analyzer algorithm for a conservative super-step simulation protocol. In *Hawaii International Conference on System Sciences (HICSS-32)*, Maui, Hawaii, USA, January 5–8 1999.

[2] D. Nicol. *Problem characteristics and parallel discrete event simulation*, volume Parallel Computing: Paradigms and Applications, chapter 19, pages 499–513. Int. Thomson Computer Press, 1996.

[3] Sematech. Modeling data standards, version 1.0. Technical report, Sematech, Inc., Austin, TX78741, 1997.

[4] B. Weissman and B. Gomes. Active threads: Enabling fine-grained parallelism. In *Proceedings of 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, Nevada USA, July 13 – 16 1998.