プログラミング言語 15－1
（１９８８．２．１９）

並列プログラムのデバッグと性能評価

大上　貴英

三菱電機株式会社　情報電子研究所
〒２４７　鎌倉市大船５－１－１

一般に，並列プログラムのデバッグと性能評価は逐次プログラムのそれよりも難しい．その理由の１つには，並列プロセスに発生するイベントが非同期であることがある．本稿では，共有オブジェクトへの操作に注目し，統一的にデバッグと性能評価が行える手法について述べる．この手法のオリジナルなアイデアは，Ｔ．Ｊ．ＬｅＢｌａｎｃと　Ｊ．Ｍ．Ｍｅｌｌｏｒ－Ｃｒｕｍｍｅｙによって提案されたＩｎｓｔａｎｔ　Ｒｅｐｌａｙで開発されたものである．これは並列プログラムの実行をリプレイする手段を提供するもので，従来より逐次プログラムの開発に用いられているサイクリックなデバッグを使うことができる．Ｉｎｓｔａｎｔ　Ｒｅｐｌａｙで開発されたアイデアは自然な形で性能評価にも拡張して使える．試作した並列プログラムの性能解析ツール，および，その使用経験についても報告する．

A Unified Approach to Debugging and Performance Evaluation of Parallel Programs

Takahide Ohkami

Information Systems and Electronics Development Laboratory
Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura City, Japan 247
Network Address: ohkami%isvax.islab.melco.junet@uunet.uu.net

Abstract

In general, tasks of debugging and performance evaluation of parallel programs are more difficult than those for sequential ones. One of the reasons lies in asynchronous occurrences of events to parallel processes. This paper describes a unified approach to debugging and performance evaluation of parallel programs, focusing on operations on shared objects. The original idea was developed by T. J. LeBlanc and J. M. Crummey for Instant Replay, which provides repeatable executions of a parallel program for cyclic debugging. It was extended to the evaluation of parallel performance. Some results from experiments are also described.

1. Introduction

Parallel computing systems are now available for a variety of applications: for example, BBN Butterfly [14], Sequent Balance [8], and Alliant FX [13]. Programmers write parallel programs for these systems to exploit their potential parallel execution capability. Parallel programs use parallel processes running in parallel on different processors. Parallel processes usually coordinate to complete a single job. Process interactions are under control of some synchronization mechanisms, such as semaphores and monitors. This is a typical scenario for parallel programming.

We have been involved in sequential programming for three decades and have compiled many know-hows and tools for that, although there are still many issues to be addressed. However, parallel programming is relatively new to us, and we do not have much knowledge and experience. If it were not much different from sequential programming, there would be no problems; conventional techniques developed for sequential programming would suffice for parallel programming. It becomes a new awareness that parallel programming presents new problems that are not found in sequential programming. As pointed out by Carver and Tai in [4], considerable research efforts have been devoted to the design and implementation of parallel systems; few efforts have concentrated on debugging and performance evaluation of parallel programs. We focus on these areas in this paper.

Debugging parallel program is more difficult than debugging sequential programs, because asynchronous events occur to parallel processes running on different processors. The execution status of one run of a parallel program is not always the same as that of the other run of the same program; that is, program execution is not always repeatable. It implies that an error found in a run of a program may not be found in another run, making a debugging task hard. In sequential programming, programmers usually go through several debugging cycles to remove errors. This conventional cyclic debugging technique is based on the repeatability of program execution. The problem is that this assumption does not hold in parallel execution.

The main purpose of parallel programming is the increased performance. In general, however, it is not necessarily an easy task to achieve high performance with parallel programming. Parallel performance heavily depends on the efficiency of coordinations of parallel processes in many cases; inefficient coordinations waste time for interprocess communications, which would be spent for making more progress. In order to achieve good parallel performance, programmers also go through several iterations of code improvement. One of the issues in parallel performance evaluation is to find an efficient way to obtain parallel execution profile so that programmers can easily find bottlenecks in their programs.

LeBlanc and Mellor-Crummey proposed a new method for debugging parallel programs, termed Instant Replay [10]. It provides repeatable execution of parallel programs and allows programmers to use the conventional debugging technique for debugging parallel programs. It focuses on the objects shared by parallel processes, which are guarded by some synchronization mechanisms. It records the history of accesses to each shared object in the monitoring phase (the first run of a program), and controls the accesses to shared objects to keep the recorded relative order of accesses in the replay phase (the subsequent runs of the program). This idea can be natually extended for performance evaluation of parallel programs. The record of the accesses to shared objects clearly shows a parallel execution profile, which can be used to pinpoint the bottlenecks in parallel execution. Therefore, we can use a single unified model of parallel execution for debugging and performance evaluation of parallel programs.

In the next section we discuss debugging and performance problems and previous work to address some of them. Section 3 gives a brief review of Instant Replay. Section 4 describes a method for performance evaluation of parallel programs, based on the idea of Instant Replay. In Section 5 we present the prototype performance evaluation tools developed for BBN Butterfly processors and some experience with the tools. Section 6 summarizes the advantages of our approach and describes our plan for future work.

2. Problems

2.1. Debugging Problems

Parallel processing is provided with parallel processes in many computing systems; parallel programming specifies how parallel processes coordinate to complete a single job. It is reasonable that no assumption is made about the relative speed of parallel processes; we only assume finite progress by each process [10]. Therefore, parallel programs are not necessarily deterministic, and hence, debugging parallel programs is more difficult than debugging sequential programs, which is based on the determinstic nature of sequential programs. Non-determinism does not guarantee that executions of parallel programs always produce the same results for the same input. In the successive executions, one may not be able to find the error which occurred in a previous execution. Thus, it complicates debugging tasks for parallel programs. This issue has been addressed by two approaches: snapshot and repeatable execution appoaches.

The first approach selectively takes snapshots of program states during execution for later analysis, with a focus on the program behavior. Bates and Wileden proposed the *behavioral abstraction* approach which monitors the specified events that are hierarchically defined in terms of primitive events and provides an abstract view of the program behavior [3]. Baiardi, DeFrancesco, and Vaglini developed a debugger for a concurrent language, which makes it possible to compare the expected behavior and the actual behavior, using the description of program behavior [1]. Chandy and Lamport proposed an algorithm to determine the global state of a distributed computing system from the recorded process states and messages exchanged among processes [5]. One of the major disadvantages of the snapshot approach is that the snapshots taken in a program execution do not necessarily indicate the general behavior of the program, but merely the behavior in the single execution. Another disadvantage is that it is very difficult to specify in advance all the events which may cause errors. Yet another disadvantage is that the amount of information collected during execution tends to be large.

The second approach provides repeatable executions of a program. This is a relatively new approach, while the snapshot approach is basically an extension to the sequential snapshot approach. Carver and Tai proposed a method to reporoduce a sequence of synchronizations for testing parallel programs, focusing on shared variables, like semaphores and monitors, to control synchronizations [4]. Chu proposed a technique to replay program execution in atomic transaction systems, which checkpoints each version of atomic objects and records a timestamp for each atomic action [6]. LeBlanc and Mellor-Crummey proposed *Instant Replay* [10], which will be reviewed in the next section. One of the advantages of the repeatable execution approach is that it allows programmers to go through conventional debugging cycles and to use the debugging tools developed for sequential programming.

2.2. Performance Problems

We expect to speedup our programs by using parallel systems. However, the *Fundamental Law of Parallel Computation* states that a parallel solution utilizing P processors can improve the best sequential solution by at most a factor of P [15]. Current parallel systems only offer low-order polynomial parallelism; the number of processors is bounded by a low-order polynomial. On the other hand, many problems to be solved on these parallel systems require medium- to high-order polynomial parallelism. Thus, there is a gap between the requirements and the actual capability of parallel systems. All we can do is to exploit as much parallelism of parallel systems as possible by reducing overhead. Parallelism offered by current parallel systems is so elusive that overhead easily eats it up. Usually, programming is the main source of overhead.

The primary motivation of parallel programming is the high performance obtained by parallel processing. It seems obvious that we obtain higher performance with parallel programming. However, this is not always the case in reality. Inefficient parallel programming incurs too much overhead to exploit parallelism or misses the potential capability of parallel systems; parallel performance may be even worse than sequential performance in some cases.

To our best knowledge, there is no widely accepted general programming method to obtain the best performance from any parallel computing system. Therefore, programmers resort to a conventional method that has been used for sequential programming; they go through performance-tuning cycles to improve their programs until the expected performance is achieved.

When we don't obtain high performance in executing a parallel program, we may call it a performance *error*. Then, we may think of performance tuning as part of program debugging. However, we don't know yet how to describe the expected performance as part of the program specification. This is compared with the situation in the hardware area, where the notion of *timing error* is now widely accepted. If an integrated circuit doesn't work with the expected performance, it is defined as a *delay fault* [9]. Current hardware design systems deal with this type of faults (errors). More research efforts are required to deal with performance error in software.

One of the previous research efforts on parallel performance evaluation is the Miller's [11]. He developed a system for monitoring the behavior of distributed programs, which is used to measure the amount of parallelism that occurs in the execution of a distributed program. His model, called the *DPM (Distributed Programs Monitor)*, traces the history of selected events in the life of the processes in a program, where the *selected events* are those that have blocking dependencies between processes for synchronizations. From the history of the events, one can construct a program history graph that shows the process interactions with timing data. Programmers can check the performance problems with the graph.

3. Instant Replay - A Review

Instant Replay is a debugging approach proposed by LeBlanc and Mellor-Crummey [10]. Its main goal is to provide repeatable execution of parallel programs and to allow programmers to go through conventional debugging cycles to remove programming errors. Its prototype was implemented for parallel programs written for BBN Butterfly processors. The following is a brief review of Instant Replay.

Instant Replay models all interactions among parallel processes as operations on shared objects. Each shared object is associated with a version, which is updated by write operations, but not by read operations. Operations on shared objects form a sequence of versions, which is totally ordered. Actually, it is partially ordered since we do not need to impose an ordering on multiple processes that read a particular verion of each object. From the recorded sequence of write operations on each shared object, it is possible to reconstruct the proper sequence of state changes for all shared objects. Similarly, from the recorded version number of each shared object read by a process, it is possible to reconstruct the proper input values for that process. Therefore, by recording the sequence of versions of each shared object accessed by each process during the execution of a parallel program, one can replay the execution of a parallel program. Since program execution can be repeated without recording data generated during execution, the volume of recorded information is very small.

Instant Replay requires that the set of operations on each shared object has a valid serialization, which is achieved if the result of each individual operation is the same as it would be if the operations had all been executed in some sequential order. Every process has to use a protocol that ensures a valid serialization for access to each shared object. A CREW (Conccurent Read and Exclusive Write) protocol [7] is a good example. It ensures a total order of writers with respect to each shared object, a total order of readers with respect to writers of each shared object, and a partial order of readers with respect to each shared object.

There are two phases in Instant Replay: monitoring and replay phases. The monitoring phase is the very first execution of a parallel program. In this phase all the accesses to shared objects by a process are recorded in the history tape of the process, and the version number of each shared object is updated by

```
struct {
 lock;          /* for P and V */
 version;       /* object version */
 a_readers;     /* # active readers */
 t_readers;     /* # total readers */
 /* other data */
} object;
```
Figure 1. Structure of Shared Object.

```
Entry_Read (object, process) {
 if (mode == MONITOR) then
  P(object.lock);
  Add_Atomic(object.a_readers, 1);
  V(object.lock);
  Tape_Write(process, object.version);
 else /* in replay mode */
  key = Tape_Read(process);
   while (object.version != key) do delay;
 end_if;
}
Exit_Read (object) {
 Add_Atomic(object.t_readers, 1);
 if (mode == MONITOR) then
  Add_Atomic(object.a_readers, -1);
 end_if;
}
```
Figure 2. Entry and Exit Procedures for Readers.

```
Entry_Write (object, process) {
 if (mode == MONITOR) then
  P(object.lock);
  while (object.a_readers != 0) do delay;
   Tape_Write(process, object.version);
   Tape_Write(process, object.t_readers);
  else /* in replay phase */
   key = Tape_Read(process);
   while (object.version != key) do delay;
   key = Tape_Read(process);
   while (object.t_readers < key) do delay;
 end_if;
}
Exit_Write (object) {
 object.t_readers = 0;
 if (mode == MONITOR) then
  object.version += 1;
  V(object.lock);
 else /* in replay mode */
  Add_Atomic(object.version, 1);
 end_if;
}
```
Figure 3. Entry and Exit Procedures for Writers.

every write operation. The replay phase is the subsequent execution of the same program, which is the replay of the first execution. In this phase the information recorded in the history tapes is used to control accesses of each process to shared objects so that the first execution be repeated.

We now show a CREW access protocol for shared objects. It uses four procedures: entry and exit procedures for readers, and those for writers. Figure 1 shows the structure of shared objects; Figures 2 and 3 show entry and exit procedures for readers and those for writers, respectively. Each process has to use the entry procedure when it reads/writes a shared object and the exit procedure when it finishes reading/writing. These procedures use semaphore operations (P and V) and atomic add operations (Add_Atomic) for synchronizations.

4. Parallel Performance Evaluation

In Instant Replay the behavior of parallel programs is determined by the interactions among parallel processes, which are modeled as operations on shared objects. This basic idea is naturally extended for parallel performance evaluation; the timings of operations on shared objects are measured [12]. Measurements reflect the efficiency of synchronizations in parallel programs. If a programmer finds unreasonably long synchronizations, he can improve his program to shorten them.

We focus on process interactions, because they are the main source of difficulties in debugging and performance evaluation of parallel programs; the difficulties that are also found in sequential programming are not considered here, but left to the conventional techniques developed for sequential programming over years.

For performance evaluation, we model a single read/write operation on a shared object as a sequence of three procedures: the *entry*, *work*, and *exit* procedures. The entry and exit procedures are the same as defined for a CREW protocol in the previous section. The work procedure performs the user-specified operation on the shared object. We measure the timimgs for the entry and work procedures in each parallel process; the exit procedure usually executes so quickly that we don't need to measure it. The timings are recorded in the history tape of each process.

We now show a data structure for performance evaluation. Every time a parallel process accesses a shared object, the following 6-tuple is recorded:

(soid, vrsn, opid, stime, etime, wtime),

where

soid = the shared object ID;
vrsn = the version number of the shared object;
opid = the (read/write) operation ID;
stime = the start time of the operation;
etime = the time spent by the entry procedure;
wtime = the time spent by the work procedure.

5. Parallel Performance Analysis Tools

Based on the idea described in the previous section, we implemented a prototype version of parallel performance analysis tools, called the PARPAT (PARallel Performance Analysis Tools), for performance analysis of parallel programs written for BBN Butterfly processors [12].

We model an access to a shared object as the sequence of three operations: the *entry*, *work*, and *exit* operations, as described in the previous section. The definitions of these operations are basically left to the user; typical *entry* and *exit* operations for a CREW access protocol are provided through a library. The user has to define the *work* operation.

A parallel program to be monitored has to be instrumented with the monitor procedures, which take care of creation and deletion of data structures and recording performance data. The main procedures for users are as follows:

- **mon_init()**
- **mon_fin()**
- **mon_process()**
- **mon_sobj()**
- **mon_sobj_access()**

The **mon_init()** procedure opens the data structures for performance monitoring and initializes them. It must be called at the beginning of the master process. Note that we assume that every parallel program begins execution with a single master process, which eventually spawns parallel child processes. The **mon_fin()** procedure closes the data structures; the data blocks storing performance data are retained after execution of the program, and the others are deleted. The **mon_process()** procedure stores the ID numbers of newly created parallel processes in the process table. Similarly, the **mon_sobj()** procedure stores the ID numbers of newly created shared objects in the shared object table. The **mon_sobj_access()** procedure takes the *entry*, *work*, and *exit* procedures and invokes them in that order for the specified shared object. The *stime*, *etime*, and *wtime* are measured and recorded within this procedure. The recorded performance data are stored in the data blocks (tape) of each process and retrieved after program execution. A print procedure is provided for display of the data.

Figure 4 shows an example of a simple parallel program (producer-consumer program). In this example, the *master* process spawns *producer* and *consumer* processes and communicate with them through the *sobj-1*; the *producer* and *consumer* processes communicate with each other through the *sobj-2*.

Figures 5 and 6 illustrate the **master()** and **child()** procedures, respectively. The *producer* and *consumer* processes share the code of the **child()** procedure. After creating two shared objects and spawing two processes, the master process starts the two processes by a write operation on *sobj-1* by using the first **mon_sobj_access()** procedure and then performs a read operation by using the second **mon_sobj_access()** procedure to check the end of the activities of the two processes. The *producer* and *consumer* processes produce and consume, respectively, items by using *sobj-2*. The *ew*, *ww*, and *xw* in the code are some *entry*, *work*, and *exit* procedures for write access; similarly, the *er*, *wr*, and *xr* are those for read access.

Figure 7 shows part of displayed performance

```
                Master
                  |
  +----------- sobj-1 --------+
  |                           |
Producer ----- sobj-2 ----- Consumer
```

Figure 4. Producer-Consumer Program.

```
master() {
 mon_init();
 create sobj-1 and sobj-2;
 mon_sobj(sobj-1);
 mon_sobj(sobj-2);
 create producer and consumer;
 mon_process(producer);
 mon_process(consumer);
 mon_sobj_access(sobj-1, ew, ww, xw);
 mon_sobj_access(sobj-1, er, wr, xr);
 mon_fin();
}
```

Figure 5. Producer-Consumer Program Master Code.

```
child() {
 mon_init();
 mon_sobj_access(sobj-1, er, wr, xr);
 if this is the producer process
 then producer();
 else consumer();
 mon_sobj_access(sobj-1, ew, ww, xw);
 mon_fin();
}
producer() {
 while there are still items to produce
  mon_sobj_access(sobj-2, ew, ww, xw);
}
consumer() {
 while there are still items to consume {
  mon_sobj_access(sobj-2, ew, ww, xw);
}
```

Figure 6. Producer-Consumer Program Child Code.

data for the producer-consumer program. It shows the recorded accesses on two shared objects with times; the 8331aa object corresponds to *sobj-1* and the dc30ce to *sobj-2*. All timings are measured in ticks with the local clock. The *OPID* or Operation ID is the number assigned to each access operation, which is passed to the **mon_sobj_access()** as an argument for easy identification of the operation in performance analysis.

```
******* MB: Producer
MB       Object ID: 0x0114356a
Process Object ID: 0x01682a7c
Process User   ID: 0x00000200
Processor :       0x0001
Process Start Time: 0xa15057db (ticks)
Operations on Shared Objects:
Time  R/W OPID SOID   VRSN ENTRY WORK
----- --- ---- ------ ---- ----- ----
  289  R  0200 8331aa 0001   977    1
 1598  W  0201 dc30ce 0001     2    2
 1966  W  0202 dc30ce 0002     2    1
 2323  W  0203 dc30ce 0003     2    2
 2691  W  0204 dc30ce 0004     2    1
 3025  W  0205 dc30ce 0006     2    2
 3393  W  0206 dc30ce 0008  1255    2
 4980  W  0207 dc30ce 000a  1282    1
              . . . . . . . .
19687  W  0210 dc30ce 001c  1302    1
20995  W  0211 8331aa 0002     2    2
Processor Life Time: 21037 (ticks)
Processes Created: None
Shared Objects Created: None
Operations on Shared Objects (Summary):
SOID   R/W #Ops Total ENTRY WORK
------ --- ---- ----- ----- ----
8331aa  R     1   978   977    1
dc30ce  W    16 14131 14108   23
8331aa  W     1     4     2    2
```

Figure 7. Displayed Performance Data.

As a major experiment using the PARPAT, we implemented the odd-even sort [2], based on the butterfly connection. This parallel sort algorithm is well described in Chapter 6 of [16]. We ran the parallel sort program instrumented with the monitor procedures for data sizes 1024, 2048, and 4096 using 2, 4, 8,16, 32, 64, and 128 processes/processors on a Butterfly processor.

When we checked the performance data for various runs of the parallel sort program , we found that the parallel performance got better as the number of processors increased, but in the range of up to 64 processors. If the program used more than 64 processors, then the performance decreased as the number of processors increased. When the program used 128 processors, the parallel performance was much worse than the sequential performance.

After careful analysis of the monitored performance data, we found the problem. The problem was the start signal, which was sent by the master process to all the child processes to start their tasks. When the program used 64 processors, the longest time for a child to get the signal was 80-90% of its lifetime. Without the PARPAT we could not have found this problem. We were very surprized to see the importance and usfulness of the parallel performance analysis tools.

6. Conclusion

We have presented a unified approach to debugging and performance evaluation of parallel programs. It models the behavior of parallel programs as operations on shared objects. It allows progrmmers to go through conventional debugging and performance-tuning cycles to remove programming errors and to obtain high performance with the help of the tools developed for sequential programming.

For future, we plan to refine our approach for efficient implementation and to efficiently combine Instant Replay and PARPAT. On the other hand, we think we need to investigate the advantanges and disadvantages of our approach for a variety of contexts in more depth.

Acknowledgements

This work was done while the author stayed at Department of Computer Science, University of Rochester (Rochester, New York), from the summer of 1985 to the spring of 1987. The author is grateful to Prof. R. Fowler, Prof. T. LeBlanc, and their Ph. D. students, especially J. Mellor-Crummey, for fruitful suggestions and discussions.

References

[1] F. Baiardi, N. DeFrancesco, and G. Vaglini, *Development of a Debugger for a Concurrent Language*, IEEE Trans. Software Engineering, Vol.SE-12, No.4, Apr. 1986, pp.547-553.
[2] K. E. Batcher, *Sorting Networks and Their Applications*, Proc. Spring Joint Computer Conf., 1968, pp.307-314.
[3] P. Bates and J. Wileden, *High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, TR COINS 83-29, Depart. CIS, Univ. Massachusetts, 1983.
[4] R. H. Carver and K. C. Tai, *Reproducible Testing of Concurrent Programs Based on Shared Variables*, Proc. 6th Int. Conf. Distributed Computing Systems, 1986, pp.428-433.
[5] K. M. Chandy and L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. Computer Systems, Vol.3, No.1, Feb. 1985, pp.63-75.
[6] S. Y. Chu, *Debugging Distributed Computations in a Nested Atomic Action System*, MIT/LCS/TR 327, Depart. EECS, MIT, 1984.
[7] P. J. Courtois, F. Heymans, and D. L. Panas, Concurrent Control with Readers and Writers, Comm. ACM, Vol.14, No.10, Oct. 1971, pp.667-668.
[8] G. Fielland and D. Rodgers, *32-Bit Computer System Shares Load Equally Among up to 12 Processors*, Electronic Design, Sept. 6, 1984, pp.153-168.
[9] E. P. Hsieh, R. A. Rasmussen, L. J. Vidunas, and W. T. Davis, *Delay Test Generation*, Proc. 14th Design Automation Conf., 1977, pp.486-491.
[10] T. J. LeBlanc and J. M. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay*, IEEE Trans. Computers, Vol.C-36, No.4, Apr. 1987, pp.471-482.
[11] B. P. Miller, *Parallelism in Distributed Programs: Measurement and Prediction*, TR 574, Depart. CS, Univ. Wisconsin at Madison, May 1985.
[12] T. Ohkami, *PARPAT: Parallel Performance Analysis Tools*, TM, Depart. CS, Univ. Rochester, Apr. 1987.
[13] R. Perron and C. Mundie, *The Architecture of the Alliant FX/8 Computer*, Proc. COMPCON Spring, 1986, pp.390-393.
[14] Q. E. Schmidt, *The Butterfly Parallel Processor*, Proc. 2nd Int. Conf. Supercomputing, 1987, pp.362-365.
[15] L. Snyder, *Type Architectures, Shared Memory and the Corollary of Modest Potential*, TR 86-03-04, Depart. CS, Univ. Washington, March 1986.
[16] J. D. Ullman, *Computational Aspects of VLSI*, Computer Sience Press, 1984.