A Model for Coherent Distributed Memory For Race Condition Detection

Franck Butelle
Franck.Butelle@lipn.univ-paris13.fr
Camille Coti
Camille.Coti@lipn.univ-paris13.fr
LIPN, CNRS-UMR7030, Université Paris 13, F-93430 Villetaneuse, France

***Abstract*—We present a new model for distributed shared memory systems, based on remote data accesses. Such features are offered by network interface cards that allow one-sided operations, remote direct memory access and OS bypass. This model leads to new interpretations of distributed algorithms allowing us to propose an innovative detection technique of race conditions only based on logical clocks. Indeed, the presence of (data) races in a parallel program makes it hard to reason about and is usually considered as a bug.**

I. Introduction

The *shared-memory model* is a convenient model for programming multiprocessor applications: all the processes of a parallel application running on different processors have access to a common area of memory. Another possible communication model for distributed systems is the *message-passing model*, in which each process can only access its own local memory and can send and receive message to other processes.

The message-passing model on distributed memory requires to move data between processes to make it available to other processes. Under the shared-memory model, all the processes can read or write at any address of the shared memory. The data is *shared* between all the processes.

One major drawback of the shared-memory model for practical situations is its lack of scalability. A direct implementation of shared memory consists in plugging several processors / cores on a single motherboard, and letting a single instance of the operating system orchestrate the memory accesses. Recent blades for supercomputers gather up to 32 cores per node, Network on Chip (NoC) systems embed 80 cores on a single chip: although the "many-core" trend increased drastically the number of cores sharing access to a common memory bank, it is several orders of magnitude behind current supercomputers: in the Top 500[1] list issued in November 2010, 90% of the systems have 1K to 16K cores each.

The solution to benefit from the flexibility and convenience of shared memory on distributed hardware is *distributed shared memory*. All the processes have access to a *global address space*, which is distributed over the processes. The memory of each process is made of two parts: its *private* memory and its *public* memory. The private memory area can be accessed from this process only. The public memory area can be accessed remotely from any other process without notice to the process that maps this memory area physically.

The notion of global address space is a key concept of parallel programming languages, such as UPC [1], Titanium [2] or Co-Array Fortran [3]. The programmer sees the global memory space as if it was actually shared memory. The compiler translates accesses to shared memory areas into remote memory accesses. The run-time environment performs the data movements. As a consequence, programming parallel applications is much easier using a parallel language than using explicit communications (such as MPI [4]): data movements are determined by the compiler and handled automatically by the run-time environment, not by the programmer himself.

The memory consistency model followed by these languages, such as the one defined for UPC [5], does not define a global order of execution of the operations on the public memory area. As a consequence, a parallel program defines a set of possible executions of the system. The events in the system may happen in different orders between two consecutive executions, and the result of the computation may be different. For example, if a process writes in an area of shared memory and another process reads from this location. If the writer and the reader are two different processes, the memory consistency model does not specify any kind of control on the order in which these two operations are performed. Regarding whether the reader reads before or after the data is written, the result of the writing may be different.

In this paper, we introduce a model for distributed shared memory that represents the data movements and accesses between processes at a *low* level of abstraction. In this model, we present a mechanism for detecting race conditions in distributed shared memory systems.

This model is motivated by Remote Direct Memory Access capabilities of high-speed, low-latency networks used for high-performance computing, such as the InfiniBand

[1] http://www.top500.org

standard[2] or Myrinet[3].

The remainder of this paper is organized as follows. In section II, we present an overview of previous models for distributed shared memory and how consistency and coherency has been handled in these models. In section III we present our model for distributed shared memory and how it can be related to actual systems. In section IV we present how race conditions can be represented in this model, and we propose an algorithm for detecting them.

II. Previous work

Distributed shared memory is often modeled as a large cached memory [6]. The local memory of each node is considered as a cache. If a process running on this node tries to access some data, it gets it directly if the data is located in its cache. Otherwise, a page fault is raised and the distributed memory controller is called to resolve the localisation of the data. Once the data has been located (*i.e.*, once the local process knows on which process it is physically located and at which address in its memory), the communication library performs a point-to-point communication to actually transfer the data.

In [7], L. Lamport defines the notion of *sequential consistency*: on each process, memory requests are issued in the order specified by the program. However, as stated by the author, sequential consistency is not sufficient to guarantee correct execution of multiprocessor shared memory programs. The requirement to ensure correct ordering of the memory operations in such a distributed system is that a single FIFO queue treats and schedules memory accesses from all the processes of the system.

Maintaining the coherence of cache-based distributed shared memory can then be considered as a cache-coherency problem. [8] describes several distributed and centralized memory managers, as well as how coherence can be maintained using these memory managers.

However, in a fully distributed system (*i.e.*, with no central memory manager) with RDMA and OS bypass capabilities, a process can actually access another process's memory without help from any memory manager. In parallel languages such as UPC [1], Titanium [2] and Co-Array Fortran [3], data locality (*i.e.*, which process holds the data in its local memory) is resolved at compile-time.

The MPI-2 standard [9] defines remote memory access operations. The MARMOT error checking tool [10] checks correct usage of the synchronization features provided by MPI, such as fences and windows.

III. Memory and communication model

In this section, we define a model for *distributed shared memory*. This model works at a lower level than most models described previously in the literature. It considers inter-process communications for remote data accesses.

A. Distributed shared memory model

In many shared-memory models that have been described in the literature [11], [12], [13], pairs of processors communicate using registers where they read and write data. Distributed shared memory cannot use registers between processors because they are *physically* distant from each other; like message-passing systems, they can communicate only by using an interconnection network.

Figure 1 depicts our model of organization of the public and private memory in a multiprocessor system. In this model, each processor maps two distinct areas of memory: a *private* memory and a *public* memory. The private memory can be accessed from this processor only.

The public address space is made of the set of all the public memories of the processors (the *Global Address Space*). Processors can copy data from/to their private memory and the public address space, regardless of data locality.

Public memory can be accessed by any processor of the application, in *concurrent* read and write mode. In particular, no distinction is made between accesses to public memory from a remote process and from the process that actually maps this address space.

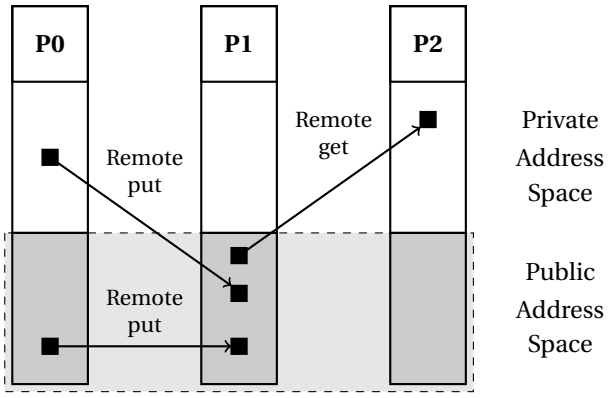


Figure 1: Memory organization of a three-processor distributed shared memory system.

The compiler is in charge with data locality, *i.e.*, putting shared data in the public memory of processors. For instance, if a data x is defined as shared by the programmer, the compiler will decide to put it into the memory of a processor P. Instead of accessing it using its address in the local memory, processors use the processor's name and its address in the memory of this processor. This couple $(processor\_name, local\_address)$ is the addressing system used in the global address space. The compiler also makes the address resolution when the programmer asks a processor to access this shared data x.

[2] http://www.infinibandta.org/
[3] http://www.myri.com

In addition, since NICs (Network Interface Controllers) are in charge with memory management in the public memory space, they can provide *locks* on memory areas. These locks guarantee exclusive access on a memory area: when a lock is taken by a process, other processes must wait for the release of this lock before they can access the data.

B. Communications

Processor access areas of public memory mapped by other processors using point-to-point communications. They use *one-sided communications*: the process that initiates the communication can access remote data without any notification on the other processor's side. Hence, a processor A is not aware of the fact that another processor B has accessed (*i.e.*, read or written) in its memory.

Accessing data in another processor's memory is called *Remote Direct Memory Access* (RDMA). It can be performed with no implication from the remote processor's operating system by specific network interface cards, such as InfiniBand and Myrinet technologies. It must be noted that the operating system is not aware of the modifications in its local shared memory. The SHMEM [14] library, developed by Cray, also implements one-sided operations on top of *shared* memory. As a consequence, the model and algorithms presented in this paper can easily be extended to shared memory systems.

RDMA provides two communication primitives: *put* and *get*. These two operations are represented in figure 2. They are both *atomic*.

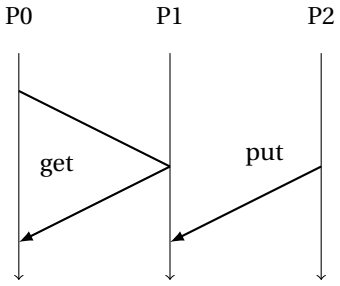


Figure 2: Remote R/W memory accesses.

Put consists in writing some data into the public memory of another processor. It involves one message, from the source processor to the destination processor, containing the data to be written. In figure 2, $P2$ writes some data into $P1$'s memory.

Get consists in reading some data from another processor's public memory. It involves two messages: one to request the data, from the requesting processor to the processor that holds the data, and one to actually transfer the data, from the processor that holds the data to the requesting processor. In figure 2, $P0$ reads some data from $P1$'s memory.

Communications can also be done within the public space, when data is copied from a place that has affinity to a process to a place that has affinity to another process.

The *get* operation is *atomic* (and therefore, blocking). If a thread gets some data and writes it in a given place of its public memory, no other thread can write at this place before the *get* is finished. The second operation is delayed until the end of the first one (figure 3).

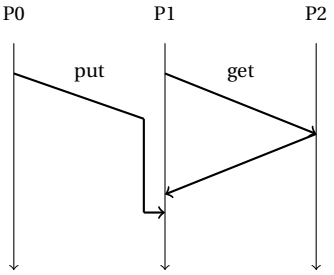


Figure 3: A put operation is delayed until the end of the get operation on the same data.

C. Race conditions

One major issue created by one-sided communications is that several processors can access a given area of memory without any synchronization nor mutual knowledge. For example, two processors A and B can write at the same address in the shared memory of a third processor C. Neither B nor C knows that A has written or is about to write there.

Concurrent memory accesses can lead to *race conditions* if they are performed in a totally anarchic way (although some authors precise *data* race conditions, we will use only "race conditions" throughout this paper). A race condition is observed when the result of a computation differs between executions of this computation. Race condition makes, at least, hard to reason about a program and therefore is usually considered as a bug.

In the kind of systems we are considering here, a race condition can occur when several operations are performed by different processors on a given area of shared memory, and at least one of these operations is a write.

For instance, if a piece of data located in the shared memory is initialized at a given value v_0 and is accessed concurrently by a process A that reads this data and a process B that writes the value v_1. If A reads it before B writes, it will read the value v_0. If B writes before A reads, A will read v_1.

More formally, we can consider read and write operations as *events* in the distributed system formed by the set of processors and the communication channels that interconnects them.

Two events e_1 and e_2 are *ordered* iff there exists an *happens before* (as defined by [15] and denoted $\rightarrow$)

relationship between them such that $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$. Race conditions are defined in [16] by the fact that there exists no causal order between e_1 and e_2 (further denoted by $e_1 \times e_2$).

IV. Detecting race conditions

In this section, we present an algorithm for detecting race conditions in parallel applications that follow the distributed shared memory model presented in section III.

A. Causal ordering of events

In section III-C, we stated that there exists a race condition between a set of inter-process events when there exists no causal order between these events. In practice, this definition must be refined: concurrent accesses that do not modify the data are not problematic. Hence, when an event occurs between two processes, we need to determine whether it is *causally ordered* with the *latest write* on this data.

Lamport clocks [15] keep track of the logical time on a process; vector clocks (introduced by [17]) allow for the partial causal ordering of events. A vector clock on a given process contains the logical time of each other process at the moment when the other process had an influence on the process (*i.e.*, last time it had a causal influence on this process).

When the causality relationship between a set of events that contains at least a write event cannot be established, we can conclude that there exists a race condition between them. More specifically, when we compare the vector clocks that are associated with these events and the latest write.

Lemma 1 (Mattern, Theorem 10): $\forall e, e' \in E : e < e'$ iff $H(e) < H(e')$ and $e \parallel e'$ iff $C(e) \parallel C(e')$

Corollary 1: Consider two events denoted e_1 and e_2 and their respective clocks H_1 and H_2. If no ordering can be determined between H_1 and H_2, there exists a race condition between e_1 and e_2 ($e_1 \times e_2$).

In the following algorithms, we detail the *put* and *get* commands. Algorithm 1 describes a *put* performed from $P0$ by the library to write the content of src address into process $P1$'s memory at address dst. Algorithm 2 describes a *get* performed by the library to retreive content of src address from process $P1$'s memory to process $P0$'s memory at address dst. Each process associates two clocks to areas of shared memory: a general-purpose clock V and a write clock W that keeps track of the latest write operation.

Figure 4 shows an example of two concurrent remote read operations (*i.e.*, *get* operations) on a variable a. This variable is initialized at a given value A before the remote accesses. Since none of the concurrent operations modifies its value, this is not a race condition. As stated in section III-C, there exists a race condition between concurrent data accesses iff at least one access modifies the value of the data. As a consequence, concurrent read-only accesses must not be considered as race conditions.

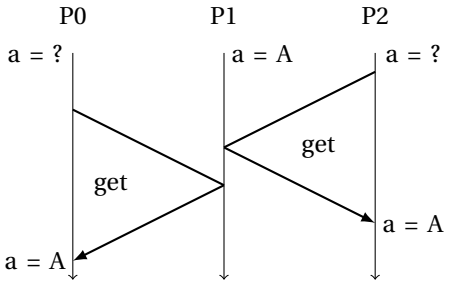


Figure 4: Two concurrent get operations

Algorithm 1: *Put* operation from $P0$ to $P1$

```
begin
    lock(P0, src);
    lock(P1, dst);
    V = update_local_clock(P0, src);
    W' = get_clock_W(P1, src);
    if ¬ compare_clocks(V, V')
      ∧¬ compare_clocks(V', V) then
      signal_race_condition() ;
    put(P0, src, P1, dst);
    update_clock_W(P1, dst);
    update_clock(P1, dst);
    unlock(P1, dst);
    unlock(P0, src);
end
```

Algorithm 2: *Get* operation from $P0$ to $P1$

```
begin
    lock(P0, dst);
    lock(P1, src);
    V = update_local_clock(P0, dst);
    W = V  V' = get_clock(P1, src);
    if ¬ compare_clocks(W, V')
      ∧¬ compare_clocks(V', V) then
      signal_race_condition() ;
    get(P1, src, P0, dst);
    update_clock(P1, src);
    update_clock(P0, dst);
    unlock(P1, dst);
    unlock(P0, dst);
end
```

The $lock$ primitive takes care of mutual exclusion if the addressed value is in public space or not. If the address is in private space, there is no need of a real lock (except in

multithreading). The $compare\_clocks(P0, a, P1, b)$ primitive first read the vector clock $V_1(b)$ from $P1$'s memory and then compare it with $V_0(a)$. The comparison is done as described in algorithm 3.

Algorithm 3: compare_clocks algorithm

begin
  return $\forall n \in \{0, \ldots, N-1\} : V_{Pi} < V_{Pj} \Leftrightarrow V_{Pi}[n] < V_{Pj}[n]$) ;
end

In figure 5, we present three use-cases of our algorithm: two situations of race conditions and one when the messages are causally ordered.

B. Clock update

The clock matrix V_{Pi} is maintained by each process Pi. This matrix is a *local* view of the global time. It is initially set to zero. Before Pi performs an event, it increments its local logical clock $V_{Pi}[i,i]$ ($update\_local\_clock$). Clocks are updated by any event as follows (algorithm 4, see [18]).

Algorithm 4: max_clock algorithm

begin
  $\forall l, V'[l] = \max(V_{Pi}[l], V_{Pj}[l])$;
  return V' ;
end

The remote clock update is performed as follows:

Algorithm 5: update_clock algorithm

begin
  V_{Pj} = get_clock(P_j, dst);
  V' = max_clock(V_{Pi}, V_{Pj});
  put_clock(P_j, dst, V');
end

The update_clock_W algorithm is similar to the update_clock algorithm, except that it updates the value of the "write clock" W.

Since the shared memory area is locked, there cannot exist a race condition between the remote memory accesses induced by the race condition detection mechanism.

C. Discussion on the size of clocks

If n denotes the number of processes in the system, it has been shown that the size of the vector clocks must be at least n [19]. As a consequence, the size of the clocks cannot be reduced.

D. Discussion on error signalisation

A race condition may not be fatal: some algorithms contain race conditions on purpose. For example, parallel master-worker computation patterns induce a race condition between workers when the results are sent to the master. Therefore, race conditions must be *signaled* to the user (*e.g.*, by a message on the standard output of the program), but they must not abort the execution of the program.

In the algorithm presented here, we refine the error detection by using two distinct clocks, a general-purpose one and a "write clock". The drawback of this approach is that it doubles the necessary amount of memory. On the other end, it offers more precision and eliminates numerous cases of false positives (*e.g.*, concurrent read-only accesses).

V. Conclusion and perspective

In this paper, we presented a model for distributed shared memory. This model considers interactions between processes and causal dependencies, while taking into account specific features from hardware used to implement such systems.

In this model, we propose an algorithm for detecting race conditions caused by the absence of ordering between events in the distributed system. This algorithm can be implemented in the communication library of the run-time support system that executes the program on a distributed system.

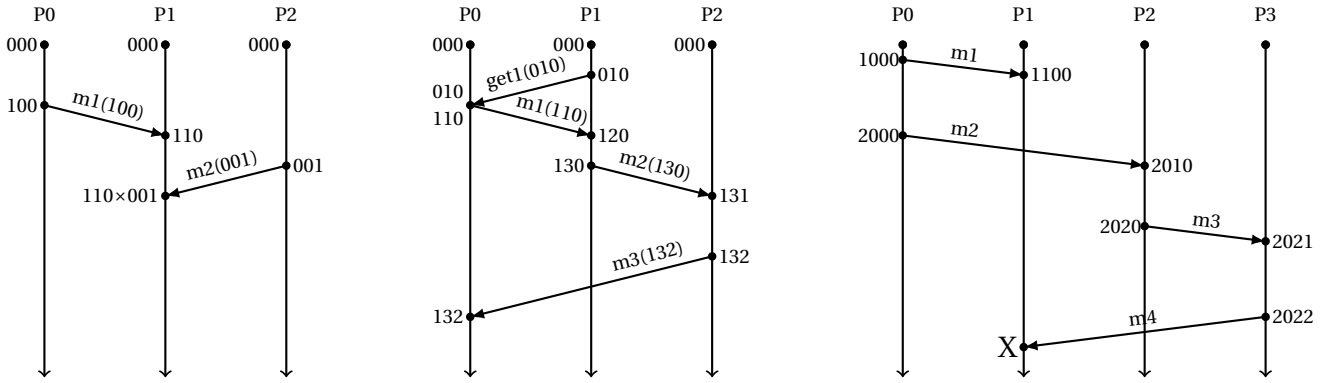
A. Discussion

As stated in section IV-C, the size of the matrices cannot be smaller than n, if n denotes the number of processes in the system. Moreover, a clock must be used for each shared piece of data. As a consequence, our algorithm has an overhead on data storage space (clocks associated with shared data) and with communication performance. However, race condition detection is typically a *debugging* technique. It does not need to be enabled on a parallel application that is actually running at full performance and large-scale systems. Parallel programmes are typically debugged on small data sets and a few processes (typically, about 10 processes).

B. Future works

The model presented in this paper leads to new interpretations of distributed algorithms. New operations can also be imagined, such as non-collective, global operations: for example, a process can perform a reduction (*i.e.*, a global operation on some data held by all the other processes) without any participation for the other processes, by fetching the data remotely.

Our race condition detection algorithm can be implemented at two levels: in the communication library of a parallel language, for automatic detection of conflictual accesses, or in the pre-compiler, as wrappers around remote data accesses.



(a) Race condition detected on reception of m1 (put) and m2 (put)

(b) No race condition between m1 (get) and m3 (put)

(c) Race condition detected between m1 (put) and m3 (put)

Figure 5: Detecting race conditions with vector clocks

REFERENCES

[1] U. Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National, Tech. Rep. LBNL-59208, 2005.

[2] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," *Concurrency - Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[3] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[4] M. P. I. Forum, "MPI: A message-passing interface standard," Department of Computer Science, University of Tennessee, Tech. Rep. UT-CS-94-230, Apr. 1994. [Online]. Available: ftp://netlib.org/tennessee/ut-cs-94-230.ps

[5] D. B. K. Yelick and C. Wallace, "A Proposal for a UPC Memory Consistency Model," Lawrence Berkeley National, Tech. Rep. LBNL-54983, 2004.

[6] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy, "The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors," Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California 94305-4055, Tech. Rep. CSL-TR-95-660, 1995.

[7] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[8] K. Li and P. R. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings 1986 5th Annual ACM Symposium on Principles of Distributed Computing*. New York, NY: ACM, 1986, pp. 229–239.

[9] A. Geist, W. D. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, A. Skjellum, and M. Snir, "MPI-2: Extending the message-passing interface," in *1st European Conference on Parallel and Distributed Computing (EuroPar'96)*, ser. Lecture Notes in Computer Science, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1123. Springer, 1996, pp. 128–135.

[10] B. Krammer and M. M. Resch, "Correctness checking of mpi one-sided communication using marmot," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, Eds. Springer, 2006, pp. 105–114.

[11] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations and advanced topics*. The McGraw-Hill Companies, March 1998.

[12] S. Dolev, *Self-Stabilization*. MIT Press, March 2000.

[13] G. Tel, *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[14] R. Brightwell, "A new MPI implementation for Cray SHMEM," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of 11th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 122–130.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[16] D. P. Helmbold and C. E. McDowell, "A taxonomy of race detection algorithms," University of California, Santa Cruz, Tech. Rep. UCSC-CRL-94-35, Sep. 1994, (paper copy $6.00).

[17] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1988, pp. 215–226.

[18] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," *Computer*, vol. 29, pp. 49–56, 1996.

[19] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Inf. Process. Lett.*, vol. 39, pp. 11–16, July 1991. [Online]. Available: http://portal.acm.org/citation.cfm?id=117603.117606