Data Coherency in Distributed Shared Memory

Franck Butelle, Camille Coti

HAL Id: hal-00656075
https://hal.science/hal-00656075

Data Coherency in Distributed Shared Memory

Franck Butelle

Franck.Butelle@lipn.univ-paris13.fr

LIPN, CNRS-UMR7030, Université Paris 13, F-93430 Villetaneuse, France

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.

Keywords: Distributed shared memory, concurrent systems, race condition

1   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 [8], Titanium [26] or Co-Array Fortran [21]. 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 [10]): 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 [15], 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 reading 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 standard[2] or Myrinet[3].

The remainder of this paper is organized as follows. In section 2, we present an overview of previous models for distributed shared memory and how consistency and coherency has been handled in these models. In section 3 we present our model for distributed shared memory and how it can be related to actual systems. In section 4 we present how race conditions can be represented in this model, and we propose an algorithm for detecting them.

2 Previous work

Distributed shared memory is often modeled as a large cached memory [14]. 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 [18], 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.

---

[1] http://www.top500.org
[2] http://www.infinibandta.org/
[3] http://www.myri.com

Maintaining the coherence of cache-based distributed shared memory can then be considered as a cache-coherency problem. [19] 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 [8], Titanium [26] and Co-Array Fortran [21], data locality (*i.e.*, which process holds the data in its local memory) is resolved at compile-time.

These languages rely on a concept called *Parallel Global Address Space* (PGAS). In this global address space, a chunk of data can be localized by a tuple of two elements: the aforementioned data locality, which is also called the process the data has *affinity to*, and the local address in the memory of the process this given chunk has affinity to.

In UPC, data can be declared as either *shared* or *private*. When a process (called a *thread* in the UPC terminology, by analogy with shared memory systems) accesses a shared chunk of data that does not have affinity to itself, it is accessed remotely by the run-time system (GASNet [4] in the case of UPC, Titanium and the LANL implementation of Co-Array Fortran). The semantics of these remote data accesses will be discussed further in section 3.2.

The coordination language LINDA [1] implements a model where the distributed shared memory is called a *tuple space*. Chunks of data called *tuples* can be added into the tuple space and read (and removed or not) from it. It can also create new processes on-the-fly to evaluate tuples. Besides the particular semantics of the language (based on tuples matching), the model it defines for implementing distributed applications is quite different from other PGAS languages such as UPC and Co-Array Fortran. In the LINDA model, the programmer explicitly pushes and pulls data from the distributed address space (the tuple space), and the run-time system is in charge with data locality.

Queries to the tuple space involve a search for matching tuples in the whole tuple space. In [3], the distributed memory manager is called the *tuple handler*. This implementation uses a centralized tuple server, and the processes of the application are clients. As stated by the author of this work, this creates a data bottleneck and is a major drawback of this implementation.

Javaspace [24] implements this distributed memory model using RMI and some features of the Java language such as class types comparisons. The transaction model relies on a two-phase commit model.

The simplicity of the LINDA language and its siblings make them attractive for implementing distributed applications [12]. However, this simplicity has one major drawback: it has little (or no) safety features. The programmer has no control on the data accesses, such has the locks provided by languages from the family of UPC.

Locks are the most basic data safety mechanism there is. Their basic task is to ensure mutual exclusion in a critical section: the process that holds the lock is the only one that can access the critical section, while the other processes must wait until the lock is released to try to get it. As described more formally in section 3.1, locks can also be defined on memory areas. They guarantee the fact that only one process is accessing a given area of memory. Hence, they provide a certain form of data safety, in a sense that a chunk of data cannot be altered by any other process which a given process is accessing it. However, they are not sufficient to guarantee that the operations on this chunk of data are causally ordered and that the distributed program is free from race conditions [13].

The MPI-2 standard [11] defines remote memory access operations. The MARMOT error checking tool [16] checks correct usage of the synchronization features provided by MPI, such as fences and windows. A window is an area of a process's memory that is made available to other processes for remote memory accesses. Windows are created and destructed collectively by all the processes of the communicator. MARMOT associates an *access epoch* to each window to determine whether conflicting accesses are made to the same memory location in a window. MPIRACE-CHECKER [22] uses a *mirror window* and marks it such in a way that unsafe, concurrent memory accesses can be detected as such. These techniques are using features that are specific to the MPI-2 standard and its remote memory access model.

3 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.

3.1 Distributed shared memory model

In many shared-memory models that have been described in the literature [2, 9, 25], pairs of processes communicate using registers where they read and write data. Distributed shared memory cannot use registers between processes 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 multiprocess system. In this model, each process maps two distinct areas of memory: a *private* memory and a *public* memory. The private memory can be accessed from this process only.

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

Public memory can be accessed by any process 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.
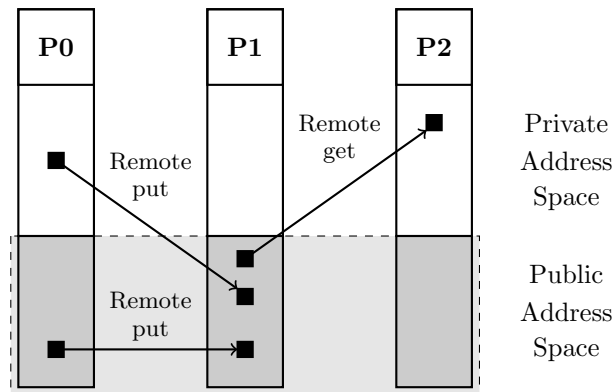

Figure 1: Memory organization of a three-process distributed shared memory system.

The compiler is in charge with data locality, *i.e.*, putting shared data in the public memory of processes. 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 process P. Instead of accessing it using its address in the local memory, processes use the process's name and its address in the memory of this process. This couple $(process\_name, local\_address)$ is the addressing system used in the global address space. The compiler also performs the address resolution when the programmer asks a process to access this shared data x.

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.

3.2 Communications

Processes access areas of public memory mapped by other processes 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 process's side. Hence, a process A is not aware of the fact that another process B has accessed (*i.e.*, read or written) its memory.

Accessing data in another process's memory is called *Remote Direct Memory Access* (RDMA). It can be performed with no implication from the remote process'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 [6] 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*.
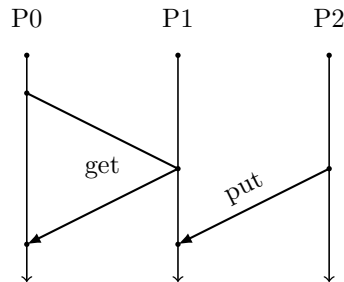

Figure 2: Remote R/W memory accesses.

Put consists in writing some data into the public memory of another process. It involves one message, from the source process to the destination process, 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 process's public memory. It involves two messages: one to request the data, from the requesting process to the process that holds the data, and one to actually transfer the data, from the process that holds the data to the requesting process. 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).
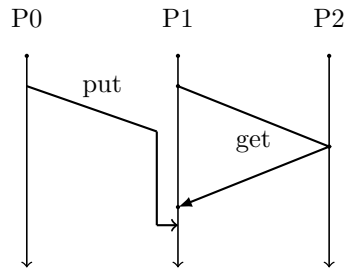

Figure 3: A put operation is delayed until the end of the get operation on the same data.

3.3 Race conditions

One major issue created by one-sided communications is that several processes can access a given area of memory without any synchronization nor mutual knowledge. For example, two processes A and B can write at the same address in the shared memory of a third process 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. A race condition makes, at least, hard to reason about a program and therefore is usually considered as a bug or at least as a design flaw.
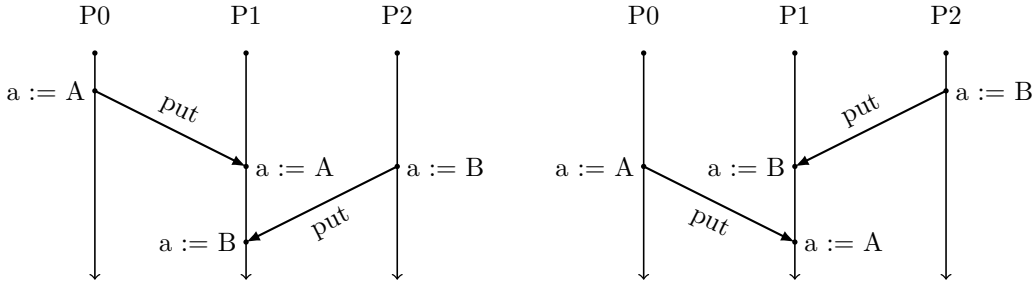

Figure 4: Example of race condition

In figure 4 we present a very simple case of race condition. The order of the execution (due for example to differences between the relative speeds of the processes) generate two different results.

In the kind of systems we are considering here, a race condition can occur when several operations are performed by different processes 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 processes and the communication channels that interconnects them.

Two events e_1 and e_2 are *causally ordered* iff there exists an *happens before* (as defined by [17] and denoted $\rightarrow$) relationship between them such that $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$. Race conditions are defined in [13] by the fact that there exists no causal order between e_1 and e_2 (further denoted by $e_1 \times e_2$).

3.4 A parallel pseudo-language

In this section, we describe a parallel pseudo-language. This language is meant to describe parallel algorithms using a parallel global address space in the same way as pseudo-code describes sequential algorithms. Its purpose is not to replace UPC nor any parallel programming language (Titanium, Co-Array Fortran...) but to describe algorithms with a language-agnostic description language.

Computation units are called *processes*. Variables have a *visibility*. They can be either *private* or *shared*. Private means that the variable is visible by the current process only, and is physically located in its memory. Shared means that the variable is in the distributed public memory space. It can be physically located in any process's memory. Each process is assigned a unique number, called its *rank*. If the total number of processes in the system is N, ranks are consecutive and range from 0 to $(N-1)$. The physical location of this variable (*i.e.*, the rank of the process that maps the chunk of memory where it is physically located) is called its *affinity*.

The compiler and the underlying run-time system translate accesses to shared variables that do not have affinity to the current process into remote data accesses. Accesses to the local area of shared memory are performed using the local memory controller.

An example using this parallel pseudo-language is given in algorithm 1. This algorithm defines two variables of type *Integer*: a and b; a is *shared* (line 2) and b is *private* (line 3). We put the value of the process rank into this private variable b (line 4). Then the value of b is put into the shared variable a (line 5). A global synchronization (*i.e.*, a barrier, line 6) waits until all the processes have reached this point of the program. Then one process prints the value of a (line 8).

The equivalent code in UPC (Unified Parallel C) is given by algorithm 2. UPC calls *threads* its parallel processes. It predefines some identifiers, such as $MYTHREAD$, which provides the rank of

the current thread (line 4 and line 7). Shared variables are declared using the qualifier *shared* (line 2). By default, variables are *private*: hence, if nothing is specified, a variable is private (line 3).

Algorithm 1: Example using our parallel pseudo-language	**Algorithm 2**: Equivalent program in UPC
1 **begin**	1 **begin**
2 shared *a*: Integer ;	2 shared int a ;
3 private *b*: Integer ;	3 int b ;
4 $b \longleftarrow myrank$;	4 b = MYTHREAD ;
5 $a \longleftarrow b$;	5 a = b ;
6 barrier() ;	6 upc_barrier();
7 **if** $myrank == 0$ **then**	7 **if** $0 == MYTHREAD$ **then**
8 print *a* ;	8 printf("%d\n", a) ;
9 **end**	9 **end**

The compiler chooses where the shared variables are physically located. In our example, we assume without loss of generality that process rank 0 has been chosen to store integer a in its local memory. Algorithm 3 describes the sequence of instructions executed by each process with explicit communications (remote data accesses). The shared variable is actually declared on process 0 only (line 2-3). Access to this shared variable is done between lines 6 and 9. If the local process is the process that holds the data (lines 6-7), it corresponds to a simple local variable. If the shared variable does not have affinity to the local process, it must be accessed using a remote data access. Since our algorithm is *writing* into this variable, it corresponds to a *put(localaddr, rank, remoteaddr)* operation. A *reading* will be denoted by *get(localaddr, rank, remoteaddr)* ($rank$ will always be the rank of the process owning *remoteaddr* variable physically).

The corresponding inter-process communications and the state of the shared variable a through the execution of this algorithm are depicted on figure 5 and figure 6 on an execution involving three processes P0, P1 and P2. As we can see here, all the processes write into the variable a without coordination. These two figures show two possible executions of the algorithm. Figure 5 depicts a situation where P0 writes first into a (upon this point, $a = 0$), then P1 performs a *put* into a ($a = 1$) and last, P2 performs a *put* into a (the value of a at the end is $a = 2$). Another possible execution is depicted in figure 6. In this case, P2's put operation is executed before P1's one. As a consequence, the final value for a is $a = 1$. This example contains a race condition.

4 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 3.

As stated in section 2, previous works on race condition detection have focused on specific communication models. Here we present a new algorithm designed specifically for the model presented in the previous section, and how our algorithm takes specific advantage of one-sided communications and remote memory access.

4.1 Causal ordering of events

In section 3.3, 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 [17] keep track of the logical time on a process; vector clocks (introduced by [20]) allow for the partial causal ordering of events. A vector clock on a given process contains the logical

Algorithm 3: Algorithm 1 as executed by the run-time system

```
1  begin
2      if  myrank == 0 then
3          a: Integer ;
4      b: Integer ;
5      b ⟵ myrank ;
6      if  myrank == 0 then
7          a ⟵ b ;
8      else
9          put( b, 0, a ) ;
10     barrier() ;
11     if  myrank == 0 then
12         print a ;
13 end
```
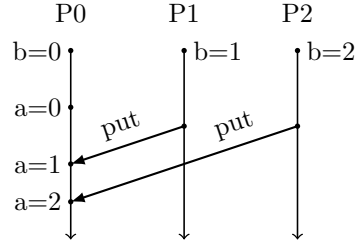

Figure 5: Communications performed by algorithm 1: first situation
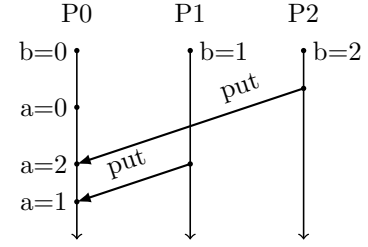

Figure 6: Communications performed by algorithm 1: second situation

time of each other process at the moment when the other process had an influence on this 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 (See [20], 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 4 describes a *put(src, j, dst)* performed from $P0$ by the library to write the content of *src* address into process j's memory at address *dst*. Algorithm 5 describes a *get(dst, j, src)* performed by the library to retreive content of *src* address from process Pj'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 7 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 3.3, 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.

The *lock* (resp. *lock_local*) 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 *lock(j, addr)* command locks an area of memory located at address *addr* on process j. The *local_lock(addr)* command locks an area of memory located at address *addr* on the local process.

In figure 8, we present three use-cases of our algorithm: two situations of race conditions and one when the messages are causally ordered.
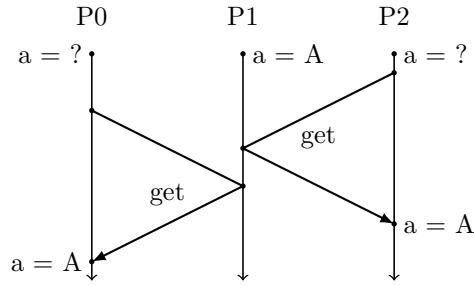
Figure 7: Two concurrent get operations

Algorithm 4: *Put(localaddr = src, rank = j, remoteaddr = dst)* operation from local proc. to Pj

---

1 begin
  2   lock_local(src);
  3   lock(j, dst);
  4 V = update_local_clock(src);
  5 W = get_clock_W(j, src);
  6   **if** $\neg$ *compare_clocks*(V, W) $\wedge \neg$ *compare_clocks*(W, V) **then**
  7   signal_race_condition() ;
  8   put(src, j, dst);
  9   update_clock_W(j, dst);
 10   update_clock(j, dst);
 11   unlock(j, dst);
 12   unlock_local(src);
13 end

4.2 Clock update

The clock vector V_{Pi} (or simply V in the algorithms to denote the local one) is maintained by each process Pi. This vector 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 shown in algorithm 7, see [23].

Algorithm 5: *Get(localaddr = dst, rank = j, remoteaddr = src)* operation from local proc. to Pj

---

1 begin
  2   lock_local(dst);
  3   lock(j, src);
  4 W = update_local_clock_W(dst);
  5 V = get_clock(j, src);
  6   **if** $\neg$ *compare_clocks*(W, V) $\wedge \neg$ *compare_clocks*(V, W) **then**
  7   signal_race_condition() ;
  8   get(dst, j, src);
  9   update_clock(j, src);
 10   update_local_clock(dst);
 11   unlock(j, src);
 12   unlock_local(dst);
13 end

Algorithm 6: compare_clocks(V, W) algorithm

---

1 **begin**
2   return ($\forall n \in \{0, \ldots, N-1\} : V[n] < W[n]$) ;
3 **end**

(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 m4 (put)
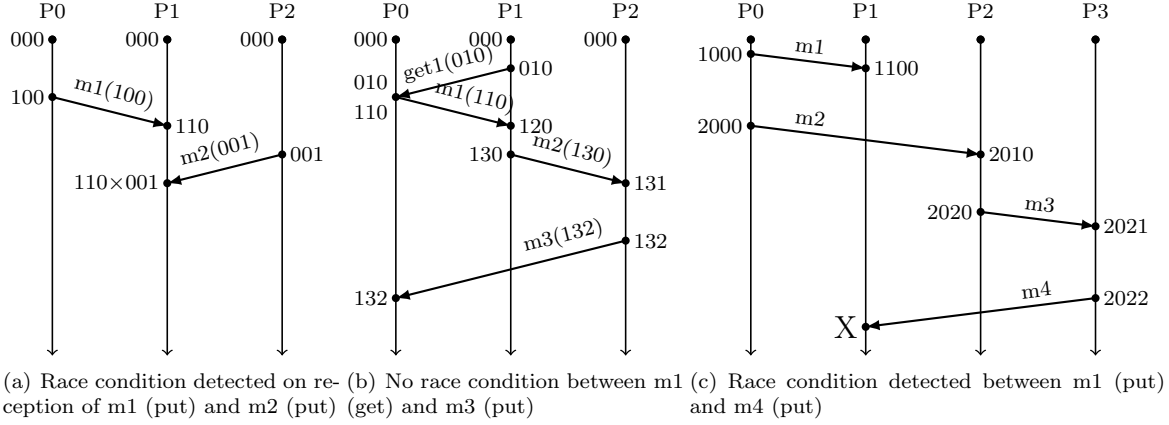
Figure 8: Detecting race conditions with vector clocks

Algorithm 7: update_clock($rank = j, remoteaddr = dst$) algorithm

---

1 **begin**
2      $V_{Pj} = \text{get\_clock}(j, dst)$;
3      $V = \text{get\_local\_clock}(dst)$;
4      $V' = \text{max\_clock}(V, V_{Pj})$;
5      $\text{put\_clock}(V', j, dst)$;
6 **end**

Algorithm 8: max_clock(V, W) algorithm

---

1 **begin**
2      $\forall l, V'[l] = \max(V[l], W[l])$;
3   return V' ;
4 **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.

4.3 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 [7]. As a consequence, the size of the clocks cannot be reduced.

4.4 Discussion on error signalisation

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
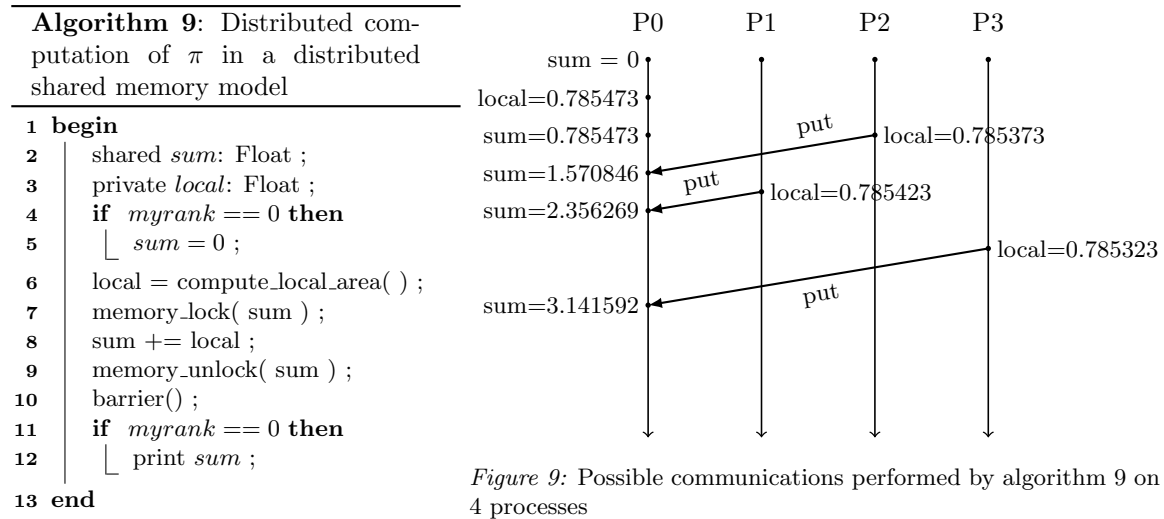
amount of memory. On the other hand, it offers more precision and eliminates numerous cases of false positives (*e.g.*, concurrent read-only accesses).

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.

As an example where race conditions are not an issue, we can consider a distributed sum computation. For instance, approximation of the Pi number using the sub-curve area method (it calculates the area under the curve of a quarter of a circle) distributes computation between the available parallel processes and computes a global sum of their partial results. Since the sum operation is commutative, this global sum can be computed in any order.

Algorithm 9 gives a distributed algorithm that computes the value of π in the parallel pseudo-language we described in section 3.4. There exists a race condition on line 8, since the processes can write into the shared variable *sum* in any order.

This race condition is depicted in figure 9, on which a possible execution of this algorithm is represented. In this execution, the relative computing and communication speeds of the four processes are such that line 8 is first executed by process $P0$, then by process $P2$, followed by process $P1$ and finally by process $P3$.

Algorithm 9: Distributed computation of π in a distributed shared memory model

```
1  begin
2  |    shared sum: Float ;
3  |    private local: Float ;
4  |    if  myrank == 0 then
5  |    |   sum = 0 ;
6  |    local = compute_local_area( ) ;
7  |    memory_lock( sum ) ;
8  |    sum += local ;
9  |    memory_unlock( sum ) ;
10 |    barrier() ;
11 |    if  myrank == 0 then
12 |    |   print sum ;
13 end
```

Figure 9: Possible communications performed by algorithm 9 on 4 processes

However, any other order would have given the correct result. As a consequence, this race condition does not have any influence on the final result of the computation. Hence, it is not fatal, and the program can be executed normally. A race condition detection system ought to notify the programmer about it but not abort the execution.

5   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.

5.1   Discussion about overheads

As stated in section 4.3, the size of the clocks 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. Our algorithm is presented using additional messages ($O(1)$ messages carrying $O(n)$ bytes) for clock update introducing an overhead on latency. Note that sometimes it can be implemented by extending existing messages (by piggybacking techniques or extension of headers/enveloppe of active messages) when the underlying communication network allow longer messages.

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).

5.2   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 of 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. It does not require any modification to the UPC language, since our algorithms can be implemented in the implementation of the communication primitives called by UPC.

It is also possible to extend this technique to perform trace analysis. When doing trace analysis, it is necessary to keep all messages interactions (generally speaking it is done at the sender side) and label them with a kind of "date". It is often hard (or impossible) to have a real physical global clock (for basic physics reasons, as stated in [17]). Some authors use some special counters available on processors (for example Time Stamp Counter of Intel processors) to achieve this; however, there is no reason for the timestamp counters of multiple CPUs to stay synchronized. We can use vector clocks as timestamps for this purpose, which would be a better way to memorize a "date". It can also be used for the so-called "replay" debugging technique: it logs the causality information of the execution of deployed application processes and replays them deterministically, reproducing race conditions faithfully and non-deterministic failures, enabling careful offline analysis and failure confinement [5].

References

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computers*, 19:26–34, August 1986.

[2] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations and advanced topics*. The McGraw-Hill Companies, March 1998.

[3] Jim Basney. *A Distributed Implementation of the C-Linda Programming Language*. PhD thesis, Oberlin College, Computer Science Program, May 1995.

[4] Dan Boachea. Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[5] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PRecent Advances in Parallel Virtual Machine and Message Passing*

Interface, *14th European PVM/MPI User's Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 297–306. Springer, 2007.

[6] Ron Brightwell. A new MPI implementation for Cray SHMEM. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 122–130. Springer, 2004.

[7] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39:11–16, July 1991.

[8] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National, 2005.

[9] Shlomi Dolev. *Self-Stabilization*. MIT Press, March 2000.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994.

[11] Al Geist, William D. Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Anthony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *1st European Conference on Parallel and Distributed Computing (EuroPar'96)*, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135. Springer, 1996.

[12] Petr Hanáček. Parallel simulation using the linda language. In *5th Moravo-Silesian International Symposium on Modelling and Simulation of Systems*, pages 263–267, 1993. http://www.fit.vutbr.cz/ hanacek/papers/SISY93.pdf.

[13] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, September 1994. (paper copy $6.00).

[14] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, California 94305-4055, 1995.

[15] D. Bonachea K. Yelick and C. Wallace. A Proposal for a UPC Memory Consistency Model. Technical Report LBNL-54983, Lawrence Berkeley National, 2004.

[16] Bettina Krammer and Michael M. Resch. Correctness checking of mpi one-sided communication using marmot. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, pages 105–114. Springer, 2006.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[19] 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*, pages 229–239, New York, NY, 1986. ACM.

[20] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[21] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[22] Mi-Young Park and Sang-Hwa Chung. Detecting Race Conditions in One-Sided Communication of MPI Programs. In *2009 Eigth IEEE/ACIS International Conference on Computer and Information Science (icis 2009)*, June 2009.

[23] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29:49–56, 1996.

[24] Sun Microsystems. *JavaSpaces Specification 1.0*, 1999.

[25] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[26] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.