TACC Technical Report IMP-23

Distributed computing theory in IMP

Victor Eijkhout*

January 3, 2017

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

We address a number of distributed computing issues in IMP, such as

- sequential consistency;
- causal ordering and distributed time.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)

1 Introduction

Distributed computing has a long history, and a repertoire of typical issues. Here we address a few.

2 Sequential consistency

2.1 Introduction

A multi-processor program is called *sequentially consistent* [4] if it corresponds to a single-processor execution that interleaves the processor programs, preserving the individual orderings.

Consider the following multi-processor program:

Process 1:

```
a := 1
if iszero(b) then
  execute1()
  a := 0
```

Proces 2:

```
b := 1
if iszero(a) then
  execute2()
  b := 0
```

The flags `a,b` guarantee that at most one of `execute1,2` is active simultaneously (it is possible that neither is executed).

Interleaving these statements can correspond to the following sequential programs:

```
a := 1
if iszero(b) then
  execute1()
  a := 0
b := 1
if iszero(a) then
  execute2()
  b := 0
```

```
b := 1
if iszero(a) then
  execute2()
  b := 0
a := 1
if iszero(b) then
  execute1()
  a := 0
```

```
// in either order:
a := 1 & b := 1
// in either order:
if iszero(a) & iszero(b)
// no execute
```

However, testing `a,b` involves fetching these quantities from memory to a local register `a',b'` on process 2 and 1 respectively. Thus we actually have:

Process 1:

```
a := 1
b' := b
if iszero(b') then
  execute1()
  a := 0
```

Proces 2:

```
b := 1
a' := a
if iszero(a') then
  execute2()
  b := 0
```

which a compiler could turn into

Process 1:

```
b' := b
a := 1
if iszero(b') then
  execute1()
  a := 0
```

Proces 2:

```
a' := a
b := 1
if iszero(a') then
  execute2()
  b := 0
```

since the load of `b` is independent of the setting of `a` (the Lamport paper cited above probably has a typo when it talks about 'the `b:=1` and `fetch b` operations of process 1'). However, this has a possible interleaving where both `execute1/2` are active at the same time.

Thus Lamport states 'Requirement 1' that memory requests are made in the order specified by the program.

Lamport has a second requirement that memory modules need to service their requests in FIFO order (for instance the `a:=1` of processor 1 and the `a' := a` of processor 2. This seems unnecessary to me: having memory modules work out of order is equivalent to introducing a third processor which has the reorderings forbidden by Requirement 1.

2.2 Equivalent IMP programs

IMP programs explicitly send and receive data, as opposed to merely fetching data in shared memory programs. This allows us to solve the sequential consistency conundrum.

We take the Lamport example, and – since the 'else' clause is missing that would guarantee eventual execution of the critical region – we abstract its basic problem to: *guarantee that at most one of* `execute1/2` *is executed.*

The case where neither of `execute1/2` is called corresponds to the program (where we make the initialization explicit):

Process 1:

```
a := 0
a := 1
send(a)
receive(b)
if iszero(b):
  // false
```

Process 2:

```
b := 0
b := 1
send(b)
receive(a)
if iszero(a):
  // false
```

On the other hand, a program that would execute only `execute1` would be:

Process 1:

```
a := 0
a := 1
send(a)
receive(b)
if iszero(b):
  // true
```

Process 2:

```
b := 0
send(b)
b := 1
receive(a)
if iszero(a):
  // false
```

The following IMP program satisfies the property that the processes can communicate the initial or the changed value of `a,b`:

Process 1:

```
a := 0
if c1 then
  send(a)
a := 1
if not c1 then
  send(a)
receive(b)
if iszero(b):
```

Process 2:

```
b := 0
if c2 then
  send(b)
b := 1
if not c2 then
  send(b)
receive(a)
if iszero(a):
```

However, it also generates the execution where both processes communicate their initial value of `a,b`, so neither process calls `execute`.

The solution involves a global synchronization step and an identically computed condition `c` on all processes:

Process 1:
```
c in [0,1]
d := 0
a := 0
if c then
  send(a)
  d := 1
d = max(d)
a := 1
if not c or not d then
  send(a)
receive(b)
if iszero(b):
```

Collective:
```
replicated


d = max(d)
```

Process 2:
```
c in [0,1]
d := 0
b := 0
if not c then
  send(b)
  d := 1
d = max(d)
b := 1
if c or not d then
  send(b)
receive(a)
if iszero(a):
```

The generalization to more than two processes depends on the semantics of Lamport's example, which are ambiguous because of the missing 'else' clauses.

2.3 Discussion

We see that we can solve the sequential consistency problem by making as many kernels as there are processes, and letting at most one critical operation execute per kernel. Conflicts are prevented by a global barrier between the updates. While a barrier sounds onerous, it is no worse than Lamport's requirement of FIFO processing of memory requests.

(Note that we use a slight extension of strict IMP semantics here. By making the send operations conditional, we lose that the receive operation knows precisely from which kernel it receives.)

3 Causal ordering and distributed time

Much distributed computing literature is also concerned with temporal ordering of events. While in actuality events are temporaly ordered, in the context of the computation they are only partially ordered [2]: the temporal ordering between independent events can be reversed without it influencing the result of the computation.

Thus we need to define *distributed time*, that is, a true ordering between distributed events. For instance, *causal ordering* [5] ensures that if between two messages a causal relation exists, the 'effect' message is received after the 'cause' one.

In general, one seeks define a clock concept such that $C(e)$, where e is an event satisfies

$$e_1 < e_2 \Rightarrow C(e_1) < C(e_2) \tag{1}$$

where the partial ordering $\cdot < \cdot$ on events is defined as:

$$e_1 < e_2 \equiv \begin{cases} \text{there is a message from } e_1 \text{ to } e_2, \text{ or} \\ e_1 \text{ happens before } e_2 \text{ on the same process, or} \\ \exists e_3 : e_1 < e_3 < e_2 \end{cases}$$

This situation is considerably simplified in IMP, since we have no *a priori* concept of process: a process can be defined as a collection of tasks, but this is not a fundamental construct of the model.

Thus we define the partial relation as follows in IMP:

$$e_1 < e_2 \equiv \begin{cases} (e_1, e_2) \text{ is an edge in the task graph, or} \\ k(e_1) < k(e_2) \wedge p(e_1) = p(e_2), \text{ or} \\ \exists e_3 : e_1 < e_3 < e_2 \end{cases} \tag{2}$$

where e_1, e_2 are tasks, $p(e)$ is the process on which e is executed, $k(e)$ is the kernel to which a task e belongs, and the relation $k_1 < k_2$ between kernels follows from the algorithm, with a possible linearizing of partially ordered kernels.

Now, recognizing that (e_1, e_2) being an edge in the task graph implies that $k(e_1) < k(e_2)$, we see that defining

$$C(e) \equiv k(e)$$

is a clock that satisfies the consistency condition (1).

3.1 Causal ordering

An execution is said not to violate *causal ordering* if

$$\left.\begin{array}{l} s < s' \text{ causally} \\ r, r' \text{ on the same process} \end{array}\right\} \Rightarrow r < r' \text{ causally}$$

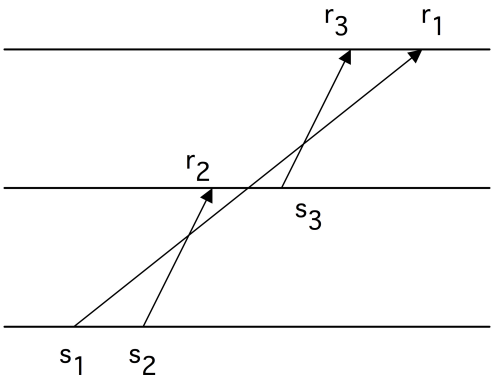

Figure 1: An execution that violates causal ordering

where the causal ordering was defined in (2).

Figure 1 illustrates a violation of the causal ordering: $s_1 < s_3$ but $r_3 < r_1$.

One motivation for causal ordering is making a distributed *snapshot*. If an event e is part of the snapshot, and $e' < e$, then e' should be part of the snaphot too.

4 Deadlock

Deadlock is impossible in IMP because we don't have cycles. It's a Directed *Acyclic* Graph, duh!

4.0.1 Process deadlock

Deadlock has been studied in the context of operating systems, databases, and other forms of asynchronous processes. Coffman [1] formulated four conditions that together lead to deadlock; breaking any of the four will resolve the deadlock situation.

- Mutual exclusion: at least one process involved has access to a resource that cannot be shared.
- Resource holding: at least one process is trying to access a resource held by some other processes.
- No pre-emption: all resources have to be explicitly released by the processes; there is no operating system that can pre-emptively de-allocate them.
- Circular wait: there must be a cycle of processes each waiting for another process to release a resource.

In the Integrative Model for Parallelism (IMP) model these clauses correspond to the following. If (α,β) is an edge in a kernel:

- the output β is a non-shareabled resource;
- the input α is a resouce, and for deadlock it needs to be the output of an edge in another kernel;
- pre-emption does not apply since there is nothing corresponding an operating system: producing an output corresponds to the resource being released for reading;
- for deadlock to occur it is necessary that there are kernels with instructions that transitively use β as input and have α as output.

Thus, in the IMP model an algorithm is deadlock-free if the pointwise task graph is acyclic. If the pointwise task graph has cycles we can still have a deadlock-free execution if each cycle is contained in a process, as in a block LU algorithm with irreducible diagonal blocks.

4.0.2 Deadlock and distributions

Considering again the distribution formulation of a prefix sum (equation (**??**))

$$\begin{cases} y(u) &= \sum_{\mathrm{scan},2} t(u,*) \\ t(u,*) &= y(*) \end{cases}$$

If we translate this to a task graph we immediately see that the graph has no cycles, so the algorithm will not deadlock. On the other hand,

$$\begin{cases} y(u) &= \sum_{2} t(u,*) \\ t(u,*) &= y(*) \end{cases}$$

is an implicit description of a result, not a specification of an algorithm. (In fact, it only has the solution $y \equiv 0$.) Its task graph has cycles, and its implementation using MPI blocking sends and receives will deadlock. In threading models we similarly get deadlock because of the circular data dependencies. Non-blocking operations in MPI similarly deadlock deadlock with only `IRecv` operations being posted.

5 Distributed knowledge

See the discussion in [3].

References

[1] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[2] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[3] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. *Scalable Memory Use in MPI: A Case Study with MPICH2*, pages 140–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[4] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[5] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, October 1991.