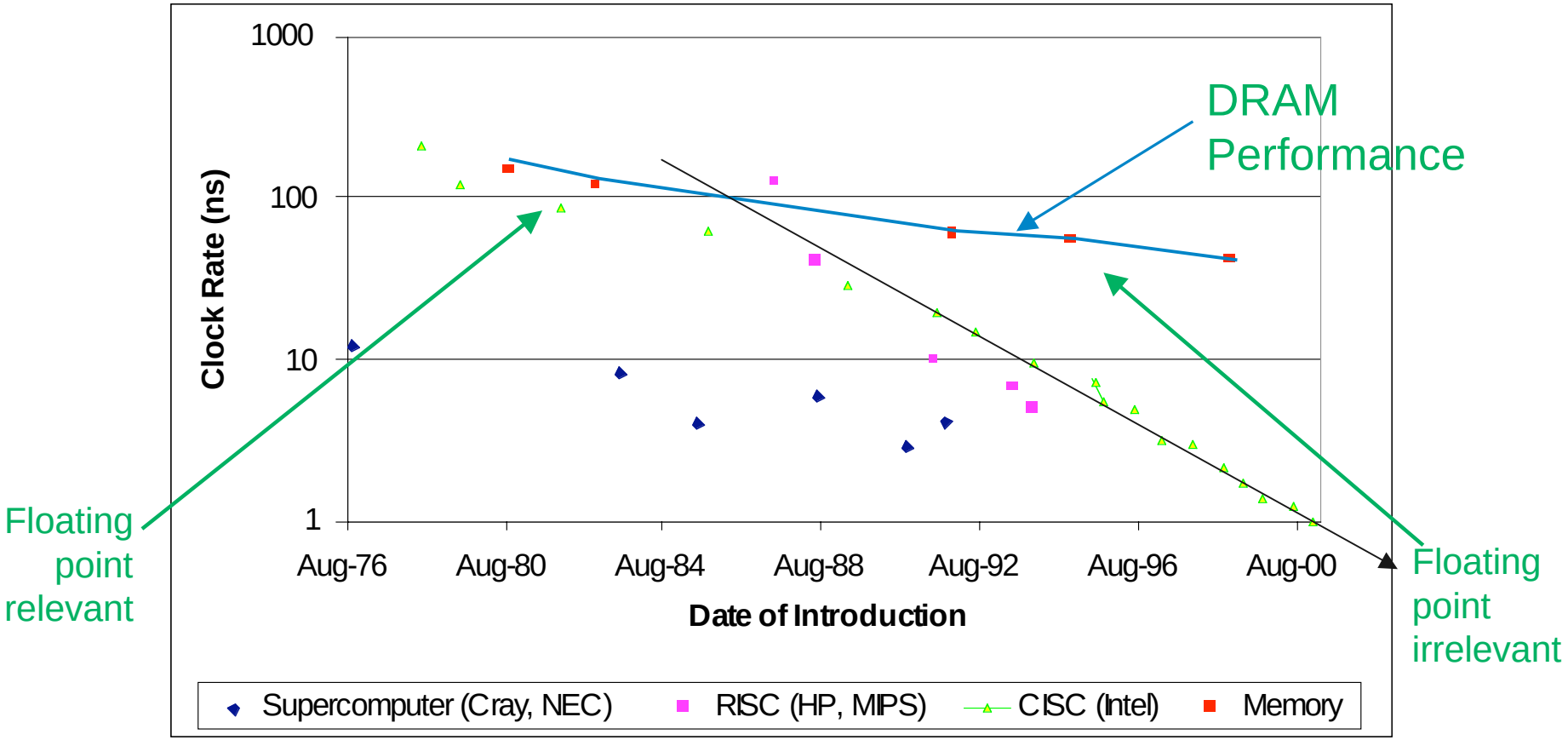
Issues in Developing a Thread-Safe MPI Implementation

William Gropp
Rajeev Thakur

Why MPI and Threads

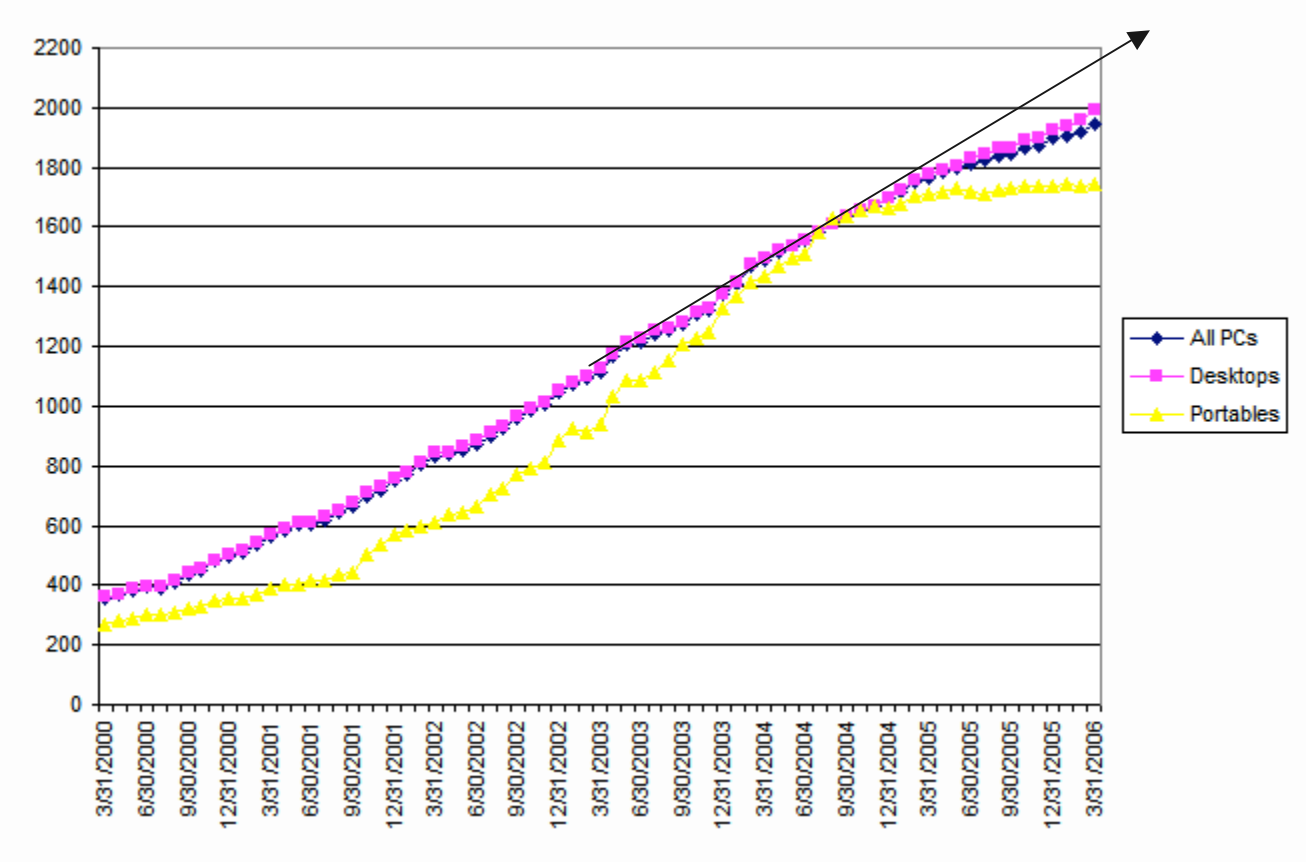
- Applications need more computing power than any one processor can provide
  - Parallel computing (proof follows)
- Fast, PRAM hardware does not exist; computer hardware trades simplicity for performance
- CPU performance trends pushing vendors to “multicore” and similar SMP-style hardware models, for which threads is an obvious programming model
- Threads are also a useful way to implement event-driven computations

Why is achieved performance on a single node so poor?

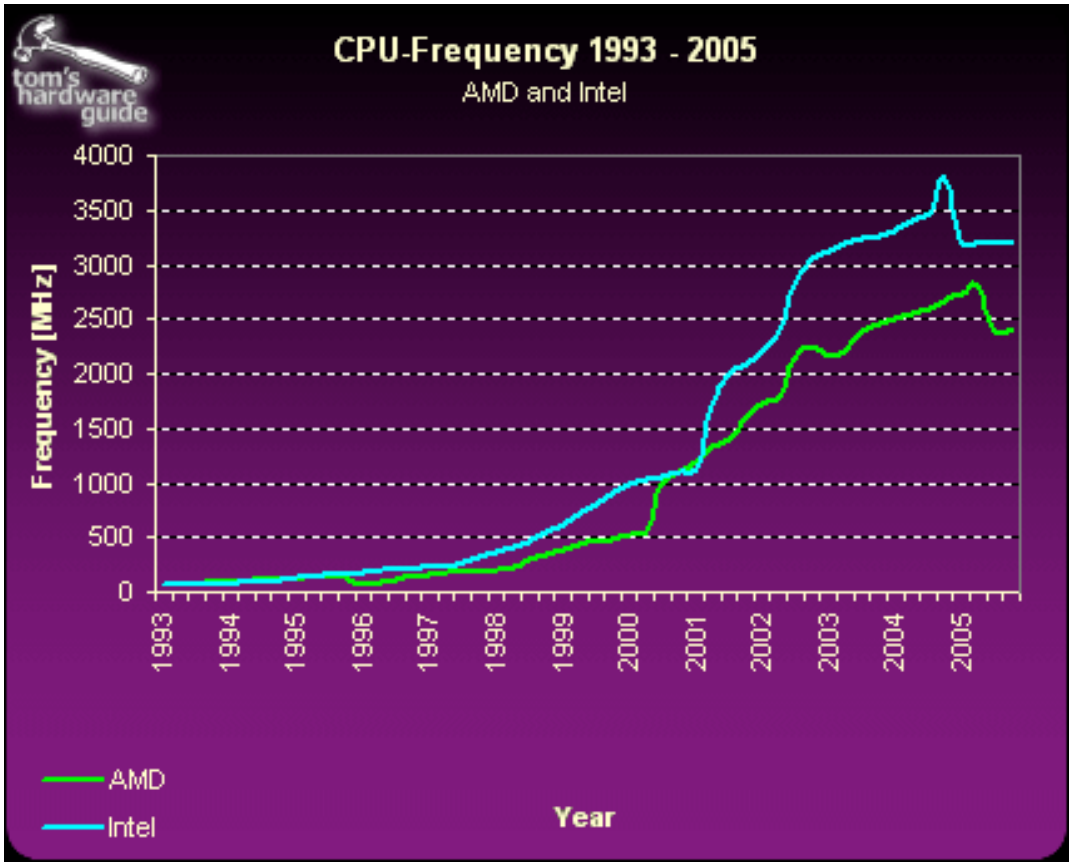


CPU performance is not doubling any more

- Avera
  http://



Peak CPU speeds are stable



- From http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/

Parallel programming models

- Threads for SMPs
  - Single sided access model – loads and stores with languages normal mechanism
  - Rules on memory access are vague, often requires language extensions
    - *Volatile* is necessary*, but* is not sufficient
- Message Passing Interface (MPI) for massively scalable systems
  - MPI-1 two-sided message passing
  - MPI-2 adds remote direct memory access, with careful (and complicated) access rules
  - MPI can be deterministic

The Talk From Here

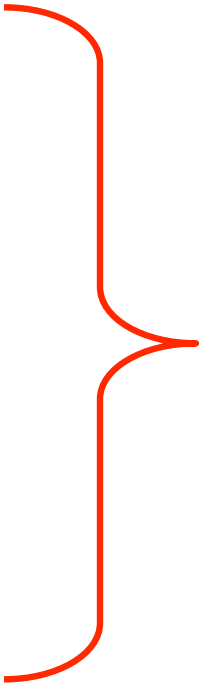
- Some comments on threads, including some of the tricky bits
- Some comments on MPI, then MPI and Threads
- MPI, Threads, and the programmer
  - What MPI does to “play well with threads”
  - Some notes on OpenMP (the HPC thread model)
- MPI, Threads, and the MPI implementer
  - Issues in providing a thread-safe MPI implementation

Some Thread Background

- What are Threads?
  - Executing program (process) is defined by
    - *Address space*
    - *Program Counter*
  - Threads are multiple program counters
- Kinds of Threads
  - Almost a process
    - *Kernel (Operating System) schedules*
    - *Each thread can make independent system calls*
    - *A blocking system call blocks only that thread*
  - Co-routines and lightweight processes
    - *User schedules (sort of…)*
    - *System calls may block all threads in process*
  - Memory references
    - *Hardware schedules*

Why Use Threads?

- Manage multiple points of interaction
  - Low overhead steering/probing
  - Background checkpoint save
- Alternate method for nonblocking operations
- Hiding memory latency
- Fine-grain parallelism
  - Compiler parallelism



Shared Memory Examples

- Example from a book section on threads programming:
- Int ncount = 0;

```
main(int argc, char **argv){
int ncreate = atoi( argv[1] ), i;
for (i=0; i<ncreate; i++) {
    CreateThread( threadproc );
}
while (ncount < ncreate) ;
}
threadproc() {
    ncount++;
}
```

- Will this program exit?
  (a) Yes (b) No (c) Maybe

C: Maybe

- Memory update operations are not atomic:
  ncount++ becomes (SGI):

  lw r14, ncount
  addu r15,r14,1
  sw r15, ncount
- Any of these could interleave with another thread.

Shared Memory Examples

Threads share data and communicate with each other through simple loads a stores. This is attractive to the programmer but has dangers:

- Thread 0:

```
while (A != 1) ;
B = 1;
A = 2;
```

- Thread 1:

```
B = 2;
A = 1;
while (A == 1) ;
C = B;
```

What is the value of C?

(a) 1 (b) 2 (c) Either (d) None of these

C: Either

- A: Requires that writes be ordered; many processors do not guarantee this as a performance tradeoff (and some architectures have changed their mind)
  - And there is no easy way to enforce this is most languages
  - An entire course can be devoted to the memory consistency models needed to achieve both performance and correctness
  - C “volatile” doesn’t address this problem, you need things like
    __asm__ __volatile__ (""": : :"memory");
    in your code (!)
  - OpenMP provides a FLUSH directive, but this is usually overkill
  - Java has “synchronized” methods (procedures) that guarantee ordering (effectively doing what OpenMP does with FLUSH)

- B: If the compiler doesn’t know that B is “volatile”, it can eliminate the use of B and directly assign 2 to C

MPI and Threads

- MPI describes parallelism between *processes*
- MPI-1 (the specification) is thread-safe
  - This means that the design of MPI has (almost) no global state or other features that prevent an MPI *implementation* from allowing multiple threads to make MPI calls
  - An example is the convenient concept of a “current message” or “current buffer”
    - *MPI’s datatype pack/unpack routines provide a thread-safe alternative*
- *Thread* parallelism provides a shared-memory model within a process
- MPI specifies that MPI calls can only block their thread
- OpenMP and POSIX threads (pthreads) are common
  - OpenMP provides convenient features for loop-level parallelism

MPI-2 Thread Modes

- MPI-2 introduced 4 modes:
  - MPI_THREAD_SINGLE — One thread (MPI_Init)
  - MPI_THREAD_FUNNELED — One thread making MPI calls
  - MPI_THREAD_SERIALIZED — One thread at a time making MPI calls
  - MPI_THREAD_MULTIPLE — Free for all
- Use with MPI_INIT_THREAD(argc,argv,required,&provided)
- Not all MPI implementations are thread-safe
  - Thread-safety is not free
  - If it was, there would be no xlf_r etc.
- Most MPI-1 implementations provide MPI_THREAD_FUNNELLED when linked with other thread libraries (e.g., thread-safe mallocs).
- Coexist with compiler (thread) parallelism for SMPs
- MPI could have defined the same modes on a communicator basis (more natural, and MPICH2 may do this through attributes)

MPI Thread Levels and OpenMP

- MPI_THREAD_SINGLE
  - OpenMP not permitted
- MPI_THREAD_FUNNELED
  - All MPI calls in the “main” thread
  - MPI calls outside of any OpenMP sections
  - Most if not all MPI implementations support this level
- MPI_THREAD_SERIALIZED
  - #pragma omp parallel
    …
    #pragma omp single
    {
      MPI calls allowed here as well
    }
- MPI_THREAD_MULTIPLE
  - Any MPI call anywhere
  - But be careful of races
  - Some MPI implementations (include MPICH2 for some communication methods) support this level

Making an MPI Implementation Thread-safe

- Can you lock around each routine (synchronized in Java terms)?
  - No. Consider a single MPI process with two threads
    T0: MPI_Ssend(itself)
    T1: MPI_Recv(itself)
  - The MPI spec says that this program must work, but if each routine holds a lock, the program will deadlock

Can you lock around just the communication routines?

- That is, can you implement something like this:
- MPI_Recv(…)
  … various setup stuff
  lock(communication)
  if communication would block, release lock and require once communication completes before proceeding
  unlock(communication)
  … various finishing stuff
- Not in general. Replace the MPI_Recv with MPI_Irecv:
- MPI_Irecv(…, datatype, …, communicator, &request)
  … various setup stuff
  lock(communication)
  release if necessary
  unlock(communication)
  … various finishing stuff
- The problem is with the datatype and communicator. If the MPI_Irecv did not match the message, then it left a “posted” receive in a queue that will be matched later by an arriving message
- Processing this message requires using the datatype and communicator
- MPI uses reference count semantics on these objects, implicitly incrementing the reference count in the MPI_Irecv
- This ref count must be atomically updated (as in the first thread example)

What you can do

- Coarse Grain
  - Use the one-big-lock approach, but be sure to release/re-acquire it around any blocking operation
  - Don't forget to use the lock around any update of data that might be shared
- Fine Grain
  - Identify the shared items and ensure that updates are atomic
  - Benefit: You may be able to avoid using locks
  - Cost: There is more the think about and there can be "dining philosopher" deadlocks if more than one critical section must be acquired at a time
  - What are the items for MPI?
    - *We've looked at the ~305 functions and MPI and found the following classes:*

Thread Safety Needs of MPI Functions

- None: The function has no thread-safety issues
  Examples: MPI_Address, MPI_Wtick
- Access Only: The function accesses fixed data for an MPI object, such as the size of a communicator. This case differs from the ``None'' case because an erroneous MPI program could free the object in a race with a function that accesses the read-only data.
  Examples: MPI_Comm_Rank, MPI_Get_count.

Thread Safety Needs of MPI Functions

- Update Ref: The function updates the reference count of an MPI object.
  Examples: MPI_Comm_group, MPI_File_get_view
- Comm/IO: The function needs to access the communication or I/O system in a thread-safe way. This is a very coarse-grained category but is sufficient to provide thread safety.
  Examples: MPI_Send, MPI_File_read
- Collective: The function is collective. MPI requires that the user not call collective functions on the same communicator in different threads in a way that may make the order of invocation depend on thread timing (race). The communication part of the collective function is assumed to be handled separately through the communication thread locks.
  Examples: MPI_Bcast, MPI_Comm_spawn

Thread Safety Needs of MPI Functions

- Read List: The function returns an element from a list of items, such as an attribute or info value.
  Examples: MPI_Info_get, MPI_Comm_get_attr.
- Update List: The function updates a list of items that may also be read. Multiple threads are allowed to simultaneously update the list, so the update implementation must be thread safe.
  Examples: MPI_Info_set, MPI_Type_delete_attr.

Thread Safety Needs of MPI Functions

- Allocate: The function allocates an MPI object (may also need memory allocation such as with malloc).
  Examples: MPI_Send_init, MPI_Keyval_create.
- Own: The function has its own thread-safety management.
  Examples: MPI_Buffer_attach, MPI_Cart_create.
- Other: Special cases. Examples: MPI_Abort and MPI_Finalize.

Other Issues

- Thread-Private Memory
  - Values that are global to a thread but private to that thread are sometimes needed. In MPICH2, these are used to implement a "nesting" level (used to ensure that the correct error handling routine is called if the routine is used in a nested fashion within the MPI implementation) and for performance and debugging statistics.
- Memory consistency
  - When heavy-weight thread lock/unlocks (and related critical sections or monitors) are not used, the implementation must ensure that the necessary write ordering is enforced and that values that the compiler may leave in register are marked volatile
- Thread failure
  - A major problem with any lock-based thread-safety model is defining what happens when a thread fails or is canceled (e.g., with pthread_cancel).

Other Issues con't

- Performance and code complexity
  - The advantage of the "one big lock" is its (relative) simplicity
  - It serializes MPI function execution among threads, potentially impacting performance.
  - Fine grain locks avoid (much of) the serialization, but at added complexity. In addition, they can be *more* costly if a routine must acquire multiple fine grain locks rather than a single coarse grain lock
- Thread scheduling
  - Should a thread busy wait or let the OS (or another thread) schedule it? Can condition variables be used? A problem is not all events may wake up a thread, particularly when low-latency shared memory is being used between processes.
- What level of thread support should an MPI implementation provide?
  - Performance matters…

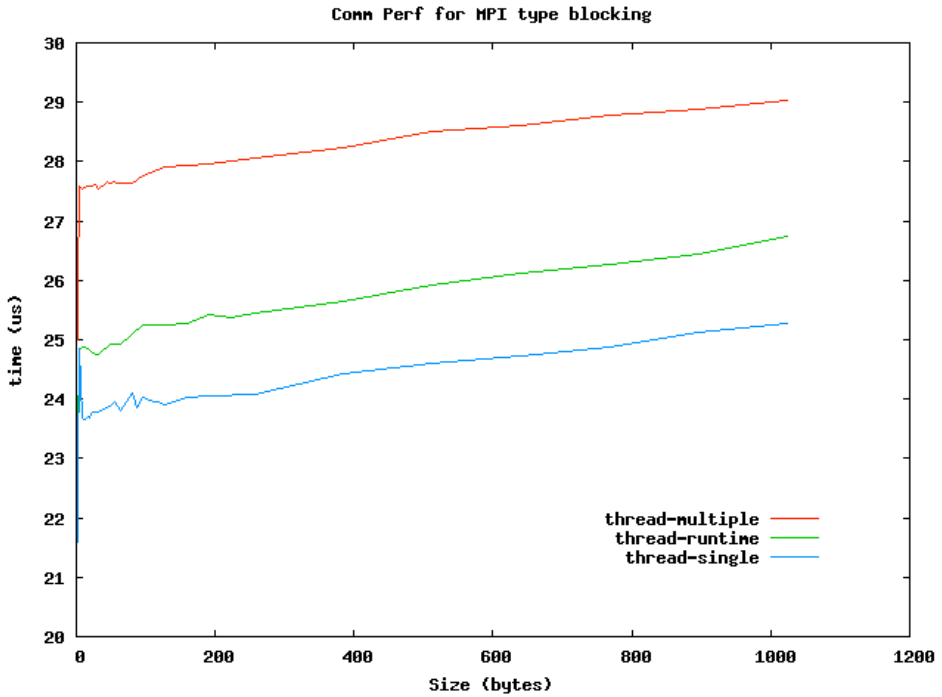
Performance Issues with Threads

- MPI_THREAD_SINGLE
  - No thread-shared data structures in program. All operations proceed without locks
- MPI_THREAD_FUNNELLED
  - MPI data structures do not need locks, but other operations (e.g., system calls) must use thread-safe versions.
- MPI_THREAD_SERIALIZED
  - Almost like MPI_THREAD_FUNNELLED, except some MPI operations may need to be completed before changing the thread that makes MPI calls
- MPI_THREAD_MULTIPLE
  - All MPI data structures need locks or other atomic access methods

- What are the performance consequences of these thread levels? Just how much does THREAD_MULTIPLE cost?

Thread Overhead in MPICH2

- Three versions of MPICH2, configured with –enable-threads=
  - multiple
    - *Always MPI_THREAD_MULTIPLE*
  - single
    - *Always MPI_THREAD_SINGLE*
  - runtime
    - *Thread level is MPI_THREAD_FUNNELLED unless THREAD_MULTIPLE is explicitly selected with MPI_Thread_init*
  - Ch3:sock (sockets communication only, using TCP, on a single SMP (OS can optimize communication
  - Mpptest (ping-pong latency test)



What you've forgotten

- Collective communications operations
  - Performed on a group of processes described by a *communicator*
- In a typical MPI implementation, all processes in a communicator share a *context id*, typically implemented as an integer value. All processes must agree on this value
  - (Other implementations are possible, but this is the easiest, most scalable choice)
- Determining this context id requires an agreement among the processes
- First, a short aside on *why* context ids are important

Why Contexts?

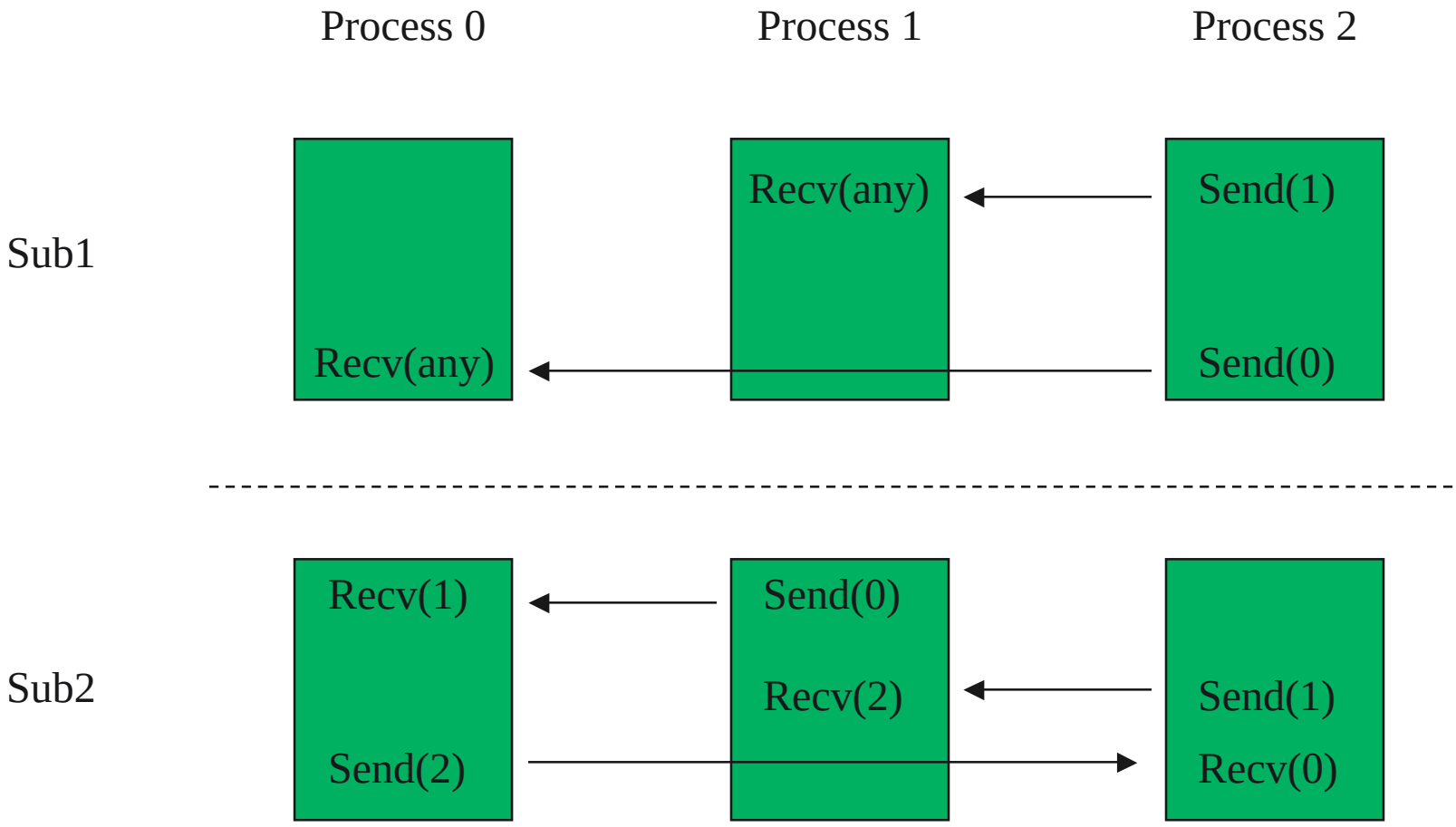
- Parallel libraries require isolation of messages from one another and from the user that cannot be adequately handled by tags.
- The following horrible examples are due to Marc Snir.
  - Sub1 and Sub2 are from different libraries

```
Sub1();
Sub2();
```

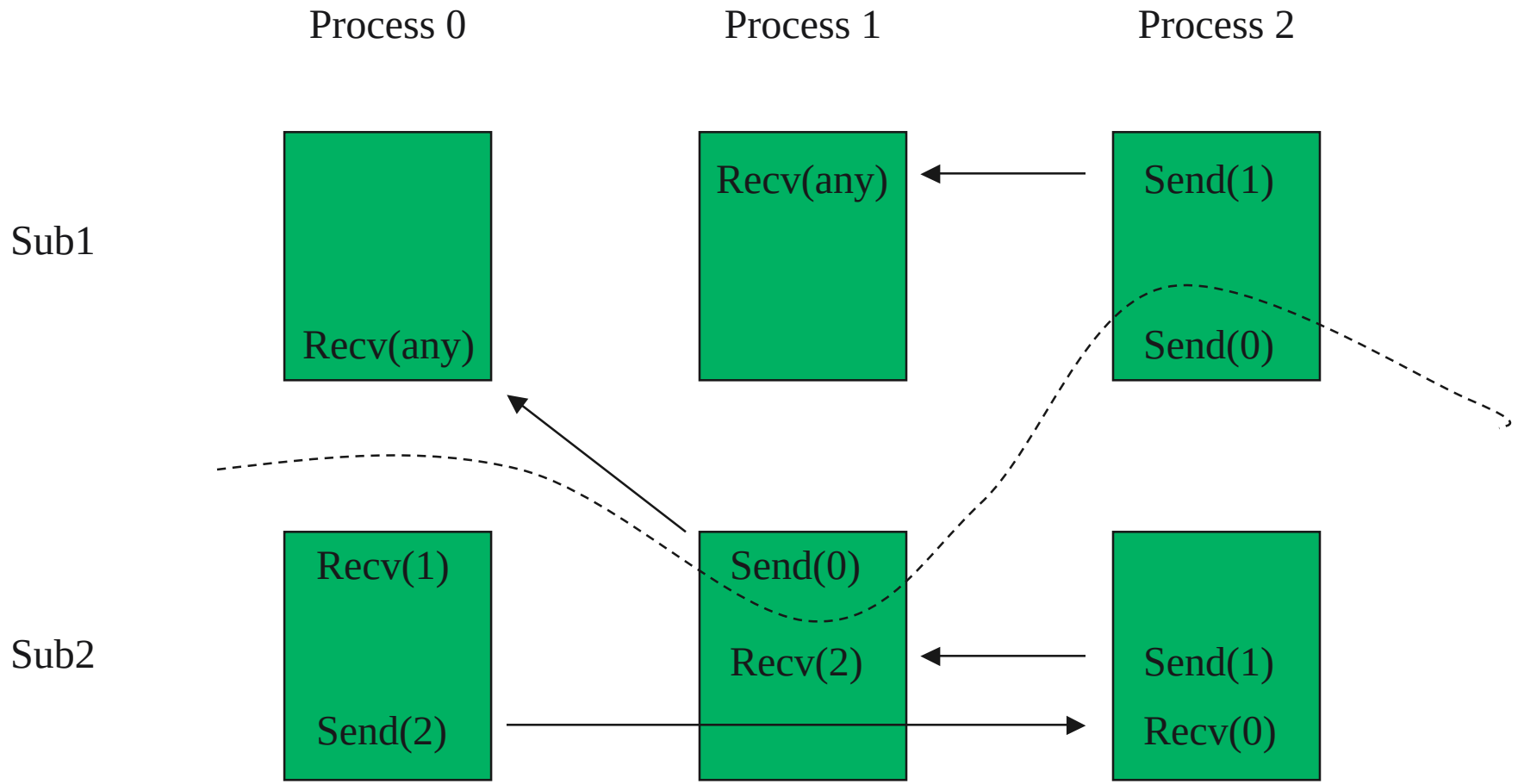
  - Sub1a and Sub1b are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

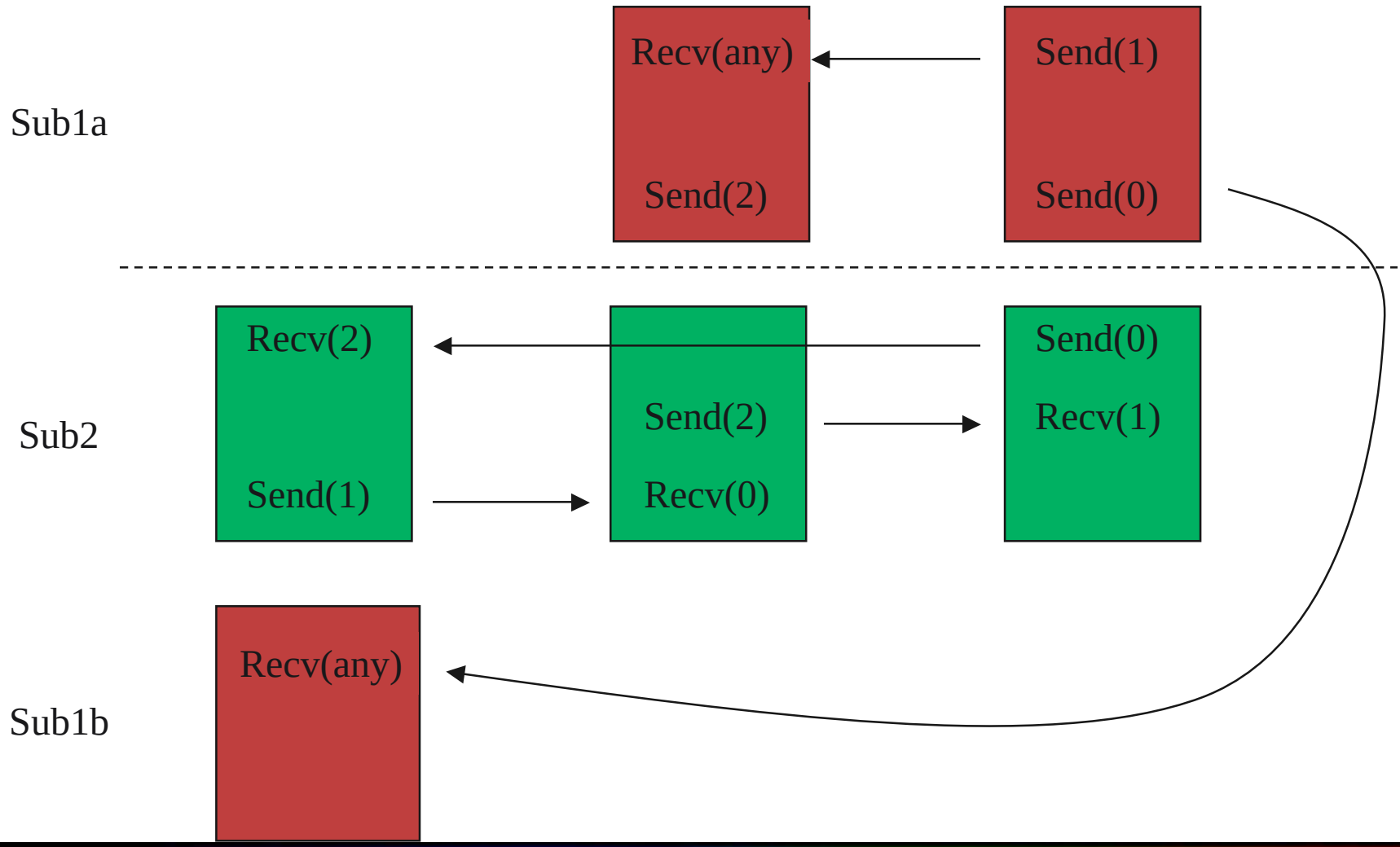
Correct Execution of Library Calls



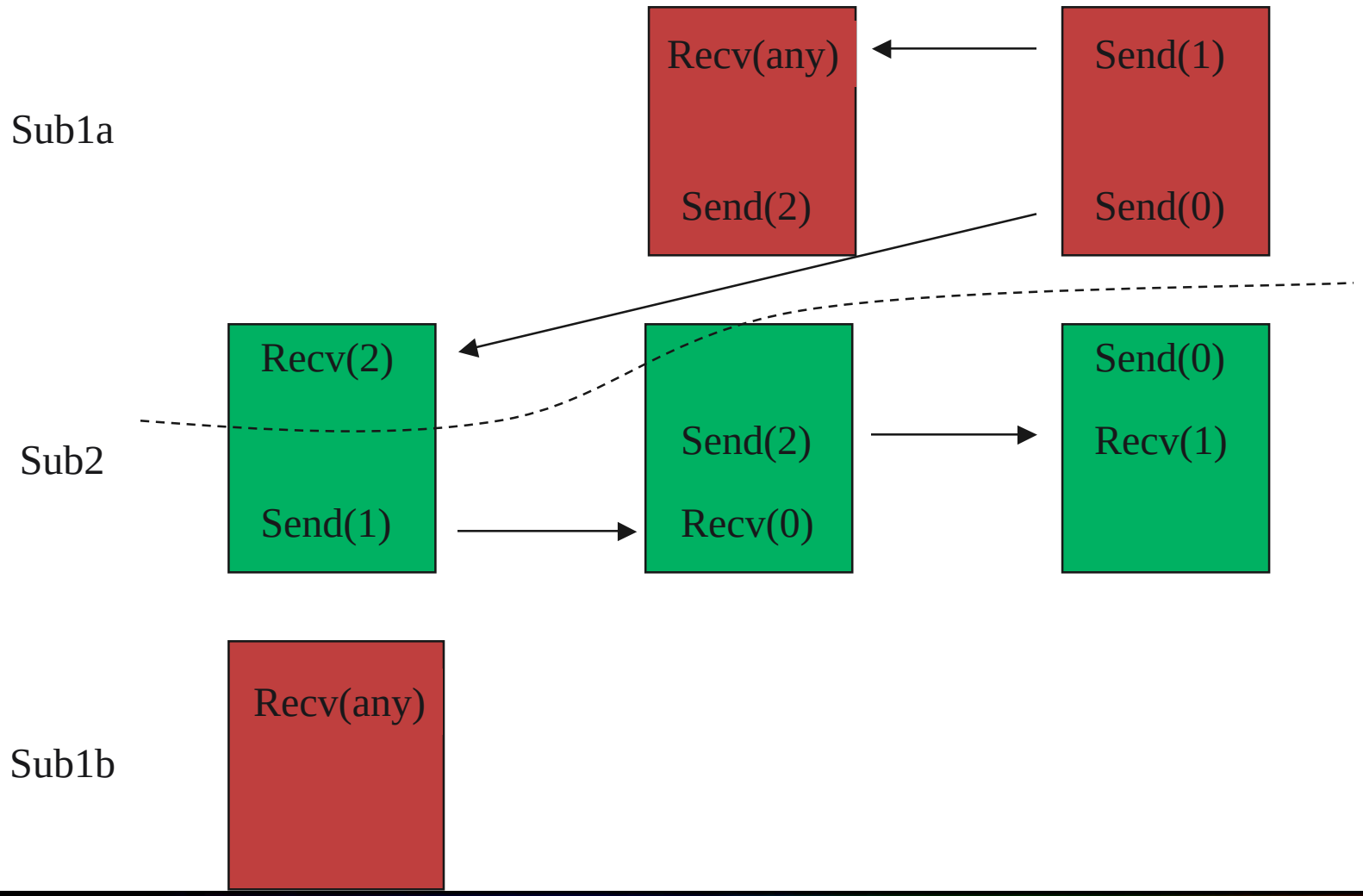
Incorrect Execution of Library Calls



Correct Execution of Library Calls with Pending Communication



Incorrect Execution of Library Calls with Pending Communication

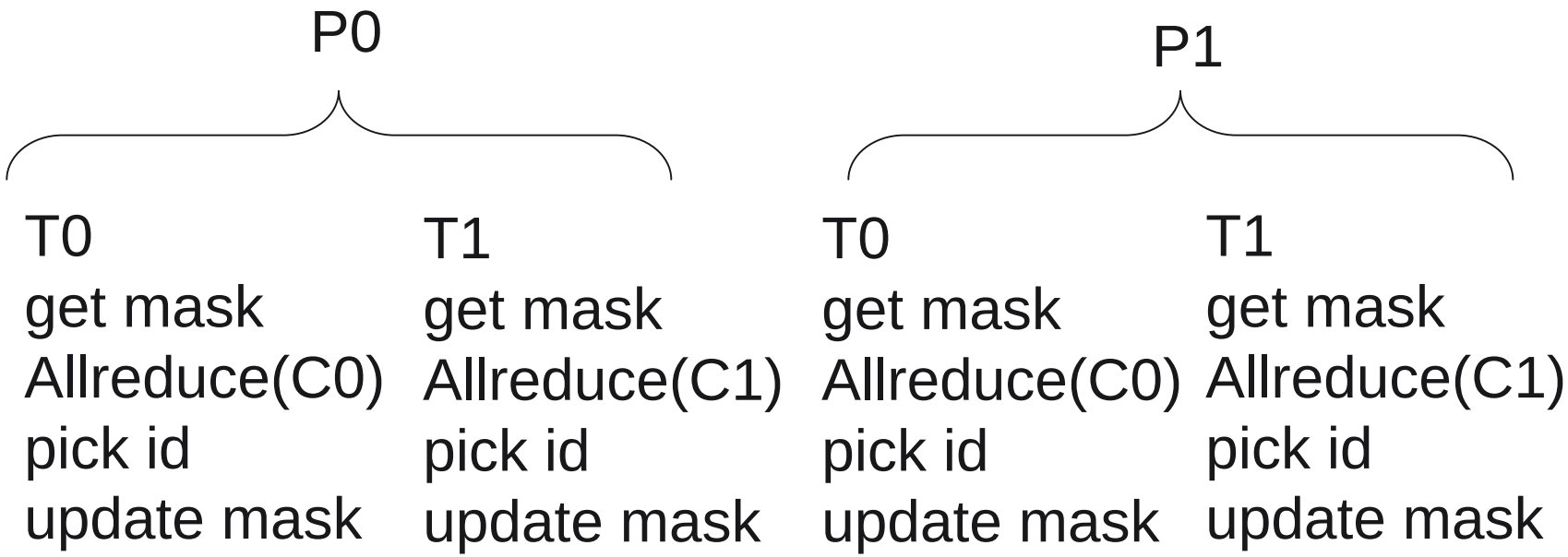


Consider MPI_Comm_dup

- Comm_dup simply creates a copy of a communicator with a new context id
  - Used to guarantee message separation between modules
  - All processes in the input communicator must call (and follow collective semantics)
- A simple algorithm (for the single threaded case):
  - Each process duplicates the data structure representing the group of processes (shallow dup; increment reference count)
  - All participating processes perform and MPI_Allreduce on a bit mask of available context id values, using MPI_BAND (bitwise AND), using the original (input) communicator

What can go wrong in the multithreaded case

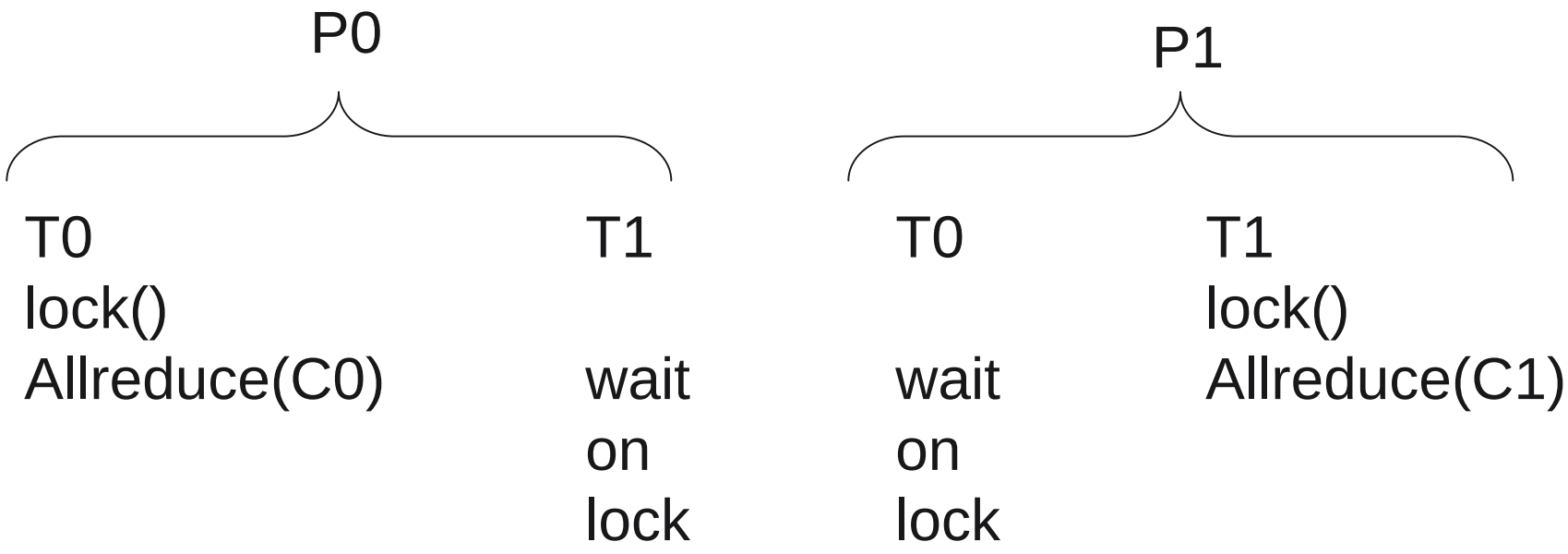
- Consider this case with two processes, each with two threads, each doing an MPI_Comm_dup on two communicators



All four threads (both new communicators) get the *same* context id. We clearly need to atomically update the mask.

What can go wrong in the multithreaded case

- Consider this case with two processes, each with two threads:



Deadlock — Neither Allreduce will complete, so the lock will never be released

An efficient algorithm

- Definition of efficient
  - Requires no more communication between processes than the single-threaded algorithm in the common case (only one thread on each process creating a new communicator)
- Speculative approach
  - Try to succeed and have a backup plan on failure
  - RISC systems use this approach instead of atomic memory updates (load-link, store-conditional and similar names)

The Idea

- Atomically
  - Keep track that the mask is in use
  - If mask was not in use,
    - *make a copy of the mask*
  - Else
    - *Use all zero bits for mask*
- MPI_Allreduce(mask) (the regular algorithm
- If found a value (no process had all zeros)
  - Atomically
    - *Remove selected bit from mask*
    - *Clear in-use*
  - Return context value
- Else
  - Try again

What happens in the typical case?

- A single thread from each process is trying to get a new context id
  - Gets mask (all processes get mask, none get zero masks)
  - Allreduce finds a free bit
  - All processes remove the same bit from their mask
  - Context id returned
- Same communication cost as single-threaded algorithm
- Only two thread locks an increment, test, decrement in addition to single-threaded algorithm

What happens when there are competing threads?

- How do we avoid this case:
  - Each process has 2 threads, call them A and B
  - On some processes, thread A call MPI_Comm_dup first, on others thread B calls MPI_Comm_dup first.
  - As a result, some thread calling MPI_Allreduce always provides a zero-mask because the other thread got to the mask first
  - This is live lock (as opposed to dead lock) because the threads never stop working. They just never get anywhere…

Avoiding Live Lock

- When multiple threads discover that they are contending for the same resource, they need to ensure that they can make progress.
  - One way to do this is to order the threads so that the group of threads (e.g., the “A” threads) is guaranteed to get access to the resource (the context id mask in our case)
  - We need a way to sort the threads that gives the same ordering for threads on different processes
- Use the context id of the *input* communicator
  - All processes have the same context id value for the same communicator
  - Let the thread with the minimum context id value take the mask
  - Repeat that test each iteration (in case a new thread arrives)

A Fast, Thread-Safe Context-Id Algorithm

```
/* global variables (shared among threads of a process) */
mask        /* bit mask of context ids in use by a process */
mask_in_use       /* flag; initialized to 0 */
lowestContextId   /* initialized to MAXINT */

/* local variables (not shared among threads) */
local_mask     /* local copy of mask */
i_own_the_mask /* flag */
context_id     /* new context id; initialized to 0 */

while (context_id == 0) {
    Mutex_lock()
    if (mask_in_use || MyComm->contextid > lowestContextId) {
        local_mask = 0 ; i_own_the_mask = 0
        lowestContextId = min(lowestContexId,MyComm->contextid)
    }
    else {
        local_mask = mask ; mask_in_use = 1 ; i_own_the_mask = 1
        lowestContextId = MyComm->contextid
    }
    Mutex_unlock()
    MPI_Allreduce(local_mask, MPI_BAND)
    if (i_own_the_mask) {
        Mutex_lock()
        if (local_mask != 0) {
            context_id = location of first set bit in local_mask
            update mask
            if (lowestContextId == MyComm->contextid) {
                lowestContextId = MAXINT;
            }
        }
        mask_in_use = 0
        Mutex_unlock()
    }
}
return context_id
```

Conclusions

- MPI (the specification) is thread-safe
- MPI (an implementation) can be made thread-safe but there are some subtle issues
- We can also say something about threads as a programming model
  - Yuk!
  - Locks are bad (state)
  - Memory flags are bad (ordering/consistency)
  - Little help from language system (volatile not enough)