Timo Schneider (subst. Torsten Hoefler)
Parallel Programming
Finish STM & Distributed Memory Programming: Actors, CSP, and MPI

Microsoft, Google: We've found a fourth data-leaking Meltdown-Spectre CPU hole
Design blunder exists in Intel, AMD, Arm, Power processors
By Chris Williams, Editor in Chief 21 May 2018 at 23:00

How the fourth variant works
Variant 4 is referred to as a speculative store bypass. It is yet another "wait, why didn't I think of that?" design oversight in modern out-of-order-executor engineering. And it was found by Google Project Zero's Jann Horn, who helped uncover the earlier Spectre and Meltdown bugs, and Ken Johnson of Microsoft.

It hinges on the fact that when faced with a bunch of software instructions that store data to memory, the CPU will look far ahead to see if it can execute any other instructions out of order while the stores complete. Writing to memory is generally slow compared to other instructions. A modern fast CPU won't want to be held up by store operations, so it looks ahead to find other things to do in the meantime.

If the processor core, while looking ahead in a program, finds an instruction that loads data from memory, it will predict whether or not this load operation is affected by any of the preceding stores. For example, if a store is writing to memory that a later load fetches back from memory, you'll want the store to complete first. If a load is predicted to be safe to run ahead of the pending stores, the processor executes it speculatively while other parts of the chip are busy with other code.
Bank account (ScalaSTM)

class AccountSTM {
    private final Integer id; // account id
    private final Ref.View<Integer> balance;

    AccountSTM(int id, int balance) {
        this.id = new Integer(id);
        this.balance = STM.newRef(balance);
    }
}
void withdraw(final int amount) {
    // assume that there are always sufficient funds...
    atomic {
        int old_val = balance.get();
        balance.set(old_val - amount);
    }
}

void deposit(final int amount) {
    atomic {
        int old_val = balance.get();
        balance.set(old_val + amount);
    }
}
void withdraw(final int amount) {
    // assume that there are always sufficient funds...
    STM.atomic(new Runnable() { public void run() {
       int old_val = balance.get();
       balance.set(old_val - amount);
    }});
}

void deposit(final int amount) {
    STM.atomic(new Runnable() { public void run() {
       int old_val = balance.get();
       balance.set(old_val + amount);
    }});
}
GetBalance (return a value)

```java
public int getBalance() {
    int result = STM.atomic(
        new Callable<Integer>() {
            public Integer call() {
                int result = balance.get();
                return result;
            }
        });
    return result;
}
```
Bank account transfer

```java
static void transfer(final AccountSTM a, final AccountSTM b, final int amount) {
    atomic {
        a.withdraw(amount);
        b.deposit(amount);
    }
}
```

What if account a does not have enough funds?
How can we wait until it does in order to retry the transfer?

locks → conditional variables

TM → retry
Bank account transfer with retry

```java
static void transfer_retry(final AccountSTM a,
                          final AccountSTM b,
                          final int amount) {

    atomic {
        if (a.balance.get() < amount)
            STM.retry();
        a.withdraw(amount);
        b.deposit(amount);
    }
}
```

**retry**: abort the transaction and retry when conditions change
How does retry work?

Implementations need to track what reads/writes a transaction performed to detect conflicts

- Typically called **read-/write-set of a transaction**
- When retry is called, transaction aborts and will be retried when *any of the variables that were read*, change
- In our example, when a.balance is updated, the transaction will be retried
Simplest STM Implementation

Ingredients

Threads that run transactions with thread states
- active
- aborted
- committed

Objects representing state stored in memory (the variables affected by a transaction)
- offering methods like a constructor, read (get), write (set)
- and copy!
Clock-based STM System

- Transaction A
  - birthdate
  - read at start

- Transaction C
  - commits
  - increases

- Transaction B
  - birthdate
  - read at start

global clock
Atomic Objects

Each transaction uses a local **read-set** and a local **write-set** holding all locally read and written objects.

Transaction calls **read**
- check if the object is in the write set → return this (new) version
- otherwise check if object's time stamp ≤ transaction's birthdate, if not throw aborted exception, otherwise add new copy of the object to the read set

Transaction calls **write**
- if object is not in write set, create a copy of it in the write set
Transaction life time

- Birthdate of T
- Read set of T: T reads Y, T reads X, T reads Z
- Time:
  - X.date
  - Y.date
  - Z.date
Commit

- Lock all objects of read- and write-set (in some defined order to avoid deadlocks)
- Check that all objects in the read set provide a time stamp $\leq$ birthdate of the transaction, otherwise return "abort"
- Increment and get the value $T$ of current global clock
- Copy each element of the write set back to global memory with timestamp $T$
- Release all locks and return "commit"
Successful commit
Aborted commit

X.date  Y.date  Z.date

birthdate of T

T reads Y  T reads X

T writes Y (local copy!)  T writes Z (local copy!)

read set of T

write set of T

T commits
Dining philosophers

- 5 philosophers
- 5 forks
- each philosopher requires 2 forks to eat
- forks cannot be shared

Solution that can lead to deadlock

Philosopher:
- think
- lock left
- lock right
- eat
- unlock right
- unlock left

\[ P_1 \text{ takes } F_1, \ P_2 \text{ takes } F_2, \ P_3 \text{ takes } F_3, \ P_4 \text{ takes } F_4, \ P_5 \text{ takes } F_5 \rightarrow \text{Deadlock} \]
Dining Philosophers Using TM

```java
private static class Fork {
    public final Ref.View<Boolean> inUse = STM.newRef(false);
}
class PhilosopherThread extends Thread {
    private final int meals;
    private final Fork left;
    private final Fork right;

    public PhilosopherThread(Fork left, Fork right) {
        this.left = left;
        this.right = right;
    }

    public void run() { ... }
}
```
Dining Philosophers Using TM

```java
Fork[] forks = new Fork[tableSize];

for (int i = 0; i < tableSize; i++)
    forks[i] = new Fork();

PhilosopherThread[] threads = new PhilosopherThread[tableSize];

for (int i = 0; i < tableSize; i++)
    threads[i] = new PhilosopherThread(forks[i],
                                        forks[(i + 1) % tableSize]);
```
Dining Philosophers Using TM

class PhilosopherThread extends Thread {
    ...
    public void run() {
        for (int m = 0; m < meals; m++) {
            // THINK
            pickUpBothForks();
            // EAT
            putDownForks();
        }
    }
    ...
}
Dining Philosophers Using TM

class PhilosopherThread extends Thread {
    ...
    private void pickUpBothForks() {
        STM.atomic(new Runnable() {
            public void run() {
                if (left.inUse.get() || right.inUse.get())
                    STM.retry();

                left.inUse.set(true);
                right.inUse.set(true);

            }
        });
    }
    ...
}
Dining Philosophers Using TM

class PhilosopherThread extends Thread {
  ...
  private void putDownForks() {
    STM.atomic(new Runnable() {
      public void run() {
        left.inUse.set(false);
        right.inUse.set(false);
      }
    });
  }
  ...
}
Issues with transactions

- It is not clear what are the best semantics for transactions
- Getting good performance can be challenging
- I/O operations (e.g., print to screen)
  Can we perform I/O operations in a transaction?
Summary

- Locks are too hard!
- Transactional Memory tries to remove the burden from the programmer
- STM / HTM

- Remains to be seen whether it will be widely adopted in the future
Additional Reading

Simon Peyton Jones,
*Beautiful concurrency*

Dan Grossman,
*The Transactional Memory / Garbage Collection Analogy*
Distributed Memory & Message Passing
So far

Considered

- Parallel / Concurrent
- Fork-Join / Threads
- OOP on Shared Memory
- Locking / Lock Free / Transactional
- Semaphores / Monitors
Sharing State

Many of the problems of parallel/concurrent programming come from sharing state

- Complexity of locks, race conditions, ....

What if we avoid sharing state?
Alternatives

Functional Programming
- Immutable state → no synchronization required

**Message Passing: Isolated** mutable state
- State is mutable, but not shared: Each thread/task has its private state
- Tasks cooperate via message passing
Concurrent Message Passing

Programming Models
- CSP: Communicating Sequential Processes
- Actor programming model

Framework/library
- MPI (Message Passing Interface)
Shared vs Distributed memory
Isolated mutable state

Tasks exchange messages

Mutable (private) state
Example: Shared state counting

→ shared state must be protected (lock/atomic counter)
Isolated mutability: counting

\[
\begin{align*}
&\uparrow \quad \uparrow \\
&\text{Local cnt} \\
&\text{inc()} \\
&\text{Local cnt} \\
&\text{inc()} \\
&\text{Local cnt} \\
&\text{inc()} \\
&\text{Local cnt} \\
&\text{inc()}
\end{align*}
\]
Isolated mutability: accessing count

Local cnt

Local cnt

Local cnt

Local cnt
Rethinking managing state

Bank account
– Sequential programming
  ● *Single balance*

– Parallel programming: sharing state
  ● *Single balance* + *protection*

– Parallel programming: distributing state
  ● *Each thread has a local balance (a budget)*
  ● *Threads exchange amounts at coarse granularity*
Distributed Bank account

Total balance: $100 + 300 + 150 = 550$

- Each task can operate independently
- And communicate with other tasks only when needed
  - This lecture: via messaging
Synchronous vs Asynchronous messages

Synchronous:
– sender blocks until message is received

Asynchronous:
– sender does not block (fire-and-forget)
– placed into a buffer for receiver to get
The Actor Model*

Actor = Computational agent that maps communication to

- a finite set of communications sent to other actors (messages)
- a new behavior (state)
- a finite set of new actors created (dynamic reconfigurability)

- Undefined global ordering
- Asynchronous Message Passing
- Invented by Carl Hewitt 1973**


**Carl Hewitt; Peter Bishop and Richard Steiger (1973). A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI.
The Actor Model

Actor model provides a dynamic interconnection topology
- dynamically configure the graph during runtime (add channels)
- dynamically allocate resources

An actor sends messages to other actors using "direct naming", without indirection via port / channel / queue / socket (etc.)

Implemented in various languages such as Erlang, Scala, Ruby and in frameworks such as Akka (for Scala and Java)
Event-driven programming model

Typically actors react to messages

- Event-driven model

A program is written as a set of event handlers for events (events can be seen as received messages)

Example: Graphical User Interface

- user presses OK button → ...
- user presses Cancel button → ...
- ...

Example: Erlang

Functional Programming Language

- code might look unconventional at first

Developed by Ericsson for distributed fault-tolerant applications

- if no state is shared, recovering from errors becomes much easier

Open source

Concurrent, follows the actor model

```erlang
-module(pingpong).
-export([start/1, ping/2, pong/0]).

ping(0, Pong_Node) ->
  {pong, Pong_Node} ! finished,
  io:format("ping finished~n", []).

ping(N, Pong_Node) ->
  {pong, Pong_Node} ! {ping, self()},
  receive
    pong ->
      io:format("Ping received pong~n", []),
      end,
      ping(N - 1, Pong_Node).
  end.

pong() ->
  receive
    finished ->
      io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
      io:format("Pong received ping~n", []),
      Ping_PID ! pong,
      pong()
      end.

start(Ping_Node) ->
  register(pong, spawn(pingpong, pong, [])),
  spawn(Ping_Node, pingpong, ping, [3, node()]).
```
Erlang example

```
start() ->
    Pid = spawn(fun() -> hello() end),
    Pid ! hello,
    Pid ! bye.

hello() ->
    receive
        hello ->
            io:fwrite("Hello world\n"),
            hello();
        bye ->
            io:fwrite("Bye cruel world\n"),
            ok
    end.
```

- **new task (actor) that will execute the hello function**
- **spawn** returns address (Pid) of new task
- **Address (Pid) can be used to send messages to task**
- **Messages sent to a task are put in a mailbox**
- **Receive** reads the first message in the mailbox, which is matched against patterns (similar to a switch statement)
- **Event-driven programming:** code is structured as reactions to events
Actor example: distributor

- Forward received messages to a set of nodes in a round-robin fashion
Actor example: distributor

State:
– an array of actors
– the array index of the next actor to forward a message

Receive:
– messages → forward message and increase index (mod)
– control commands (e.g., add/remove actors)
Actor example: serializer

```
4 1
2 3

unordered

serializer

4 3 2 1

ordered
```

ordered

unordered
Actor example: serializer

State:
- a sorted list of items we have received
- the last item we forwarded

```
<table>
<thead>
<tr>
<th>last item</th>
<th>sorted list of pending items</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12 14 16 17</td>
</tr>
</tbody>
</table>
```
Actor example: serializer

Receive: If we receive an item that is larger than the last item plus one:
– add it to the sorted list

Example: receive 13

<table>
<thead>
<tr>
<th>old state:</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>new state:</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Actor example: serializer

- Receive: If we receive an item that is equal to the last item plus one:
  - send the received item plus all consecutive items from the list
  - reset the last item

Example: receive 11

<table>
<thead>
<tr>
<th>old state:</th>
<th>10</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
</table>

new state: 14 16 17

send

| 14 | 13 | 12 | 11 |
Communicating Sequential Processes (1978, 1985)

Sir Charles Antony Richard Hoare (aka C.A.R. / Tony Hoare)

*Formal* language defining a process algebra for concurrent systems. Operators `seq` (sequential) and `par` (parallel) for the hierarchical composition of processes. Synchronisation and Communication between parallel processes with Message Passing.

- Symbolic channels between sender and receiver
- Read and write requires a rendezvouz (synchronous!)

CSP was first implemented in Occam.
CSP: Indirect Naming

- Many message passing architectures (such as CSP) include an intermediary entity (port/channel) to address send destination.
- Process issuing send() specifies the port to which the message is sent.
- Process issuing receive() specifies a port number and waits for the first message that arrives at the port.
CSP Example (from Hoare's seminal Paper)

Conway's Problem

- Write a program that transforms a series of cards with 80-character columns in a series of printing lines with 125 characters each. Replace each "**" by "^"

- Separation into processes (Threads)
  R par C par P
  - R: Reading process reading 80-character records
  - C: Converting process converting "**" into "^"
  - P: Printing process: write records with 125 characters
CSP Example (from Hoare's seminal Paper)

\[
[\text{west} :: \text{DISASSEMBLE}] \mid \mid \text{X} :: \text{SQUASH} \mid \mid \text{east} :: \text{ASSEMBLE}]
\]

SQUASH

\[
\text{X} ::
\ast [\text{c:character}; \text{west}?\text{c} \rightarrow
  \begin{align*}
  \text{[c \# asterisk} & \rightarrow \text{east)!c} \\
  \text{|c = asterisk} & \rightarrow \text{west}?\text{c};
  \end{align*}
\]

\[
\begin{align*}
  \text{[c \# asterisk} & \rightarrow \text{east)!asterisk; east)!c} \\
  \text{|c = asterisk} & \rightarrow \text{east!upward arrow}
  \end{align*}
\]

Repetition of guarded command

Guarded receive

Blocking send

Guarded alternatives
OCCAM

First programming language to implement CSP (1983)

ALT
    count1 < 100 & c1 ? data
    SEQ
        count1 := count1 + 1
        merged ! data
    count2 < 100 & c2 ? data
    SEQ
        count2 := count2 + 1
        merged ! data
status ? request
    SEQ
        out ! count1
        out ! count2
Go programming language

Concurrent programming language from Google

Language support for:

- Lightweight tasks (called goroutines)
- Typed channels for task communications
  - channels are synchronous (or unbuffered) by default
  - support for asynchronous (buffered) channels

Inspired by CSP
Language roots in Algol Family: Pascal, Modula, Oberon [Prof. Niklaus Wirth, ETH]
[One of the inventors, Robert Griesemer: PhD from ETH]
func main() {
    msgs := make(chan string)
done := make(chan bool)

go hello(msgs, done);

msgs <- "Hello"
msgs <- "bye"

ok := <-done

fmt.Println("Done:", ok);
}

func hello(msgs chan string,
done chan bool) {
    for {
        msg := <-msgs
        fmt.Println("Got:", msg)
        if msg == "bye" {
            break
        }
    }

done <- true;
}
Go example

```go
func main() {
    msgs := make(chan string)
    done := make(chan bool)

    go hello(msgs, done);

    msgs <- "Hello"
    msgs <- "bye"

    ok := <- done
    fmt.Println("Done:", ok);
}

func hello(msgs chan string, done chan bool) {
    for {
        msg := <- msgs
        fmt.Println("Got:", msg)

        if msg == "bye" {
            break
        }
    }

    done <- true;
}
```

Create two channels:
- `msgs`: for strings
- `done`: for boolean values
func main() {
    msgs := make(chan string)
    done := make(chan bool)

    go hello(msgs, done);

    msgs <- "Hello"
    msgs <- "bye"

    ok := <-done

    fmt.Println("Done:", ok);
}

func hello(msgs chan string, done chan bool) {
    for {
        msg := <-msgs
        fmt.Println("Got:", msg)
        if msg == "bye" {
            break
        }
    }

    done <- true;
}
func main() {
    msgs := make(chan string)
    done := make(chan bool)

    go hello(msgs, done);

    msgs <- "Hello"
    msgs <- "bye"

    ok := <-done

    fmt.Println("Done:", ok);
}

func hello(msgs chan string,
            done chan bool) {

    for {
        msg := <-msgs
        fmt.Println("Got:", msg)

        if msg == "bye" {
            break
        }
    }

    done <- true;
}
Go example

```go
func main() {
    msgs := make(chan string)
done := make(chan bool)
go hello(msgs, done);
    msgs <- "Hello"
    msgs <- "bye"
    ok := <-done
    fmt.Println("Done:", ok);
}

func hello(msgs chan string, done chan bool) {
    for {
        msg := <-msgs
        fmt.Println("Got:", msg)
        if msg == "bye" {
            break
        }
    }
    done <- true;
}
```

Write arguments to msgs channel

Read result via done channel
Q: what will happen in this program?

```go
func t(in chan string, done chan bool) {
    m := <-in // receive from in channel
    fmt.Println("Got message:", m); // print received message
    done <- true // send true to done channel
}

func main() {
    c := make(chan string) // create a string channel
    done := make(chan bool) // create a boolean channel

    go t(c, done) // spawn goroutine

    ok := <-done // receive from done channel
    fmt.Println("Got ok:", ok); // print ok
    c <- "Hello" // send hello to channel c
}
```

A: fatal error: all goroutines are asleep - deadlock!
Example: Concurrent prime sieve

Each station removes multiples of the first element received and passes on the remaining elements to the next station.
Concurrent prime sieve

```go
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}

func Filter(in <-chan int, out chan<- int, prime int) {
    for {
        i := <-in // Receive value from 'in'.
        if i%prime != 0 {
            out <- i // Send 'i' to 'out'.
        }
    }
}

func main() {
    ch := make(chan int)
    go Generate(ch)
    for i := 0; i < 10; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go Filter(ch, ch1, prime)
        ch = ch1
    }
}
```

source code from golang.org
Message Passing Interface (MPI)
Message Passing Interface (MPI)

Message passing libraries:
- PVM (Parallel Virtual Machines) 1980s
- **MPI** (Message Passing Interface) 1990s

**MPI = Standard API**
- Hides Software/Hardware details
- Portable, flexible
- Implemented as a library
Process Identification

- MPI processes can be collected into groups
  - Each group can have multiple colors (some times called context)
  - Group + color == communicator (it is like a name for the group)
  - When an MPI application starts, the group of all processes is initially given a predefined name called **MPI_COMM_WORLD**
    - The same group can have many names, but simple programs do not have to worry about multiple names
- A process is identified by a unique number within each communicator, called *rank*
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator
MPI Communicators

- Defines the communication domain of a communication operation: set of processes that are allowed to communicate with each other.

- Initially all processes are in the communicator MPI_COMM_WORLD.

- The rank of processes are associated with (and unique within) a communicator, numbered from 0 to n-1.
Communicators

When you start an MPI program, there is one predefined communicator MPI_COMM_WORLD.

Communicators do not need to contain all processes in the system.

Every process in a communicator has an ID called as “rank”.

The same process might have different ranks in different communicators.

Communicators can be created “by hand” or using tools.

Simple programs typically only use the predefined communicator MPI_COMM_WORLD (which is sometimes considered bad practice).

When you start an MPI program, there is one predefined communicator MPI_COMM_WORLD.

Can make copies of this communicator (same group of processes, but different “aliases”).

mpiexec -np 16 ./test
Processes are identified by nonnegative integers, called \textit{ranks}.

\( p \) processes are numbered 0, 1, 2, .. \( p-1 \)

```java
public static void main(String args []) throws Exception {
    MPI.Init(args);
    // Get total number of processes (p)
    int size = MPI.COMM_WORLD.Size();
    // Get rank of current process (in \([0..p-1]\))
    int rank = MPI.COMM_WORLD.Rank();
    MPI.Finalize();
}
```
**SPMD**

**Single Program**

**Multiple Data**
(Multiple Instances)

we compile one program

the if-else makes it SPMD

```c
if (rank == 0)
    do this
else
    do that
```

```
if (rank == 0)
    do this
else
    do that
```

```
if (rank == 0)
    do this
else
    do that
```

```
if (rank == 0)
    do this
else
    do that
```
`void Comm.Send(
    communicator
    Object buf,
    int offset,
    int count,
    Datatype datatype,
    int dest,
    int tag
) from MPJ Spec`
Parallel Sort using MPI Send/Recv

**Rank 0**

8 23 19 67 45 35 1 24 13 30 3 5

**Rank 0**

8 19 23 35 45 67

sort in parallel
O(N log N)

**Rank 1**

1 3 5 13 24 30

send in O(N)

**Rank 0**

8 19 23 35 45 67 1 3 5 13 24 30

send in O(N)

**Rank 0**

1 3 5 8 13 19 23 24 30 35 45 67

merge in O(N)
Message Tags

- Communicating processes may need to send several messages between each other.
- Message tag: differentiate between different messages being sent.
Message matching

Send

- sender communicator
- sender tag
- dest = r

Receive

- receiver communicator
- receiver tag
- source = q
Receiving messages

```c
void Comm.Recv(
    communicator
    Object buf,
    int offset,
    int count,
    Datatype datatype,
    int src,
    int tag
    )
)
```

A receiver can get a message without knowing:
- the amount of data in the message,
- the sender of the message,
- or the tag of the message.
Synchronous Message Passing

Synchronous send (Ssend)
• waits until complete message can be accepted by receiving process before completing the send

Synchronous receive (Recv)
• waits until expected message arrives

Synchronous routines can perform two actions
• transfer data
• synchronize processes
Asynchronous Message Passing

Send does not wait for actions to complete before returning

- requires local storage for messages
  - sometimes explicit (programmer needs to care)
  - sometimes implicit (transparent to the programmer)

In general

- no synchronisation
- allows local progress
Blocking / Nonblocking

Blocking: return after *local actions* are complete, though the message transfer may not have been completed

Non-blocking: return immediately

- sometimes assumes that data storage to be used for transfer is not modified by subsequent statements until transfer complete
- sometimes implementation dependent local buffers are or have to be provided
A Non-Blocking communication example

Blocking Communication

Non-blocking Communication
Synchronous / Asynchronous vs Blocking / Nonblocking

Synchronous / Asynchronous

- about communication between sender and receiver

Blocking / Nonblocking

- about local handling of data to be sent / received
MPI Send and Receive Defaults

Send

• blocking,
• synchrony **implementation dependent**
  ▪ depends on existence of buffering, performance considerations etc

Recv

• blocking

Danger of Deadlocks.
Don’t make any assumptions!

There are a lot of different variations of this in MPI.
Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received
Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Supply receive buffer at same time as send:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>
More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bsend(1)</td>
<td>Bsend(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
Many parallel programs can be written using just these six functions, only two of which are non-trivial:

- **MPI_INIT** – initialize the MPI library (must be the first routine called)
- **MPI_COMM_SIZE** – get the size of a communicator
- **MPI_COMM_RANK** – get the rank of the calling process in the communicator
- **MPI_SEND** – send a message to another process
- **MPI_RECV** – send a message to another process
- **MPI_FINALIZE** – clean up all MPI state (must be the last MPI function called by a process)

For performance, however, you need to use other MPI features.
The irrational number Pi has many digits
- And it’s not clear if they’re randomly distributed!
- But they can be computed

\[ \pi \approx h \sum_{i=0}^{N-1} \frac{4}{1 + (h(i + \frac{1}{2}))^2} \]

```java
for(int i=0; i<numSteps; i++) {
    double x=(i + 0.5) * h;
    sum += 4.0/(1.0 + x*x);
}

double pi=h * sum ;
```
Pi’s parallel version

```cpp
MPI_Init(args);
...
// declare and initialize variables (sum=0 etc.)
int size = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();

for(int i=rank; i<numSteps; i=i+size) {
    double x=(i + 0.5) * h;
    sum += 4.0/(1.0 + x*x);
}

if (rank != 0) {
    double [] sendBuf = new double []{sum};
    // 1-element array containing sum
    MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 10);
} else { // rank == 0
    double [] recvBuf = new double [1] ;
    for (int src=1 ; src<P; src++) {
        MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 10);
        sum += recvBuf[0];
    }
}
double pi = h * sum; // output pi at rank 0 only!
MPI_Finalize();
```
COLLECTIVE COMMUNICATION
Group Communication

Up to here: point-to-point communication

MPI also supports communications among groups of processors
  • not absolutely necessary for programming (but very nice!)
  • but essential for performance
Examples: broadcast, gather, scatter, reduce, barrier, ...
Collective Computation - Reduce

public void Reduce(java.lang.Object sendbuf, int sendoffset, java.lang.Object recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)

root = rank 0
Reduce implementation: a tree-structured global sum

1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.

2. Second phase:
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their new values.

3. Finally:
   (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.
Collective Data Movement - Broadcast

P0
P1
P2
P3

Broadcast

Processes

0 1 2 3 4 5 6 7
Collective Computation - Allreduce

Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```java
public void Allreduce(java.lang.Object sendbuf, int sendoffset, java.lang.Object recvbuf, int recvoffset, int count, Datatype datatype, Op op)
```
Allreduce = Reduce + Broadcast?

A global sum followed by distribution of the result.

Q: What is the number of steps needed?
Allreduce ≠ Reduce + Broadcast

A butterfly-structured global sum.

Q: What is the number of steps needed?
Collective Data Movement – Scatter/Gather

- Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

- Gather collects all of the components of the vector onto destination process, then destination process can process all of the components.
More Collective Data Movement – some more (16 functions total!)

![Diagram showing data movement and operations for P0, P1, P2, P3 with labels A, B, C, D and A0, A1, A2, A3, B0, B1, B2, B3, C0, C1, C2, C3, D0, D1, D2, D3. The operations include Allgather and Alltoall.](image-url)
Matrix-Vector-Multiply

Compute $y = A \cdot x$, e.g., $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$, $y = \begin{bmatrix} A_1 \cdot x \\ A_2 \cdot x \\ A_3 \cdot x \end{bmatrix}$

1. Broadcast $x$
Matrix-Vector-Multiply

Compute \( y = A \cdot x \), e.g. 
\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]
\[
x = \begin{bmatrix}
10 \\
20 \\
30 \\
\end{bmatrix}
\]
\[
y = \begin{bmatrix}
A_1 \cdot x \\
A_2 \cdot x \\
A_3 \cdot x \\
\end{bmatrix}
\]

2. Scatter A

```
P0  1  2  3
    4  5  6
    7  8  9

P1  4  5  6

P2  7  8  9
```

\( A_{0,0} \) \( A_{0,1} \) \( A_{0,2} \) \( A_{0,3} \) \( A_{0,4} \) \( A_{0,5} \) \( A_{0,6} \) \( A_{0,7} \) \( A_{0,8} \) \( A_{0,9} \)
Matrix-Vector-Multiply

Compute $y = A \cdot x$, e.g. $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, $x = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$, $y = \begin{bmatrix} A_1 \cdot x \\ A_2 \cdot x \\ A_3 \cdot x \end{bmatrix}$

3. Compute locally

- $P_0 \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = 140$
- $P_1 \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = 320$
- $P_2 \begin{bmatrix} 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = 500$
Matrix-Vector-Multiply

Compute $y = A \cdot x$, e.g.

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}, \quad x = \begin{bmatrix}
10 \\
20 \\
30
\end{bmatrix}, \quad y = \begin{bmatrix}
A_1 \cdot x \\
A_2 \cdot x \\
A_3 \cdot x
\end{bmatrix}
\]

4. Gather result $y$
Iterations

Assume we want to apply the matrix-vector product iteratively

\[ y_n = A \cdot y_{n-1} \]

Example Application:
Eigenvalue Problem for Probability Matrix, as used in Google's Pagerank algorithm.

Then each process needs the results of other processes after one step.
→ Need for Gather + Broadcast in one go.
Visualizing Program Behavior
MPI conclusion

- The de-facto interface for distributed parallel computing (nearly 100% market share in HPC)
- Elegant and simple interface
  - Definitely simpler than shared memory (no races, limited conflicts, avoid deadlocks with nonblocking communication)
- We only covered the basics here, MPI-3.1 (2015) has 600+ functions
  - More concepts:
    - Derived Datatypes
    - Process Topologies
    - Nonblocking and neighborhood collectives
    - One-sided accesses (getting the fun of shared memory back ...)
    - Profiling interfaces
    - ...