Course 02158

Message Passing Variants

Hans Henrik Løvengreen

DTU Compute

Message Passing — Examples

- The Message Passing Interface (MPI) library
- The *synchronous channel communication* of the Go language
- The *rendezvous notion* in Ada
- The *actor model* of Erlang and Scala
- The *tuple space* model of Linda
- The Protocol Buffer (ProtoBuf) IDL for message encoding
- The ZeroMQ (ZMQ) message passing library
- The MQTT IoT message passing library

Message Passing Interface (MPI) — History

Background $\sim$ 1990

- Many vendors of *"supercomputers"* with similar software
- A group of 80 people (vendors and researchers) formed *MPI Forum*
- Goal: A uniform way of programming *distributed memory systems*
- June 1994: Version 1.0
- API and protocols for interaction (ca. 130 functions)

Status

- *De facto standard* for programming distributed memory systems
- Official bindings for: C, C++, FORTRAN
- Unofficial bindings for: .NET, Java, Python, ...
- OpenMPI is a very common, open source implementation
- Current major versions: 3.1 (2015), 4.1 (2021), 5.0 (2023) ...

MPI — Key Notions

Processes

- An MPI application consists of a set of communicating *processes*
- Processes are (usually) *single-threaded*

— *Single Program Multiple Data (SPMD)*

Communicators

- A *communicator* is communication universe with a set of processes
- Each process within a communicator has a unique *rank* (0...)
- The full set of started processes belong to `MPI_COMM_WORLD`
- Processes within a communicator may communicate using:
  - Point-to-point communication
  - Collective communication

MPI — Point-to-Point Communication

Operations

- In s: `MPI_Send`($bufp_s, count_s, type_s, dest, tag_s, comm_s$)
- In r: `MPI_Recv`($bufp_r, count_r, type_r, source, tag_r, comm_r, statusp$)
- Type codes: `MPI_INT`, `MPI_CHAR`, `MPI_BYTE`, ...
- *source* and tag_r may be *wildcards*: `MPI_ANY_TAG`, `MPI_ANY_SOURCE` ($*$)

Communication

- If operations *match*: $comm_r = comm_s, dest = r,$ $source = s$ **or** $source = *,$ $tag_r = tag_s,$ **or** $tag_r = *,$ $type_s = type_r$
- Then: $count_s$ elements are copied from $bufp_s$ to $bufp_r$.
- If $count_s > count_r$ an *error* occurs.
- From *statusp*, the source s, tag_s, and $count_s$ may be retrieved.

MPI — Point-to-Point Semantics

General

- Communication is *reliable*
- Communication between a given sender and a given receiver is *ordered*
- *No fairness* guarantee — starvation may occur

Standard mode

- MPI may buffer the message (or not).
- Both `MPI_Send` and `MPI_Recv` are *blocking*
- When `MPI_Send` returns, its buffer may be reused

Alternatives

- Other modes: Synchronous, buffered, non-blocking, ...
- Many auxiliary operations, e.g. `MPI_Probe()`

Example: Hello MPI World

```
int main(void) {
   char greeting[MAX_STRING]; /* String storing message */
   int comm_sz; /* Number of processes */
   int my_rank; /* My process rank */

   MPI_Init(NULL, NULL);

   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

     ⋮

   MPI_Finalize();
}
```

Example: Hello MPI World

```
if (my_rank != 0) {
   sprintf(greeting, "Greetings from process %d of %d!",
         my_rank, comm_sz);
   /* Send message to process 0 */
   MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
         MPI_COMM_WORLD);
} else {
   printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
   for (int q = 1; q < comm_sz; q++) {
      /* Receive message from process q */
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
         0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("%s\n", greeting);
   }
}
```

Collective Communication

Basis

- Barrier:

```
        ⋮                ⋮                ⋮                ⋮
MPI_Barrier(c);  MPI_Barrier(c);  MPI_Barrier(c);  MPI_Barrier(c);
        ⋮                ⋮                ⋮                ⋮
```

- To be called by **all** in communicator c

Extensions

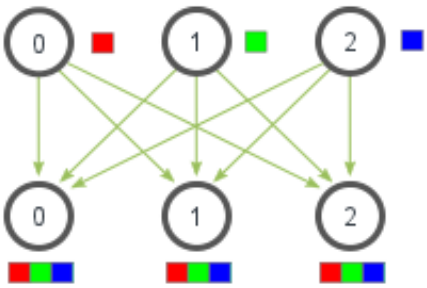
- Broadcast
- Scattering
- Gathering
- Reduction

Collective Communication — Allgather

Operation

- `MPI_Allgather`$(bufp_s, count_s, type_s, bufp_r, count_r, type_r, comm)$
- To be called by **all** in communicator — senders/receivers
- Only $bufp_r$ acts as output (for all)— all others as input
- Must all agree on $type_s = type_r, count_s = count_r, comm$

Effect

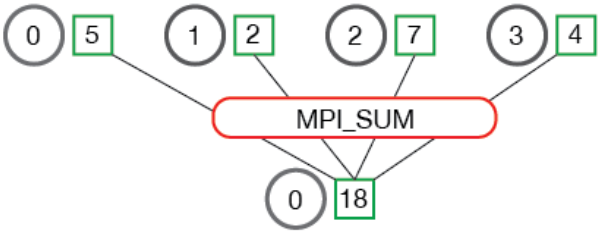


Collective Communication — Gather + processing = Reduce

Operation

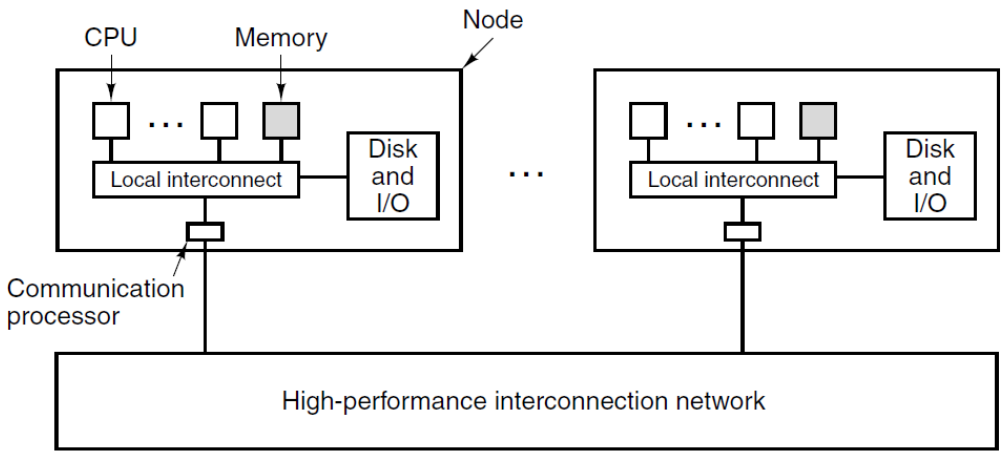
- `MPI_Reduce`($bufp_s$, $bufp_r$, *count*, *type*, *op*, *dest*, *comm*)
- To be called by **all** in communicator — senders and receiver
- Only $bufp_r$ acts as output (for *dest*)— all others as input
- Op codes: `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`, ...
- Must all agree on *type*, *count*, *op*, *dest*, *comm*

Effect



MPI on Hybrid Architectures

- Cluster



- DTU HPC clusters: Eg. 30 nodes with 12+12 cores

The Go Language

- Language developed by Google released in 2012
- Idea: Powerful language based on simple concepts
- Concurrency notions based on CSP
- Large set of *libraries* — especially for networking
- Aimed at handling many simultaneous events and scaling well
- Open source at `golang.org`

Applications

- Used in many Google components
- Used for parts of DropBox, NetFlix, Uber, Twitch, ...
- Basis for the Docker virtual container environment

Go: Sequential language

- Statically typed, compiled imperative language with C-like concrete syntax
- Simple types: `byte`, `bool`, `int`, `int`X, `string`, ...
- Composite types: Arrays, structures, maps
- Go has both *alias types* and *defined types*
- Recently *generic types* have been added
- $x := e; \ldots$ abbreviates **var** $x = e; \ldots$
- Generic loop construct: `for`
- Variables may be allocated on the stack or on the heap (**new**)
- Structure types may have accociated *methods* $\sim$ classes

Go: Concurrency

Goroutines

- Asynchronous execution of function calls: **go** *f*(...)

```
func p(name string) {
    for i := 0; i < 10; i++ {
       fmt.Printf("I am %s\n", name)
    }
}

func main() {
    go p("Huey")
    go p("Dewey")
    go p("Louie")
}
```

- A goroutine is a very lightweight process notion
- Internally, goroutines are scheduled on a thread pool (no need to known)

Go: Communication

Channels

- Channels are typed fixed-size buffered message carriers
- A *channel variable* holds a reference to a *channel (object)* (initially **nil**)
- A *synchronous channel* is created by: **make**(chan *T*)
- A *buffered channel* with capacity *N* is created by: **make**(chan *T*, *N*)
- A channel ch may be *closed* preventing further sends: **close**(ch)
- Channel parameters may be constrained to sending or receiving only

Communication operations

- Send value of expresion e to channel ch: ch<- e (ch!e)
- Receive value from ch in variable x: x = <-ch (ch?x)
- A **nil** channel is never ready to communicate
- A closed channel ch returns *null values*
- Test for closedness: x, ok = <-ch

Go: Communication — example

```
func sender(ch chan<- int) {
    for i := 0; i < 10; i++ { ch<- i }
    close(ch)
}

func main() {
    var votes = make(chan int)

    go sender(votes)

    var sum = 0;
    for {
      var x, ok = <-votes
      if !ok { break }
      fmt.Println("Got ", x); sum += x
    }
    fmt.Println("No more votes. Tally: ", sum)
}
```

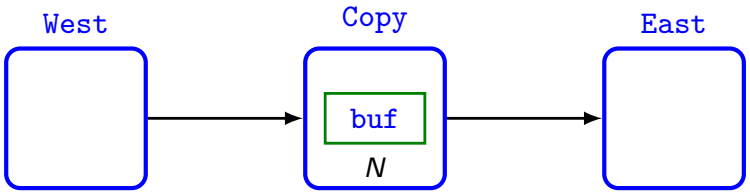
Go Selection

- Choice among posssible communications

```
select {
  case x := <- ch1 :
    SL1
  case y := <- ch2 :
    SL2
  case <- time.After(5 * time.Second) :
    SL3
}
```

- Both *send* and *receive guards* — no boolean guards
- *Randomized choice* among enabled communications
- If none enabled: Blocks unless **default** branch
- *Timeout* may be emulated by a timed channel communication

CSP: N element buffer



- **process** Copy
  var buf[N] : char;
  in, out, count : integer := 0;
  do count < N; West ? buf[in] →
  count := count + 1; in := (in + 1) mod N
  [] count > 0; East ! buf[out] →
  count := count − 1; out := (out + 1) mod N
  od

Go: Buffer Server

- **type** Buf **struct** { depositc, fetchc **chan** T; }
- **func** when(b **bool**, c **chan** T) **chan** T { **if** b {**return** c} **else** {**return nil**} }
- **func** (buf *Buf) init() {

```
    buf.depositc = make(chan T)
    buf.fetchc = make(chan T)

    go func () {
        buffer:= make([]T, N)
        in, out, count := 0, 0, 0
        for {
            select {
                case buffer[in] = <- when(count < N, buf.depositc):
                    in = (in + 1) % N; count++
                case when(count > 0, buf.fetchc) <- buffer[out]:
                    out = (out + 1) % N; count--
    }}}()
}
```

Go: Synchronization

- Go discourages use of shared variables — but does not forbid them
- Access to shared variables may be synchronized via channel communication
- Alternative means for synchronization of goroutines are in the `sync` package

Sync package

- `WaitGroup` to await a group of goroutines to have terminated
- `Once` may be used for iniitialization
- `Mutex` and `Cond`ition
- Thread-safe versions of datastructures

WaitGroup

- `wg.Add(n)` Add atomically *n* to wait count
- `wg.Done()` Decrement wait count by 1
- `wg.Wait()` Block until wait count is 0

Go: Synchronization — WaitGroup example

- ```
  var wg sync.WaitGroup

  func p(name string) {
      defer wg.Done()
      for i := 0; i < 10; i++ {
         fmt.Printf("I am %s\n", name)
      }
  }

  func main() {
      wg.Add(3);

      go p("Huey")
      go p("Dewey")
      go p("Louie")

      wg.Wait()
  }
  ```
  ```

Go: Barrier Monitor

- `type Barrier struct { count int; release *sync.Cond }`

```
func (b *Barrier) init() {
    b.count = 0
    b.release = sync.NewCond(&sync.Mutex{})
}

func (b *Barrier) sync() {
    b.release.L.Lock()
    defer b.release.L.Unlock()
    b.count++
    if b.count==N {
        b.count = 0; b.release.Broadcast(); printBar()
    } else {
        b.release.Wait() // NO Spurious wakeups in Go!!
    }
}
```

The Ada Language

History

- Developed for US DoD around 1980
- Named after Ada Augusta Lovelace (1815–1852) — World's first programmer
- First compilers around 1985 (price 100.000$ — not for universities)
- Has primarily been used within avionics and space

Ada Characteristics

- Imperative language with modules and exceptions (but no classes!)
- Rich syntactic concurrency support (*tasks, rendezvous, protected objects*)
- Real-time support
- Strongly typed, mnay constructs, verbose
- Some run-time systems run without underlying OS

Ada: Bounded Buffer

- ```
  task Buffer is
    entry Append(I: in Integer);
    entry Take(I: out Integer);
  end Buffer;
  ```
  ```

Ada: Bounded Buffer

```
task body Buffer is
  B: Buffer_Array;
  In_ptr, Out_ptr, Count := 0;
begin
  loop
    select
      when Count < Index'Last =>
        accept Append(I: in Integer) do
         B(In_ptr) := I;
        end Append;
      Count := Count + 1; In_ptr := In_ptr + 1;
    or
      when Count > 0 =>
        accept Take(I: out Integer) do
          I := B(Out_ptr);
        end Take;
      Count := Count - 1; Out_ptr := Out_ptr + 1;
    end select
  end loop
end Buffer
```

Ada: Protected Object

```
protected Semaphore is
  entry P;
  procedure V;
private
  S: Integer := 0;
end Semaphore;
```

```
protected body Semaphore is

  entry P when S > 0 is
  begin
    S := S - 1;
  end;

  procedure V is
  begin
    S := S +1;
  end;

end Semaphore;
```

The Erlang Language

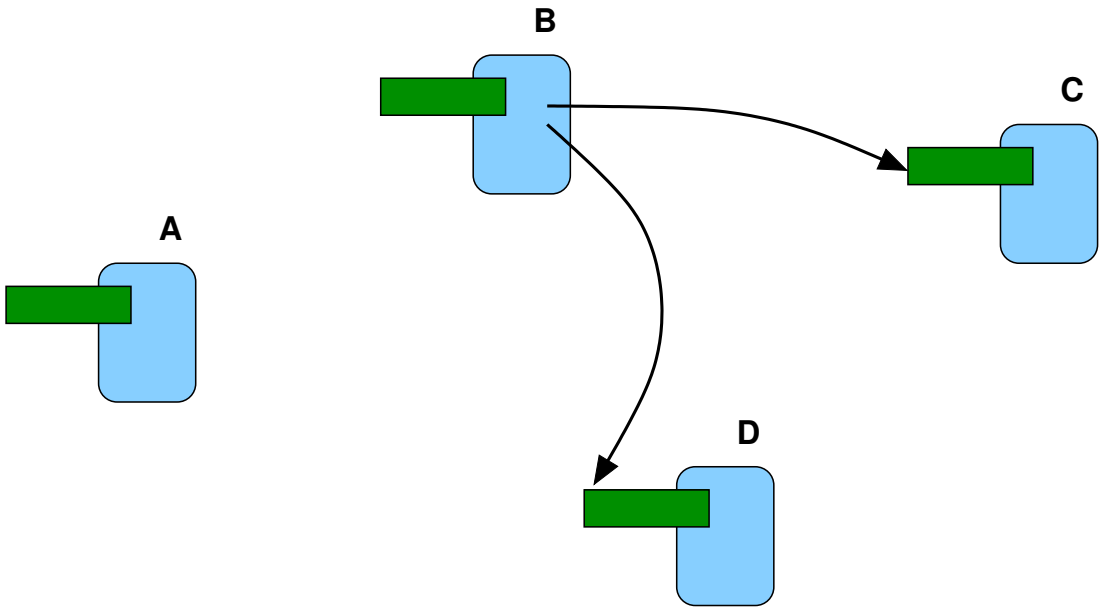
History

- Developed by Ericsson during 1980'ies
- Open Source version of compiler/libraries in 2000
- Used especially for telephone switches

Erlang Characteristics

- Functional language with concurrency
- Asyncronous (buffered) communication
- Distributes readily
- Small contex switch times
- **Not** hard real-time
- Runs on top of (RT-)OS

Actor Model



The Scala Language

- Multi-paradigm langauge developed at EPFL since 2001
- Lead by *Martin Odersky* (Java compiler, Java generics)
- Predecessors: Pizza, Funnel
- Open source, active community
- Famous industrial application: Twitter distribution engine.

Aims

- Expressive, expandable langaguage (contrast to. eg. Java, C#)
- Consise, flexible syntax
- Production quality (static typing, interoperability, JVM, .NET?)
- Simple support for concurrency

Recent adoption

- Basis for the chisel hardware description languge (for FPGA programming)

Scala Example: Resource Allocator

```
class TypedAllocator[T](pool : Seq[T]) extends Actor{
  trait AllocatorReq
    case class Acquire() extends AllocatorReq
    case class Release(r : T) extends AllocatorReq
  val free =new ListBuffer[T]
  def act() = {
    free ++= pool
    loop {
      receive {
        case Acquire() if !free.isEmpty => reply(free.remove(0))
        case Release(r) => free+ = r
      }
    }
  }
}
```

Scala Example: Resource Allocator — usage

- Interface wrappers:

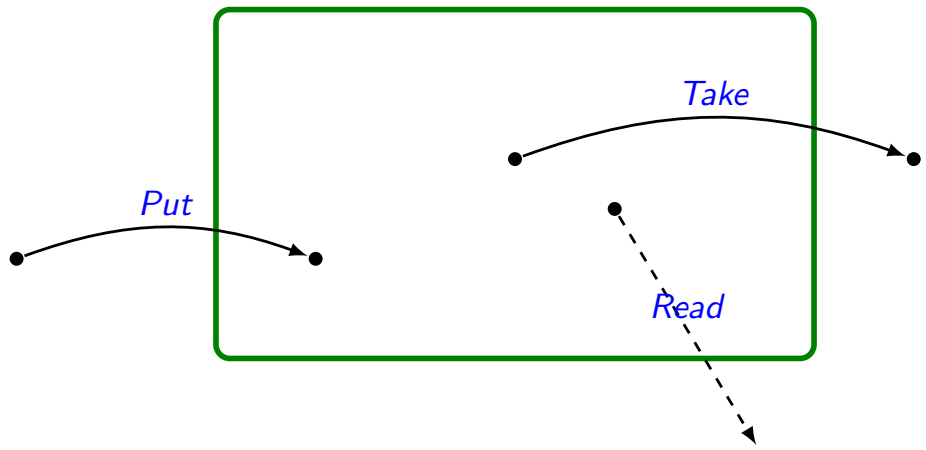
```
def acquire() : T = (this !? Acquire()).asInstanceOf[T]
def release(r : T) = this ! Release(r)
```

-
```
class Client(manager : TypedAllocator[Res]) extends Actor{
  def act() = {
    valres = manager.acquire()
    res.use()
    manager.release(res)
  }
}
```

-
```
val manager = newallocator.TypedAllocator[Res](resseq)
manager.start

for (i <− 1 to 5) {new Client(manager).start}
```

The Tuple Space Model



- Most known tuple space implementation: Linda