The Message Passing Paradigm

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

walker@msr.epm.ornl.gov

WEB: http://www.epm.ornl.gov/~walker/

Presented at
CERN School of Computing, Arles, France
August 22, 1995

Overview

- **The Message Passing Paradigm**
  - Message passing and parallel computing
  - Flavors of message passing
  - Alternative paradigms
- **Features and use of PVM**
  - Heterogeneous parallel computing and PVM
  - Message passing routines
  - Dynamic processes

Overview (cont.)

- **An Introduction to MPI**
  - Main features of MPI
  - Communicators
  - Message passing
  - Implementations and extensions
- **Message Passing in Application Programs**
  - A simple integration example
  - Laplace solver example
  - Particle migration example

Information Resources

- This series of 4 talks is accessible via WWW at:

  `http://www.cs.utk.edu/~dwalker/csc95.html`

- Other useful online resources include:

  `http://www.epm.ornl.gov/~walker/mpi/`
  `http://www.epm.ornl.gov/pvm/`
  `http://www.mcs.anl.gov/fortran-m/FM.html`

- Books:
  - "Using MPI," Gropp, Lusk and Skjellum, MIT Press
  - "PVM," Geist et al., MIT Press
  - "MPI: A Message Passing Standard," Int. J. Supercomp. Appl., 8:3&4, 1994.
  - "Designing and Building Parallel Programs," Foster, Addison-Wesley. `http://www.mcs.anl.gov/dbpp`

What Is Message Passing?

- Message passing is a mechanism for communicating data between processes.
- In point-to-point communication one process sends data and another process receives that data.
- In collective communication there may be more than one sending and/or receiving process.

Message passing between processes is important in many areas,

- parallel computing
- distributed computing
- operating systems

Models Of Computation

- Process models
  - **MIMD** Processes independently execute different instructions on different data.
  - **SPMD** Processes independently execute the same program with different data.
  - **SIMD** Processes execute the same instructions on different data.
- Memory models
  - **Distributed memory**. Each process has a separate local address space.
  - **Shared memory**. Processes share a global address space.
- **Threads**

  A process may contain several execution threads that share the address space of the process.

Computational Hardware

- **Distributed memory**

  Each processing node is directly connected to its own physical local memory. There is no physical shared memory.

- **Shared memory**

  Each processing node is connected to a physical shared memory. Caches are the only form of local memory.

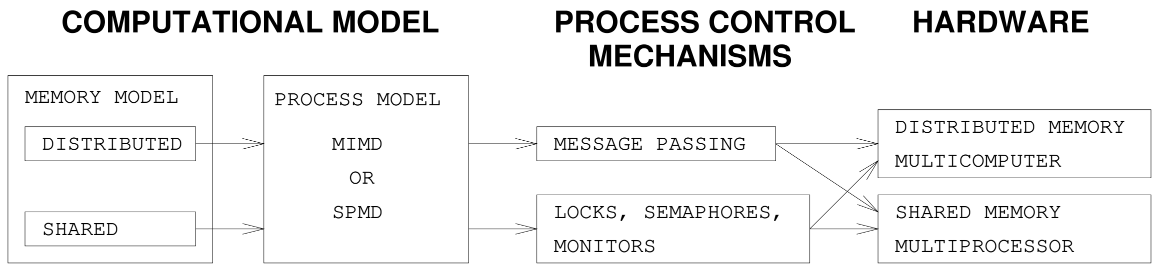
A processing node generally has several componeents, such as;

- One or more computational processors
- A processor for controlling communication
- Vector units
- One or more levels of cache memory
- Local memory

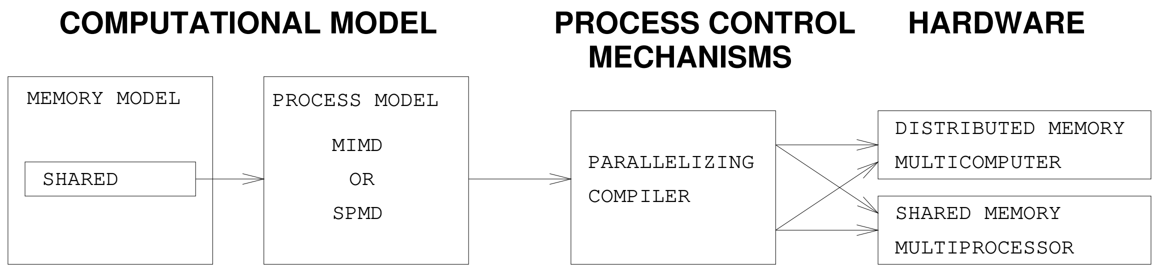
Control Mechanisms

Need mechanisms to control how processes interact.

Past and Current:



Future?



Message Passing Paradigms

- **Coordination Languages**

  Specialized languages specifying concurrency, communication, and synchronization,

  $\Rightarrow$ Strand, PCN, Fortran-M, Occam, Delerium,...

- **Message Passing Libraries**

  Usually provides point-to-point communication of typed messages between processes for blocking and nonblocking modes, plus collective communication, and perhaps process groups.

  – P4, PARMACS, Express, PVM, Zipcode, MPI

- **Shared Memory Support**

  – Split-C provides global pointers, and aims at more effectively utilizing the hardware

  – Linda provides read/write on shared tuple space, emulates shared data structures

Other Paradigms

- **Implicit Parallel Languages**

  Languages employing functional parallelism, such as ParAlfl, Sisal, and Id.

- **Data Parallel Extensions**

  Sequential languages extended with directives specifying data decomposition and distribution. Computation partitioned by "owner computes" rule.

  ⇒ Fortran-D, Vienna Fortran, HPF

- **Shared Memory Support**

  Varying degrees of support of shared memory,

  - Concurrent C++
  - Virtual shared memory
  - Shared Objects, e.g., Orca
  - MetaMP

Introduction to Occam

Main features

- Based on Hoare's CSP
- Explicit message passing via channels
  - Uni-directional
  - Typed
  - Synchronous
- "Assembly language of the transputer"
- Can check for deadlock and non-determinism.

Deadlock

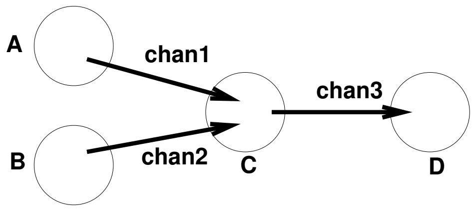
- Synchronous message passing can result in deadlock

```
CHAN OF INT chan1, chan2 :
PAR
  INT x, y :
  SEQ
    chan1 ! x
    chan2 ? y
  INT s, t :
  SEQ
    chan2 ! s
    chan1 ? t
```

Nondeterminism

- Occam introduces nondeterminism through the `ALT` construct

```
SEQ
  WHILE TRUE
    CHAN OF INT chan1, chan2, chan3 :
    INT x, y :
    ALT
      chan1 ? x
        SEQ
          y := 2*x + 1
          chan3 ! y
      chan2 ? x
        SEQ
          y := 2*x
          chan3 ! y
```



Occam Protocols

- Protocols allow channels to carry data of mixed types

```
PROTOCOL Particle REAL; REAL; INT :
CHAN OF Particle chan1 :
REAL pos, vel :
INT type :
PAR
  SEQ
    chan1 ! pos; vel; type
  SEQ
    chan1 ? pos; vel; type
```

- Protocols can also be used for communicating fixed and variable length arrays.
- Variant protocols allow
  - a single channel to carry messages of differing format
  - tagged messages to be communicated

Occam Replicators

- SEQ, PAR, ALT, and IF constructs can all be replicated
- Multiple similar sequential processes:

```
CHAN OF INT chan1 :
INT x :
VAL n IS 100 :
SEQ i = 0 FOR n-1
  chan1 ? x
```

- Replicated PAR creates array of similar processes – used for SPMD style parallelism.
- Replicated ALT can be used to monitor array of channels and act according to which one receives input first.
- Replicated IF can be used to create conditional with similar choices.

Process Placement

- Can affect placement of processes to physical processors using the PLACED PAR construct

  ```
  PLACED PAR i = 0 FOR n-1
    PROCESSOR i
  ```

  places process i on processor i

- Process placement does not affect correctness of program, but may affect performance.

Main Points of Occam

- Primitive input, output, and assignment processes.
- Basic constructs;
  - SEQ for sequential processes,
  - PAR for parallel process,
  - ALT for nondeterminism,
  - IF conditional construct.
  - WHILE loop construct,
- Synchronous communication by typed channels
  - single datatypes,
  - mixed datatypes,
  - fixed and variable length arrays,
  - tagged messages.
- Replicators for SEQ, PAR, ALT, and IF.
- Process placement for efficiency.

Overview of Fortran M

- Fortran M is a set of extensions to Fortran
- Fortran M has language constructs for processes and channels
- Fortran M is broadly similar to Occam
  - Constructs for creating parallel processes
  - Unidirectional typed channels
  - No shared data between processes
  - Support for mapping processes to processors

Differences Between Fortran M and Occam

In Fortran M

- send operation is nonblocking
- there is no need for a SEQ construct
- there is no ALT construct; nondeterminism is introduced through MERGER and PROBE constructs
- a process receives data through inports, and sends data through outputs
  - ports may be communicated ⇒ channels are dynamic
  - multi-dimensional port arrays may be declared ⇒ logical process topologies
- mapping of processes to processors is controlled by specifying a virtual machine

Basic FM Message Passing

- Since SEND is nonblocking deadlock will not result in the following;

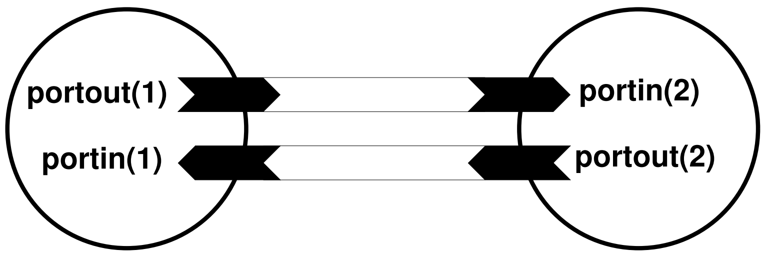
```
program swap1
INPORT  (integer) portin(2)
OUTPORT (integer) portout(2)
CHANNEL (in=portin(1), out=portout(2))
CHANNEL (in=portin(2), out=portout(1))
PROCESSES
  PROCESSCALL swap (portin(1), portout(1))
  PROCESSCALL swap (portin(2), portout(2))
ENDPROCESSES
end

PROCESS swap (in, out)
INPORT  (integer) in
OUTPORT (integer) out
integer p
SEND (out) p
RECEIVE (in) p
end
```

Replicating Processes

```
PROCESSDO i=1,2
  PROCESSCALL swap (portin(i), portout(i), i)
ENDPROCESSDO

PROCESS swap (in, out, p)
INPORT  (integer) in
OUTPORT (integer) out
integer p
SEND (out) p
RECEIVE (in) p
end
```

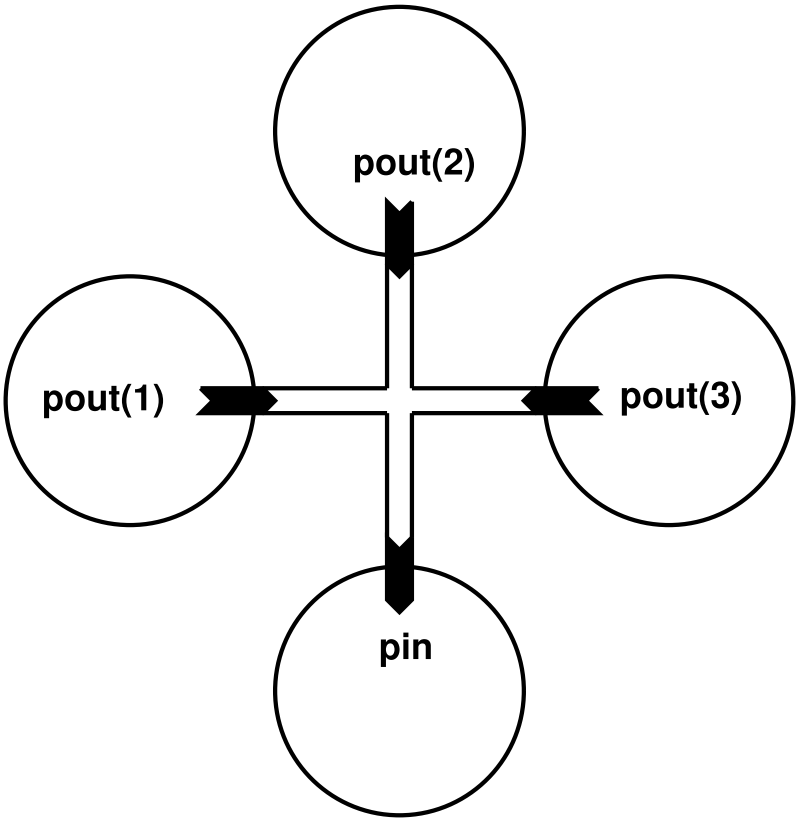


MERGER Construct

- The MERGER construct connects multiple outports to a single inport
- Can be used to introduce nondeterminism into a program
- Is equivalent to a replicated ALT in Occam

```
[3] CHAN OF INT inchan :
CHAN OF INT outchan :
PAR
  ...processes feeding inchan
  WHILE TRUE
    INT x :
    ALT i=0 FOR 2
      inchan[i] ? x
        outchan ! x
  ...process receiving message from outchan
```

Example of MERGER



Example of MERGER

- This code uses a merger to connect three producer processes to one consumer process

```
INPORT (integer) pin
OUTPORT (integer) pout(3)
MERGER (in=pin, out=pout(:))
PROCESSES
  call consumer (pin)
  PROCESSDO i=1,3
    PROCESSCALL producer (pout(i))
  ENDPROCESSDO
ENDPROCESSES
```

The PROBE Construct

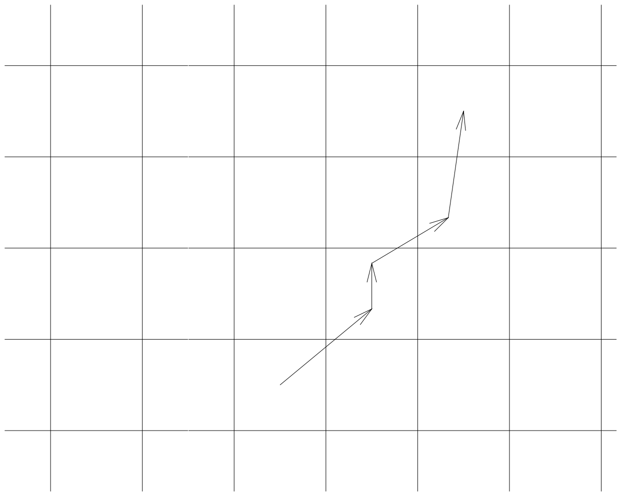
- PROBE can be used to check if a message is pending on an inport.
- It introduces nondeterminism

```
PROCESS handlemsg (inport, x)
INPORT  (integer) inport
integer x
logical flag
do while (flag)
  PROBE (inport, empty=flag)
  if (flag) then
    ...do some more work
  end if
end while
RECEIVE (inport) x
end
```

Dynamic Channels

- A port can be communicated from one process to another in a message
- Useful for "migrating computations" such as semi-Legrangian transport

```
OUTPORT (INPORT (integer)) outport
INPORT (integer) moveport
SEND (outport) moveport
```



Process Mapping

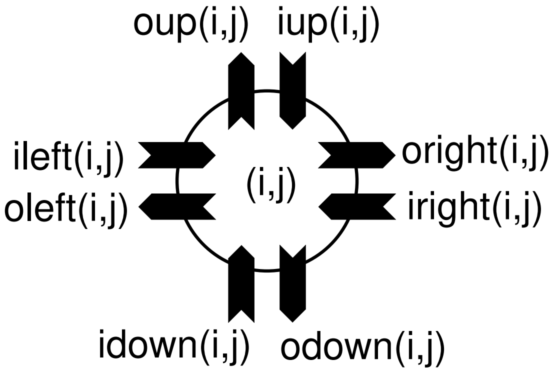
- Process placement affects performance only
- Processes are placed on processors on the basis of a virtual machine specified by the application
- PROCESSORS declaration specifies the size and shape of the virtual processor array
- The LOCATION annotation specifies where in the virtual processor array a process is to run
- The SUBMACHINE annotation is a way of subdividing a virtual processor array

2D Process Mapping

```
 program pde
 parameter (P=3, Q=4)
 INPORT  (real) iup(P,Q),   idown(P,Q)
 INPORT  (real) ileft(P,Q), iright(P,Q)
 OUTPORT (real) oup(P,Q),   odown(P,Q)
 OUTPORT (real) oleft(P,Q), oright(P,Q)
 PROCESSORS (P,Q)
 ...set up channels
 PROCESSDO i=1,P
   PROCESSDO j=1,Q
     PROCESSCALL solver (iup(i,j),   idown(i,j),
#                        ileft(i,j), iright(i,j),
#                        oup(i,j),   odown(i,j),
#                        oleft(i,j), oright(i,j),
#                        i, j)
#    AT LOCATION (i,j)
   ENDPROCESSDO
 ENDPROCESSDO
 end
```

Setting Up 2D Channels

- For a nonperiodic $P \times Q$ grid each process has an outgoing and an incoming channel in each direction, except for processes on the edge of the grid.




```
do i=2,P
  do j=2,Q
    channel (in = idown(i,j),    out = oup(i-1,j))
    channel (in = iup(i-1,j),    out = odown(i,j))
    channel (in = ileft(i,j),    out = oright(i,j-1))
    channel (in = iright(i,j-1), out = oleft(i,j))
  end do
end do
```

F90 Syntax for 2D Channels

```
channel (in = idown(2:P,1:Q),    out = oup(1:P-1,1:Q))
channel (in = iup(1:P-1,1:Q),    out = odown(2:P,1:Q))
channel (in = ileft(1:P,2:Q), out = oright(1:P,1:Q-1))
channel (in = iright(1:P,1:Q-1), out = oleft(1:P,2:Q))
```

Linda

- Linda is an extension to a conventional programming language such as C or Fortran
- Based on an associative shared memory known as “tuple space”
- A tuple is an ordered sequence of data, e.g.,

  ```
  ('array x', 5, [1 4 2 7 9])
  ('pt2pt', dest, source, tag, 'real', n, data(1:n))
  ```

- To create tuples and place in tuple space use `out` and `eval`
- To read tuple, or remove tuple from tuple space, use `rd` and `in`

The out Operation

- The `out` operation
  1. evaluates the tuple fields sequentially on the calling process, and then
  2. places the resulting tuple in tuple space
- For example,

```
out('function', i, f(i))
out('common data', /params/)
out('array section', i, j, x(i:j))
```

- Similar to a SEND operation, but more general.
- Natural support for “self-describing messages”.

The eval **Operation**

- The `eval` operation
  1. evaluates the tuple fields in parallel on separate processes, and then
  2. places the resulting tuple in tuple space
- For example,

```
do i=1,100
  eval('function', i, f(i))
end do
```

- The `eval` operation is nonblocking on the calling process
- Can be regarded as a general way to spawn processes.

Getting Tuples From Tuple Space

- There are two operations for retrieving tuples from tuple space;
  1. the `in` operation finds a tuple that associatively matches a given templates and removes it from tuple space,
  2. the `rd` operation is similar, but does not remove the matched tuple from tuple space
- For example, if a tuple is placed in tuple space with

  ```
  out('function', i, f(i))
  ```

  it can be retrieved with

  ```
  in('function', i, ?result)
  ```

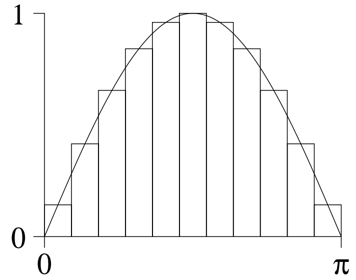
  or with

  ```
  rd('function', i, ?result)
  ```

- If `rd` is used then the tuple remains in tuple space
- If `in` is used then the tuple is removed from tuple space

Integration Example

```
program integrate
parameter (npts=100, pi=3.141592654)
deltax = pi/npts
do i=2,npts
   x = (i-0.5)*deltax
   eval(sin(x))
end do
sum = sin(0.5*deltax)
do i=2,npts
   in(?result)
   sum = sum + result
end do
print *,'The integral is ',sum*deltax
end
```



Linda Summary

- Linda uses only a small set of language extensions
- Linda can simulate conventional message passing libraries, but has more expressivity
- Linda can handle dynamic load balancing quite easily by treating tuple space as a "bag of tasks." Tasks to be performed can be placed in tuple space (with `eval`), and on completion of a task the results are put into tuple space
- Associative shared memory references data by content, not by process and/or name
- There is a loose coupling between processes. The process that consumes data in tuple space does not have to even exist at the time the data is placed into tuple space
- Linda is supported commercially by Scientific Computing Associates, Inc.

Overview of Split-C

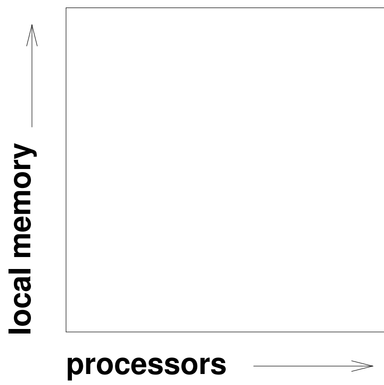
- Split-C is a parallel extension to C
- Split-C is based on an SPMD model
- Split-C features
  - A 2D global address space and global pointers
  - Spread pointers and arrays
  - Split phase assignment
  - Bulk assignment
- Library routines and macros augment the language extensions
- Atomic operations are provided for updating global objects atomically

2D Global Address Space

- Regard global address space as 2D with processors running in one direction and local memory in the other
- Each processors has its own local memory and can access it using standard points
- But any processor can access any object in the global address space using global pointers
- Dereferencing a global pointer may involve communication and be more expensive than dereferencing a local pointer
- Usage:

```
int *global gptr;
int xval, proc, *lptr;
xval = &gptr;
gptr = toglobal(MYPROC,lptr);
lptr = tolocal(gptr);
proc = toproc(gptr);
```

2D Global Address Space



- Arithmetic on global pointer accesses another object on same processor. It is an error to do arithmetic on global pointer that goes outside of local address space
- How to use global pointers to broadcast data (inefficiently)

```
all_bcast_int(int root, int *val, int n)
{
  int i;
  int *global gptr = toglobal(root,val);
  barrier();
  for(i=0;i<n;i++) *(val++) = *(gptr++);
  barrier();
}
```

Spread Pointers

- Spread pointers are global pointers that traverse the 2D global address space in the processors direction
- Successive objects are on different processors

```
/* include files should go here */
splitc_main(){
  int *spread sptr;
  int *lptr, i, sum=0;
  sptr  = all_spread_malloc(PROCS,sizeof(int));
  lptr  = tolocal(sptr);
  *lptr = MYPROC;
  barrier();
  on_one{
    for(i=0;i<PROCS;++i) sum += *(sptr++);
    printf("\nThe sum is %i\n",sum);
  }
  barrier();
  all_spread_free(sptr);
  exit(0);
}
```

Spread Arrays

- A spread array has one or more dimensions spread over the processors, e.g.

  ```
  int x[PROCS]::[10];
  ```

- `x[i][j]` refers to the `j`th element on processor `i`.
- This is equivalent to a block distribution of `10*PROCS` elements with a block size of 10 over `PROCS` processors.

```
static int sarray[PROCS]::;
/* include files should go here */
splitc_main(){
  int i, sum=0;
  sarray[MYPROC] = MYPROC;
  barrier();
  on_one{
    for(i=0;i<PROCS;++i) sum += sarray[i];
    printf("\nThe sum is %i\n",sum);
  }
  barrier();
  exit(0);
}
```

Bulk Assignment

- Bulk assignment routines are used to copy to or from arrays of global objects
- Basic bulk copying routines are
  - `bulk_read` for copying from a global array
  - `bulk_write` for copying to a global array

```
all_bcast_int(int root, int *val, int n)
{
  int i;
  int *global gptr = toglobal(root,val);
  barrier();
  bulk_read(val,gptr,n*sizeof(int));
  barrier();
}
```

- Could also do broadcast by having root write to each processor.

Split Phase Operations

- Split phase assignment initiates access to a global object
- The assignment is guaranteed to have completed after a subsequent call to `sync()`
- This allows the access to be overlapped with other useful work
- The split phase assignment operator is "`:=`"
- The "get" form of the broadcast example:

```
all_bcast_int(int root, int *val, int n)
{
  int i;
  int *global gptr = toglobal(root,val);
  barrier();
  for(i=0;i<n;i++) *(val++) := *(gptr++);
  sync();
  barrier();
}
```

- There is also a routine for bulk split phase assignment

```
  bulk_get(val,gptr,n);
```

Store Assignment

- The store operation stores a value into a global location without acknowledging to the initiating processor when the store has completed

```
all_bcast_int(int root, int *val, int n)
{
  int *global gptr;
  int i, ip;
  barrier();
  on_proc(root){
     for(ip=0;ip<PROCS;ip++) {
        gptr = toglobal(ip,val);
        for (i=0;i<n;i++) *(gptr++) :- *(val++);
  }  }
  all_store_sync();
}
```

- `all_store_sync` synchronizes the processors and waits until all stores have completed.
- There is also a bulk store routine

```
        bulk_store(gptr,val,n);
```

Parallel Virtual Machine

- PVM enables a collection of different computing systems to be viewed as a single parallel virtual machine
- MIMD process model
- Distributed memory model
- PVM transparently handles message passing, data conversion, and task scheduling
- Tasks (or processes) interact by message passing
- Message passing
  - is done by calls to a message passing library
  - uses system controlled buffers
  - is typed
- Point-to-point and collective message passing operations are provided
- PVM has support for process groups

PVM Components

The main components of PVM are:

- **The PVM daemon**. This runs on every host. It controls the PVM resources on each host and mediates interactions with other hosts.
- **The PVM console**. This is a user interface that allows the user to interactively configure the parallel virtual machine.
- **The PVM group server**. This manages process groups within PVM
- **The PVM library**. A set of routines for managing tasks and process groups, and for performing message passing.

The Parallel Virtual Machine

- PVM runs on a large number of machines ranging from networks of workstations up to massively parallel processors.
- The PVM hosts can be specified by
  - adding hosts via the PVM console
  - a host file command line argument to the PVM console

PVM Processes

- Usually a PVM program starts as a single process that spawns other processes to do useful work

  ```
  call pvmfspawn ('worker', PvmTaskDefault, '', 4, tids, numt)
  ```

  will start four processes, each running the excutable **`worker`**.

- This is called the "host-worker" model
- Each PVM process has an integer task ID that uniquely identifies it.
- Spawning process knows all task IDs (in array **`tids`**).
- Process can find its own task ID thus:

  ```
  call pvmfmytid (mytid)
  ```

- Task IDs are used to identify processes in all communication.

PVM Message Passing

- Task IDs are used to specify the source and destination processes.
- In general, messages are buffered by the PVM system
- On receipt messages are selected by source and message tag. These may be wildcarded
- Data to be sent are packed into the send buffer on the source process, and are unpacked from the receive buffer on the destination process
- Datatypes of message components are specified when packing and unpacking messages permitting representation conversion.

Message Exchange

In this example 2 processes exchange an integer and two real numbers.

```
call pvmfmytid (tid)
call pvmfinitsend (PvmDataDefault, bufid)
call pvmfpack (integer4, i, 1, 1, info)
call pvmfpack (real4, vec, 2, 1, info)
if (tid .eq. task1) then
   call pvmfsend (task2, tag, info)
   call pvmfrecv (task2, tag, bufid)
elseif (tid .eq. task2) then
   call pvmfsend (task1, tag, info)
   call pvmfrecv (task1, tag, bufid)
end if
call pvmfunpack (integer4, i, 1, 1, info)
call pvmfunpack (real4, vec, 2, 1, info)
```

Homogeneous Messages

- PVM provides routines for communicating data of a single data type.

```
call pvmfmytid (tid)
if (tid .eq. task1) then
   call pvmfpsend (task2, tag, vec, 10, REAL4, info)
   call pvmfprecv (task2, tag, vec, 10, REAL4,
                   atid, atag, alen, info)
elseif (tid .eq. task2) then
   call pvmfpsend (task1, tag, vec, 10, REAL4, info)
   call pvmfprecv (task1, tag, vec, 10, REAL4,
                   atid, atag, alen, info)
end if
```

- Avoids need for pack and unpack in simple case.

PVM Process Groups

- Can form processes into groups
- Groups are used in collective communication calls to specify the processes involved
- Processes can join or leave a group asynchronously
- Group-oriented operations are mediated by the PVM group server
- Within group processes are labeled by an instance number, or rank, in the range 0 to n-1, where n is the size of the group.
- To join/leave a group

```
call pvmfjoingroup ('group1', inum)
call pvmflvgroup   ('group1', info)
```

- Group information routines

```
call pvmfgsize   ('group1', size)
call pvmfgettid  ('group1', inum, tid)
call pvmfgetinst ('group1', tid, inum)
```

PVM Collective Communication

PVM provides three collective operations that apply to groups of processes

- **Barrier**

  Synchronizes processes. No process exits the routine until all have entered it.

- **Broadcast**

  Tagged broadcast of current active send buffer to all members of group

- **Reduce**

  Reduces data over process group using function $\oplus$

  $$\mathcal{D} = D_0 \oplus D_1 \oplus \ldots D_{n-1}$$

  Here D_0 is data on process i of group, and n is size of group. Result is stored on specified root process.

- For reduce, the function is supplied by the user.
- Can use pre-defined functions for common operations, e.g., summation

What Is MPI?

A proposed standard message passing interface for,

- **explicit message passing** in
- **application programs** for
- **message passing environments**
- Designed to support safe, modular, software libraries
- Designed by the MPI Forum, an international committee of researchers from industry, academia, and government labs

What Is In MPI?

- Point-to-point message passing
- Collective communication
- Support for process groups
- Support for modular communication
- Support for application topologies
- Support for derived datatypes
- A caching mechanism
- Environmental inquiry routines
- Profiling interface

Process Model and Groups

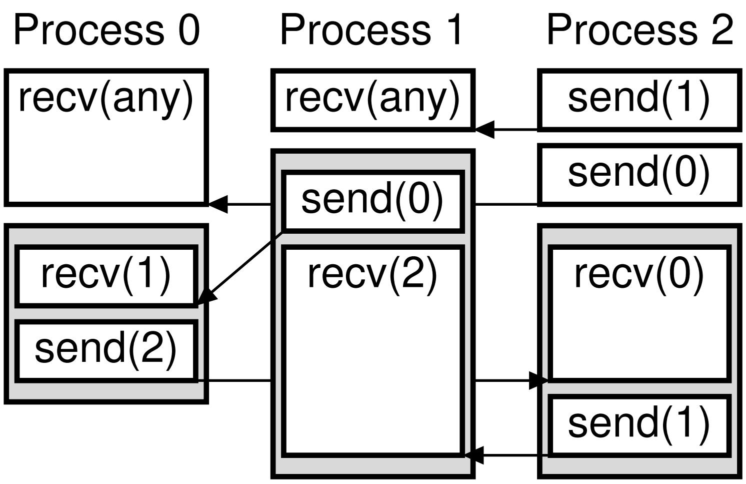
- Fundamental computational unit is the process. Each process has:
  - an independent thread of control,
  - a separate address space
- MPI processes execute in MIMD style, but:
  - No mechanism for loading code onto processors, or assigning processes to processors
  - No mechanism for creating or destroying processes
- MPI supports dynamic process groups.
  - Process groups can be created and destroyed
  - Membership is static
  - Groups may overlap
- No explicit support for multithreading, but MPI is designed to be thread-safe.

Communicators

- Communicators are used to create independent "message universes".
- All message passing routines take a communicator argument as input
- All processes involved in a message passing operation must use the matching communicator argument
- Creation of communicators
  - When an application creates a new communicator it is guaranteed to be unique
- Attributes of communicators
  - Group associated with communicator. Process ranks (source, destination, root) are interpreted relative to this group.
  - Topology of group associated with communicator.
  - User-defined attributes associated via caching mechanism

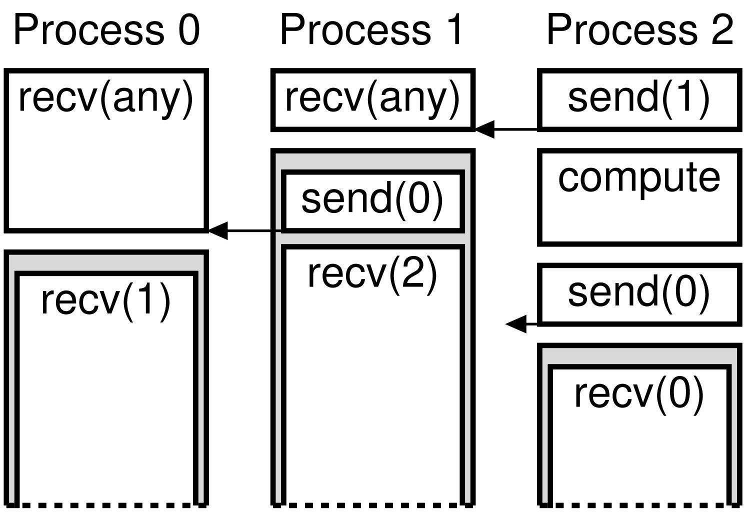
Communication and Nondeterminism

- Communicators are used to disambiguate message selection when an application calls a library routine that performs message passing. Nondeterministic behavior may arise if processes enter the library routine
  - asynchronously,
  - synchronously, but there are outstanding communication operations.



Possible Behavior

- The following shows an incorrect sequence of communication operations



- Deadlock results in this case.
- Need to use communicators to differentiate between messages sent in library routine and rest of application.

Point-to-point Communication

- MPI provides for point-to-point communication between pairs of processes.
- Message selectivity is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Rank and tag may be wildcarded.
- The communicator may not be wildcarded.
- Blocking and nonblocking send and receive routines
- Different communication modes

Blocking Send

- Consider the standard blocking send routine:

```
mpi_send (
          IN  start_of_buffer,
          IN  number_of_items,
          IN  datatype,
          IN  destination_rank,
          IN  tag,
          IN  communicator,
          OUT error_code)
```

Blocking Receive

- Consider the standard blocking receive routine:

```
mpi_recv (
            OUT start_of_buffer,
            IN  max_number_of_items,
            IN  datatype,
            IN  source_rank,
            IN  tag,
            IN  communicator,
            OUT return_status,
            OUT error_code)
```

- In a receive routine, `source_rank` and `tag` can have the values `mpi_any_source` and `mpi_any_tag`.

Collective Communication

- Involves coordinated communication within a group of processes.
- The scope is specified by the communicator.
- No message tags used
- All collective routines block until they are locally complete
- Two broad classes
  - data movement routines
  - global computation routines

Broadcast

- Consider the broadcast routine:

```
mpi_bcast (
           INOUT start_of_buffer,
           IN    number_of_items,
           IN    datatype,
           IN    root,
           IN    communicator,
           OUT   error_code)
```

Process Topologies

- Can arrange processes with a virtual topology
- Two types of topology
  - Cartesian topology
  - General graph topology
- Can shift data in specified direction of Cartesian topology
- Can perform collective communication within a specified hyperplane of Cartesian topology

Environmental Management

- Routines for
  - initializing MPI
  - finalizing MPI
  - aborting MPI
- MPI elapsed wall-clock timers
- Pre-defined or used-defined error handlers may be associated with communicators
- Routine to get processor name
- Environmental inquiry;
  - upper bound for tag values
  - host process rank, if any
  - ranks of processes capable of doing I/O

Profiling Interface

- MPI profiling interface allows a library for profiling MPI calls to be written
- The profiling interface does not specify
  - what routines are profiled by the library
  - what profiling information is gathered
- Profiling interface uses name-shifted versions of the MPI routines starting with "PMPI_" instead of "MPI_".

```
static double timeinsend;
int MPI_Send ( /* arguments */)
{
   double tstart = MPI_Wtime();
   int    result = PMPI_Send( /* arguments */);
   timeinsend   += MPI_Wtime() - tstart;
   return result;
}
```

Example of Profiling Code

```
static double totalBytes;
static double totalTime;

int MPI_Send(void * buffer, const int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_comm comm)
{
   double tstart = MPI_Wtime();
   int extent;
   int result    = PMPI_Send(buffer,count,datatype,
                             dest,tag,comm);
   MPI_Type_size(datatype,&extent);
   totalBytes += count*extent;
   totalTime  += MPI_Wtime() - tstart;
   return result;
}
```

Summary

- We have looking at 6 different approaches to programming parallel computers
  - Coordination languages: occam and Fortran M
  - Extensions supporting shared memory: Linda and Split-C
  - Message Passing libraries: PVM and MPI
- We have introduced the ideas of deadlock and nondeterminism, and looked at how these are handled.
- Fortran M and occam make parallelism explicit, are modular, and can make guarantees of program correctness
- Split-C aims at good performance through split phase and bulk transfer operations, but also has ease-of-use of a global address space
- Linda is simple and expressive, and is commercially supported
- Message passing libraies are most commonly used. They are flexible and permit the explicit management of the memory hierarchy $\Rightarrow$ better performance
- Which is best?