Overview of PVM and MPI

Jack Dongarra

Computer Science Department
University of Tennessee

and

Mathematical Sciences Section
Oak Ridge National Laboratory

(http://www.netlib.org/utk/people/JackDongarra.html)

Outline

- **Motivation for MPI**
- **The process that produced MPI**
- **What is different about MPI?**
  - **the "usual" send/receive**
  - **the MPI send/receive**
  - **simple collective operations**
- **New in MPI: Not in MPI**
- **Some simple complete examples, in Fortran and C**
- **Communication modes, more on collective operations**
- **Implementation status**
- **MPICH - a free, portable implementation**
- **MPI resources on the Net**
- **MPI-2**

What is SPMD?

- *Single Program, Multiple Data*
- Same program runs everywhere.
- Restriction on the general message-passing model.
- Some vendors only support SPMD parallel programs.
- General message-passing model can be emulated.

Messages

- **Messages are packets of data moving between sub-programs.**
- **The message passing system has to be told the following information:**
  - **Sending processor**
  - **Source location**
  - **Data type**
  - **Data length**
  - **Receiving processor(s)**
  - **Destination location**
  - **Destination size**

Access

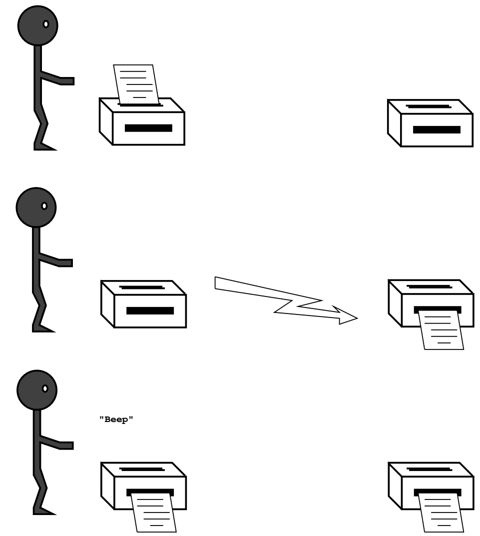
- A sub-program needs to be connected to a message passing system.
- A message passing system is similar to:
  - Mail box
  - Phone line
  - fax machine
  - etc.

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another
- Different types of point-to point communication

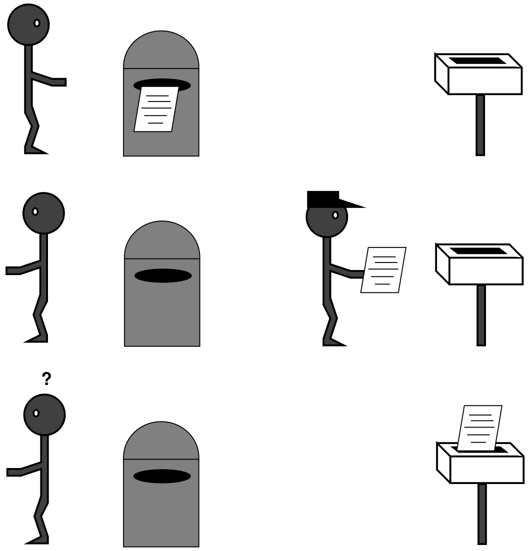
Synchronous Sends

- Provide information about the completion of the message.



Asynchronous Sends

- Only know when the message has left.

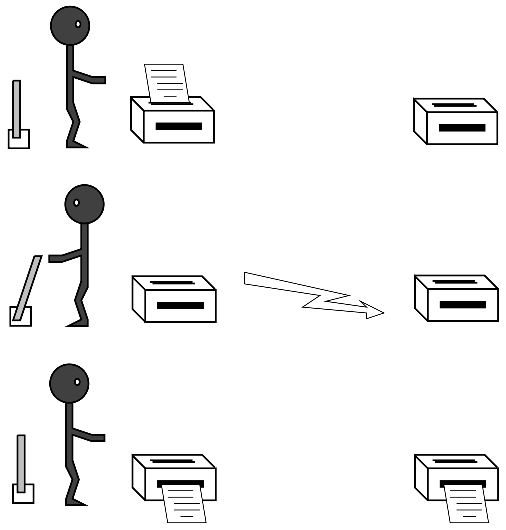


Blocking Operations

- Relate to when the operation has completed.
- Only return from the subroutine call when the operation has completed.

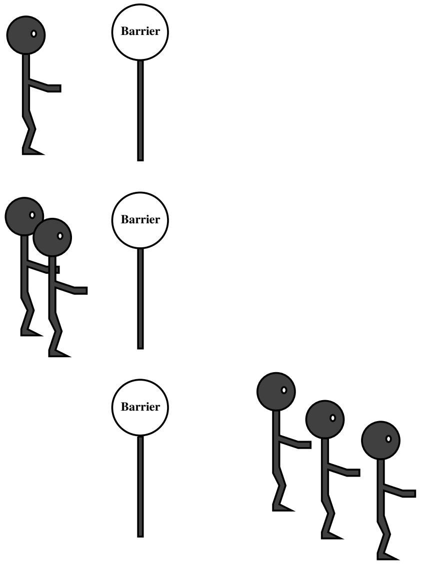
Non–Blocking Operations

- Return straight away and allow the sub–program to continue to perform other work. At some later time the sub–program can TEST or WAIT for the completion of the non–blocking operation.



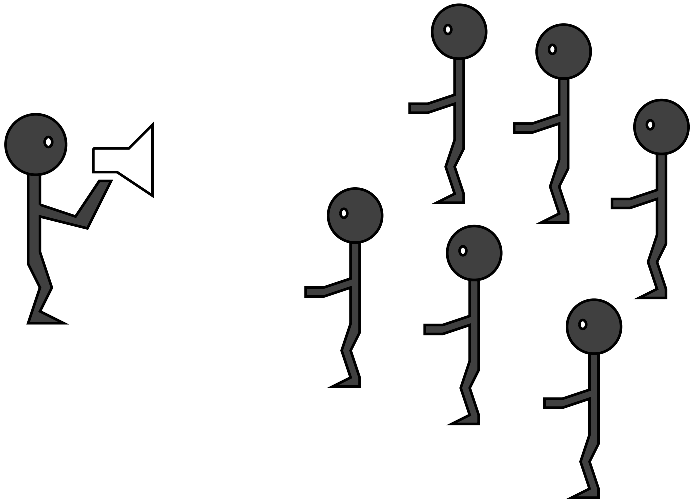
Barriers

- Synchronise processes.



Broadcast

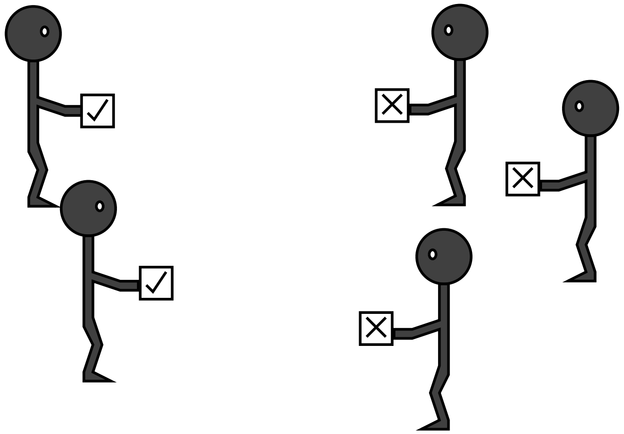
- A one–to–many communication.



Reduction Operations

- **Combine data from several processes to produce a single result.**

STRIKE



Parallelization – Getting Started

- Starting with a large serial application
  - Look at the Physics –
    Is problem inherently parallel?
  - Examine loop structures –
    Are any independent? Moderately so?
    Tools like Forge90 can be helpful
  - Look for the core linear algebra routines –
    Replace with parallelized versions
- Already been done. (check survey)

Popular Distributed Programming Schemes

- Master / Slave

  Master task starts all slave tasks and coordinates their work and I/O
- SPMD (hostless)

  Same program executes on different pieces of the problem
- Functional

  Several programs are written; each performs a different function in the application.

Parallel Programming Considerations

- Granularity of tasks

  Key measure is communication/computation ratio of the machine: Number of bytes sent divided by number of flops performed. Larger granularity gives higher speedups but often lower parallelism.
- Number of messages

  Desirable to keep the number of messages low but depending on the algorithm it can be more efficient to break large messages up and pipeline the data when this increases parallelism.
- Functional vs. Data parallelism

  Which better suits the application? PVM allows either or both to be used.

Network Programming Considerations

- Message latency

  Network latency can be high. Algorithms should be designed to account for this (f.e. send data before it is needed).

- Different Machine Powers

  Virtual machines may be composed of computers whose performance varies over orders of magnitude. Algorithm must be able to handle this.

- Fluctuating machine and network loads

  Multiple users and other competing PVM tasks cause the machine and network loads to change dynamically. Load balancing is important.

Load Balancing Methods

- Static load balancing

  Problem is divided up and tasks are assigned to processors only once. The number or size of tasks may be varied to account for different computational powers of machines.

- Dynamic load balancing by pool of tasks

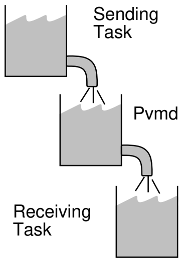
  Typically used with master/slave scheme. The master keeps a queue of tasks and sends them to idle slaves until the queue is empty. Faster machines end up getting more tasks naturally. (see *xep* example in PVM distribution)

- Dynamic load balancing by coordination

  Typically used in SPMD scheme. All the tasks synchronize and redistribute their work either at fixed times or if some condition occurs (f.e. load imbalance exceeds some limit)

Communication Tips

- Limit size, number of outstanding messages
  - Can load imbalance cause too many outstanding messages?
  - May have to send very large data in parts



- Complex communication patterns
  - Network is deadlock-free, shouldn't hang
  - Still have to consider
    - Correct data distribution
    - Bottlenecks
  - Consider using a library
    - ScaLAPACK: LAPACK for distributed-memory machines
    - BLACS: Communication primitives
      - Oriented towards linear algebra
      - Matrix distribution w/ no send-recv
      - Used by ScaLAPACK

Bag of Tasks

- Components
  - Job pool
  - Worker pool
  - Scheduler

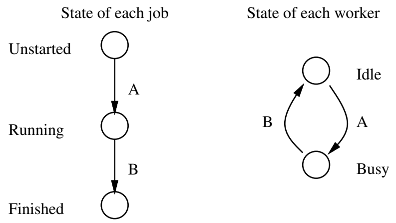


Figure 1: Bag of tasks state machines

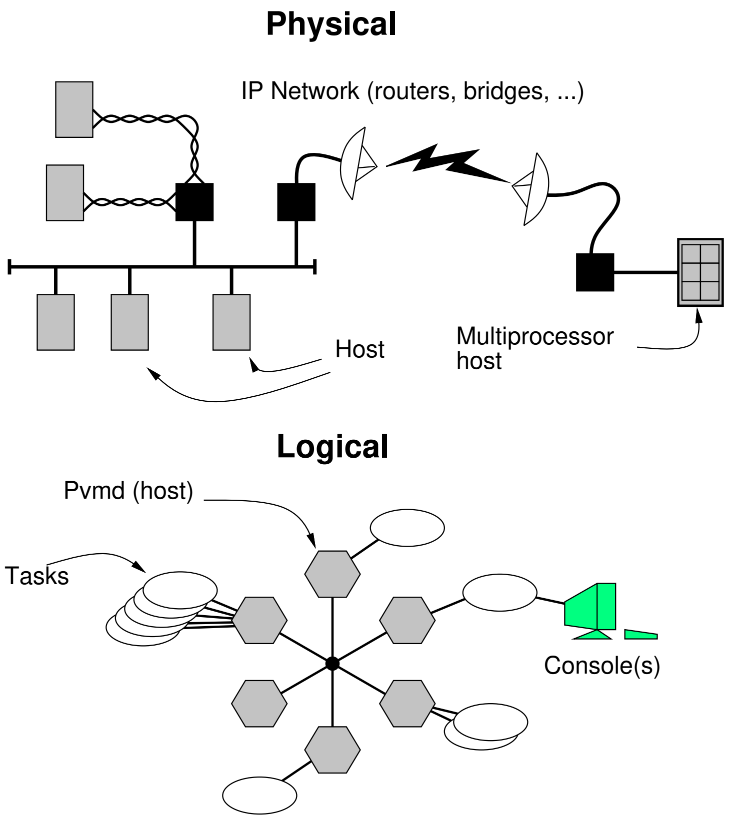
- Possible improvements
  - Adjust size of jobs
    - To speed of workers
    - To turnaround time (granularity)
  - Start bigger jobs before smaller ones
  - Allow workers to communicate
    (more complex scheduling)

PVM Is

PVM is a software package that allows a collection of serial, parallel and vector computers on a network to be managed as one large computing resource.

- Poor man's supercomputer
  - High performance from network of workstations
  - Off-hours crunching
- Metacomputer linking multiple supercomputers
  - Very high performance
  - Computing elements adapted to subproblems
  - Visualization
- Educational tool
  - Simple to install
  - Simple to learn
  - Available
  - Can be modified

Physical and Logical Views of PVM



Parts of the PVM System

- PVM daemon (*pvmd*)
  - One manages each host of virtual machine
  - Mainly a message router, also has kernel-like functions
  - Has message *entry points* where tasks request service
  - Inter-host point of contact
  - Authentication
  - Creates processes
  - Collects output printed by processes
  - Fault detection of processes, network
  - More robust than application components
- Interface library (*libpvm*)
  - Linked with each application component
  - 1. Functions to compose, send, receive messages
  - 2. PVM *syscalls* that send requests to pvmd
  - Machine-dependent communication part can be replaced
  - Kept as simple as possible
- PVM Console
  - Interactive control of virtual machine
  - Kind of like a *shell*
  - Normal PVM task, several can be attached, to any host

Programming in PVM

- A simple message-passing environment
  - Hosts, Tasks, Messages
  - No enforced topology
  - Virtual machine can be composed of any mix of machine types
- Process Control
  - Tasks can be spawned/killed anywhere in the virtual machine
- Communication
  - Any task can communicate with any other
  - Data conversion is handled by PVM
- Dynamic Process Groups
  - Tasks can join/leave one or more groups at any time
- Fault Tolerance
  - Task can request notification of lost/gained resources
- Underlying operating system (usually Unix) is visible
- Supports C, C++ and Fortran
- Can use other languages (must be able to link with C)

Hellos World

- Program hello1.c, the main program:

```
#include <stdio.h>
#include "pvm3.h"

main()
{
    int tid;        /* tid of child */
    char buf[100];

    printf("I'm t%x\n", pvm_mytid());

    pvm_spawn("hello2", (char**)0, 0, "", 1, &tid);
    pvm_recv(-1, -1);
    pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
    pvm_upkstr(buf);
    printf("Message from t%x: %s\n", tid, buf);
    pvm_exit();
    exit(0);
}
```

- Program hello2.c, the slave program:

```
#include "pvm3.h"

main()
{
    int ptid;       /* tid of parent */
    char buf[100];

    ptid = pvm_parent();
    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, 1);
    pvm_exit();
    exit(0);
}
```


Unique Features of PVM

- Software is highly portable
- Allows fully heterogeneous virtual machine (hosts, network)
- Dynamic process, machine configuration
- Support for fault tolerant programs
- System can be customized
- Large existing user base
- Some comparable systems
  - Portable message-passing
    - MPI
    - p4
    - Express
    - PICL
  - One-of-a-kind
    - NX
    - CMMD
  - Other types of communication
    - AM
    - Linda
- Also DOSs, Languages, ...

Portability

- Configurations include

803/486 (BSDI, NetBSD, FreeBSD)	Alliant FX/8
803/486 (Linux)	BBN Butterfly TC2000
DEC Alpha(OSF-1), Mips, uVAX	Convex C2, CSPP
DG Aviion	Cray T-3D, YMP, 2, C90 (Unicos)
HP 68000, PA-Risc	Encore Multimax
IBM RS-6000, RT	Fujitsu 780(UXP/M)
Mips	IBM Power-4
NeXT	Intel Paragon, iPSC/860, iPSC/2
Silicon Graphics	Kendall Square
Sun 3, 4x (SunOS, Solaris)	Maspar
	NEC SX-3
	Sequent
	Stardent Titan
	Thinking Machines CM-2, CM-5

- Very portable across Unix machines, usually just pick options
- Multiprocessors:
  - Distributed-memory: T-3D, iPSC/860, Paragon, CM-5, SP-2/MPI
  - Shared-memory: Convex/HP, SGI, Alpha, Sun, KSR, Symmetry
  - Source code largely shared with generic (80%)
- PVM is portable to non-Unix machines
  - VMS port has been done
  - OS/2 port has been done
  - Windows/NT port in progress
- PVM differences are *almost* transparent to programmer
  - Some options may not be supported
  - Program runs in different environment

How to Get PVM

- PVM home page URL (Oak Ridge) is
`http://www/epm/ornl/gov/pvm/pvm_home.html`
- PVM source code, user's guide, examples and related material are published on Netlib, a software repository with several sites around the world.
  - To get started, send email to netlib:

```
% mail netlib@ornl.gov
Subject: send index from pvm3
```

    A list of files and instructions will be automatically mailed back
  - Using xnetlib: select directory `pvm3`
- FTP: host `netlib2.cs.utk.edu`, login `anonymous`, directory `/pvm3`
- URL: `http://www.netlib.org/pvm3/index.html`
- Bug reports, comments, questions can be mailed to:
`pvm@msr.epm.ornl.gov`
- Usenet newsgroup for discussion and support:
`comp.parallel.pvm`
- Book:
PVM: Parallel Virtual Machine
A Users' Guide and Tutorial for Networked Parallel Computing
MIT press 1994.

Installing PVM

- Package requires a few MB of disk + a few MB / architecture
- Don't need root privelege
- Libraries and executables can be shared between users
- PVM chooses machine *architecture name* for you
  more than 60 currently defined
- Environment variable `PVM_ROOT` points to installed path
  - E.g. `/usr/local/pvm3.3.4` or `$HOME/pvm3`
  - If you use `csh`, add to your `.cshrc`:
    ```
    setenv PVM_ROOT /usr/local/pvm3
    ```
  - If you use `sh` or `ksh`, add to your `.profile`:
    ```
    PVM_ROOT=/usr/local/pvm3
    PVM_DPATH=$PVM_ROOT/lib/pvmd
    export PVM_ROOT PVM_DPATH
    ```
- Important directories below `$PVM_ROOT`

`include`	Header files
`man`	Manual pages
`lib`	Scripts
`lib/`*ARCH*	System executables
`bin/`*ARCH*	System tasks

Building PVM Package

- Software comes with configurations for most Unix machines
- Installation is easy
- After package is extracted
  - `cd $PVM_ROOT`
  - `make`
- Software automatically
  - Determines architecture type
  - Creates necessary subdirectories
  - Builds *pvmd*, *console*, libraries, group server and library
  - Installs executables and libraries in `lib` and `bin`

Starting PVM

- Three ways to start PVM
- `pvm [-d`*debugmask*`] [-n`*hostname*`] [`*hostfile*`]`
  PVM console starts pvmd, or connects to one already running
- `xpvm`
  Graphical console, same as above
- `pvmd [-d`*debugmask*`] [-n`*hostname*`] [`*hostfile*`]`
  Manual start, used mainly for debugging or when necessary to enter passwords
- Some common error messages
  - `Can't start pvmd`
    Check `PVM_ROOT` *is set,* `.rhosts` *correct, no garbage in* `.cshrc`
  - `Can't contact local daemon`
    PVM crashed previously; socket file left over
  - `Version mismatch`
    Mixed versions of PVM installed or stale executables
  - `No such host`
    Can't resolve IP address
  - `Duplicate host`
    Host already in virtual machine or shared `/tmp` *directory*
  - `failed to start group server`
    Group option not built or `ep=` *not correct*
  - `shmget:  ...  No space left on device`
    Stale segments left from crash or not enough are configured

XPVM

- Graphical interface for PVM
  - Performs console-like functions
  - Real-time graphical monitor with
    * View of virtual machine configuration, activity
    * Space-time plot of task status
    * Host utilization plot
    * Call level debugger, showing last libpvm call by each task
- Writes SDDF format trace files
- Can be used for post-mortem analysis
- Built on top of PVM using
  - Group library
  - Libpvm trace system
  - Output collection system

Programming Interface

About 80 functions

Message buffer manipulation	Create, destroy buffers
	Pack, unpack data
Message passing	Send, receive
	Multicast
Process control	Create, destroy tasks
	Query task tables
	Find own tid, parent tid
Dynamic process groups	*With optional group library*
	Join, leave group
	Map group members $\longleftrightarrow$ tids
	Broadcast
	Global reduce
Machine configuration	Add, remove hosts
	Query host status
	Start, halt virtual machine
Miscellaneous	Get, set options
	Request notification
	Register special tasks
	Get host timeofday clock offsets

Process Control

- `pvm_spawn(file, argv, flags, where, ntask, tids)`
  Start new tasks
  - Placement options

`PvmTaskDefault`	Round-robin
`PvmTaskHost`	Named host ("." is local)
`PvmTaskArch`	Named architecture class

  - Other flags

`PvmHostCompl`	Complements host set
`PvmMppFront`	Start on MPP service node
`PvmTaskDebug`	Enable debugging (*dbx*)
`PvmTaskTrace`	Enable tracing

  - Spawn can return *partial* success
- `pvm_mytid()`
  Find my task id / enroll as a task
- `pvm_parent()`
  Find parent's task id
- `pvm_exit()`
  Disconnect from PVM
- `pvm_kill(tid)`
  Terminate another PVM task
- `pvm_pstat(tid)`
  Query status of another PVM task

Basic PVM Communication

- Three-step send method
  - `pvm_initsend(encoding)`
    Initialize send buffer, clearing current one
    Encoding can be `PvmDataDefault` `PvmDataRaw` `PvmDataInPlace`
  - `pvm_pk`*type*`(data, num_items, stride)`
    ...
    Pack buffer with various data
  - `pvm_send(dest, tag)`
    `pvm_mcast(dests, count, tag)`
    Sends buffer to other task(s), returns when safe to clear buffer
- To receive
  - `pvm_recv(source, tag)`
    `pvm_nrecv(source, tag)`
    Blocking or non-blocking receive
  - `pvm_upk`*type*`(data, num_items, stride)`
    Unpack message into user variables
- Can also `pvm_probe(source, tag)` for a message
- Another receive primitive: `pvm_trecv(source, tag, timeout)`
  Equivalent to pvm_nrecv if timeout set to zero
  Equivalent to pvm_recv if timeout set to null

Higher Performance Communication

- Two matched calls for high-speed low-latency messages
  - `pvm_psend(dest, tag, data, num_items, data_type)`
  - `pvm_precv(source, tag, data, num_items, data_type, asource, atag, alength)`
- Pack and send a contiguous, single-typed data buffer
- As fast as native calls on multiprocessor machines

Collective Communication

- Collective functions operate across all members of a *group*
  - `pvm_barrier(group, count)`
    Synchronize all tasks in a group
  - `pvm_bcast(group, tag)`
    Broadcast message to all tasks in a group
  - `pvm_scatter(result, data, num_items, data_type, msgtag, rootinst, group)`
    `pvm_gather(result, data, num_items, data_type, msgtag, rootinst, group)`
    Distribute and collect arrays across task groups
  - `pvm_reduce((*func)(), data, num_items, data_type, msgtag, group, rootinst)`
    Reduce distributed arrays. Predefined functions are
    - `PvmMax`
    - `PvmMin`
    - `PvmSum`
    - `PvmProduct`

Virtual Machine Control

- `pvm_addhosts(hosts, num_hosts, tids)`
  Add hosts to virtual machine
- `pvm_config(nhosts, narch, hosts)`
  Get current VM configuration
- `pvm_delhosts(hosts, num_hosts, results)`
  Remove hosts from virtual machine
- `pvm_halt()`
  Stop all pvmds and tasks (shutdown)
- `pvm_mstat(host)`
  Query status of host
- `pvm_start_pvmd(argc, argv, block)`
  Start new master pvmd

PVM Examples in Distribution

- Examples illustrate usage and serve as templates
- Examples include

hello, hello_other	Hello world
master, slave	Master/slave program
spmd	SPMD program
gexample	Group and collective operations
timing, timing_slave	Tests communication performance
hitc, hitc_slave	Dynamic load balance example
xep, mtile	Interactive X-Window example

- Examples come with Makefile.aimk files
- Both C and Fortrans versions for some examples

Compiling Applications

- Header files
  - C programs should include

`<pvm3.h>`	Always
`<pvmtev.h>`	To manipulate trace masks
`<pvmsdpro.h>`	For resource manager interface

  - Specify include directory: `cc -I$PVM_ROOT/include ...`
  - Fortran: `INCLUDE '/usr/local/pvm3/include/fpvm3.h'`
- Compiling and linking
  - C programs must be linked with

`libpvm3.a`	Always
`libgpvm3.a`	If using group library functions
possibly other libraries (for socket or XDR functions)	

  - Fortran programs must additionally be linked with `libfpvm3.a`

Compiling Applications, Cont'd

- Aimk
  - Shares single makefile between architectures
  - Builds for different architectures in separate directories
  - Determines PVM architecture
  - Runs make, passing it `PVM_ARCH`
  - Does one of three things
    - If `$PVM_ARCH/[Mm]akefile` exists:
      Runs make in subdirectory, using makefile
    - Else if `Makefile.aimk` exists:
      Creates subdirectory, runs make using `Makefile.aimk`
    - Otherwise:
      Runs make in current directory

Load Balancing

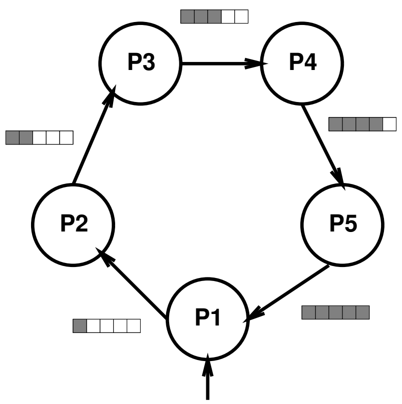
- Important for application performance
- Not done automatically (yet?)
- Static – Assignment of work or placement of tasks
  - Must predict algorithm time
  - May have different processor speeds
  - Externally imposed (static) machine loads
- Dynamic – Adapting to changing conditions
  - Make simple scheduler: E.g. Bag of Tasks
    - Simple, often works well
    - Divide work into small jobs
    - Given to processors as they become idle
    - PVM comes with examples
      - C – xep
      - Fortran – hitc
    - Can include some fault tolerance
  - Work migration: Cancel / forward job
    - Poll for cancel message from master
    - Can interrupt with `pvm_sendsig`
    - Kill worker (expensive)
  - Task migration: Not in PVM yet
- Even with load balancing, expect performance to be variable

Six Examples

- Circular messaging
- Inner product
- Matrix vector multiply (row distribution)
- Matrix vector multiply (column distribution)
- Integration to evaluate π
- Solve 1-D heat equation

Circular Messaging

A vector circulates among the processors
Each processor fills in a part of the vector



Solution:

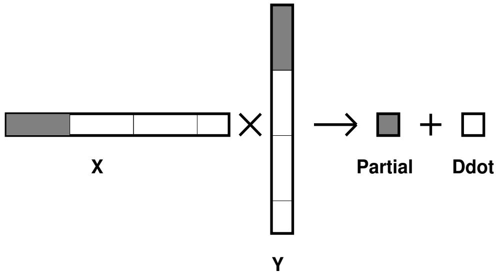
- SPMD
- Uses the following PVM features:
  - spawn
  - group
  - barrier
  - send-recv
  - pack-unpack


```
      program spmd1
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

      PARAMETER( NPROC=4 )
      integer rank, left, right, i, j, ierr
      integer tids(NPROC-1)
      integer data(NPROC)

C     Group Creation

      call pvmfjoingroup( 'foo', rank )
      if( rank .eq. 0 )  then
        call pvmfspawn('spmd1',PVMDEFAULT,'*',NPROC-1,tids(1),ierr)
      endif

      call pvmfbarrier( 'foo', NPROC, ierr )

C     compute the neighbours IDs

      call pvmfgettid( 'foo', MOD(rank+NPROC-1,NPROC), left )
      call pvmfgettid( 'foo', MOD(rank+1,NPROC), right)

      if( rank .eq. 0 ) then

C     I am the first process

         do 10 i=1,NPROC
 10         data(i) = 0
         call pvmfinitsend( PVMDEFAULT, ierr )
         call pvmfpack( INTEGER4, data, NPROC, 1, ierr )
         call pvmfsend( right, 1 , ierr )
         call pvmfrecv( left, 1, ierr)
         call pvmfunpack( INTEGER4, data, NPROC, 1, ierr)
         write(*,*) ' Results received :'
         write(*,*)  (data(j),j=1,NPROC)
      else
C     I am an intermediate process

         call pvmfrecv( left, 1, ierr )
         call pvmfunpack( INTEGER4, data, NPROC, 1, ierr )
         data(rank+1) = rank
         call pvmfinitsend(PVMDEFAULT, ierr)
         call pvmfpack( INTEGER4, data, NPROC, 1, ierr)
         call pvmfsend( right, 1, ierr)
      endif

      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

Inner Product

Problem: In parallel compute

$$s = \sum_{i=1}^{n} x^T y$$



Solution:

- Master - Slave
- Uses the following PVM features:
  - spawn
  - group
  - barrier
  - send-recv
  - pack-unpack
- Master sends out data, collects the partial solutions and computes the sum.
- Slaves receive data, compute partial inner product and send the results to master.

Inner Product - Pseudo code

- **Master**

```
Ddot = 0
for i = 1 to <number of slaves>
  send ith part of X to the ith slave
  send ith part of Y to the ith slave
end for
Ddot = Ddot + Ddot(remaining part of X and Y)
for i = 1 to <number of slaves>
  receive a partial result
  Ddot = Ddot + partial result
end for
```

- **Slave**

```
Receive a part of X
Receive a part of Y
partial = Ddot(part of X and part of Y)
send partial to the master
```

```
      program inner
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

      PARAMETER( NPROC=7 )
      PARAMETER( N = 100)
      double precision ddot
      external ddot
      integer remain, nb
      integer rank, i, ierr, bufid
      integer tids(NPROC-1), slave, master
      double precision x(N),y(N)
      double precision result,partial

      remain = MOD(N,NPROC-1)
      nb = (N-remain)/(NPROC-1)

      call pvmfjoingroup( 'foo', rank )
      if( rank .eq. 0 )  then
        call pvmfspawn('inner',PVMDEFAULT,'*',NPROC-1,tids,ierr)
      endif

      call pvmfbarrier( 'foo', NPROC, ierr )

      call pvmfgettid( 'foo', 0, master )

C                           MASTER
      if( rank .eq. 0 ) then

C                           Set the values
         do 10 i=1,N
            x(i) = 1.0d0
10          y(i) = 1.0d0

C                               Send the data
         count = 1
         do 20 i=1,NPROC-1
           call pvmfinitsend( PVMDEFAULT, ierr )
           call pvmfpack( REAL8, x(count), nb, 1, ierr )
           call pvmfpack( REAL8, y(count), nb, 1, ierr )
           call pvmfgettid( 'foo', i, slave)
           call pvmfsend( slave, 1, ierr )
           count = count + nb
20       continue
```

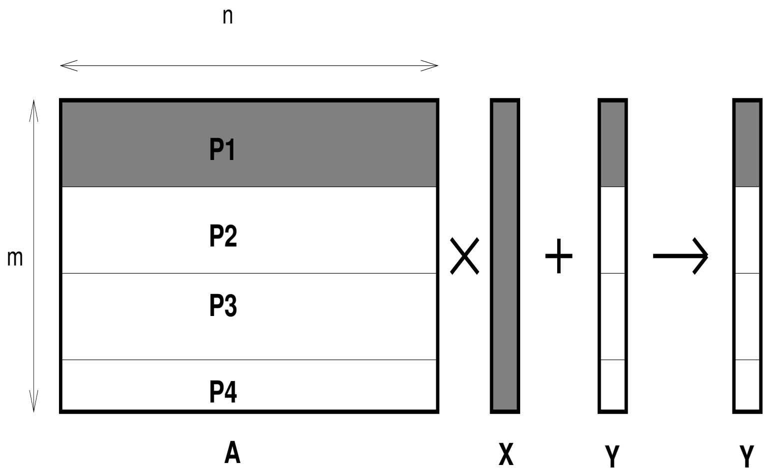


```
         result = 0.d0
C                                Add the remainding part
         partial = ddot(remain,x(N-remain+1),1,y(N-remain+1),1)
         result = result + partial

C                                Get the result
         do 30 i =1,NPROC-1
           call pvmfrecv(-1,1,bufid)
           call pvmfunpack( REAL8, partial, 1, 1, ierr)
           result = result + partial
30       continue
         print *, ' The ddot = ', result

C                                      SLAVE
      else

C                                      Receive the data
         call pvmfrecv( -1, 1, bufid )
         call pvmfunpack( REAL8, x, nb, 1, ierr )
         call pvmfunpack( REAL8, y, nb, 1, ierr )
C                                      Compute the partial product
         partial = ddot(nb,x(1),1,y(1),1)
C                                      Send back the result
         call pvmfinitsend( PVMDEFAULT, ierr)
         call pvmfpack( REAL8, partial, 1, 1, ierr)
         call pvmfsend( master, 1, ierr)

      endif


      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

Matrix - Vector Product (Row Distribution)

Problem: In parallel compute $y = y + Ax$, where y is of length m, x is of length n and A is an $m \times n$ matrix.
Solution:

- Master - Slave
- Uses the following PVM features:
  - spawn
  - group
  - barrier
  - send-recv
  - pack-unpack



Matrix - Vector Product (Row Distribution) Pseudo Code

- **Master**

```
for i = 1 to <number of slaves>
  send X to the ith slave
  send Y to the ith slave
end for
for i = 1 to <number of slaves>
  receive a partial result from a slave
  update the corresponding part of Y
end for
```

- **Slave**

```
Receive X
Receive Y
Compute my part of the product
   and Update my part of Y
Send back my part of Y
```

```
      program matvec_row
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

C
C       y <--- y + A * x
C
C       A : MxN (visible only on the slaves)
C       X : N
C       Y : M
C
      PARAMETER( NPROC = 4)
      PARAMETER( M = 9, N = 6)
      PARAMETER( NBY = INT(M/(NPROC)+1))

      double precision X(N), Y(M)

      integer tids(NPROC)
      integer mytid, rank, i, ierr, from


      call pvmfmytid( mytid )
      call pvmfjoingroup( 'foo', rank )
      if( rank .eq. 0 )  then
        call pvmfspawn('matvecslv_row',PVMDEFAULT,'*',NPROC,tids,ierr)
      endif
      call pvmfbarrier( 'foo', NPROC+1, ierr )

C                   Data initialize for my part of the data

      do 10 i = 1,N
        x(i) = 1.d0
10    continue
      do 15 i = 1,M
        y(i) = 1.d0
15    continue

C     Send X and Y to the slaves

      call pvmfinitsend( PVMDEFAULT, ierr )
      call pvmfpack(REAL8, X, N, 1, ierr)
      call pvmfpack(REAL8, Y, M, 1, ierr)
      call pvmfbcast('foo', 1, ierr)
```



```
C                     I get the results

      do 20 i = 1, NPROC
        call pvmfrecv(-1, 1, ierr)
        call pvmfunpack( INTEGER4, from, 1, ierr)
        if (from .EQ. NPROC) then
          call pvmfunpack( REAL8, Y((from-1)*NBY+1),
     $                    M-NBY*(NPROC-1), 1, ierr)
        else
          call pvmfunpack( REAL8, Y((from-1)*NBY+1),
     $                    NBY, 1, ierr)
        endif
20    continue

      write(*,*) 'Results received'
      do 30 i=1,M
        write(*,*) 'Y(',i,') = ',Y(i)
30    continue
      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

```
      program matvecslv_row
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

C
C       y <--- y + A * x
C       A : MxN (visible only on the slaves)
C       X : N
C       Y : M
C
      PARAMETER( NPROC = 4)
      PARAMETER( M = 9, N = 6)
      PARAMETER( NBY = INT(M/NPROC)+1)

      double precision A(NBY,N)
      double precision X(N), Y(M)

      integer rank, i, ierr, to

      external dgemv

      call pvmfjoingroup( 'foo', rank )
      call pvmfbarrier( 'foo', NPROC+1, ierr )

C       Data initialize for my part of the data

      do 10 j = 1,N
        do 20 i = 1,NBY
          A(i,j) = 1.d0
20      continue
10    continue

C     I receive X and Y

      call pvmfrecv( -1, 1,  ierr )
      call pvmfunpack(REAL8, X, N, 1, ierr)
      call pvmfunpack(REAL8, Y, M, 1, ierr)

C     I compute my part

      if (rank .NE. NPROC) then
        call dgemv('N', NBY, N, 1.d0, A, NBY,
     $             X, 1, 1.d0, Y((rank-1)*NBY+1),1)
      else
        call dgemv('N', M-NBY*(NPROC-1), N, 1.d0, A, NBY,
     $             X, 1, 1.d0, Y((NPROC-1)*NBY+1), 1)
      endif
```

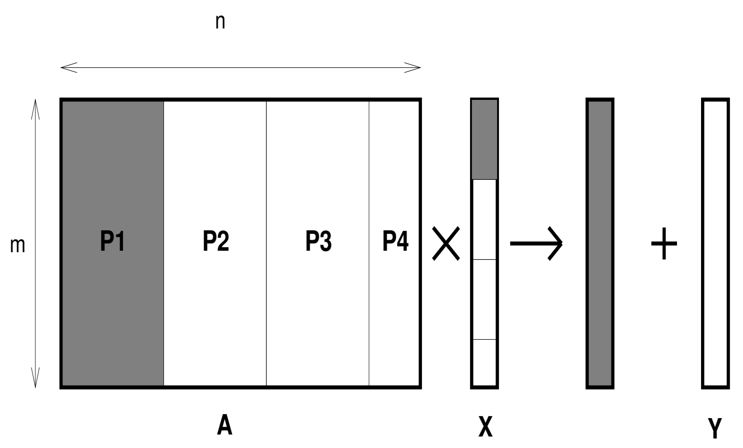


```
C     I send back my part of Y

      call pvmfinitsend(PVMDEFAULT, ierr)
      call pvmfpack( INTEGER4, rank, 1, 1, ierr)
      if (rank .NE. NPROC) then
        call pvmfpack( REAL8, Y((rank-1)*NBY+1),NBY, 1, ierr)
      else
        call pvmfpack( REAL8, Y((rank-1)*NBY+1),M-NBY*(NPROC-1),1,ierr)
      endif
      call pvmfgettid('foo',0,to)
      call pvmfsend(to, 1, ierr)

C     done

      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

Matrix - Vector Product (Column Distribution)

Problem: In parallel compute $y = y + Ax$, where y is of length m, x is of length n and A is an $m \times n$ matrix.
Solution:

- Master - Slave
- Uses the following PVM features:
  - spawn
  - group
  - barrier
  - reduce
  - send-recv
  - pack-unpack



Matrix - Vector Product (Column Distribution) Pseudo Code

- **Master**

```
for i = 1 to <number of slaves>
  send X to the ith slave
end for
Global Sum on Y (root)
```

- **Slave**

```
Receive X
Compute my Contribution to Y
Global Sum on Y (leaf)
```

```
      program matvec_col
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

C
C       y <--- y + A * x
C
C       A : MxN (visible only on the slaves)
C       X : N
C       Y : M
C
      PARAMETER( NPROC = 4)
      PARAMETER( M = 9, N = 6)

      double precision X(N), Y(M)

      external PVMSUM

      integer tids(NPROC)
      integer mytid, rank, i, ierr


      call pvmfmytid( mytid )
      call pvmfjoingroup( 'foo', rank )
      if( rank .eq. 0 )  then
        call pvmfspawn('matvecslv_col',PVMDEFAULT,'*',NPROC,tids,ierr)
      endif
      call pvmfbarrier( 'foo', NPROC+1, ierr )

C                 Data initialize for my part of the data

      do 10 i = 1,N
        x(i) = 1.d0
10    continue
      do 15 i = 1,M
        y(i) = 1.d0
15    continue
```



```
C     Send X

      call pvmfinitsend( PVMDEFAULT, ierr )
      call pvmfpack(REAL8, X, N, 1, ierr)
      call pvmfbcast('foo', 1, ierr)

C                 I get the results

      call pvmfreduce(PVMSUM,Y,M,REAL8,1,'foo',0,ierr)

      write(*,*) 'Results received'
      do 30 i=1,M
        write(*,*) 'Y(',i,') = ',Y(i)
30    continue
      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

```
      program matvecslv_col
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

C
C       y <--- y + A * x
C       A : MxN (visible only on the slaves)
C       X : N
C       Y : M
C
      PARAMETER( NPROC = 4)
      PARAMETER( M = 9, N = 6)
      PARAMETER( NBX = INT(N/NPROC)+1)

      external PVMSUM

      double precision A(M,NBX)
      double precision X(N), Y(M)

      integer rank, i, ierr

      external dgemv

      call pvmfjoingroup( 'foo', rank )
      call pvmfbarrier( 'foo', NPROC+1, ierr )

C       Data initialize for my part of the data

      do 10 j = 1,NBX
        do 20 i = 1,M
          A(i,j) = 1.d0
20      continue
10    continue

C     I receive X

      call pvmfrecv( -1, 1,  ierr )
      call pvmfunpack(REAL8, X, N, 1, ierr)
```



```
C     I compute my part

      if (rank .NE. NPROC) then
        call dgemv('N', M, NBX, 1.d0, A, M,
     $             X((rank-1)*NBX+1), 1, 1.d0, Y,1)
      else
        call dgemv('N', M,N-NBX*(NPROC-1), 1.d0, A, M,
     $             X((NPROC-1)*NBX+1), 1, 1.d0, Y, 1)
      endif

C     I send back my part of Y

      call pvmfreduce(PVMSUM,Y,M,REAL8,1,'foo',0,ierr)

C     done

      call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

Integration to evaluate π

Computer approximations to π by using numerical integration
Know

$$tan(45^0) = 1;$$

same as

$$tan\frac{\pi}{4} = 1;$$

So that;

$$4 * tan^{-1}1 = \pi$$

From the integral tables we can find

$$tan^{-1}x = \int \frac{1}{1+x^2}dx$$

or

$$tan^{-1}1 = \int_0^1 \frac{1}{1+x^2}dx$$

Using the mid-point rule with panels of uniform length $h = 1/n$, for various values of n.
Evaluate the function at the midpoints of each subinterval (x_{i-1}, x_i).
$i * h - h/2$ is the midpoint.
Formula for the integral is

$$x = \sum_{i=1}^{n} f(h * (i - 1/2))$$

$$\pi = h * x$$

where

$$f(x) = \frac{4}{1+x^2}$$

Integration to evaluate π (continued)

Number of ways to divide up the problem.
Each part of the sum is independent.

- Divide the interval into more or less equal parts and give each process a part.
- Let each processor take the p^{th} part.
- Compute part of integral.
- Sum pieces.

The example given let's each processor take the p^{th} part. Uses the following PVM features:

- spawn
- group
- barrier
- bcast
- reduce

```
      program spmd2
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

      PARAMETER( NPROC=3 )
      EXTERNAL PVMSUM

      integer mytid, rank, i, ierr
      integer tids(0:NPROC-1)

      double precision PI25DT
      parameter (PI25DT = 3.141592653589793238462643d0)
      double precision mypi,pi,h,sum,x,f,a

C                                 function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

      call pvmfmytid( mytid )

      call pvmfjoingroup( 'foo', rank )
10    if( rank .eq. 0 )  then
        call pvmfspawn('spmd2',PVMDEFAULT,'*',NPROC-1,tids(1),ierr)
      endif

      call pvmfbarrier( 'foo', NPROC, ierr )

      if( rank .eq. 0 ) then

         write(6,98)
98       format('Enter the number of intervals: (0 quits)')
         read(5,99)n
99       format(i10)
         if (n .GT. 100000) then
            print *, 'Too large value of pi'
            print *, 'Using 100000 instead'
            n = 100000
         endif

         call pvmfinitsend( PVMDEFAULT, ierr)
         call pvmfpack( INTEGER4, n, 1, 1, ierr)
         call pvmfbcast('foo',0,ierr)
      endif

      if(rank .ne. 0) then
         call pvmfrecv(-1,-1,ierr)
         call pvmfunpack(INTEGER4, n, 1, 1, ierr)
      endif
```



```
C                             check for quit signal
      if (n .le. 0) goto 30
C                             calculate the interval size
      h = 1.0d0/n

      sum = 0.0d0
      do 20 i = rank+1, n, NPROC
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
20    continue
      mypi = h * sum
C                             collect all the partial sums
      print *,' reduce'
      call pvmfreduce(PVMSUM,mypi,1,REAL8,0,'foo',0,ierr)
C                             node 0 prints the number
      if (rank .eq. 0) then
        pi = mypi
        write(6,97) pi, abs(pi - PI25DT)
97      format('  pi is approximatively: ',F18.16,
     +          '  Error is: ',F18.16)
        goto 10

      endif


30    call pvmflvgroup( 'foo', ierr )
      call pvmfexit(ierr)
      stop
      end
```

1-D Heat Equation

Problem: Calculating heat diffusion through a wire.

The one-dimensional heat equation on a thin wire is :

$$\frac{\partial A}{\partial t} = \frac{\partial^2 A}{\partial x^2}$$

and a discretization of the form :

$$\frac{A_{i+1,j} - A_{i,j}}{\Delta t} = \frac{A_{i,j+1} - 2A_{i,j} + A_{i,j-1}}{\Delta x^2}$$

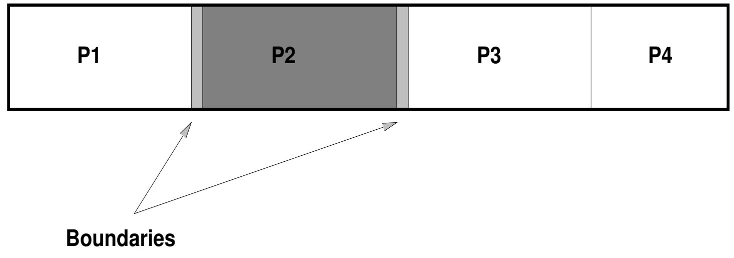
giving the explicit formula :

$$A_{i+1,j} = A_{i,j} + \frac{\Delta t}{\Delta x^2}(A_{i,j+1} - 2A_{i,j} + A_{i,j-1})$$

initial and boundary conditions:

$$A(t,0) = 0, A(t,1) = 0 \text{ for all } t$$
$$A(0,x) = sin(\pi x) \text{ for } 0 \leq x \leq 1$$

1-D Heat Equation Continuation



Solution:

- Master Slave
- Slaves communicate their boundaries values
- Uses the following PVM features:
  - spawn
  - group
  - barrier
  - send-recv
  - pack-unpack

1-D Heat Equation Pseudo Code

- **Master**

```
Set the initial temperatures
for i = 1 to <number of slaves>
  send the ith part of the initial
    temperatures to the ith slave
end for
for i = 1 to <number of slaves>
  receive results from ith slave
  update my data
end for
```

- **Slave**

```
Receive my part of the initial values
for i = 1 to <number of time iterations>
  send my left bound to my left neighbor
  send my right bound to my right neighbor
  receive my left neighbor's left bound
  receive my right neighbor's right bound
  compute the new temperatures
end for

  send back my result to the master
```



```
C
C  Use PVM to solve a simple heat diffusion differential equation,
C  using 1 master program and 5 slaves.
C
C  The master program sets up the data, communicates it to the slaves
C  and waits for the results to be sent from the slaves.
C  Produces xgraph ready files of the results.
C

      program heat
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

      integer NPROC, TIMESTEP, PLOTINC, SIZE
      double precision PI

      PARAMETER(PI = 3.14159265358979323846)
      PARAMETER(NPROC = 3)
      PARAMETER(TIMESTEP = 10)
      PARAMETER(PLOTINC = 1)
      PARAMETER(SIZE = 100)
      PARAMETER(SLAVENAME = 'heatslv')

      integer num_data
      integer mytid, task_ids(NPROC), i, j
      integer left, right, k, l
      integer step
      integer ierr
      external wh
      integer wh

      double precision init(SIZE)
      double precision result(TIMESTEP*SIZE/NPROC)
      double precision solution(TIMESTEP,SIZE)
      character*20 filename(4)
      double precision deltat(4), deltax2
      real etime
      real t0(2)
      real eltime(4)

      step = TIMESTEP
      num_data = INT(SIZE/NPROC)

      filename(1) = 'graph1'
      filename(2) = 'graph2'
      filename(3) = 'graph3'
      filename(4) = 'graph4'
      deltat(1) = 5.0E-1
      deltat(2) = 5.0E-3
      deltat(3) = 5.0E-6
      deltat(4) = 5.0E-9
```

```
C                       enroll in pvm
       call pvmfmytid( mytid)

C                       spawn the slave tasks
       call pvmfspawn('heatslv',PVMDEFAULT,'*',NPROC,task_ids,ierr)

C                       create the initial data set
       do 10 i = 1,SIZE
         init(i) = SIN(PI * DBLE(i-1) / DBLE(SIZE-1))
10     continue
       init(1) = 0.d0
       init(SIZE) = 0.d0

C  run the problem 4 times for different values of delta t

       do 20 l=1,4

         deltax2 = (deltat(l)/((1.0/(DBLE(SIZE)))**(2.0)))
C                     start timing for this run

         eltime(l) = etime(t0)

C     send the initial data to the slaves.
C     include neighbor info for exchanging boundary data
         do 30 i =1,NPROC
           call pvmfinitsend(PVMDEFAULT,ierr)
           IF (i .EQ. 1) THEN
             left = 0
           ELSE
             left = task_ids(i-1)
           ENDIF
           call pvmfpack(INTEGER4, left, 1, 1, ierr)
           IF (i .EQ. NPROC) THEN
             right = 0
           ELSE
             right = task_ids(i+1)
           ENDIF
           call pvmfpack(INTEGER4, right, 1, 1, ierr)
           call pvmfpack(INTEGER4, INT(step), 1, 1, ierr)
           call pvmfpack(REAL8, deltax2, 1, 1, ierr)
           call pvmfpack(INTEGER4, INT(num_data), 1, 1, ierr)
           call pvmfpack(REAL8, init(num_data*(i-1)+1),num_data,1,ierr)
           call pvmfsend(task_ids(i), 4, ierr)
30       continue

C    wait for the results
         do 40 i = 1,NPROC
           call pvmfrecv(task_ids(i), 7, ierr)
           call pvmfunpack(REAL8, result, num_data*TIMESTEP, 1, ierr)
```



```
C   update the solution
          do 50 j = 1, TIMESTEP
            do 60 k = 1, num_data
               solution(j,num_data*(i-1)+1+(k-1))  =
     $               result(wh(j-1,k-1,num_data)+1)
60          continue
50        continue
40      continue

C   stop timing
        eltime(l) = etime(t0) - eltime(l)

C   produce the output
        write(*,*) 'Writing output to file ',filename(l)
        open(23, FILE = filename(l))
        write(23,*) 'TitleText: Wire Heat over Delta Time: ',deltat(l)
        write(23,*) 'XUnitText: Distance'
        write(23,*) 'YUnitText: Heat'
        do 70 i=1,TIMESTEP,PLOTINC
          write(23,*) '"Time index: ',i-1
          do 80 j = 1,SIZE
            write(23,*) j-1,REAL(solution(i,j))
81     FORMAT(I5,F10.4)
80        continue
          write(23,*) ''
70      continue
        endfile 23
        close(UNIT = 23, STATUS = 'KEEP')

20    continue

      write(*,*) 'Problem size: ', SIZE
      do 90 i = 1,4
        write(*,*) 'Time for run ',i-1,': ',eltime(i),' sec.'
90    continue

C  kill the slave processes
      do 100 i = 1,NPROC
        call pvmfkill(task_ids(i),ierr)
100   continue
      call pvmfexit(ierr)
      END

      integer FUNCTION wh(x,y,z)
      integer x,y,z
      wh = x * z + y

      RETURN
      END
```

```
C
C    The slaves receive the initial data from the host,
C    exchange boundary information with neighbors,
C    and calculate the heat change in the wire.
C    This is done for a number of iterations, sent by the master.
C
C

      program heatslv
      include '/src/icl/pvm/pvm3/include/fpvm3.h'

      PARAMETER(MAX1 = 1000)
      PARAMETER(MAX2 = 100000)

      integer  mytid, left, right, i, j, master
      integer timestep

      external wh
      integer wh

      double precision init(MAX1), A(MAX2)
      double precision leftdata, rightdata
      double precision delta, leftside, rightside

C  enroll in pvm

      call pvmfmytid(mytid)
      call pvmfparent(master)

C  receive my data from the master program
10    continue

      call pvmfrecv(master,4,ierr)
      call pvmfunpack(INTEGER4, left, 1, 1, ierr)
      call pvmfunpack(INTEGER4, right, 1, 1, ierr)
      call pvmfunpack(INTEGER4, timestep, 1, 1, ierr)
      call pvmfunpack(REAL8, delta, 1, 1, ierr)
      call pvmfunpack(INTEGER4, num_data, 1, 1, ierr)
      call pvmfunpack(REAL8, init, num_data, 1, ierr)

C copy the initial data into my working array

      do 20 i = 1, num_data
        A(i) = init(i)
20    continue
      do 22 i = num_data+1, num_data*timestep
        A(i) = 0
22    continue
```



```
C perform the calculation

      do 30 i = 1, timestep-1

C      trade boundary info with my neighbors
C      send left, receive right

        IF (left .NE. 0) THEN
          call pvmfinitsend(PVMDEFAULT, ierr)
          call pvmfpack(REAL8, A(wh((i-1),0,num_data)+1), 1, 1, ierr)
          call pvmfsend( left, 5, ierr)
        ENDIF
        IF (right .NE. 0) THEN
          call pvmfrecv( right, 5, ierr)
          call pvmfunpack(REAL8, rightdata, 1, 1, ierr)
          call pvmfinitsend(PVMDEFAULT, ierr)
          call pvmfpack(REAL8, A(wh((i-1), num_data-1,num_data)+1),
     $                  1, 1, ierr)
          call pvmfsend(right, 6, ierr)
        ENDIF
        IF (left .NE. 0) THEN
          call pvmfrecv(left, 6, ierr)
          call pvmfunpack(REAL8, leftdata, 1, 1, ierr)
        ENDIF

C do the calculations for this iteration

        do 40 j = 1, num_data
          IF (j .EQ. 1) THEN
            leftside = leftdata
          ELSE
            leftside = A(wh(i-1,j-2,num_data)+1)
          ENDIF

          IF (j .EQ. num_data) THEN
            rightside = rightdata
          ELSE
            rightside = A(wh(i-1,j,num_data)+1)
          ENDIF

          IF ((j. EQ. 1) .AND. (left. EQ. 0)) THEN
            A(wh(i,j-1,num_data)+1) = 0.d0
          ELSE IF ((j .EQ. num_data) .AND. (right .EQ. 0)) THEN
            A(wh(i,j-1,num_data)+1) = 0.d0
          ELSE
            A(wh(i,j-1,num_data)+1) = A(wh(i-1,j-1,num_data)+1) +
     $    delta*(rightside - 2*A(wh(i-1,j-1,num_data)+1)+leftside)
          ENDIF
40      continue
30    continue
```

```
C send the results back to the master program

      call pvmfinitsend(PVMDEFAULT, ierr)
      call pvmfpack(REAL8, A, num_data*timestep, 1, ierr)
      call pvmfsend(master, 7, ierr)

      goto 10

C just for good measure
      call pvmfexit(ierr)

      END

      integer FUNCTION wh(x,y,z)

      integer x,y,z

      wh = x*z + y

      RETURN
      END
```


Motivation for a New Design

- Message Passing now mature as programming paradigm
  - well understood
  - efficient match to hardware
  - many applications
- Vendor systems not portable
- Portable systems are mostly research projects
  - incomplete
  - lack vendor support
  - not at most efficient level

Motivation (cont.)

Few systems offer the full range of desired features.

- modularity (for libraries)
- access to peak performance
- portability
- heterogeneity
- subgroups
- topologies
- performance measurement tools

The MPI Process

- Began at Williamsburg Workshop in April, 1992
- Organized at Supercomputing '92 (November)
- Followed HPF format and process
- Met every six weeks for two days
- Extensive, open email discussions
- Drafts, readings, votes
- Pre-final draft distributed at Supercomputing '93
- Two-month public comment period
- Final version of draft in May, 1994
- Widely available now on the Web, ftp sites, netlib (`http://www.netlib.org/mpi/index.html`)
- Public implementations available
- Vendor implementations coming soon

MPI Lacks...

- Mechanisms for process creation
- One sided communication (put, get, active messages)
- Language binding for Fortran 90 anc C++

There are a fixed number of processes from start to finish of an applicaiton. Many features were considered and not included

- Time constraint
- Not enough experience
- Concern that additional features would delay the appearance of implementations

Who Designed MPI?

- Broad participation
- Vendors
  - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers
  - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- Application specialists and consultants

Companies	Laboratories	Universities
ARCO	ANL	UC Santa Barbara
Convex	GMD	Syracuse U
Cray Res	LANL	Michigan State U
IBM	LLNL	Oregon Grad Inst
Intel	NOAA	U of New Mexico
KAI	NSF	Miss. State U.
Meiko	ORNL	U of Southampton
NAG	PNL	U of Colorado
nCUBE	Sandia	Yale U
ParaSoft	SDSC	U of Tennessee
Shell	SRC	U of Maryland
TMC		Western Mich U
		U of Edinburgh
		Cornell U.
		Rice U.
		U of San Francisco

What is MPI?

- A *message-passing library specification*
  - message-passing model
  - not a compiler specification
  - not a specific product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

New Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

New Features of MPI (cont.)

- Application-oriented process topologies
  - Built-in support for grids and graphs (uses groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - inquiry
  - error control

Features not in MPI

- Non-message-passing concepts not included:
  - process management
  - remote memory transfers
  - active messages
  - threads
  - virtual shared memory
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)

Is MPI Large or Small?

- MPI is large (125 functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

Header files

- **C**

  #include <mpi.h>

- **Fortran**

  include 'mpif.h'

MPI Function Format

- C:

  error = MPI_xxxxx(parameter, ...);
  MPI_xxxxx(parameter, ...);

- Fortran:

  CALL MPI_XXXXX(parameter, ..., IERROR)

Initializing MPI

- C

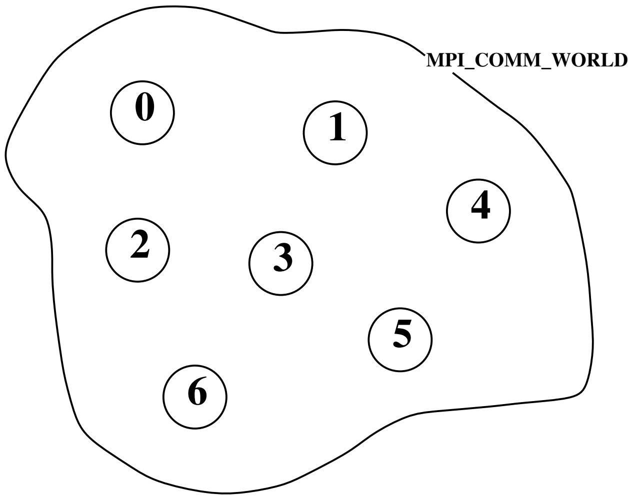
  int MPI_Init(int *argc, char ***argv)

- Fortran

  MPI_INIT(IERROR)
  INTEGER IERROR

- Must be first routine called.

MPI_COMM_WORLD communicator



Rank

- **How do you identify different processes?**

MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR

Size

- How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
   INTEGER COMM, SIZE, IERROR
```


Exiting MPI

- C

```
int MPI_Finalize()
```

- Fortran

```
MPI_FINALIZE(IERROR)
   INTEGER IERROR
```

- Must be called last by all processes.

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic types.
  - Derived types.
- Derived types can be built up from basic types.
- C types are different from Fortran types.

MPI Basic Datatypes - C

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Example 3.1 MPI code that sends an upper triangular matrix.

```
double a[100][100]
disp[100],blocklen[100],i;
MPI_Datatype upper;
...
 /* compute start and size of each row */
for (i=0; i<100; ++i) {
   disp[i] = 100 * i + i;
   blocklen[i] = 100 - i;
}
 /* create datatype for upper triangular part */
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);
 /* .. and send it */
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
```

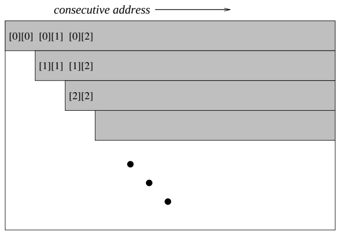


Figure 3.1
A diagram of the memory cells represented by the user-defined datatype `upper`. The shaded cells are the locations of the array that will be sent.

Derived datatypes are constructed from basic datatypes using the constructors described in Section 3.3. The constructors can be applied recursively.

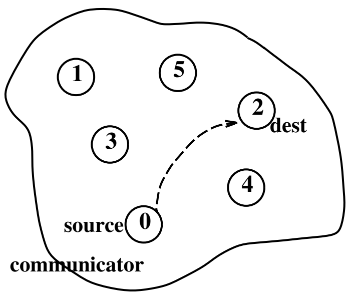
A **derived datatype** is an opaque object that specifies two things:

- A sequence of primitive datatypes and,
- A sequence of integer (byte) displacements.

MPI Basic Datatypes - Fortran

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Point–to–Point Communication



- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator.

Simple Fortran example

```
      program main
      include 'mpif.h'

      integer rank, size, to, from, tag, count, i, ierr
      integer src, dest
      integer st_source, st_tag, st_count
      integer status(MPI_STATUS_SIZE)
      double precision data(100)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
      print *, 'Process ', rank, ' of ', size, ' is alive'
      dest = size - 1
      src = 0
C
      if (rank .eq. src) then
         to     = dest
         count  = 10
         tag    = 2001
         do 10 i=1, 10
 10         data(i) = i
         call MPI_SEND( data, count, MPI_DOUBLE_PRECISION, to,
     +                  tag, MPI_COMM_WORLD, ierr )
      else if (rank .eq. dest) then
         tag   = MPI_ANY_TAG
         count = 10
         from  = MPI_ANY_SOURCE
         call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
     +                 tag, MPI_COMM_WORLD, status, ierr )
```

Simple Fortran example (cont.)

```
         call MPI_GET_COUNT( status, MPI_DOUBLE_PRECISION,
     +                       st_count, ierr )
         st_source = status(MPI_SOURCE)
         st_tag    = status(MPI_TAG)
C
         print *, 'Status info: source = ', st_source,
     +            ' tag = ', st_tag, ' count = ', st_count
         print *, rank, ' received', (data(i),i=1,10)
      endif

      call MPI_FINALIZE( ierr )
      end
```


Fortran example

```
      program main

      include "mpif.h"

      double precision  PI25DT
      parameter        (PI25DT = 3.141592653589793238462643d0)

      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, rc
c                                 function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

 10   if ( myid .eq. 0 ) then
         write(6,98)
 98      format('Enter the number of intervals: (0 quits)')
         read(5,99) n
 99      format(i10)
      endif

      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

Fortran example (cont.)

```
c                                 check for quit signal
     if ( n .le. 0 ) goto 30

c                                 calculate the interval size
     h = 1.0d0/n

     sum  = 0.0d0
     do 20 i = myid+1, n, numprocs
        x   = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
 20  continue
     mypi = h * sum

c                                 collect all the partial sums
     call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
    $     MPI_COMM_WORLD,ierr)

c                                 node 0 prints the answer.
     if (myid .eq. 0) then
        write(6, 97) pi, abs(pi - PI25DT)
 97     format('  pi is approximately: ', F18.16,
    +          '  Error is: ', F18.16)
     endif

     goto 10

 30  call MPI_FINALIZE(rc)
     stop
     end
```


C example

```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

C example (cont.)

```
  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```


Communication modes

Sender mode	Notes
Synchronous send	Only completes when the receive has started.
Buffered send	Always completes (unless an error occurs), irrespective of receiver.
Standard send	Either synchronous or buffered.
Ready send	Always completes (unless an error occurs), irrespective of whether the receive has completed.
Receive	Completes when a message has arrived.

MPI Sender Modes

OPERATION	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

Sending a message

- C:

  int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- Fortran:

  MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG INTEGER COMM, IERROR

Receiving a message

- C:

  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

- Fortran:

  MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

  <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

Synchronous Blocking Message-Passing

- Processes synchronize.
- Sender process specifies the synchronous mode.
- Blocking - both processes wait until the transaction has completed.

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.

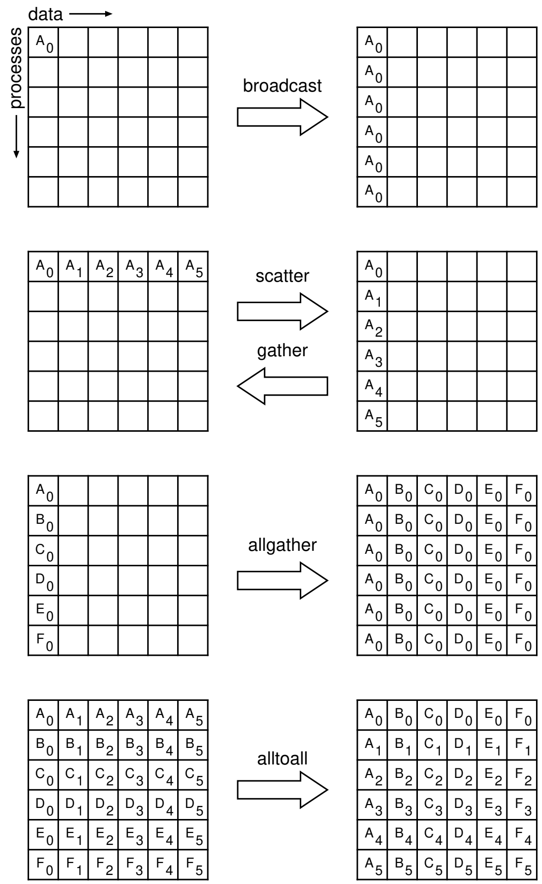
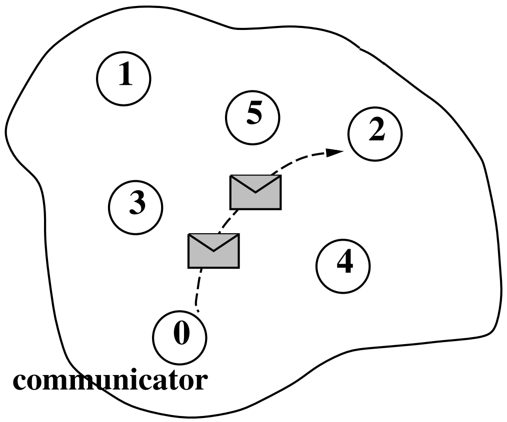


Figure 4.1
Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the item A_0, but after the broadcast all processes contain it.

Wildcarding

- Receiver can wildcard.
- To receive from any source - MPI_ANY_SOURCE
- To receive with any tag - MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter.

Message Order Preservation

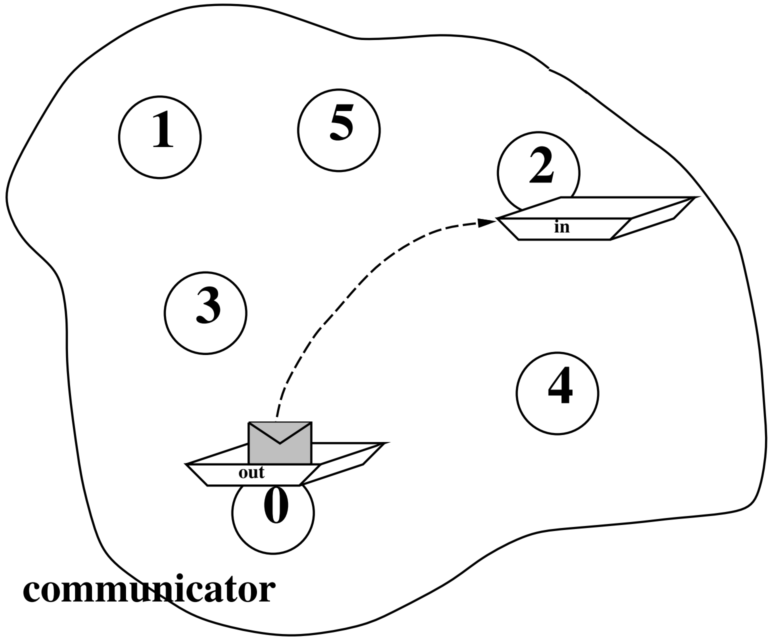


- Messages do not overtake each other.
- This is true even for non–synchronous sends.

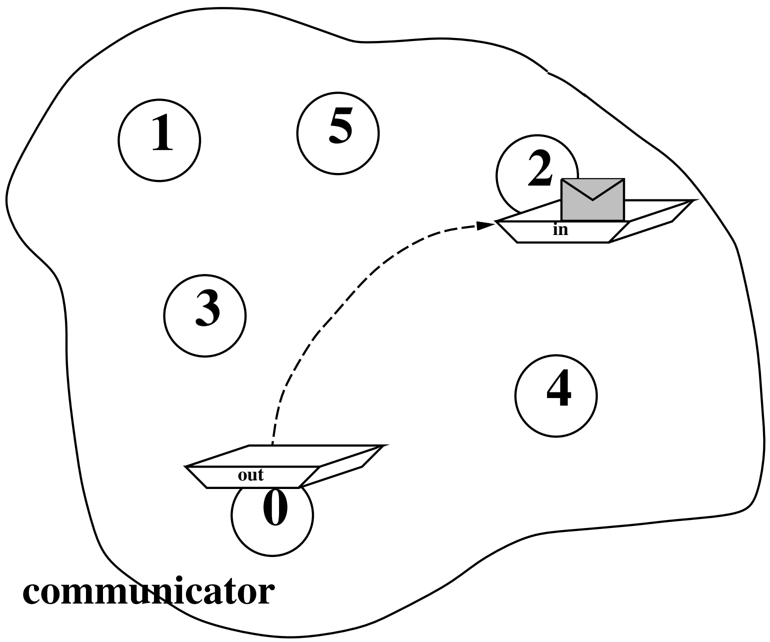
Non-Blocking Communications

- Separate communication into three phases:
- Initiate non-blocking communication.
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete.

Non–Blocking Send



Non–Blocking Receive



Non-blocking Synchronous Send

- **C:**

  MPI_Issend(buf, count, datatype, dest, tag, comm, handle)

  MPI_Wait(handle, status)

- **Fortran:**

  MPI_ISSEND(buf, count, datatype, dest, tag, comm, handle, ierror)

  MPI_WAIT(handle, status, ierror)

Non-blocking Receive

- C:

  MPI_Irecv(buf, count, datatype, src, tag, comm, handle)

  MPI_Wait(handle, status)

- Fortran:

  MPI_IRECV(buf, count, datatype, src, tag, comm, handle, ierror)

  MPI_WAIT(handle, status, ierror)

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode - synchronous, buffered, standard, or ready.
- Synchronous mode affects completion, not initiation.

Communication Modes

NON-BLOCKING OPERATION	MPI CALL
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_IBSEND
Ready send	MPI_IRSEND
Receive	MPI_IRECV

Completion

- **Waiting versus Testing.**
- **C:**

  MPI_Wait(handle, status)
  MPI_Test(handle, flag, status)

- **Fortran:**

  MPI_WAIT(handle, status, ierror)
  MPI_TEST(handle, flag, status, ierror)

Characteristics of Collective Communication

- **Collective action over a communicator**
- **All processes must communicate.**
- **Synchronisation may or may not occur.**
- **All collective operations are blocking.**
- **No tags**
- **Receive buffers must be exactly the right size.**

Barrier Synchronization

- **C:**

```
int MPI_Barrier (MPI_Comm comm)
```

- **Fortran:**

```
MPI_BARRIER (COMM, IERROR)
   INTEGER COMM, IERROR
```

Broadcast

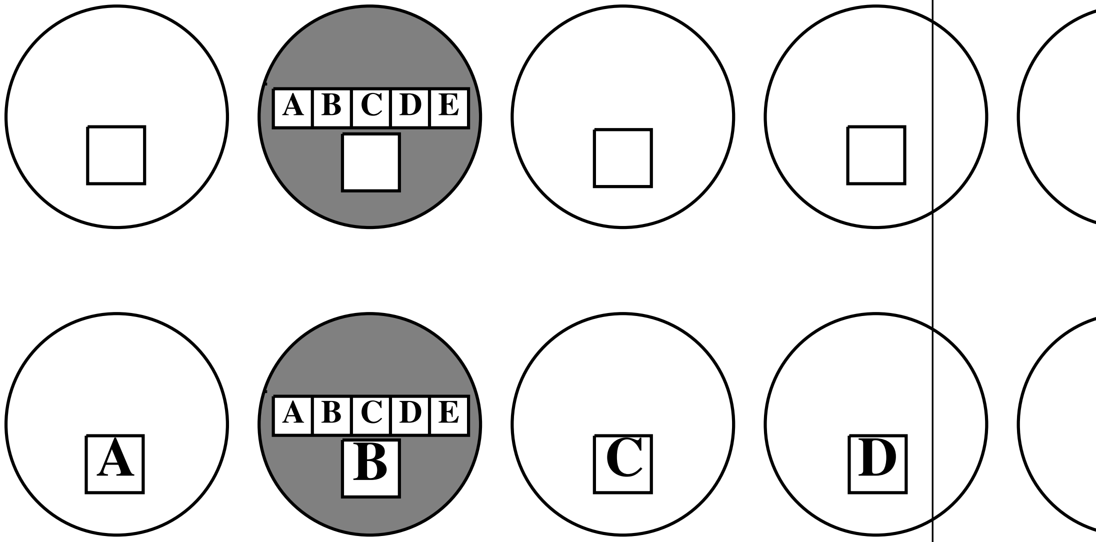
- C:

  int MPI_Bcast (void *buffer, int count, MPI_datatype, int root, MPI_Comm comm)

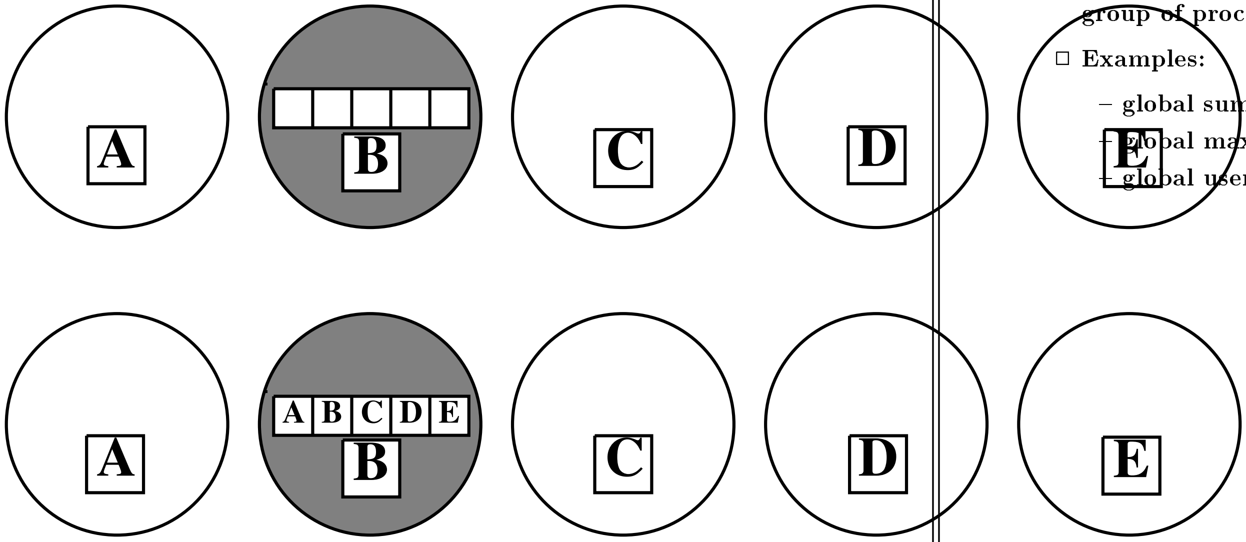
- Fortran:

  MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
  <type> BUFFER(*)
  INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

Scatter



Gather



Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

Example of Global Reduction

Integer global sum

- **C:**

```
MPI_Reduce(&x, &result, 1, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD)
```

- **Fortran:**

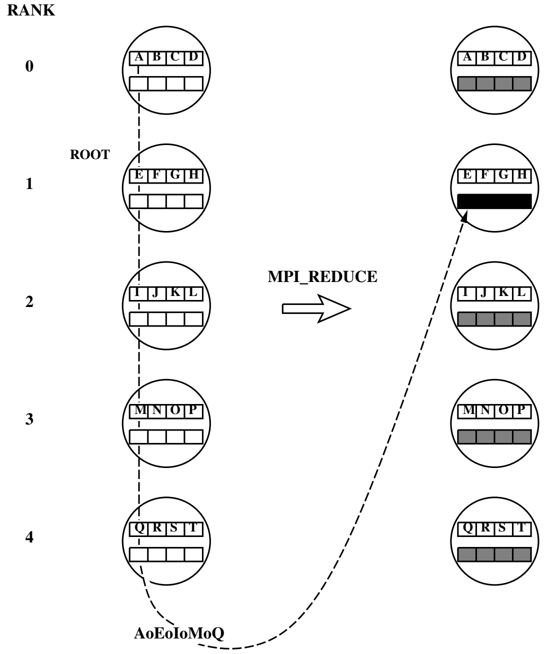
```
CALL MPI_REDUCE(x, result, 1, MPI_INTEGER,
           MPI_SUM, 0, MPI_COMM_WORLD, IERROR)
```

- **Sum of all the** x **values is placed in** result
- **The result is only placed there on processor 0**

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

MPI_REDUCE



User-Defined Reduction Operators

- **Reducing using an arbitrary operator,**
- **C - function of type** MPI_User_function:

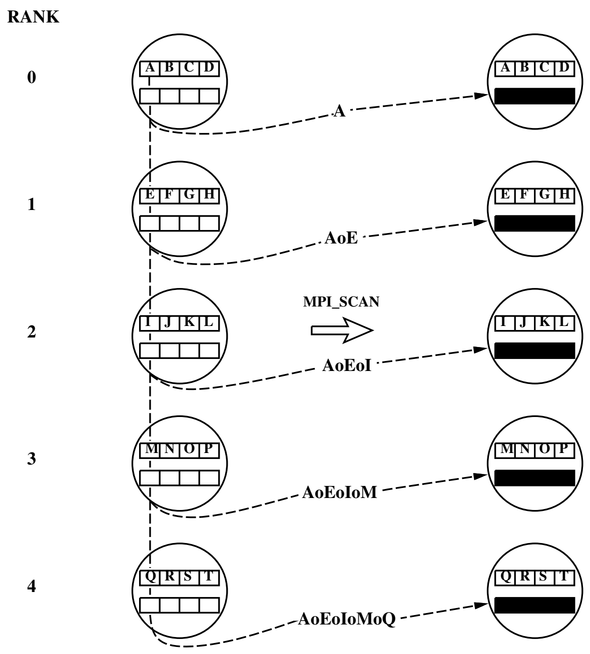
  void my_operator (void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)

- **Fortran - function of type**

  FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*), LEN, DATATYPE)

  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, DATATYPE

MPI_SCAN



Summary

We have covered

- **Background and scope of MPI**
- **Some characteristic features of MPI (communicators, datatypes)**
- **Point-to-Point communication**
  - **blocking and non-blocking**
  - **multiple modes**
- **Collective communication**
  - **data movement**
  - **collective computation**

Summary

- **The parallel computing community has cooperated to develop a full-featured standard message-passing library interface.**
- **Implementations abound**
- **Applications beginning to be developed or ported**
- **MPI-2 process beginning**
- **Lots of MPI material available**

Current MPI Implementation Efforts

Vendor Implementations
IBM Research (MPI-F)
IBM Kingston
Intel SSD
Cray Research
Meiko, Inc.
SGI
Kendall Square Research
NEC
Fujitsu (AP1000)
Convex
Hughes Aircraft
Portable Implementations
Argonne–Mississippi State (MPICH)
Ohio supercomputer Center (LAM)
University of Edinburgh
Technical University of Munich
University of Illinois

Other interested groups: Sun, Hewlett-Packard, Myricom (makers of high-performance network switches) and PALLAS (a German software company), Sandia National Laboratory (Intel Paragon running SUNMOS)

MPI Implementation Projects

- **Variety of implementations**
  - **Vendor proprietary**
  - **Free, portable**
  - **World wide**
  - **Real-time, embedded systems**
  - **All MPP's and networks**
- **Implementation strategies**
  - **Specialized**
  - **Abstract message-passing devices**
  - **Active-message devices**

MPICH – A Freely-available Portable MPI Implementation

- **Complete MPI implementation**
- **On MPP's: IBM SP1 and SP2, Intel IPSC860 and Paragon, TMC CM-5, SGI, Meiko CS-2, NCube, KSR, Sequent Symmetry**
- **On workstation networks: Sun, Dec, HP, SGI, Linux, FreeBSD, NetBSD**
- **Includes multiple profiling libraries for timing, event logging, and animation of programs.**
- **Includes trace upshot visualization program, graphics library**
- **Efficiently implemented for shared-memory, high-speed switches, and network environments**
- **Man pages**
- **Source included**
- **Available at ftp.mcs.anl.gov in pub/mpi/mpich.tar.Z**

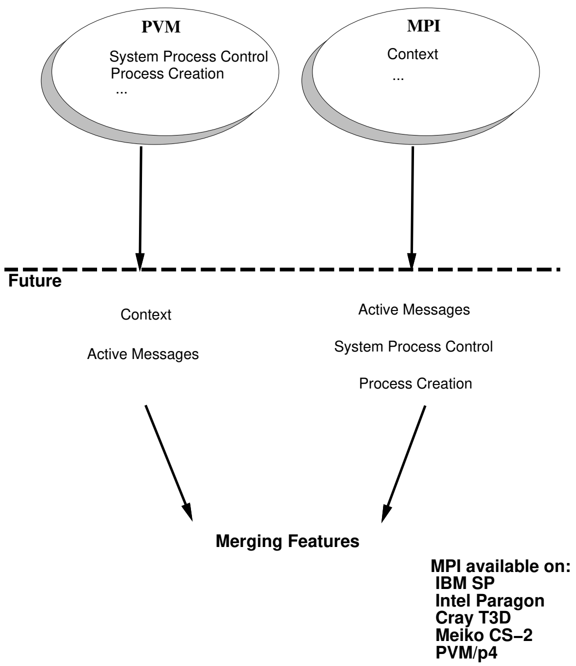
Sharable MPI Resources

- **The Standard itself:**
  - **As a Technical report: U. of Tennessee. report**
  - **As postscript for ftp: at** `info.mcs.anl.gov` **in** `pub/mpi/mpi-report.ps`.
  - **As hypertext on the World Wide Web:** `http://www.mcs.anl.gov/mpi`
  - **As a journal article: in the Fall issue of the Journal of Supercomputing Applications**
- **MPI Forum discussions**
  - **The MPI Forum email discussions and both current and earlier versions of the Standard are available from** `netlib`.
- **Books:**
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, **by Gropp, Lusk, and Skjellum, MIT Press, 1994**
  - *MPI Annotated Reference Manual*, **by Otto, Dongarra, Lederman, Snir, and Walker, MIT Press, 1995.**

Sharable MPI Resources, continued

- **Newsgroup:**
  - `comp.parallel.mpi`
- **Mailing lists:**
  - `mpi-comm@cs.utk.edu`: **the MPI Forum discussion list.**
  - `mpi-impl@mcs.anl.gov`: **the implementors' discussion list.**
- **Implementations available by** `ftp`:
  - **MPICH is available by anonymous** `ftp` **from** `info.mcs.anl.gov` **in the directory** `pub/mpi/mpich`, **file** `mpich.*.tar.Z`.
  - **LAM is available by anonymous** `ftp` **from** `tbag.osc.edu` **in the directory** `pub/lam`.
  - **The CHIMP version of MPI is available by anonymous** `ftp` **from** `ftp.epcc.ed.ac.uk` **in the directory** `pub/chimp/release`.
- **Test code repository (new):**
  - `ftp://info.mcs.anl.gov/pub/mpi-test`

PVM and MPI Future



MPI-2

- **The MPI Forum (with old and new participants) has begun a follow-on series of meetings.**
- **Goals**
  - **clarify existing draft**
  - **provide features users have requested**
  - **make extensions, not changes**
- **Major Topics being considered**
  - **dynamic process management**
  - **client/server**
  - **real-time extensions**
  - **"one-sided" communication (put/get, active messages)**
  - **portable access to MPI system state (for debuggers)**
  - **language bindings for C++ and Fortran-90**
- **Schedule**
  - **Dynamic processes, client/server by SC '95**
  - **MPI-2 complete by SC '96**

Conclusions

- **MPI being adopted worldwide**
- **Standard documentation is an adequate guide to implementation**
- **Implementations abound**
- **Implementation community working together**