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:
  - Mailbox
  - 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.

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 Forge® 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 (i.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 rep 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 (i.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-recev
    - Used by ScALAPACK

Bag of Tasks

- Components
  - Job pool
  - Worker pool
  - Scheduler

- 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
  - Odd-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

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 specs 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

Physical and Logical Views of PVM

Physical

IP Network (routers, bridges, ...)

Host

Multihost

Logical

Pvmd (host)

Tasks

Console(s)

Programming in PVM

- A simple message-passing environment
  - Host, 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)
Hello World

Program hello.c, the main program:

```c
#include<stdio.h>
#include"pvm.h"

main(){
    int tid; /* tid of child*/
    char buf[100];
    printf("I'm tid:\n");
    pvm_mytid();
    pvm_spawn("hello", 0, "", 1, stdin);
    pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
    pvm_upkstr(buf);
    printf("Message from tid/: t\n", tid, buf);
    pvm_exit();
}
```

Program hello2.c, the slave program:

```c
#include"pvm.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_pktstr(buf);
    pvm_send(ptid, 1);
    pvm_exit();
}
```

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
    - PCL
  - One-of-a-kind
    - NX
    - CMMD
  - Other types of communication
    - AM
    - Linda
- Also DOOs, Languages, ...

Portability

- Configurations include
  - HP (2.0), DEC VAX (4.0, 5.1, 6)
  - IBM (3090, 3081, 3083, 3086, 3088, 3090)
  - DEC (Sparc, Ultrasparc, Alpha, VAX)
  - Convex C2, CSSP
  - DG/Alliant
  - Convex/HP, SGI, Alpha, Sun, KSR, Paragon
  - Source code largely shared with generic (8000)
- PVM is portable to non-Unix machines
  - Windows/NT port has been done
  - OS/2 port has been done
  - Distributed-memory: T-3D, IPSC/860, Paragon, CM-5, SP-2
  - 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://oss/epm/orl/epm/propvm/index.html
- FTP: host netlib2.cs.utk.edu, login anonymous, directory /pvm
  - Bug reports, comments, questions can be mailed to pvmnetlib@ornl.gov
  - Usenet newsgroup for discussion and support: comp.parallel.pvm
- Book:
  PVM: Parallel Virtual Machine
  A User’s Guide and Tutorial for Networked Parallel Computing
Installing PVM

- Package requires a few MB of disk + a few MB / architecture
- Don’t need root privilege
- 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/pvm3.4 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
    export PVM_ROOT PVM_DPATH
- Important directories below $PVM_ROOT
<table>
<thead>
<tr>
<th>Include</th>
<th>Header files</th>
<th>Manual pages</th>
<th>Scripts</th>
<th>bin/ARCH</th>
<th>System executables</th>
<th>bin/ARCH</th>
<th>System tasks</th>
</tr>
</thead>
</table>

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 pvm, console, libraries, group server and library
- Installs executables and libraries in lib and bin

Starting PVM

- Three ways to start PVM
- pvm [-d debugmask] [-hostname] [hostfile]
  PVM console starts pvm, or connects to one already running
- xpvm
  Graphical console, same as above
- pvm [-d debugmask] [-hostname] [hostfile]
  Manual start, used mainly for debugging or when necessary to enter passwords
- Some common error messages
  - Can’t start pvm
    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 $PVM_ROOT not correct
  - simget: ... 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 \rightarrow tids |
| Machine configuration | Add, remove hosts | Query host status | Start, halt virtual machine |
| Miscellaneous | Get, set options | Request notification | Register special tasks | Get host times/day clock offsets |

### Basic PVM Communication

- Three-step send method
  - `pvm_initsend(encoding)`
    - Initialize send buffer, clearing current one
  - `pvm_send(dest, tag, data, num_items, data_type)`
  - `pvm_recv(source, tag)`
    - Pack buffer with various data
  - `pvm_send(dest, tag)`
  - `pvm_recv(source, tag)`
    - Sends buffer to other task(s), returns when safe to clear buffer

- To receive
  - `pvm_recv(source, tag)`
  - `pvm_probe(source, tag)`
    - Blocking or non-blocking receive
  - `pvm_unpack(data, num_items, stride)`
    - Unpack message into user variables
  - Can also `pvm_probe(source, tag)` for a message

- Another receive primitive: `pvm_probe(source, tag, timeout)`
  - Equivalent to `pvm_recv` 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)`
  - `pvm نيوز(anonymous, tag, data, num_items, data_type)`
    - Pack and send a contiguous, single-typed data buffer
  - As fast as native calls on multiprocessor machines

### Process Control

- Start new tasks
  - Placement options
    - `PvmFull"default"`\text{default}
    - `PvmStart"unmatched"`\text{unmatched}
    - `PvmStart"unmatched"`\text{unmatched}
  - Other flags
    - `PvmStart"unmatched"`\text{unmatched}
    - `PvmStart"unmatched"`\text{unmatched}
    - `PvmStart"unmatched"`\text{unmatched}
  - Spawn can return `partial success`

- Find my task id / enroll as a task

- Find parent’s task id

- Disconnect from PVM

- Terminate another PVM task

- Query status of another PVM task
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)
  - pvm_reduce(*func(), data, num_items, data_type, msgtag, group, rootinst)

Distribute and collect arrays across task groups

Virtual Machine Control

- pvm_addhosts(hosts, numhosts, tids)
  - Add hosts to virtual machine
- pvm_config(hosts, narch, hosts)
  - Get current VM configuration
- pvm_delhosts(hosts, numhosts, 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
  - master, slave
  - spmd
  - gexample
  - timing, timing_slave
  - hitc, hitc_slave
  - xep, xinf
- Examples come with Makefile/aimk files
- Both C and Fortran versions for some examples

Compiling Applications

- Header files
  - C programs should include
    - <pvm.h> Always
    - <pvmtev.h> To manipulate trace masks
    - <pvmsdpro.h> For resource manager interface
  - Specify include directory: cc -I$PVM_ROOT/include ...
  - Formats: INCLUDE '/usr/local/pvm/include/pvms.h'
- Compiling and linking
  - C programs must be linked with
    - libpvm.a Always
    - libgpvm.a If using group library functions
  - Fortran programs must additionally be linked with libfpvm.a
Compiling Applications, Cont'd

- Aim
  - Shares single makefile between architectures
  - Builds for different architectures in separate directories
  - Determines PVM architecture
  - Runs make, passing in $PVM_ARCH
  - Does one of three things
    - If $PVM_ARCH/Makefile exists:
      Runs make in subdirectory, using makefile
    - Else if Makefile.simk exists:
      Creates subdirectory, runs make using Makefile.simk
    - 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 = xer
      - Forum = hive
      - Can include some fault tolerance
    - Work migration: Cancel / forward job
      - Poll for cancel message from master
      - Can interrupt with pvm_sendmsg
      - 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 spmd
#include <src/impl/pvm/pvm.h>
PARAMETER(NPROC=4)
integer rank, left, right, i, j, ierr
integer tids[NPROC-1]

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

call pvmfbarrier('foo', NPROC, ierr)

C compute the neighbours IDs

call pvmfgettid('foo', MOD(rank+1, NPROC), NPROC, left)
call pvmfgettid('foo', MOD(rank, NPROC), NPROC, right)
if(rank.eq.0) then
  C I am the first process
  do i=1, NPROC
    data(i) = 0
    call pvmfinitsend(PVM_DEFAULT, ierr)
call pvmfpack(INTEGER/4, data, NPROC, ierr)
call pvmfsend(right, 1, ierr)
call pvmfrecv(left, 1, ierr)
call pvmfunpack(INTEGER/4, data, NPROC, ierr)
    write/*,/* /' Results received /:/'
    write/*,/* /' /data(j)/j = ', j=1, NPROC
  enddo
else
  C I am an intermediate process
    call pvmfrecv(left, 1, ierr)
call pvmfunpack(INTEGER/4, data, NPROC, ierr)
data(rank+1) = rank
    call pvmfinitsend(PVM_DEFAULT, ierr)
call pvmfpack(INTEGER/4, data, NPROC, ierr)
call pvmfsend(right, 1, ierr)
endif

call pvmf grown('foo', ierr)
call pvmfexit(ierr)
stop
end

Inner Product
Problem: In parallel compute
\[ s = \sum_{j=1}^{n} x_{j} y_{j} \]

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/impl/pvm/pvm.h>
PARAMETER(NPROC=4)
PARAMETER(N=100)
double precision ddot extern ddo
integer rank, i, ierr, bufid
integer tids[NPROC], slave, master
double precision x(N), y(N)
double precision rem, result

rem = MOD(N, NPROC-1)
if(rank.eq.0) then
  call pvmfspawn('inner', PVM_DEFAULT, '/*', NPROC-1, tids, ierr)
endif

C MASTER
if(rank.eq.0) then
  C Set the values
  do i=1, N
    x(i) = 1.d0
    y(i) = 1.d0
  enddo

  C Send the data
  count = 1
  do i=1, NPROC-1
    call pvmfinitsend(PVM_DEFAULT, ierr)
call pvmfpack(REAL/8, x(count), rem, ierr)
call pvmfpack(REAL/8, y(count), rem, ierr)
call pvmfgettid('foo', i, slave)
call pvmfsend(slave, 1, ierr)
call pvmfrecv(left, 1, ierr)
call pvmfunpack(REAL/8, x(count), rem, ierr)
call pvmfunpack(REAL/8, y(count), rem, ierr)
call pvmfrecv(right, 1, ierr)
call pvmfunpack(REAL/8, x(count), rem, ierr)
call pvmfunpack(REAL/8, y(count), rem, ierr)
    if(rank.eq.0) then
      C MASTER
      do i=1, N
        x(i) = x(i) + result
        y(i) = y(i) + result
      enddo
    endif
endif

**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-receive
  - pack-unpack

---

**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
  ```
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-rev
  - pack-unpack

\[ \begin{align*}
A & \quad X \quad Y \\
P1 & \quad P2 & \quad P3 & \quad P4 \\
\end{align*} \]
Matrix - Vector Product
(Column Distribution)

Pseudo Code

Master
for i = 1 to <number of slaves>
send i 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/pvmlinc/pvmlinc.h'

C y = y + A * x
C A : MxN (visible only on the slaves)
C X : N
C Y : M

PARAMETER(NPROC = 4)
PARAMETER(M = 9, N = 6)
double precision X, Y
external PVMSUM
integer tids(NPROC)
intrinsic INT
integer mytid, rank, i, ierr

call pvmfmytid(mytid)
call pvmfjoingroup('foo', rank)
if(rank .eq. 0) then
call pvmfspawn('matvec_col', PVM DEFAULT, '/*', NPROC, tids, ierr)
endif

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

C Data initialize for my part of the data
do i = 1, N
x(i) = 1.d0
continue

do i = 1, M
y(i) = 1.d0
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(PVM SUM, Y, M, REAL8, 1, 'foo', 0, ierr)
write(*,*) 'Results received'
do i = 1, M
write(*,*) 'Y('||,i,') = ', Y(i)
continue

call pvmfreduce(PVM SUM, Y, M, REAL8, 1, 'foo', 0, ierr)
ostop
end
Integration to evaluate $\pi$ (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
- broadcast
- reduce

Integration to evaluate $\pi$

Computer approximations to $\pi$ by using numerical integration

Know

$$\tan(45^\circ) = 1;$$

so that

$$\tan^2 \frac{\theta}{4} = \frac{1}{1 + \tan^2 \frac{\theta}{4}}$$

From the integral tables we can find

$$\tan^2 \frac{\theta}{4} = \int \frac{1}{1 + u^2} \, du$$

or

$$\tan^2 \frac{\theta}{4} = \int \frac{1}{1 + \tan^2 \frac{\theta}{4}} \, du$$

Using the midpoint rule with panels of uniform length $h = 1/n$, for various values of $n$.
Evaluate the function at the midpoints of each subinterval $[x_{i-1}, x_i]$.
$i\times h = h/2$ is the midpoint.

Formula for the integral is

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

$$\pi = h \times x$$

where

$$f(x) = \frac{4}{1 + x^2}$$
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:
\[
A_{i+1,j} = A_{i,j} - \Delta t \frac{A_{i,j+1} - 2A_{i,j} + A_{i,j-1}}{\Delta x^2}
\]

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

Solution:

- **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

- **Slave**
  - Receive my part of the initial values
  - for i = 1 to number of time iterations
  - compute the new temperatures
  - send back my result to the master
  - end for

1-D Heat Equation

Continuation

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

- **Slave**
  - Receive my part of the initial values
  - compute the new temperatures
  - send back my result to the master
The program was designed to solve a simple heat diffusion differential equation. It includes neighbor information for exchanging boundary data, and calculates the heat change in the wire. The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.

The slaves receive the initial data from the host, enroll in pvm, and calculate the heat change in the wire. Exchange boundary information with neighbors, and calculate the heat change in the wire.
Motivation for a New Design

- Message Passing now mature as programming paradigm
  - well understood
  - efficient match to hardware
  - many applications
- Vendor systems not portable
- Portatile 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 and C++

There are a fixed number of processes from start to finish of an application. 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, Melko, Cray, Convex, Neculae
- Library writers
  - PVM, p4, Zipeode, TCGMSG, Chameleon, Express, Linda
- Application specialists and consultants

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

<table>
<thead>
<tr>
<th>Company</th>
<th>Laboratory</th>
<th>University</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>ARMH</td>
<td>Synopsys U</td>
</tr>
<tr>
<td>Convex</td>
<td>Convex</td>
<td>Michigan St U</td>
</tr>
<tr>
<td>Cray Inc</td>
<td>LANL</td>
<td>Oregon Grad Bk</td>
</tr>
<tr>
<td>IBM</td>
<td>LANL</td>
<td>ORNL</td>
</tr>
<tr>
<td>Intel</td>
<td>NOAA</td>
<td>U of New Mexico</td>
</tr>
<tr>
<td>KAI</td>
<td>NASA</td>
<td>Mass. State U</td>
</tr>
<tr>
<td>Melko</td>
<td>ORNL</td>
<td>U of Southampton</td>
</tr>
<tr>
<td>NAG</td>
<td>PML</td>
<td>U of Colorado</td>
</tr>
<tr>
<td>nCUBE</td>
<td>Sandia</td>
<td>Yale U</td>
</tr>
<tr>
<td>Paracomp</td>
<td>NCSL</td>
<td>U of Pennsylvania</td>
</tr>
<tr>
<td>StM</td>
<td>SRC</td>
<td>U of Stuyvesant</td>
</tr>
<tr>
<td>TMC</td>
<td>SRC</td>
<td>Western Miss U</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U of Edinburgh</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cornell U</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rice U</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U of San Francisco</td>
</tr>
</tbody>
</table>
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**
  ```c
  #include <mpi.h>
  ```
- **Fortran**
  ```fortran
  include 'mpif.h'
  ```

MPI Function Format

- **C**
  ```c
  error = MPIxxxxx(parameter, ...);
  MPIxxxxx(parameter, ...);
  ```
- **Fortran**
  ```fortran
  CALL MPIXXXXX(parameter, ..., IERROR)
  ```

Initializing MPI

- **C**
  ```c
  int MPIInit(int argc, char **argv)
  ```
- **Fortran**
  ```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

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>Real</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>Double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>Long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>byte</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Packaged</td>
</tr>
</tbody>
</table>

MPI Basic Datatypes - Fortran

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>BYTE</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Packaged</td>
</tr>
</tbody>
</table>

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

```fortran
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)

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
if (rank .eq. src) then
    to = dest
    count = 1
    tag = 2
    do i = 1, count
        data(i) = i
        call MPI_SEND(data, count, MPI_DOUBLE_PRECISION, to, tag, MPI_COMM_WORLD, status, ierr)
    end do
else if (rank .eq. dest) then
    tag = MPI_ANY_TAG
    count = 1
    from = MPI_ANY_SOURCE
    call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from, tag, MPI_COMM_WORLD, status, ierr)

end if
```

```c
#include <mpi.h>

int main()

int rank, size, to, from, tag, count, i, ierr
int src, dest
int st_source, st_tag, st_count
int status[MPI_STATUS_SIZE]

call MPI_Init(& ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, & rank, & ierr)
call MPI_Comm_size(MPI_COMM_WORLD, & size, & ierr)
printf("Process %d of %d is alive", rank, size)
dest = size - 1
src = 0
if (rank == src) then
    to = dest
    count = 1
    tag = 2
    do i = 1, count
        data[i] = i
        call MPI_Send(data, count, MPI_DOUBLE_PRECISION, to, tag, MPI_COMM_WORLD, & status, & ierr)
    end do
else if (rank == dest) then
    tag = MPI_ANY_TAG
    count = 1
    from = MPI_ANY_SOURCE
    call MPI_Recv(data, count, MPI_DOUBLE_PRECISION, from, tag, MPI_COMM_WORLD, & status, & ierr)

end if
```
Simple Fortran example (cont.)

```fortran
program main
  include "mpif.h"
  double precision PI/2/5DT
  parameter (PI/2/5DT = 3.141592653589793238462643)
  double precision mypi, pi, h, sum, x, f, a
  integer n, myid, numprocs, i, rc

  c function to integrate
  f(a) = 4 * a * a * a

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

  if (myid.eq.0) then
    write(*,98)
  endif

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

  if (n.le.0) goto 30

  c calculate the interval size
  h = 1.0d0/n
  sum = 0.0d0

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

  mypi = h * sum

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

  c node 0 prints the answer.
  if (myid.eq.0) then
    write(*,97) pi, abs(pi - PI/2/5DT)
  endif
  goto 30

30 call MPI_FINALIZE(ierr)
stop
end
```

Fortran example

```fortran
program main
  include 'mpi.h'
  include <math.h>

  int main(argc, argv)
  int argc;
  char /*argv[5B]*/;

  int done = 0, n, myid, numprocs, i, rc;
  double PI/2/5DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, f, a

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

  if (myid == 0) then
    write(6, 97) pi, abs(pi - PI/2/5DT)
  endif
  goto 30
30 call MPI_Finalize(&rc)
  exit(0)
```

C example

```c
#include 'mpif.h'
#include <mpi.h>

int main(argc, argv)
int argc;
char /*argv[5B]*/;

int done = 0, n, myid, numprocs, i, rc;

double PI/2/5DT = 3.141592653589793238462643;

double mypi, pi, h, sum, x, f

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;
    s = 0.0;
    for (i = myid + 1; i <= n; i = i + numprocs) {
        s += h * (double) i / (i + h);
    }
    mypi = h * s;
    MPI_Bcast(mypi, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (myid == 0) {
        printf("pi is approximately %1.6f, Error is %1.6f\n",
               pi, fabs(pi - PI));
    }
    MPI_Finalize();
}

### Communication modes

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

### MPI Sender Modes

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>MPI CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_SEND</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_SSEND</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_BSEND</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_RSEND</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_RECV</td>
</tr>
</tbody>
</table>

### Sending a message

- **C:**
  ```c
  int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  ```

- **Fortran:**
  ```fortran
  MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  ```
Receiving a message

- C:
  ```c
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  ```
- Fortran:
  ```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.
**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 Receive

Non-blocking Synchronous Send

C:

MPI_Isend(buf, count, datatype, dest, tag, comm, handle)
MPI_Wait(handle, status)

Fortran:

MPI_ISSEND(buf, count, datatype, dest, tag, comm, handle, error)
MPI_WAIT(handle, status, error)

Non-blocking Receive

C:

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

Fortran:

MPI_RECV(buf, count, datatype, src, tag, comm, handle, error)
MPI_WAIT(handle, status, error)

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

<table>
<thead>
<tr>
<th>Non-blocking operation</th>
<th>MPI call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI ISEND</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI IBSEND</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI ISSEND</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI IRSEND</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI IRECV</td>
</tr>
</tbody>
</table>

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

- Fortran:
  ```fortran
  MPI_BCAST (BUFFER, 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:
  ```
  MPIReduce(&x, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
  ```
- Fortran:
  ```
  CALL MPIREDUCE(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

<table>
<thead>
<tr>
<th>MPI Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIMAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPIMIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPISUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPIPROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPILAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPIBAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPILOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPIBOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPILXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPIBXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPIMAXLOC</td>
<td>Maximum and location</td>
</tr>
<tr>
<td>MPIMINLOC</td>
<td>Minimum and location</td>
</tr>
</tbody>
</table>

User-Defined Reduction Operators

- Reducing using an arbitrary operator,
- C - function of type MPIUserFunction:
  ```
  void my_operator (void *invec, void *inoutvec, int *len,
                   MPIDataType *datatype)
  ```
- Fortran - function of type
  ```
  FUNCTION MY_OPERATOR (INVEC(*), INOUTVEC(*), LEN, DATATYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, DATATYPE
  ```
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

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

Current MPI Implementation Efforts

<table>
<thead>
<tr>
<th>Vendor Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Research (MPIFL)</td>
</tr>
<tr>
<td>IBM Kingston</td>
</tr>
<tr>
<td>Intel SSD</td>
</tr>
<tr>
<td>Cray Research</td>
</tr>
<tr>
<td>Meiko, Inc.</td>
</tr>
<tr>
<td>SGI</td>
</tr>
<tr>
<td>Kendall Square Research</td>
</tr>
<tr>
<td>NEC</td>
</tr>
<tr>
<td>Fujitsu (AP1000)</td>
</tr>
<tr>
<td>Convex</td>
</tr>
<tr>
<td>Hughes Aircraft</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Portable Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argonne—Mississippi State (MPICH)</td>
</tr>
<tr>
<td>Ohio supercomputer Center (LAM)</td>
</tr>
<tr>
<td>University of Edinburgh</td>
</tr>
<tr>
<td>Technical University of Munich</td>
</tr>
<tr>
<td>University of Illinois</td>
</tr>
</tbody>
</table>

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/mpi.ch.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:

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 thag.unc.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

PVM
System Process Control
Process Creation

MPI
Context

Active Messages
System Process Control
Process Creation

Merging Features

MPI available on:
IBM SP
Intel Paragon
Cray T3D
Meiko CS−2
PVM/p4

PVM and MPI Future

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