MetaComputing Lecture-2c

Dr Graham E Fagg

MPI InterOperation and MPI-2
Overview

Web based MetaComputing
- Webflow / Gateway

MetaApplications: Inter-operating MPI applications

MPI-2
- Why?
- Dynamic features
- Parallel-IO
Web based

Uses a web client and Java Applets to submit computational requests and then to display the results.

A middle (agent/broker) layer handles the requests and sends it to an MPP/Cluster. The cluster uses MPI/Scalapack/etc to do the computation.
Webflow Design

The front end can be a simple Java Applet, the middle is a Java application that handles the requests in much the same way as a Netsolve agent.

The back end is a high performance (i.e. no Java) MPI application.
Next Generation?

The second generation Webflow is based on a Corba middle layer, which is required to allow strong authentication.

All backend services and some of the middle layer brokers are to be based on Globus components.
Next Generation?

It's called Gateway
Netsolve and Gateway?

Yep, its coming to a Hydra and Cetus Lab near you!
Tier 1
Client Side
Front End

Request & Auth Info

Tier 2
Service / Broker
Level

Netsolve function input
and result data

IIOP

Proxy
for secure client-server coms

Tier 3
HPC Server
Back End
Level

Netsolve Client

list of servers

Netsolve function input
and result data

IIOP

Netsolve function input
and result data

IIOP

Netsolve function input
and result data

Netsolve function input
and result data

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP

Netsolve function input
and result data

IIOP

IIOP

IIOP
Interoperating MPI applications

• Message Passing and MPI-1.1
  – advantages - disadvantages
• MPI-2
• PVMPI-1
  – Global Addressing
• PVMPI-2
  – Transparent communications
  – Examples
  – Internals
    • Message passing layers
    • Data management
  – Cost of transparent operation
  – Process Management
MPI-1 and Message Passing

A specification for safe and efficient message passing
Created by a forum of users, system developers and hardware vendors
Complements a wide number of message passing styles from simple SPMD to complex MIMD
Advantages of MPI-1

It's a standard, not a changing research project or a library for a short lived specialist machine. Here today and tomorrow.

Large number of implementations from both research institutes and vendors allows ease of migration to and from an increasing number of current/new/future architectures.

Choices increase user confidence.

Bottom line.. It will be used even if it has restrictions.
Basics of MPI

**All** characteristics of message passing are contained within *communicators*

Communicators contain:

- Process lists or groups
- Connection/communication structures *topologies*
- System derived message tags/envelopes to separate messages from each other
All processes in an MPI-1 application belong to a global communicator called MPI_COMM_WORLD.

All other communicators are derived from this global communicator.

Communication can only occur within a communicator.

– Safe communication
Before

SEND

SEND

??

RECV

Users Code

3rd Party Library
Under MPI-1
Library writing becomes safe

SEND

RECV

SEND

RECV

3rd Party Library

Users Code
MPI Internals
Processes

All process groups are derived from the membership of the MPI_COMM_WORLD communicator.

- i.e., no external processes.

MPI-1 process membership is **static** not *dynamic* like PVM or LAM 6.5

- simplified consistency reasoning
- fast communication (fixed addressing) even across complex topologies.
- interfaces well to simple run-time systems as found on many MPPs.
MPI-1 Application

MPI_COMM_WORLD

Derived_Communicator
Disadvantages of MPI-1

Static process model

– If a process fails, all communicators it belongs to become invalid. I.e. No fault tolerance.
– Dynamic resources either cause applications to fail due to loss of nodes or make applications inefficient as they cannot take advantage of new nodes by starting/spawning additional processes.
– When using a dedicated MPP MPI implementation you cannot usually use off-machine or even off-partion nodes.
Disadvantages of MPI-1

Multi-language operation is not supported
- MPI specifies both ANSI C and F77 binding
- No agreed standard for data type conversions between languages even upon the same architecture
- Communicators cannot be passed between different language modules due to their representation
  - In F77 a communicator is an INTEGER
  - In C it is usually a pointer to a structure
  - non standard methods exist in different implementations
Problem areas and needed additional features identified in MPI-1 are being addressed by the MPI-2 forum. These include:

- inter-language operation
- dynamic process control / management
- parallel IO
- extended collective operations

Support for inter-implementation communication/control is being considered ??

- Maybe through the port operations.. That’s a maybe.
  - Check out the IMPI forum.. They mean to do it.
PVMPI 1.0 was a dual MPI / PVM API system that allowed separately initialed MPI applications to locate and communicate with each other using PVM as a communications layer.

PVMPI 1.0 provided:

- Global Naming Service for identification of application modules
- (Inter-)Communications layer
- Process control so that PVM spawn could initiate MPI applications instead of forcing users to configure multiple different MPIRUN systems.
PVMP 1.0 address resolution before registration

MPI Application Ocean Model
MPI Application Air Model

MPI_COMM_WORLD

PVM Group Server

proc_id = {comm, rank}
No inter-communication.
PVMP 1.0 address resolution

During registration

pvmpi_register("Ocean Model")

pvmpi_register("Air Model")

MPI_COMM_WORLD

proc_id = {comm, rank} or PVM tid
PVMPi 1.0 address resolution
After registration

Ocean Model

Air Model

proc_id = {comm, rank}, {groupname, rank} or PVM tid

pvm_send(pvm_gettid("Air Model",0),tag)

pvm_send(pvm_gettid("Air Model",3),tag)
Separate MPI-1 applications could intercommunicate

- Across different languages
- different hardware systems
- different MPI implementations

Only requirement was that applications could access a PVM daemon process.

- PVM has been ported to practically all Unix, many MPP systems as well as MS Windows and NT systems
PVMPi 1.0 API problems?

MPI-1 has a large number of functions in its user API

PVMPi 1.0 added its own function calls

Users also had to use PVM calls

Result was function calls in three classes of libraries.

- additional calls were added to reduce the number of nested PVM calls required to send/recv messages such as:
  - pvmpi_send and pvmpi_recv
User requirements for Inter-Operation

Transparent inter-operation
Single style of API.
– i.e. MPI only!
Access to virtual machine when required not mandatory use
Support for complex features across implementations such as derived data types etc.
PVMPi 2.0

New API in the MPI-1 style
Inter-application communication using standard point to point MPI-1 functions calls
– Supporting all variations of send and receive
– All MPI data types including user derived
Naming Service functions similar to the semantics and functionality of current MPI inter-communicator functions
– ease of use for programmers only experienced in MPI and not other message passing systems such as PVM
PVMPRI2 process identification

Process groups are identified by a character name. Individual processes are addressed by a set of tuples.

In MPI:
- \{communicator, rank\}
- \{process group, rank\}

In PVMPRI2:
- \{name, rank\}
- \{name, instance\}

Instance and rank are identical and range from 0..N-1 where N is the number of processes.
Process groups register their name with a global naming service which returns them a system handle used for future operations on this name.

- `int PVMPI_Register (char *name, MPI_Comm local_comm, int * handle)`

Processes can remove their name from the naming service with:

- `int PVMPI_Remove (int handle)`

A process may have multiple names associated with it. Names can be registered, removed and reregistered multiple times without restriction.
An inter-communicator is used for point to point communication between disjoint groups of processes. Inter-communicators are formed using `MPI_Intercomm_create` which operates upon two existing non-overlapping intra-communicator and a bridge communicator.

PVMPI could not use this mechanism as there is not a normal MPI bridge communicator between groups formed from separate MPI applications as their `MPI_COMM_WORLDs` do not overlap.
PVMP12 Inter-communicators

PVMP12 forms its own inter-communicators using the addressing scheme used in PVMP1 1.0.

The bridging communication is performed automatically and the user only has to specify the remote groups registered with `name`.

- `int PVMP12_Intercomm_create (int local_handle, char *remote_group_name, MPI_Comm *new_inter_comm)`
Once an inter-communicator has been formed it can be used almost exactly as any other MPI inter-communicator:

- All point to point operations
- Communicator comparisons and duplication
- Remote group information

Resources released by MPI_Comm_free()
PVMPPI simple example
Air Model

```c
/* air model */
MPI_Init (&argc, &argv);
PVMPPI_Register ("AirModel", MPI_COMM_WORLD, &air_handle);
PVMPPI_Intercomm_create ( handle, "OceanModel", &ocean_comm);
MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
while (!done) {
    /* do work using intra-comms */
    /* swap values with other model */
    MPI_Send( databuf, cnt, MPI_DOUBLE, myrank, tag, ocean_comm);
    MPI_Recv( databuf, cnt, MPI_DOUBLE, myrank, tag, ocean_comm, status);
}
PVMPPI_Remove ( air_handle );
PPI_Comm_free ( &ocean_comm );
PPI_Finalize();
```
/* ocean model */
MPI_Init (&argc, &argv);
MPI_Register ("OceanModel", MPI_COMM_WORLD, &ocean_handle);
MPI_Intercomm_create ( handle, "AirModel", &air_comm);
MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
while (!done) {
  /* do work using intra-comms */
  /* swap values with other model */
  MPI_Recv( databuf, cnt, MPI_DOUBLE, myrank, tag, air_comm, &status);
  MPI_Send( databuf, cnt, MPI_DOUBLE, myrank, tag, air_comm);

  MPI_Remove ( ocean_handle );
  MPI_Comm_free ( &air_comm );
  MPI_Finalize();
PVMPPI Internals

PVM

Uses PVM as a communications layer

Naming services is provided by the PVM Group Server in PVM3.3.x and by the Mailbox system in PVM 3.4.

PVM 3.4 is simpler as it has user controlled contexts and message handlers

MPI application startup is provided by specialist PVM tasker processes.

– These understand LAM, MPICH, MPIF (POE)
PVMPI Internals

MPI

PVMPI is built as an MPI profiling interface. Thus it is transparent to user applications. During building it can be configured to call other profiling interfaces and hence allow inter-operation with other MPI monitoring/tracing tool sets.
MPI Layering

Users Code

MPI function

PVMPI Library

Look up communicators etc

If true MPI intracomm
then use profiled MPI call

PMPI_Function

Else translate into PVM
addressing and use PVM
functions

PVM library

Work out correct return
code
All communicators are mapped to a key value used by an internal hash table that points to the real MPI communicator and/or inter-communicator data structures. The key values are cardinal and thus allows direct translation between the C and FORTRAN MPI bindings in a single application.
PVMPPI Internals
Messaging and Data types

PVM handles point to point messaging and any required data type conversion.

Messages are sequenced, and tagged correctly
– no overtaking due to PVM
– no mixed messages even if both sender and receiver are in multiple groups that concurrently inter-communicate.
– Full range of MPI tags allowed.

Sends do not block*.
Receives (non-blocking etc.) are implemented using complex message handling routines and thus support all the Wait options.

Incorrect data unpacking can be resolved by an auto-unpack option included.
- Improved logging of messaging errors.
PVMP\textsc{i}
Communications

Ready and Sync send/recvs
Style of operation controlled by compile time flags to allow for strictness.
Ready: error, waits or buffered non-blocking
PVMPPI Internals
Performance Issues

• PVMPPI is an additional software layer
  – Intra-communicator performance can be effected.

• Communicator lookup has been stream lined so that intrusion into intra-communication is minimal.

• Inter-communication uses the best options available depending on sender and receiver
  – i.e. pack data as raw bytes instead of XDR
  – psend / precv where possible.
Overheads from PVMP on an MPI implementation have been measured upon the NASA NAS NPB benchmarking suite contained within ParkBench.

Using MPICH on a cluster of Sun Sparc5 machines on a 100Mbit Ethernet, perturbation was not accurately measurable.

– Testing continues on various MPPs.
PVMPi 2.0 Demo

Simple heat flow problem
– Calculates heat flow from one corner of a metal plate suspended by a central cooled member.

Uses two separate MPI applications
– First performs heat flow analysis
– Second performs visualization
  • Visualization may require additional computation and so is performed by a parallel application.
PVMPi 2.0 Demo

Heat flow model computed by a SPMD program using a Jacobi iterative method across a 1D domain decomposition.

Visualization system is a SPMD model across a smaller number of nodes.

– Uses parallelism to accelerate post-processing of computed data into visual data.
– X interface provided by the MPE graphics library from ANL.
PVMPI 2.0

Demo

flow application

Visualization application

{ LAM 6.0 }

PVMPI Inter-communicator

Users Graphical Display

MPICH

CH / ch_shmem

1-D Jacobi solver
The problem is that many of MPIs internal data structures have to be duplicated

– Until a send or receive is posted it is unknown which system will have to deal with these data types.

• Communicators, derived types, message handles etc
• Constants cause problems.. MPI_COMM_WORLD stays the same.
SNIPE is a meta-computing system from UTK that was designed to support long-term distributed applications. Uses SNIPE as a communications layer. Naming services is provided by the RCDS RC_Server system (which is also used by HARNESS for repository information).

MPI application startup is via SNIPE daemons that interface to standard batch/queuing systems

– These understand LAM, MPICH, MPIF (POE), SGI MPI, qsub variations
  • soon condor and LSF
MPI 2?
Do we need PVMPi?

Process Creation and Management in MPI-2

Is PVMPi/MPI_Connect still on target?
– After the standard came out… looks like it.
Other Projects?

PACX
– Parallel Computer eXentions
• From Stuttgart University Computer Centre
• Aimed at interfacing Paragon and Cray T3E machines.
• MPI Application has a single MPI_COMM_WORLD, and cannot do dynamic connect and detach.

PLUS
– formally from PC2 and currently Berlin University, interfaces MPI and PVM applications.
MPI-2

Lots of new features and lots of new interface calls (I.e. almost 400 now!)

– Process Creation and Management
– Single-Sided Communications
– Extended Collective Operations
– C to F77 binding issues
Process management
- Dynamic spawning of tasks
- Both blocking and non-blocking
- Single/Multiple Exe or different ARG lists

- MPI_Comm_spawn (exe, *argv[], #procs, info_keys, root, spawning_comm, new_inter_comm, err_codes[])
More spawning

If spawned your parent inter-comm can be found using: `MPI_Comm_get_parent()`
This is like `ptid = pvm_parent()`

Spawning multiple tasks so that they share a single inter-communicator
– `MPI_COMM_SPAWN_MULTIPLE`
Spawning example
Spawning example

MPI_Comm_spawn()
Spawning example

Inter-communicator
Spawning example

Comm_spawn()
Spawning example
When no common parent

Independent tasks
– Not started by the same parent
– visualization tools
– server-client model

Requires some addition naming service
Names, addresses and ports

Well know address host:port
- MPI_PORT_OPEN makes a port
- MPI_ACCEPT lets client connect
- MPI_CONNECT client side connection

Service naming
- MPI_NAME_PUBLISH (port, info, service)
- MPI_NAME_GET client side to get port
Server-Client model

Server

Client
Server-Client model

Server

MPI_Port_open()

Client
Server-Client model

Server

Client

MPI_Port_open()

{host:port}
Server-Client model

Server

Client

MPI_Port_open()

{host:port}

Magic !

{host:port}
Server-Client model

Server

MPI_Port_open()

{host:port}

Client

MPI_Connect

{host:port}
Server-Client model

Server

Client

MPI_Accept()

Inter-comm

MPI_Connect {host:port}
Server-Client model

Server

Client

MPI_Port_open()

{host:port}

MPI_Name_publish

NAME

MPI_Name_get

{host:port}
Problems

Host:port does it have a meaning outside any implementation?
– I.e. LAM 6.1  nX:port

Who provides the name-address binding services and who can access them?

Inter-comms can be built using sockets
– But what about the data formats uses?
– Data types key maps exported?
MPI IO

Based on Posix file access semantics
– I.e. MPI_FILE_OPEN(), MPI_FILE_CLOSE()
Files are ordered collections of typed data items
Access is per *etype* or element type
– just like a derived datatype for message passing

*filetype* defines how the file is divided between MPI processes.

*View* is the extend of which data is viewable, and can be changed during run-time, unlike *filetype*. 
MPI IO

IO operations can be blocking or non-blocking just like message passing.

- `MPI_FILE_WRITE` and `MPI_FILE_IWRITE`

There are many options for controlling the synchronous updating of single files that are shared, including shared file pointers and buffering options.

- Not for the faint hearted. Lots of options and lots of chances to make big big mistakes.
Doing some work

Two assessments
– First a comparison of different features
– Second a programming exercise
First

Compare Globus, Legion and SNIPE with regard to services offered, naming, communication types and support as well as how users write/develop applications for each.

Comment on the future of using Java as a front end to high performance computing and give an example application (I.e. look it up on the web)

Compare the different methods of inter-operating MPI, in particular MPI_Connect, PACX and IMPI.

– 8-10 pages total, plain ASCII text or HTML with GIF diagrams. To be completed and emailed to me by the morning of the 18th of April 2000.