Developing Applications with Open MPI on an OSCAR-Based Cluster

Thomas Naughton
Stephen L. Scott
Oak Ridge National Laboratory,
USA

Jeffrey M. Squyres
Andrew Lumsdaine
Indiana University, USA
Speakers

- Thomas Naughton, Oak Ridge National Laboratory, USA
  - naughtont@ornl.gov
- Jeffrey M. Squyres, Indiana University, USA
  - jsquyres@open-mpi.org
Tutorial Goals

- OSCAR
  - Brief background
  - Cluster installation overview
  - Helpful tools for administrators and users
- Introduce Open MPI
- Advanced MPI techniques
  - Multi-threading and concurrency
  - MPI-2 dynamic processes
Target Audiences

• System / network administrators
  ▪ Setup and tune MPI for a parallel resource

• MPI users
  ▪ Write and / or run MPI applications
Overview

- OSCAR Introduction
- Open MPI Introduction
- Installing OSCAR and Open MPI
- Threading and MPI
- MPI-2 Dynamic Processes
- Conclusions
Open Source Cluster Application Resources
What is OSCAR?

• Framework for cluster installation, configuration and management
  ▪ Integrates commonly used cluster tools
  ▪ Automatically configures cluster components
  ▪ Wizard based cluster installation
    • Operating system
    • Cluster environment (Administration & Operation)

• Advantages
  ▪ Increase consistency among cluster builds
  ▪ Reduce time to build / install a cluster
  ▪ Reduces need for expertise
OSCAR Background

• Concept first discussed in January 2000
• First organizational meeting in April 2000
  ▪ Cluster assembly is time consuming & repetitive
  ▪ Nice to offer a toolkit to automate
• First public release in April 2001
• Use “best practices” for HPC clusters
  ▪ Leverage wealth of open source components
  ▪ Targeted modest size cluster (single network switch)
• Form umbrella organization to oversee cluster efforts
  ▪ Open Cluster Group (OCG)
Open Cluster Group

- Informal group formed to make cluster computing more practical for HPC research and development

- Membership is open, directed by steering committee
  - Research/Academic
  - Industry

- Current active working groups
  - OSCAR (core group)
  - Thin-OSCAR (Diskless Beowulf)
  - HA-OSCAR (High Availability)
  - SSS-OSCAR (Scalable Systems Software)
  - SSI-OSCAR (Single System Image)
  - BIO-OSCAR (Bioinformatics cluster system)
OSCAR Core Organizations

- Bald Guy Software
- Intel
- RevolutionLinux

- Indiana University
- Oak Ridge National Lab.
- Université de Sherbrooke
- Louisiana Tech Univ.
- Canada’s Michael Smith Genome Sciences Centre

2005-2006
The OSCAR strategy

- OSCAR is a snap-shot of best-known-methods for building, programming and using clusters of a “reasonable” size.
- To bring uniformity to clusters, foster commercial versions of OSCAR, and make clusters more broadly acceptable.
- Consortium of research, academic & industry members cooperating in the spirit of open source.

Open source
OSCAR with
Linux

Commercially supported
value-added instantiations of
OSCAR

Offer variety of flavors: HA-
OSCAR, Thin-OSCAR, SSS-
OSCAR, SSSi-OSCAR
• Red Hat 9.0, Fedora Core 2 support on x86.
• Experimental Mandrake 10.0 support on x86.
• Experimental Red Hat Enterprise Linux (RHEL) 3 on Itanium and x86.
• Fully integrated support for new RPM dependency finder to help build server and clients (Depman/PackMan).
• Ganglia now included in the default package set.
• Torque now included / OpenPBS is now an optional package.
• Enhanced testing framework to use the APITest tool for more thorough post installation testing.
• Multiple bug fixes and Wizard improvements.
• Updated user interface (updated wizard)

state @ May 2005
OSCAR Components

• Administration/Configuration
  ▪ Installers: System Installation Suite (SIS), Cluster Command and Control (C3), OPIUM, Kernel picker
  ▪ Config cluster services: DHCP, NFS, NTP, …
  ▪ Security: Pfilter, OpenSSH

• HPC Services/Tools
  ▪ Parallel: LAM/MPI, MPICH, PVM
  ▪ Batch/Scheduler: Torque, Maui, OpenPBS
  ▪ Development: HDF5
  ▪ Monitoring: Ganglia, Clumon
  ▪ Other 3rd party OSCAR Packages

• Core Infrastructure/Management
  ▪ Management: SIS, C3, Env-Switcher
  ▪ OSCAR tools: OSCAR DAtabase (ODA), OSCAR Package Downloader (OPD)
System Installation Suite (SIS)

Enhanced suite to the SystemImager tool
Adds SystemInstaller and SystemConfigurator

- **SystemInstaller**: interface to installation, includes a stand-alone GUI – TkSis. Allows for description based image creation.
- **SystemImager**: base tool used to construct & distribute machine images.
- **SystemConfigurator**: extension that allows for on-the-fly style configurations once the install reaches the node, e.g., /etc/modules.conf
Switcher

• Switcher provides a clean interface to edit environment without directly tweaking .dot files
  ▪ e.g., PATH, MANPATH (path for mpicc, etc.)

• Edit / set at both system and user level

• Leverages existing Modules system

• Changes are made to future shells
  ▪ To help with “foot injuries” while making shell edits
  ▪ Modules already offers facility for current shell manipulation, but no persistent changes
Switcher Examples

- List all defined tags for the name, *mpi*:
  
  ```
  root# switcher mpi
  lam-7.0.6
  mpich-1.2.5
  ```

- List / change user-level defaults:
  
  ```
  shell$ switcher mpi --show
  default=mpich-1.2.5
  
  shell$ which mpicc
  /opt/mpich-1.2.5/bin/mpicc
  
  shell$ switcher mpi = lam-7.0.6
  ```

- Examine new user-level defaults (i.e., future shells):
  
  ```
  shell$ which mpicc
  /opt/lam-7.0.6/bin/mpicc
  ```

- Remove user-level default:
  
  ```
  shell$ switcher mpi = none
  
  shell$ switcher mpi --rm-attr default
  ```
C3 Power Tools

- Command-line interface for cluster system administration and parallel user tools.

- Parallel execution `cexec`
  - Execute across a single cluster or multiple clusters at same time

- Scatter/gather operations `cpush/cget`
  - Distribute or fetch files for all node(s)/cluster(s)

- Used throughout OSCAR and as underlying mechanism for tools like OPIUM’s `useradd` enhancements.
User & system tools

- **cpush** - push single file -to- directory
- **crm** - delete single file -to- directory
- **cget** - retrieve files from each node
- **ckill** - kill a process on each node
- **cexec** - execute command on each node

  - **cexecs** – serial mode, useful for debugging
C3 Building Blocks (2)

- C3 management tools
  - `clist` – list each cluster available and its type
  - `cname` – returns a node name from a given node position
  - `cnum` – returns a node position from a given node name
- System administration
  - `cpushimage` - “push” image across cluster
  - `cshutdown` - Remote shutdown of cluster
Example to run hostname on all nodes of default cluster:

```shell
shell$ cexec hostname
```

Example to push an RPM to /tmp on the first 3 nodes

```shell
shell$ cpush :1-3 helloworld-1.0.i386.rpm /tmp
```

Example to get a file from node1 and nodes 3-6

```shell
shell$ cget :1,3-6 /tmp/results.dat /tmp
```

* Can leave off the destination with `cget` and will use the same location as source.
Technical Contributors

- Indiana University
- The University of Tennessee
- Los Alamos National Laboratory
- High Performance Computing Center, Stuttgart
- Sandia National Laboratory - Livermore
• Developers of FT-MPI, LA-MPI, LAM/MPI
  ▪ Kept meeting at conferences in 2003
  ▪ Culminated at SC 2003: Let’s start over
  ▪ Open MPI was born

• Started serious design and coding work January 2004
  ▪ All of MPI-2 except one-sided operations
  ▪ Demonstrated at SC 2004
MPI From Scratch: Why?

• Each prior project had different strong points
  ▪ Could not easily combine into one code base

• New concepts could not easily be accommodated in old code bases

• Easier to start over
  ▪ Start with a blank sheet of paper
  ▪ Decades of combined MPI implementation experience
MPI From Scratch: Why?

- Merger of ideas from
  - FT-MPI (U. of Tennessee)
  - LA-MPI (Los Alamos)
  - LAM/MPI (Indiana U.)
  - PACX-MPI (HLRS, U. Stuttgart)

…one MPI to rule them all
Open MPI Project Goals

• All of MPI-2
• Open source
  ▪ Vendor-friendly license (modified BSD)
• Prevent “forking” problem
  ▪ Community / 3rd party involvement
  ▪ Production-quality research platform (targeted)
  ▪ Rapid deployment for new platforms
• Shared development effort
Design Goals

• Extend / enhance previous ideas
  ▪ Message fragmentation / reassembly
  ▪ Design for heterogeneous environments
    • Multiple networks (run-time selection and striping)
    • Node architecture (data type representation)
  ▪ Automatic error detection / retransmission
  ▪ Process fault tolerance
Design Goals

• Design for a changing environment
  ▪ Hardware failure
  ▪ Resource changes
  ▪ Application demand (dynamic processes)

• Portable efficiency on any parallel resource
  ▪ Small cluster
  ▪ “Big iron” hardware
  ▪ “Grid” (everyone a different definition)
  ▪ …
Implementation Goals

- All of MPI-2
- Low latency
  - E.g., minimize memory management traffic
- High bandwidth
  - E.g., stripe messages across multiple networks
- Production quality
- Thread safety and concurrency (MPI_THREAD_MULTIPLE)
Implementation Goals

- Natively support commodity networks
  - TCP
  - Shared memory
  - Myrinet
    - GM, MX
  - Infiniband
    - mVAPI, OpenIB
  - Portals
  - Quadrics Elan4 §
  - LAPI §

- Based on a component architecture
  - Flexible run-time tuning
  - “Plug-ins” for different capabilities (e.g., different networks)

  ...additional slides at end about components

(§ = future)
OSCAR Installation
Server Installation and Configuration

• Install Linux on server machine (cluster head node)
  ▪ Workstation install w/ software development tools
  ▪ 57-page installation document!
    • (quick install available)

• Download copy of OSCAR and unpack on server

• Configure and install OSCAR on server
  ▪ Readies the wizard install process

• Configure server Ethernet adapters
  ▪ Public
  ▪ Private

• Launch OSCAR Install Wizard
Welcome to the OSCAR wizard!

<table>
<thead>
<tr>
<th>Step 0:</th>
<th>Download Additional OSCAR Packages...</th>
<th>Help...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1:</td>
<td>Select OSCAR Packages To Install...</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 2:</td>
<td>Configure Selected OSCAR Packages...</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 3:</td>
<td>Install OSCAR Server Packages</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 4:</td>
<td>Build OSCAR Client Image...</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 5:</td>
<td>Define OSCAR Clients...</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 6:</td>
<td>Setup Networking...</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 7:</td>
<td>Complete Cluster Setup</td>
<td>Help...</td>
</tr>
<tr>
<td>Step 8:</td>
<td>Test Cluster Setup</td>
<td>Help...</td>
</tr>
</tbody>
</table>

Before continuing, network boot all of your nodes. Once they have completed installation, reboot them from the hard drive. Once all the machines and their ethernet adaptors are up, move on to the next step.

The following buttons are for managing your node definitions after the initial install.

| Add OSCAR Clients... | Help... |
| Delete OSCAR Clients... | Help... |
| Install/Uninstall OSCAR Packages... | Help... |

Quit
Step 0

Enables you to download additional packages
OPD – Oscar Package Downloader does download
OPDer – GUI frontend to OPD
clumon and PVFS selected for download
Alternate repositories, possibly a local machine
Step 1

Select OSCAR packages to install.
Core packages are automatically selected for you and cannot be "unselect".

Download does not equal installation!

Packages downloaded with OPDer are selected for installation here.
Configure OSCAR packages that require special configuration tasks
Environment Switcher does configuration for default MPI use

- Default for tag "mpi" can be set to:
  - lam ("lam-7.0.6")
  - mpich ("mpich-ch_p4-gcc-1.2.5.10")
Step 3

Install OSCAR Server (cluster head node) specific packages on cluster head node

May take a few minutes

Wait for Success notice…
Install Server Packages

success

Successfully installed OSCAR server
Step 4

Specify and build system image for client (compute) nodes
Build Image Configuration

- name your image
- list of packages
- package file location
- disk partition file location
- static or dynamic
- enable multicast
- halt, reboot, beep

![Build OSCAR Client Image](image)
Building Image

![Image showing progress](image-url)
Building Image Finished

Successfully created image oscarimage

success
Define client nodes
Define Client Nodes

specify image name (from step 4 – or other saved image)

client IP domain name

client base name (oscarnodeXXX)

node count

starting index to append to base

padding to client names (3 = oscarnode009)

starting IP address

Subnet Mask

Default Gateway
Define Client Nodes

Successfully created clients for image oscarimage

success
Step 6

in one operation – setup networking for all cluster client nodes

for first time in installation process we will “touch” the client nodes
Setup network – Initial Window

machines named as specified in prior step 5

IP address as specified in prior step 5

Scan network for MACs or import from file
Setup network – Scanning Network

found MAC addresses will show here for network setup

stop collecting when done
Setup network – Initial Window

found and assigned all MAC addresses
Reboot Clients

reboot on own – “post install action” from step 4

or

manually reboot
Step 7

only after ALL clients have rebooted

runs “post install” scripts for packages that have them

cleanup and reinitialize where needed
Complete Setup

Successfully completed the cluster install
test suite provided to ensure that key cluster components are functioning properly
Test Cluster Setup

All Passed!!!

* Note on v4.1 there are additional API tests for PVM, which are not shown here.
Your OSCAR cluster is now ready to use.
Add OSCAR Clients

增加计算节点的数量

增加计算节点的数量在外围节点中

增加计算节点的数量向外围节点中
Add OSCAR Clients

Operates in similar manner to steps 5, 6, and 7 in OSCAR installation

Behind the scene action differs somewhat…

compare to standard install process:

step 5
step 6
step 7
Delete OSCAR Clients

decrease the number of compute nodes in the cluster
Delete OSCAR Clients

client selected to delete

In order to delete OSCAR clients from your cluster, select the nodes you wish to delete and press the Delete Clients button.

oscamode1
oscamode2

Delete clients Close
Delete OSCAR Clients

In order to delete OSCAR clients from your cluster, select the nodes you wish to delete and press the Delete Clients button.

oscarnode1

Clients deleted

Close

Delete clients
Install/Uninstall OSCAR Package

select to install or uninstall an OSCAR package
### Install / Uninstall Packages

#### Install/Uninstall OSCAR Packages

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Class</th>
<th>Location/Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoupdate</td>
<td>included</td>
<td>OSCAR 4.8.2-6</td>
</tr>
<tr>
<td>base</td>
<td>core</td>
<td>OSCAR 1-1</td>
</tr>
<tr>
<td>c3</td>
<td>core</td>
<td>OSCAR 4.0.1-1</td>
</tr>
<tr>
<td>clumon</td>
<td>third-party</td>
<td>OPD 1.2.1-2</td>
</tr>
<tr>
<td>Disable Uncommon Client Node Services</td>
<td>included</td>
<td>OSCAR 1.1-1</td>
</tr>
<tr>
<td>Environment Switcher</td>
<td>core</td>
<td>OSCAR 1.0.7-1</td>
</tr>
<tr>
<td>hdf5</td>
<td>included</td>
<td>OSCAR 1.6-2</td>
</tr>
<tr>
<td>kernel_picker</td>
<td>included</td>
<td>OSCAR 1.3-1</td>
</tr>
</tbody>
</table>

**HDF5** is a Hierarchical Data Format product consisting of a data format specification and a supporting library implementation.

[Execute] [Cancel]
Getting Open MPI Software

- First [beta] release “soon”
  - [http://www.open-mpi.org/](http://www.open-mpi.org/)
    - Source code repository will eventually be open
- Available in multiple forms:
  - Source code tarball
  - SRPM
- Future
  - Mac OSX package
  - OSCAR package
• Expand the tarball (on NFS sever)
  shell$ cd /home/build
  shell$ tar zxf openmpi-0.9b1.tar.gz

• Configure the source code
  shell$ cd openmpi-0.9b1
  shell$ ./configure \ 
    --prefix=/opt/openmpi-0.9b1 \ 
    --with-ptl-gm=/san/shared/gm \ 
    --with-ptl-ib=/san/shared/mellanox
Building Open MPI From a Distribution Tarball

- Build the software
  
  ```
  shell$ make all
  (go visit Starbucks)
  ```

- Install to the head node and cluster
  
  ```
  root# make install
  root# cexec make install
  ```
Create New Modulefile

- Copy from existing modulefile
  ```bash
  root# cd /opt/env-switcher/share/mpi
  root# cp lam-7.0.6 openmpi-0.9b1
  ```
- Modify for Open MPI paths, etc.
  (see next slide)
- Distribute this modulefile out to cluster
  ```bash
cpush /opt/env-switcher/share/mpi/openmpi-0.9b1
  ```
#%Module -*- tcl -*-

# Open MPI modulefile for OSCAR clusters

proc ModulesHelp { } {
    puts stderr "\tThis module adds Open MPI to the PATH and MANPATH."
    puts stderr "\tHence, the mpicc, mpiCC, mpif77, and mpirun commands"
    puts stderr "\tthat you run will be from Open MPI."
}

module-whatis "Sets up the Open MPI environment for an OSCAR cluster."

# Don't let any other MPI module be loaded while this one is loaded
conflict mpi

# It's real simple. Append to the PATH and to the MANPATH.
append-path PATH /opt/openmpi-0.9b1/bin
append-path MANPATH /opt/openmpi-0.9b1/man
Modify Default MPI

• What is the default set to now?
  
  root# switcher mpi --show
  default=lam-7.0.6

• Query user defaults in same fashion.
  
  root# switcher mpi --show --user sgrundy
  shell$ switcher mpi --show

• Set user level defaults
  
  shell$ switcher mpi = openmpi-0.9b1
Threads and MPI
Threading

• Multi-threading can improve performance
  ▪ Better CPU utilization
  ▪ IO latency hiding
  ▪ Simplified logic (letting threads block)

• Most useful on SMPs
  ▪ Each thread can have its own CPU

• Overloading CPU’s can be ok
  ▪ Depends on application (e.g., latency hiding)
  ▪ Even on uniprocessors
Threads and MPI

• Extend the threaded model to multi-level parallelism
  ▪ Threads within an MPI process
  ▪ Possibly spanning multiple processors
  ▪ Allowing threads to block in communication

• Two kinds:
  ▪ Application-level threading
  ▪ Implementation-level threading
Application Level Threading

• Freedom to use blocking MPI functions
  ▪ Allow threads to block in MPI_SEND / MPI_RECV
  ▪ Simplify application logic

⇒ Separate and overlap communication and computation
Implementation Threading

• Asynchronous communication progress
  ▪ Allow communication “in the background”
  ▪ Even while no application threads in MPI
• Can help single-threaded user applications
  ▪ Non-blocking communications can progress independent of application
## Asynchronous Progress

<table>
<thead>
<tr>
<th>App</th>
<th>MPI implementation</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>One thread</td>
<td>Multiple threads</td>
</tr>
<tr>
<td>One thread</td>
<td><img src="hourglass.png" alt="Hourglass" /></td>
<td><img src="files.png" alt="Files" /></td>
<td></td>
</tr>
<tr>
<td>Multiple threads</td>
<td><img src="hourglass.png" alt="Hourglass" /></td>
<td><img src="files.png" alt="Files" /></td>
<td><img src="files.png" alt="Files" /></td>
</tr>
</tbody>
</table>
What About “One Big Lock”?

- Put a mutex around MPI calls
  - Only allow one application thread in MPI at any given time
  - Allows a multi-threaded application to use MPI
- Problem: can easily lead to deadlock
  - Example
    - Thread 1 calls MPI_RECV
    - Thread 2 later calls matching MPI_SEND
Why Not Use Non-Blocking?

• Why not use MPI_ISEND? (and friends)
  ▪ This has worked for years
  ▪ MPI implementations already support it
  ▪ Allows at least some degree of overlap

• Threads can allow simplicity of logic
  ▪ Do not have to poll for MPI completion
  ▪ Concurrency within application
  ▪ Let threads block in MPI_SEND / MPI_RECV
Doesn’t MPI Do This Already?

• MPI_SEND: Does it progress after return?
  - Example: in TCP, MPI typically calls write(2)
  - OS buffers and sends “in the background”
  - But does not effect MPI flow control

• If the MPI implementation can use threads:
  - True asynchronous progress
  - Progress pending communications while application is outside of MPI (even flow control)
Threads and MPI

• MPI does not define if a MPI process is a thread or an OS process
  ▪ Threads are not addressable
  ▪ MPI_SEND(…thread_id...) is not possible

• MPI-2 specification
  ▪ Does not mandate thread support
  ▪ Does define “Thread Compliant MPI”
  ▪ Specifies four levels of thread support
Thread Compliant MPI

- All MPI library calls are thread safe
- Blocking calls block the calling thread only and allow progress on other threads
Threads and MPI

- Instead of MPI_INIT:
  - MPI_INIT_THREAD(argc, argv, requested, provided)
    - Tells MPI application threading requirements
    - MPI returns what it can provide
- If MPI cannot support a requested thread level, it returns its highest supported level
MPI Threading Rules

- MPI_INIT_THREAD and MPI_FINALIZE can only be called once
  - Should only be called by a single thread
  - Both should be called by the same thread
  - Known as the main thread
Threads and Requests

- Multiple threads should not attempt to complete the same request
- Erroneous example:

```
Thread1
MPI_Wait (req...)

Thread2
MPI_Wait (req...)
```

Time
Exception handlers can arise in a different thread context than the one making the MPI call.
More Thread Rules

• Undefined behavior of MPI call when:
  ▪ If a thread executes an MPI call that is cancelled by another thread
  ▪ If a thread executes an MPI call and catches a signal

• How to deal with signals?
Avoiding Signal Problems

- Create extra thread that waits in `sigwait()`
- MPI threads mask signals

Diagram:
- User Thread
  - `MPI_Send / Recv / Wait / etc`
  - `sigmask()`
- Extra Thread
  - `sigwait()`
- Time
- OS signals etc
  - Thread catches almost all signals
MPI Levels of Thread Support

- MPI_THREAD_SINGLE
- MPI_THREAD_FUNNELED
- MPI_THREAD_SERIALIZED
- MPI_THREAD_MULTIPLE
MPI_THREAD_SINGLE

- Application is NOT allowed to use threads
  - This allows an MPI implementation to avoid potentially expensive locking *
- Might cause problems / errors if the application actually does use threads
  - So don’t do it!

* Specification is unclear on if the MPI implementation can use threads
• The user application can be multi-threaded but only the main thread calls MPI functions.

Diagram:

- User Thread
  - MPI_Init_thread
  - MPI_Send / Recv / Wait / etc.
  - MPI_Finalize
- Other User threads
  - Threads cannot make MPI calls

Time

MPI Send & Recvs
Users application is multi-threaded any thread can make MPI calls
  - But only one thread can / will be in MPI at a time
MPI_THREAD_SERIALIZED

- Application can be multi-threaded any thread can make MPI calls
  - But only one thread can / will be in MPI at a time
Application can be multi-threaded and any thread can make an MPI call at any time

- Least restricted and most flexible programming model

```
User Thread
MPI_Init_thread
User Thread

MPI_Send(..)

MPI_Recv(..)

MPI_Send(..)

MPI_Recv(..)
```

Time
Threads and MPI

• MPI_QUERY_THREAD
  ▪ Returns provided level of thread support
  ▪ Useful if MPI_INIT was invoked (vs. MPI_INIT_THREAD)
    ➞ Thread level may be set via environment variable!

• MPI_IS_THREAD_MAIN
  ▪ Returns true if this is the thread that invoked MPI_INIT / MPI_INIT_THREAD
Threading Example

• Use a common master / slave framework
  ▪ Master sends out work
  ▪ Workers receive work, do work, return work
  ▪ Loop until complete

• Show how threads can be beneficial in this scenario
Method 1: Pure Master / Slave

- Total of N processes
  - 1 Master process
  - (N-1) Slave processes

- Master
  - Send initial set of work
  - Loop receiving / sending

- Worker
  - Loop: receive, work, send
Pure Master / Slave

Master → Slave 1 → Slave N-1

- Work
- Results
- Finish

Time
Application main()

MPI_Init(...);
MPI_Comm_rank(..., &rank);
if (rank == 0)
    do_master()
else
    do_slave()
MPI_Finalize()
for (i = 0; i < n; ++i)
    MPI_Send(work[i], ..., slaves[i], ...);
while (i < total_work) {
    MPI_Recv(answer, ..., MPI_ANY_SOURCE, ...);
    process_answer(answer);
    if (++i < total_work) {
        MPI_Send(work[i], ..., slave[X], ...);
    } else {
        MPI_Send(you_are_done, ..., slave[X], ...);
    }
}
Slave Main Loop

```c
while (1) {
    MPI_Recv(work, ...);
    if (work == you_are_done)
        break;
    answer = do_work(work);
    MPI_Send(answer, ...);
}
```
• Benefits
  ▪ Easily understood paradigm
  ▪ Robust algorithm

• Drawbacks
  ▪ Master process cannot do any work other than calculating the final result
  ▪ To improve: Master needs to do work and control simultaneously
Method 2: Combined Master / Slave

- Total of N MPI processes
  - N Slave processes
  - Master is combined with Slave 1
- Not wasting a full process for the Master
Combined Master / Slave

- Combined master and slave routines in 1st Slave
  - Send / receive work
  - Do work / calculate answers
- Use non-blocking receives to collect results
  - Use MPI_TEST calls to poll for results
- Master must track state of receives rather than simple outstanding work counter
Combined Master / Slave

- New combined master algorithm
  - Send initial work set
    - Post MPI_Irecv for each item of work sent
  - Loop
    - If work available, do work locally
    - Check for completion of other slaves
    - If completion, send more work or “finish” message
  - End loop when no more work to be done and all slaves finished
Combined Master / Slave

Master / Slave 1

Slave 2

Slave N

Time

Work

Results

Finish
Combined Master / Slave

Overall completion time is shorter, BUT…
Combined Master / Slave

Master / Slave 1

Slave 2

Slave N

 Idle workers awaiting response from master

Results cannot be processed while master is working even when using IRECV / TEST
Summary

• **Benefits**
  - Does not waste a process for the Master

• **Drawbacks**
  - Complicated application code
  - Master does not asynchronously process messages while working
  - Not just simple overlapping of computation and communication
  - Stalls the work pipeline -- idle workers
Method 3: Thread Based Combined Master / Slave

- Use threads
  - Master code in one thread
  - Slave code in another thread
  - Independent progress
- Code now almost identical to Method 1
  - Simplified code / less custom code = less errors
Thread Based Combined Master / Slave

- Total of N MPI processes
  - N Slave processes
  - Master is combined with Slave 1
- Similar concept to Method 2 (one process)
- But similar code to Method 1 (simple code)
Application main()

MPI_Init_thread(...,
    MPI_THREAD_MULTIPLE, ...);
MPI_Comm_rank(..., &rank)
if (rank == 0)
    pthread_create(..., do_master, ...);
do_slave();
pthread_join(...);
MPI_Finalize();
Thread Based
Combined Master / Slave

Master / Slave 1  Slave 2  Slave N

Time

Shortest completion time
Workers not left idle
Threads use blocking MPI_SEND and MPI_RECV

Work  Results  Finish
Summary

• Benefits
  - Simple code -- similar to method 1
  - Overlap communication and computation

• Drawbacks
  - 1st Slave might run somewhat slower than its peers
Dynamic Processes
Dynamic Processes

• Adding processes to a running job
  ▪ As part of the algorithm i.e. branch and bound
  ▪ When additional resources become available
  ▪ Some master-slave codes where the master is started first and asks the environment how many processes it can create

• Joining separately started applications
  ▪ Client-server or peer-to-peer

• Handling faults/failures
MPI-1 Processes

• All process groups are derived from the membership of the MPI_COMM_WORLD
  ▪ No external processes
• Process membership static
  ▪ Simplified consistency reasoning
  ▪ Fast communication (fixed addressing) even across complex topologies
  ▪ Interfaces well to many parallel run-time systems
Static MPI-1 Job

- MPI_COMM_WORLD
- Contains 16 processes
Static MPI-1 Job

- MPI_COMM_WORLD
- Contains 16 processes
- Can only subset the original MPI_COMM_WORLD
  - No external processes
Disadvantages of Static Model

- Cannot add processes
- Cannot remove processes
  - If a process fails or otherwise disappears, all communicators it belongs to become invalid

→ Fault tolerance undefined
Types of Communicators

- **Intra**communicator
  - “Normal” communicator
  - MPI_COMM_WORLD is an intracommunicator
  - One group of processes

- **Inter**communicator
  - Two groups of processes: local and remote
  - Always communicate relative to remote group

- MPI_SEND / MPI_RECV can use both
• MPI_COMM_WORLD and one derived communicator
  ▪ Both are intracommms
• MPI_COMM_WORLD and one derived communicator
  ▪ Both are intracomms
• Create another derived communicator
  ▪ Now have 2 groups
• **MPI_COMM_WORLD** and one derived communicator
  ▪ Both are intracomms
• Create another derived communicator
  ▪ Now have 2 groups
• Create intercomm from the two groups
MPI-2 Process Management

- MPI-2 provides “spawn” functionality
  - Launch a child MPI job from a parent MPI job
- Some MPI implementations support this
  - Open MPI
  - LAM/MPI
  - NEC MPI
  - Sun MPI
  - ...
• **MPI_COMM_SPAWN**
  - Starts a set of new processes with the same command line (SPMD)

• **MPI_COMM_SPAWN_MULTIPLE**
  - Starts a set of new processes with potentially different command lines
  - Different executables and / or different arguments (MPMD)
• Group of parents collectively call spawn
  ▪ Launches a new set of children processes
  ▪ Children processes become an MPI job
  ▪ An \textit{inter}communicator is created between parents and children

• Parents and children can then use the usual MPI functions to pass messages
  ▪ MPI\_SEND / MPI\_RECV
  ▪ etc.
Spawn Example
Spawn Example

Parent processes collectively call MPI_COMM_SPAWN
Spawn Example

Two processes are launched
Spawn Example

Children collectively call MPI_Init
Spawn Example

Children create their own MPI_COMM_WORLD
An intercommunicator is formed between parents and children.
Spawn Example

Intercommunicator is returned from MPI_COMM_SPAWN
Children call MPI_COMM_GET_PARENT to get intercommunicator.
How is This Useful?

- It isn’t… yet (IMNSHO)
  - Can to PVM-style launching
  - “./master” launches its own slaves
  - But mpirun can do MPMD launches with no user code changes -- so why bother?

- More interesting / useful for fault scenarios
  - A node dies
  - Spawn process(es) to replace the dead ones
  - Technology not quite there… yet
MPI “Connected”

- “Two processes are connected if there is a communication path directly or indirectly between them.”
  - E.g., belong to a common communicator
  - SPAWN Parents and children are connected
- Connectivity is transitive
  - If A is connected to B, and B is connected to C
  - A is connected to C
MPI “Connected”

• Why does “connected” matter?
  ▪ MPI_FINALIZE is collective over set of connected processes
  ▪ MPI_ABORT may abort all connected processes

• How to disconnect?
  ▪ …stay tuned
Multi-Stage Spawning

- What about multiple spawns?
  - Can sibling children jobs communicate directly?
  - Or do they have to communicate through a common parent?
- Is all MPI dynamic process communication hierarchical in nature?
Multi-Stage Spawning

Intercommunicator
Multi-Stage Spawning
Multi-Stage Spawning

Do we have to do this?
Multi-Stage Spawning

Or can we do this?
Establishing Communications

- MPI-2 has a TCP socket-style abstraction
  - Process can accept and connect connections from other processes
- Client-server interface
  - MPI_COMM_CONNECT
  - MPI_COMM_ACCEPT
Establishing Communications

- How does the client find the server?
  - With TCP sockets, use IP address and port
  - What to use with MPI?
- Use the MPI name service
  - Server opens an MPI “port”
  - Server assigns a public “name” to that port
  - Client looks up the public name
  - Client gets port from the public name
  - Client connects to the port
Server Side

- Open and close a port
  - MPI_OPEN_PORT(info, port_name)
  - MPI_CLOSE_PORT(port_name)
- Publish the port name
  - MPI_PUBLISH_NAME(service_name, info, port_name)
  - MPI_UNPUBLISH_NAME(service_name, info, port_name)
Server Side

- Accept an incoming connection
  - `MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)`
  - `comm` is a **intra**communicator
  - `newcomm` is an **inter**communicator
Client Side

• Lookup port name
  - MPI_LOOKUP_NAME(service_name, info, port_name)

• Connect to the port
  - MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)
    - comm is a **intra**communicator
    - newcomm is an **inter**communicator
Connect / Accept Example

Port A

Server calls MPI_OPEN_PORT
Server calls MPI_PUBLISH_NAME("ocean", info, port_name)
Connect / Accept Example

Server blocks in MPI_COMM_ACCEPT("Port A", ...)

ocean:Port A
Client calls MPI_LOOKUP_NAME("ocean", ...), gets "Port A"
Client calls MPI_COMM_CONNECT("Port A", …)
Connect / Accept Example

Intercommunicator formed; returned to both sides
Server calls MPI_UNPUBLISH_NAME("ocean", ...)

Connect / Accept Example
Server calls MPI_CLOSE_PORT
Both sides call MPI_COMM_DISCONNECT
How is This Useful?

• Only with MPI_THREAD_MULTIPLE
  ▪ MPI_COMM_ACCEPT blocks!
• Connect to a long-running MPI job
  ▪ Query current status
  ▪ Change direction of the job
• Large scale distributed computing
  ▪ A la Distributed.net, SETI@Home, etc.
  ▪ Secretary’s machine launches cron job at 6pm, MPI_COMM_CONNECTs to server
Summary

- Server opens a port, publishes public "name"
- Client looks up public name, connects
- Server unpublishes name, closes port
- Both sides disconnect

Similar to TCP sockets / DNS lookups

- Useful in a variety of situations
MPI_COMM_JOIN

- A third way to connect MPI processes
  - User provides a socket between two MPI processes
  - MPI creates an intercommunicator between the two processes

➤ Will not be covered in detail here
Collective Operations

• Collective operations are defined on both intra- and inter-communicators
  ▪ Hence, can use collectives on the communicators returned by SPAWN, ACCEPT, CONNECT

• However -- beware!
  ▪ Intra-communicator collectives are “familiar”
  ▪ Inter-communicator collectives are different
  ▪ Read the MPI-2 chapter on “Extended Collectives”
Disconnecting

- Once communication is no longer required
  - MPI_COMM_DISCONNECTION
  - Waits for all pending communication to complete
  - Then formally disconnects groups of processes -- no longer “connected”
- Cannot disconnect MPI_COMM_WORLD
Takeaway Points

• OSCAR
  ▪ Cluster configuration & installation
  ▪ Common tools to manage / use cluster
  ▪ Reduces time and expertise costs

• Advanced MPI techniques
  ▪ Threads and MPI (e.g., blocking in threads)
  ▪ Dynamic processes: spawn, accept / connect
  ▪ Open MPI components / run-time tuning
    (see extra slides)
Takeaway Points

• Open MPI is the culmination of years of research and MPI implementation experience
  ▪ Designed for research and production usage
  ▪ *External collaboration encouraged!*
  ▪ Vendor-friendly license
• First [beta] release “soon”
• Sign up on “announcement” mailing list
Questions?

http://oscar.openclustergroup.org/
http://www.open-mpi.org/
Additional Slides

Not enough time to cover this material during the tutorial
Open MPI Architecture
Traditional MPI Implementations

• Monolithic in nature
  ▪ Large, unwieldy, tightly-integrated code
  ▪ Difficult to maintain

• Practical difficulties for 3rd parties
  ▪ Hard / impossible to learn code base
  ▪ Forking of original code base

➤ This has stifled independent research
A New Approach: Components in MPI

• LAM/MPI introduced first components-based MPI implementation
  ▪ Think “plug-in”, like Netscape
  ▪ “System Services Interface” (SSI)
  ▪ Small, independent components
  ▪ Four different component types
  ▪ Eased implementation / maintenance
  ▪ Allowed 3rd parties to explore and research

• Provided the foundation for this work
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components

- Formalized interfaces
  - Specifies “black box” implementation
  - Different implementations available at run-time
  - Can compose different systems on the fly
Components in HPC

- Components traditionally associated with heavy-weight systems such as:
  - CORBA
  - COM
  - Java beans
- HPC needs much smaller / simpler / faster
- Components therefore only slowly being accepted by the HPC community
Open MPI and Components

- Modular Component Architecture (MCA)
- Logical progression of LAM’s component architecture research
  - More component types
  - More services provided to components
  - Decentralized management
- End result is a “highly pluggable” MPI
Component Benefits

• Stable, production quality environment for 3rd party researchers
  ▪ Can experiment inside the MPI implementation
  ▪ Small learning curve (learn a few components, not the entire implementation)

• Vendors can quickly roll out support for new platforms
  ▪ Write a few components
Open MPI and Components

- Components are shared libraries
  - Central set of components in Open MPI installation tree
  - Users can also have components under $HOME
- Can add / remove components after install
  - No need to recompile / re-link MPI apps
  - Download / install new components
  - Develop new components safely
Example: Cluster Growth

- Sysadmin installs one set of components
- Later adds Infiniband to the cluster
  - Simply add the IB component(s)
- Users unaware of change
  - No need to recompile / re-link MPI apps
  - Apps start seeing IB-level performance
Example: User Components

- 3rd party researchers writing components
  - Too unstable for general usage
  - Cannot be installed at system level
- Solution: developer installs development component under $HOME
  - Open MPI install still finds / uses it at run time
Four-Tier MCA Organization

- **Architecture**
  - Top-tier “glue” and service provider
- **Frameworks** in the architecture
  - Targeted to specific functionality
- **Components** in each framework
  - Implementations of a framework
- **Modules** in each component
  - Components paired with resources
MCA Organization
(not a call stack!)

User application

MPI API

Architecture services

Framework

Component

Modules
Architecture Services

• Top-tier services
  ▪ Find valid components
  ▪ Load found components on demand
  ▪ Unload components when finished
  ▪ Run-time parameter services

• “Glue” that ties the frameworks together
Frameworks

- Divided into three categories
  1. Back-end to MPI API functions
  2. Run-time environment
  3. Infrastructure / management
- Rule of thumb:
  - “If we’ll ever want more than one implementation, make it a framework”
Frameworks

- Dedicated to a single task, such as:
  - MPI collective algorithms
  - MPI point-to-point transfer
  - Starting a process in a run-time environment
- Defines an interface for components and modules
  - Provides framework-specific “glue”
- Defines “scope” for components
  - Many-to-many / many-to-one
[Some] Framework Types

- **MPI types**
  - P2P management
  - P2P transport
  - Collectives
  - Reduction operations §
  - Topologies
  - MPI-2 one-sided §
  - MPI-2 IO
  - Checkpoint / restart §
    (§ = future)

- **Run-time env. Types**
  - Out of band
  - Process control
  - Node list management
  - Global data registry
  - Daemon service
  - Name server

- **Management types**
  - Memory pooling
  - Memory allocation
Components

• Implementation of a framework interface
  ▪ Independent units of software execution

• Examples:
  ▪ TCP point-to-point protocols
  ▪ Infiniband point-to-point protocols
  ▪ Shared memory point-to-point protocols
  ▪ Linear collective algorithms
  ▪ MagPie-based collective algorithms
Modules

• A component paired with resources
  ▪ Analogous to a C++ “instance”

• Examples (in a single process):
  ▪ TCP p2p component with a NIC
  ▪ IB p2p component with a NIC
  ▪ Linear collective algorithms with a communicator
  ▪ MagPIE-based algorithms with a communicator
MCA Parameters

• Companion concept: *parameterize everything*
  ▪ Allow values to be changed at run-time
  ▪ Never use constants in code

• Examples
  ▪ “Short” message size (per network)
  ▪ Number of pre-posted receives
  ▪ Maximum fragment size
  ▪ Which network interfaces to use
Sources of MCA Parameters

1. MPI attributes
2. Command line
   $ mpirun -mca <param> <value>
3. Environment
   $ export OMPI_MCA_<param>=<value>
   % setenv OMPI_MCA_<param> <value>
4. Files (resolved analogous to $PATH)
   <param>=<value>
5. Default value
One Open MPI Installation

- Parameters can be set in multiple places
- Typical scheme:
  - System / network admin tunes performance, sets default MCA values (in a system file)
  - Most users utilize default values
  - Users can selectively override if they want
- This is not just a “feature”
  - Critical infrastructure for flexibility and independent development
3rd Party Components

• Independent development and distribution
  ▪ No need to be part of main Open MPI distribution
  ▪ No need to “fork” Open MPI code base
• Compose to create unique combinations
  ▪ A p2p-based collective component can utilize new ABC network p2p component
• Can distribute as open or closed source
3rd Party Components

CoMPIdea (vendor)

Open MPI distribution

Univ. Southern North Dakota

Open MPI installation on your cluster:
3rd Party Example:
MPI Collective Components

• How to implement new collective algorithms?

• Before components:
  ▪ MPI profiling layer
  ▪ Edit existing MPI implementation
  ▪ Create new MPI implementation (!)
  ▪ Use alternate function names
  ▪ Compiler substitution

• All have benefits / tradeoffs
“coll” Component Framework

- Components as the back-end
  - MPI_BCAST, MPI_BARRIER, MPI_GATHER, etc.
coll Framework Goals

- Intuitive interface
- Maximize performance
- Allow (but not require) layering on MPI point-to-point
- Allow exploitation of back-end hardware
- Allow component layering
- Fine-grained component selection at run-time (per communicator)
- Support both intra- and intercommunicators
1. Layered over point-to-point
   - Use MPI_SEND, MPI_RECV
2. Alternate communication channels
   - Native hardware support for collectives
3. Hierarchical coll components
   - Let one coll component use another
coll Module Lifecycle

- Selection
- Initialization
- Checkpoint restart
- Normal usage
- Finalization

 MPI_INIT
 MPI_COMM_CREATE
 MPI_COMM_DUP
 MPI_COMM_SPLIT

 MPI_ALLGATHER
 ...
 MPI_SCATTERV

 MPI_COMM_FREE
 MPI_COMM_DISCONNECT
 MPI_FINALIZE