Message-Passing Environments & Systems

- Origins of Cluster Computing or Commodity Supercomputing:
  - Limitations of Homogeneous Supercomputing Systems
  - Heterogeneous Computing (HC)
  - Broad Issues in Heterogeneous Computing:
    - Analytical Benchmarking
    - Code-type Profiling
    - HC Task Matching and Scheduling (Mapping)
    - Heterogeneous Computing System Interconnect Requirements
    - Development of Portable Message Passing Environments for HC

- Message-Passing Parallel Systems:
  - Commercial Massively Parallel Processor Systems (MPPs) Vs. Clusters

- Message-Passing Programming & Environments
  - Process Creation In Message Passing
  - Message-Passing Modes
  - Overview of Parallel Virtual Machine (PVM)
    - History
    - Components
    - Setup
    - PVM Function Types
    - PVM Examples
Origins of Cluster Computing:

Limitations of Homogeneous Supercomputing Systems

- Traditional homogeneous supercomputing system architectures usually support a single homogeneous mode of parallelism including: Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and vector processing.
- Such systems perform well when the application contains a single mode of parallelism that matches the mode supported by the system.
- In reality, many supercomputing applications have subtasks with different modes of parallelism.
- When such applications execute on a homogeneous system, the machine spends most of the time executing subtasks for which it is not well suited.
- The outcome is that only a small fraction of the peak performance of the machine is achieved.
- Image understanding is an example application that requires different types of parallelism.
Origins of Cluster Computing:
Heterogeneous Computing (HC)

- Heterogeneous Computing (HC), addresses the issue of computational mode homogeneity in supercomputers by:
  - Effectively utilizing a heterogeneous suite of high-performance autonomous machines that differ in both speed and modes of parallelism supported to optimally meet the demands of large tasks with diverse computational requirements.
  - A network with high-bandwidth low-latency interconnects handles the intercommunication between each of the machines.
  - Heterogeneous computing is often referred to as Heterogeneous supercomputing (HSC) reflecting the fact that the collection of machines used are usually supercomputers.
Motivation For Heterogeneous Computing

• Hypothetical example of the advantage of using a heterogeneous suite of machines, where the heterogeneous suite time includes inter-machine communication overhead. Not drawn to scale.
Heterogeneous Computing Driving Application: Image Understanding

- **Lowest Level (Sensory Processing):**
  - Consists of pixel-based operators and pixel subset operators such as edge detection
  - Highest amount of data parallelism
  - Best suited to mesh connected SIMD machines

- **Intermediate Level (Symbolic Processing):**
  - Grouping and organization of features previously extracted
  - Communication is irregular, parallelism decreases as features are grouped
  - Best suited to medium-grained MIMD machines

- **Highest Level (Knowledge Processing):**
  - Uses the data from the previous levels to infer semantic attributes of an image
  - Requires coarse-grained loosely coupled MIMD machines.
Steps of Application Processing in Heterogeneous Computing

- Analytical Benchmarking:
  - This step provides a measure of how well a given machine is able to perform when running a certain type of code.
  - This is required in HC to determine which types of code should be mapped to which machines.
- Code-type Profiling:
  - Used to determine the type and fraction of processing modes that exist in each program segment.
  - Needed so that an attempt can be made to match each code segment with the most efficient machine.
Code-Type Profiling Example

- Example results from the code-type profiling of a task.
- The task is broken into S segments, each of which contains embedded homogeneous parallelism.
HC Task Matching and Scheduling (Mapping)

- Task matching involves assigning a task to a suitable machine.
- Task scheduling on the assigned machine determines the order of execution of that task.
- Goal of mapping is to maximize performance by assigning code-types to the best suited machine while taking into account the costs of the mapping including computation and communication costs based on information obtained from analytical benchmarking, code-type profiling and possibly system workload.
- The problem of finding optimal mapping has been shown in general to be NP-complete even for homogeneous environments.
- For this reason, the development of heuristic mapping and scheduling techniques that aim to achieve “good” sub-optimal mappings is an active area of research resulting in a large number of heuristic mapping algorithms for HC.
- Two different types of mapping heuristics for HC have been proposed, static or dynamic.
HC Task Matching and Scheduling (Mapping)

Static Mapping Heuristics:

- Most such heuristic algorithms developed for HC are static and assume the ETC (expected time to compute) for every task on every machine to be known from code-type profiling and analytical benchmarking and not change at run time.
- In addition many such heuristics assume independent or meta-tasks that have no data dependencies.
- Even with these assumptions static heuristics have proven to be effective for many HC applications.

Dynamic Mapping Heuristics:

- Mapping is performed on-line taking into account current system workload.
- Research on this type of heuristics for HC is fairly recent and is motivated by utilizing the heterogeneous computing system for real-time applications.
Origins of Cluster Computing:
Heterogeneous Computing System Interconnect Requirements

• In order to realize the performance improvements offered by heterogeneous computing, communication costs must be minimized.
• The interconnection medium must be able to provide high bandwidth (multiple gigabits per second per link) at a very low latency.
• It must also overcome current deficiencies such as the high overheads incurred during context switches, executing high-level protocols on each machine, or managing large amounts of packets.
• While the use of Ethernet-based LANs has become commonplace, these types of network interconnects are not well suited to heterogeneous supercomputers (high latency).
• This requirement of HC led to the development of cost-effective scalable system area networks (SANS) that provide the required high bandwidth, low latency, and low protocol overheads including Myrinet and Dolphin SCI interconnects.

• These system interconnects developed originally for HC, currently form the main interconnects in high performance cluster computing.
Origins of Cluster Computing:

Development of Portable Message Passing Environments for HC

- Since the suite of machines in a heterogeneous computing system are loosely coupled and do not share memory, communication between the cooperating subtasks must be achieved by exchanging messages over the network.

- This requirement led to the development of a number of platform-independent message-passing programming environments that provide the source-code portability across platforms.

- Parallel Virtual Machine (PVM), and Message Passing Interface (MPI) are the most widely-used of these environments.

- This also played a major role in making cluster computing a reality.
Commodity Supercomputing:

Cluster Computing

- The research in heterogeneous supercomputing led to the development of high-speed system area networks and portable message passing environments.
- These developments in conjunction with the impressive performance improvements and low cost of commercial general-purpose microprocessors led to the current trend in high-performance parallel computing of moving away from expensive specialized traditional supercomputing platforms to cluster computing that utilizes cheaper, general purpose systems consisting of loosely coupled commodity off-the-shelf (COTS).
- Such clusters are commonly known as Beowulf clusters and are comprised of three components:
  - **Computing Nodes:** Each low-cost computing node is usually a small Symmetric Multi-Processor (SMP) system that utilizes COTS components including commercial General-Purpose Processors (GPPs) with no custom components.
  - **System Interconnect:** Utilize COTS Ethernet-based or system area interconnects including Myrinet and Dolphin SCI interconnects originally developed for HSC.
  - **System Software and Programming Environments:** Such clusters usually run an open-source royalty-free version of UNIX (Linux being the de-facto standard). The message-passing environments (PVM, MPI), developed originally for heterogeneous supercomputing systems, provide portable message-passing communication primitives.
Message-Passing Parallel Systems: Commercial Massively Parallel Processor Systems (MPPs) Vs. Clusters

Operating system?
MPPs: Proprietary
Clusters: royalty-free (Linux)

Communication Assist:
MPPs: Custom
Clusters: COTS

Distributed Memory

Node: O(10) SMP
MPPs: Custom node
Clusters: COTS node (workstations or PCs)

Custom-designed CPU?
MPPs: Custom or commodity
Clusters: commodity

Parallel Programming:
Between nodes: Message passing using PVM, MPI
In SMP nodes: Multithreading using Pthreads, OpenMP

Scalable Network:
Low latency
High bandwidth
MPPs: Custom
Clusters: COTS

• Gigabit Ethernet
• System Area Networks (SANS)
  • ATM
  • Myrinet
  • SCI

MPPs vs. Clusters
• MPPs: “Big Iron” machines
  – COTS components usually limited to using commercial processors.
  – High system cost
• Clusters: Commodity Supercomputing
  – COTS components used for all system components.
  – Lower cost than MPP solutions
Message-Passing Programming

- Deals with parallel programming by passing messages among processing nodes and processes.
- Several message passing environments have been created in recent years with most of the ideas developed merged into the PVM and MPI standards.
- Message-Passing Interface (MPI):
  - A standard specification for a library of message-passing functions developed by the MPI Forum.
  - Achieves portability using public-domain platform-independent message-passing library.
  - Not self-contained; relies on underlying platform for process management.
- Parallel Virtual Machine (PVM):
  - Originally developed to enable a heterogeneous network of UNIX computers to be used as a large-scale message-passing parallel computer.
  - Using PVM, a virtual machine, a set of fully connected nodes is constructed with dynamic process creation and management. Each node can be a uniprocessor or a parallel computer.
  - PVM provides a portable self-contained software system with library routines to support interprocess communication and other functions.
Process Creation In Message Passing

Possible methods of generating processes:

1. Static process creation
   – In the static process creation, the processes are specified before the program is executed, and the system will execute a fixed number of processes.
   – The programmer usually explicitly identifies the processes.

2. Dynamic process creation.
   – In the dynamic process creation, processes can be created and started for execution during the execution of the main program using process creation constructs or system calls; processes can also be destroyed.
   – Process creation and destruction might be done conditionally.
   – The number of processes may vary during execution.
   – Clearly dynamic process creation (and destruction) is a more powerful technique than static process creation, but it does incur very significant overheads when the processes are created.
Process Creation In Message Passing

Figure 2.1 Spawning a process

Process 1

\texttt{spawn()}

Start execution of process 2

Process 2

Time
Message-Passing Modes

• Synchronous Message Passing:
  Process X executing a synchronous send to process Y has to wait until process Y has executed a synchronous receive from X.

• Blocking Send/Receive:
  A blocking send is executed when a process reaches it without waiting for a corresponding receive. Returns when the message is sent. A blocking receive is executed when a process reaches it and only returns after the message has been received.

• Non-Blocking Send/Receive:
  A non-blocking send is executed when reached by the process without waiting for a corresponding receive. A non-blocking receive is executed when a process reaches it without waiting for a corresponding send.
Generic Message-Passing Routines

- Send and receive message-passing procedure/system calls often have the form:
  
  `send(parameters)`
  
  `recv(parameters)`
  
  - where the parameters identify the source and destination processes, and the data.

*Figure 2.2 Passing a message between processes using send() and recv() system calls*
Blocking `send()` and `recv()` System Calls

(a) When `send()` occurs before `recv()`

(b) When `recv()` occurs before `send()`

Figure 2.3 Blocking `send()` and `recv()` system calls
Non-blocking send() and recv() System Calls

![Diagram of non-blocking send and recv system calls]

*Figure 2.4* Non-blocking send() system call
Parallel Virtual Machine (PVM) History

• PVM is a software environment that permits a heterogeneous collection of serial, parallel, and vector computers on a network to appear as a single large parallel computer.

• The PVM project began in the summer of 1989 at Oak Ridge National Laboratory (ORNL).

• The prototype system, PVM 1.0, was constructed by Vaidy Sunderam and Al Geist as a byproduct of ORNL’s Heterogeneous Distributed Computing research project. This version was only used internally at ORNL.

• Version 2 of PVM was written at the University of Tennessee and was released in March 1991.

• After user feedback and a number of changes (PVM 2.1 - 2.4), a complete rewrite was undertaken, and version 3 was completed in February 1993.

• Version 3.4 released in 1998, adding:
  – Communication Context - safe messages.
  – Message Handlers, for extending features.
  – Persistent messages, tuple space model.
  – User-defined tracing for specialized tools.
  – Windows 9x/NT Port.
Major Features of PVM

- Resource Management:
  - Add/delete hosts from a virtual machine (VM).

- Process Control:
  - Spawn/kill tasks dynamically.

- Message Passing:
  - Nonblocking send, blocking and nonblocking receive.

- Dynamic Task Groups:
  - A task can join or leave a group at any time.

- Fault Tolerance:
  - VM automatically detects faults and adjusts accordingly.
• The PVM system consists of two parts:
  – A PVM daemon (pvmd) that resides on every computer of the virtual machine.
  – A user-callable library (libpvm3.a) linked to user applications for process management, message-passing and virtual machine management.

• PVM Console: Invoking “pvm host_file” creates a PVM console (interactive, stand-alone, PVM process with a pvm> prompt) and starts a single master pvmd daemon on the invoking host and a slave pvmd daemon on every host listed in an optional host_file forming the virtual machine.

• Only the master daemon can add or delete slaves via rsh or rexec( ), and may do so at the request of other processes in the virtual machine.

• Dynamic Virtual Machine Configuration: A user application calling PVM library functions is accomplished using pvm_addhosts and the pvm_delhosts functions sent to the master pvmd.

• Host table: A data structure residing in every host. For each host in the virtual machine a host descriptor in the host table holds its configuration information.
PVM Components

The PVM System is composed of:

- **Pvmd3 daemon program** that:
  - Runs on each host of the virtual machine.
  - Provides inter-host point of contact.
  - Authenticates tasks.
  - Executes processes on the hosts.
  - Provides fault tolerance.
  - Is mainly a message router, but is also a source and sink of messages.

- **libpvm programming library** that is linked with each application component (program)
  - Contains the PVM message passing functions.

- **Application components:**
  - Written by users in PVM message-passing calls.
  - Executed as PVM tasks.
PVM Components

pvmd - One PVM daemon per host

libpvm - Tasks linked to PVM library

tcp

direct connect

host (one per IP address)

OS network interface

pvm's fully connected using UDP

internal interconnect

distributed memory MPP

shared memory multiprocessor
Parallel Virtual Machine (PVM)

- In PVM, the problem is decomposed into separate tasks by the programmer.
- Each program is written in C (or Fortran) containing embedded calls to the PVM library and compiled to run on specific types of computers in the network.
- Process-based computation: The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.
- If the network contains a homogeneous collection of computers (i.e. computers all the same type) the programs only have to be compiled once, for that type of computer.
- In a network of workstations, each workstation has access to the file system containing the compiled programs.
- PVM will automatically allocate computers if the user does not specify the specific computers that are to be used for executing the programs.
Message Passing Between Workstations/Hosts Using PVM

Figure 2.6 Message passing between workstations using pvm
Parallel Virtual Machine (PVM)

- User-configured host pool: The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program.
- The set computers used on a problem first must be defined prior to running the programs, and form the "virtual parallel machine".
- The most convenient way of doing this is by forming a list of the names of the computers that form the virtual machine in a file (a hostfile).
- The hostfile is then read by pvm. An alternative way is to start with one machine and add others manually by using pvm control commands.
Some Terminology Associated with PVM

**Host:**
A physical machine; for example, Unix workstation or parallel computer.

**Virtual machine:**
Combination of hosts running as a single concurrent computational resource.

**Process:**
A program, data stack, etc... For example, a Unix process or a node program.

**Task:**
A PVM process - the smallest unit of computation.

**TID:**
A unique (per virtual machine) identifier associated with each task.

**Message:**
An ordered list of data sent between tasks.
Setup to Use PVM

- One of the reasons for PVM's popularity is that it is simple to set up and use.
- PVM does not require special privileges to be installed. Anyone with a valid login on the hosts can do so. In addition, only one person at an organization needs to get and install PVM for everyone at that organization to use it.
- PVM uses two environment variables when starting and running, PVM_ROOT, and PVM_ARCH.
- Each PVM user needs to set these two variables to use PVM.
- The first variable is PVM_ROOT, which is set to the location of the installed pvm3 directory.
- The second variable is PVM_ARCH, which tells PVM the architecture of this host and thus what executables to pick from the PVM_ROOT directory.
- The easiest method is to set these two variables in your .cshrc file. Here is an example for setting PVM_ROOT:

```bash
setenv PVM_ROOT $HOME/pvm3
```
Setup to Use PVM

- The PVM source comes with directories and makefiles for most architectures.
- Source, examples, man pages are included in the PVM 3.4 distribution which can be obtained from: http://www.epm.orl.gov/pvm3
- Building for each architecture type is done automatically by logging on to a host, going into the PVM_ROOT directory, and typing make.
- The makefile will automatically determine which architecture it is being executed on, create appropriate subdirectories, and build pvm, pvmd3, libpvm3.a, and libfpvm3.a, pvmgs, and libgpvm3.a.
- It places all these files in:
  
  $PVM_ROOT/lib/PVM_ARCH,

  with the exception of pvmgs which is placed in:

  $PVM_ROOT/bin/PVM_ARCH.
# PVM_ARCH Names Used in PVM 3

<table>
<thead>
<tr>
<th>PVM_ARCH</th>
<th>Machine</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFX8</td>
<td>Alliant FX/8</td>
<td></td>
</tr>
<tr>
<td>ALPHA</td>
<td>DEC Alpha</td>
<td>DEC OSF-1</td>
</tr>
<tr>
<td>BAL</td>
<td>Sequent Balance</td>
<td>DYNIX</td>
</tr>
<tr>
<td>BFLY</td>
<td>BBN Butterfly TC2000</td>
<td></td>
</tr>
<tr>
<td>BSD386</td>
<td>80386/486 PC running Unix</td>
<td>BSDI, 386BSD, NetBSD</td>
</tr>
<tr>
<td>CM2</td>
<td>Thinking Machines CM2</td>
<td>Sun front-end</td>
</tr>
<tr>
<td>CM5</td>
<td>Thinking Machines CM5</td>
<td>Uses native messages</td>
</tr>
<tr>
<td>CNVXN</td>
<td>Convex C-series</td>
<td>native f.p.</td>
</tr>
<tr>
<td>CRAY</td>
<td>C-90, YMP, T3D port available</td>
<td>UNICOS</td>
</tr>
<tr>
<td>CRAY2</td>
<td>Cray-2</td>
<td></td>
</tr>
<tr>
<td>CRAYSMP</td>
<td>Cray S-MP</td>
<td></td>
</tr>
<tr>
<td>DGAV</td>
<td>Data General Aviion</td>
<td></td>
</tr>
<tr>
<td>E88K</td>
<td>Encore 88000</td>
<td></td>
</tr>
<tr>
<td>HP300</td>
<td>HP-9000 model 300</td>
<td>HPUX</td>
</tr>
<tr>
<td>HPPA</td>
<td>HP-9000 PA-RISC</td>
<td></td>
</tr>
<tr>
<td>I860</td>
<td>Intel iPSC/860</td>
<td>Uses native messages</td>
</tr>
<tr>
<td>IPSC2</td>
<td>Intel iPSC/2 386 host</td>
<td>SysV, Uses native messages</td>
</tr>
<tr>
<td>KSR1</td>
<td>Kendall Square KSR-1</td>
<td>OSF-1, uses shared memory</td>
</tr>
<tr>
<td>LINUX</td>
<td>80386/486 PC running Unix</td>
<td>LINUX</td>
</tr>
<tr>
<td>MASPAR</td>
<td>Maspar</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>MIPS 4680</td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td>NeXT</td>
<td></td>
</tr>
<tr>
<td>PGON</td>
<td>Intel Paragon</td>
<td>Uses native messages</td>
</tr>
<tr>
<td>PMAX</td>
<td>DECstation 3100, 5100</td>
<td>Ultrix</td>
</tr>
<tr>
<td>RS6K</td>
<td>IBM/RS6000</td>
<td>AIX 3.2</td>
</tr>
<tr>
<td>RT</td>
<td>IBM RT</td>
<td></td>
</tr>
<tr>
<td>SGI</td>
<td>Silicon Graphics IRIS</td>
<td>IRIX 4.x</td>
</tr>
<tr>
<td>SGI5</td>
<td>Silicon Graphics IRIS</td>
<td>IRIX 5.x</td>
</tr>
<tr>
<td>SGIMP</td>
<td>SGI multiprocessor</td>
<td>Uses shared memory</td>
</tr>
<tr>
<td>SUN3</td>
<td>Sun 3</td>
<td>SunOS 4.2</td>
</tr>
<tr>
<td>SUN4</td>
<td>Sun 4, SPARCstation</td>
<td>SunOS 4.2</td>
</tr>
<tr>
<td>SUN4SOL2</td>
<td>Sun 4, SPARCstation</td>
<td>Solaris 2.x</td>
</tr>
<tr>
<td>SUNMP</td>
<td>SPARC multiprocessor</td>
<td>Solaris 2.x, uses shared memory</td>
</tr>
<tr>
<td>SYMM</td>
<td>Sequent Symmetry</td>
<td></td>
</tr>
<tr>
<td>TITN</td>
<td>Stardent Titan</td>
<td></td>
</tr>
<tr>
<td>U370</td>
<td>IBM 370</td>
<td>AIX</td>
</tr>
<tr>
<td>UVAX</td>
<td>DEC MicroVAX</td>
<td></td>
</tr>
</tbody>
</table>
User PVM Directory Tree

- **src directory:** PVM source.c files are stored here, along Makefile.aimk files.
- **bin directory:** Makefile.aimk will compile programs, and store the executable files in sub-directories under this directory according to host architecture.
Starting PVM

Three ways to start PVM:

• `pvm [-nhostname] [hostfile]`
  – PVM console. Command prompt: `pvm>`
  – Starts PVM or just attaches to a running PVM.
  – Recommended way to start and stop PVM 3.

• `xpvm`
  – PVM graphical interface.
  – Includes console functions, and more.
  – Starts PVM or attaches to running PVM.

• `Pvmd [-nhostname] [hostfile]`
  – Direct startup
  – Seldom used except for debugging.
  – Option `-n` useful if host has multiple network cards.
Host File

- While only one person at a site needs to install PVM, but each PVM user can have their own hostfile, which describes his/her own personal virtual machine.
- The hostfile defines the initial configuration of hosts that PVM combines into a virtual machine. It also contains information about hosts that you may wish to add to the configuration later.
- The hostfile in its simplest form is just a list of hostnames one to a line. Blank lines are ignored, and lines that begin with a # are comment lines. This allows you to document the hostfile and also provides a handy way to modify the initial configuration by commenting out various hostnames:

```
# configuration used for my run
cluster
  a
  b
  c
  ...
  n
  o
  p
```

Simplest form just a list of host names

hostfile listing virtual machine configuration
Host File Options

Several options can be specified on each line after the hostname. The options are separated by white space.

lo= userid
allows you to specify an alternative login name for this host; otherwise, your login name on the start-up machine is used.

so=pw
will cause PVM to prompt you for a password on this host. This is useful in the cases where you have a different userid and password on a remote system. PVM uses rsh by default to start up remote pvmd's, but when pw is specified, PVM will use rexec() instead.

dx= location of pvmd
allows you to specify a location other than the default for this host. This is useful if you want to use your own personal copy of pvmd,

ep= paths to user executables
allows you to specify a series of paths to search down to find the requested files to spawn on this host. Multiple paths are separated by a colon. If ep= is not specified, then PVM looks in $HOME/pvm3/bin/PVM_ARCH for the application tasks.

sp= value
specifies the relative computational speed of the host compared with other hosts in the configuration. The range of possible values is 1 to 1000000 with 1000 as the default.

bx= location of debugger
specifies which debugger script to invoke on this host if debugging is requested in the spawn routine. Note: The environment variable PVM_DEBUGGER can also be set. The default debugger is pvm3/lib/debugger.

wd= working directory
specifies a working directory in which all spawned tasks on this host will execute. The default is $HOME.

ip= hostname
specifies an alternate name to resolve to the host IP address.

so=ms
specifies that a slave pvmd will be started manually on this host. This is useful if rsh and rexec network services are disabled but IP connectivity exists. When using this option you will see in the tty of the pvmd
PVM Setup Summary

• Can be installed by any user in their $HOME
• can be installed by root in /usr/local for all users.
• Set PVM_ROOT= full path to pvm3 directory, and PVM_ARCH in your .cshrc file
• Build PVM for each architecture type.
• If installed by root, each user should copy:
  PVM_ROOT/[bin, include, examples] to $HOME/pvm3
• Create a .rhosts file on each host listing all the hosts you wish to use.
• Create a $HOME/.xpvm_hosts file listing all the hosts you wish to use prepended by an `&'".
• Create hostfile to define initial configuration of hosts that PVM can use in the virtual machine.
PVM Console Program

- The PVM console is used to manage the virtual machine to reconfigure it or start and stop tasks in VM.
- In addition, it's an example program that makes use of most of the libpvm functions.
- `pvm_getfds()` and `select()` are used to check for input from the keyboard and messages from the pvmd simultaneously.
- Keyboard input is passed to the command interpreter, while messages contain notification (for example, HostAdd) or output from a task.
- The console can collect output or trace messages from spawned tasks.
## Major PVM Console Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add hostname</code></td>
<td>Add host(s) to virtual machine (can list several)</td>
</tr>
<tr>
<td><code>pvm&gt;add mealy</code></td>
<td></td>
</tr>
<tr>
<td><code>alias</code></td>
<td>Define/list command aliases</td>
</tr>
<tr>
<td><code>conf</code></td>
<td>List hosts in the virtual machine</td>
</tr>
<tr>
<td><code>delete hostname</code></td>
<td>Delete hosts from virtual machine</td>
</tr>
<tr>
<td><code>pvm&gt;delete mealy</code></td>
<td></td>
</tr>
<tr>
<td><code>halt</code></td>
<td>Shut down PVM and tasks</td>
</tr>
<tr>
<td><code>help [command]</code></td>
<td>Print information about commands and options</td>
</tr>
<tr>
<td><code>kill tid</code></td>
<td>Kill a task</td>
</tr>
<tr>
<td><code>ps -a</code></td>
<td>List all running tasks</td>
</tr>
<tr>
<td><code>quit</code></td>
<td>Exit console - leave PVM and tasks running</td>
</tr>
<tr>
<td><code>reset</code></td>
<td>Kill all tasks and reset PVM</td>
</tr>
<tr>
<td><code>sig</code></td>
<td>Send a signal to a task</td>
</tr>
<tr>
<td><code>spawn</code></td>
<td>Spawn task (many options)</td>
</tr>
<tr>
<td><code>pvm&gt;spawn -count 4 app</code></td>
<td></td>
</tr>
<tr>
<td><code>trace</code></td>
<td>Set/display trace events</td>
</tr>
<tr>
<td><code>version</code></td>
<td>Print PVM version</td>
</tr>
</tbody>
</table>
XPVM

Graphical Console and Monitor for PVM

Performs the functions:

• Add/delete hosts, spawn tasks, reset, etc.
• Real-time performance monitor:
  – Display views of network, space-time, utilization, more..
• Call level debugger (and “print viewer”)
  – Display last operation performed by each task
• Post-mortem analysis tool:
  – Can play back and single step through trace file
XPVM Real-time Performance Monitor: Space-Time: Tasks vs. Time Screen Shot
PVM Function Types

- Communication Functions
- Process Control Functions
- Information Functions
- Group Functions
PVM Message-Passing Modes

- All pvm send routines are non-blocking (or asynchronous in pvm terminology) while pvm receive routines can be either blocking (synchronous) or non-blocking.

- If the data being sent is a list of items of a same data type, the pvm call pvm_psend() and pvm_precv() can be used where a parameter in pvm_psend() points to an array of data in the source process to be sent.

- However if the data to be sent is composed of various types, say an integer followed by a float, followed by another integer:
  - The data has to be packed into a PVM send buffer prior to sending the data.
  - The receiving process must unpack its receive message buffer according to the format in which it was packed.
Sending Messages in PVM

The C programs procedure for sending messages in PVM involves three steps:

• A send buffer is initialized by a call to:
  \[
  \text{pvm\_initsend}(\text{int encoding})
  \]

• The message must be packed using the proper function from list on the next slide.

• The completed message is sent to another process by calling:
  \[
  \text{pvm\_send}(\text{int tid, int msgtag})
  \]
  or multi-cast with:
  \[
  \text{pvm\_mcast}(\text{int *tids, int ntask, int msgtag})
  \]

• These three steps may be combined (in PVM 3.3-3.4) in a single call to:
  \[
  \text{pvm\_psend}(\text{int tid, int msgtag, void *vp, int cnt, int type})
  \]
Packing Data Into Send Buffers

- Each of the following C routines packs an array of the given data type into the active send buffer:

  ```c
  pvm_pkbyte( char *cp, int nitem, int stride )
pvm_pkcplx( float *xp, int nitem, int stride )
pvm_pkdcplx( double *zp, int nitem, int stride )
pvm_pkdouble( double *dp, int nitem, int stride )
pvm_pkfloat( float *fp, int nitem, int stride )
pvm_pkint( int *np, int nitem, int stride )
pvm_pklong( long *np, int nitem, int stride )
pvm_pkshort( short *np, int nitem, int stride )
pvm_pkstr( char *cp )
pvm_packf( const char *fmt, ... )
  ```
Receiving Messages in PVM

The C programs procedure for receiving messages involves two steps:

• A message is received using the blocking routine:

\[ \text{pvm\_recv}(\text{int } tid, \text{ int } msgtag) \]

or the non-blocking routine:

\[ \text{pvm\_nrecv}(\text{int } tid, \text{ int } msgtag) \]

• The message is then unpacked using the proper function from list on the next slide.

• These two steps may be combined (in PVM 3.3) in a single call to:

\[ \text{pvm\_precv}(\text{int } tid, \text{ int } msgtag, \text{ void* } vp, \text{ int } cnt, \text{ int type, int } *rtid, \text{ int } *rtag, \text{ int } *rcnt) \]
Unpacking Data From Receive Buffers

- The following C routines unpack (multiple) data types from the active receive buffer:

  ```c
  pvm_upkbyte( char *cp, int nitem, int stride )
pvm_upkcplx( float *xp, int nitem, int stride )
pvm_upkdcplx( double *zp, int nitem, int stride )
pvm_upkdouble( double *dp, int nitem, int stride )
pvm_upkfloat( float *fp, int nitem, int stride )
pvm_upkint( int *np, int nitem, int stride )
pvm_upklong( long *np, int nitem, int stride )
pvm_upkshort( short *np, int nitem, int stride )
pvm_upkstr( char *cp )
pvm_unpackf( const char *fmt, ... )
  ```
If the data being sent is a list of items of a same data type, the pvm call pvm_psend() and pvm_precv() can be used where a parameter in pvm_psend() points to an array of data in the source process to be sent.
If the data to be sent is composed of various types:
- The data has to be packed into a PVM send buffer prior to sending the data.
- The receiving process must unpack its receive message buffer according to the format in which it was packed.
PVM Task Identifier: TID

- PVM uses a 32-bit integer to address any task, group, or pvmd within a virtual machine.
- This integer is called a task identifier (or TID), although it can also refer to a group or a daemon.
- The 32 bits are divided into four fields:
  - A server daemon bit \( S \) - A group bit \( G \)
  - A 12-bit host field \( H \) - An 18-bit local field \( L \)

<table>
<thead>
<tr>
<th>S</th>
<th>G</th>
<th>H</th>
<th>L</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1..4095</td>
<td>1..262143</td>
<td>A task ID</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1..4095</td>
<td>Don’t care</td>
<td>Multicast address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>The local pvmd</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1..4095</td>
<td>0</td>
<td>A pvmd ID</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Small negative number</td>
<td>An error code</td>
<td></td>
</tr>
</tbody>
</table>

PVM Task Identifier: TID

PVM uses a 32-bit integer to address any task, group, or pvmd within a virtual machine.

This integer is called a task identifier (or TID), although it can also refer to a group or a daemon.

The 32 bits are divided into four fields:

- A server daemon bit \( S \)
- A group bit \( G \)
- A 12-bit host field \( H \)
- An 18-bit local field \( L \)

<table>
<thead>
<tr>
<th>S</th>
<th>G</th>
<th>H</th>
<th>L</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1..4095</td>
<td>1..262143</td>
<td>A task ID</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1..4095</td>
<td>Don’t care</td>
<td>Multicast address</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>The local pvmd</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1..4095</td>
<td>0</td>
<td>A pvmd ID</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Small negative number</td>
<td>An error code</td>
<td></td>
</tr>
</tbody>
</table>
## PVM Process Control Functions

<table>
<thead>
<tr>
<th>PVM Function Call</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvm_mytid()</td>
<td>Get the task ID of the calling task.</td>
</tr>
<tr>
<td>pvm_parent()</td>
<td>Get the task ID of the parent task.</td>
</tr>
<tr>
<td>pvm_exit()</td>
<td>The calling task exits PVM.</td>
</tr>
<tr>
<td>pvm_spawn(...)</td>
<td>Spawn a PVM task.</td>
</tr>
<tr>
<td>pvm_kill(tid)</td>
<td>Terminate a PVM task.</td>
</tr>
<tr>
<td>pvm_pstat(tid)</td>
<td>Get the status of a PVM task.</td>
</tr>
<tr>
<td>pvm_tasks(...)</td>
<td>Get the information of all tasks running of VM</td>
</tr>
<tr>
<td>pvm_mstat(host)</td>
<td>Get the status of a host</td>
</tr>
<tr>
<td>pvm_config(...)</td>
<td>Get the configuration information of the entire virtual machine.</td>
</tr>
</tbody>
</table>
Sample PVM
Process Control Functions

pvm_mytid( void )

- The routine pvm_mytid() returns the TID of this process and can be called multiple times.
- It enroll this process into PVM if this is the first PVM call.
- Any PVM system call (not just pvm_mytid) will enroll a task in PVM if the task is not enrolled before the call, but it is common practice to call pvm_mytid first to perform the enrolling.
Sample PVM
Process Control Functions

pvm_exit( void )

• The routine pvm_exit( ) tells the local pvmd that this process is leaving PVM.
• This routine does not kill the process, which can continue to perform tasks just like any other UNIX process.
• Users typically call pvm_exit right before exiting their C programs.
Sample PVM Process Control Functions

pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

• The routine pvm_spawn() starts up ntask copies of an executable file task on the virtual machine.

• argv is a pointer to an array of arguments to the task with the end of the array specified by NULL.

• If the task takes no arguments, then argv is NULL.

• The flag argument is used to specify options, and is a sum of:

<table>
<thead>
<tr>
<th>Value</th>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PvmTaskDefault</td>
<td>PVM chooses where to spawn processes.</td>
</tr>
<tr>
<td>1</td>
<td>PvmTaskHost</td>
<td>where argument is a particular host to spawn on.</td>
</tr>
<tr>
<td>2</td>
<td>PvmTaskArch</td>
<td>where argument is a PVM_ARCH to spawn on.</td>
</tr>
<tr>
<td>4</td>
<td>PvmTaskDebug</td>
<td>starts tasks under a debugger.</td>
</tr>
<tr>
<td>8</td>
<td>PvmTaskTrace</td>
<td>trace data is generated.</td>
</tr>
<tr>
<td>16</td>
<td>PvmMppFront</td>
<td>starts tasks on MPP front-end.</td>
</tr>
<tr>
<td>32</td>
<td>PvmHostCompl</td>
<td>complements host set in where.</td>
</tr>
</tbody>
</table>
Sample PVM Information Functions

pvm_parent( void )

• The routine pvm_parent() returns the TID of the process that spawned this task or the value of PvmNoParent if not created by pvm_spawn().

pvm_tidtohost( int tid )

• The routine pvm_tidtohost() returns the TID dtid of the daemon running on the same host as TID. This routine is useful for determining on which host a given task is running.
pvm_config( int *nhost, int *narch, 
        struct pvmhostinfo **hostp )

- The routine \texttt{pvm_config( )} returns information about the virtual machine including the number of hosts, nhost, and the number of different data formats, narch.
- hostp is a pointer to a user-declared array of pvmhostinfo structures.
- The array should be of size at least nhost.
- On return, each pvmhostinfo structure contains the pvmd TID, host name, name of the architecture, and relative CPU speed for that host in the configuration.
PVM Program Example

• The PVM program hello.c, is a simple example that illustrates the basic concepts of PVM programming.

• This program is intended to be invoked manually; after printing its task id (obtained with pvm_mytid()), it initiates a copy of another program called hello_other.c using the pvm_spawn() function.

• A successful spawn causes the program to execute a blocking receive using pvm_recv.

• After receiving the message, the program prints the message sent by its counterpart, as well its task id; the buffer is extracted from the message using pvm_upkstr.

• The final pvm_exit call dissociates the program from the PVM system.
main()
{
    int cc, tid, msgtag;
    char buf[100];

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

    cc = pvm_spawn("hello_other", (char**)0, 0, ",", 1, &tid);

    if (cc == 1) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
        pvm_upkstr(buf);
        printf("from t%x: %s\n", tid, buf);
    } else
        printf("can't start hello_other\n");

    pvm_exit();
}
PVM program  hello_other.c

```c
#include "pvm3.h"

main()
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);

    pvm_exit();
}
```

- A listing of the ```slave`` or spawned program;
- Its first PVM action is to obtain the task id of the ```master`` using the pvm_parent call.
- This program then obtains its hostname and transmits it to the master using the three-call sequence -
  - pvm_initsend to initialize the send buffer;
  - pvm_pkstr to place a string, in a strongly typed and architecture-independent manner, into the send buffer;
  - and pvm_send to transmit it to the destination process specified by ptid, ```tagging`` the message with the number 1.
PVM 3.4 Examples Included in Package

Example included with PVM 3.4 that illustrate PVM usage and serve as templates for user applications

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello hello_other</td>
<td>PVM equivalent to hello world</td>
</tr>
<tr>
<td>master slave</td>
<td>Master/slave example</td>
</tr>
<tr>
<td>spmd</td>
<td>SPMD example</td>
</tr>
<tr>
<td>gexample</td>
<td>Group and collective ops.</td>
</tr>
<tr>
<td>timing timing_slave</td>
<td>Timing example - comm. perf.</td>
</tr>
<tr>
<td>hitc hitc_slave</td>
<td>Dynamic load balance example</td>
</tr>
<tr>
<td>inheritb</td>
<td>Communication context</td>
</tr>
<tr>
<td>imbi gmbi</td>
<td>Persistent messages template</td>
</tr>
<tr>
<td>mhf_server mhf_tickle</td>
<td>Message handlers example</td>
</tr>
</tbody>
</table>
PVM Group Functions

pvm_joingroup("World")
Calling task joins group World and is assigned an instance number inum.

pvm_lvgroup("World")
The calling task leaves group World.

pvm_gettid("World",inum)
Get task ID from instance number.

ppvm_getinst("World",tid)
Get instance number in a group from TID.

pvm_gsize("World")
Get group size.

pvm_barrier("World",10)
The calling task blocks until 10 members of group World have reached the barrier by pvm_barrier.

pvm_bcast("World",tag)
Broadcast a message identified by tag to all members of World excluding sending task

pvm_reduce(…)
Perform a reduction.

pvm_gather( ...)
gathers data from group members of the into a single array.

pvm_scatter( ...)
Sends a different portion of an array to group members.
Partitioned PVM Program Example

• This program adds $N = 1000$ numbers stored in a file “rand_data.txt”.

• The master “psum” divides the work evenly among $M$ slave processes ($M = 10$ here), running “spsum”.

• Each process adds its portion of the data, and passes the partial sum back to the master.

• It is the master's job to collect all the partial sums, and then compute the overall sum.

• Due to the even distribution of the data, the overall speed of the program is limited by the speed of the slowest processor in the virtual system.

• To avoid this, the work load should be distributed so that the faster processors perform more of the work.
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#define SLAVENAME "spsum"
#define PROC 10
#define NELEM 1000

main()
{
    int mytid;
    int tids[PROC];
    int n = NELEM;
    int nproc = PROC;
    int numt, i, who, msgtype;
    int data[NELEM];
    int result[PROC];
    int tot = 0;
    char fn[255];
    FILE *fp;

    /* Enroll In PVM */
    mytid = pvm_mytid();

    /* Start Slave Tasks */
    numt = pvm_spawn(SLAVENAME, (char**)0, 0, ",", nproc, tids);
    if (numt < nproc)
    {
        printf("Trouble spawning slaves. Aborting. Error codes are: \n");
        for (i=numt; I<nproc; i++)
            printf("TID %d %d\n", i, tids[i]);
        for (i=0; I<numt; i++)
            pvm_kill(tids[i]);
        pvm_exit();
        exit(1);
    }
/* Get Proper Path For Input File */
strcpy(fn, getenv("HOME"));
strcat(fn, "/pvm3/src/rand_data.txt");

/* Open Input File and Initialize Data */
if ((fp = fopen(fn, "r")) == NULL)
{
    printf("Can't open the input file: %s
\n", fn);
    exit(1);
}
for (i = 0; i < 10; i++)
    fscanf(fp, "%d", &data[i]);

/* Broadcast Initial Data To Slaves */
pvm_initsend(PvmDataDefault);
pvm_pkint(&nproc, 1, 1);
pvm_pkint(tids, nproc, 1);
pvm_pkint(&n, 1, 1);
pvm_pkint(data, n, 1);
pvm_mcast(tids, nproc, 0);

/* Wait For Results From The Slaves */
msgtype = 5;
for (i = 0; i < nproc; i++)
{
    pvm_recv(-1, msgtype);
    pvm_upkint(&who, 1, 1);
    pvm_upkint(&result[who], 1, 1);
    printf("I got %d from %d
\n", result[who], who);
}

/* Compute The Global Sum */
for (i = 0; i < nproc; i++)
    tot += result[i];
printf("The total is %d
\n", tot);

/* Program Finished. Exit PVM */
pvm_exit();
return(0);
#include <stdio.h>
#include "pvm3.h"

#define PROC 10
#define NELEM 1000

int add(int me, int n, int *data, int nproc)
{
  int i;
  int sum = 0;
  int low, high;

  low = me *(n/nproc);
  high = low +(n/nproc);
  for(i=low; I<high; i++)
    sum += data[i];

  return(sum);
}

main()
{
  int mytid; /* my task id */
  int tids[PROC]; /* task ids */
  int n; /* number of data elements */
  int me; /* integer id of my task */
  int i, msgtype;
  int nproc; /* number of processes */
  int master; /* integer id of master program */
  int data[NELEM]; /* data array */
  int result; /* my partial sum */
/* Enroll In PVM */
mytid = pvm_mytid();

/* Receive Data From Master */
msgtype = 0;
pvm_recv(-1, msgtype);
pvm_upkint(&nproc, 1, 1);
pvm_upkint(tids, nproc, 1);
pvm_upkint(&n, 1, 1);
pvm_upkint(data, n, 1);

/* Determine Which Slave I Am (0 -- nproc-1) */
for (i=0; I<nproc; i++)
    if (mytid == tids[i]) { me = i; break; }

/* Add My Portion Of The Data */
result = add(me, n, data, nproc);

/* Send Result To The Master */
pvm_initsend(PvmDataDefault);
pvm_pkint(&me, 1, 1);
pvm_pkint(&result, 1, 1);
msgtype = 5;
master = pvm_parent();
pvm_send(master, msgtype);

/* Program Finished. Exit PVM */
pvm_exit();

return(0);
Performance of Divide And Conquer
Addition Example

- The communication time $t_{\text{com}}$ required for the master process to transmit the numbers to slave processes is proportional to $N$.
- The $M$ processes each add $N/M$ numbers together which requires $N/M - 1$ additions.
- Since all $M$ processes are operating together we can consider all the partial sums will be obtained in the $N/M - 1$ steps.
- The master process has to add the $m$ partial sums which requires $M - 1$ steps.
- Hence the parallel computation time, $t_p$, is:
  
  \[ t_{\text{comp}} = \frac{N}{M} - 1 + M - 1 \]

  or a parallel time complexity of:

  \[ t_{\text{comp}} = O(N/M + M) \]

  Hence:

  \[ t_p = t_{\text{com}} + t_{\text{comp}} = N + N/M + M - 2 = O(N + N/M + M) \]

  where the first $N$ term is the communication aspect and the remaining terms are the computation aspects.

- This is worse than the sequential time of $t_s = N - 1$ and complexity of $O(N)$.
Divide And Conquer
Addition Example

\[ t_{\text{com}} = O(N) \]

\[ t_{\text{comp}} \]

\[ t_p = t_{\text{com}} + t_{\text{comp}} = O(N + N/M + M) \]