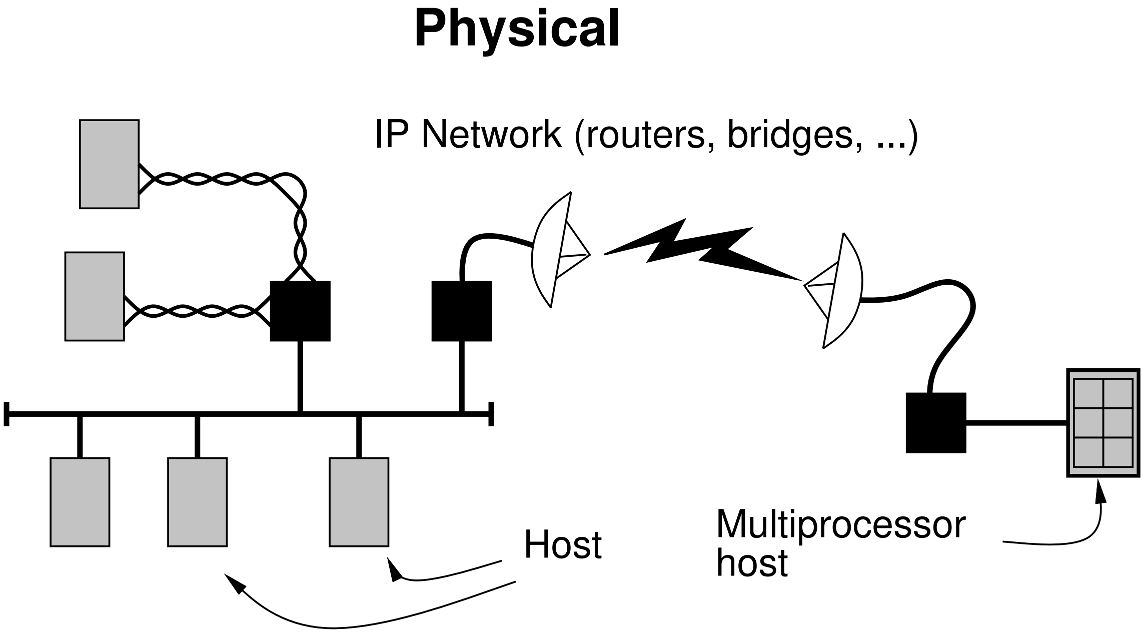
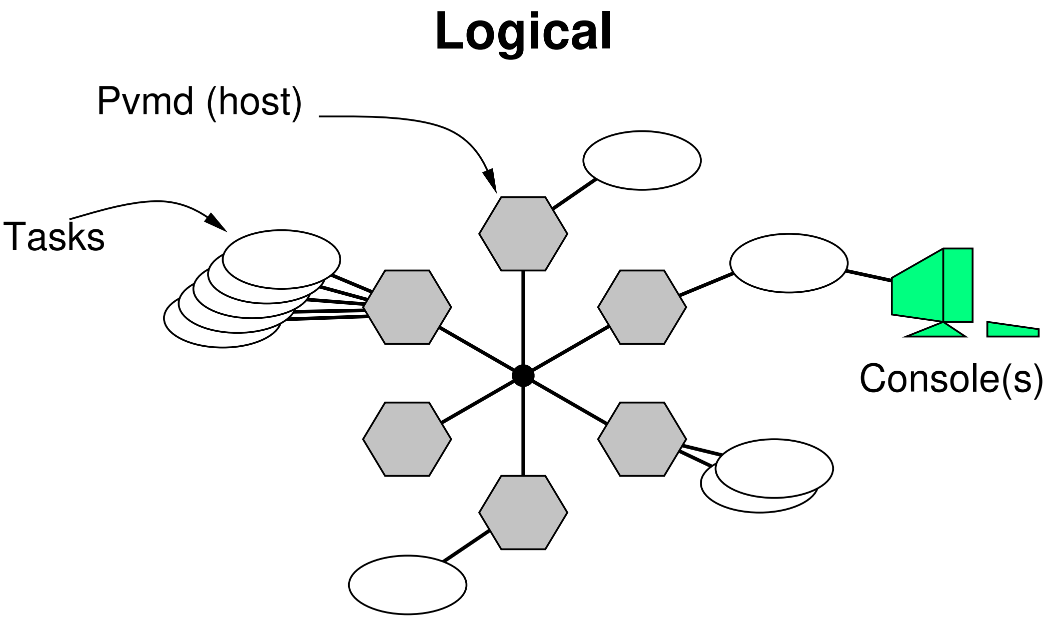
PVM Is

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

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

Physical and Logical Views of PVM





Parts of the PVM System

- PVM daemon (*pvmd*)
  - One manages each host of virtual machine
  - Is a message router
  - Also provides other services
    - Has message *entry points* where tasks request service
    - Provides authentication
    - Creates processes
    - Collects output printed by processes
    - Detects failure of processes, network
- Interface library (*libpvm*)
  - Linked with each application component
    - 1. Functions to compose, send, receive messages
    - 2. PVM *syscalls* that send requests to pvmd
  - Machine-dependent communication part can be replaced
  - Kept as simple as possible
- PVM Console (*pvm*)
  - Kind of like a *shell*
  - Interactive control of tasks, virtual machine
  - Is itself a PVM task
    - Several can be attached
    - To any host

Programming in PVM

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

Hellos World

- Program hello1.c, the main program:

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

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

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

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

- Program hello2.c, the slave program:

```
#include "pvm3.h"

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

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

Unique Features of PVM

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

Portability

- Configurations include

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

- Very portable across Unix machines, usually just pick options
- Multiprocessors:
  - Distributed-memory: T-3D, iPSC/860, Paragon, CM-5, SP2
  - Shared-memory: Convex/HP, SGI, Alpha, Sun, KSR, Symmetry
  - Source code largely shared with generic (80%)
- PVM has been ported to non-Unix machines
  - VMS
  - OS/2
  - Windows 95/NT
- PVM differences are *almost* transparent to programmer
  - Some options may not be supported
  - Program runs in different environment

How to Get PVM

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

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

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

Installing PVM

– Package requires a few MB of disk + a few MB / architecture

– Don't need root privilege

– Libraries and executables can be shared between users

– PVM chooses machine *architecture name* for you
more than 60 currently defined

– Environment variable `PVM_ROOT` points to installed path

  * E.g. `/usr/local/pvm3.3.11` or `$HOME/pvm3`
  * If you use `csh`, add to your `.cshrc`:

```
setenv PVM_ROOT /usr/local/pvm3
```

  * If you use `sh` or `ksh`, add to your `.profile`:

```
PVM_ROOT=/usr/local/pvm3
PVM_DPATH=$PVM_ROOT/lib/pvmd
export PVM_ROOT PVM_DPATH
```

– Important directories below `$PVM_ROOT`

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

Building PVM Package

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

Starting PVM

– Three ways to start PVM

– `pvm [-d`*debugmask*`] [-n`*hostname*`] [`*hostfile*`]`
  PVM console starts pvmd, or connects to one already running

– `xpvm`
  Graphical console, same as above

– `pvmd [-d`*debugmask*`] [-n`*hostname*`] [`*hostfile*`]`
  Manual start, used mainly for debugging or when necessary to enter passwords

– Some common error messages

  * `Can't start pvmd`
    Check `PVM_ROOT` *is set,* `.rhosts` *correct, no garbage in* `.cshrc`
  * `Can't contact local daemon`
    PVM crashed previously; socket file left over
  * `Version mismatch`
    Mixed versions of PVM installed or stale executables
  * `No such host`
    Can't resolve IP address
  * `Duplicate host`
    Host already in virtual machine or shared `/tmp` *directory*
  * `failed to start group server`
    Group option not built or `ep=` *not correct*
  * `shmget: ... No space left on device`
    Stale segments left from crash or not enough are configured

Special files used by PVM

- Pvmd socket address file
  * Pvmd publishes its socket address for local tasks in a file: `/tmp/pvmd.`*uid*
  * Existence also indicates that a pvmd is running
  * Another won't start (unless PVM built with `OVERLOADHOST`)
  * If pvmd killed by uncatchable signal or catastrophe, file must be removed by hand
- Pvmd error log file
  * Pvmd logs errors and warnings to
    · `stderr` (only until fully up and running)
    · `/tmp/pvml.`*uid*
  * Always check log file when something goes wrong
  * If two hosts involved, check both logfiles
  * Unclaimed output from applications also goes here
- To get your *uid*

```
(grep `whoami` /etc/passwd || ypmatch `whoami` passwd) \
  | awk -F: '{print $3;exit}'
```

- If PVM is built with option `SHAREDTMP` these are named
  * `/tmp/pvmd.`*uid.hostname*
  * `/tmp/pvml.`*uid.hostname*

Creating a Host File

– A host file lists machine names for PVM use

– File can be named anything you want

– In the simplest form, contains just a list of names

```
thud
ig
rudolph
```

– Options can be appended to each line

`lo=`*name*	Different login name
`so=pw`	Prompt for password
`dx=`*file*	Pvmd executable
`ep=`*path*	Task executable search path
`wd=`*dir*	Task working directory
`bx=`*file*	Debugger program
`ip=`*name*	Alternate IP address
`so=ms`	User performs manual startup
`sp=`*speed*	Relative speed of CPU

– Special characters at beginning of line:

`#`	Comment line
`*`	Change default settings
`&`	Set options only; don't start

PVM Console

– PVM console is a special *task* for managing the machine

– Some important commands are

`conf`	List virtual machine configuration
`add`	Add hosts to virtual machine
`delete`	Delete hosts from virtual machine
`halt`	Stop all tasks and shutdown pvmds
`spawn`	Spawn tasks
`ps`	List running tasks
`kill`	Terminate tasks
`reset`	Kill all tasks and clean up
`trace`	Set or display trace mask
`jobs`	Display list of running jobs
`help`	Get more information

– Commands to execute at startup can be put in `$HOME/.pvmrc`

```
alias ? help
alias j jobs
setenv PVM_EXPORT DISPLAY
# print my id
echo new pvm shell
id
```

XPVM

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

Programming Interface

About 80 functions

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

Process Control

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

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

  - Other flags

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

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

Basic PVM Communication

– Three-step send method

  * `pvm_initsend(encoding)`
    Initialize send buffer, clearing current one
    Encoding can be `PvmDataDefault` / `PvmDataRaw` / `PvmDataInPlace`
  * `pvm_pk`*type*`(data, num_items, stride)`
    ...
    Pack buffer with various data
  * `pvm_send(dest, tag)`
    `pvm_mcast(dests, count, tag)`
    Sends buffer to other task(s), returns when safe to clear buffer

– To receive

  * `pvm_recv(source, tag)`
    `pvm_nrecv(source, tag)`
    Blocking or non-blocking receive
  * `pvm_upk`*type*`(data, num_items, stride)`
    Unpack message into user variables

– Can also `pvm_probe(source, tag)` for a message

– Another receive primitive: `pvm_trecv(source, tag, timeout)`
  Equivalent to pvm_nrecv if timeout set to zero
  Equivalent to pvm_recv if timeout set to null

Higher Performance Communication

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

Collective Communication

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

Virtual Machine Control

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

PVM Examples in Distribution

– Examples illustrate usage and serve as templates

– Examples include

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

– Examples come with Makefile.aimk files

– Both C and Fortrans versions for some examples

Architectures and Data Encodings

- *Architecture name* assigned to each "machine type"
  - Traditionally a short string: Uppercase + digits, about 50 defined
  - Same name indicates compatible `a.out` format
    - Similar hardware, operating system
    - For some meaning of *similar*
  - Choose binaries for execution
  - Allow parallel compilation for different architectures
  - Built into `lib/pvmgetarch`
- *Data encoding* assigned to machine type
  - Number indicates data format in messages

Encoding	Data format
0	XDR (`PvmDataDefault`)
1	`PvmDataFoo`
	Below are for `PvmDataRaw`, `PvmDataInPlace`
11	Integers of 2, 4, 4 bytes, Floats of 4, 8 bytes, IEEE floating point format, Big-endian byte order
12	Like 11, but little-endian
13..22	*other formats*

  - Receiver knows how to decode data, whether possible
  - Can decode its own raw format, XDR and foo
  - Pvmd, libpvm contain look-up tables

Compiling Applications

– Header files

∗ C programs should include

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

∗ Specify include directory: `cc -I$(PVM_ROOT)/include ...`

∗ Fortran: `INCLUDE '/usr/local/pvm3/include/fpvm3.h'`

– Compiling and linking

∗ C programs must be linked with

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

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

Parallelization – Getting Started

– Starting with a large serial application
  * Look at the Physics –
    Is problem inherently parallel?
  * Examine loop structures –
    Are any independent? Moderately so?
    Tools like Forge90 can be helpful
  * Look for the core linear algebra routines –
    Replace with parallelized versions

– Already been done. (check survey)

Popular Distributed Programming Schemes

- Master / Slave

  Master task starts all slave tasks and coordinates their work and I/O

- SPMD (hostless)

  Same program executes on different pieces of the problem

- Functional

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

Parallel Programming Considerations

- Granularity of tasks

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

- Number of messages

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

- Functional vs. Data parallelism

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

Network Programming Considerations

- Message latency

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

- Different Machine Powers

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

- Fluctuating machine and network loads

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

Load Balancing Methods

– Static load balancing

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

– Dynamic load balancing by pool of tasks

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

– Dynamic load balancing by coordination

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

Packing Messages

– Ways to pack:

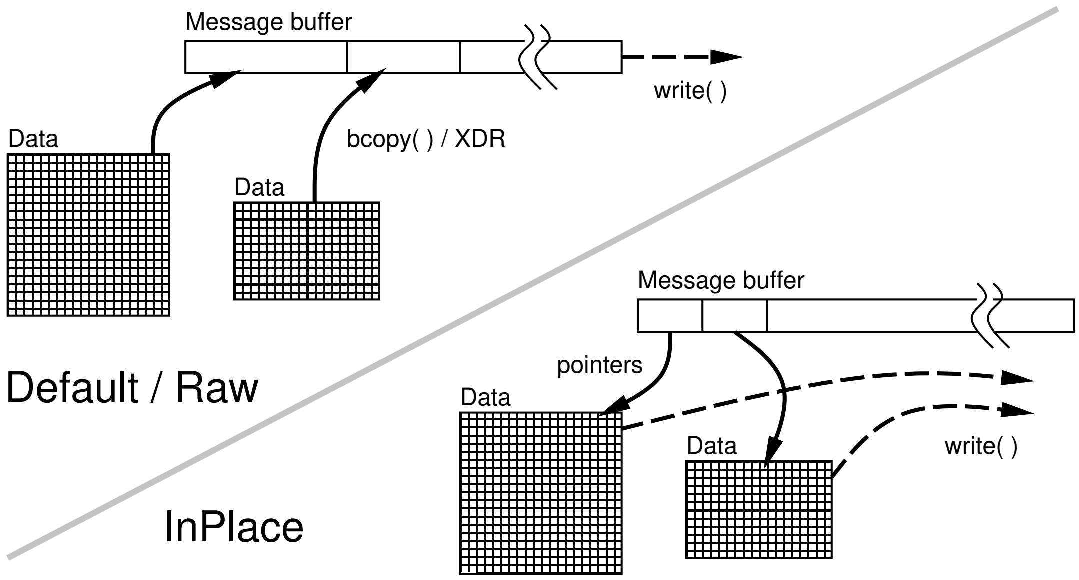


Figure 1: Packing options

– Speed and Memory Tradeoffs

Encoding	Packing Speed	Mix Arch	Buffer Memory Cost			
			Socket/MPP		Shared Mem	
			Sndr	Rcvr	Sndr	Rcvr
`PvmDataDefault`	slowest	Y	$\sim 1x$	$1x$	$1x$	0
`PvmDataRaw`	medium	N	$1x$	$1x$	$1x$	0
`PvmDataInPlace`	fastest	N	~ 0	$1x$	$1x$	0

– `PvmDataInPlace` side effects

- Allowed for dense data only
- Implies raw
- Data evaluated at send time, not pack time

Packing Messages, cont'd

– No one encoding is best

– Make it a parameter:

```
    static int encoding = PvmDataDefault;
...
    if (p = getenv("ENCODING")) {
        if (!strcmp(p, "PvmDataRaw"))
            encoding = PvmDataRaw;
        if (!strcmp(p, "PvmDataInPlace"))
            encoding = PvmDataInPlace;
    }
```

– Or chosen by host configuration:

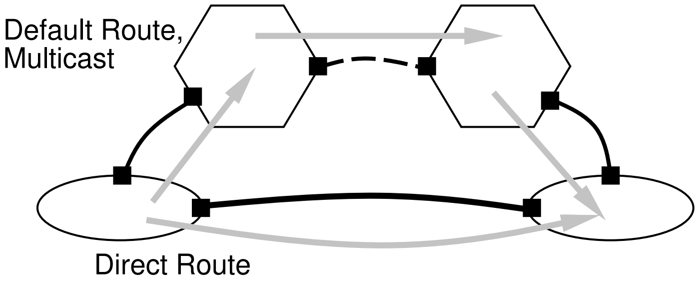
```
    static int encoding = PvmDataDefault;
...
    int narch;

    pvm_config((int *)0, &narch, (struct pvmhostinfo *)0);
    if (narch == 1)
        encoding = PvmDataInPlace;
```

– Or chosen by destination:

```
    if (my_archcode == rcvr_archcode)
        encoding = PvmDataRaw;
    else
        encoding = PvmDataDefault;
```

Sending Messages

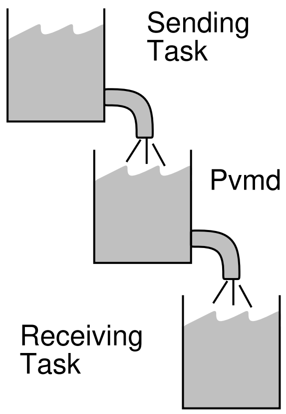


- `pvm_send` with `PvmRouteDefault`
  * Always works
- `pvm_send` with `PvmRouteDirect`
  * Better performance/reliability: No pvmd involved
  * Not available in Shared memory, some multiprocessor ports
- `pvm_mcast`
  * Default route; performance depends on fanout
- `pvm_bcast`
  * Implemented by group lookup followed by mcast
  * Special case on T-3D: Null group
- `pvm_psend/precv`
  * Special multiprocessor implementations possible
  * Good for dense, raw, single-typed data
  * Uses pack(InPlace) then send in unix port
  * Can't be mixed with regular probe/receive functions

Communication Tips

– Limit size, number of outstanding messages

  * Can load imbalance cause too many outstanding messages?
  * May have to send very large data in parts



– Complex communication patterns

  * Network is deadlock-free, designed not to hang
  * Still have to consider
    · Correct data distribution
    · Bottlenecks
  * Consider using a library
    · ScaLAPACK: LAPACK for distributed-memory machines
    · BLACS: Linear algebra communication primitives

Load Balancing

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

Bag of Tasks

- Components
  - Job pool
  - Worker pool
  - Scheduler

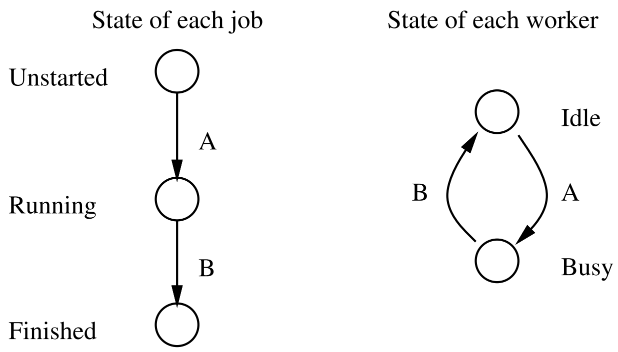
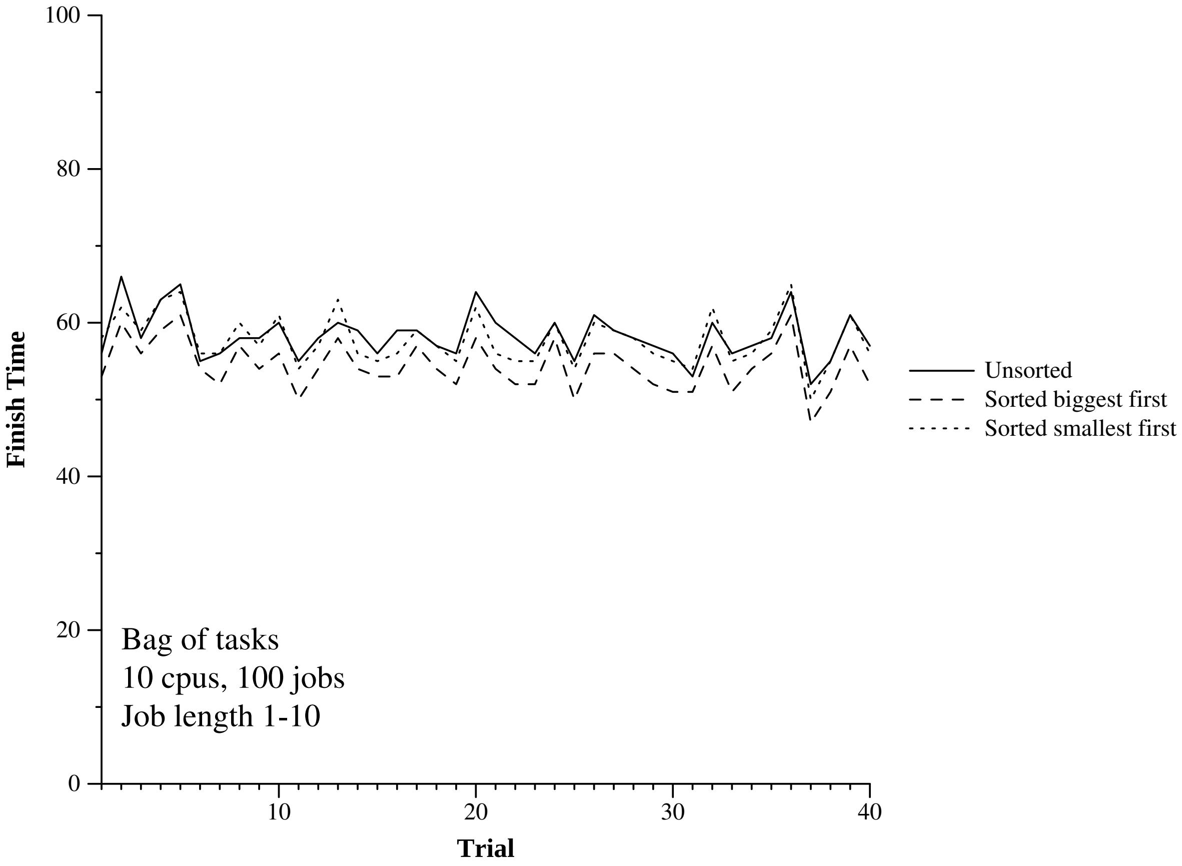


Figure 2: Bag of tasks state machines

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

Bag of tasks simulation



Bag of Tasks (FT)

- If a worker fails, master waits forever
- Can detect this and reschedule job

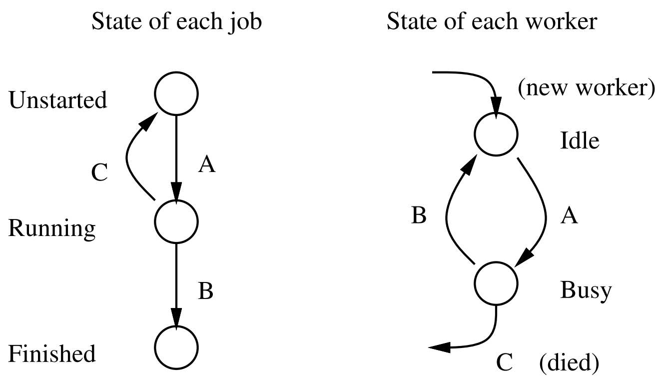


Figure 3: B.O.T. state machines with fault tolerance

- Worker tasks, hosts can go down
- Master still essential (PVM host, scheduler task)
- Use either
  - `pvm_notify`
  - `pvm_trecv` + `pvm_pstat`

Notification

- Pvmds detect loss of host or network, and adapt
- Task can be sent a message when an exceptional event occurs
- Task can use information to
  - Recover application from loss of task or host
  - Be aware of new resources to utilize
- Task to be notified calls

  `pvm_notify(int what, int tag, int count, int *tids)`
- Task specifies message tag to be used for replies
- Message body contains data specific to the event

What	Reply Message Contains
`PvmTaskExit`	{ `int task_id` }
`PvmHostDelete`	{ `int pvmd_tid` }
`PvmHostAdd`	{ `int numhosts,` `int pvmd_tids[numhosts]` }

- Number of reply messages is equal to `count`
- For `PvmHostAdd`
  - `count` is usually -1, 0 or 1
  - `tids` is ignored
- Notifies (except `PvmHostAdd`) can't be cancelled
- Multiple notifies for same (task, host) send multiple replies
- TaskExit and HostDelete reply immediately if event already happend

Notification, Examples

```
int tids[2], x, i;

pvm_spawn("/bin/true", (char **)0, 0, "", 2, tids);
printf("spawned t%x and t%x\n", tids[0], tids[1]);
pvm_notify(PvmTaskExit, 666, 2, tids);
for (i = 0; i < 2; i++) {
    pvm_recv(-1, 666);
    pvm_unpackf("%d", &x);
    printf("t%x exited\n", x);
}
```

```
int n, *tids, nhosts, i, j;
struct pvmhostinfo *hostlist;

pvm_notify(PvmHostAdd, 13, -1, (int *)0);
...
pvm_recv(-1, 13);
pvm_unpackf("%d", &n);
tids = (int *)malloc(n * sizeof(int));
pvm_unpackf("%*d", n, tids);
printf("%d new host%s added\n", n, (n == 1 ? "" : "s"));
if (!pvm_config(&nhosts, (int *)0, &hostlist)) {
    fputs("                    HOST     ARCH\n", stdout);
    for (j = 0; j < n; j++)
        for (i = nhosts; i-- > 0; )
            if (tids[j] == hostlist[i].hi_tid) {
                printf("%24s %8s\n",
                        hostlist[i].hi_name,
                        hostlist[i].hi_arch);
                break;
            }
}
free(tids);
```

Signals

- Task can send signal to another PVM task with `pvm_sendsig`
- A normal unix signal is sent
- Use to kill task (we recommend using `pvm_kill`)
- Use to interrupt computation
  - Load balancing
  - Interrupt search
- But, be warned that
  - Order is not preserved between signals and messages
  - Have to be careful in handler function
    - PVM not reentrant
    - Libc may not be reentrant
  - Signal numbers are not mapped between different unixes
  - Signals aren't available on some architectures
- What is safe?
  - Setting a flag to indicate signal has arrived
  - Pushing data into e.g. a FIFO or a pipe
  - "Section 2" libc functions

Example: Signaling a worker

```
#include <pvm3.h>
#include <signal.h>

static int f = 0;

void handler(int sig) { f = 1; }

main()
{
   int tid;

   pvm_mytid();
   signal(SIGUSR1, handler);
   while (1) {
getwork:
      f = 0;
      pvm_recv(-1, -1);                            /* receive work */
      pvm_bufinfo(pvm_getrbuf(),
            (int *)0, (int *)0, &tid);

      for (... ; ... ; ...) {                      /* computation */
         if (f) {
            pvm_packf("%+", PvmDataDefault);       /* empty reply */
            pvm_send(tid, 1);
            goto getwork;
         }
         ...
      }

      pvm_packf("%+ ...", PvmDataDefault, ...); /* send result */
      pvm_send(tid, 1);
   }
   exit(pvm_exit());
}
```

Reentrancy in Libpvm

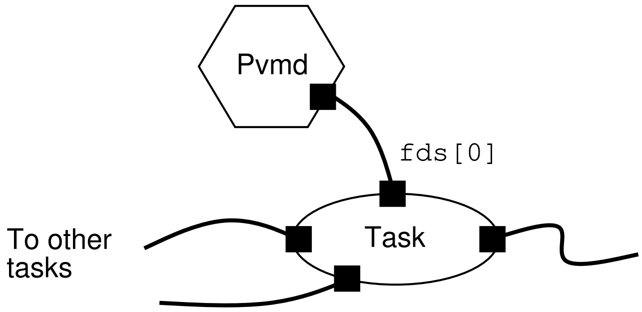
- Libpvm isn't signal or thread safe
  - Signals aren't blocked in critical sections
  - Functions aren't reentrant
  - Data structures, protocol state will get botched
- Would be nice to be have
  - Interrupts (active messages)
  - Separate threads
    - Interrupts
    - Graphics display, ...
    - Improve PVM performance
- Could try to make signals safe
  - Block signals in PVM functions
  - Unblock in `mxfer`
  - Queue messages sent in signal handler
  - Problems with this are
    - Cost to block, unblock signals (`pk_`*xxx*)
    - Libc still may not be reentrant
    - Some machines don't have signals

Reentrancy in Libpvm, WIP

- Could make threads work
  - Communication thread started at beginning of process
  - Used to move data in background
  - New threads invoked to handle messages
- Plan changes to code, PVM programming interface
  - Eliminate state where possible
    - Easy – `pvm_errno`
    - Hard – current trace message
  - Add parameters to functions
    - Message handles
    - More spawn parameters
  - Locks on data structures
- Evaluate thread packages
  - Don't want to be tied to non-portable package
  - Maybe can remain independent?

Multiplexed I/O

- Pvm tasks are usually either computing or blocked in receive
- May need to service input from other sources
  - Xep program wants input from PVM or X-Windows
  - Console checks for input from stdin (keyboard) or PVM
- If program waits in `pvm_recv`, other input will starve
- If program waits on other input, PVM messages will starve
- Can poll both sources, using `pvm_trecv`
  - Wastes some cpu
  - Simple, may be the right choice
- Can wait for messages on PVM sockets
  - `int nfds = pvm_getfds(int **fds)`
  - Returns FDs of sockets used by PVM
  - Use with `select()`



Multiplexed I/O Example

```
#include <sys/types.h>
#include <pvm3.h>

main_loop()
{
   int *d;
   fd_set fds, rfds;
   int n;
   char buf[256];

   pvm_mytid();
   pvm_getfds(&d);
   FD_ZERO(&rfds);
   FD_SET(0, &rfds);
   FD_SET(d[0], &rfds);

   while (1) {
      fds = rfds;
      if (select(d[0] + 1, &fds, (fd_set*)0, (fd_set*)0,
               (struct timeval*)0) > 0) {

         /* check for keyboard input */
         if (FD_ISSET(0, &fds)) {
            if ((n = read(0, buf, sizeof(buf))) > 0)
               do_command(buf, n);
         }

         /* check for PVM messages */
         if (FD_ISSET(d[0], &fds))
            while (pvm_nrecv(-1, -1) > 0)
               got_a_message();
      }
   }
}
```

Multiplexed I/O, Cont'd

- Getfds is
  - Simple to use with default message routing
  - Messy with direct routing (set of FDs is dynamic)
  - Only available on Unix systems (and not shared-memory)
- Remember
  - `fds[0]` is always the pvmd
  - Socket ready doesn't mean message received
    - Maybe just a fragment
    - A control message
  - There may be more than one message
- What needs to be done
  - Easier way to cope with dynamic routes
    (`notify(PvmRouteChange)`)
  - Done another way?
    (receive message when file descriptor ready)

Hiding PVM from Users

- Normally, virtual machine is started before application
- Application might want to
  - Start PVM
  - Add hosts to configuration (or start with host file)
  - Do work
  - Shut down PVM
- Can use `pvm_start_pvmd` and `pvm_halt`
- Application users don't have to know about PVM
- Must set `PVM_ROOT` (could do that automatically)
- Need machine configuration (could use PVM hostfile)
- May need to allow overlapping virtual machines
  - Build PVM with `OVERLOADHOST` option
  - Allows > 1 pvmd per user per host
  - Fix to this in V 3.3.8 or 3.4 (sockaddr)

How to fork a PVM task

- Can't just call `fork` to get two PVM tasks
- A `pvm_fork` function in the same spirit:

```
int
pvm_fork(int *cpid)    /* cpid returns child pid if nonnull */
{
   int tid;            /* tid to return (child tid or 0) */
   int pid;
   int sbf = pvm_setsbuf(0);
   int rbf = pvm_setrbuf(0);

   if ((tid = pvm_mytid()) > 0) {
      if (pid = fork()) {                /* parent */
         if (cpid)
            *cpid = pid;
         if (pid == -1)
            tid = PvmOutOfRes;
         else
            if ((tid = pvm_recv(-1, 1)) > 0)
                pvm_upkint(&tid, 1, 1);
      } else {                           /* child */
         pvmendtask();
         pvm_packf("%+ %d", PvmDataRaw, pvm_mytid());
         pvm_send(tid, 1);
         tid = 0;
      }
   }
   if (sbf = pvm_setsbuf(sbf))
       pvm_freebuf(sbf);
   if (rbf = pvm_setrbuf(rbf))
       pvm_freebuf(rbf);
   return tid;
}
```

Task Environment

- Comes from task's pvmd *and* parent task
- Working directory comes from pvmd
  - Use host file `wd=` to set it
  - Can override by passing directory from master task, calling `chdir` in slaves
- Environment variables come from pvmd
  - Unless overridden using `PVM_EXPORT`
  - To check actual values, `spawn -> /bin/env`
- Resource limits come from pvmd
  - Memory limit
  - Max number of open files
  - Corefile size
  - To check, `spawn -> /bin/csh -c limit`

Environment Variables

- PVM uses unix environment variables
  - Find resources (`HOME, PVM_ROOT, ...`)
  - Specify defaults (`PVM_DEBUGGER`)
  - Inherit attributes (`PVMSOCK, PVMTMASK`)
- User programs can use them (`DISPLAY`)
- Passing through `pvm_spawn`
  - What PVM needs is passed automatically
  - Other selected by `PVM_EXPORT` variable
  - Value of `PVM_EXPORT` is `NAME1:NAME2...`
- Unix function to get environment value
  - Declare `char *getenv(char *);`
  - Retrieve value with `char *val = getenv("NAME");`
- No standard put/setenv in libc
  - Use `pvmputenv("NAME=value")`
  - Adds or replaces entry in environment
  - String must stay around after call (don't free or use auto var.)
- What would be nice
  - Extend to non-unix machines
  - Spawn should take array of new environment values
  - Get rid of `PVM_EXPORT`?
    (... but big environments, non-uniform installation, ...)

Example - Setting working directory

This only works if the directory is exactly the same on each host

```
exportmypath()
{
   static char path[4096];

   FILE *ff;
   int i;

   strcpy(path, "MYWD=");
   i = strlen(path);
   ff = popen("pwd", "r");
   fgets(path + i, sizeof(path) - i, ff);
   fclose(ff);
   i = strlen(path);
   if (path[i - 1] == '\n')
      path[i - 1] = 0;
   pvmputenv(path);
   pvmputenv("PVM_EXPORT=MYWD");
}

setmypath()
{
   chdir(getenv("MYWD"));
}
```

Using pvm_setopt

- Sets parameters in libpvm
- Catch-all function for seldom-used options and tuning
- `int oval = pvm_setopt(int what, int newval)`
- `int val = pvm_getopt(int what)`
- `what` choices are:

What	Newval	Default
	`PvmDontRoute`	
`PvmRoute`	`PvmAllowDirect`	⇐
	`PvmRouteDirect`	
`PvmDebugMask`	bit-wise OR	
	0 (no reporting)	
`PvmAutoErr`	1 (print message)	⇐
	2 (exit on error)	
`PvmFragSize`	40 – 1M *	*
`PvmResvTids`	0, 1 (allow)	0
`PvmOutputTid`	collector-tid	0
`PvmOutputCode`	message tag	0
`PvmTraceTid`	collector-tid	0
`PvmTraceCode`	message tag	0
`PvmSelfOutputTid`	collector-tid	0
`PvmSelfOutputCode`	message tag	0
`PvmSelfTraceTid`	collector-tid	0
`PvmSelfTraceCode`	message tag	0
`PvmShowTids`	0 (no), 1 (yes)	1
`PvmPollType`	`PvmPollConstant`	
	`PvmPollSleep`	⇐
`PvmPollTime`	*	100000

(*) Implementation-dependent

Debugging PVM Applications

- First, the bad news
  - Adding prints to your code is still state-of-the-art technology
  - There are few viable parallel debuggers
  - They may not understand PVM concepts
- It can be difficult to determine the scope of the problem
  - Bugs in the computation
  - In the parallel algorithm
  - Nondeterminacy
  - Resource limits
- Libpvm can go insane if something scribbles on it
  (e.g. by going past bounds of an array)
  Lesson: Figure out what caused the *first* error
- You may need to combine methods to figure it out
- Use PVM tracing (or XPVM)
  - To see what tasks are failing
  - To find out where tasks are hanging
- Then use debuggers to analyze the serial code

Debugging PVM Applications, cont'd

- To trace programs
  - Set trace mask with console `trace` command
  - Spawn first task with `spawn -@`*filename* `...`
  - Other tasks in job inherit trace context
- Where do trace messages go if a task is started by hand?
  - Nowhere.
  - Spawn an xterm and then start the task there
    `alias tr-xterm spawn -@ -.  /bin/csh -cf 'exec xterm -n TASK'`
    Task context is held open by xterm
    (put this in your `$HOME/.pvmrc`)
- Note: Console and XPVM tracefiles are not compatible
- To run tasks under a debugger
  - `setenv DISPLAY` *myhost*`:0.0`
  - `setenv PVM_EXPORT DISPLAY`
  - When calling `pvm_spawn()`, set `PvmTaskDebug`
  - From the console, use `spawn -?  ...`
  - PVM starts an xterm with debugger running process
    (To do something else, edit `$PVM_ROOT/lib/debugger`)

Debugging PVM Applications, cont'd

- Getting corefiles can be a problem
- Corefiles from different tasks will be scribbled on top of each other
  - For tasks running on the same host
  - With home directory cross-mounted between hosts
- Can't usually change the corefile name
- Change to unique directory for each task

```
main()
{
    int tid;
    char buf[32];

    tid = pvm_mytid();
    sprintf(buf, "0x%08x", tid);
    mkdir(buf, 0777);
    chdir(buf);

    /* rest of program */
}
```

Or install a signal handler to do this just before dumping
May need to catch `SIGSEGV`, `SIGBUS`, `SIGFPE` or others

Resource Limitations in PVM

- Memory use
  - Pvmds and tasks use `malloc` to get memory
  - Malloc never gives memory back – image size won't shrink
  - It is reused, however
  - Unreceived messages are stored at destination (task or pvmd)
    Don't send messages that will never be received
  - Be careful when using `pvm_sets/rbuf()`
    Don't save buffers and forget about them
- Resource limitations imposed by
  - System memory
    * Size and number of messages that can exist
  - Number of open files allowed in system or process
    * Number of *direct route* connections a task can establish
      About N_{files}
    * Number of tasks a single pvmd can manage
      About $N_{files}/2$ if spawned
      About N_{files} otherwise
  - Number of shared memory segments
    * Number of tasks on shared memory machine
  - Number of machine partitions
    * Number of tasks on distributed memory multiprocessors
  - You may not see a message when you reach a resource limit