# Programmer's Guide

## Scyld ClusterWare Release 5.11.7-5117g0000

## April 25, 2016

**Programmer's Guide: Scyld ClusterWare Release 5.11.7-5117g0000; April 25, 2016**

Revised Edition

Published April 25, 2016



# Table of Contents

# Preface

Welcome to the Programmer's Guide. It is written for use by the Scyld ClusterWare cluster application programmer. In this document we outline the important interfaces, services, and mechanisms made available to the programmer by the Scyld ClusterWare system software.

In many cases, many of the relevant interfaces are the same as on any standard parallel computing platform. For example, most of the MPI interface provided by MPICH is the same on Scyld ClusterWare as on any cluster. In these cases, this document will refer to relevant documentation for the details. In some cases, while the interface may be the same, subtle differences in Scyld's implementation compared to other implementations will be discussed.

The interfaces discussed in this document include those that might be pertinent to application programs in addition to those that might be used by system programmers developing servers or environments. In all cases, it is assumed that the readers of this document are intent on developing programs. Most information of interest to more casual users or administrators may be found in the *User's Guide* or the *Administrator's Guide*.

## Feedback

We welcome any reports on errors or difficulties that you may find. We also would like your suggestions on improving this document. Please direct all comments and problems to support@penguincomputing.com.

When writing your email, please be as specific as possible, especially with errors in the text. Please include the chapter and section information. Also, please mention in which version of the manual you found the error. This version is *Scyld ClusterWare HPC, Revised Edition*, published April 25, 2016.

# Chapter 1. Introduction

Scyld ClusterWare system software has been designed to make it easier and more intuitive to write, debug, and tune cluster applications. On the surface, programming a Scyld ClusterWare computer is much like programming any other message passing computer system — the programmer uses a message passing library such as MPI, PVM or directly using TCP/IP sockets to pass data between processes running on different nodes. The resulting programs may be run interactively, using the Scyld BBQ queuing system, or submitted using site-wide job schedulers like PBS or LSF. Programs are debugged using tools such as gdb or TotalView. While the Scyld ClusterWare programming environment utilizes all of these software components, there is a subtle difference in the machine model that affects how these components are used and makes program development activities flow more naturally.

Two fundamental concepts underlying Scyld ClusterWare software design are providing the ability to deploy packaged applications, and making those applications easy for end users to use. The Scyld ClusterWare system provides new interfaces that allow an application to dynamically link against the communication library that matches the underlying communication hardware, determine the cluster size and structure, query a user-provided process-to-node mapping function and present the resulting set of processes as a single locally-controlled job. All of these interfaces are designed to allow creating effective applications that run over a cluster, without the end user needing to know the details of the machine, or even that the system is composed of multiple independent machines.

One subsystem that the Scyld ClusterWare system software uses to provide this functionality is a unified process space. This is implemented with an in-kernel mechanism named BProc, the Beowulf Process space extension. In the simplest case, BProc is a mechanism for starting processes on remote nodes, much like rsh or ssh. The semantics go considerably beyond merely starting remote processes: BProc implements a global Process ID (PID) and remote signaling. Using global PID space over the cluster allows users on the master node to monitor and control remote processes exactly as if they were running locally.

A second subsystem, closely related to BProc, is VMA-migrate. This mechanism is used by programmers to create processes on remote machines, controlled by the BProc global PID space. This mechanism is invisible to end users, and has a familiar programming interface. Creating remote processes is done with new variants of the fork() and exec() system calls. Controlling those processes is done with the existing process control system calls such as kill() and wait(), with the semantics unchanged from the local process POSIX semantics. An additional system call is bproc_move(), which provides the ability to migrate the entire set of Virtual Memory Areas (VMA) of a process to a remote machine, thus the name VMA-migrate.

The VMA-migrate system is implemented with a transparent system-wide mechanism for library and executable management. This subsystem isolates requests for libraries and executables from other filesystem transactions. The Scyld ClusterWare system takes advantage of the special semantics of libraries and executables to implement a highly effective whole-file-caching system invisible to applications and end users. The end result is a highly efficient process creation and management system that requires little explicit configuration by programmers or administrators.

A third subsystem is the integrated scheduling and process mapping subsystem, named Beomap. This subsystem allows an application to request a set of remote nodes available for use. Either the system administrator or end user may provide alternate scheduling functions. Those functions typically use information from the BeoStat subsystem or a system network topology table to suggest an installation-specific mapping of processes to nodes.

In this document we detail the important interfaces, services, and mechanisms made available to the programmer by the Scyld ClusterWare system software. In many cases the relevant interfaces are the same as on any standard parallel computing platform. For example, the MPI communication interface provided by BeoMPI on Scyld ClusterWare is identical to any cluster using MPICH. In these cases, this document will refer to relevant documentation for the details. In other cases the interface may be the same, yet there are differences in behavior. For instance MPI programs run on a single processor until they call MPI_Init(), allowing the application to modify or specify how it should be mapped over the cluster.

The interfaces discussed in this document include those that might be pertinent to application programs in addition to those that might be used by system programmers developing servers or environments. In all cases, it is assumed that the readers of

this document are intent on developing programs. Most information of interest to more casual users or administrators may be found in the *User's Guide* or the *Administrator's Guide*.

## Software Design Cycle

The design cycle for developing parallel software is similar to most other software development environments. First a program is designed, then coded using some kind of text editor, then compiled and linked, and finally executed. Debugging consists of iterating through this cycle.

Developing parallel code on a Scyld ClusterWare computer is very similar. All of these activities, except execution, are typically performed on a single machine, just as in traditional software development. Execution occurs on multiple nodes, and thus is somewhat different. However using the Scyld ClusterWare system the execution of the program is managed from a single machine, so this distinction is less dramatic than with earlier cluster system designs.

Where the development cycle is different for a parallel machine is in many of the details of the basic development steps. In particular, the programmer must design the program for parallel execution, must code the program using libraries for parallel processing, must compile and link against these libraries, and then must run the program as a group of processes on multiple nodes. Debugging a parallel program is complicated by the fact that there are multiple processes running as part of the program. Thus, the things a programmer does to debug code — such as inserting a print statement into the code — must take into account that those things may affect all of the processes. Thus, this document focuses on issues related to the design of parallel programs, the use of parallel programming libraries, compiling and linking, and debugging.

Another important activity in developing software for parallel computers is porting code from other parallel systems to Scyld ClusterWare. The few differences in the Scyld ClusterWare system are usually the result of cleaner semantics or additional features. This document lists known issues so that porting software is an easier task.

# Chapter 2. BProc Overview

The heart of the Scyld ClusterWare single system image is the BProc system. BProc consists of kernel modules and daemons that work together to create a unified process space and implement directed process migration. BProc is used to create a single cluster process space, with processes anywhere on the cluster appearing on the master as if they were running there. This allows users to monitor and control processes using standard, unmodified Unix tools.

The BProc system provides a library API that is used to create and manage processes on the compute nodes, as well as provide cluster membership state and limit node access. These calls are used by libraries such as MPI and PVM to create and manage tasks. They may also be used directly by the programmer to implement parallel software that is less complex and more functional than most other cluster programming interfaces.

For applications that have intense, complex communication patterns working on a single end goal, it is normally advisable to use message passing libraries such as MPI or PVM. The MPI and PVM interfaces are portable, provide a rich set of communication operations, and are widely understood among application programmers. The Scyld ClusterWare system provides specialized implementations of PVM and MPI that internally use the BProc system to provide cleaner semantics and higher performance.

There are many applications where it is not appropriate to use these message passing libraries. Directly using the native BProc API can often result in a much less complex implementation or provide unique functionality difficult to achieve with a message passing system. Most programming libraries, environments, tools, and services are better implemented directly with the BProc API.

A master node in a Scyld ClusterWare system manages a collection of processes that exist in a distinct process space. Each process in this process space has a unique PID, and can be controlled using Unix signals. New processes created by a process in a given process space remain in the process space as children of the process that created them. When processes terminate, they notify the parent process through a signal. BProc allows a process within a master node's process space to execute on a compute node. This is accomplished by starting a process on the compute node — which technically is outside the master node's process space — and binding it to a "ghost" process on the master node. This ghost process represents the process on the master node, and any operations performed on the ghost process are, in turn, performed on the remote process. Further, the remote node's process table must be annotated to indicate that the process is actually bound to the process space of the master. Thus, if the process performs a system call relative to its process space, such as creating a new process, or calling the kill() system call, that system call is performed in the context of the master's process space.

A process running on a compute node has a PID in the process space of the compute node, and an independent process running on the compute node can see that process with that PID. From the perspective of the process itself, it appears to be executing on the master node and has a PID within the master node's process space.

The BProc API is divided into three groups of functions. First, the machine information functions provide a means of discovering what compute nodes are available to the program, what the status of nodes is, what the current node is, etc. Second, the process migration functions provide for starting processes, moving processes, and running programs on compute nodes. Finally, a set of process management functions allows programs to manage how processes are run on compute nodes.

## Starting a Process with BProc

The most fundamental task one performs with BProc is to start a new process. For traditional Linux programs the mechanism for starting a new process is fork(). Using fork(), a Linux process creates an exact duplicate of itself, right up to and including the contents of memory and the context of the program. The new process is a child of the original process and has a new process ID which is returned to the original process. The fork() call returns zero to the new process.

BProc provides a variation on the fork() system call:

```
int bproc_rfork(int node);
```

This function behaves like Linux fork(), except that the process is started on a remote node specified by the node argument, and has a ghost process created on the master node. The function returns the PID of the ghost process to the original process and zero to the new process.

Note that there are important differences between Linux fork() and bproc_rfork(). BProc has a limited ability to deal with file descriptors when a process forks. Under Scyld ClusterWare, files are opened directly on the compute node, thus a file opened on one node before a call to bproc_rfork() does not translate cleanly into an open file on the remote node. Similarly, open sockets do not move when processes move and should not be expected to.

A process started with bproc_rfork does have stdout and stderr connected to the stdout and stderr of the original process and IO is automatically forwarded back to those sockets. Variations on the bproc_rfork() call allow the programmer to control how IO is forwarded for stdout, stderr, and stdin, and can also control how shared libraries are moved for the process. See the BProc reference manual for more details.

The bproc_rfork() call is a powerful mechanism for starting tasks that will act as part of a parallel application. Most applications that are written for Beowulf class computers consist of a single program that is replicated to all of the tasks that will run as part of that job. Each task identifies itself with a unique number, and in doing so each selects part of the total computation to perform. The bproc_rfork() call works well in this scenario because a program can readily duplicate itself using the bproc_rfork() system call. In order to coordinate the tasks, it is necessary to establish communication between the tasks and between each task and the original process. Again, the semantics of bproc_rfork() are ideal because the original task can set up a communication endpoint (such as a socket) before it calls bproc_rfork() to start the tasks at which point each task will automatically know the address of the original process and can establish communication. Once this is accomplished, the original process can exchange information between the tasks so that they can communicate among themselves.

```
/* Create running processes on nodes 1, 5, 24, and 65. */
int target_nodes[] = { 1, 5, 24, 65};
for (i = 0; i < tasks; i++)
{
 int pid = bproc_rfork(target_nodes[i]);
 if (pid == 0) /* I'm the child */
  break; /* only the parent does bproc_rfork(). */
}
```

Thus the bproc_rfork() call forms the core of parallel processing libraries such as MPI or PVM by providing a convenient means for starting tasks and establishing communication. This same facility can be used to start parallel tasks when a non-MPI (or non-PVM) program is desired. While this is not as portable and convenient — and thus not well suited to most applications — it may be appropriate for some system services, library development, and toolkits.

BProc does not limit the programmer to using fork() semantics for starting processes. Some situations call for starting a new program from an existing process much like the Linux system call execve(). For this, BProc provides two different calls:

```
int bproc_rexec(int node, char *cmd, char **argv, char **envp);
int bproc_execmove(int node, char *cmd, char **argv, char **envp);
```

These two functions are almost identical in that they cause the current process to be overwritten by the contents of the executable file specified by cmd which is then executed from the beginning with the arguments and environment given in argv and envp. In this the semantics are very similar to the Linux execve() system call. The difference is that the process moves to the node specified, leaving a ghost process on the master node. All of the same caveats that apply to bproc_rfork() apply to these calls as well, and there are other variations that allow additional control of IO forwarding and library migration.

The difference between these two calls is where the arguments are evaluated. With bproc_rexec(), the executable file is expected to reside on the compute node where the process will run. Thus with this call, the process first moves to the new node, and then loads the new program. With the bproc_execmove() call, the executable file is expected to reside on the node where the original process resides. Thus, this call loads the new program first, and then moves the process to the new node. In the case where the executable file resides on a network file system that is visible from all nodes, the effect of these two

calls is essentially the same, though the bproc_execmove() call may be more efficient if the original process exists on the node where the executable file actually resides.

Finally, BProc provides a means to simply move a process from one node to another:

```
int bproc_move(int node);
```

This call does not create a new process, other than possibly creating a ghost process, but simply causes the original process to move to a different node. Again, all caveats of bproc_fork() apply. This call is probably of most value in load balancing mechanisms but may be useful in situations where each task in a program needs to perform some IO on the master node to load its memory before moving out to a remote node.

## Getting Information About the Parallel Machine

In order to start processes on compute nodes with BProc the programmer needs to know the number of the compute node. When using a library like MPI, some kind of scheduler selects the nodes to run each task on, thus hiding this from the programmer. Using BProc, some mechanism must be used to select a compute node to start processes on. If a scheduler is available, it can be called to get node numbers to run processes on. Otherwise, the programmer must select nodes. Similarly, if the programmer's task is to write a scheduler — as may be the case if special scheduling is needed for the application — then the programmer must have a means of learning what nodes are available to the BProc system for running processes.

BProc includes a set of system calls that allow a program to learn what nodes are known to BProc, and what the status of those nodes is. Under BProc, all of the compute nodes that are known to the system are numbered from 0 to P-1, where P is the number of compute nodes known to BProc. In addition, there is the master node, which is numbered -1. Thus, the first call a program will make is often:

```
int bproc_numnodes(void);
```

This call returns the number of compute nodes known to BProc. Sometimes it is important for a program to know which node it is currently executing on. It may choose to perform some actions only if it is on a special node or it may use the information to identify itself to other processes. BProc returns this information with the call:

```
int bproc_currnode(void);
```

This call returns the number of the node the process is currently executing on. Its return values range from -1 to P-1.

A node known to BProc is not necessarily available for running user processes. Not all nodes may be operational at any given point in time. Nodes may be in various stages of booting (and thus might be available momentarily) or might have suffered an error (which might bear reporting) or be shut down for maintenance. In any case, BProc provides a mechanism to learn the state of each compute node, as far as BProc is currently aware. The master node is always assumed to be up. The call provided for this is:

```
int bproc_nodestatus(int node);
```

This function returns one of several values, which are defined in the header file bproc.h as follows:

```
bproc_node_down, bproc_node_unavailable, bproc_node_error,
bproc_node_up, bproc_node,reboot, bproc_node_halt,
bproc_node_pwroff, bproc_node_boot
```

BProc is a low level system utility and does not generally get involved in node usage policies. Thus, if a node is up and permissions are set correctly, BProc will allow any user process to start a process on a node. In many environments there is a need to manage how nodes are used. For example, it may be desirable to guarantee that only one user is using a particular node for a period of time. It may be important to assign a set of nodes to a user or group of users for exclusive use over a

period of time. It may be important to require users to start processes via some intermediary like a batch queue system or accounting system and thus prevent processes from being started directly.

In order to accommodate these issues, BProc provides a permission and access mechanism that is based on the Unix file access model. Under this model, every node is owned by a user and by a group and has execute permission bits for user, group and all. In order for a user to start a process on a node, an execute permission bit must be set for a matching user, matching group, or all.

These functions need not be used at all in many systems, but allow system programs to enforce policies controlling which users can run processes on which nodes. For example, a batch queue scheduler could decide to allocate 16 nodes to a job, and change the user of those nodes to match the user who submitted the job, thus preventing other users from inadvertently running processes on those nodes.

A program that is making scheduling decisions needs to know not only what nodes are known, and what nodes are actually operational, but what nodes have permissions set appropriately to allow processes to be run. BProc provides a call to retrieve this information for each node known to BProc.

```
int bproc_nodeinfo(int node, struct bproc_node_info_t *info);
```

This function fills in the following struct:

```
struct bproc_node_info_t
{
     int      num_nodes;         /* Info about your state of the world */
  int      curr_node;
  int      node;              /* Info about a particular node */
  int      status;
  uint32_t addr;
  int      user;
  int      group;
};
```

Note that this call not only returns permission and node ownership information, but also includes the node status and network address. Thus this single call provides the information to decide if a process may be started on a node, along with the information needed to contact the node.

Permission to start a process on a node is distinct from the decision of whether that node has the resources, such as available memory and CPU cycles, to support the process. The scheduling and mapping policy is support by resource information that may be retrieved through the Beostat API. See the "Programming with Beostat API" section in this manual.

In order to implement schedulers, system programs must be able to control the ownership and access permission of compute nodes. BProc provides functions that set those attributes. These functions can only be used by root or a user with appropriate permissions.

```
int bproc_setnodestatus(int node, int status);
```

This function allows a node management program to force node status to a particular state.

```
int  bproc_chmod(int node, int mode);
int  bproc_chown(int node, int user);
int  bproc_chgrp(int node, int group);
```

These functions change the user and group that own a node, or the User/Group/Others execute permission for a node. They use the same semantics as Unix file ownership and execute permission.

```
#define BPROC_X_OK 1
```

```
int bproc_access(node, mode);
```

This function checks if the current user may start a process on the specified node. This is similar to the access() system call for files. The only useful value for mode is BPROC_X_OK.

The last set of node management functions is used to implement multiple master systems and redundant, robust, and available servers.

```
int bproc_slave_chroot(int node, char *path);
```

This function sets a new root directory on a compute node, similar to the chroot() system call on the master node (local machine). This is used internally to implement isolated multiple master clusters, and allows a node to be installed with more than one configuration with the specific configuration set based on the process to be started.

```
int bproc_detach(long code);
```

This function allows a process to detach itself from control by the master. The master sees the code as the apparent exit status of the process.

# Chapter 3. BeoStat Overview

Scyld Beostat is a layered state, status, and alert system. It operates from two sources of data: a snapshot of static information gathered when a compute node first joins the cluster, and dynamic information updated by a light-weight program on each cluster node that unicasts or multicasts real-time status. This information is collected into a shared memory region on the master, which in turn is read by library routines.

Programs may hook into the status information at multiple levels. Directly receiving the multicast data stream provides an immediate indication of changed data, but this puts an additional, constant load on the monitoring system and the communication protocol is not documented as an end-user format. Reading the data gathered into the shared memory region is very effective for programs such as schedulers that need very efficient access to all data, but the format may unpredictably change between release versions. Most programs should instead use the highest level access, the BeoStat library.

The Scyld BeoStat library provides a collection of functions that allow a system program to retrieve a wide variety of performance and status data about any node in the cluster. These functions include information on CPU, memory, swap area, file system usage, network statistics, current process load, and details of the system architecture. This data is useful for monitoring system usage and making scheduling decisions and thus may be of interest to programmers developing system monitoring and scheduling tools, and may optionally be directly used by applications to modify their scaling behavior.

## Node CPU Load Information

The functions in this set abstract details of the processor architecture and speed. They return general load information about CPU, nodes, and the cluster as a whole. Performance data may be used relative to other nodes, but may be useful as an absolute metric.

```
float beostat_get_cpu_percent (int node, int cpu);
unsigned long beostat_get_net_rate (int node);
int beostat_get_disk_usage (int node, int *max, int *curr);
int beostat_count_idle_cpus (float threshold);
int beostat_count_idle_cpus_on_node(int node, float cpu_idle_threshold);

extern int beostat_get_avail_nodes_by_id (int  **node_list,
                                          uid_t  uid,
                                          gid_t *gid_list,
                                          int    gid_size );

extern int beostat_is_node_available (int    node,
                                      uid_t  uid,
                                      gid_t *gid_list,
                                      int    gid_size  );
```

## Specific Node Information

The functions in this set provide details of node CPU load and resource usage. They return exact load averages and numeric values for specific resources. The information returned by these functions are precise, although in the case of processor MHz may require unpredictable correction factors.

```
int beostat_get_MHz (int node, float *MHz);

int beostat_get_stat_cpu (int node, int cpu, struct beostat_stat_cpu *stat_cpu);

struct beostat_stat_cpu
```

```
{
 long user;
 long system;
 long nice;
 long idle;
};

int beostat_get_time (int node, struct node_time *node_time);

struct node_time
{
 time_t time;
};


int beostat_get_meminfo (int node, struct beostat_meminfo *meminfo);

struct beostat_memusage
{
 long used;
 long free;
};

struct beostat_meminfo
{
 struct beostat_memusage mem;
 struct beostat_memusage swap;
 long shared;
 long buffers;
 long cached;
};


int beostat_get_loadavg (int node, struct beostat_loadavg *loadavg);

struct beostat_loadavg
{
 float load[3];
 int num_active_procs;
 int total_procs;
 int last_pid;
};
```

## Detailed CPU State and Status

The functions in this set are processor architecture specific. The examples are for the IA32 "x86" architecture, with similar functions provided for the other architectures.

```
int beostat_get_cpuinfo_x86 (int node, struct beostat_cpuinfo_x86 *cpuinfo);

struct beostat_cpuinfo_x86
{
 int processor;            /* [which cpu (SMP)] */
```

```
 char vendor_id[16];       /* x86_vendor_id */
 int family;               /* x86 */
 int model;                /* x86 model */
 char name[64];            /* x86 model ID */
 int stepping;             /* x86 mask */
 float MHz;                /* derived from bogomips */
 int cache_size_KB;        /* x86_cache_size */
 boolean fdiv_bug;         /* same */
 boolean hlt_bug;          /* ~hlt_works_ok */
 boolean sep_bug;          /* [Derived] */
 boolean f00f_bug;         /* same */
 boolean coma_bug;         /* same */
 boolean fpu;              /* hard_math */
 boolean fpu_exception;    /* based on exception 16 */
 int cpuid_level;          /* same */
 boolean wp;               /* wp_works_ok */
 //  char flags[256];          /* x86_capability */
 float bogomips;           /* loops_per_sec derived */
};
```

# Chapter 4. Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard interface for libraries that provide message passing services for parallel computing. There are multiple implementations of MPI for various parallel computer systems, including two widely used open source implementations, MPICH and LAM-MPI. The Scyld ClusterWare system software includes BeoMPI, our enhanced version of MPICH. For most operations, BeoMPI on a Scyld ClusterWare system is equivalent to running MPICH under Linux. There are a few places where ClusterWare's implementation of MPICH is different, and these will be pointed out in this chapter. This chapter also includes a basic introduction to MPI for new parallel programmers. Readers are referred to one of the standard texts on MPI or to the MPI Home Page: http://www.mcs.anl.gov/mpi

There are, in fact, two MPI standards. MPI-1 and MPI-2. MPI-2 is not actually a newer version of MPI, but a set of extensions to MPI-1. Most of what a typical parallel programmer needs is in MPI-1, with the possible exception of parallel IO, which is covered in MPI-2. Many implementations cover all of MPI-1 and some of MPI-2. At the time of this writing very few MPI implementations cover all of MPI-2, thus a new programmer should be wary of seeking out features of MPI-2 that, while interesting, may not be required for their particular problem.

MPI-1 includes all of the basic MPI concepts and in particular:

- Point-to-point communication
- Collective communication
- Communicators
- MPI datatypes

While MPI-2 covers a number of extensions and additional features such as:

- MPI-IO
- Single-ended communication
- Connection-based communication
- Issues such as threading

The MPI home page provides detailed specifications of all of the MPI features and references to a number of texts on the subject.

## Fundamental MPI

Every MPI program must call two MPI functions. Almost every MPI program will call four additional MPI functions. Many MPI programs will not need to call any more than these six. Beyond these six, there are another dozen or so functions that will satisfy most MPI programs. Thus, this introduction takes the approach of introducing these functions in order of need. Hopefully, the new programmer will be able to progress through the text as the complexity of his programs increases.

The six most fundamental MPI function calls are as follows:

- MPI_Init - start using MPI
- MPI_Comm_size - get the number of tasks
- MPI_Comm_rank - the unique index of this task
- MPI_Send - send a message

- MPI_Recv - receive a message
- MPI_Finalize - stop using MPI

The first MPI call for all MPI programs must be to MPI_Init. This function initializes MPI for the program. Its arguments are the standard C command line arguments — which allows some implementations to check for MPI specific arguments, process them, and remove them (so your program doesn't have to parse MPI arguments). The last call every MPI program must make is to MPI_Finalize. Failure to call MPI_Finalize can result in some rather interesting run-time errors that can be quite difficult to debug. Below is an example of the minimum possible MPI program.

```
#include <mpi.h>
main(int argc, char **argv)
{
  MPI_Init(&argc, &argv );
  // put program here
  MPI_Finalize();
}
```

When an MPI program runs, there is not one, but many copies of exactly the same program. These programs are called "tasks" in this manual, and they are the means by which parallel processing is achieved. To get better performance, each task should run on a different processor — or compute node in a Scyld ClusterWare system. It is possible to write MPI programs that expect a fixed number of tasks to be running, but it is far better to write programs that can run correctly using any number of tasks. In this case, it is important to know how many tasks are running as part of the current collection of tasks or "jobs." Furthermore, it is often important to know which of those tasks is the currently running task. In other words, if there are 16 tasks running, each task should have a unique number from 0 to 15 that identifies that task. This number is called a "rank."

In fact, MPI includes a complex system for arbitrarily grouping sets of tasks together and giving each task within a group a rank that is unique relative to that group. This is an advanced concept that will not be covered here. Let it suffice to say that when an MPI job is created there is a group that consists of all of the tasks in that job. Furthermore, all functions that involve interaction among tasks in MPI do so through a structure called a "communicator" which can be thought of as a communications network between tasks in a group. The communicator associated with the global group that contains all processes in the job is known as:

```
MPI_COMM_WORLD
```

This symbol appears in most of the function calls that follow. For the beginning MPI programmer, this argument is always the same. Considerably more advanced programmers can construct their own communicators for building much more complex programs, but this feature is not discussed in detail here.

Thus, most MPI programs need to know how many tasks there are in MPI_COMM_WORLD, and what their rank is within MPI_COMM_WORLD. The next two MPI function calls provide this information. These will often be called early in the execution of a program and the information saved in a variable of some sort, but they can be called repeatedly as needed through the execution of the program.

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI_Comm_size returns the number of tasks in the job, while MPI_Comm_rank returns the rank of the current task (0 .. size-1). These values are typically used to determine what part of the computation each task will perform, and what is the rank of another task that the current task wants to communicate with. For example, if we wanted every task to communicate with the task whose rank is one greater than itself — and for the task whose ranks is size-1 to communicate with task 0, then the following could be used:

```
int size, rank, partner;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
partner = rank + 1 % size;
```

The most fundamental of all MPI functions are MPI_Send and MPI_Recv. These are the functions actually used to send and receive data between tasks. In both cases the functions include a pointer to a user buffer that contains the data to send (for MPI_Send) or will receive the data (for MPI_Recv), two arguments that identify the data, two arguments that identify the other task involved in the communication, and a special argument called a "tag." The following examples show these calls:

```
[SCOUNT], rbuf[RCOUNT];
MPI_Status status;
MPI_Send(sbuf, SCOUNT, MPI_CHAR, rank+1%size, 99, MPI_COMM_WORLD);
MPI_Recv(rbuf, RCOUNT, MPI_CHAR, rank+1%size, 99, MPI_COMM_WORLD, &status);
```

The arguments are identified as follows:

- sbuf : pointer to send buffer
- rbuf : pointer to receive buffer
- SCOUNT : items in send buffer
- RCOUNT : items in receive buffer
- MPI_CHAR : MPI datatype
- rank+1%size : source or destination task rank
- 99 : message tag
- MPI_COMM_WORLD : communicator
- &status : pointer to status struct

The first argument of each call is the pointer to the user's buffer. The second two arguments specify the number of those items stored contiguously in the buffer and the type of the data being sent or received. MPI provides pre-defined MPI_Datatypes for the standard scalar types:

```
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT,
MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED
```

MPI also provides methods for creating user defined types that can represent all kinds of structures and records. These methods are not discussed here. The send buffer is assumed to be at least as big as SCOUNT times the size of the MPI_Datatype. When receiving messages, it is possible that the task may not know exactly what size the incoming message is. The arguments to the call specify how big the receive buffer is, and in this case the receive buffer must be big enough for the incoming message, or an error will occur and be reported through the MPI_Status struct. If the message is smaller than the buffer, then only part of the buffer will be filled in. MPI provides a function that allows the programmer to check the size of the next incoming message before receiving it:

```
MPI_Probe(rank+1%size, 99, MPI_COMM_WORLD, &status); // src, tag, comm, stat
```

The next argument to MPI_Send and MPI_Receive (after the MPI_Datatype) is the rank of the task that the program wants to send to or receive from. For MPI_Recv this can be MPI_ANY_SOURCE which will allow messages to be received from any other task. The source task can be determined through the MPI_Status argument. After the rank argument is the

message tag argument. All messages are sent with an integer tag. MPI_Receive will only receive a message sent with a tag that matches the tag specified in the MPI_Recv call. There isn't any predefined meaning to message tags. Some programs can be written with all messages using the same tag. In more complex programs the message tag allows the receiving task to control the order that messages are received. The choice of how the tags are used is entirely up to the programmer. MPI_Recv can receive a message with any tag by specifying MPI_ANY_TAG for the tag field.

The MPI_Comm argument specifies the group of tasks that is communicating. Messages sent with a given communicator can only be received using the same communicator. The rank specified as the destination or source of the message is interpreted in terms of the communicator specified. Unless the programmer is working with multiple communicators (an advanced topic) this should always be MPI_COMM_WORLD.

MPI_Recv has one more argument than MPI_Send. The MPI_Status struct defines three integer fields that return information to the program about a prior call to MPI_Recv. The fields are:

```
MPI_Status status;
printf( ... , status.MPI_SOURCE); // the source rank of the message
printf( ... , status.MPI_TAG);    // the message tag
printf( ... , status.MPI_ERROR);  // error codes
```

These are particularly useful if the program uses MPI_ANY_SOURCE or MPI_ANY_TAG. In addition, the status can be used to get a count of the actual number of items received using another MPI function call:

```
int count;
MPI_Get_count(&status, MPI_CHAR, &count)
```

This call is useful when the receiver doesn't know the actual size of the message being received.

When a task calls either MPI_Send or MPI_Recv, the corresponding message is said to be "posted." In order for a message to be delivered, an MPI_Send must be matched with a corresponding MPI_Recv. Until a given message is matched, it is said to be "pending." There are a number of rules governing how MPI_Sends and MPI_Recvs can be matched in order to complete delivery of the messages. The first set of rules are the obvious ones that specify that the message must have been sent to the receiver and received from the sender and they must be within the same communicator and the tags must match. Of course MPI_ANY_TAG matches any tag and MPI_ANY_SOURCE matches any sender. These however, only form the basis for matching messages. It is possible to have many messages posted at the same time, and it is possible that some posted MPI_Sends may match many MPI_Recvs and that some MPI_Recvs may match many MPI_Sends.

This comes about for a variety of reasons. On the one hand, messages can be sent and received from multiple tasks to multiple tasks. Thus all tasks could post an MPI_Send to a single task. In this case MPI makes no assumptions or guarantees as to which posted message will match an MPI_Recv. This, in fact, potentially creates non-determinism within the parallel program as one generally cannot say what order the messages will be received (assuming the MPI_Recv uses MPI_ANY_SOURCE to allow it to receive any of the posted messages). As stated in the MPI specification, fairness is not guaranteed. On the other hand, it is possible for more than one message to be sent from the *same* source task to a given destination task. In this case, MPI *does* guarantee that the first message sent will be the first message received, assuming both messages could potentially match the first receive. Thus MPI guarantees that messages are non-overtaking.

A more subtle issue is that of guaranteeing progress. Under MPI, if one task posts an MPI_Send that matches an MPI_Recv posted by another task, then at least one of those two operations must complete. It is possible that the two operations will not be matched to each other — if, for example, the MPI_Recv matches a different MPI_Send or the MPI_Send matches a different MPI_Recv, but in each of those cases at least one of the original two calls will complete. Thus, it is not possible for the system to hang with potentially matching sends and receives pending.

Finally, MPI addresses the issue of buffering. One way that a task can have two different MPI_Sends posted at the same time (thus leading to the situations described above) is that the MPI_Send function returns to the program (thus allowing the program to continue and potentially make another such call) even though the send is still pending. In this case the MPI library or the underlying operating system may have copied the user's buffer into a system buffer — thus the data can be said

to be "in" the communication system. Most implementations of MPI provide for a certain amount of buffering of messages and this is allowed for standard MPI_Send calls. However, MPI does not guarantee that there will be enough system buffers for the calls made, or any buffering at all. Programs that expect a certain amount of buffering for standard MPI_Send calls are considered incorrect. As buffering is an important part of many message passing systems, MPI does provide mechanisms to control how message buffering is handled, thus giving rise to the various semantic issues described here.

To summarize, MPI provides a set of semantic rules for point-to-point message passing as follows:

- MPI_Send and MPI_Recv must match
- Fairness is not guaranteed
- Non-overtaking messages
- Progress is guaranteed
- System buffering resources are not guaranteed

These semantic issues play an even bigger role when asynchronous send and receive are employed. This is a more advanced form of message passing that is very powerful and important for many applications. Asynchronous send and receive can quickly lead to all of the semantic issues listed above. The details of this are not discussed here.

## More MPI Point-to-Point Features

MPI provides several different communication modes which allow the programmer to control the semantics of point-to-point communication. Specifically, the point-to-point communication modes allow the programmer to control how and when message buffering is done, and in some cases may allow the underlying network transport to optimize message delivery. These different modes are selected by using one of four different variations of the MPI_Send function. All of these variations have the same argument list and work the same way, except as to how the message delivery semantics are affected. Each of these match with the same standard MPI_Recv function all.

```
// Standard Mode
MPI_Send(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
// Buffered Mode
MPI_Bsend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
// Synchronous Mode
MPI_Ssend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
// Ready Mode
MPI_Rsend(buf, count, datatype, dest, tag, MPI_COMM_WORLD);
```

In standard mode messages are buffered if the operating system has buffer space available. Otherwise, the call to MPI_Send will block until the message delivered or system space becomes available. In this case completion of the MPI_Send call does not imply that the message has been delivered, because it may be buffered by the operating system.

In buffered mode the programmer provides buffer space for messages by allocating user space memory and providing it to the MPI library for use in buffering messages. MPI_Bsend returns as soon as the message is copied into the buffer. If insufficient buffer space exists for the message, the call returns with an error. Thus, MPI_Bsend will not block as MPI_Send can. Completion of MPI_Bsend does not imply that the message has been delivered, because it can be buffered in the space provided.

MPI provides two functions for providing buffer space to the MPI library, and removing the buffer space.

```
MPI_Buffer_attach(void *buffer, int size);
MPI_Buffer_detach(void **buffer_addr, int *size);
```

MPI_Buffer_detach waits until all buffered messages are sent before it returns. After it returns, the buffer can be reused or freed as needed.

In synchronous mode MPI_Ssend blocks until a matching receive is posted and data is either buffered by the operating system or delivered to the destination task. Completion of MPI_Ssend does not imply completion of the corresponding MPI_Recv, but *does* imply the start of the MPI_Recv.

Ready Mode is a special communication mode used with systems that have network transports that can optimize throughput when sender and receiver are well synchronized. In this case, if the receiver is ready for the message (has already called MPI_Recv) when the send is made, data can be transfered directly between the user memory and the network device on both sending and receiving ends. MPI_Rsend can only be correctly called if the matching MPI_Recv has already been posted. MPI_Rsend completes when the message is either received or copied to a system buffer. On ClusterWare systems using traditional networking such as Ethernet, this call is no different than standard mode MPI_Send. Some advanced network technologies such as Myrinet may be able to make use of MPI_Rsend.

Many MPI applications work by having all of the tasks in the job exchange data, either in a ring of some sort, or with a specific partner task. In these cases each task must both send and receive data. Depending on the communication mode and composition of the messages, it is possible that this can result in deadlock. MPI provides a special send/recv primitive that both sends and receives a message, potentially with different tasks and guarantees that the semantics will not result in a deadlock.

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
 int dest, int sendtag,
 void *recvbuf, int recvcount, MPI_Datatype recvtype,
 int source, int recvtag,
 MPI_Comm comm, MPI_Status *status);
MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
 int dest, int sendtag,
 int source, int recvtag,
 MPI_Comm comm, MPI_Status *status);
```

In MPI_Sendrecv, the programmer specifies a buffer, count, type, tag, and destination/source for each message with a common communicator and status. With MPI_Sendrecv_replace, a single buffer, count, and type are specified with a source/send tag and destination/receive tag. This allows data of the same type and size to be sent from and received into a single buffer.

## Unique Features of MPI under Scyld ClusterWare

BeoMPI, ClusterWare's implementation of MPI, is based on MPICH. MPICH is a standard, portable MPI implementation developed at Argonne National Laboratories. The most common configuration for MPICH under Scyld ClusterWare is based on using a message passing library named "P4." The P4 library provides message transport over TCP/IP connections, and thus is suitable for use with Ethernet and Ethernet-like networks. (An alternative is GM, which works only with Myricom's Myrinet hardware.)

The most striking difference in how BeoMPI operates under Scyld ClusterWare versus MPICH on other clusters is how MPI tasks are started. In the basic MPICH implementation MPI programs are started using a special script "mpirun." This script takes a number of arguments that specify, among other details, how many tasks to create and where to run these tasks on the parallel machine. The script then uses some mechanism to start the tasks on each node — often utilities such as "rsh" or "ssh" are used. Thus a typical execution of an MPI program would look like:

```
mpirun -np 16 mpiapp inf outf
```

which runs 16 tasks of the program "mpiapp" with arguments "inf" and "outf."

Scyld ClusterWare system software provides an "mpirun" script that works just like the standard script, except for two things. First, the Scyld ClusterWare script has a few options that are unique to Scyld ClusterWare.

```
-np <int>  run <int> tasks
--all_cpus  run a task on each cpu
--all_local  run all tasks on master node
--no_local  run no tasks on master node
--exclude <int>[:<int>] run on any nodes not listed
--map <int>[:<int>]  run on nodes listed
```

Second, the way MPI jobs are run under Scyld ClusterWare is semantically cleaner than on other distributed or old-generation cluster systems. By using the internal BProc mechanism, processes are initiated on remote machines faster and are directly monitored and terminated.

Rather than start independent copies of the program, which later coordinate, the Scyld ClusterWare MPI implementation starts a single program. When MPI_Init() is called, bproc_rfork() is used to create multiple copies of the program on the compute nodes. Thus, the Scyld ClusterWare implementation of MPI has semantics similar to shared memory supercomputers such as the Cray T3E, rather than semantics of a collection of independent machines. The initialization code is run only once, which allows updating configuration files or locks without the need for a distributed file system or the need for explicit file locking. MPI_Init determines how many tasks to create by checking an environment variable. Thus, the Scyld ClusterWare implementation of mpirun actually just parses the mpirun flags, sets the appropriate environment variables, and then runs the program. The environment variables recognized by MPI_Init are as follows:

```
NP=<int>
ALL_CPUS=Y
ALL_LOCAL=Y
NO_LOCAL=Y
EXCLUDE=<int>[:<int>]
BEOWULF_JOB_MAP=<<int>[:<int>]
```

Using these environment variables, an MPI program can be run on a Scyld ClusterWare system without any special scripts or flags. Thus, if the flag "NP" has been set to the integer 16, the example listed above could be run as:

```
mpiapp inf outf
```

Further, turn-key applications can be developed that set the environment variables from inside the program before MPI_Init is called so that the user doesn't even have to be aware that the program runs in parallel.

A number of the flags provided under Scyld ClusterWare are useful for things like scheduling and debugging. The --all_local flag (ALL_LOCAL variable) is important for debugging programs, because it is much simpler to debug programs with all tasks running on the master node than actually running on the compute nodes. The --no_local flag (NO_LOCAL variable) is usually used to keep ALL tasks running on compute nodes and not load the master node with processes that must compete with interactive processing. The --exclude flag (EXCLUDE variable) can be used to keep tasks off of nodes that are reserved for special purposes such as I/O, visualization, debugging, etc. Also, the --map flag (BEOWULF_JOB_MAP variable) can be used to run tasks on specific nodes. This can be used by a scheduler to control where processes are run in a multi-user/multi-job environment.

## Building MPICH Programs

MPICH programs are those written to the MPI specification and linked with the Scyld ClusterWare MPICH libraries. This section discusses how to build (or compile) MPICH programs on Scyld ClusterWare using open source and commercial compilers.

### MPICH mpicc, mpiCC, mpif77 and mpif90

Scyld ClusterWare provides the **mpicc**, **mpiCC**, **mpif77** and **mpif90** compile wrappers to build C, C++, Fortran 77 and Fortran 90 MPICH binaries. These wrappers are found in `/usr/bin`. For example, to compile the C program `my_prog.c` into an MPICH-compatible binary `my_prog`:

```
[user@cluster ~]$ mpicc -o my_prog my_prog.c
```

By default, the Scyld ClusterWare compile wrappers use the GNU Compiler Collection (GCC) to build and link the code. You can see the parameters used to execute GCC tools by using the *-show* switch with **mpicc**, **mpiCC**, **mpif77** and **mpif90**:

```
[user@cluster ~]$ mpicc -show
gcc -L/usr/lib64/MPICH/p4/gnu -I/usr/include -lmpi -lbproc

[user@cluster ~]$ mpif77 -show
f77 -L/usr/lib64/MPICH/p4/gnu -I/usr/include -lmpi -lmpichfarg -lbproc -lmpe
```

### Using non-GNU compilers to build MPICH programs

You can change the compiler used to build and link MPICH programs by using the *-cc* switch with **mpicc**. Use *-cxx* for **mpiCC**, *-f77* for **mpif77**, and *-fc* for **mpif90**. For example, to compile a program `my_prog.c` with the Intel **icc** compiler, use:

```
[user@cluster examples]$ mpicc -cc=icc -o my_prog my_prog.c
```

Scyld ClusterWare supports the Intel and PGI compilers. Again, you can use *-show* to see how the Scyld ClusterWare compile wrappers will build and link your program:

```
[user@cluster examples]$ mpicc -cc=icc -o my_prog my_prog.c -show
icc -L/usr/lib64/MPICH/p4/intel -I/usr/include -o my_prog my_prog.c \
    -lmpi -lbproc
```

Notice that the Intel-compatible MPICH libraries from `/usr/lib64/MPICH/p4/intel` are linked in. Corresponding libraries exist in `/usr/lib64/MPICH` for PGI compilers.

## Building OpenMPI Programs

OpenMPI programs are those written to the MPI specification and linked with the Scyld ClusterWare OpenMPI libraries. This section discusses how to build (or compile) OpenMPI programs on Scyld ClusterWare using open source and commercial compilers.

### OpenMPI mpicc, mpiCC, mpif77 and mpif90

Scyld ClusterWare provides the **mpicc**, **mpiCC**, **mpif77** and **mpif90** compile wrappers to build C, C++, Fortran 77 and Fortran 90 OpenMPI binaries. Unlike MPICH, there exists a set of wrapper for various compilers. The env-modules package provides an easy means for the user to switch between the various compilers. Be sure to load the correct module to favor

the OpenMPI wrappers so that you do not use the MPICH compile wrappers found in `/usr`. For example, to compile the C program `my_prog.c` using the GNU compiler:

```
[user@cluster ~]$ module load openmpi/gnu
[user@cluster ~]$ mpicc -o my_prog my_prog.c
```

When you are done using OpenMPI, or if you wish to switch back to MPICH, simple unload the module:

```
[user@cluster ~]$ module unload openmpi/gnu
```

### Using non-GNU compilers to build OpenMPI programs

You can change the compiler used to build and link OpenMPI programs by loading the env-module for the compiler of your choice. For example, to compile the C program `my_prog.c` with the Intel **icc** compiler:

```
[user@cluster ~]$ module load openmpi/intel
[user@cluster ~]$ mpicc -o my_prog my_prog.c
```

Currently, there are modules for the GNU, Intel, and PGI compilers. To see a list of all of the available modules:

```
[user@cluster ~]$ module avail openmpi
--------------------------- /opt/modulefiles ------------------------------
openmpi/gnu(default) openmpi/intel        openmpi/pgi
```

For more detail, or for cases where using the wrappers isn't an option, you can use *-show* to see how the compiler wrappers will build and link your program:

```
[user@cluster ~]$ module load openmpi/intel
[user@cluster ~]$ mpicc -o my_prog my_prog.c -show
icc -I/usr/openmpi/include -I/usr/lib64/OMPI/intel -o my_prog my_prog.c \
    -L/usr/lib64/OMPI/intel -lmpi -lopen-rte -lopen-pal -libverbs -ldat \
    -lrt -ltorque -ldl -lnsl -lutil -lm
```

For more information about creating your own modules, see http://modules.sourceforge.net and the manpages **man module** and **man modulefile**.

## Notes

1. http://www.mcs.anl.gov/mpi
2. http://modules.sourceforge.net

# Chapter 5. Parallel Virtual File System (PVFS)

PVFS, the Parallel Virtual File System, is a very high performance filesystem designed for high-bandwidth parallel access to large data files. It's optimized for regular strided access, with different nodes accessing disjoint stripes of data.

Scyld has long supported PVFS, both by providing pre-integrated versions with Scyld ClusterWare and funding the development of specific features.

## Writing Programs That Use PVFS

Programs written to use normal UNIX I/O will work fine with PVFS without any changes. Files created this way will be striped according to the file system defaults set at compile time, usually set to 64 Kbytes stripe size and all of the I/O nodes, starting with the first node listed in the .iodtab file. Note that use of UNIX system calls read() and write() result in exactly the data specified being exchanged with the I/O nodes each time the call is made. Large numbers of small accesses performed with these calls will not perform well at all. On the other hand, the buffered routines of the standard I/O library fread() and fwrite() locally buffer small accesses and perform exchanges with the I/O nodes in chunks of at least some minimum size. Utilities such as tar have options (e.g. -block-size) for setting the I/O access size as well. Generally PVFS will perform better with larger buffer sizes.

The setvbuf() call may be used to specify the buffering characteristics of a stream (FILE *) after opening. This must be done before any other operations are performed:

```
FILE *fp;
fp = fopen("foo", "r+");
setvbuf(fp, NULL, _IOFBF, 256*1024);
/* now we have a 256K buffer and are fully buffering I/O */
```

See the man page on **setvbuf()** for more information.

There is significant overhead in this transparent access both due to data movement through the kernel and due to our user-space client-side daemon (pvfsd). To get around this the PVFS libraries can be used either directly (via the native PVFS calls) or indirectly (through the ROMIO MPI-IO interface or the MDBI interface). In this section we begin by covering how to write and compile programs with the PVFS libraries. Next we cover how to specify the physical distribution of a file and how to set logical partitions. Following this we cover the multi-dimensional block interface (MDBI). Finally, we touch upon the use of ROMIO with PVFS.

In addition to these interfaces, it is important to know how to control the physical distribution of files as well. In the next three sections, we will discuss how to specify the physical partitioning, or striping, of a file, how to set logical partitions on file data, and how the PVFS multi-dimensional block interface can be used.

### Preliminaries

When compiling programs to use PVFS, one should include in the source the PVFS include file by:

```
#include <pvfs.h>
```

To link to the PVFS library, one should add *-lpvfs* to the link line.

Finally, it is useful to know that the PVFS interface calls will also operate correctly on standard, non-PVFS, files. This includes the MDBI interface. This can be helpful when debugging code in that it can help isolate application problems from bugs in the PVFS system.

## Specifying Striping Parameters

The current physical distribution mechanism used by PVFS is a simple striping scheme. The distribution of data is described with three parameters:

- *base* — The index of the starting I/O node, with 0 being the first node in the file system
- *pcount* — The number of I/O servers on which data will be stored (partitions, a bit of a misnomer)
- *ssize* — Strip size, the size of the contiguous chunks stored on I/O servers

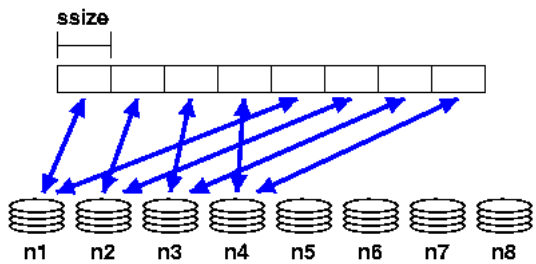


**Figure 5-1. Striping Example**

In Figure 5-1 we show an example where the base node is 0 and the pcount is 4 for a file stored on our example PVFS file system. As you can see, only four of the I/O servers will hold data for this file due to the striping parameters.

Physical distribution is determined when the file is first created. Using **pvfs_open()**, one can specify these parameters.

```
pvfs_open(char *pathname, int flag, mode_t mode);
pvfs_open(char *pathname, int flag, mode_t mode, struct pvfs_filestat *dist);
```

If the first set of parameters is used, a default distribution will be imposed. If instead a structure defining the distribution is passed in and the O_META flag is OR'd into the flag parameter, the physical distribution is defined by the user via the pvfs_filestat structure passed in by reference as the last parameter. This structure, defined in the PVFS header files, is defined as follows:

```
struct pvfs_filestat
{
 int base;   /* The first iod node to be used */
 int pcount; /* The number of iod nodes for the file */
 int ssize;  /* stripe size */
 int soff;   /* NOT USED */
 int bsize;  /* NOT USED */
}
```

The *soff* and *bsize* fields are artifacts of previous research and are not in use at this time. Setting the pcount value to -1 will use all available I/O daemons for the file. Setting -1 in the ssize and base fields will result in the default values being used.

If you wish to obtain information on the physical distribution of a file, use **pvfs_ioctl()** on an open file descriptor:

```
pvfs_ioctl(int fd, GETMETA, struct pvfs_filestat *dist);
```

It will fill in the structure with the physical distribution information for the file.

## Setting a Logical Partition

The PVFS logical partitioning system allows an application programmer to describe the regions of interest in a file and subsequently access those regions in a very efficient manner. Access is more efficient because the PVFS system allows disjoint regions that can be described with a logical partition to be accessed as single units. The alternative would be to perform multiple seek-access operations, which is inferior both due to the number of separate operations and the reduced data movement per operation.

If applicable, logical partitioning can also ease parallel programming by simplifying data access to a shared data set by the tasks of an application. Each task can set up its own logical partition, and once this is done all I/O operations will "see" only the data for that task.

With the current PVFS partitioning mechanism, partitions are defined with three parameters: offset, group size (gsize), and stride. The offset is the distance in bytes from the beginning of the file to the first byte in the partition. Group size is the number of contiguous bytes included in the partition. Stride is the distance from the beginning of one group of bytes to the next. Figure 5-2 shows these parameters.

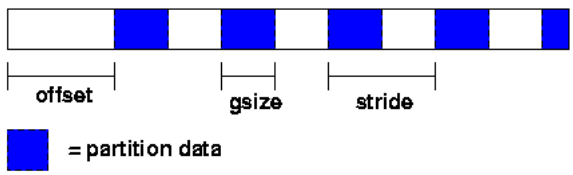


**Figure 5-2. Partitioning Parameters**

To set the file partition, the program uses a pvfs_ioctl() call. The parameters are as follows:

```
pvfs_ioctl(fd, SETPART, ∂);
```

where part is a structure defined as follows:

```
struct fpart
{
 int offset;
 int gsize;
 int stride;
 int gstride; /* NOT USED */
 int ngroups; /* NOT USED */
};
```

The last two fields, gstride and ngroups, are remnants of previous research, are no longer used, and should be set to zero. The **pvfs_ioctl()** call can also be used to get the current partitioning parameters by specifying the GETPART flag. Note that whenever the partition is set, the file pointer is reset to the beginning of the new partition. Also note that setting the partition is a purely local call; it does not involve contacting any of the PVFS daemons, thus it is reasonable to reset the partition as often as needed during the execution of a program. When a PVFS file is first opened a "default partition" is imposed on it which allows the process to see the entire file.

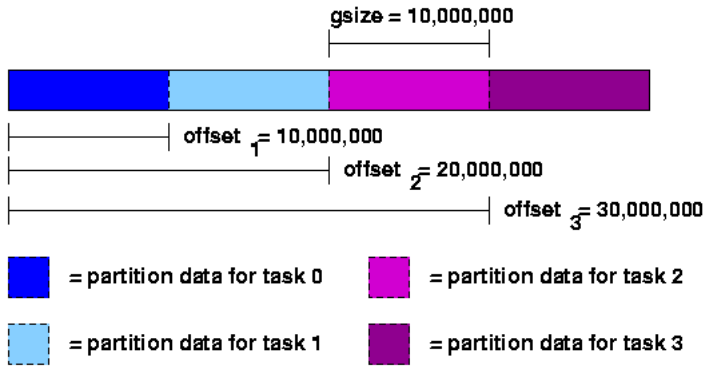


**Figure 5-3. Partitioning Example 1**

As an example, suppose a file contains 40,000 records of 1000 bytes each, there are 4 parallel tasks, and each task needs to access a partition of 10,000 records each for processing. In this case one would set the group size to 10,000 records times 1000 bytes or 10,000,000 bytes. Then each task (0..3) would set its offset so that it would access a disjoint portion of the data. This is shown in Figure 5-3.

Alternatively, suppose one wants to allocate the records in a cyclic or "round-robin" manner. In this case the group size would be set to 1000 bytes, the stride would be set to 4000 bytes and the offsets would again be set to access disjoint regions, as shown in Figure 5-4.

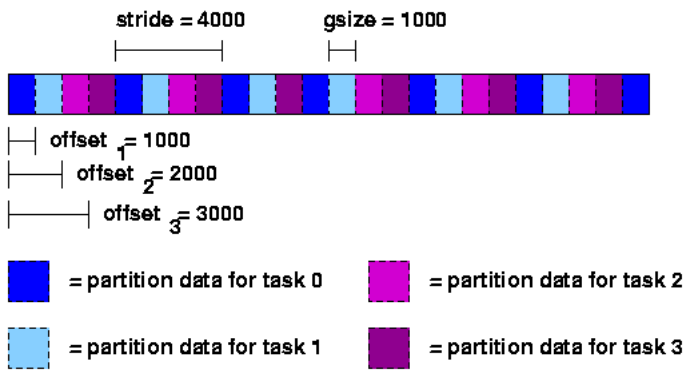


**Figure 5-4. Partitioning Example 2**

It is important to realize that setting the partition for one task has no effect whatsoever on any other tasks. There is also no reason that the partitions set by each task be distinct; one can overlap the partitions of different tasks if desired. Finally, there is no direct relationship between partitioning and striping for a given file; while it is often desirable to match the partition to the striping of the file, users have the option of selecting any partitioning scheme they desire independent of the striping of a file.

Simple partitioning is useful for one-dimensional data and simple distributions of two-dimensional data. More complex distributions and multi-dimensional data is often more easily partitioned using the multi-dimensional block interface.

## Using Multi-Dimensional Blocking

The PVFS multi-dimensional block interface (MDBI) provides a slightly higher-level view of file data than the native PVFS interface. With the MDBI, file data is considered as an N dimensional array of records. This array is divided into "blocks" of records by specifying the dimensions of the array and the size of the blocks in each dimension. The parameters used to describe the array are as follows:

D - number of dimensions
rs - record size
nbn - number of blocks (in each dimension)
nen - number of elements in a block (in each dimension)
bfn - blocking factor (in each dimension), described later

Once the programmer has defined the view of the data set, blocks of data can be read with single function calls, greatly simplifying the act of accessing these types of data sets. This is done by specifying a set of index values, one per dimension.

There are five basic calls used for accessing files with MDBI:

```
int open_blk(char *path, int flags, int mode);
int set_blk(int fd, int D, int rs, int ne1, int nb1, ..., int nen, int nbb);
int read_blk(int fd, char *buf, int index1, ..., int indexn);
int write_blk(int fd, char *buf, int index1, ..., int indexn);
int close_blk(int fd);
```

The **open_blk()** and **close_blk()** calls operate similarly to the standard UNIX **open()** and **close()**. **set_blk()** is the call used to set the blocking parameters for the array before reading or writing. It can be used as often as necessary and does not entail communication. **read_blk()** and **write_blk()** are used to read blocks of records once the blocking has been set.

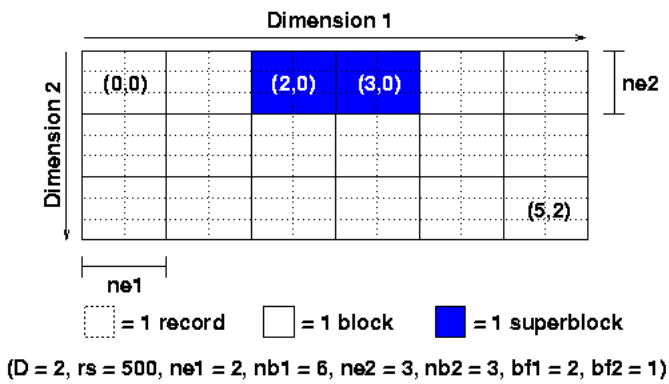


**Figure 5-5. MDBI Example**

In Figure 5-5 we can see an example of blocking. Here a file has been described as a two dimensional array of blocks, with blocks consisting of a two by three array of records. Records are shown with dotted lines, with groups of records organized into blocks denoted with solid lines.

In this example, the array would be described with a call to **set_blk()** as follows:

```
set_blk(fd, 2, 500, 2, 6, 3, 3);
```

If we wanted to read block (2, 0) from the array, we could then:

```
read_blk(fd, &buf, 2, 0);
```

Similarly, to read block (5, 2):

```
write_blk(fd, &blk, 5, 2);
```

A final feature of the MDBI is block buffering. Sometimes multi-dimensional blocking is used to set the size of the data that the program wants to read and write from disk. Other times the block size has some physical meaning in the program and is set for other reasons. In this case, individual blocks may be rather small, resulting in poor I/O performance an under utilization of memory. MDBI provides a buffering mechanism that causes multiple blocks to be read and written from disk and stored in a buffer in the program's memory address space. Subsequent transfers using **read_blk() and write_blk()** result in memory-to-memory transfers unless a block outside of the current buffer is accessed.

Since it is difficult to predict what blocks should be accessed when ahead of time, PVFS relies on user cues to determine what to buffer. This is done by defining "blocking factors" which group blocks together. A single function is used to define the blocking factor:

```
int buf_blk(int fd, int bf1, ..., int bfn);
```

The blocking factor indicates how many blocks in the given dimension should be buffered.

Looking at Figure 5-5 again, we can see how blocking factors can be defined. In the example, the call:

```
buf_blk(fd, 2, 1);
```

is used to specify the blocking factor. We denote the larger resulting buffered blocks as superblocks (a poor choice of terms in retrospect), one of which is shaded in the example.

Whenever a block is accessed, if its superblock is not in the buffer, the current superblock is written back to disk (if dirty) and the new superblock is read in its place — then the desired block is copied into the given buffer. The default blocking factor for all dimensions is 1, and any time the blocking factor is changed the buffer is written back to disk if dirty.

It is important to understand that no cache coherency is performed here; if application tasks are sharing superblocks, unexpected results will occur. It is up to the user to ensure that this does not happen. A good strategy for buffering is to develop your program without buffering turned on, and then enable it later in order to improve performance.

## Using ROMIO MPI-IO

ROMIO is an implementation of the MPI-IO interface which supports PVFS and includes a pair of optimizations which can be of great benefit to applications performing collective I/O.

See *the ROMIO web site*[1] for additional information.

## Notes

1. http://www.mcs.anl.gov/romio/

# Chapter 6. Porting Applications to Scyld ClusterWare

Porting applications to Scyld ClusterWare is generally straight-forward. Scyld ClusterWare is based on Linux and provides the same set of APIs and services as typical Linux distributions, along with enhanced implementations of cluster-specific tools such as MPI. There are a number of areas where Scyld ClusterWare is different, or more precisely where certain services aren't provided or respond in an unexpected way. Of course, most applications for Scyld ClusterWare will already need to be parallelized using MPI, PVM, or something similar. This chapter does not deal with parallelization issues, but rather places where the standard operating system environment might not be the same as on other Linux systems.

The first concern is system and library calls that are either supported differently or not at all under Scyld ClusterWare. The master node is an extended standard Linux installation, thus all non-parallel applications work as expected on the master node. On compute nodes, however, several things are different. In general, compute nodes do not have local databases. Thus the standard "lookup" name services may be configured differently. In particular, some of the name services that use static local files such as gethostbyname(), getprotobyname(), and getrpcbyname() do not have local databases by default. Under Scyld ClusterWare, all compute nodes are numbered from 0 to N and are identified by their number and a leading "dot." Thus, node 4 is ".4" and node 12 is ".12" and so on. The bproc library provides calls to look up the current host number and to obtain addresses for other nodes. On compute nodes, host name related lookups are implemented as such by bproc, thus gethostname() will typically return the node number and gethostbyname() can be used to get the IP address of a particular node. Note however that none of the compute nodes will be aware of the master node, its host name, or its network name. Generally, these calls should not be used.

Another major difference lies in the fact that compute nodes do not typically run many of the standard services. This includes services such as sendmail, ftpd, telnetd, rexecd, and rshd. Thus, the common means of running a program on a compute node using rsh does not work with a compute node without explicitly starting the 'rexec' daemon. Nor can one get a shell on a compute node using rlogin or telnet. Rather, bpsh provides a comparable means of accessing nodes in the cluster, including running remote programs.

If desired, a user can start a shell on a compute node via bpsh, but the default compute node configuration does not have access to utility programs such as cp, mv, or even ls. Thus, executing a shell on a compute node is not particularly useful. This is especially problematic when a shell executing on a compute node tries to execute a shell script. Again, unless the script has access to the binaries needed to run the script, it will fail. Thus one properly runs scripts on the master node and uses bpsh to run programs on the remote nodes.

In some ways, Scyld ClusterWare works pretty much like standard Linux, but there are details that might not be an issue with a standard system that must be considered with a Scyld system. For example, standard output will not automatically flush as often as on a native system. Thus, interactive programs need to call fflush(stdout) to ensure that prompts are actually written to the screen. License keys may also present an issue when running on a Scyld ClusterWare system. Depending on the details of the license server your software uses, problems may exist in distributing and verifying keys for the compute nodes.

In summary, there are a number of issues that need to be considered when porting an application to Scyld ClusterWare. Among these are:

- **getprotobyname** and **getrpcbyname** don't work
- **gethostname** and **gethostbyname** might work, but shouldn't be used
- **ftp**, **telnet**, **rexec**, **rlogin**, and **rsh** don't work
- Use **bpsh** to run remote programs, but not shells or shell scripts
- Flush standard output for prompts
- License keys may behave differently

In general, the best solution to these issues is to have code on the master node do lookups and pass the results to compute nodes, use bpsh for running programs on nodes, and consider issues such as scripts, standard output, and licenses.