BProc: The Beowulf Distributed Process Space

Erik Hendriks
Advanced Computing Laboratory
Los Alamos National Laboratory*
Los Alamos, NM
hendriks@lanl.gov

ABSTRACT

The Beowulf Distributed Process Space (BProc) is a set of Linux kernel modifications which provides a single system image and process migration facilities for processes running in a Beowulf style cluster. With BProc, all the processes running in a cluster are visible on the cluster front end machine and are controllable via existing UNIX process control mechanisms. Process creation is done on the front end machine and the processes are placed on the nodes where they will run with BProc's process migration mechanism.

These two features combined greatly simplify creating and cleaning up parallel jobs as well as removing the necessity of a user login to remote nodes in the cluster. Removing the need for user logins drastically reduces the mount of software required on cluster nodes.

Job startup with BProc's process migration mechanism is faster than the traditional method of logging into a node and starting the process with `rsh`. BProc does not affect file or network I/O of processes running on remote nodes so the vast majority of MPI applications will experience no performance loss as a result of being managed by BProc.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Network Operating Systems; D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Distributed Systems

General Terms

Design,Management

Keywords

Linux, cluster, single system image, process migration

*Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. LANL LA-UR-02-2583.

This paper is authored by an employee(s) of the United States Government and is in the public domain.
ICS'02, June 22-26, 2002, New York, New York, USA.
ACM 1-58113-483-5/02/0006.

1. INTRODUCTION

Traditionally, Beowulf style cluster computers have been set up as a collection of autonomous machines running Linux each with a local disk and its own Linux installation. Starting a parallel job on n nodes requires logging into n nodes and running some command there. Much of this complexity is normally hidden by communication libraries such as PVM [3] and MPI [4]. However, users still need to know enough about the system configuration to make sure that the libraries will be able to find the binary images to execute on each node.

Once a parallel job is started, monitoring and controlling the job is a problem. Viewing what a job is doing requires querying every node that a job is running on for process status. This is both time consuming and clumsy. The lack of readily available job status leads to runaway jobs being left on nodes because most users simply do not think to actively look for errant processes most of the time.

Current solutions to these problems [7, 2] all add additional layers of software. This additional software increases the complexity of the system (new programs for users to use, new daemons for nodes to run) and adds possible points of failure.

The Beowulf Distributed Process Space (BProc) is a Linux kernel modification which addresses the process creation and management problem by providing a single system image for all the processes in the cluster combined with a process migration mechanism to populate the cluster with processes.

BProc extends the existing process management infrastructure in Linux to include processes running on other machines. This means that all existing UNIX process management utilities work for entire parallel jobs. Users can see their entire parallel job state with `ps` on the front end machine. Killing entire jobs is as easy as sending a signal to a process group or running the standard Linux `killall` command.

BProc's process migration facility allows for fast and simple parallel job startup without having to worry about distributing binary images to nodes. This greatly simplifies some existing pieces of cluster infrastructure such as schedulers which now contain a large amount of code to create and control processes on many machines in a cluster.

2. OVERVIEW

In a BProc cluster, there is a single master (or front end) machine and many slave machines which receive and run processes from the master. All the processes distributed to

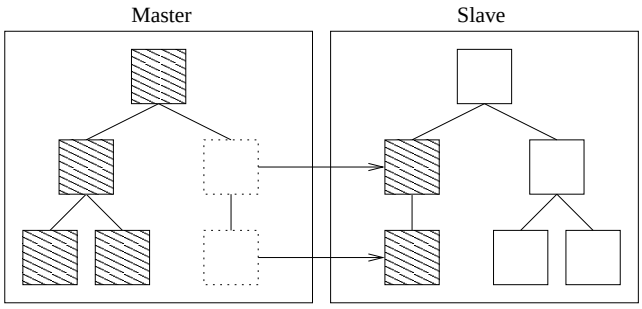

Figure 1: An example process tree spanning two machines. The shaded processes on the slave machine exist in the master's process space. The two dotted boxes in the master's process tree are the ghost processes which act as place holders representing the corresponding remote processes which are running on the slave node.

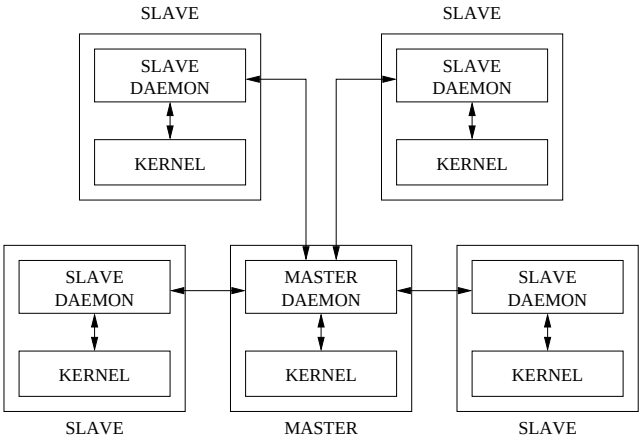

Figure 2: Component interconnection in BProc. All slave machines connect to the master daemon. The master daemon is the router at the center of the system. It routes BProc messages between the kernel layer and the daemons.

the slave machines are visible on the master in the same manner as any other process. Existing UNIX process viewing utilities will show remote processes without modification. Signals are forwarded transparently to remote processes. The usual parent/child relationships between processes are maintained regardless of process location.

The process migration facility simplifies the creation of remote processes in the cluster. With BProc, all the processes in a parallel job are started on the master and migrated to the nodes where they will run. Users no longer need to ensure that application binaries are available on remote machines. Process startup in libraries such as MPI becomes trivial. Once running, the usual UNIX process group rules apply so entire parallel jobs started in this fashion can be controlled by sending signals to process groups (i.e. Ctrl-C from the command line).

System management is dramatically simplified. Simple remote process control facilities eliminate much of the work currently done by schedulers. Fewer daemons (all of which represent a possible point of failure) are required. BProc only requires a single daemon to have a functioning slave node.

BProc is a Linux kernel modification which extends the familiar UNIX process semantics of a single machine to include processes which are running on different machines. Specifically, it allows a portion of one machine's process space to exist on one or more other machines. A process space is a pool of process IDs and the parent/child relationships that make up its process tree. Every instance of Linux defines a single process space. The machine that is distributing its process space is the master or front end. The machines accepting pieces of a master's process space are called slaves.

BProc does *not* provide a global process space. The master and slaves are not peers in BProc. The master node maintains control over the process space that it is distributing to the slave. The slaves do not lose their own process space because they are running a piece of the master's process space. A slave node will have local processes which are not managed by the master node and some processes that are. The daemon the slave must run in order to become a slave is a local process which the master does not control. A slave can accept processes from multiple masters at the same time.

The system image that users see is only completely present on the master node. The slaves only see the portion of the process space which is present on that machine. Processes running on the slaves can still access other parts of the process space but system calls such as `kill` may have to be forwarded to the master.

Figure 1 shows a simple example where a part of the master's process space is on another machine. The shaded processes exist in the master's process space. The slave node is running two remotely managed processes. The shaded processes would appear in the slave's process tree if you were to log into the slave node. They are tagged as being remotely managed. When the shaded processes running on the slave make process related system calls (like `getpid` or `kill`), they will be handled differently than the other processes on that machine. For example, the shaded processes on the slave cannot send signals to other processes on the slave since the other processes are outside the process space in which the shaded processes exist. If the shaded processes attempt to send a signal to one of the processes on the master node, the signal will be forwarded transparently.

The dotted boxes in the master's process space are place holders called ghosts which are inserted to represent remote processes. When a user looks at the process tree on the front end, they will see one ghost for every remote process. It is impossible for a process running on a remote node to disappear from the master's process tree without exiting. It is also impossible for a remote process to create child processes which do not appear in the master's process tree.

3. IMPLEMENTATION

The bulk of the BProc system is in the kernel modifications. Both the master's and the slave's kernels are modified to support BProc. On the master, there are the place holders in the process tree to represent processes running on remote machines. On the slave, the process ID (PID)

related system calls are modified to behave differently for remotely managed processes.

The in-kernel portions of BProc are tied together with pair of user space daemons. Figure 2 shows the connections between the kernels and daemons. The master daemon is the hub of the system connecting the master's kernel and the slaves together. The slaves do not send BProc messages directly to one another.

The complete machine state is split between the master daemon and the kernel on the front end. The kernel on the front end has a complete picture of the process tree and the master daemon knows on which node each remote process exists. If the kernel cannot handle something locally, a request is passed out to the user space daemon. The kernel code on the master and slaves only knows if processes are "here" or "not here". The user space daemons handle getting all requests to the proper receiver.

The slave daemons only see snippets of the process tree that involve the processes running on that slave. The slave daemon doesn't maintain any state of its own. It will simply forward most messages and handle some others such as signal delivery and `ptrace` requests.

The only pieces that BProc needs to keep coherent are the snippets of process trees that exist on slave nodes and the corresponding parts of the process tree on the master. There are no coherency requirements between slaves.

3.1 Master - Ghost Processes

The place holders in the master's process tree are called ghost processes. There is one ghost for every remote process. Ghost processes are lightweight kernel threads. They have no user level memory space or open files. They are real processes which means they can wake up and run but they never leave kernel space. Since ghosts are kernel threads they are guaranteed to not disappear until the remote process they represent exits. Using real processes to represent remote processes on the master allows BProc to reuse all the existing Linux process infrastructure.

Ghosts are idle the vast majority of the time but they do occasionally wake up to forward a signal or to perform a system call on behalf of a remote process. There are relatively few situations in which a ghost will be required to act. Table 1 shows all the system calls which may require some action by the ghost.

In order to ensure that the ghost process is always running with the same user, group and process group IDs as the remote process, the system calls which affect those IDs notify the ghost of any changes. The `ptrace` system call also requires help from the ghost in many cases.

Ghost processes are functionally equivalent to the real processes for all the process ID related system calls. Ghosts catch and forward any signals they receive to the remote processes they represent. Since they are kernel threads, ghosts can catch and forward all signals including SIGKILL and SIGSTOP without exiting or stopping. If the ghost is not involved in performing some system call for the remote process, then the process sending the signal will hand the signal directly on the outgoing message queue without waking the ghost.

Ghosts are also suitable targets for the `ptrace` system call. Debugging tools such as `gdb` [6] and `strace` can attach to ghost processes. All `ptrace` requests are transparently forwarded to the node where the real process exists.

System Call	Explanation
`fork`,`wait`	Both the `fork` and `wait` system calls modify the process tree. The master needs to be involved to make sure that the process tree on the master matches what the slave.
`exit`	When a remote process exits, the ghost exits with the same status.
`kill`	If the process being signaled does not exist on the same slave node, the ghost will perform the `kill` system call on behalf of the remote process.
`ptrace`	`ptrace` attach and detach make modifications to the process tree that must be reflected in the master's process tree. If the process being traced does not exist on the same machine, other `ptrace` calls (i.e. `PEEK` and `POKE`) will be performed by the ghost.
`set*id`	All the calls which modify a process's user, group, process group or session IDs require interaction with the ghost because the ghost process needs to reflect the change.

Table 1: System calls that may require action by the ghost process to complete.

Since ghost threads are real processes, they have real status of their own - running/sleeping, CPU time used, etc. The remote process is the one users are interested in so ghosts mirror the process status of the remote processes they represent. The `/proc` file system is slightly modified to present the mirrored status instead of the real status for ghost processes. The mirrored status information is updated on demand when a process accesses the `/proc` file system.

When a remote process exits, the ghost exits with the same exit status. The parent process can use a normal `wait` system call to pickup the exit status. The ghost can exit with the full range of possible exit conditions, including those indicating "killed by a signal" and "core dumped".

3.2 Slave Nodes - Process ID Masquerading

Slave nodes accept processes to run from the master node. When transplanting a process from the master to a slave, a few problems arise. The process ID of the process should not change when it moves but we cannot guarantee that a particular process ID will be available on the destination machine. Once the process exists on the remote machine, process ID related system calls such as `kill` should continue to operate in the context of the master's process space.

These problems are solved on the slave by putting a second process ID on processes received from the master node. When the slave receives a process, it creates a normal local process for it. This process exists in the slave's process space and will be assigned a local process ID as usual. The slave will then attach a second process ID to the process. The process ID related system calls (i.e. `getpid`, `wait`) are modified to translate or return alternate process IDs for these processes. Having the second process ID also indicates that the process's process ID related operations are to be managed by a third party.

For example, the `getpid` system call is modified to check for the second process ID. If it exists, it returns that ID, otherwise it returns the real process ID.

Figure 3 illustrates a fork request. If a process decides to `fork`, the child should be in the same process space as the parent. Creating the child process locally happens through the normal fork mechanism but before the `fork` returns and the child is allowed to run, the slave will contact the master node for a process ID to assign to the child.

In order to allocate a new process ID, the ghost process will perform a `fork` itself and return the new child process ID to the real process running on the slave node. Obtaining a new process ID this way has an important side effect - a new ghost is created on the master to represent the child process. This keeps the process trees on the master and slave synchronized. Resource limits governing the total number of processes are also enforced on the master. Once the new child (which is also a ghost) is created on the master, it will return its process ID back to the parent process on the remote machine. The master daemon makes note of the new process and its location when it sees a successful fork response. When the new process ID is received on the slave, the process ID is attached to the child process and the `fork` system call is allowed to complete. *There is no way for a process running on a remote node to create a process which is not visible in the master's process tree.*

The `wait` system call also involves the ghost process. It is essentially the inverse of `fork` (the ghost is cleaning up children instead of creating them) except that the slave does not need to wait for the result before continuing.

3.3 Daemons

Every machine runs a user space daemon which handles passing BProc messages on the network. The master and slaves communicate with each other using TCP. The BProc kernel code does not send and receive on the network directly except while transferring data during process migration. The daemons communicate with their local kernel using a special file handle.

The master runs a daemon which listens for new connections and maintains an open connection to each slave in the system. The slave nodes each run a slave daemon. A node becomes visible on the master when the slave daemon connects to the master. If the connection is broken for any reason, all the processes running on that slave daemon will be killed. Other slaves in the system are unaffected.

The master daemon is BProc's message router and all BProc message traffic runs through it. This is necessary because the master daemon is the only part of the system that knows where each process exists. It also needs to know when processes are moving from one place to another.

When a process moves from one slave node to another, the BProc message traffic runs through the master both for routing and so that the master can update the machine state to reflect the move. The data sent to migrate the process between the slaves does *not* pass through the master daemon. The slaves establish a new connection directly to one another to send the process data.

The slave daemon is essentially a stateless message pipe between the master daemon and the slave's kernel which sometimes gets involved in message handling. For example, the slave daemon will fork to create new processes on the node and will sometimes be involved in performing remote `ptrace` requests.

```
08048000-08049000 r-xp 00000000 03:01 288816 /bin/sleep
08049000-0804a000 rw-p 00000000 03:01 288816 /bin/sleep
40000000-40012000 r-xp 00000000 03:01 911381 /lib/ld-2.1.2.so
40012000-40013000 rw-p 00012000 03:01 911381 /lib/ld-2.1.2.so
40017000-40102000 r-xp 00000000 03:01 911434 /lib/libc-2.1.2.so
40102000-40106000 rw-p 000ea000 03:01 911434 /lib/libc-2.1.2.so
40106000-4010a000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Figure 4: Example memory space for `/bin/sleep`. This is taken directly from `/proc/`*pid*`/maps`.

3.4 Process Migration With BProc

VMADump (Virtual Memory Area Dumper) is the system used by BProc to take a running process and copy it to a another node. VMADump is part of BProc but is usable as a separate component. VMADump saves or restores a process's memory space to or from any file descriptor which supports `write` and `read`. In the case of BProc, the file descriptor is a TCP socket established between two nodes.

VMADump works by walking the list of memory mapped regions in the process and storing the contents of each region on the file descriptor. The CPU state, signal handler state and other miscellaneous process details are also sent.

The contents of the process's memory space is preserved with the following exceptions. Any regions of memory which were once shared will no longer be shared. Any regions `mmap`ed from a file that are not shared libraries will no longer be mapped from the file.

Most programs on the system are dynamically linked. At run time, they will use `mmap` to map copies of various libraries into their memory spaces. Since they are demand paged, the entire library is always mapped even if most of it will never be used. These regions must be included when copying a process's memory space. This is expensive since the size of the dynamically loaded libraries is often much larger than the programs they are linked against.

Figure 4 shows a simple example memory space. The total size of the memory space for this trivial program is 1089536 bytes. All but 32K of that comes from shared libraries - VMADump takes advantage of this.

VMADump can avoid copying these memory regions when migrating a process if we are willing to guarantee that the libraries that they are mapped from will be present on the remote machine. Instead of storing the data contained in each of these regions, it stores a reference to the file and the offset from which the region was mapped. When the image is restored, those files will be `mmap`ed to the same memory location.

In order for this optimization to work, VMADump must know which files it can expect to find where the process image is restored. Since VMADump does not know anything about binary formats and therefore cannot directly recognize libraries, it simply has a list of files which it presumes are present on remote systems. VMADump looks at what file every region is mapped from. If that file is on the list, VMADump will presume that the file is available on the remote machine and store a reference to the file.

Pages which are duplicated via the copy-on-write mechanism will be recognized and sent along with the file reference. It is not uncommon for the dynamic linker to modify

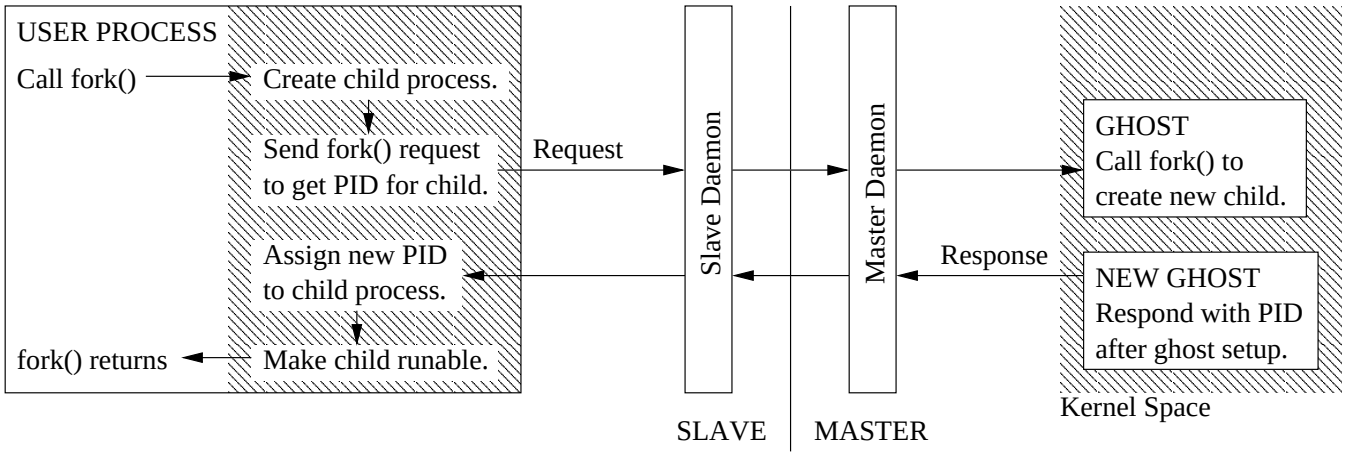

Figure 3: The `fork` system call. A user process calls fork. The child process is created on the slave node but not yet allowed to run. A fork request is sent to the user process's ghost to allocate a new process ID. The message is routed through the user level daemons and across the network. The ghost calls `fork` to create a new child on the master node. The new child process on the master (also a ghost) responds to the request once it has finished adding itself to the list of ghosts. The master daemon watches for successful `fork` responses and makes note of the position of the new remote process. Finally, the new process ID is attached to the child process on the slave node, the child is allowed to run and `fork` returns to the caller.

a few pages in a dynamic library (which was mapped with `MAP_PRIVATE`) and then mark the entire region read only. VMADump can perform this check from kernel space without faulting in all the pages in the mapped region.

A similar optimization avoids sending huge uninitialized data segments. Programs are frequently started on the front end and immediately migrated to the remote nodes via the `bproc_execmove` mechanism. If the program has a large, uninitialized data segment (BSS) the program's memory footprint could be huge but consist mostly of empty zero pages. Hundreds of megabytes of uninitialized data is not uncommon. VMADump checks for zero pages before sending them. The restoring process will initialize any page for which no data was received to zero, so zero pages are not sent at all. Checking for zero pages on a large BSS segment is relatively cheap since VMADump can perform the check without faulting in unallocated pages.

Process migration with BProc is non-preemptive and non-transparent. Once a process has been moved all its open file handles are closed. Any file related calls will be handled on the local machine. The process will see the local file system and network operations will reflect being on the new machine.

The only system calls which will *not* reflect the move are the process ID related operations such as `fork`, `wait`, `kill`, etc. The process ID of the process will not change. If the process had children, those will still be present in the context of `wait`. In short, the process will still appear to be part of the original process space.

Due to the extent of visible changes that happen during migration, migration is entirely voluntary. There is no mechanism for one process to cause another to migrate.

3.5 User API

User programs use BProc via a system call interface. The BProc API includes functions to get and set information about the nodes in the cluster and process migration calls. Since there is no third party process migration in BProc an application must use one of the following calls to migrate to a remote node.

`int bproc_move(int` *node*`)`
: `bproc_move` is the basic process migration call. The calling process is moved to *node*. It simply returns zero on success or -1 if there is an error. All the other process migration calls are internally based on `bproc_move`.

`int bproc_rfork(int` *node*`)`
: `bproc_rfork` is a normal `fork` followed by a `bproc_move`. The `fork` is done on the machine where the call is made. The reason `bproc_rfork` is relevant is that the calling process will not end up with a child process if the `fork` step succeeds but the `bproc_move` fails.

  C code for performing some task on a child process (without error checking) might be as simple as the following:

```
if (bproc_rfork(node_number) == 0) {
    /* Code for task on remote node */
} else {
    wait(0);  /* Wait for child to finish */
}
```

`int bproc_execmove(int` *node*`, const char *`*cmd*`,`
`char * const` *argv*`[], char * const` *envp*`[])`
: `bproc_execmove` combines the `execve` and `bproc_move` system calls. First, the normal `execve` is performed and then without returning to user space and letting

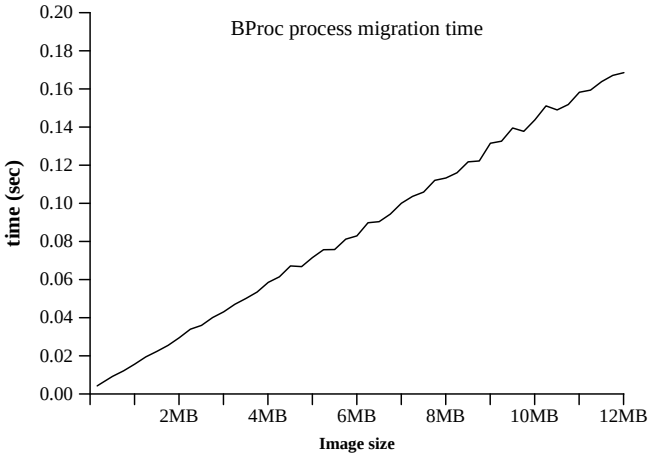

Figure 5: Process migration time. This graph shows the time required to migrate between two nodes in the cluster. The time shown was obtained by migrating to a node and back and then dividing the elapsed time by two. The image size is the amount of data that the BProc migration functions actually send, *not* the size of the program's binary file.

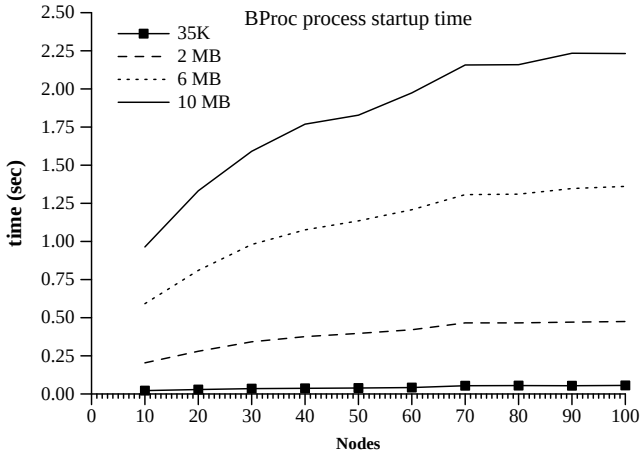

Figure 6: Process startup time using the `vexecmove` facility for various process sizes. This graph shows the amount of time required for BProc to place a copy of an arbitrary program on different numbers of nodes in the system. The process size is the amount data that the BProc migration functions actually send, *not* the size of the program's binary file.

the new program run, a `bproc_move` is performed to move the process to the remote node.

This mechanism is used to start arbitrary binaries on one node and let them run on another. BProc based clusters use this mechanism for starting MPI applications which need not be aware of BProc. This is also used for setting up nodes at boot time by migrating programs such as `mount` and `ifconfig` to the node.

4. PERFORMANCE

BProc only affects process ID related operations. All other system calls are handled locally. Once a process has moved to a remote machine, all the file I/O and network I/O is local to the node and just as fast as it would be without BProc.

BProc's single system image only covers the process related aspects of UNIX. Everything else is untouched on remote machines. Once a process has moved to a remote machine, all file I/O and network I/O is local to the node on which the process is running. Most parallel applications should experience no performance penalty when running on a slave node. Most anything that does not spend a significant amount of time in `fork`, `wait` or `kill` will experience no added overhead as a result of BProc.

On the master node, there are a large number of processes in the process table at any given time. Ghosts do very little on behalf of their remote processes which means that they are all asleep practically all of the time. Their presence will not affect the master's system performance as long as they sleep.

The only processes which should be affected by the presence of ghosts in the system are processes which are reading `/proc` entries belonging to a ghost process or processes performing `ptrace` requests on a ghost. Reading a ghost's `/proc` entries may trigger a ghost status refresh which takes some time to complete.

Since the master node is the message router in BProc, all BProc message traffic passes through the master node. Message traffic is typically very light. When remote processes are being created, the messages initiating remote process creation and returning success or failure run through the front end node. The data transfered to create the new process image will not necessarily pass through the front end.

Once a parallel application is started (i.e. a typical MPI job) there is very little message traffic in the system. A remote process doesn't communicate with the master at all if it does not call any of the system calls which require help from the master node (`fork`, `wait`, etc.)

The only remaining traffic is the occasional heartbeat and process status update requests. The frequency of remote process status updates is limited to once every five seconds. These updates will only be requested if there is an application (i.e. `ps`) actively looking at a ghost's process status.

4.1 Performance Results

The testbed used to test performance is an Alpha Linux cluster ("ed"). It consists of a master and 100 slave nodes made up of the following hardware and software:

- 5 x Compaq ES40 (4 CPUs @ 833MHz, 16GB RAM)
- 94 x Compaq DS10L (1 CPU @ 466MHz, 1GB RAM)
- 2 x API CS20 (2 CPUs @ 833MHz, 2GB RAM)
- Myrinet 2000 with GM 1.5.1
- Linux 2.4.18 with BProc 3.1.10

One of the ES40s served as the cluster front end. All BProc's communication used TCP/IP over Myrinet. The machine was unloaded during testing.

Figure 5 shows the time required for the basic process migration operation in BProc. The test program migrated

from the master node to a slave node and back and divided the elapsed time by two. The round trip time was averaged over 100 iterations. The minimum process size tested was 160K. This is approximately the minimum process image size for a dynamically linked program on these systems. The migration time for that size image averaged 4.2 msec. For larger process sizes, the time required to migrate is dominated by the time required to send the image. Note that the size of the image is the amount of data actually transmitted by the BProc process migration system, *not* the size of the program's binary file.

Figure 6 shows the time required to startup many processes using BProc's `vexecmove` facility for various process sizes. `vexecmove` transparently uses a tree spawn mechanism to more efficiently distribute the process image to all the nodes. This test measured the amount of time required to place a single copy of a process on each node. The program used for this test does nothing, just returns from main. Padding was added to this program to test larger process sizes.

As an experiment to see how many remote processes the master node could handle, a test was done starting 15000 remote processes on 100 machines (150 processes per machine). The test program used `vrfork` to place a copy of itself on every node and then used normal `fork` to create 150 copies of itself on each node. Once completed all these processes were visible on the front end using `ps`. This test averaged 2.3 seconds to complete. With that many processes in the system process viewing utilities such as `ps` became unwieldy — they open almost every file in `/proc` (there are several files per process) which makes them very slow.

5. RELATED WORK

Mosix [1] is a system which provides transparent process migration facilities for the purposes of load balancing. Transparent in this case means that a process will not have any indication that it has been moved to a remote node. The key difference between BProc and Mosix is that Mosix provides preemptive and completely transparent process migration. This is required if Mosix is to achieve its goal of load balancing a cluster without any cooperation from the applications running on it. Completely transparent migration is much more costly to achieve. In Mosix, the node where the process started (the process's home node) is involved in servicing many more system calls (file system calls, network I/O calls, etc.) than in BProc. This is necessary to maintain the transparency of process migration in Mosix. I/O intensive parallel applications may be reasonably supported in a Mosix cluster by starting each job on a different home node. This, however, defeats the purpose of having a single system image for job management.

BProc does not offer transparent or preemptive migration but adds much less overhead on runtime after migration. I believe this is a good trade-off for supporting MPI style parallel applications for which performance of system calls (particularly of I/O related system calls) is very important.

Condor [5] is another system which provides process migration. Condor is a cycle scavenging system with a checkpointing system which allows applications to be moved between systems. It's designed to tie together a large collection of heterogeneous UNIX systems with distributed ownership. Unlike BProc, it is implemented entirely in user space. This has the advantage that it can run on many different types of UNIX machines. It also does not require kernel modifications that system administrators may not be willing to make. This means that applications running under Condor need to be linked against special libraries to make use of its features but most people have access to the code they're running and this can be done with minimal effort. Condor also provides some system call forwarding for processes running on remote machines which makes the migration somewhat more transparent. This remote I/O does, however, take a performance hit much like Mosix.

BProc's migration mechanisms are implemented entirely in kernel space which means they can be used with arbitrary statically or dynamically linked binaries. This makes mechanisms such as `bproc_execmove` possible.

6. FUTURE WORK

VMADump represents a large portion of a possible checkpointing system. It will currently only save and restore single threaded processes. Adding support for multi-threaded programs and programs which share regions of memory will be necessary for creating a workable checkpointing system.

It remains to be seen how far BProc will scale. Having successfully managed many thousands of processes from a single master node, it seems that the Linux process infrastructure will be fine for a large system. So far the largest number of physical machines managed with a single BProc master is on the order of 256 machines. At that scale, there are no apparent issues with the master node — the obvious bottleneck in the system.

7. CONCLUSIONS

BProc provides simple process creation and control mechanisms for Linux based compute clusters. Remote process creation via process migration removes the need to make user's binaries available on all the nodes in a system. The monitoring capabilities combined with process migration removes the need for users to log into the nodes in a cluster. Without the login and local binary requirement, essentially all the software can be removed from the node reducing the number of points of failure and reducing the software management burden to zero. The ease of placing and managing processes on remote machines with BProc greatly simplifies existing pieces of common cluster software such as MPI and schedulers. Finally, the lack of any run time performance impact on the vast majority of MPI applications makes BProc a very attractive management system for compute clusters.

BProc currently supports x86, Alpha and PowerPC platforms. BProc is open source software licensed under the terms of the GPL and is available at `http://bproc.sourceforge.net` and `http://www.clustermatic.org`.

8. REFERENCES

[1] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of the Linux Expo '99*, pages 95–100, Raleigh, NC, May 1999.

[2] Greg Bruno and Philip M. Papadopoulos. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. October 2001.

[3] A Geist, A Beguelin, J Dongarra, W Jiang, R Manchek, and V Sunderam. *PVM: Parallel Virtual*

Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[5] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, Vol. 11, No.1, June 1997.

[6] R.M. Stallman. GDB manual. Second edition, Free Software Foundation, Inc., February 1988.

[7] The Open Cluster Group. OSCAR: A packaged cluster software stack for high performance computing. January 2001.