CSE598Z: DISTRIBUTED SYSTEMS

Process Migration via Remote Fork

Branden J. Moore
Department of Computer Science and Engineering
University of Notre Dame
<bmoore@cse.nd.edu>

Abstract

Utilizing multiple computers to complete a task in parallel offers many benefits over serial execution, however the programmatic interface to distributed computation is non-trivial at best. This paper introduces and examines the concept of a Remote Fork. Remote Fork systems allow a programmer to use a familiar parallel programming model, that of Fork, to easily harness the power of multiple machines. Many of the tradeoffs and technical challenges invoked in a Remote Fork system including such topics and Checkpoint/Restart, file descriptor and signal challenges, as well as fault tolerance, are discussed at length. One implementation of Remote Fork is described and evaluated to show near linear speedup is shown for a simple matrix multiplication benchmark utilizing Remote Fork.

1. Introduction

Standard programming models are not well suited to distributed computing. The language semantics of most standard programming languages do not provide for the execution of code on remote processors. Typically when distributed computing is shoe-horned into standard programming semantics, the abstractions break down[18]. Distributing computation across disjoint systems increases the chances for failure in ways that local semantics are not prepared to handle. Remote Procedure Calls (RPC)[4] are one mechanism to provide for distributed computing with "close-to-local" semantics. One of the main disadvantages of the RPC mechanism is that the code to be executed on a remote machine must provide a "well-known" service. The programmer wishing to use the RPC system is limited to running the services provided, and cannot "at will" decide what code should be executed remotely. This paper presents a mechanism for programmers to execute their code on remote processors in a model that is as familiar as `fork()`. This mechanism, known as a "Remote Fork" or `rfork()`, allows programmers to create a copy of the current process and invoke it on a remote processor as easily as a regular same-machine `fork()`.

The Remote Fork mechanism presents a familiar, but slightly modified programming model to the application programmer. This allows programmers to treat collections of disjoint computer systems as a single, large SMP (Symmetric MultiProcessor). Standard `fork()` based parallel programming is a well-known and understood programming model, but it is limited to working on a single system image. Remote Fork allows this same programming model to be extended to work across system boundaries. This can make certain parallel processes much faster through the use of external processing resources.

When a process issues an `rfork()`, a snapshot or checkpoint image is taken of the current process and shipped to the target remote computer. The remote machine takes this image and restores it as a running process, while the original parent continues execution. This is essentially the same manner in which standard `fork()` produces children. Child processes are almost exact copies of the parent, and as such must retain open file descriptors and inherit the processing environment of the parent. Communication between a parent process and its child, be it via file descriptors or signals, presents the paramount difficulties in designing a valid Remote Fork system.

This paper explores the design space and tradeoffs involved in the development of a Remote Fork system. It then introduces one implementation of a Remote Fork system which strives to maintain most semantics of standard `fork()`. It accomplishes this by using a local Child-

Pseudo-Process (CPP) and a Remote Fork Daemon (RFD) on the target computer. Reads and writes to open file descriptors are forward from the child to the parent and visa-versa via the RFD and CPP. Signals are forwarded as well, as are exit status messages.

With a simple matrix-multiplication benchmark, an average speedup of 83% of linear can be ascertained through the Remote Fork system presented. The main tradeoff comes in the cost of checkpointing processes with a large memory footprint, versus the amount of parallelism obtained. While computation time achieved very near linear speedup, the overhead of executing a Remote Fork overwhelms the speedup for large memory, small computational codes. Simple optimization techniques can be applied to reduce the large overhead of the `rfork()` call, however these overheads are still quite significant.

The remainder of this paper is organized as follows; alternate mechanisms for harnessing distributed computational resources are discussed in section 2. A discussion of the design requirements and tradeoffs are in section 3, followed by a description of one implementation of a Remote Fork mechanism in section 4. This implementation is evaluated and concluding remarks are made in sections 5 and 6, respectively. Appendix A contains an informal specification of the protocol used in this implementation of Remote Fork.

2. Related Work

Process Migration is by no means a new topic, and has been studied at length in many places, from automatic load-balancing mechanisms[15] to the cycle scavengers[17]. Most systems which utilize a process migration scheme concentrate on transparent migration. This removes from the programmers and/or the users the need to specify when or where the process will migrate. The Sprite Operating system[6] accomplishes this goal by integrating process migration into the internals of the operating system and tunneling all I/O of remote processes back to the originating computer. The Charlotte system[3] does this as well, but rather than doing an implicit tunnel for communication, all system communication is done via message passing. Therefore, remote communication is as trivial as local communication. [15] attempts to take transparent migration to the next level by distributing all system-level objects, such as pipes and timers.

Remote Procedure Calls (RPC) has been a popular mechanism for remote execution of code[4]. RPC requires that procedures to be executed remotely be compiled separately from the rest of the program. These remote procedures are then explicitly created as services on remote hosts. These services can be utilized by any authorized client. RPC attempts to mimic the semantics of local procedure calls, but adds an extra class of failure error codes. Through the use of “stub” functions in a library, it is not necessarily apparent to the the programmer where remote execution may occur.

Transparent process migration has the advantage of allowing for more automated load balancing and cycle usage, but at the same time removes from the programmer the ability to design the program in a manner which would best benefit from migration or parallelism. Message Passing systems such as LAM-MPI[5] are designed to have the developers split their programs logically around their parallel partitions. This tradeoff decreases run-time flexibility, but may allow for faster execution. Remote Fork allows the programmer the flexibility to choose how many, and which hosts to execute on at run-time, as well as giving the benefit of programmer-knowledgeable partitions for speed.

The ability to move a process from one system to another typically requires homogeneous systems. Almost any change from one system to another can cause a process to fail. These changes can range from architecture and operating system changes, to simply different compilations of libraries. Few attempts have been made at allowing for heterogeneous process migration[1, 7, 8]. Gulwani and Tarachandani developed a mechanism to allow for process migration in a very structured manner in heterogeneous systems by augmenting the compiler of a C program. The programmer can specify specific locations where migration can take place. These places are consistent across systems, and as such re-invocation on another system does not depend on the original executable or text segments.

Another large challenge when designing a process migration system is to keep the semantics of the system as close to those of a non-migratory system. It has been challenged that no “good” process migration system with UNIX-like semantics can be run entirely from user-space[2]. Systems such as Sprite[6], Locus[19], MDX[15], DEMOS/MP[11] and Charlotte[3] all base the process migration schemes in the Operating System. However, it has been shown that it is not the case that operating system-level support is required for process migration. [5, 16, 17] have managed to develop pure-user-level systems for process migration. The advantages to staying out of the operating system run the gamut from security to practicality. In many cases, it is not possible for a modification to be done to operating system, such as when administrative access is not available or when such actions would have adverse effects on the rest of the system.

One of the major steps in a Remote Fork is that of checkpointing and restarting processes. One of the first Checkpoint/Restart (CR) mechanisms was presented by Pikner in 1971[12]. This system was designed for a COBOL based machine, and while not directly usable for most systems today, presents many of the first arguments for a CR system. CR today sees many implementations, from LAM-MPI[14] to virtual machines[1] to hardware controlled systems[10]. The mechanisms to perform a transportable checkpoint and restart that checkpoint are well established and will be discussed in section 3.1.

Remote Fork systems have been studied in the past[16, 19], but most such systems are integrated with a specific operating system such as LOCUS, or did not provide sufficient abilities. The system developed by Smith and Ioannidis does not allow for the transfer of open file descriptors or the ability to `wait(2)` on a child process. This paper attempts to show that a user-level remote fork mechanism with acceptable semantics is possible. Waldo et. al, show that expecting standard local semantics in a distributed system is foolish at best, as distributing a system brings to the expected abstraction new ways of failure, both total and partial, in ways the abstraction is not prepared to handle[18].

3. Mechanisms & Tradeoffs

The design of a Remote Fork system, like any complex system, requires certain tradeoffs to be evaluated and decided upon, based on the manner in which the system is intended be used. One of the first questions to ask when developing a Remote Fork system is whether it is even the most appropriate system to be using. If the intended use is to immediately follow the `rfork()` with an `exec()`, perhaps a Remote Exec would be better, for it would have much less overhead due to not needing a Checkpoint/Restart mechanism. Many computer systems already have a Remote Exec mechanism available, via `rsh` or `ssh`.

The return value from `fork()` is the process id of the child (in the case of the parent), or zero (in the case of the child). Somehow, `rfork()` needs to maintain this, or a similar mechanism to determine which instance of the process is the child, and which is the parent. Also, the parent should receive some bit of information or identifier so that it can control the child; sending signals to it, for instance. One common mechanism to allow for this ability is the use of a shadow process, or child pseudo-process. This process is the result of a standard `fork()`, and resides locally to act as a stub for the remote process. `rfork()` returns the process id of the shadow process to the parent, while the shadow process acts to forward information to and from the remote child process.

The security of a Remote Fork system must be evaluated before it can be considered a viable service to use. It would not be advisable for unauthenticated clients to be able to invoke whatever code they wish on a system. Authentication and encryption methods should be considered an integral part of any complete Remote Fork system.

Other tradeoffs to consider include how to accomplish the Checkpoint/Restart, how to handle open files and signals, and what to do about failure. The following subsections explore each of these areas in turn.

3.1. Checkpoint & Recovery

Checkpoint/Restart mechanisms come with an entire suite of details to keep in mind, from where to do the checkpointing, to how to restore, to what to do about open files. This paper explores some of the options for these below.

Checkpointing can be done from a variety of places. The easiest place to checkpoint a process is from the kernel. Only the kernel has complete access to all components which make up a process. As discussed above however, it is not always possible or acceptable to modify the kernel of the operating system. Therefore, there must be methods to checkpoint a process from outside of the kernel. One mechanism to accomplish out-of-kernel checkpointing is to just cause the process to dump core. It is possible to take a core file from a process, and restart that process. This is most easily seen, and commonly done with a debugger. One of the disadvantages of this method is that generating a core file is often destructive to the original process. For example, core files are typically produced through the default action upon receipt of SIGSEGV or SIGQUIT. Both of these actions also result in the termination of the process. Also, only regular files and directories can be restored via this mechanism; pipes, sockets and other block devices cannot be restored. Another potential way to checkpoint a process is to do so from "inside" the process. This requires that the program be linked with a checkpoint library, so it cannot be used for on applications that cannot be re-linked. Internal-checkpointing is beneficial for it can be customized to a specific purpose. Some of the difficulty of a purely internal checkpointing is the recording of processor state. It is easy to write to a file the address space of the process, but it is much more difficult to dump the current processor state, for the act of writing the state changes the state. One way around this is to use a standard `fork()` call, and use the `ptrace(2)` system to get access to the stopped child process, and dump its state.

Once a checkpoint image has been created, restoring the process from that image also has its challenges. For example, unless the checkpoint has been stored as a valid executable binary file on the system, the user cannot just execute the image. Rather, there must be some way to load the image into a process space and begin execution. One design question here is where to load the process? There are two choices; either load into the current process space, similar to `exec(2)`, or into another process space created by forking a child. It can be quite difficult to replace the current process image, but it can be done through the use of temporary stacks and `longjmp`. This creates the appearance of directly invoking a checkpoint image. The other mechanism, forking a child and modifying that address space, has benefits for Remote Fork systems by continuing the environment of a child being forked from a parent. One of the challenges with placing a new process environment over an existing one is mapping memory segments into the correct position. Processes do not react well to code and data moving during execution. Libraries and other memory-mapped files must be re-mapped to the exact same location, or else the process will fail.

This leads to the next question in designing a Checkpoint/Restart mechanism. Dynamically linked processes and dynamically loaded libraries (via `dlopen(2)`) provide for a large increase in the complexity of the task. Dynamically linked processes can be checkpointed and restarted fairly trivially assuming that 1) restart is done using the same libraries as checkpoint, and 2) it is deterministic and static as to where the libraries will be loaded into memory. For clarity, if libraries `libc.so` and `libm.so` are to be linked to process `foo`, upon each invocation of `foo`, each library must be loaded into the exact same memory address. Memory-mapped files and dynamically loaded libraries must be noted specially, for an 'exec' on the original executable file will not allocate space or map these files into place. Care must be taken to restore these during process restart code.

A final question that needs to be addressed is that of how to handle open file descriptors. This question can have different answers based on how the CR mechanism is to be used. For a Remote Fork system, the programmer can make different assumptions than a CR mechanism being used for checkpointing and later restart. For example, file descriptors that are open upon `rfork()` *should* remain open after the fork. Named pipes and sockets included. Files that were open before a checkpoint that is being restarted later really has no guarantees about the state of the file descriptors, they may all be stale. The semantics for handling this case must be described. Other questions include: should an open file remain an open file? should there even be support for file descriptors? what about pipes and sockets?

3.2. File Descriptors

The semantics behind files in a Remote Fork system are quite complex. With a regular `fork()`, all open file descriptors are duplicated into the new process. Open files remain open, with file pointers in the same location. Pipes and sockets are open as well with the same permissions. When invoking a process on a remote system, duplicating file descriptors is no longer a viable option. Unless there is a global filesystem, attempting to open a local file is most likely not going to result in a valid file descriptor. Also, pipes and sockets will have no connection to the remote machine. One way around this is to define the semantics such that any open file descriptors will be closed on `rfork()`, but this is not a palatable solution. What is commonly done is to use a shadow process on the originating computer, and "tunnel" file descriptor activity between the systems. The advantage to tunneling is that all file descriptors are automatically duplicated upon the local `fork()`, and so files, pipes, sockets, etc. remain open. The disadvantage is that on the child end, the semantics of the file descriptor must change to that of a socket or a named piped. Therefore, if a seek is to be issued it must be "caught" and emulated, or else the `read()` or `write()` will return an error.

3.3. Signals & Process IDs

To some extent, the semantics of signaling and process identifiers have the same complexity as open file descriptors in the case of Remote Fork. For example, `fork()` returns the process ID of the child process. This can then be used by the parent to send signals to the child, and wait for an exit status. In the case of a remote process, either functions such as `kill(2)` and `wait(2)` must be trapped to "fake" local operation, or a shadow process must be used. If a shadow process is used, it must know to pass on signals that it receives. However, there are some signals that cannot be trapped by the process: SIGSTOP and SIGKILL. There must be some mechanism for the child process to detect that the shadow process has received one of these signals, and act upon that information. Otherwise standard POSIX semantics have been violated. One potential mechanism is to use two-layers of shadow. The process ID returned to the original process could be the grandchild of it, not just the child. The middle process can use `waitpid(2)` to watch the grandchild's status. It can then detect what signal caused the process to halt, and the appropriate actions can then be taken.

Another interesting bit is the use of `getppid(2)` by the child. This function returns the process ID of the parent process. In a Remote Fork system, this process ID will not reflect the actual parent. There should be a solid mechanism for either trapping this function, or forwarding signals sent by the child to the parent back to the parent. The latter option makes a solid argument for either a shadow parent process on the remote host, or perhaps a daemon which handles communication. The actual parent of the child process on the remote machine is also responsible for collecting the return status of the child when it terminates to avoid polluting the system with zombie processes. The shadow parent or daemon should then forward the exit status back to the parent in a meaningful manner. For example, the shadow child on the local machine can choose to exit with the exact same exit code as the remote child did.

3.4. Fault Tolerance

The laws of probability imply that as components are added to a system, the probability of failure increases. This is quite true of distributed systems, and Remote Fork is no exception. Interestingly, Remote Fork systems are in a position to easily recover from failure. Through the judicious use of checkpoint images, applications can recover from many failures without a significant loss of work.[1] Remote Fork systems already embed the ability to take checkpoints at `rfork()` time, and this can easily be extended to allow for periodic checkpoints.

Keeping checkpoints around can be expensive in terms of disk space, for all memory segments of a process must be stored. On a 32bit system, this implies that checkpoints can reach up to 4GB in size. As a result, the management of checkpoints must be considered carefully. The Condor project recommends the use of checkpoint servers to hold in stable storage the progression of the computation.[13] This solution can, however, lead to an "all eggs in one basket" situation. Holding checkpoint images on the computation nodes avoids the situation where any one system (other than the "head" node, where computation starts) can bring about the failure of the entire computation. Without very careful planning and work, the "head", or local, node of computation cannot suffer failure without causing complete program failure. This is because of the usual mechanisms of forwarding back from the remote nodes reads and writes to local file descriptors. It is not advisable to store checkpoint images on the child nodes, for all benefits are lost if the child node disappears. However, due to the size of the checkpoints, it can be quite costly in terms of storage and bandwidth to store the checkpoint images on the parent node.

Recovery from the failure of a remote child process can be quite complex, especially if the program running contains any side effects such as writing to a file or communication with other processes. Unless the side effects of executing the child process are purely idempotent, mechanisms must be developed to undo any actions which a failed child did. Systems such as Timewarp[9] are designed with rollback in mind, yet without careful programming, most applications will lead to incorrect results when communication is repeated.

One potential mechanism to avoid side effects is to require no "communication" other than the presentation of results at the end of computation. In this manner, the parent process need not be informed of a failure, and the child shadow process can re-invoke the child on another system. Alternatively, as long as communication is only one-way, the shadow process can choose to buffer all communication, and only deliver it to the parent when the child completes processing. This is analogous to a "commit" phase. Storing buffered I/O can be accomplished on either the child node or the parent node, for a computation is not assumed to "complete" without the final commit. Buffering I/O for some processes is not advisable, for they may produce very large amounts of data. Processes such as these can use a mixed mechanism of buffering I/O up to a certain amount, and checkpointing the process once the I/O buffer is full. The buffer can then be sent to the parent, and future failures can be restored from the new checkpoint. This is not always an acceptable solution, however. For example, many parallel applications require some amount of communication, be it for access to results from computation to be used as input to further computation, or even just to establish locks around shared resources.

Another challenge associated with distributed systems and failure is that of debugging. It is a rare program that runs perfectly the first time it is executed. Remote Fork adds to the challenge of debugging parallel applications by requiring a cross-system debugging mechanism to be in place. A well designed Remote Fork system will return to the parent the `wait(2)` status of a child process when it exits, so it is possible to know with what exit code or what signal caused the child to terminate. "`printf(3)` debugging"

[1] A "significant loss of work" is a metric to be determined by the programmer

can also come in very handy when attempting to debug the application. However, it should be noted that this too only works if the I/O from child processes is not buffered until the condition of successful completion. An alternate method is to shell connect to the remote nodes, and attach `gdb` to the child process, however this can be difficult to accomplish, and a Remote Fork system does not guarantee shell access is available to the remote nodes.

4. Implementation

This sections describes one implementation of a Remote Fork mechanism. This implementation is not yet complete, but is usable at this point. Mechanisms for recovery from failure and reading from open files have not yet been implemented, but reading from sockets or pipes, and writing to any file descriptor has been completed. The Remote Fork Daemon currently only supports one child at a time.

The architecture of this Remote Fork system can be seen in Figure 1. There are two libraries involved in the Remote Fork system. One for checkpointing (`libcr`), which is generic and can be used outside of the Remote Fork system, and one to export the `rfork()` interface (`librfork`). When the parent process (P1) calls `rfork()`, a local child process is forked off which checkpoints the process and then `exec`'s the `cpp` program. `cpp` handles all communication with the remote computer, and passes I/O and signal information between the remote computer and the parent. On the remote computer lives a Remote Fork Daemon (RFD) which is responsible for invoking and restoring the child process, as well as tunneling communication from the child back to the parent computer. There is a protocol which the RFD understands, called the "Remote Fork Protocol" (RFP) which is described in the appendix, section A.

The RFP requires that the client (the "parent process", or `cpp` in this case) establish a connection and transfer a copy of the executable that is being run to the RFD, followed by a checkpoint of the process. By sending the executable to the RFD, there is no requirement of a global filesystem. However, this does require that either the executable be statically linked, or that the libraries on the remote host be the same as those on the local host. Once the RFD has the executable and the checkpoint, the checkpoint is modified to point file descriptors to named pipes which are connected to the RFD. In this manner, all open file descriptors remain open, and the RFD can catch and tunnel I/O back to the `cpp` which forwards the data to the parent process. Calls to `open(2)` made from the child are not forwarded back to the parent computer, but rather open files locally. This was done in an effort to avoid re-writing libc. When the child process ceases to execute, the RFD collects the `wait(2)` status and forwards this back to the `cpp` which exits with the same status as the child did, thus allowing the parent process to use `wait(2)` in the normal manner and receive accurate results.

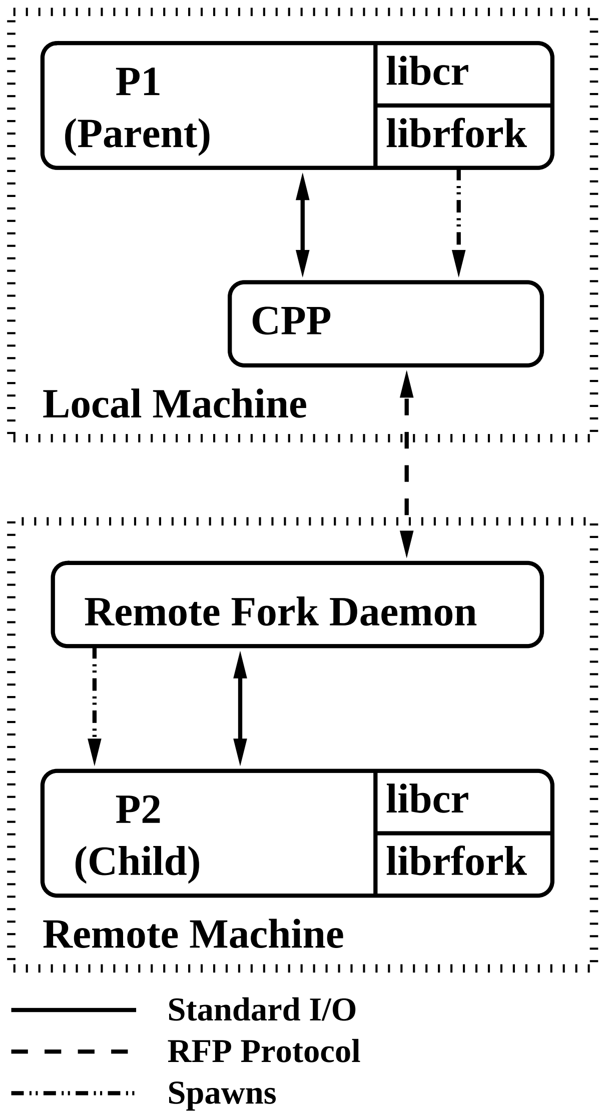

Figure 1. Architecture of a Remote Fork system

Below contains information on the implementation of the Checkpoint Restart library in section 4.1, followed by the Remote Fork Daemon in 4.4 and the programmatic interface and shadow process (`cpp`) in 4.5.

4.1. Checkpoint Restart (`libcr`)

The Checkpoint/Restart library, `libcr`, has no direct connection to the Remote Fork system. Rather, it is a self-contained library which can be used by any application wishing to utilize it. It presents two basic functions, `int cr_checkpoint( const char *fname)` and `pid_t cr_restart( const char *fname)`. To the programmer, `cr_checkpoint()` simply creates a checkpoint image, and returns. `cr_restart()` acts like `fork()` in that it creates a new process and returns the process id of the child process.

This CR library supports static and dynamically linked libraries, but does not support `mmap(2)`'d files or dynamically loaded libraries opened by `dlopen(3)`. The checkpointing will appear to work fine, but upon restart, these sections will contain stale data, and not be linked to the correct files. As for file descriptors, all types of descriptors are supported for checkpointing, with current seek offsets stored in the image. However, unnamed pipes and sockets will be closed on restart, for it is not possible to re-connect these file descriptors.

4.2. Checkpointing

The checkpointing process of `libcr` is fairly simple. First, the process forks to create another copy of itself which it can use to examine. The child process sets the “PTRACE_TRACEME” flag, and sends itself the STOP signal. This will be the re-entry point for restore.

The parent process, which is the actual process, does the bulk of the work in checkpointing. First, it waits for the child to stop and connects to it via the `ptrace(2)` facility. It then gathers the current memory break, which reflects the size of the process heap. It then allocates a structure to hold all of the data which will go into the checkpoint image. The break must be recorded before the other steps, for they can require the use of allocating memory which would change the size of the heap.

The parent then uses the `ptrace(2)` facility to record the register state of the child process. Reading the allocated memory segments into the checkpoint image structure is done by reading out of `/proc/<pid>/mem` and writing that data to the image. A listing of the allocated memory segments can be found on Linux systems by reading `/proc/<pid>/maps`. After the memory segments have been recorded, file descriptor information is obtained. A listing of open file descriptors, and what they point to is found in the `/proc/<pid>/fd/` directory. That directory contains symbolic links named after the file descriptor number, and pointing to the target of the file descriptor. This information, coupled with what can be garnered from `fcntl(2)` is used to record all pertinent information about the state of open files in the process.

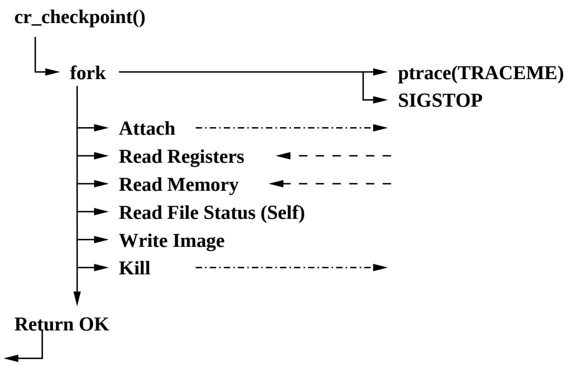

Figure 2. Checkpoint Timeline

Once the file descriptor information has been recorded, the process can then create and open the checkpoint image file and dump into it all of the recorded information. The file is then closed, the child process killed, and `cr_checkpoint()` returns.

4.3. Restarting

Restarting from a checkpoint requires a bit more work than checkpointing. To completely replace an address space can be very tricky, for it is not possible for one process to allocate memory segments inside of another process, even through the powerful `ptrace(2)` facility. However, it is also not a trivial exercise to attempt to restore processor state to a known value from inside of a process. The technique described here uses a combination of in- and out-side of process procedures to accomplish a successful restart.

The first step to restarting is to read in and parse the checkpoint image. The image contains all of the necessary information to completely restore the address space, file descriptor table and processor state of the process. Once this has been read, a `fork()` is called. Both processes now have a copy of the checkpoint information. The child process closes off any currently open file descriptors and attempts to re-open all file descriptors contained in the checkpoint. Regular files, directories and named pipes can all be opened without difficulty. Unnamed pipes and sockets are not re-opened, and a warning is issued. Subsequent attempts to read or write to those file descriptors will lead to an EBADF error.

Once the file descriptors have been restored, the child attempts to `exec(2)` the original executable. This allows

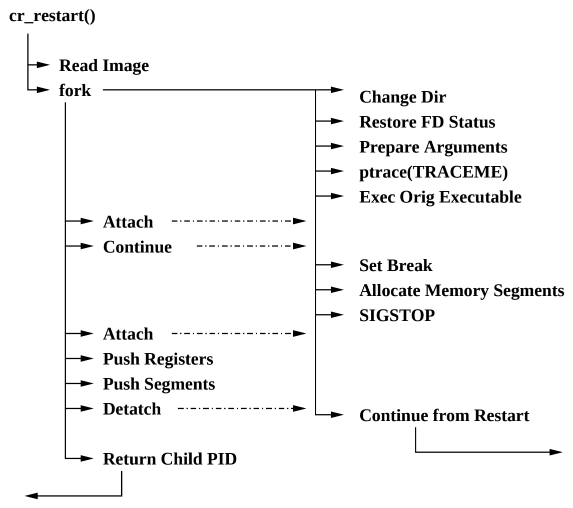

Figure 3. Restart Timeline

the operating system to take care of flushing the address space and loading shared libraries into the correct locations. When a process includes the header for `libcr` and links against the library, the '`main`' function is changed to '`app_main`', and a new '`main`' function is used. The startup code for the application now includes a check for the existence of an environment variable `$CR_RESTART`. If this is set, then the program will read from the argument list the location of the memory break (size of the heap), and a listing of memory segments that it should allocate. Using `brk(2)` to set the break, and thus allocate all the memory that resides on the heap, and `mmap(2)` to allocate other memory segments, the child establishes the address space. At this point, the child sends itself a 'SIGSTOP' and waits for its parent to continue it.

The parent process in the restart waits for the child to stop, and attaches via `ptrace(2)` to the child. It then uses the 'PTRACE_SETREGS' and 'PTRACE_SETFPREGS' functions to restore the register state of the child. The contents of the child's memory is restored through the use of 'PTRACE_POKEDATA.' The parent detaches from the child and returns the process id of the child to the calling function. The act of detaching from the child causes the child to continue execution. At this point, the entire state of the process has been restored and the child continues as if it had just received a 'SIGCONT' after the 'SIGSTOP' it received in the checkpoint procedure.

4.4. Remote Fork Daemon

The Remote Fork Daemon (RFD) runs on any computer which will be used as a target host for a child process. It's purpose is to start processes and monitor processes received from checkpoint images, and the funnel I/O to and from the client. In this section, 'client' will refer to the '`cpp`' program acting as a shadow child for the parent process. The `cpp` program is described later.

The RFD listens for connections on TCP port 30500 for perspective clients. When one connects, the client sends the executable, followed by the checkpoint image to restore. The RFP writes both of these to the local filesystem. It then modifies the checkpoint image to suit the Remote Fork system. Currently open files descriptors are changed to point to named pipes which the RFD creates and connects to. This is done so that activity on those file descriptors is forwarded on to the RFD, which in turn can forward data to the client. Once the image has been appropriately modified, the RFD asks `libcr` to perform a restart on the checkpoint image, using the executable previously supplied by the client.

Once the child has been restarted, the RFD goes into a loop of waiting for data on a file descriptor. If data appears on one of the connection pipes, the RFD packages this data according to the Remote Fork Protocol (RFP), and sends it off to the client to handle. If the client has sent data to the RFD, it will forward it on to the appropriate connection pipe to deliver it to the right file descriptor of the child. The client can also request that a signal be sent to the child. When a command of this accord is received by the RFD, the signal is sent to the child.

On each cycle through the idle loop, the RFD executes a non-blocking `waitpid(2)` on the child process Id. If the child has exited, the RFD will send the status information off to the client. When this has been completed, the RFD performs cleanup actions, closing and removing the named pipes, as well as removing the checkpoint image and local copy of the executable. The RFD then returns to await another connection.

4.5. `librfork` & `cpp`

`librfork` itself is fairly simple. It consists of a signal handler and the `rfork()` function. `rfork()` is accomplished by first performing a `fork()` to establish the local shadow process. The parent then waits for the child process to send it a signal, or for the child to exit. If the child exits, then the Remote Fork was unsuccessful, and `rfork()` returns an error code which corresponds to the exit status of the child process. This code can be used to determine what failed in the attempt.

The child process starts by recording its current process id. It does this to be able to determine, after calling `cr_checkpoint()`, if it is the original process or the restarted one. A checkpoint is then taken. At this point, the process id is gathered again. If the two process Ids are identical, then it is assumed that the process is the shadow process, if they are distinct, then it is the restarted code.

The restarted process will just return with no error at this point. The shadow process executes the `cpp` program, passing the names of the executable, the checkpoint file, and the remote host as arguments. The name of the executable is garnered by reading where the symbolic link `/proc/self/exe` points to. Using this mechanism, rather than argument zero of the program, guarantees returning the full path to the actual executable file which the loader had invoked.

When `cpp` is invoked, it first builds a map of all currently open file descriptors. This is used to verify write commands from the RFD, and for use with `select(2)` to watch for new data. Once a this map has been created, `cpp` connects to the RFD server on the remote host. Upon successful connection, the executable is sent to the RFD, followed by the checkpoint image. If the RFD returns an OK to these requests, it can be assumed that the child is running correctly on the remote host. `cpp` then sends a signal to the parent process, informing it of a successful start. If it was not successful, `cpp` exits with an error code indicative of the reason for failure.

Once the parent has been informed of a successful start, `cpp` establishes a signal handler for all potential signals it may receive (and that it can catch). It then enters a main loop of watching for data on the open file descriptors, and on the connection to the RFD. If data becomes available on an open file descriptor, it forwards the data on to the RFD. When the RFD sends a command to `cpp`, it must determine the appropriate action. For writes, the accompanying data is forwarded to the corresponding file descriptor. If the RFD claims that the child closed a file descriptor, `cpp` will do the same. When the RFD reports that the child has exited, `cpp` disconnects from the RFD and. It then analyzes the return status, and applies the same signal to itself - after resetting the signal handlers to SIG_DFL - or exits with the same return code.

If `cpp` receives a trap-able signal during execution, it crafts a message to the RFD containing the signal number received, and sends this message off to the RFD. In this manner, `cpp` acts as a forwarding agent for most all communication between the parent process and the remote child.

5. Evaluation

In this section, an evaluation of the effectiveness of the implementation of Remote Fork this paper describes is done. The benchmark used here is a naive parallel matrix-multiply algorithm. The benchmark receives a list of hosts and two matrix sizes on the command line. It then allocates space for all three matrices ($A * B = C$). Matrices A and B are filled with random integers. The computation space is divided up amongst the number of hosts listed on the command line. The parent process records the current time, and then begins calling `rfork()` to each of the children. The child processes compute their space of the C matrix, and send back to the parent the resulting data. Child processes are spawned after the matrices have been initialized, but computation does not begin until after all child processes are ready. At that time, a signal is sent to each child to tell it to begin computation. This allows for the measurement of the overhead of calling `rfork()`, as well as measuring the advantage to computation alone. Other than sending the results back to the parent process via unnamed pipe, there is no communication between processes. The overhead of communication tunneled through the RFD and `cpp` processes is not explicitly measured, however it is logical to expect that the overhead involved in communication is non-trivial, due to the many context switches and network latency incurred.

The benchmark was run on a spectrum of square matrix sizes, and on up to 8 computers. The same benchmarks were also run using standard `fork()` on a single system, which happens to be a dual processor. The benchmark was also run for each size in a serial manner to establish a baseline.

Figure 4 shows the results of running the matrix multiply benchmark on two square matrices of size $1024x1024$. There are three bars for each processor count. The first, labeled ‘fork’ is representative of the benchmark using standard `fork()` rather than an `rfork()`. Thus, the gains of parallelism are limited to 2 processes. It should be noted that all benchmarks were run on dual-cpu systems, which leads to the benefit for 2 processes on the local system, with severely diminished returns for additional processes.

The next bar, labeled ‘rfork’ represents the time required for the benchmark to complete when using `rfork()`. There is an obvious trend upwards as the number of processors increase. This is due to the large overhead of calling `rfork()`. On this size of a benchmark, it required about 6.5 seconds for each `rfork()` to establish the remote process. The third bar, labeled ‘rfork, computation’ represents the time required for computation and commu-

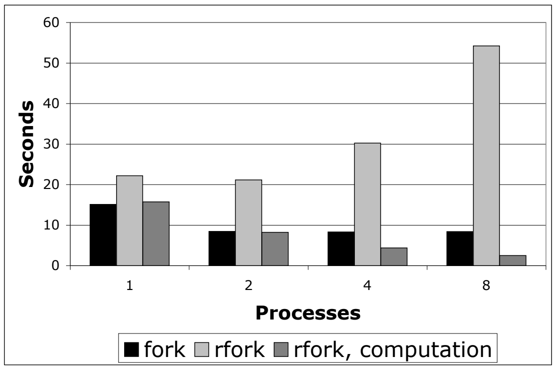

Figure 4. 1024x1024 * 1024x1024 Matrix

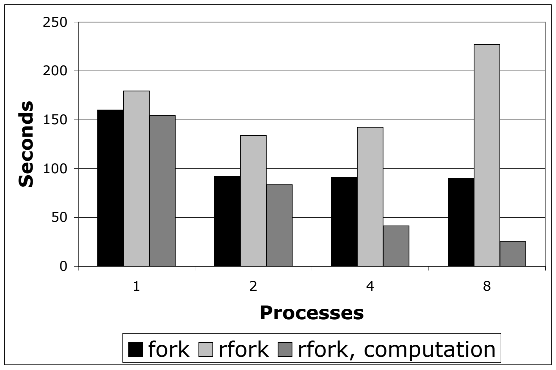

Figure 5. 2048x2048 * 2048x2048 Matrix

nication only. Timing was taken from the moment all processes were ready for computation until the time computation finished. As can be seen, the time for actual computation sees a benefit for each additional process. In fact, the speedup is close to linear, with an average speedup of $1.62x$ for a doubling of processes, or 81.2% of linear speedup.

Figure 5 reflects the same benchmark configurations run, but for matrix sizes of $2048x2048$. In this case, the computational time has increased approximately 10-fold over the $1024x1024$ matrices. As before, the local-system benchmark receives a large benefit with the addition of an extra process, but after that the returns are negligible. For this benchmark, the time required for a single `rfork()` is 25.25 seconds. With a constant per-process overhead of close to 33% of the single-process time, `rfork()` is still not well suited for this class of codes. However, after subtracting the overhead of the Remote Fork, speedup is again around 81% of linear.

By increasing the size of the benchmark again, this time to matrices of a size $4096x4096$, a clear advantage to using `rfork()` is made apparent. Figure 6 demonstrates this. Due to the large amount of computation and communication of results, there is no noticeable gain for using `fork()` on a matrix multiply of this size. However, there is a very clear gain in using Remote Fork here. Although the per-process remote fork overhead is 110 seconds, no longer is the overhead the primary consumer of time. Computation has reached the point where a single process takes over an hour to compute the answer. The speedup in computation is 86.6% of linear, and the overall speedup of 62.8% of linear, including overhead!

This demonstrates that a Remote Fork system shines the

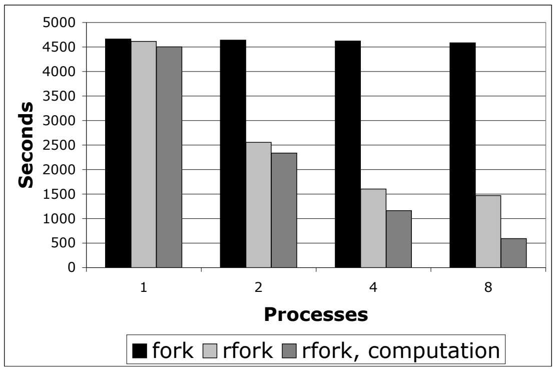

Figure 6. 4096x4096 * 4096x4096 Matrix

most when the computational requirements of the dataset are much more costly than the memory requirements. It should also be pointed out that the Remote Fork implementation was not optimized in any manner. Many trivial optimizations could be applied to the process to increase the gains seen here. One of the large time-consumers of checkpoint-restart code is in the recovery of the process's address space. The `ptrace(2)` system only allows for the reading and writing of one word (4-bytes on the x86 architecture) at a time. Restoring the state of file descriptors and cpu state is quite fast, but recovering large address spaces takes a long time. Checkpointing is quite fast through the use of reading `/proc/<pid>/mem`, but writing to that file is not available due to security concerns.

6. Conclusions & Future Work

This paper introduced the concept of a Remote Fork for the purposes of computation in a distributed environment. It discussed many of the technical challenges and tradeoffs associated with implementing a Remote Fork mechanism on Linux. Remote Fork is essentially an extension of a Checkpoint/Restart system, with connections between processes that may reside on separate computers after restart. The ability to communicate between these processes provide many technical challenges. Failure of a component in a distributed system must be expected and handled in a clean fashion. The use of CR allows for the potential ability to recover from failure without undue loss of effort.

Remote Fork has been shown to be a valid mechanism for distributed computation. However, there are significant detractors to such a mechanism. The overhead from checkpointing a processes can be quite significant, and for many codes, this overhead does not outweigh the benefits gained from running on parallel resources. However, for sufficiently complex computational codes, Remote Fork does allow for a large speed boost. Many optimizations could be applied to reduce this overhead and grant a even larger speed improvement. For example, rather than taking a new checkpoint for each `rfork()`, in a situation where many processes will be spawned in a row, such as in the matrix-multiply benchmark, one checkpoint could be taken and used for multiple `rfork()` calls.

Another potential benefit for large-memory codes would be an implementation of copy-on-access. Rather than transferring the contents of the entire address space at `rfork()`, establish a system where page-faults are trapped and at that time, pages from the checkpoint would be mapped into memory.

[18] gives a solid argument for avoiding such systems as `rfork()`, which attempts to emulate local-system semantics in a distributed environment. The draw of Remote Fork lies in extending the familiar `fork()` to a distributed environment, however with a small investment into learning alternate paradigms such as MPI, parallelism can easily be harnessed in a manner which exposes the distributed nature of the application, thus allowing for stronger semantics and better mechanisms for handling failure.

Future efforts applied towards a Remote Fork facility should be applied to the handling of failure, primarily. Before a Remote Fork facility can be considered "production quality" it must have its semantics well defined, and an appropriate mechanism to recover from failure. Rollback and buffered I/O have both benefits and detractions, so the key is to find a set of semantics that work well with the desired computational environment.

A. Remote Fork Protocol

- The Remote Fork Daemon (RFD) listens on TCP port 30500.
- The Remote Fork Protocol (RFP) uses TCP and connections are persistent. A broken TCP connection will result in any remote child processes being killed by the RFD, for it will be assumed that the Child Pseudo-Process (CPP) on the client has been killed.
- Communication inside the full-duplex TCP stream is based on messages.
- Messages contain two components, the Header and the Data. The message structure is defined below (Figure 7). The messages do not have a set size, but the headers are constant-sized, and give the length of the data segment.

```
+---------+-----------...... ------+
| Header  |   Data                 |
+---------+-----------...... ------+
```

Figure 7. Layout of an RFP message

A.1. RFP Message Header

The RFP message header has 4 fields, as described below. The values for these fields are described in tables 1 and 2. The layout of the header is shown in Figure 8.

- **Message ID:** 32-bit message identifier. Used to connect commands with responses. Client sends unique positive values. Server responses contain the same Message ID as the client sent. Server-originating messages contain a unique negative value.
- **Command:** 16-bit Command to the target, or a response code. See Tables 1 and 2 for a list of possible commands.
- **Tag:** 16-bit tag. Used as an argument to the Command.
- **Data Size:** 32-bit value to inform the receiver of the length of the data segment following the header, except in the case of `RFP_DEATH`, where it contains the `wait(2)` exit code of the child process.

```
 0                               32
+-------+------+------+-------+
|Message ID....................|  4 B
+-------+------+------+-------+
|Command........|Tag............|  8 B
+-------+------+------+-------+
|Data Size.....................| 12 B
+-------+------+------+-------+
```

Figure 8. Layout of an RFP header

Table 1. Client ⇒ Server Commands

Value	Name	Description
0	`RFP_EXEC`	Sends the executable for the later `RFP_RESTORE`
1	`RFP_RESTORE`	Invoke a checkpoint image in the child. Tag ignored. Data contains checkpoint image.
2	`RFP_SIGNAL`	Send signal ‘TAG’ to child.
3	`RFP_WRITE`	Write ‘DATA’ to file descriptor ‘TAG’.
4	`RFP_CLOSE`	Close connection, killing child if necessary.
5	`RFP_NOP`	NO-OP. Must be sent one minute after the last message sent (other than `RFP_CLOSE`) to keep the connection alive. If the RFD has not received any communication for 2 minutes, it assumes the client has disappeared and cuts the connection, terminating the child in the process.

Table 2. Server ⇒ Client Commands

Value	Name	Description
0	`RFP_OK`	Sent in acknowledgment of the SUCCESSFUL completion of a command from the child. Message ID contains the Message ID of the command from the child.
1	`RFP_NO`	Sent in acknowledgment of the FAILED completion of a command from the child. Message ID contains the Message ID of the command from the child.
2	`RFP_DEATH`	Child exited. ‘Data Size’ is the `waitpid()` ‘status’ value.
3	`RFP_WRITE`	Child wrote ‘DATA’ to file descriptor ‘TAG’.
4	`RFP_READ`	Child wishes to read from file descriptor ‘TAG’.
5	`RFP_CLOSEFD`	Child closed file descriptor ‘TAG’.

References

[1] A. Agbaria and R. Friedman. Virtual machine based heterogeneous checkpointing. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.

[2] C. Allison. Wanted: an application aware checkpointing service. In *Proceedings of the 6th workshop on ACM SIGOPS European workshop*, pages 178–183. ACM Press, 1994.

[3] Y. Artsy and R. A. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[5] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[6] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[7] F. B. Dubach and C. M. Shub. Process-originated migration in a heterogeneous environment. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990's*, pages 98–102. ACM Press, 1989.

[8] S. Gulwani and A. Tarachandani. Platform independent checkpointing of a c-program in execution. Technical report, University of California, Berkeley, 1999.

[9] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 77–93. ACM Press, 1987.

[10] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 79–88. ACM Press, 1990.

[11] B. P. Miller, D. L. Presotto, and M. L. Powell. Demos/mp: the development of a distributed operating system. *Softw. Pract. Exper.*, 17(4):277–290, 1987.

[12] H. Pikner. Programmed restarts. In *Proceedings of the 1971 26th Annual Conference*, pages 13–27. ACM Press, 1971.

[13] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop)*, volume 1162, pages 140–154. Springer, 1996.

[14] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In

Proceedings, *LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[15] H. Schrimpf. Migration of processes, files, and virtual devices in the mdx operating system. *SIGOPS Oper. Syst. Rev.*, 29(2):70–81, 1995.

[16] J. Smith and J. Ioannidis. Notes on the implementation of a remote fork mechanism, 1989.

[17] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[18] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[19] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 49–70. ACM Press, 1983.