Adding the Easy Button to the Cloud with SnowFlock and MPI

Philip Patchin, H. Andrés Lagar-Cavilla, Eyal de Lara, Michael Brudno

Department of Computer Science, University of Toronto

http://sysweb.cs.toronto.edu/snowflock

Abstract

Cloud computing promises to provide researchers with the ability to perform parallel computations using large pools of virtual machines (VMs), without facing the burden of owning or maintaining physical infrastructure. However, with ease of access to hundreds of VMs, comes also an increased management burden. Cloud users today must manually instantiate, configure and maintain the virtual hosts in their cluster. They must learn new cloud APIs that are not germane to the problem of parallel processing. Those APIs usually take several minutes to perform their VM-management tasks, forcing users to keep VMs idling and pay for unused processing time, rather than shut VMs down and power them on as needed. Furthermore, users must still configure their cluster management framework to launch their parallel jobs.

In this paper we show that all this management pain is unnecessary. We show how to combine a cloud API – SnowFlock – and a parallel processing framework – MPI – to truly realize the potential of the cloud. SnowFlock allows users to fork VMs as if they were processes, occupying in sub-second time multiple physical hosts. We exploit the synergy between this paradigm and MPI's job management to completely hide all details of cloud management from the user. Maintaining a single VM and starting unmodified applications with familiar MPI commands, a user can instantaneously leverage hundreds of processors to perform a parallel computation. Besides making use of cloud resources trivial, we also eliminate the cost of idling – VMs exist only for as long as they are involved in computation.

1. Introduction

The availability of cheap multicore hardware has allowed the deployment of large clusters suitable for use via Internet connections. This has led to the emergence of commodity computing services and the cloud paradigm, provided by services such as Amazon's Elastic Cloud (EC2) [2]. In the cloud model, users buy processing time and run their virtual machines (VMs) in the large computing facilities of the provider, scaling their applications without the need for an investment in physical hardware ownership or maintenance.

These large computing facilities are ideally suited to the execution of massively parallel applications. A common mechanism for implementing such parallel processing applications is the Message Passing Interface (MPI) standard. MPI provides an API to simplify the message passing between application processing nodes, and provides a process management environment to orchestrate the launching of application processes across hosts. However, in a cloud environment, users need not only worry about MPI. They must also set up their cluster of VMs by managing the instantiation, configuration and maintenance of the cluster members. To do so, they must learn cloud APIs [3] that are unrelated to the problems of parallel processing. Start-up of VMs in cloud or cluster environments can take "minutes" [2], which generally forces users to rarely shut down VMs and keep them idle instead. Idling of VMs incurs a financial cost and may potentially degrade the quality of service experienced by the user due to consolidation. The management burden of a parallel framework is thus compounded with that of administering the cloud resources being used.

SnowFlock [22] allows virtual clusters to be instantiated in an impromptu manner on a physical cluster by cloning a single previously-started master VM. An application that has been designed to work in the SnowFlock environment can expand its processing footprint in sub-second time, and then reduce it again when the computation is finished. SnowFlock thus ensures that no excess resources are wasted, making use of commodity computing resources such as EC2 much more economical. Further, SnowFlock provides stateful cloning: since all VMs are essentially identical to the master, configuration and application state are automatically inherited. This greatly simplifies any management burden, since users need only maintain a single VM, and can rely on stateful cloning to swiftly propagate the configuration to other VMs as needed. Finally, SnowFlock presents a low runtime overhead, making its use practical for processor-intensive parallel tasks.

This paper shows that SnowFlock can be coupled with a parallel processing framework like MPI to make use of cloud computing resources trivial. We modified MPI to leverage SnowFlock and clone its VM to multiple copies when starting an MPI job. We rendered the management of cloud resources invisible to the user, by integrating it within a fully compatible version of a well-known parallel API. Users of this system only need to maintain a single VM, and install their usual applications without any modifications. SnowFlock-MPI lets users leverage the MPI commands that they are experienced with to instantaneously and efficiently expand the computing footprint to the number of desired

• **sf_request_ticket (n)**: Requests an allocation for n clones. Returns a ticket describing an allocation for $m \leq n$ clones.
• **sf_clone(ticket)**: Clones, using the allocation in the ticket. Returns the clone ID, $0 \leq ID \leq m$.
• **sf_exit()**: For children ($1 \leq ID \leq m$), terminates the child.
• **sf_join(ticket)**: For the parent (ID = 0), blocks until all children in the ticket reach their sf_exit call. At that point all children are terminated and the ticket is discarded.
• **sf_kill(ticket)**: Parent only, immediately terminates all children in ticket and discards the ticket.

Table 1: The SnowFlock VM Fork API

processors. Our SnowFlock-MPI implementation is a modification of MPICH Version 1.2.7, an MPI implementation from the Argonne National Laboratory [4], which is fully compatible with the original implementation and supports code written in C, C++, and Fortran. SnowFlock and the SnowFlock-MPI library described here are open-source software freely available at our project site [29].

We describe several common applications which we have tested with this mechanism, and provide experimental data. While our experiments reveal a runtime overhead due to the centralized nature of our MPI management architecture, we propose solutions to remove this temporary inefficiency. Further, we believe this overhead to be negligible when compared to the ease of use of cloud resources our paradigm introduces. Further, we believe that SnowFlock's model of dynamic and stateful VM instantiation can enable novel extensions to the MPI paradigm. We discuss in this paper two of these extensions: computation resizing, and automatic application data distribution through read-only shared memory.

We open the paper with a description of SnowFlock, followed by a description of MPI and our changes to it. We then describe the applications we used for our tests, and discuss the evaluation results. Before concluding the paper we discuss future directions for MPI enabled by the SnowFlock model, and review related work.

2. SnowFlock

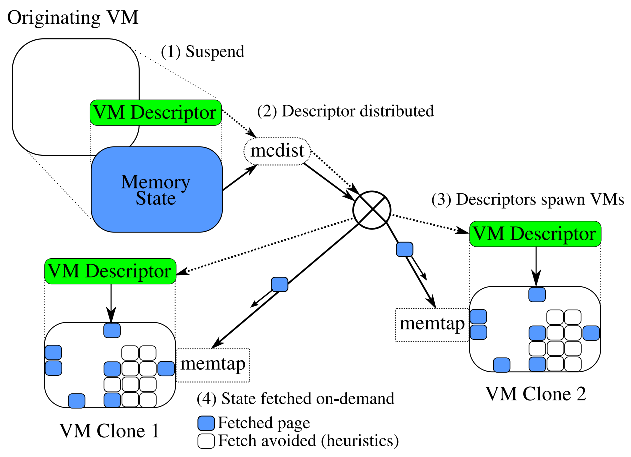
This section gives a brief description of SnowFlock's API and implementation. For more details, please see [22]. SnowFlock implements a *VM fork* primitive. Once a call to VM fork succeeds, a number of identical VMs will have been instantiated on several different physical hosts. Forking of VMs is swift, parallel, and scalable. Each VM is added to the same virtual network connecting all clones to their parent. VMs are differentiated by a clone ID, a positive integer that VMs can use to determine their role in a computation – the parent's ID is always zero. Each VM has its IP address automatically reconfigured as a function of the ID. VMs can communicate with their parent and siblings, but not with VMs from other users or with external hosts, unless explicit rules are added to a NAT and firewall engine presiding over the virtual network.

The VM fork primitive is exposed through a simple API, which we illustrate in Table 1. A VM must call sf_request_ticket to obtain an allocation ticket before cloning. This ticket indicates how many VMs can actually be created, and is a function of cluster management policies such as quotas and billing. A ticket also indicates on which physical hosts cloned VMs will be instantiated, although that information is hidden to VMs. The sf_clone call operates with the ticket as input, forks the VM and returns the clone ID. Upon return of the clone call, a VM will have been added to the virtual network with its unique IP address, and can perform network communications with its peers. The master VM can use the sf_join or sf_kill operations to synchronize with its children and eliminate them.

SnowFlock is implemented by extending the Xen [5] virtual machine monitor (VMM) version 3.0.3. The API is provided to applications via C and Python bindings, and through shell scripts. The API implementation posts requests to a shared memory interface (the XenStore). A SnowFlock daemon executes in domain 0 – Xen's administrative and privileged VM – and listens for API calls on the shared memory interface. There is one SnowFlock daemon per cluster host; the daemons form a distributed system that orchestrates the task of cloning VMs. Figure 1 illustrates SnowFlock's VM cloning mechanism.

SnowFlock achieves swift, parallel, and scalable VM cloning by attacking the problem of large transmissions of VM state. Unlike techniques such as live migration [9] that fully and eagerly replicate an entire VM, SnowFlock transmits state on-demand. Swift cloning is achieved by synthesizing a VM descriptor, a small file composed of VM metadata, virtual processor (vcpu) registers, and the contents of some special pages of memory, including the pages that make up the page tables used by the x86 MMU hardware. A VM descriptor is produced in milliseconds, has a size typically under a megabyte, and is distributed via multicast to all hosts where a cloned VM will execute. The VM descriptor is used to spawn a cloned VM that will have most of its disk and memory image empty. The entire process occurs in less than a second, and is independent of the number of hosts on which VMs will be cloned: SnowFlock can clone a VM to 32 hosts in 800 milliseconds.

SnowFlock provides memtap, a memory-on-demand mechanism that fetches pages of a cloned VM's memory as they are being accessed. On-demand replication of memory prevents large transmissions of VM state, but could adversely affect performance. SnowFlock has two optimizations that keep runtime overhead low for most workloads. First, the guest kernel page allocator is augmented to detect and prevent fetches of pages that will be immediately overwritten. This results in substantial savings of memory transfers by exploiting the common case of clones creating new application state, which tends to overwrite the contents of little-used or free memory pages. Second, we use mcdist, a subsystem



Condensed VM descriptors are distributed to cluster hosts to spawn VM replicas. Memtap populates the VM replica state on demand, using multicast distribution. Avoidance heuristics reduce the number of necessary fetches.

Figure 1: SnowFlock VM Replication Architecture

that provides optimistic multicast distribution for pages of memory. While delivery of a page of memory is guaranteed to the requester, it is also simultaneously multicast "for free" to all other clones. By exploiting locality patterns among clones performing the same task on different data, multicast effectively prefetches memory to most hosts, and leverages parallelism in the network hardware.

SnowFlock provides the same techniques (fetching state on demand, heuristics and multicast) for the cloning of a VM's disk state. Cloned VMs are involved, in general, in tight computational loops, and thus rarely formulate disk requests that exceed their buffer caches. Our techniques have thus proven appropriate for these workloads.

3. Integrating MPI and SnowFlock

This section describes the MPI architecture and the changes that were required for it to work in the SnowFlock environment. We based our work on the MPICH [4] library developed by the Argonne National Laboratory.

3.1 Standard MPI Architecture

The MPICH architecture requires the user to set up a management daemon (the mpd process) on each node which will be used to run MPI applications. These management daemons are persistent, and hence require the machine (virtual or otherwise) upon which they run to remain active. They form a one-way ring topology through which communication can take place. When an application is to be run, the mpirun program is used to initiate the execution as follows:

```
mpirun -np num_procs program arguments
```

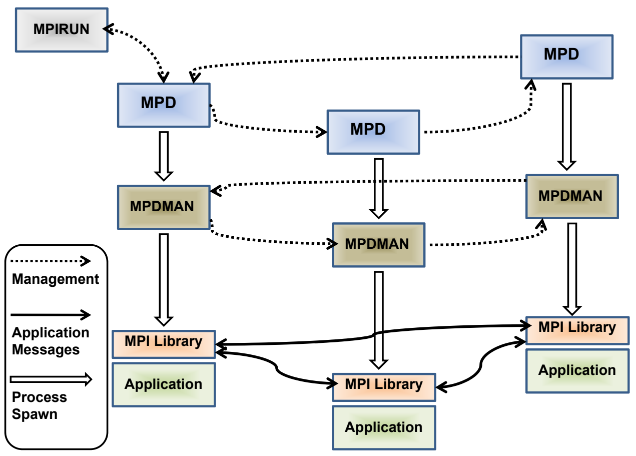
where num_procs is the desired number of MPI processes or workers. mpirun attaches to the mpd process on the local node, causing it to communicate with the rest of the ring and start up an application-specific mpdman process on each node to be used. There is one mpdman process per worker requested, and the program and arguments passed to mpirun are propagated through the mpd ring to each mpdman process. The mpdman processes themselves connect forming a secondary one-way ring topology, one that only exists for the length of this specific execution. The mpdman processes then each start an application process. One important assumption is that the application binary is available in the same path in all nodes.

Multiple mpdman and application processes can be spawned per node. For instance, if a user requests 32 processes and there are only 16 nodes, two mpdman/application process pairs will be spawned per node. When spawning the application binary, an mpdman sets up a number of environment variables which will be used by the MPI library upon application process start up. These variables include: the MPI rank, an integral number uniquely identifying each worker in the MPI computation; the file descriptors for a loopback network connection that the application process can use to talk to its managing mpdman; and the file descriptors for three loopback connections which are used to propagate the stdin, stdout, and stderr streams through the mpdman ring back to the mpirun process.

Application processes link against the MPI library and are expected to call immediately upon startup MPI_Init to parse the environment variables set up by mpdman. The application process can then perform calls such as MPI_Comm_rank to determine its rank, or MPI_Comm_size to find out about the total number of workers spawned. Throughout execution, workers transmit information to one another via the MPI_Send and MPI_Recv calls. Workers talk to one another using point-to-point connections, and identify each other by rank; workers have no knowledge of network coordinates. A connection between two workers is brokered by their mpdman managers communicating over their separate ring. The mpdman managers are also involved in tasks such as broadcast (barrier or fence) operations, application termination, etc. This architecture is depicted in figure 2. Within a node, the mpdman processes use UNIX signals to notify local application processes that they have been sent information which they must process.

3.2 Adapting MPI to SnowFlock

Given that SnowFlock allows the impromptu creation and subsequent merging of a virtual cluster of VMs, the standard MPI architecture described above does not fit well within this model, as it requires long-lived management processes on a cluster with a well-known configuration. We modified MPICH to allow the mpirun process to control the creation of an appropriately-sized cluster for an application run using SnowFlock's API. A user provides exactly the same command invocation as he or she would do with standard MPI. The num_procs argument passed to mpirun is used to de-



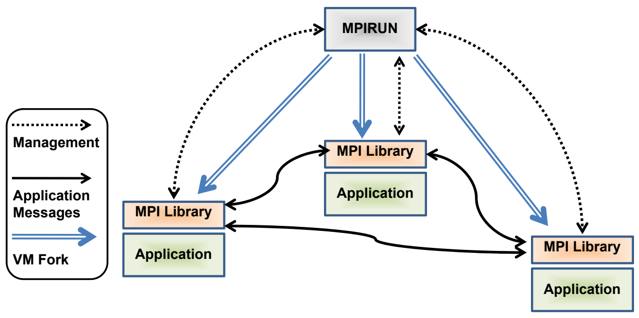
The standard MPI architecture involves a persistent one-way ring of management processes (mpd), and another one-way ring that manages each application run (mpdman). Control messages travel through the rings, while workers exchange application messages point-to-point.
Figure 2: Standard MPI Architecture

termine the number of VM clones to be created. Because SnowFlock can replicate multiprocessor VMs, mpirun performs a two stage forking process: first, it uses VM fork to span multiple hosts with VM replicas, and then it uses process fork to span the processors allocated to each VM. For example, a computation requiring 64 processes could use 16 VMs with 4 processors each.

Because the IP addresses of the cloned VMs are computed deterministically out of clone IDs, communication between workers can be established without the need for mpdman-like brokering. Following our example, once the two levels of replication have been completed there will be 64 mpirun processes spread across 16 VMs. All processes know the location of the master and fellow mpirun copies. All copies connect to four sockets set up by the master mpirun process before cloning. These sockets are used to allow redirection of standard input, output and error, and for exchanging management and control messages. In effect, the connection between each application process and its mpdman is replaced by a connection to the master mpirun process, hence making the SnowFlock mpirun a hybrid of mpirun and mpdman.

Once all of the cloned mpirun processes have established their connections to the master copy, they set up environment variables much like mpdman would do, and they use execvp to start executing the desired application. The application processes use standard MPI routines such as MPI_Init, MPI_Send, etc, to perform their tasks. As in traditional MPI, application messages are transmitted point-to-point. However, instead of sending MPI management messages (brokering new connections, synchronizing around barriers, etc) to an mpdman ring, application processes send them to the master mpirun, which forwards them to the correct application process. The MPICH library has been modified to use TCP/IP-based signaling instead of UNIX signals to inform MPI processes of the arrival of MPI management messages. This is because we have eliminated the local mpdman process which would normally pass this information to an application and alert it via a UNIX signal. While our version of MPICH can be built to work with UNIX signals, this requires the management process to rsh to the target node in order to issue the UNIX signal. This is inefficient and is thus only supported for compatibility reasons, in the case of a closed-source binary that cannot even be relinked. We do not recommend using our MPICH library in this manner.

At the end of the application run, the SnowFlock MPI management process calls the SnowFlock API to eliminate all worker VMs and shrink the processing footprint back to a single VM. Figure 3 illustrates the architecture of the SnowFlock MPI implementation: the management process is a hub, with the application processes sitting at the end of connecting spokes.



The SnowFlock MPI implementation just has a single MPI management process on the master VM, which runs only as long as the application processes and handles all their management traffic.
Figure 3: SnowFlock MPI Architecture

3.3 Discussion

Our SnowFlock-MPI implementation replaces the maintenance of a ring of management daemons by on-the-fly expansion of the footprint to the desired number of processes. One drawback of our current implementation is the centralized nature of the management infrastructure: all nodes send management (not application) messages to a master process.

We have noticed that this centralized design presents scalability issues. In section 4 we show how this manifests in our experimental evaluation. Our immediate future work plan is thus to move to a distributed management infrastructure, which will reduce the number of open sockets on any one node and the amount of work required to handle them. This can be enabled by SnowFlock, since VM IP addresses are known deterministically, which also allows one to deterministically compute ports on which each process will listen for new connections. Brokerage of MPI connections can thus be made fully distributed and point-to-point, removing most of the burden on the centralized orchestrator. Management will still be necessary for fence operations and termination.

4. Evaluation

This section details our testing: the environment and applications used. SnowFlock-MPI testing employed 32 machines with four processors each, and in each experiment a master 4-vcpu SMP VM was cloned to 31 additional copies occupying the totality of available processors. We compared our results to an optimal "zero-cost fork" baseline. Zero-cost results are obtained with VMs previously allocated, with no cloning or state-fetching overhead, and in an idle state, ready to process the jobs allotted to them. As the name implies, zero-cost results are overly optimistic and not representative of cloud computing environments, in which aggressive consolidation of VMs is the norm and instantiation times are far from instantaneous. The zero-cost VMs are vanilla Xen 3.0.3 domains configured identically to SnowFlock VMs in terms of kernel version, disk contents, RAM, and number of processors.

4.1 Machine Environment

All of our experiments were carried out on a cluster of 32 Dell PowerEdge 1950 blade servers. Each host had 4 GB of RAM, 4 Intel Xeon 3.2 GHz cores, and a Broadcom NetXtreme II BCM5708 gigabit NIC. All machines were running the SnowFlock prototype based on Xen 3.0.3, with paravirtualized Linux 2.6.16.29 running as the OS for both host and guest VMs. All VMs were configured with 640 MB of RAM. All machines were connected to two daisy-chained Dell PowerConnect 5324 gigabit switches. All results reported are the means of five or more runs, and error bars depict standard deviations.

4.2 MPI Applications

Here we describe the applications used for evaluating our SnowFlock implementation of MPI. We tested our MPI implementation with five applications representative of the domains of bioinformatics, rendering, physics, and chemistry.

4.2.1 MPI BLAST

BLAST [1], the Basic Local Alignment and Search Tool, is a popular computational biology tool offer as an Internet service [27], and is demanding of both computational and I/O resources. MPI BLAST [11] uses MPI to distribute work across multiple nodes. We performed MPI BLAST searches using 500 short protein fragments from the sea squirt *Ciona savignyi* to query a 512MB portion of the National Center for Biotechnology Information (NCBI) non-redundant protein database.

4.2.2 ClustalW

ClustalW [20] generates a *multiple alignment* of a collection of protein or DNA sequences and is also offered as a web service [15]. ClustalW uses a greedy heuristic requiring precomputation of comparisons between all pairs of sequences and a final multiple alignment stage, which requires a high degree of synchronization. ClustalW-MPI [23] parallelizes both phases of the computation using MPI. Sequence pairs are aggregated into small batches that are sent to nodes as they become available, thus preventing workers from idling. In our experiment we compute the multiple alignment of 600 synthetically generated peptide sequences with a length of one thousand amino acids each.

4.2.3 MrBayes

MrBayes [21] builds phylogenetic trees showing evolutionary relationships between species across generations, using Bayesian inference and Markov chain Monte-Carlo (MCMC) processes. MrBayes uses MPI to distribute computation across multiple nodes, although with a high degree of synchronization, and to swap state between nodes. The number of MCMC chains, probabilities, number of swaps, and frequency of swaps are all parameters that govern the degree of parallelism and synchronization. Our MrBayes experiment builds the evolutionary tree of 30 different bacterium species based on gapped DNA data, tracking back nine hundred generations and using up to 256 chains for the MCMC processes.

4.2.4 VASP

VASP [30] is a package for performing ab-initio quantum-mechanical simulations, using different pseudopotentials and plane wave basis sets. VASP is particularly suited to inorganic chemistry analyses. MPI parallelization of VASP involves partitioning a three-dimensional spatial grid into bands. This maximizes communication between nodes performing simulation of intra-band interactions, although synchronization is still required for inter-band dynamics. Further parallelization can be achieved by splitting the computation across plane wave coefficients. Our VASP test performs an electronic optimization process on a hydrogen molecule.

4.2.5 Tachyon

Tachyon [28] is a standard ray-tracing based renderer. Ray tracing is a technique that follows all rays of light in a scene as they reflect upon the different surfaces, and performs highly detailed renderings. Ray tracers like Tachyon are highly amenable to parallelization, by partitioning the rendering space into a grid and computing each grid partition in a mostly independent fashion. We used Tachyon to render a twenty-six thousand atom macro-molecule using ambient occlusion lighting and eight anti-aliasing samples.

4.3 Results

In comparing a pre-allocated set of VMs with MPI daemons already running on them against the SnowFlock variants, we expected some overhead, due to the time that it would take for VMs to be cloned and MPI connections to be established.

As can be seen in Figure 4, the level of this overhead varied from application to application. In some cases, such as MrBayes, there was a considerable extra cost, but in others, such as Tachyon, the overhead was less noticeable. This was probably due to the differences in the levels of interprocess communication of the various applications. This indicates that the choice of a hub-style architecture was probably overly-optimisitic and is something that should be revisited. Given that SnowFlock's cluster instantiation time is of the order of a second [22], it seems likely that it is the single master management process which is causing the bottleneck. In cases where there is a lot of MPI management traffic, the central management process becomes a hot spot and must handle a lot of connections. Consequently, for applications where there is not much management activity, performance is better, and for those where there is a lot, performance is worse. We also note that for longer runs the proportion of the overhead tends to be less, although its absolute value increases. We will address these issues by adopting a fully-distributed architecture for connection management that exploits SnowFlock's deterministic allocation of IP addresses to VMs. Allowing more flexibility in the way MPI processes are allocated, e.g. by not forcing an application process onto every processor, could also be a useful enhancement that would permit better tailoring of the environment to specific applications.

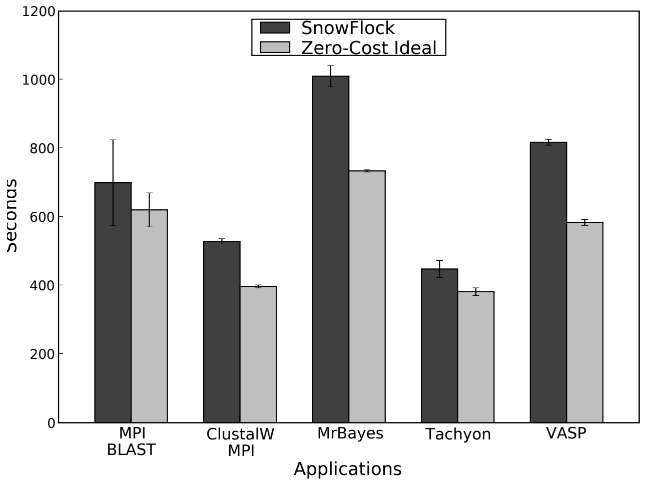


Figure 4: Application Benchmarks. Bars show execution time. **Error bars** show standard deviation. Five or more runs.

Although the SnowFlock-based applications did run slower than the standard MPI ones, we have proven the point that SnowFlock can be used successfully to enable easy parallelization with MPI, especially when further modifications to the MPICH library bring the performance overheads to a more acceptable level. Given that no modification was required to the code of the MPI applications themselves, this should make SnowFlock a realistic way of setting up MPI computations in a cloud environment.

5. Beyond Traditional MPI

Our SnowFlock implementation of MPI supports unmodified MPI applications. In this section we describe two extensions to MPI that the SnowFlock model can enable.

5.1 Computation Resizing

With SnowFlock, a master can create multiple successive sets of clones, and they are all able to communicate over the same private virtual network. This enables a coordinator to dynamically add more workers to a computation as needed. This feature is already an integral part of other parallel computation frameworks such as MapReduce [12]. For instance, when spawning jobs over several hundred nodes, the likelihood of failure of a given worker grows significantly. The ability to add more workers to the computation allows for replacement of failing workers. MPI implementations themselves do not provide fault tolerance; it is something that has to be built into the application. See [19] for some discussion on this topic.

Other instances when dynamic computation resizing are useful involve inputs that are not known a priori. For instance, data-driven computations may only be able to decide the optimal number of workers after parsing some amount of data. Internet services such as NCBI BLAST [27], or EBI ClustalW2 [15] use parallel computation to respond to complex user queries in interactive times (tens of seconds). A burst in user requests will add pressure to the parallel system that can be satisfied with the addition of more workers. Finally, fluctuations in the availability of nodes in the underlying cluster may present opportunities to add additional workers to speed up a long-lived computation.

SnowFlock makes it extremely easy to resize the computation and add workers. However, this enhancement requires careful adaptation of the MPI logic. APIs like MapReduce can readily accommodate adding new workers because they are targeted to tasks that are essentially embarrassingly parallel. There is very low or no coupling between the participants in a computation: new members can thus be added freely because they do not break the assumptions of, or have dependencies with, other members. MPI is generally used for more complex parallel computations that need to exchange state or synchronize during runtime, such as n-body or molecular dynamics jobs. While SnowFlock makes new workers with identical configuration instantaneously available, MPI programs will need adaptation to utilize this feature.

5.2 Read-only Shared Memory

MPI programs assume workers are executing on independent nodes. Application programmers are thus burdened with explicitly pushing state to all workers. For example, the initial stage in ClustalW-MPI has the worker with rank zero reading all sequences and then pushing those sequences to all its peers.

SnowFlock can greatly simplify this task. By providing stateful cloning semantics, SnowFlock allows each worker to automatically inherit the entire application state up to the point of cloning. This works conceptually as a read-only shared memory: all workers after cloning can read application state from their address space with the guarantee that the same state is visible to their peers. Application programmers are thus freed from the concern of marshaling and transferring input data and other state.

This paradigm can be combined with dynamic resizing to enable new application structures. For instance, the first stage of a computation can involve creating an initial set of parallel workers. The workers automatically inherit the input data and perform their task. Once finished, the workers are killed and the footprint is reduced again to a single VM. This VM can be checkpointed, prior to initiating a second stage: checkpointing allows long-lived computations to be recovered in the case of failure. In the second phase, a different number of workers is spawned to operate over the intermediate data, which they also inherit automatically. Examples of such computations include map and reduce phases, or ClustalW-MPI, which consists of separate phases of pair-wise sequence comparison and multiple alignment computation.

6. Related Work

A number of projects have explored the area of VM replication. In the original SnowFlock paper [22] we addressed the topic of fast stateful VM replication, but we did not address issues pertaining to integration with parallel APIs. The Potemkin project [31] implements a honeypot spanning a large IP address range. Honeypot machines are short-lived lightweight VMs cloned from a static template in the same machine with memory copy-on-write techniques. Potemkin does not address parallel applications and does not fork multiple VMs to different hosts. Remus [10] provides instantaneous failover by keeping an up-to-date replica of a VM in a separate host. Denali [32] dynamically multiplexes VMs that execute user-provided code in a web-server, with a focus on security and isolation.

Work focusing on multiplexing a set of VMs on a physical cluster has typically resorted to legacy techniques such as migration [9] or suspend/resume, without providing the performance capabilities or the convenient programming model of SnowFlock. The term "virtual cluster" is used by many projects [13, 16, 8] focusing on resource provisioning and management.

OpenCirrus [26] and EUCALYPTUS [14] are cloud management packages that only allow VM suspend and resume; they do not offer the VM fork functionality that SnowFlock provides. MOAB [25] is a high-level cluster management system that allows the monitoring and administration of certain aspects of a cluster, but does not provide an application developer the control that SnowFlock offers. Microsoft's .Net [24] framework is a completely different offering, which does not readily compare with these products. Compared with Amazon's EC2, SnowFlock offers the advantages of more rapidly instantiated VMs and an easy-to-use API, including removing the need for VM golden image creation.

The suitability of virtualization for MPI-based applications has been explored by Youseff et al. [33], who demonstrated that MPI applications can run with minimal overhead in Xen environments. We have endeavoured to go further, by showing that it is possible to make MPI work in a cloud-friendly manner, using the functionality of SnowFlock to rapidly provide a virtual cluster, while maintaining reasonable levels of performance.

Several other implementations of MPI exist beyond MPICH, including Open MPI [17] and LAM MPI [6]. With the expectation that only software changes should be required, it should be no more difficult to adapt these packages to work within SnowFlock than it was MPICH. Other parallel programming frameworks include the Parallel Virtual Machine (PVM) [18], which provides a message passing model very similar to that of MPI; OpenMP [7], which targets parallel computing in a shared memory multiprocessor; and MapReduce. MapReduce [12] is particularly relevant today as it targets the emergent class of large parallel data-intensive computations. These embarrassingly parallel problems arise in a number of areas spanning bioinformatics, quantitative finance, search, machine learning, etc. We believe that the approach presented here can be equally applied to interfaces such as MapReduce or PVM.

7. Conclusions

The offer of computing infrastructure for rent as a commodity enables research groups to leverage processing resources otherwise far outside their financial and logistic capacity. MPI is a highly popular abstraction used for programming and managing the execution of parallel jobs apt for deployment on large rented infrastructure. With SnowFlock, MPI programs can be run in virtualization-based commodity computing environments minimizing configuration burden and maximizing investment. SnowFlock minimizes configuration burden because it allows the programmer to configure and interact with a single VM. By providing stateful cloning of the master VM, SnowFlock replicates the user environment on-the-fly to dozens of clones, and releases programmers from managing multiple hosts. By combining SnowFlock with MPI, we allow users to leverage the full benefits of the cloud while providing a familiar usage model. SnowFlock maximizes investment because no idle VMs are ever needed; MPI programs expand in sub-second time their processing footprint to the number of VMs needed, and shrink their footprint equally fast.

We believe the benefits of SnowFlock extend beyond the usual MPI practices into new application models. In the future, with some further work, SnowFlock can enable modified MPI applications to instantaneously resize their footprint to involve more processors, compute more data, serve more requests, or replace failing workers dynamically. SnowFlock provides a mixed model of read-only shared memory and message passing, in which application programmers need not worry about explicitly pushing application state as this is automatically available to all clones after replication. We intend to explore these new avenues for MPI-based parallel programming in our future work.

Acknowledgments

We thank the members of the SnowFlock team, who have helped in the implementation of the SnowFlock framework and in providing feedback for this work: Joe Whitney, Adin Scannell, Steve Rumble, Charlotte Lin, Roy Bryant, Olga Irzak, Bianca Schroeder, and M. Satyanarayanan.

References

[1] Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res. 25* (1997), 3389–3402.

[2] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[3] Amazon.com. Amazon Elastic Compute Cloud Developers Guide. `http://docs.amazonwebservices.com/AWSEC2/latest/DeveloperGuide/`.

[4] Argonne National Laboratory. Mpich2. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proc. of the* 17th *Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, Oct. 2003).

[6] Burns, G., Daoud, R., and Vaigl, J. LAM: An Open Cluster Environment for MPI. In *Proc. Supercomputing* (1994), pp. 379–386.

[7] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. *Parallel Programming in OpenMP*. Elsevier, 2000.

[8] Chase, J. S., Irwin, D. E., Grit, L. E., Moore, J. D., and Sprenkle, S. E. Dynamic Virtual Clusters in a Grid Site Manager. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (Washington, DC, 2003).

[9] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005).

[10] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. 5th NSDI* (San Francisco, CA, Apr. 2008).

[11] Darling, A., Carey, L., and Feng, W.-C. The Design, Implementation, and Evaluation of mpiBLAST. In *Proc. 4th International Conference on Linux Clusters: The HPC Revolution 2003* (San Jose, CA, June 2003). `http://www.mpiblast.org/`.

[12] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)* (Dec. 2004).

[13] Emeneker, W., and Stanzione, D. Dynamic Virtual Clustering. In *Proc. Cluster* (Austin, TX, Sept. 2007).

[14] Eucalyptus. `http://eucalyptus.cs.ucsb.edu/`.

[15] European Bioinformatics Institute - ClustalW2. `http://www.ebi.ac.uk/Tools/clustalw2/index.html`.

[16] Foster, I., Freeman, T., Keahey, K., Scheftner, D., Sotomayor, B., and Zhang, X. Virtual Clusters for Grid Communities. In *Proc. Cluster Computing and the Grid* (Singapore, May 2006).

[17] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc., 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[18] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[19] Gropp, W., and Lusk, E. Fault Tolerance in MPI Programs. *International Journal of High Performance Computing Applications 18*, 3 (2004), 363–372. `http://www-unix.mcs.anl.gov/~gropp/bib/papers/2002/mpi-fault.ps`.

[20] Higgins, D., Thompson, J., and Gibson, T. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res. 22* (1994), 4673–4680.

[21] Huelsenbeck, J. P., and Ronquist, F. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics 17*, 8 (2001), 754–755. `http://mrbayes.csit.fsu.edu/`.

[22] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. of Eurosys 2009* (Nüremberg, Germany, Apr. 2009). To appear.

[23] Li, K.-B. ClustalW-MPI: ClustalW Analysis Using Distributed and Parallel Computing. *Bioinformatics 19*, 12 (2003), 1585–1586. `http://www.bii.a-star.edu.sg/achievements/applications/clustalw/index.php`.

[24] Microsoft .Net. `http://www.microsoft.com/NET/`.

[25] Moab. Moab Cluster Suite, Cluster Resources Inc., 2008. `http://www.clusterresources.com/pages/products/moab-cluster-suite.php`.

[26] Open Cirrus (TM). `http://opencirrus.org/`.

[27] RPS-BLAST. `http://www.ncbi.nlm.nih.gov/Structure/cdd/cdd_help.shtml`.

[28] Tachyon Parallel / Multiprocessor Ray Tracing System. `http://jedi.ks.uiuc.edu/~johns/raytracer/`.

[29] University of Toronto. SnowFlock Project Webpage. `http://sysweb.cs.toronto.edu/snowflock`.

[30] VASP – Vienna Ab initio Simulation Package. `http://cms.mpi.univie.ac.at/vasp/`.

[31] Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A., Voelker, G., and Savage, S. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. In *Proc.* 20th *Symposium on Operating Systems Principles (SOSP)* (Oct. 2005).

[32] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)* (Dec. 2002).

[33] Youseff, L., Wolski, R., Gorda, B., and Krintz, C. Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems. In *Proc. 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (Washington, DC, Nov. 2006).