ClusterShell, a scalable execution framework for parallel tasks

Stéphane Thiell, Aurélien Degrémont, Henri Doreau, Aurélien Cedeyn
Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA)
{stephane.thiell,aurelien.degremont,henri.doreau,aurelien.cedeyn}@cea.fr

Abstract

Cluster-wide administrative tasks and other distributed jobs are often executed by administrators using locally developed tools and do not rely on a solid, common and efficient execution framework. This document covers this subject by giving an overview of ClusterShell, an open source Python middleware framework developed to improve the administration of HPC Linux clusters or server farms.

ClusterShell provides an event-driven library interface that eases the management of parallel system tasks, such as copying files, executing shell commands and gathering results. By default, remote shell commands rely on SSH, a standard and secure network protocol. Based on a scalable, distributed execution model using asynchronous and non-blocking I/O, the library has shown very good performance on petaflop systems. Furthermore, by providing efficient support for node sets and more particularly node groups bindings, the library and its associated tools can ease cluster installations and daily tasks performed by administrators.

In addition to the library interface, this document addresses resiliency and topology changes in homogeneous or heterogeneous environments. It also focuses on scalability challenges encountered during software development and on the lessons learned to achieve maximum performance from a Python software engineering point of view.

1 Introduction

From a logical perspective, cluster system software is what differentiates a cluster from a collection of individual nodes. Having a scalable and resilient cluster system management toolkit is essential to the successful operation of clusters. According to the TOP500 [1] list, more than 80% of installed HPC systems are running Linux, and open source software is now used as the foundation of most general-purpose[1] supercomputers. But even the simplest cluster-wide administrative task can become a nightmare when executed on a petaflop supercomputer of thousands of nodes. Often, tools or services are tuned to scale on a case-by-case basis. As a result, clusters often rely on a fragile administration software layer, suffering from the lack of robustness, usability and from management complexity. ClusterShell answers this by providing an open source, robust and scalable framework for cluster management and administration, that can be used by both system administrators and software developers. Indeed, benefiting from full-featured and scalable tools can save a lot of time for administrators, resulting in more efficient daily operations and reduced downtime during scheduled maintenance.

ClusterShell is available as a Free Software product under the terms of the CeCILL-C license [5], a French transposition of the GNU LGPL, and is fully LGPL-compatible. It consists in a Python (v2.4 to 2.7) library and a small set of command-line tools. It takes care of common administration issues encountered on clusters, such as operating on groups of nodes, running distributed commands using optimized execution algorithms, as well as helping result analysis by gathering and merging identical command outputs, or retrieving return codes. It takes advantage of existing remote shell facilities already installed on most systems, such as SSH. The command-line tools, `clush`, `clubak` and `nodeset` are efficient building blocks for administrators that also allow traditional shell scripts to benefit from some of the library features. Figure 1 shows an overview of the ClusterShell framework.

Primarily, the ClusterShell Python library implements an efficient event-based mechanism for parallel administrative tasks, whether they be local or distant. The ClusterShell Python API[2] provides an event-driven

[1] *General purpose* as defined in [7]
[2] http://packages.python.org/ClusterShell/

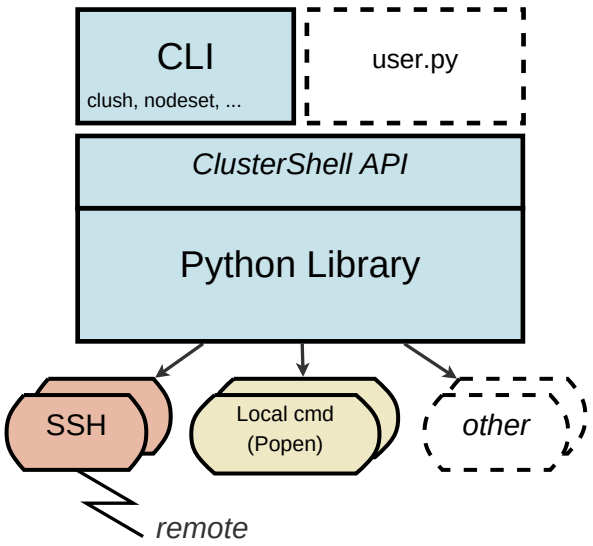

Figure 1: ClusterShell overview

and object-oriented interface that allows application to schedule actions, like running shell commands or copying a file, and to register for specific events from these local or remote tasks. Several helper methods are then available to analyze results during the execution or afterwards.

2 Cluster naming scheme

Computer clusters often use a systematic naming scheme, such as a common node prefix conforming to RFC 1178 *"Choosing a name for your computer"* [14] plus a numbering scheme. For example, high-availability clusters [10], database clusters and Hadoop clusters (for HDFS DataNodes [19]) frequently use simple serial naming procedures for individual servers. The naming policy of high-performance computing cluster nodes is often as simple, but can also be more complex by adopting a multi-dimensional numbering scheme. A numbering system that matches physical locations in a server room is sometimes adopted (eg. nodes are named according to the rack and slot number in [20, 6, 18] like `r01n03`). Another example could be to use logical positions in a multi-dimensional interconnection network (eg. `torus-1-2-3`). In this section, we define the *nodeset* and node group notations and present associated features available in ClusterShell to easily and efficiently manage these cluster naming schemes.

```
curie0
curie0,hwm0
stor[02-08]
curie[50,100-120,1500-6000]
curie[200-249]-ipmi
curie[200-249],gaia[20-59]
da[10-19]c[1-2]
```

Figure 2: Common *nodeset* examples

```
curie[2-8/2]  ⟺ curie[2,4,6,8]
stor[01-10/3] ⟺ stor[01,04,07,10]
curie[50,1500-2000/2,3000-6000/4]-ipmi
```

Figure 3: Stepped *nodeset* examples

2.1 The *nodeset* notation

The *nodeset* notation presented here is a syntax for specifying a set of cluster nodes or host names. The comma (`,`) is used to separate different naming patterns (eg., to address multiple clusters). This notation is an extension of the one already used in cluster tools, such as SLURM resource manager [11], pdsh [16] or kanif [9]. Indeed with clusters growing, a commonly adopted notation has emerged mainly to allow the use of ranges whenever possible. A continuous range is specified by starting and an ending number, separated by a dash (e.g. `0-9`). Numbers may be expressed with a fixed length padding of leading zeros (`0`). Each discontinuous range, or single number, is separated by a comma (`,`). This makes a set or ranges, or *rangeset*. Within a *nodeset*, square brackets are used to signal a *rangeset*. Figure 2 shows this naturally compact cluster nodes naming scheme.

As an extension to the traditional *rangeset* notation, the stepping syntax already seen for Lustre™ networking [2] appends to a continuous range a slash character (`/`) and a step size. For example, the `2-8/2` format indicates a range of `2-8` stepped by `2`; that is `2,4,6,8`. Figure 3 shows some examples of this `nodeset` notation extension.

Moreover, our practical experience in cluster administration has shown that being able to embody some basic set operations in *nodeset* can be very convenient, and thus even more when working with node groups (discussed below). We define as a valid *nodeset* notation the following special operators:

- `,` as the union operator,

- ! as the difference operator,
- & as the intersection operator,
- ^ as the symmetric difference operator.

nodeset patterns are read from left to right, and character operators are processed as they are met. Figure 4 shows a simple example.

`curie[0-50]!curie5` ⟺ `curie[0-4,6-50]`

`curie[0-10]&curie[8-20]` ⟺ `curie[8-10]`

Figure 4: *nodeset* character operator examples

2.2 Node groups

A node group represents a collection of nodes. Working with node groups is much more convenient and much safer when administrating large compute clusters or server farms. For example, a node group can correspond to nodes using the same set of resources or a specific type of hardware. Node groups can be used for a variety of reasons. In most cases, cluster software already provides several sources of static or dynamic node groups (eg., from a cluster database, genders [15], SLURM *nodes*, *partitions* or *jobs* [11], etc.). ClusterShell is able to bind to these node group sources and to provide unified information to the cluster management software. Node group provisioning is done through user-defined shell commands or through library extensions in Python. That is, ClusterShell itself doesn't manage node group definitions. Still, binding to a node group source based on flat files is straightforward[3].

The unified node group string notation introduced with ClusterShell is invariably prefixed by the arobase character (`@`) and constructed from the node group source followed by a separator character (`:`) and a node group name, the latter being freely expressed by the source. The notation can be further simplified using relative naming by omitting the node group source. In this case, the node group source configured by default is used to resolve the group. Figure 5 illustrates this syntax.

Moreover, when node group names are themselves adopting a systematic naming scheme as seen in section 2 for node names, we are able to represent a set

[3] A node group source example, based on a flat file, is provided by default.

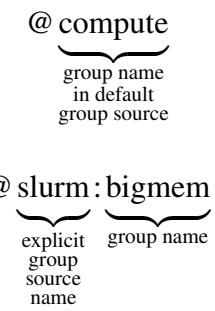

Figure 5: Overview of *nodeset* syntax for node groups

of node groups in a similar fashion. The following example illustrates how to represent twenty Scalable Storage Units in a storage cluster: `@ssu[00-19]`, that is `@ssu00`, `@ssu01`, etc., up to node group `@ssu19`, each one corresponding to a set a nodes.

Evaluating node groups in a *nodeset* notation is quite straightforward, they are simply substituted by their corresponding nodes when needed. As for a regular set of nodes, the special operator characters seen in section 2.1 are supported with node groups. Figure 6 shows an example.

> `@slurm:standard&@ethswnode:sw[0-6/2]`
> stands for nodes from the SLURM partition named *standard* which are also connected to even-numbered switches (*sw0*, *sw2*, *sw4* and *sw6*)

Figure 6: Example of *nodeset* notation using node groups and the intersection operator

Using a node group explicitly indicates a grouping intention so operations are computed on the whole group, but also on the whole set of groups when brackets are used to designate a set of ranges. Otherwise, the operator-separated list of elements is evaluated from left to right. Intentionally, there is no support for parentheses or other ways to explicitly indicate precedence by grouping specific parts. Indeed, we tried to keep the syntax simple enough, focusing on the wide variety of tasks that cluster administrators perform.

2.3 Working with *nodesets*

nodeset objects are omnipresent in the ClusterShell framework within the `NodeSet` Python class. Two user-interfaces are available to manipulate *nodeset* strings whose syntax is described in section 2.1: one is

```
$ nodeset -f da1c1 da1c2 da3c1 da3c2
da[1,3]c[1-2]
```

Figure 7: Example of multi-dimensional *nodeset* folding using the `nodeset` command-line tool

the `NodeSet` Python class and the second one is the `nodeset` command-line tool.

For instance, `nodeset` provides optional switches to count the number of nodes within a *nodeset* (`-c`), to expand it (`-e`), to fold nodes into a *nodeset* (`-f`), to access node groups information, etc. It has become for us an essential command for daily cluster administration and an integral part of our shell scripts. All of its features are described in the documentation and on the ClusterShell Wiki[4].

The rest of this section describes some implementation aspects of different *nodeset* features.

2.3.1 *nodeset* folding

To fold a *nodeset*, we need a way to fold a set of ranges (a *rangeset*), as seen on section 2.1. `RangeSet` is the Python class that manages a *rangeset*. The latest implementation uses a standard Python `set` object to store the set of indices. We discuss in section 3.2 performance issues encountered on this topic. The folding implementation uses an iterator[5] on *slice* found objects, each one representing a set of indices specified by a range, plus a possible step. This is called, for example, when displaying a *nodeset* as a string.

Uni-dimensional *nodeset* is thus mainly solved by having a way to fold a *rangeset*. Multi-dimensional *nodeset* folding is more complicated. While expanding a multi-dimensional *nodeset* is easily achieved through a Cartesian product of all dimensions (we use Python's `itertools.product()`), folding is achieved by comparing *rangeset* vectors two by two, and to merge these vectors if they differ only by one item. Figure 7 shows an example of this multi-dimensional folding feature, available starting with ClusterShell version 1.7.

[4] https://github.com/cea-hpc/clustershell/wiki

[5] `RangeSet._folded_slices()`

2.3.2 Node groups regrouping

Another interesting ClusterShell feature is the ability to find fully matching node groups for a specified *nodeset*. This is called the *regroup* functionality. A simple heuristic implementation determines whether to use the `list` (list all groups) plus `map` (group to nodes) external commands, or to use `reverse` (node to groups). It then resolves node groups, returning largest groups first.

3 Scalability challenges with CPython

As a system software, ClusterShell is relying on CPython, the most-widely used implementation of the Python programming language. It is also the default of all Linux distributions used for clustering that we know of. This sections addresses performance challenges we faced in order to use CPython at scale.

3.1 Parallel programming

Because of its *Global Interpreter Lock* (or GIL), the standard CPython interpreter is unable to achieve actual concurrency with multithreaded programming [3]. Nevertheless, modules from the Python standard library can be leveraged to bypass this limitation and write high performance parallel code.

ClusterShell uses a combination of non-blocking I/O management and multiprocessing. The event-based I/O notification infrastructure is described in section 4.1. For CPU-intensive operations such as SSH connections, ClusterShell spawns external processes via the `fastsuprocess` module (see section 3.3). It therefore delegates scheduling operations to the OS, removing GIL-based contention constraints.

3.2 `RangeSet` performance

`RangeSet` is the Python class that manages a set of ranges as seen in section 2.1. In its first implementation[6] used in-memory *slice* objects representing the set of indices specified by a range, plus an optional step value (≥ 1). We first thought that direct access to ranges and operations done on these objects for 10k nodes (eg., on a range like `1-10000`) would be optimal with limited

[6] up to ClusterShell 1.5

memory footprint. But performance issues were quickly encountered when running on thousand node HPC clusters. The complexity of most related algorithms being in $O(R)$ with a number of discontinuous ranges of R, the bottleneck was then the high number of discontinuous ranges seen on these clusters. These sparse *nodesets* are commonly seen on large clusters (the way nodes are replying, in a random fashion, can create such "holes").

We then developed an intermediate implementation in Python using a `bintrees`-based AVL tree [13] to operate on ranges in $O(log(n))$. While it significantly outperformed the first implementation, we still did not achieve the performance we aimed for in all cases, probably because of the CPython overhead when creating a large number of objects. As a comparison, `bintrees` benchmarks using the *pypy* interpreter[7] show a 10 to 40 times speedup over CPython[8].

In the current implementation, the `RangeSet` class finally uses a Python `set`. Ranges are expanded as numeric indices in the set and a folding algorithm is used in case it needs to display a *rangeset*. It probably looks less elegant than using a balanced tree of ranges, but it is significantly faster than the AVL-tree implementation, mainly because sorting and set-like operations are very efficient in CPython.

3.3 `fastsubprocess`

Early versions of ClusterShell used the `subprocess` Python module to spawn new processes and connect to their input, output or error pipes. When using a large *fanout* value (> 128), that is, the number of child processes allowed to run at a time, we noticed a significant overhead localized in `subprocess.Popen`, even on medium size clusters. We found out that the parent Python process spends its time in a blocking `read(2)` operation, waiting for its children, leading to a serialization of all forked processes. Indeed, a pipe allows exceptions raised in the child process before the new program has started to execute, to be re-raised in the parent for convenience. This problem has been discussed on Python issue #11314[9] and the choice of feature *vs.* performance has been kept for now.

[7] http://pypy.org/
[8] http://pypi.python.org/pypi/bintrees/
[9] http://bugs.python.org/issue11314

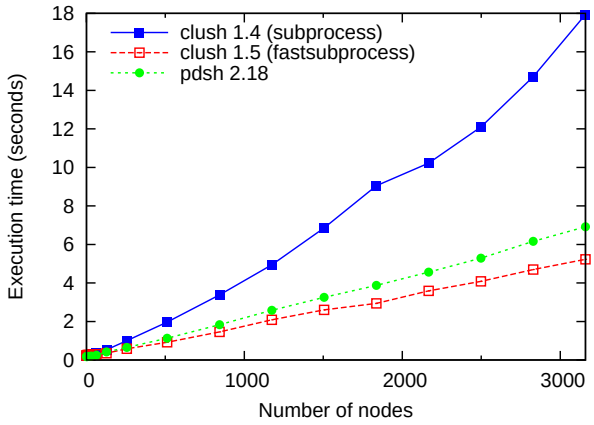

Figure 8: Performance comparison between ClusterShell *engines* based on `subprocess` and `fastsubprocess` and with C-based `pdsh` using a *fanout* value of 128

To work around this issue, we decided to adapt the `subprocess` module to make a faster, performance oriented version of the module for ClusterShell, that we named `fastsubprocess`. We removed the pipe used to transfer potential execution failures from the child to its parent, thus avoiding the blocking `read(2)` call. A child process returns a status code of 255 on `execv(3)` failure, which is handled by `Popen.wait()` in the ClusterShell library on proper event. We now also return file descriptors instead of file objects to avoid calling `fdopen()`. The only drawback of `fastsubprocess` is that it is not able to distinguish between an explicit return code of 255 from the child and an `execv(3)` failure, which we considered being an acceptable shortcoming considering the performance gain presented below.

Experiment: We evaluated the performance of the `fastsubprocess` module on Tera-100, CEA largest HPC Linux cluster, composed of a four-socket eight-core Intel® Xeon Nehalem EX (X7560) head node[10] running at 2.27 GHz with 64 GB of RAM, and more than 3000 compute nodes[11] each also four-socket X7560 nodes with 64 GB of RAM. ClusterShell version 1.4 was implemented using the regular Python `subprocess` module. Figure 8 clearly illustrates the scalability problem of this module when used intensively. As of version 1.5, we switched to our own `fastsubprocess` optimized module. `Pdsh` and ClusterShell 1.5 produced very similar results. How-

[10] S6030 bullx node
[11] S6010 bullx nodes

ever, ClusterShell execution times were slightly lower.

4 Scalable execution framework

In order to make ClusterShell production-ready on 10k-nodes clusters, we focused on both vertical and horizontal scalability aspects.

Numerous optimizations spread over the whole codebase brought scale-up improvements. Low memory and CPU footprint, as well as high performance I/O management have been achieved by leveraging efficient I/O notification facilities provided by the operating system.

Starting with ClusterShell version 1.6, the library is shipped with a major horizontal scalability improvement, allowing commands to be propagated to the targets through a tree of gateways (or proxies).

4.1 Vertical scalability

Dealing with the I/O streams from the multiple SSH instances that ClusterShell spawns can be a performance bottleneck. ClusterShell addresses this issue with a specific I/O management layer. Basically, massively parallel applications such as ClusterShell face the same problems as heavily loaded servers handling thousands of clients. In this regard, ClusterShell uses non-blocking I/Os and the most efficient I/O management paradigms [12].

Within a library instance, I/O management is done by a backend module, referred to as the *engine*. Several *Engines* are implemented. Each one relies on an non-blocking I/O demultiplexing system call (such as `select(2)` or `epoll(7)`) and exports a well defined interface to the upper layers of the library. This is entirely transparent and the other layers are fully *engine-agnostic*.

An engine provides primitives for registering and unregistering read, write or exception events on file descriptors, as well as an event loop entry point.

Each SSH process gets its standard input, output and error pipes registered to the library *engine* when starting. The engine processes the events from each I/O stream, and the potential timers, in a single-threaded loop.

The best available backend is selected at runtime, given that some OS-specific system calls might be unavailable on the running platform. This strategy allows ClusterShell to leverage the most efficient I/O notification subsystem [8] amongst the ones available.

Three backends are currently implemented:

- High performance, Linux-specific `epoll(7)`-based engine.
- Intermediate `poll(2)`-based engine
- Fallback `select(2)`-based engine

The most efficient backends being system-specific, this redundancy allows ClusterShell to achieve high performance while staying significantly portable. ClusterShell is available on a large number of systems, and packaged into several GNU/Linux distributions, including Red Hat® Enterprise Linux (RHEL) through the Fedora Extra Packages for Enterprise Linux (EPEL) repository, Fedora[12], Debian[13] and Arch linux[14].

Because of the system load generated by starting numerous concurrent SSH processes, the performance differences between the `epoll(7)` and `poll(2)`-based engines is hardly measurable. Therefore, further performance and scalability improvements have been done on the horizontal aspects.

4.2 Horizontal scalability

Even though the most efficient engines can handle thousands of I/O streams, the number of concurrent SSH processes is a blocking limitation [9] due to the CPU and memory load generated on the root node (from which commands are issued).

That is why we designed and implemented a new distributed propagation mode within the project. Commands are delivered through a network of gateways and results are sent back to the root node upward the created propagation tree.

The load gets shared between gateways, and the $O(\lambda N)$ propagation time we observe with a flat-tree mode (λ being the unit execution time) becomes $O(\lambda K log_k(N))$, with an arity of K [17] (K being the number of branches a gateway connects to). Figure 9 shows a schematic illustrating this principle.

[12] https://admin.fedoraproject.org/pkgdb/acls/name/clustershell

[13] http://packages.debian.org/fr/sid/clustershell

[14] http://aur.archlinux.org/packages.php?ID=53476

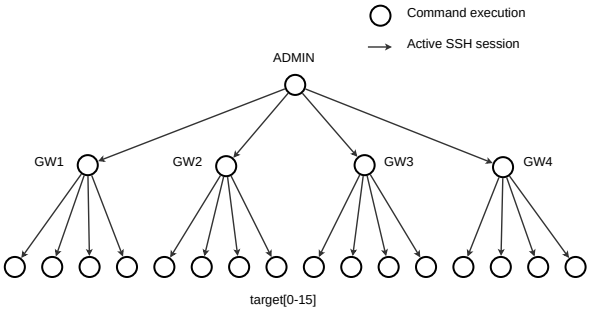

Figure 9: Hierarchical command propagation scheme

We also implemented a *grooming* mode that allows gateways to aggregate responses received within a certain timeframe before transmitting them back to the root node in a batch fashion. This contributes to reducing the load on the root node by delegating the first steps of this CPU intensive task to the gateways.

ClusterShell uses the same command sending techniques it uses in "normal" mode to control the gateways. As a result, the only requirement to setup a propagation tree is to have ClusterShell installed on the nodes that are susceptible to act as gateways, along with running a SSH server. SSH was chosen as a transport channel as it allows the propagation tree subsystem to use already in-place ClusterShell mechanisms, and also because of its reliability and security mechanisms. Nevertheless, the ClusterShell connector manager was designed with modularity in mind to ease support of additional protocols (such as RSH, PDSH or a ClusterShell-specific communication protocol).

A lightweight communication protocol ensures proper exchanges within the tree, using serialized Python objects embedded in a XML stream. As Python has a built-in incremental SAX parser (which is event-based), XML was a natural choice to represent the data and to guide the execution flow of the parser when they are received.

4.2.1 Communication within the tree

Gateways are implemented as ClusterShell-based state machines. Once instantiated from the remote ClusterShell process, gateways receive the topology to use, the targets to reach and the command to execute. Gateways recursively contact the next hop machines, deploying the propagation tree until final targets are reached.

```
# Allow connections from admin nodes
# to gateways
admin[0-2]: gateways[0-20]

# Allow connections from gateways to
# compute nodes
gateways[0-20]: compute[0-5000]
```

Figure 10: Topology syntax

The communication channel between the root node and a gateway (as well as between two gateways) is a single SSH connection that remains open until all results have been collected and sent back to the root node, which is also responsible for closing the channel at the transport layer.

4.2.2 Adaptive propagation

Topology is expressed through a configuration file on the root node as a list of possible connections between source and destination nodesets.

Mechanisms are implemented within ClusterShell to mark a gateway as unreachable and exclude it from the topology. Additionally, a work-stealing mechanism could be interesting, to let gateways adjust the load in real time between each other. The work made by C. Martin in that domain for the TakTuk project [17] stresses how valuable those mechanisms are when dealing with heterogeneous clusters and grids.

4.3 Experiments

In this section, we evaluate the performance of the scalable ClusterShell execution model, as introduced on section 4.2. To perform this experiment, we used Curie, a 2 Petaflop HPC Linux cluster operated by CEA. More precisely, we used the *Thin Nodes* partition of Curie, which consists of 5040 dual-socket nodes[15] each containing two eight-core Intel® Sandy Bridge EP (E5-2680) processors running at 2.7 GHz and 64 GB of RAM. Curie's operating system is Bullx Linux Advanced Edition, based on Red Hat® Enterprise Linux 6.1. The experiments have been done during a scheduled maintenance so no job was running. We avoided any external perturbation (such as the one that could be

[15] B510 bullx nodes

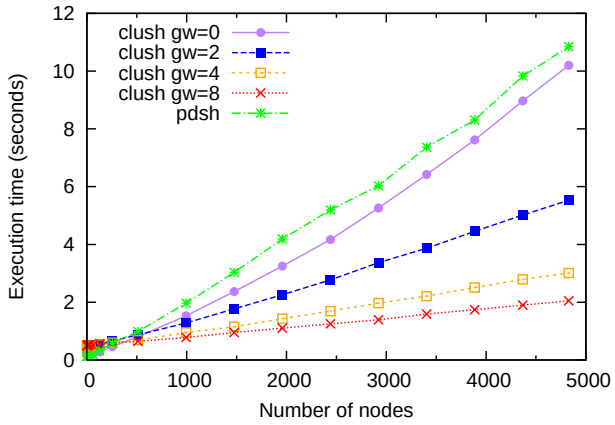

Figure 11: Performance comparison between `clush` v1.6 in basic mode (sliding window), in distributed mode (1 level of *n* gateways) and `pdsh` v2.18 on Curie (using ssh, fanout=128, command="echo ok")

induced by NFS, LDAP, etc.) by using a properly configured *root* superuser.

To measure the command propagation time, we remotely execute a command with the help of the `clush` command-line tool which is part of the ClusterShell framework. A command option allows an easy setup of the topology configuration file as seen in section 4.2.2.

For the experiment, we chose the command `echo ok` which has a negligible execution time and still enables some parsing code to be covered with a lightweight payload. Figure 11 presents the execution time of this command on up to 4828 remote nodes, with different execution models: basic model with a fixed fanout value (sliding window), tree-based propagation model with a varying number of gateways. `pdsh` v2.18 was used as a reference (using a fanout of 128, which we found to be the optimal value).

In basic execution mode (gw=0), `clush`'s curve looks smoother than `pdsh`'s one. Also, execution time is slightly lower, which is probably due to the event-based `epoll(7)`-based *engine*.

In distributed mode, with a single level of gateways, `clush` induces a constant overhead of about 300 ms, which is slightly noticeable on this figure at the leftmost part of the graph. This overhead is rapidly hidden by the gain of using a distributed command propagation (at about 250 nodes). The performance gain of the tree-based propagation is significant when increasing the number of gateways.

5 Related works

Several solutions exist to distribute administration tasks on parallel systems.

In terms of integration, these approaches can be classified in two categories: those providing a library API, like `func`[16] or `fabric`[17], and standalone applications like `pdsh` [16] or `gexec`[18]. ClusterShell combines both approaches by providing a library and tools built on top of it. Also, unlike other tools like `gexec`, ClusterShell does not require installation of an additional daemon on remote nodes.

In terms of scalability, existing solutions can also be classified in two categories, those that streamline direct commands, like `capistrano`[19], and the ones that propagate commands through a scalable (eg. hierarchical) scheme like `taktuk` [17].

Developed to facilitate production on large-scale systems, ClusterShell leverages the best of both approaches. Indeed, ClusterShell provides a convenient and scalable Python library along with efficient administration tools, especially designed for HPC clusters.

6 Conclusion

In this paper, we have presented ClusterShell, a lightweight Python framework used daily in production on the largest CEA HPC Linux clusters. System administrators and developers at CEA are working very closely, and this cooperation allowed us to improve the ClusterShell library to address the wide area of needs that administrators express for compute clusters as well as storage, post-processing clusters and even server farms.

From a Python performance perspective, limitations we faced were not the ones we initially expected. Also, by using original and creative techniques, we managed to circumvent common pitfalls.

Today, ClusterShell is used as a building block for other HPC software projects, such as Shine [4], an open source solution designed to setup and manage the

[16] https://fedorahosted.org/func/
[17] http://fabfile.org/
[18] http://www.theether.org/gexec/
[19] https://github.com/capistrano/capistrano

Lustre™ file system on a cluster, or Sequencer [21], an open source tool to efficiently control hardware and software components in HPC clusters.

We also presented the scalable execution engine of ClusterShell and the performance experiments we conducted, reflecting the success of our approach on large homogeneous clusters.

References

[1] Top 500 supercomputer sites. http://www.top500.org/, 2012.

[2] Oracle and/or its affiliates. Lustre™ 2.0 Operations Manual, 2011.

[3] David Beazley. Inside the Python GIL, 2009.

[4] CEA. Shine, Open Source Lustre management tool. http://lustre-shine.sourceforge.net/.

[5] CEA, CNRS and INRIA. CeCILL and Free Software. http://www.cecill.info/index.en.html.

[6] Brooks Davis, Michael AuYeung, Matt Clark, Craig Lee, Mark Thomas, James Palko, and Robert Varney. Lessons learned building a general purpose cluster. In *Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology*, SMC-IT '06, pages 226–234, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Allan R. Hoffman et. al National Academies. *Supercomputers: Directions in Technology and Applications*. The National Academies Press, 1989.

[8] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Ottawa Linux Symposium*, volume 19, 2004.

[9] Guillaume Huard. Kanif, a TakTuk wrapper for cluster management and administration. http://taktuk.gforge.inria.fr/kanif/, 2007.

[10] Red Hat Inc. et al. Red Hat Enterprise Linux 6 cluster administration, configuring and managing the high availability add-on, 2011.

[11] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[12] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html, 2003.

[13] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.

[14] D. Libes. Choosing a name for your computer. RFC 1178 (Informational), August 1990.

[15] LLNL. Genders. https://computing.llnl.gov/linux/genders.html, 2007.

[16] LLNL. Pdsh. https://computing.llnl.gov/linux/pdsh.html, 2007.

[17] Cyrille Martin. *Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2004.

[18] Hiroyuki Mishima and Jun Ni. Rocks Cluster Installation Memo. Technical report, Medical Imaging HPC & Informatics Lab, 2008.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Simon Fraser University. ClusterAdmin - HPC wiki. https://wiki.cs.sfu.ca/HPC/ClusterAdmin, 2010.

[21] Pierre Vignéras. Sequencer: smart control of hardware and software components in clusters (and beyond). In *Proceedings of the 25th international conference on Large Installation*

System Administration, LISA'11, pages 4–4,
Berkeley, CA, USA, 2011. USENIX Association.