Automatic Dependency Management for Scientific Applications on Clusters

Ben Tovar, Nicholas Hazekamp, Nathaniel Kremer-Herman, and Douglas Thain
{btovar,nhazekam,nkremerh,dthain}@nd.edu
Department of Computer Science and Engineering
University of Notre Dame

***Abstract*—Software installation remains a challenge in scientific computing. End users require custom software stacks that are not provided through commodity channels. The resulting effort needed to install software delays research in the first place, and creates friction for moving applications to new resources and new users. Ideally, end-users should be able to manage their own software stacks without requiring administrator privileges. To that end, we describe VC3-Builder, a tool for deploying software environments automatically on clusters. Its primary application comes in cloud and opportunistic computing, where deployment must be performed in batch as a side effect of job execution. VC3-Builder uses workflow technologies as a means of exploiting cluster resources for building in a portable way. We demonstrate the use of VC3-Builder on three applications with complex dependencies: MAKER, Octave, and CVMFS, building and running on three different cluster facilities in sequential, parallel, and distributed modes.**

I. Introduction

Software installation and management remains a significant challenge for researchers who are the end-users of computing facilities ranging from small personal clusters to national leadership computing facilities. Historically, software installation has been seen as the domain of the professional system administrator, who selects the most appropriate combination of tools for a given facility and user. Of course, end-users can ask for additional software to be installed, but this only happens within the limited administrator time and effort available.

End users do have the basic permissions available to build, install, and execute software in their home directories, but even highly skilled users may struggle to install a complex software package. One package may depend implicitly on many others, which must first be installed and configured, sometimes in peculiar ways. An effort to install one simple piece of software may unexpectedly turn into a day long march to discover dependencies. Unfortunately, the complex set of steps may be forgotten as soon as the task is complete, so the details must be re-discovered if the same software is required in another facility or by another user.

Of course, this problem is largely solved in the limited universe of personal computers. Widely used tools like `yum` and `apt` install software from a repository of carefully vetted and confirmed packages designed to work together in a given operating system. Each package is designed to install in a fixed system location and requires root privileges to do so. This is insufficient in the context of a shared facility, where end-users are unprivileged, and multiple incompatible versions may need to exist simultaneously.

To address this problem, we present VC3-Builder, a user-level system for managing software dependencies across multiple clusters. VC3-Builder shares some elements of design with recent systems such as Nix [8] and Spack [10], but it is more suited for rapid deployment by exploiting pre-existing software where available and using the resources of the cluster itself to perform parallel and distributed builds.

We demonstrate VC3-Builder by several case studies of complex applications that present a challenge for users to build and deploy manually: the MAKER bioinformatics pipeline, the CVMFS distributed filesystem, and the Octave mathematics system. Each of these is shown to build and execute in reasonable time on several campus clusters and XSEDE resources, using the VC3-Builder capability to perform distributed builds and package the results for reproducibility and archival.

VC3-Builder is the first component of a larger project titled "VC3: Virtual Clusters for Community Computation". The VC3 effort will enable unprivileged end-users to provision virtual clusters across existing computing resources using only standard user-level privileges. This involves multiple issues including resource selection, dynamic provisioning, middleware deployment, and more. The ability to quickly deploy software stacks at user-level is one element underlying the larger VC3 project.

II. Objectives

These considerations drive the design of VC3-Builder:

1) **No special privileges.** Many software packages and build tools require administrator access to install, even if the software does not inherently require it. While obtaining root access on a single machine may be possible in some cases, it requires manual interactions through administrative channels and is not practical to expect at all computing facilities. Instead, we assume that users wish to install software using ordinary user accounts available through ordinary means. Software that inherently requires root privileges, such as kernel drivers or low level performance monitors are outside the scope of this work.
2) **Batch context.** Our aim is to enable the easy execution of a large number of jobs in a batch context, where interaction with the user is not desirable. Software

installation must operate as a reliable preliminary step to job execution, and then reliably clean up afterwards, so as not to create a garbage collection problem. Likewise, if a large number of jobs start simultaneously on a given cluster, the build process should not itself become a performance bottleneck in the system. Software deployment should be a hands-free, automatic part of workload execution.

3) **Lightweight bootstrap.** An installation tool must have an absolute minimum set of dependencies, so that itself may be installed without creating a dependency management problem for the end-user. Of course, every software has some sort of dependency, if only for a given CPU or interpreter. Such dependencies must be widely known, easily discovered, and trivially provided.
4) **Dynamic composition.** The set of dependencies needed may change from minute to minute, and incompatible dependencies may be needed simultaneously by different jobs. For example, Python 2 and Python 3 are mutually incompatible but may be needed simultaneously. To permit this, dependencies must be installed in a way that allows each to be selected individually at runtime. This is most easily accomplished by putting each software and version in a distinct directory, rather than the historic Unix practice of installing everything in `/usr/bin`.
5) **Efficient cluster utilization.** Large cluster facilities have implied constraints upon utilization that are necessary to ensure that usage is properly accounted, and all users have access to sufficient resources. For example, it is generally discouraged to place long-running intensive tasks on a cluster head node; large complex builds should be performed on the cluster itself, if possible. Likewise, parallel filesystems are suitable for distributing large amounts of data, but can suffer when used for distributed synchronization or metadata-heavy loads.
6) **Technology independence.** Today, a variety of virtual infrastructures are under development, experimentation, and use. Virtual machine monitors, container technologies (Docker [17], Singularity [15], Shifter [7]), and operating facilities such as namespaces and control groups are in wide use as of this writing. However, these did not exist just a few years ago, and we fully expect that new technologies will be popular a few years hence. We aim to rely on simple, perennial abstractions (such as packages, filesystems, search paths) that can be deployed *within* these technologies if desired, but not tie the user down to a single technology.

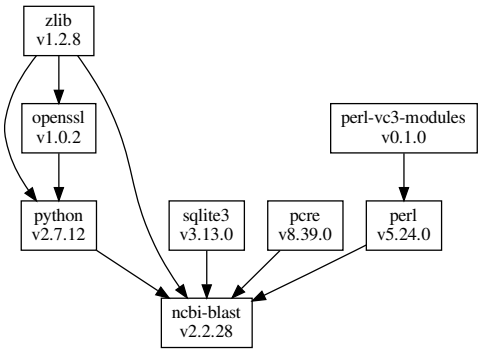


Fig. 1. Dependency graph for ncbi-blast.

III. Overview

From the end-user perspective, VC3-Builder is invoked as a command line tool which states the desired dependencies. The builder will perform whatever work is necessary to deliver those dependencies, then start a shell with the software activated. For example, to run NCBI BLAST:

```
% vc3-builder --require ncbi-blast:2.2.28
```

VC3-Builder evaluates and (if necessary) downloads and installs the set of dependencies required, represented internally as a graph (shown in Figure 1).

The user is presented with a new shell running in a sandbox directory, with the desired software present in the search path:

```
% which blastall

% blastall --help
blastall 2.2.18
...
% exit
```

In a batch context, the user would give one command line combining VC3-Builder with the desired job to run:

```
% vc3-builder --require ncbi-blast \
              -- blastall ...
```

This permits VC3-Builder to be easily submitted as a standalone job to a batch system or other similar middleware. When the job begins to execute, the necessary environment is created, the contained command is run, and then the sandbox and its associated resources are freed.

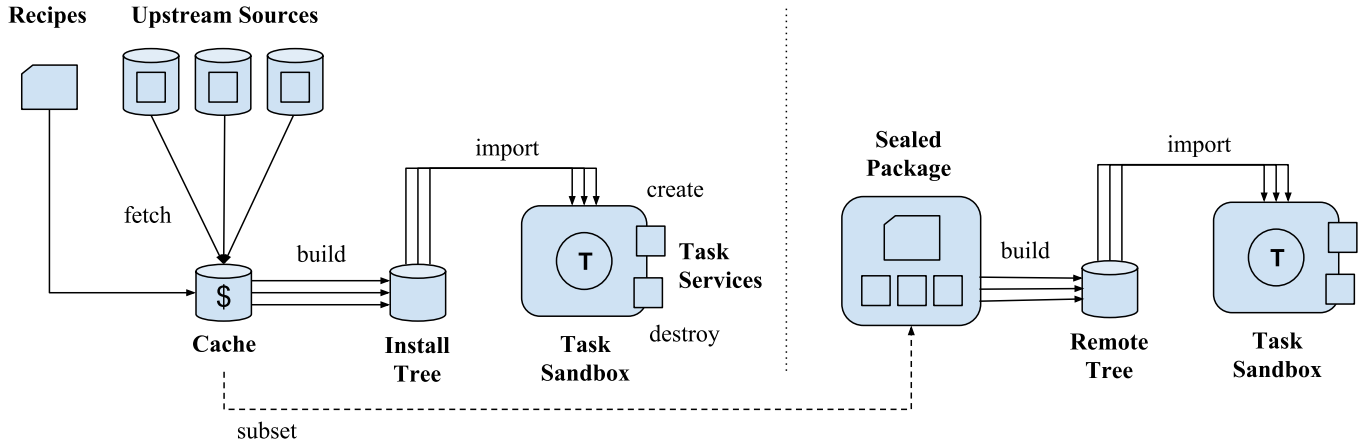


Fig. 2. VC3-Builder Architecture. In the simplest case, the builder works by first fetching recipes for the desired packages explicitly mentioned on the command line and their dependencies. Each package not found natively is built and installed into a local install tree. Before running a task, a sandbox is created, dependencies are imported, and services are started. When the task ends, services are stopped and the sandbox destroyed. Alternatively, the necessary recipes and packages can be collected together into a standalone package which can be copied, built on a node without network access, or preserved for reproducibility purposes.

```
"charm":[
 {
  "version":"v6.007.01",
  "sources":[
      {
          "type":"generic",
          "files":[ "charm-6.7.1.tar.gz" ],
          "recipe":[
              "tar -C ${VC3_PREFIX} -xpf charm-6...",
              "cd ${VC3_PREFIX}",
              "env MPICXX=mpicxx ./build charm++ ..."
          ]
      }
  ],
  "dependencies":{
      "openmpi":[ "v1.010" ]
  },
  "environment_variables":[
      {
          "name":"PATH",
          "value":
          "$VC3_ROOT_CHARM/mpi-linux-x86_64/bin",
          "absolute":1
      }
      ...
] } ]
```

Fig. 3. Sample Recipe for Charm++

IV. Architecture

Figure 2 shows the overall components of VC3-Builder.

To facilitate bootsrapping, VC3-Builder is designed to have as few dependencies as possible. The tool is written in Perl; although this is not the most modern of languages, it is nearly universally installed on Unix-like machines, and does not suffer the language compatibility problems of more recent languages like Python. All Perl libraries used by VC3-Builder are incorporated into a single standalone file using `App:FatPacker` which can, if necessary, be compiled into a native, static executable[1].

A global master repository contains a set of package *recipes* expressed in JSON. An example is shown in Figure 3. Each recipe indicates the source packages, build commands, import procedure, and dependencies of a given software page. For each software package requested, a recipe and its dependencies are located and constructed into a dependency graph.

Then, for each dependency in the graph, VC3-Builder checks to see if the package is already available natively. For example, the desired compiler may already be found in `/usr/bin/gcc`. If not, then the necessary source packages are downloaded from *upstream sources* into a *cache directory*. Each package is built and installed into the *install tree* which is by default in the user's home directory.

Finally, the builder creates a temporary *sandbox* directory to run the job, starts a new shell, and *imports* each package into the environment by setting the appropriate environment variables indicated in the recipe, typically the `PATH` and the `LD_LIBRARY_PATH` or other variables specific to the package. The desired task is executed, and then the sandbox and its environment are discarded.

Several alternate modes of operation are available:

Sterile Build. Relying on local software installation can sometimes result in unexpected behavior, if local modifications are not reflected in the version information returned by tools. A sterile build will ignore all local software, build all dependencies from scratch, and include only the install tree in the user's environment.

[1]We achieve this using the musl C library [3] together with `staticperl` [16].

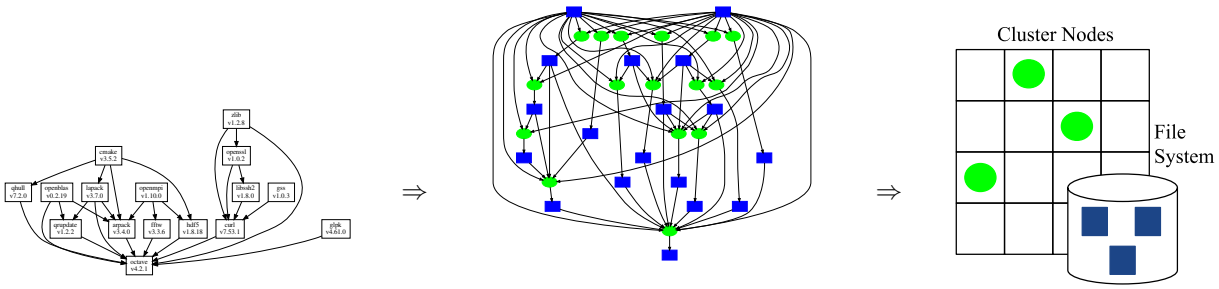


Fig. 4. Distributed Build Sequence. To accelerate a build on a cluster, the builder transforms the dependency graph into a graph of tasks executable by a workflow manager. The workflow manager then dispatches independent tasks to the cluster through the usual batch system. Each task consists of an invocation of the basic builder process, to build one package from its immediate dependencies, leaving results in the shared file system.

Sealed Package. Rather than build and install the packages, VC3-Builder can instead download the source packages and subset of recipes needed, and collect them together into a *sealed package* which is completely self-contained. This package can be used in place of an online repository for future builds, facilitating builds on cluster nodes or other machines without network access. It can also be used to facilitate archival for scientific reproducibility needs.

Shared Installations. To avoid the likely common case of duplicated downloads and installations, the user may configure a list of locations for the download cache and the install tree. All entries, except the last, in each list are used read-only. This allows the user to exploit software already downloaded or installed by an administrator or other collaborating user, while still building up their own private install tree.

V. Cluster Considerations

Consider the situation of a user who wishes to execute a large number of jobs across one or more clusters. The jobs may be submitted by hand from a user logged into the head node, or they could be delivered to the cluster via some external middleware that manages both clusters. Either way, a naive application of VC3-Builder in this context may result in some undesirable outcomes for both the user and the owners of the system. We consider several cases and the appropriate remedies.

One approach is for the user to submit every single job as a separate invocation of `vc3-builder` wrapped around the payload job. While this could technically function correctly, it would be inefficient in several different ways. The most obvious is that every single node would be performing an identical and independent built, which could be a significant waste of valuable computing resources. Even if the build were relatively fast, each job would independently download upstream sources, which could overload local network capacity, or be seen as a denial-of-service attack against the source.

A tempting remedy for this situation would be to develop a sort of coordinated build in which each node performing a build first checks for prebuilt directories and downloaded packages in a shared filesystem, and then only attempts to build whatever components are outstanding. Of course, file locks, sentinel files or similar other measures would have to be used to provide mutual exclusion for activities in progress. While this would work in principle for a distributed filesystem that provides precise consistency semantics, large parallel filesystems are notorious for having weak consistency semantics designed to support high throughput data movement, rather than distributed synchronization.

Instead, an external approach to synchronization is likely to be more robust. Instead of performing a build at each individual node, we can carry out the build once on the head node, deploying the various packages into the shared filesystem. We call this a *sequential build* if a single build job runs at once, and a *parallel build* if multiple build jobs run on different codes in the head node. Once the build is complete, then the individual jobs can run using the shared installation. In this case, we must still prefix each job with `vc3-builder` so as to verify the presence of the needed packages and import the necessary environment.

However, this introduces a new problem. A common policy of computing centers is to prohibit intensive computing activities on the head nodes, which are shared by a large number of users. A particularly complicated build can conceivably run for more than an hour while placing a significant CPU load on the head node and metadata traffic on the shared filesystem.

Instead, we can push the load out to the cluster (where it belongs) by running the builds as jobs in the local batch system; we call this a *distributed build*. Independent builds can run concurrently on different nodes, within the constraints of package dependencies. Rather than create this capability from scratch, we can express it in terms of a known problem by causing VC3-Builder to emit a *workflow* describing the steps to be carried out, and passing the execution to a *workflow manager*. This process is shown in Figure 4.

In this case, we use the Makeflow [6] workflow manager, which accepts workflow specifications in the form of the traditional Make language, where each task to be executed is a build of a single package by recursively invoking VC3-Builder. To handle the common case where a cluster node may not have access to the outside network, the “parent”

builder extracts only the necessary recipes and packages and passes them directly to the "child" builder which then operates without network access.

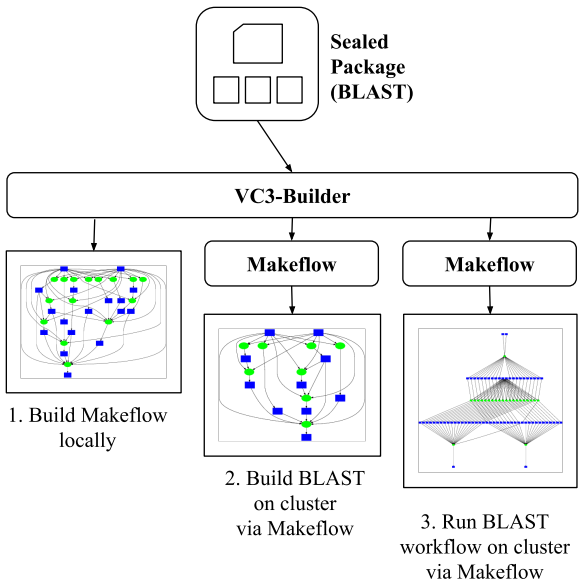
One problem remains: the workflow software itself must be delivered to the cluster for execution. Once again, we reduce this to a known problem using VC3-Builder to build Makeflow itself before generating the distributed build.

VI. Preserving Workflows

A related use case is the problem of dependency management for *scientific workflows*. As used today, a scientific workflow is a way of encoding a large graph of tasks and data that, combined together, comprise a large parallel program that runs across a cluster. Workflows are widely used in fields such as astronomy [14], bioinformatics [11], earth science [12], and more.

Most workflow languages do a good job of capturing the *structure* of a workflow, but not the *software environment* necessary to run that workflow. For example, the Makeflow Examples Archive [18] is typical in that it captures a number of workflow graphs which can be browsed and manipulated. However, to actually run the workflows requires that the user manually download and install the software invoked by each job, which is a time-consuming and error-prone process.

Instead, we can complete the description of a workflow by annotating it with sufficient information to reconstruct the software on demand. For example, suppose that we preserved the BLAST software in a sealed package of recipes and sources, along with the workflow that uses it. Then, the entire workflow can then be bootstrapped in three steps: (1) VC3-Builder builds Makeflow using local resources. (2) VC3-Builder invokes Makeflow to build BLAST from online repositories or a sealed package. (3) VC3-Builder invokes Makeflow to execute the BLAST workflow. Figure 5 depicts these steps.



Building makeflow and getting sources:	1 job, 20s
Building BLAST dependencies (distributed):	9 jobs, 11m
Running the user workflow (distributed):	5079 jobs, 86m

Fig. 5. Bootstrapping a BLAST Workflow. To bootstrap a preserved workflow, VC3-Builder reads the sealed package, builds Makeflow locally, uses Makeflow to build the scientific application, then uses Makeflow to carry out the workflow using the application.

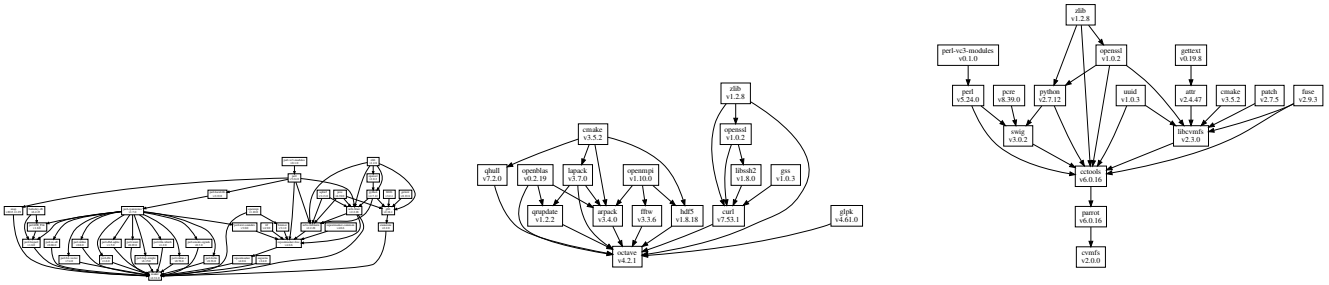
VII. Case Studies

We present results of installing three complex software installations, which build dependency graphs, builder recipe dependencies, and time execution graphs are presented in Table I.

- MAKER is a genome annotation pipeline used in the bioinformatics community. As a pipeline, it depends on several external programs, with MAKER itself written in Perl. In total, an installation of MAKER consists of 39 dependencies. Even if administration privileges are available, many of these dependencies do not have rpms or similar available. Further, some dependencies cannot be freely distributed as they have restrictive licensing. For these dependencies, VC3-Builder ask the user to manually download the sources before continuing.
- Octave is an environment and programming language to perform numerical computations, a popular and free alternative to Matlab. As a numerical computation tool, it depends on libraries such as LAPACK to perform linear algebra computations, and fftw for discrete Fourier transforms.
- CVMFS is a read-only filesystem used by the High Energy Physics community to deliver software. CVMFS is originally designed as a FUSE kernel module by which remote mount points are accessed using HTTP. As a kernel module, it cannot be installed by regular users, which would limit the computational resources available. One solution is to use `parrot` [19], a user-level system call interposition agent, which can redirect I/O operations according to several drivers to emulate unavailable filesystems. When VC3-Builder is required to install CVMFS, first it checks if access is already available (e.g., access at `/cvmfs/`), and if not, it builds parrot and executes a shell wrapped in parrot so that paths starting with `/cvmfs` become available.

	MAKER	Octave	CVMFS
Build Jobs	39	15	17
Upstream Sources	27	16	16
Size of Sources	758M	98M	171M
Size of Install	4.2G	1.2G	1.1G

We tested the installation on three different sites: University of Notre Dame CRC, PSC Bridges, and Comet. The resources at the Center for Research Computing (CRC) at University of Notre Dame are available opportunistically when they are not being used by their owners via HTCondor [20]. HTCondor



(a) MAKER Package Graph (b) Octave Package Graph (c) CVMFS Package Graph

Fig. 6. Example Applications. Three example applications constructed using the builder. MAKER is a complex pipeline of multiple extant bioinformatics applications, commonly used together via an MPI driver program. CVMFS is a user-level global-scale filesystem used widely in the high energy physics community for software distribution. Octave is a widely used symbolic mathematics toolkit.

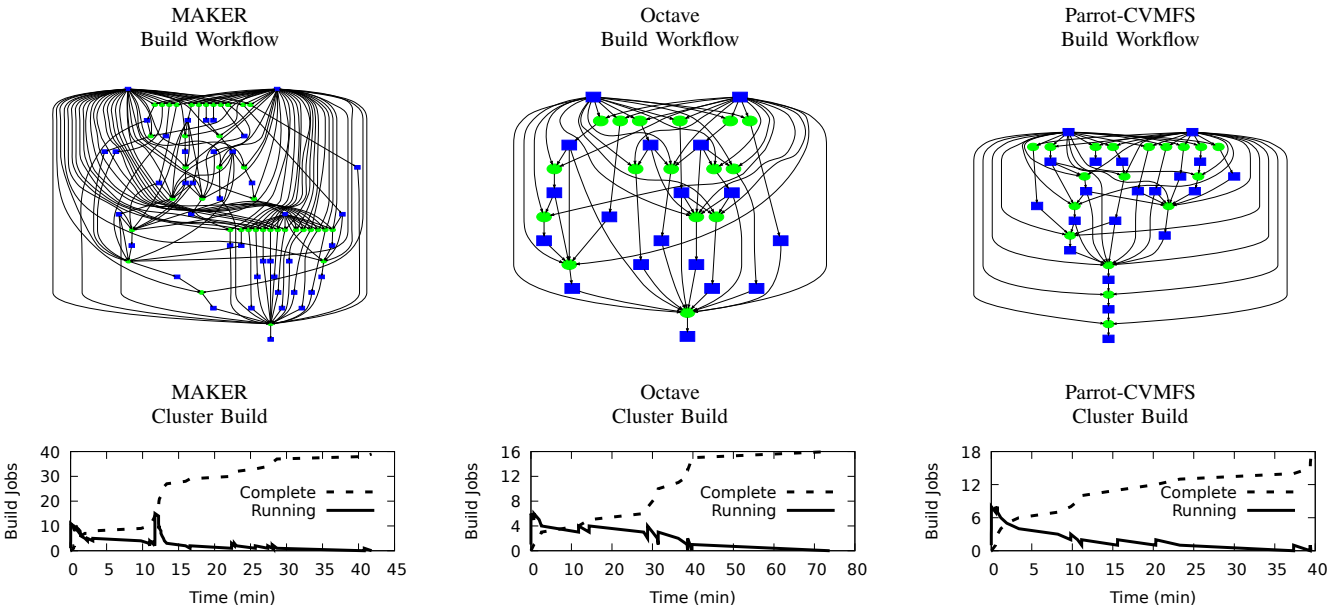


TABLE I

THREE CASE STUDIES. THE GRAPHS SHOWN ARE THE WORKFLOWS EXECUTED, ENCODING THE DEPENDENCIES SHOWS IN FIG.6. A GREEN CIRCLE REPRESENTS A SOFTWARE RECIPE, AND A BLUE RECTANGLE AN INSTALLATION DIRECTORY. THE TIME GRAPHS SHOW THE NUMBER OF CONCURRENT JOBS FOR A DISTRIBUTED INSTALLATION.

is a workload management system that is able to harness otherwise idle workstations. The Pittsburgh Supercomputing Center (PSC) Bridges is a supercomputer with more than 800 nodes which is available through the XSEDE portal [21]. Comet, also available through XSEDE, is a hybrid computer cluster with more than 1,900 nodes available. Both Bridges and Comet use SLURM [22] as their scheduling system.

Installation was done following three configurations:

- **Sequential.** A single package installed at a time.
- **Parallel.** Packages are installed concurrently as dependencies permit, in a single multicore machine.
- **Distributed.** Packages are installed concurrently as dependencies permit, on different computational nodes on the cluster using the batch system.

In the three configurations, no single package installation uses more than 4 cores at a time.

A summary of our results is presented in Table II. At Notre Dame we ran each of the cases in sequential and parallel on a front end machine, and distributed was run using HTCondor. Running on the batch system was slower than the other two options as jobs needed to wait to be scheduled, and further, jobs would sometimes be suspended given particular policies for HTCondor at Notre Dame. While slower, using the batch scheduler is much friendlier to the system, as computational heavy tasks are discouraged and often forbidden, at the front-end machines.

As a comparison, observe the results on Comet with the distributed configuration using SLURM, where jobs did not spend much time on the queue and were not suspended while running. The running times are comparable to running in parallel in the front end. Of note, we were not able to install Octave using the front end, as the resources needed for such computation were not available. Finally, with Bridges, actual

MAKER

site	Notre Dame			Comet		Bridges	
mode	sequential	parallel	distributed	parallel	distributed	parallel	distributed
time	00h56m	00h17m	00h42m	00h29m	00h27m	00h23m	00h52m
concurrency	1	17	15	15	16	17	15

Octave

site	Notre Dame			Comet		Bridges	
mode	sequential	parallel	distributed	parallel	distributed	parallel	distributed
time	00h51m	00h30m	01h14m	—	00h33m	00h35m	01h30m
concurrency	1	5	6	—	6	5	5

CVMFS

site	Notre Dame			Comet		Bridges	
mode	sequential	parallel	distributed	parallel	distributed	parallel	distributed
time	00h32m	00h14m	00h40m	00h20m	00h26m	00h18m	00h31m
concurrency	1	8	8	8	8	8	8

TABLE II
RESULTS ACROSS THREE DIFFERENT SITES. VC3-BUILDER WAS USED TO INSTALL MAKER, OCTAVE, AND CVMFS AT NOTRE DAME, XSEDE COMET, AND XSEDE BRIDGES. SEQUENTIAL AND PARALLEL (FE) MODES WHERE EXECUTED UNIQUELY AT THE RESPECTIVE FRONT-ENDS, WHILE HTCONDOR AND SLURM USE THE RESPECTIVE BATCH SYSTEM. THE TIME ROW INDICATES THE OVERALL COMPILATION TIME, AND THE CONCURRENCY ROW INDICATES THE MAXIMUM NUMBER OF INDIVIDUAL SOFTWARE PACKAGES COMPILING AT THE SAME TIME.

execution of jobs was comparable to Comet, but the jobs had slowdowns as they spent more time waiting in queue (we used the RM–shared queue, which gives partial nodes, but schedules faster).

VIII. REPRODUCIBILITY

To reproduce the builds described on this paper, you can obtain a copy of the builder executable with:[2]

```
% wget https://raw.githubusercontent.com/
     vc3-project/vc3-builder/
     clusterp/vc3-builder
```

Assuming that the builder executable is in the current directory, the sequential builds can be reproduced with:

```
$ ./vc3-builder --require octave
% octave --help
% exit
$ ./vc3-builder --require maker
% maker --help
% exit
$ ./vc3-builder --require cvmfs
% ls /cvmfs/cms.cern.ch
% exit
```

To generate a local parallel build of Octave (similarly for maker and cvmfs):

```
% ./vc3-builder --require octave\
     --parallel my-build --parallel-mode local
```

For Condor, SGE, or SLURM, `-Tlocal` should be replaced with `-Tcondor`, `-Tsge`, or `-Tslurm`, respectively.

IX. RELATED WORK

There have been multiple efforts dealing with software dependency installation. Tools such as `yum` and `apt` allow a system administrator to install dependencies from a curated repository. This repository may have conflicting versions of software, but in the end, only a consistent set of packages may be installed in the system. Such packages then become available to all the users of the system. Installation is straightforward but depends on having administrator privileges, and the existence of rpm or deb packages, which are often not available in scientific software.

There are approaches that allow conflicting versions, such as Nix [8], Spack [10], 0install [9], conda [5], and GoboLinux [1]. The general idea is to install a particular version of a package in its own directory, rather than `/usr` or `/usr/local`. After installation, the packages are incorporated into the execution environment through the use of symbolic links and environment variables. The VC3-Builder follows a similar approach.

The Nix [8] system has a strict model for the management of dependencies, which allows it to provide strong functional guarantees. Software recipes are written in a custom functional language. Nix tries to be as distinct from the host system as possible, even compiling a version of a C compiler as part of its bootstrap process. This is because the installation destination not only depends on checksums of the recipes and dependencies, but also takes into account the version of the compiler used. This allows the sharing of whole swaths of binaries across systems, if they share the same installation prefix. While Nix recipes provide strong guarantees across different configurations of systems and versions, the objectives of VC3-Builder are more modest, as it focus on describing a set of versions known to execute a particular workflow, using the building chain already in a system.

In this regard, VC3-Builder is more similar to Spack [10]. Spack is a tool borne from the HPC community, in which recipes are written in Python, and that uses known building chains of common Linux distributions used in HPC, such as

[2]The full development files can be obtained from our git repository: `https://github.com/vc3-project/vc3-builder`.

RedHat or Centos. Spack developers have taken particular care in designing recipes that take the best advantage of the compilers available, with packages having different installation flavors to be as flexible as possible. In order to do this, Spack resolves particular installations as late as possible. In contrast, VC3-Builder resolves all packages to be installed beforehand, as it is needed by the parallel installations. The conda[5] system takes a similar approach, but its distribution is not lightweight, which limits its deployment as part of a batch job.

GoboLinux is a Linux distribution which aims to create a more logical and modular representation of the underlying filesystem. To accomodate the more modular representation of GoboLinux, specialized recipes are written to ensure programs install correctly. GoboLinux can be installed within another Linux distribution in Rootless mode. Rootless mode [1] places all software installed by GoboLinux within the user's home directory. We can view Rootless GoboLinux as a different way to implement our goal since a researcher could use a GoboLinux space as a sandboxed environment to install all their software at user-level. However, the user would need to handle the dependencies of their software rather than having that happen automatically. Gentoo Linux has a mode similar to Rootless GoboLinux which allows the user to install Gentoo packages via the package manager Portage, without root permissions, in the userspace. This project is called Prefix [13], and it is set up by a bootstrapping script which sets up the Prefixed Portage. Like Rootless GoboLinux, Prefix allows the user to set up a sandboxed environment and run recipes to compile their needed software.

Regarding distributed compilation, distcc[4] allows C and C++ program to be compiled across machines without the need of a shared filesystem. The machines need to be running a distcc daemon to be able to receive units of work. distcc itself is a frontend to a compiler, and it is designed with C/C++ in mind. For example, in its initial implementation, only the output of the preprocessor was sent to remote compilation. In contrast, VC3-Builder does require a shared filesystem, but does not require a local daemon, and it is not limited to C/C++ packages.

Container technology has recently become a popular method of virtualizing an environment. When compared to virtual machines, containers are more lightweight at the cost of using the same underlying kernel as the host. Containers construct their environment from an image file which contains the code, system libraries, and settings to run the contained software. Docker [17] and Singularity [15] are two popular container technologies which approach the virtualization process somewhat differently. On face value, we would think a container would be an ideal solution to our problem. However, Docker does not satisfy two important aspects of our use case which VC3-Builder addresses. First, Docker requires root access to install and run the underlying Docker daemon process, and second, the image has to be constructed in some way, even when rpm or deb files are not available. In contrast, Singularity does not require a daemon process, but it does need a setuid executable to create images.

Homebrew is a specialized package manager for MacOS. The purpose of it is to help install dependencies in MacOS which is similar to the purpose of VC3-Builder. Homebrew allows users to create their own packages. However, it assumes root privileges in order to install software which we avoid using VC3-Builder.

The 0install (Zero Install) [9] tool provides a decentralized software installation system which can install software and its dependencies at the user-level much like VC3-Builder. It reads an XML specification for the desired software which specifies the dependencies needed before the software can be installed. Once the dependencies are installed through 0install, it will then install the software. By default, all software installed through 0install is placed in a cache for the user. System-wide access can also be provided by installing to a different location. To effectively use 0install, there needs to be a large network of developers to provide online download pages with the 0install XML specification for their software. VC3-Builder addresses similar goals as 0install, however we have focused on making VC3-Builder well-suited for cloud scientific applications without much user interaction.

So far we have considered only building from source. In our case, the major challenge for binary distributions is that an executable may not be relocatable (i.e., it has a hard-coded path). This is greatly ameliorated by using true statically linked binaries, such as the ones provided by the Minos project [2]. This project not only provides binaries for hundreds of Linux packages, but also the recipes to manually build them.

X. Conclusions

This paper presents VC3-Builder, a user-level system for managing software dependencies across multiple clusters. With VC3-Builder, we aim to help with some of the more difficult social dynamics of scientific software operation. In particular, a user may be able to experiment with complex software stacks, or with unusual compiler flags, without affecting a system administrator. Once the user has created a working configuration (i.e., a set of recipes), they can share that configuration with the system administrator, who now has a concrete object to evaluate before applying any changes to the overall system. Conversely, a system administrator may communicate the working configuration to a user, such that a workload can be ported between sites. In developing further components of VC3: Virtual Clusters for Community Computation, we have found VC3-Builder to be a valuable in-house tool to share working configurations among the collaborators, as no two setups are the same (some do not have Docker, some do not have root access, etc.). We believe VC3-Builder has a wide applicability for scientific applications on cloud systems.

References

[1] Gobolinux. https://gobolinux.org.
[2] Minos: portable binaries for linux. http://s.minos.io.
[3] musl libc. https://www.musl-libc.org/.
[4] distcc. https://github.com/distcc/distcc, 2007.
[5] conda. https://conda.io, 2012.

[6] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[7] D. Bahls. Evaluating shifter for hpc applications. In *Cray User Group Conference Proceedings*, 2016.

[8] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, 2004.

[9] B. Eicher and T. Leonard. Zeroinstall. https://0install.net.

[10] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 40:1–40:12, New York, NY, USA, 2015. ACM.

[11] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol*, 11(8):R86, 2010.

[12] R. Graves, T. H. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi. Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics*, 168(3):367–381, 2011.

[13] F. Groffen, G. Amadio, M. Haubenwaller, B. Xu, and J. Callen. Prefix.

[14] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Int. J. Comput. Sci. Eng.*, 4(2):73–87, July 2009.

[15] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.

[16] M. Lehmann. Staticperl. http://search.cpan.org/dist/App-Staticperl/, 2015.

[17] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[18] D. Thain and N. Hazekamp. Makeflow examples, 2017.

[19] D. Thain and M. Livny. Parrot: Transparent User-Level Middleware for Data Intensive Computing. In *Workshop on Adaptive Grid Middleware at PACT*, 2003.

[20] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[21] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, et al. Xsede: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, 2014.

[22] A. B. Yoo, M. A. Jette, and M. Grondona. *SLURM: Simple Linux Utility for Resource Management*, pages 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.