Job Script Portability Tool: Cat-based Localization Scheme (CLS)

Craig P Steffen
(National Center for Supercomputing Applications)
csteffen@illinois.edu
University of Illinois
Urbana, Illinois, USA

Abstract

A scheme and tool for combining system-independent script fragments with system-specific localization information to make it easy to write HPC job scripts to run on any system. The structure of the fragments is discussed, with templates of how to set up new job scripts, localizations, and an example of use.

CCS Concepts

• **Computing methodologies** → *Concurrent algorithms*; • **Software and its engineering** → **Massively parallel systems**.

Keywords

Job scripts, localization, cluster portability

ACM Reference Format:
Craig P Steffen, (National Center for Supercomputing Applications). 2025. Job Script Portability Tool: Cat-based Localization Scheme (CLS) . In *Proceedings of 12th International Workshop on HPC User Support Tools (HUST 2025).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Running (or testing) code across multiple parallel HPC platforms often requires job script customization for each system. When the code and run scripts get updated, the new features need to be propagated across the platforms but merged with the existing system-specific customizations. This can be done with line-by-line merges in git but there's always the possibility of propagation error. This paper describes a scheme to separate out system localization, user customization, and functional code in Slurm (or any kind of) job scripts to aid in creation, separation, and maintenance of job scripts that need to run across multiple platforms for production codes. The implementation is called "Cat-based Localization Scheme", or CLS, and the file fragments use use the file extension ".CLS" for identification and to avoid colliding with other file extensions.

2 Philosophy

2.1 Job Script Structure

Moab[1] and Slurm[2] job engines treat job scripts the same (and the author believes, but without the ability to directly test, that PBS[3] does the same). The job script is parsed once at submit time, only interpreting lines that begin with the job-systems-specific tag ("#SBATCH" in the case of Slurm) for directives for the job system. The submission parser stops at the first active shell command[4]. Later, when the job is instantiated and the job script is run, the shell engine interprets the entire script as a shell script. Since the job-specific tags are defined so that they are shell *comments*, these type of job-specific tags are ignored. For the following discussion, this effectively divides the job script into a "static" section, containing job-system tags only interpreted by the job pre-processor, and the "active" section of the script that is interpreted by the shell at run time.

2.2 Job Script Functions

Lines in job scripts are further subdivided by three different *roles*. (The reasons for splitting these apart become apparent in the next paragraphs). The first role is the functionality of the line to run the code that it's coupled to. This corresponds to features in the code like how long it runs, what variables it needs to have set up, and so on. We refer to these as "core" statements; the core tags and statements are the main part of the job script. The second statement role to localize the code on the *system* that it's running. There may be local directory definitions, and allocation names, module loads, and so on. These are system localization statements, or "system" lines for short. The third role for statements are to customize the script for the *user* that's using it. This could be the user's email address for job status notification, or perhaps customizations for the user's runs, or for testing or other user-specific customizations. These are "personal" or "user" statements.

2.3 Script Subdivision

The job execution defines two different sections of the job script, static and active. Within that, there are three different potential roles for statements in the job script that will be typically intermixed within the static and active sections of a submitted job script. The roles are system localization, personalization, and core script. Dividing up a whole job script by both role and by section, than we can classify any statement into one of six classifications:

- static system
- static personal
- static core
- active system
- active personal
- active core

Any job script, with any job system that follows the same execution model[4] as Slurm could be structured as six sections (this paper refers to them as "fragments") that are then concatenated together to form the real job script submitted to the job system. The implementation discussed in this paper is a simple way to automate this process, and further, to use it to increase the efficiency and reduce the maintenance effort of having a script that runs on multiple disparate systems with multiple users.

3 Implementation

The implementation of this technique was to design a scheme were the six partial script fragments, as listed above, are written and stored on disk separately, and then combined by a script when it's time to submit the script to the job system. The setup assumes that script fragments representing four of the classifications exist and are set up on any computer system that the user wants to launch a job script on. The static system fragment, the static user fragment, and the active system and user fragments. The files that contain just those lines exist on disk on the HPC system, and the user has environment variables set that contain the path to each of these files.

The design of the CLS system also includes a specification for the core script itself. A core script contains static elements and active elements specific to the job script, two of the six fragments as outlined above (the static core and the active core). To form the final job script, those two must be separated out for concatenation into the final job script. The user composes the static script section and active script section in one file (the core fragment file), separated by an isolated line of "equals" signs: "=============". The concatenating script will recognize this line as the divider between those two fragments and will not propagate it into the final job script. Since a line of equals signs is not a valid line of bash, in the case that the concatenation fails somehow, that line will cause an interpretation error and the final script would fail.

The heart of the implementation of the CLS system is a tool called "cls_go.pl". This script is installed in the user's environment and placed in their path. Its one requirement argument is the name of the .CLS core script that contains the static core and active core fragments in a file, separated by a line of equals signs, as in the previous paragraph. The other four fragments of the job script are specified by the user by defining environment variables as follows:

```
CLS_STATIC_SYS_FILE
CLS_ACTIVE_SYS_FILE
CLS_STATIC_PERS_FILE
CLS_ACTIVE_PERS_FILE
```

The cls_go.pl tool can be run safely without any arguments. If it receives no arguments on the command line, it complains and says it needs the name of the CLS script fragment. If it gets one, then it checks for the existence of full paths in the four environment variables above. If those are all populated, it then concatenates the four files configured and the two parts of the file specified on the command line to a job script, which it then outputs to stdout. It presumes that the user will redirect this output to the desired job script on disk or perhaps into sbatch itself.

4 Example Script Construction

4.1 Workflow

This example workflow illustrates how a new group would deploy a CLS core script on a new system, say, the Expanse system. Alice, a senior postdoc and first user on the system would download (or pull from git) the core CLS fragment that was prototyped on Bridges. She would set up two empty files in the group's shared work area, `static_system.CLS` and `active_system.CLS` in which to define those fragments. She would also create two files in her personal subdirectory called `static_personal.CLS` and `active_personal.CLS` for their own information.

As outlined below, Alice would populate the `static_system.CLS` with group tags for the job system. She would define bash variables pertaining to the group's job scripts in `active_system.CLS`. Then she would put job tags and bash variables in `static_personal.CLS` and `active_personal.CLS` files, respectively. She would then set up environment variables (section 3) to point to those four files. With those defined, she can get the core script fragment from git and build it, and the final job script will have all the system-specific and account specific lines embedded in it as shown below.

Bob, a junior postdoc working for Alice, would log onto Expanse to run that same core script. He would need to set up his own `static_personal.CLS` and `active_personal.CLS`, but he could point his variables to the system files already set up by Alice, and be sure to be using the same resouces for his runs as Alice was.

4.2 Static System Fragment (`CLS_STATIC_SYS_FILE`)

This fragment would contain system-local definitions pertaining to job system. Typically statements in this fragment would be the location of the bash executable, and the group's allocation account, which is unique to the group and the system, but is stated in an "`SBATCH`" line. This also would probably contain a default partition to run in for that system. A minimal fragment might be as follows. (To be clear, the `bash` definition line *is* part of the static system fragment even though it's after the "begin" line, because the structure of bash scripts is that script definition must be on the first line.)

```
#!/usr/bin/bash
#### begin CLS static system fragment on expanse
#SBATCH --account=XYZ123         # allocation
#SBATCH --partition=shared       # Partition (queue)
#### end CLS static system fragment on expanse
```

4.3 Active System Fragment (`CLS_ACTIVE_SYS_FILE`)

This fragment contains group-wide system-specific definitions that can set active bash variables that the core script expects and uses. The example shows defining the "machine name" for data logging, and sets the path for an archive directory that the group is using. This fragment also shows defining flags that are required for running MPI codes on this system due to the configuration of the MPI librariles and compilers.

```
#### begin CLS active system fragment on expanse
TEST_SYSTEM="expanse"
# system-specific file system array definitions go here
```

```
TAR_DIR="${DATA}/transfer"
MY_SRUN_FLAGS=" --mpi=pmix "
#### end CLS active system fragment on expanse
```

4.4 Static User Fragment (CLS_STATIC_PERS_FILE)

This fragment is where the user can defined job attributes in SBATCH lines that pertain specifically to them, like where to send job notifications. The user could also use this section to override definitions in the above static system fragment, for instance, for testing.

```
#### begin Alice's CLS static personal fragment
#SBATCH --mail-user=Alice.Example@gmail.com
#SBATCH --mail-type=ALL
#### end Alice's CLS static personal fragment
```

4.5 Active User Fragment (CLS_ACTIVE_PERS_FILE)

```
#### begin Alice's CLS active personal fragment
MY_TARFILE[0]="some_user_2022jan07a.tar"
MY_TARFILE[1]="some_user_2022jan08a.tar"
#### end Alice's CLS active personal fragment
```

4.6 Core Fragment

Here is the example actual core fragment file specified to cls_go.pl on the command line. It contains the static core fragment, then a line of equals signs, and then the active core fragment. Note that there are variables defined in the above sections that are referred to in the active script section; since the script will only ever be run as a completed script, that's fine. This is the script that would have been composed on other systems, typically. The purpose of the CLS system is that this file can be the same across multiple systems; all the system-specific and user-specific customizations exist in the other fragments above.

```
#### begin CLS static core fragment
#SBATCH --time=02:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --job-name=CLS_untar
#SBATCH --output=untar_test_%j.out
#SBATCH --error=untar_test_%j.err
#### end CLS static core fragment
====================
####begin CLS active core fragment
if [ -z $TAR_DIR ] ; then
    echo "Variable TAR_DIR must be set!"
    exit
fi
ORIGINAL_DIR=`pwd`
DATASET_INDEX=0
MY_TARFILE=${TAR_DIR}/${FP_TARFILE[${DATASET_INDEX}]}
MY_OUTFILE=\
$ORIGINAL_DIR/tar_out_${TEST_SYSTEM}_${SLURM_JOBID}.out
tar xf ${MY_TARFILE} >& ${MY_OUTFILE}
```

4.7 Final Script

The `cls_go.pl` tool constructs the following submittable job script from the two system fragments, the two personal fragments, and the core double-fragment:

```
#!/usr/bin/bash
#### begin CLS static system fragment on expanse
#SBATCH --account=XYZ123         # allocation
#SBATCH --partition=shared       # Partition (queue)
#### end CLS static system fragment on expanse
#### begin Alice's CLS static personal fragment
#SBATCH --mail-user=hirop@gmail.com
#SBATCH --mail-type=ALL
#### end Alice's CLS static personal fragment
#### begin CLS static core fragment
#SBATCH --time=02:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --job-name=CLS_untar
#SBATCH --output=untar_test_%j.out
#SBATCH --error=untar_test_%j.err
#### end CLS static core fragment
#### begin CLS active system fragment on expanse
TEST_SYSTEM="expanse"
# system-specific file system array definitions go here
TAR_DIR="${DATA}/transfer"
MY_SRUN_FLAGS=" --mpi=pmix "
#### end CLS active system fragment on expanse
#### begin Alice's CLS active personal fragment
MY_TARFILE[0]="some_user_2022jan07a.tar"
MY_TARFILE[1]="some_user_2022jan08a.tar"
#### end Alice's CLS active personal fragment
#### begin CLS active core fragment
if [ -z $TAR_DIR ] ; then
    echo "Variable TAR_DIR must be set!"
    exit
fi
ORIGINAL_DIR=`pwd`
DATASET_INDEX=0
MY_TARFILE=${TAR_DIR}/${FP_TARFILE[${DATASET_INDEX}]}
MY_OUTFILE=\
$ORIGINAL_DIR/tar_out_${TEST_SYSTEM}_${SLURM_JOBID}.out
tar xf ${MY_TARFILE} >& ${MY_OUTFILE}
```

5 Experience So Far

The author developed the CLS scheme as a part of testing a completely unrelated software suite called "parfu"[5] (that is no longer under active development). The parfu code was being tested on ACCESS systems Darwin, Ookami, Anvil, Expanse, Stampede, Delta, and Bridges. CLS allowed the author to compose setup scripts (similar to the untar example here) and timing and testing scripts, where the main "script" sections were the same and distributed by git repositories, but with all of the localization information on "system" and "personal" files written once specific to each system and tied into the user environment. Multiple times during this development, a new version of the test scripts were propagated to the systems via git, new job scripts were built with `cls_go.pl`, submitted, and run successfully without having to integrate them in script editors.

6 Compatibility and Future Work

As noted in Section 2.1, the CLS organizational scheme will work in any job system where the division between lines interpreted statically by the job submission system and the active (bash or otherwise) body of the submitted job script is where the first active line appears. Static fragments would have to be written with an eye for which system was being used, of course (`#PBS` vs `#SBATCH`).

If there is enough interest in CLS, the author will re-release it as a separate open-source work until the MIT open-source license[6].

7 Conclusions

The author found the CLS script setup extremely useful in propagating scripts across seven ACCESS resources. This work is being presented at a Supercomputing Workshop to gauge the interest in pushing this, or something based on it, out as a separate software product for the HPC community.

8 Acknowledgments

References

[1] Adaptive Computing. Moab HPC Suite. URL https://adaptivecomputing.com/moab-hpc-suite/.
[2] SchedMD. Slurm Workload Manager, . URL https://slurm.schedmd.com/documentation.html.
[3] Altair Engineering Inc. OpenPBS: Industry-leading workload manager and job scheduler for high-performance computing. URL https://www.openpbs.org/.
[4] SchedMD. Sbatch, . URL https://slurm.schedmd.com/sbatch.html.
[5] Craig P Steffen. Parfu tar-like MPI tool for creating or extracting directory tree archives. URL https://github.com/ncsa/parfu_archive_tool.
[6] The MIT License. URL https://opensource.org/license/mit.

Received 12 August 2025; revised 19 September 2025; accepted September 2025