Continuous Integration of the HPC SW Stack in DEEP-SEA

Jülich Supercomputing Centre & ParTec AG

2024-01-16 – Final SEA Projects Workshop

The Research leading to these results has received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements nº 287530, 610476, 754304, and 955606

What We Talk About When We Talk About CI/CD

Theory: Continuous Integration and Continuous Delivery/Deployment

- Integration: Continually merge changes into the main branch of a software repository, requiring frequent building and testing
- Delivery/Deployment: Continually build and deploy the main branch of a software repository into a staging or production environment

What We Talk About When We Talk About CI/CD

Theory: Continuous Integration and Continuous Delivery/Deployment

- Integration: Continually merge changes into the main branch of a software repository, requiring frequent building and testing
- Delivery/Deployment: Continually build and deploy the main branch of a software repository into a staging or production environment

What We Actually Mean By CI

Automate development workflows with CI/CD infrastructure, integrated in our collaboration tools (i.e. GitLab)

CI/CD Infrastructure

GitLab CI/CD

GitLab comes with it's own CI/CD infrastructure

- Automated, programmable actions on new commit, merge request, tag, release, manually, ...
- Configurable for each GitLab project/repository
- Executed as *pipelines* running in different environments (shell processes, docker containers, ... on dedicated system) and can store the resulting files as *artifacts*

GitLab CI/CD in HPC

Jacamar CI

A *GitLab runner* for HPC environments

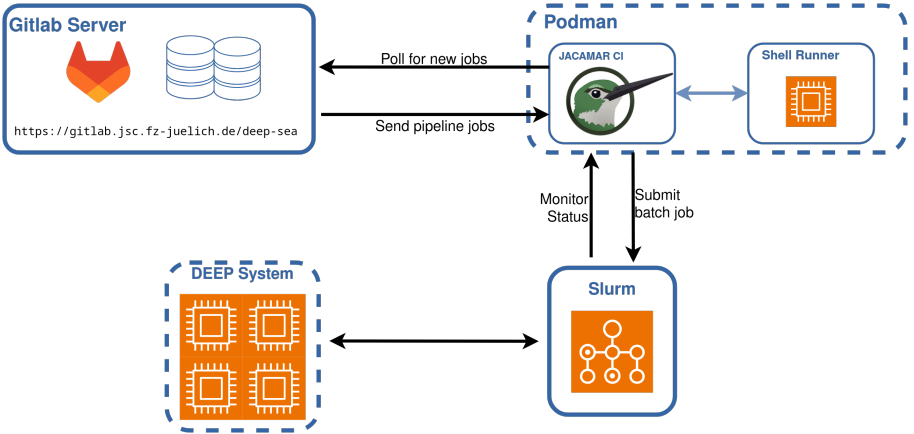
GitLab CI/CD in HPC

Jacamar CI

A *GitLab runner* for HPC environments

- Developed as part of the ECP, runs on the DEEP system
- Can run CI/CD pipelines on dedicated machine (*shell runner*), in docker container, or as Slurm batch jobs!
- Manages permissions and ensures mapping GitLab users to user accounts on cluster

GitLab CI/CD in DEEP-SEA



What CI/CD Can Do For You

Typical Use Cases for CI/CD Infrastructure

What Could We Do With CI/CD

- Automatically run tests and checks for new merge requests
- Generate new releases from a repository for every tag/release/merge
- Automatically deploy new releases and monitor the results
- Build complex workflows between different repositories or projects

What CI/CD Did For DEEP-SEA

Why are we here now?

Problem:

We want to apply more standard software engineering techniques for developments in DEEP-SEA projects

Solution:

Let us find out where we can utilize CI/CD

CI Usage in DEEP-SEA

In Individual Projects

- Run automated test suites
- Perform style and consistency checks

Automated Benchmarking Runs

Automated Build and Deployment of the Software Stack

CI/CD For Automated Software Deployment

Software and Build Management in HPC

HPC software stacks are large and *complicated*

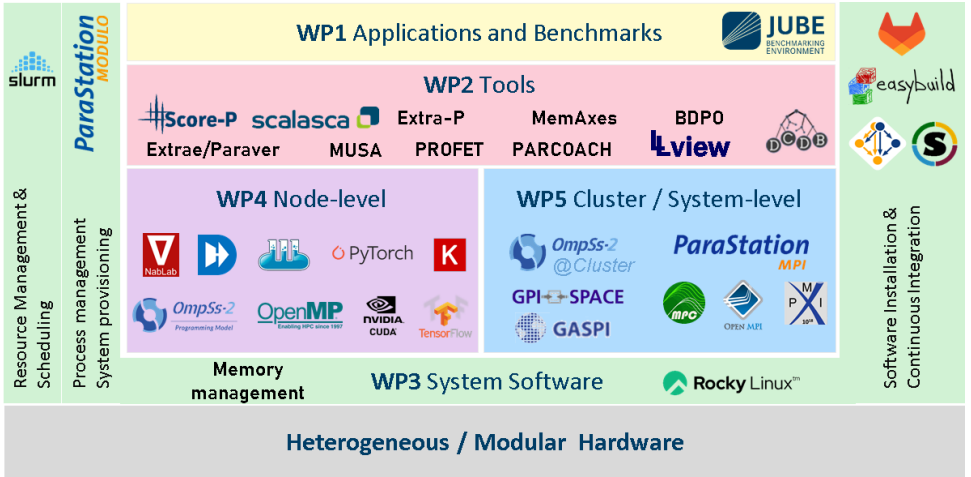
- Vast number of applications, libraries, tools
- Available with multiple compilers and MPI implementations
- ...

Individual software building and packaging is already hard:

- Different languages with specific build and packaging systems, sometimes with more than one popular solution (looking at you Python)
- Different build systems with platform dependent options

Lots of additional work is needed!

The DEEP-SEA Software Stack

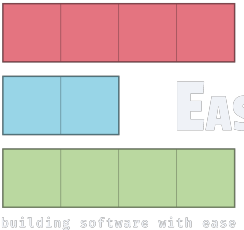


Software and Build Management: Spack

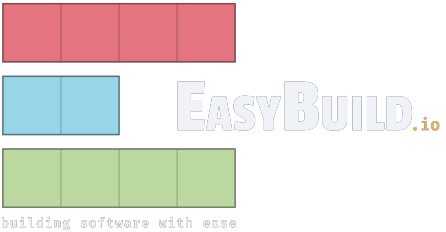
- Self contained package management system, suitable for both individual users and site admins
- Upstream support for many software packages commonly used in HPC environments
- Comes with GitLab CI/CD integration, already
- Originally developed @ LLNL, but also used by partners in the DEEP-SEA project

Software and Build Management: EasyBuild

- Build and installation framework for HPC software focused on site admins
- Tight integration of the host system
- Used at JSC and many other European HPC centers



Automated Deployment on the DEEP System



EasyBuild was chosen for software management on the DEEP system.

The DEEP-SEA EasyBuild CI/CD

Requirement: Software Re-build and Deployment

Let's continually build a *bleeding edge* software stage for the DEEP-SEA project!

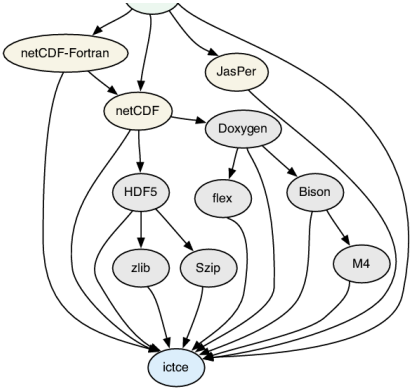
- Software should be re-built if there are changes (as decided by individual software's developers)
- (Optionally) *Reverse dependencies* should be re-built for that software
- Everything should be automatically deployed to the DEEP system

This integrates *everything* and should be usable in other CI/CD workflows!

Requirement: Reverse Dependency Re-building

Example:

- `netCDF` depends on `HDF5`
- If `HDF5` has a new version, then `netCDF` should get re-built, too



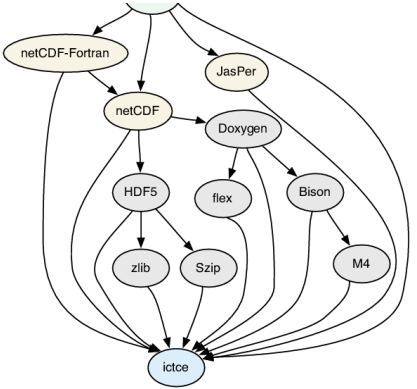
Requirement: Reverse Dependency Re-building

Example:

- `netCDF` depends on `HDF5`
- If `HDF5` has a new version, then `netCDF` should get re-built, too

How far should the reverse dependency rebuilt go?

- Question for users!



Reverse Dependencies with EasyBuild

Problem:

EasyBuild has no built-in mechanism to query reverse dependencies of a package

Solution:

EasyBuild can dump the dependency graph of a given list of EasyBuild recipes.

- We dump dependency graph of *all recipes* installed on DEEP
- Implement our own reverse dependency search on the resulting graph file

This introduces limitations on the scope of the reverse dependency resolution!

Implementation: EasyBuild Recipes

EasyBuild Recipes

To have separate versions for each CI run, we need to modify EasyBuild recipes

- EasyBuild supports changing recipes software name, version from the command line
- But not for dependencies

EasyBuild Recipe Rewriting

We built a more general solution that parses and rewrites EasyBuild recipes.

Better access to EasyBuild's own parsing and transformation functionalities would help here.

```
name = 'Scalasca'
version = '2.6.1'

toolchain = {
    'name': 'gpsmpi',
    'version': '2022a'
}

source_urls = ['...']
sources = [SOURCELOWER_TAR_GZ]
checksums = ['...']

dependencies = [
    ('CubeGUI', '4.8'),
    ('CubeLib', '4.8'),
    ('OTF2', '3.0.2'),
    ('Score-P', '8.0'),
]
```

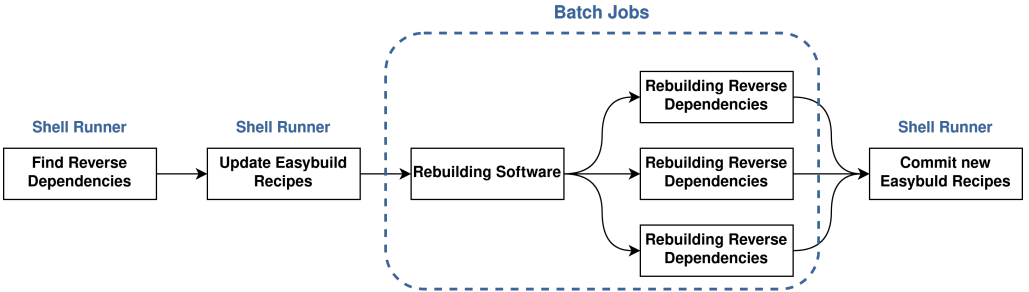
Implementation Putting Together a Pipeline

Implementation Putting Together a Pipeline

1. Software repository pipeline prepares new release
   - Sources need to be prepared and packaged to be installed by EasyBuild

Implementation Putting Together a Pipeline

1. Software repository pipeline prepares new release
   - Sources need to be prepared and packaged to be installed by EasyBuild
2. The EasyBuild repository pipeline performs the re-build and deployment



Practical Example: ParaStation MPI Stack

ParaStation

DEEP Projects

ParaStation CLUSTERTOOLS

Tools for Provisioning and Management

- System management CLI
- Image management
- Rolling updates
- Stateless & stateful booting
- Post-install configuration
- Slurm integration
- Distributed database for system configuration
- HealthChecker integration

ParaStation HEALTHCHECKER

Integrity of the Computing Environment

- Automated error detection & error handling
- Various hook-in points
- No interference with jobs
- TicketSuite integration
- Highly configurable

- 100+ tests (HW/SW):
- Node/System/Fabric level

ParaStation TICKETSUITE

Issue Tracking on System Level

- Manual and automatic ticket creation
- Prioritization
- Routing/Triage
- Documentation and central information hub
- Maintenance planning
- Interfaces with external ticketing systems

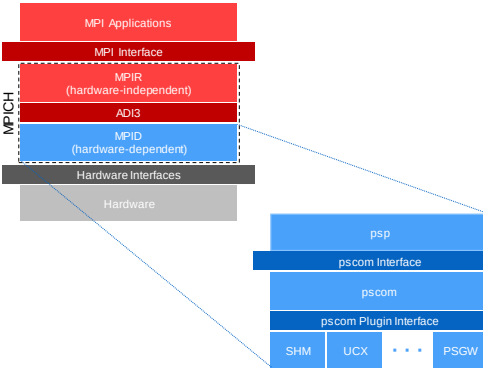
ParaStation MPI

Execution Environment and MPI Library

- MPI-4.0-compliant
- MPICH ABI-compatible
- Supports multiple interconnects in parallel
- Modularity support
- Network bridging
- PMIx support
- Full Slurm integration

ParaStation MPI

- Based on MPICH 4.1.2
  - Support MPICH tools for tracing, debugging, etc.
  - Integrates into MPICH on the MPID layer by implementing an ADI3 device
  - The PSP Device is powered by pscom—a low-level point-to-point communication library
  - Support the MPICH ABI Compatibility Initiative
- Support for various transports/protocols via pscom plugins
  - Support for InfiniBand, Omni-Path, BXI, etc.
  - Concurrent usage of different transport
  - Transparent bridging between any pair of networks enabled by gateway capabilities
- Additional features
  - MSA awareness
  - Support for malleability
  - Enhanced PMIx support
- Proven scalability



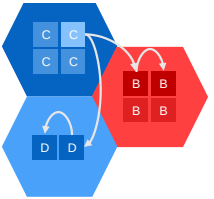
ParaStation MPI

MSA Awareness

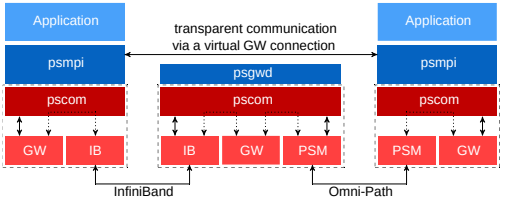
- Support for multi-level hierarchy-aware collectives
  - Optimize communication patterns to the topology of the MSA
  - Assumption: Inter-module communication is the bottleneck
  - Dynamically update the communication patterns (experimental)

- API extensions for accessing modularity information
  - New MPI split type for communicators (`MPIX_COMM_TYPE_MODULE`)
  - Provide the module id via the `MPI_INFO_ENV` object

- MPI Network Bridging
  - Connect any pair of interconnect and protocol
  - Transparent to the application layer



Hierarchical Bcast
(MSA-aware)



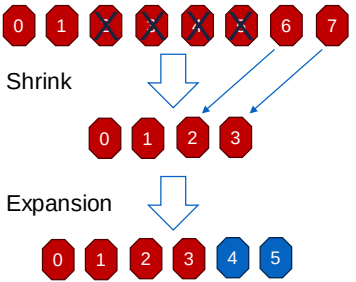
Transparent Network Bridging

Malleability for MPI

- Dynamic resource adaptations within an MPI application
  - Adding or removing of HPC resources during job run time
  - Ensure maximum MPI standard compliance
  - Exploit MPI-4 features (e.g., MPI Sessions)
  - Dense, monotonic MPI rank numbering (i.e., no gaps or overlaps)

- Usage Models
  - Job-initiated (according to current job needs)
  - Scheduler-initiated (maximize system utilization)
  - Externally initiated (based on application models)

- Initially, focus on Job-initiated malleability

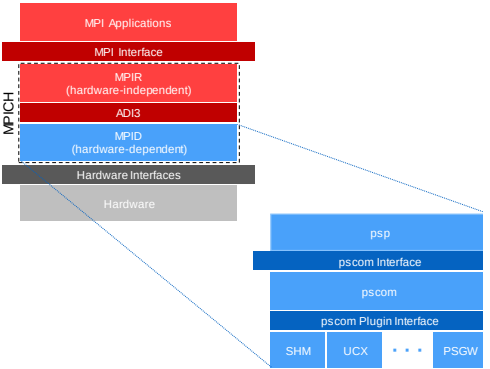


Integration into the DEEP-SEA CI

- Communication stack with two components
  - Upper MPICH-based MPI layer (psmpi)
  - Lower-level point-to-point communication layer (pscom)

- Goals
  - Updates of the pscom should automatically trigger re-builds of the upper psmpi layer
  - Avoid re-builds of the whole DEEP-SEA SW stack upon updates to the upper psmpi layer *(→ the MPI API and ABI stay stable!)*

- Approach
  1) Create source tarball in the source repository (psmpi or pscom)
  2) Decide whether to re-build *inverse* dependencies
     - *TRUE for pscom*
     - *FALSE for psmpi*
  3) Trigger downstream pipeline in the DEEP-SEA EasyBuild Repository
     - *Generate EB script specific to the previously generated tarball*
     - *Build the component*
     - *Deploy on the DEEP system*



ParaStation MPI

Preparing ParaStation MPI Sources

- Autotools are used for psmpi; hence we have to create the release tarball first

```
create_tarball:
    stage: prep
    script:
    - module load Autotools
    - module load Python
    - echo "Pipeline triggered by and commiting as \${GITLAB_USER_LOGIN}"
    - echo "Providing tarball on the DEEP system"
    - mkdir -p -m 777 /path/to/shared/folder
    - python ./scripts/release.py \
        --suffix=${CI_MERGE_REQUEST_TARGET_BRANCH_NAME:-${CI_COMMIT_BRANCH}}
[...]
```

- And upload it to a shared folder

```
[...]
    - cp psmpi-*.tar.gz /path/to/shared/folder/${CI_PIPELINE_ID}.tar.gz
[...]
```

Preparing pscom Sources

- For pscom no additional files have to be generated as it uses CMake

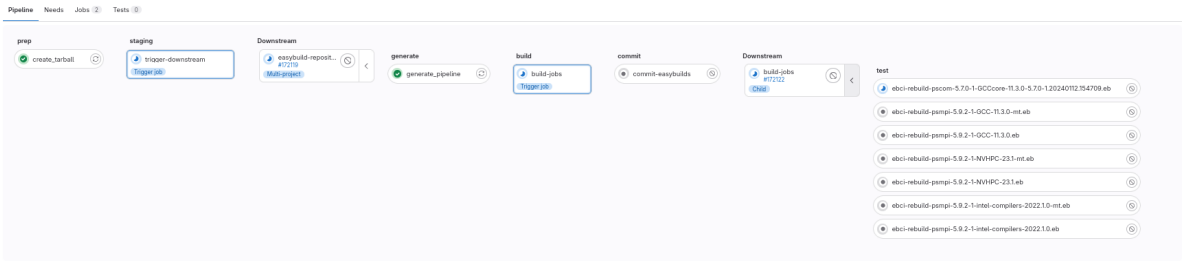
```
create_tarball:
    stage: prep
    script:
    - echo "Pipeline triggered by and commiting as ${GITLAB_USER_LOGIN}"
    - echo "Providing tarball on the DEEP system"
    - mkdir -p -m 777 /path/to/shared/folder/pscom
    - git archive \
        --format=tar.gz -o /path/to/shared/folder/pscom/${CI_PIPELINE_ID}.tar.gz HEAD
[...]
```

Triggering EasyBuild CI (pscom)

- Trigger re-build. For pscom: build inverse dependencies so that psmpi is built as well

```
[...]
trigger-downstream:
    stage: staging
    variables:
        BUILD_INVERSE_DEPENDENCIES: 'TRUE'
        EB_FILE_CURRENT: 'pscom-5.7.0-1-GCCcore-11.3.0.eb'
        SRC: '${CI_PIPELINE_ID}.tar.gz'
        REBUILD_ONLY: 'psmpi'
    trigger:
        project: path/to/easybuild-repository/in/gitlab
        strategy: depend
        branch: ci-2023-dev
```

Running the Pipeline



Artifacts Available

```
    $ module use /p/project/deepsea/ci-stage-2023-dev/easybuild/modules/all
    $ module avail

    ---------- /p/project/deepsea/ci-stage-2023-dev/easybuild/modules/all ----------
[...]
    Compiler/GCCcore/11.3.0/pscom/.5.7.0-1.20231211.122941              (H,u)
    Compiler/GCCcore/11.3.0/pscom/.5.7.0-1.20231211.124852              (H,u)
    Compiler/GCCcore/11.3.0/pscom/.5.7.0-1.20231211.142922
[...]

    Where:
    g:  built for GPU
    u:  Built by user

    Use "module spider" to find all possible modules.
    Use "module keyword key1 key2 ..." to search for all possible modules matching
    any of the "keys".

    $ module load Compiler/GCCcore/11.3.0/pscom/.5.7.0-1.20231211.142922
```

Summary

Lessons Learned

The EasyBuild CI now works for software in DEEP-SEA!

Nix would fix this

Lessons Learned

The EasyBuild CI now works for software in DEEP-SEA!

EasyBuild currently does not have a clear story for our CI/CD use case

- EasyBuild misses some features for this use case:
  - No consistent tracking of installed packages → dependency management not script-able
  - Functionality for modifying recipes exists within EasyBuild, but not readily accessible
  - Running EasyBuild in isolation of main system is difficult
- Implementation of missing functionality outside of EasyBuild tends to be fragile

Nix would fix this

Lessons Learned

The EasyBuild CI now works for software in DEEP-SEA!

EasyBuild currently does not have a clear story for our CI/CD use case

- EasyBuild misses some features for this use case:
  - No consistent tracking of installed packages → dependency management not script-able
  - Functionality for modifying recipes exists within EasyBuild, but not readily accessible
  - Running EasyBuild in isolation of main system is difficult
- Implementation of missing functionality outside of EasyBuild tends to be fragile

Nix would fix this

User training and UX is important to implement CI/CD for a project!

Thank You

www.deep-projects.eu

@DEEPprojects

DEEP Projects

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements nr 287530, 610476, 754304, and 955606