Capítulo
3

Are you root? Experimentos Reprodutíveis em Espaço de Usuário

Jessica Imlau Dagostini, Vinicius Garcia Pinto,
Lucas Leandro Nesi, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

O minicurso aborda o gerenciamento de pacotes de software e a reprodutibilidade de experimentos. Gerir pacotes em ambiente de usuário pode ser desafiador, caso o mesmo não possua devidos conhecimentos do tema. Todavia, tendo tal conhecimento, é possível não só corretamente utilizar ambiente de supercomputadores como também criar e gerenciar ambientes de forma a torná-lo reprodutível. O presente minicurso tem como objetivo apresentar técnicas e comandos para criar ambientes reprodutíveis utilizando o gerenciador de pacotes Spack e criando contêineres com Docker e Singularity.

3.1. Introdução

O método científico, pilar da ciência moderna, se baseia na observação controlada de eventos de maneira que os fatos sejam verificáveis e hipóteses possam ser testadas. Esse processo culmina em um entendimento da realidade, uma teoria científica. Tal teoria, uma vez exposta, gera implicações e conclusões sobre o mundo que por sua vez leva a novos experimentos a serem observados de maneira controlada. Esse ciclo gerador de conhecimento toma por base a necessidade de que todos os experimentos sejam suficientemente reprodutíveis. De fato, qualquer teoria científica deve ser constantemente confirmada ou refutada com novas observações.

A Ciência da Computação é parte integral do mundo atual e deve seguir os preceitos do método científico no que diz respeito à investigação, à descoberta de novos algoritmos, etc. Na área de Processamento de Alto Desempenho (PAD), por exemplo, novos algoritmos e estratégias para melhorar o desempenho de aplicações paralelas e sistemas computacionais são frequentementes apresentados. É de extrema importância que os ganhos computacionais observados a partir de um algoritmo sejam verificáveis de maneira reprodutível. Neste sentido, a computação, de uma maneira geral, necessita de um

ferramental que permita a instalação de ambientes de testes controlados. Para se obter um ambiente controlado, é necessário que toda a pilha de *software* (sw) e *hardware* (hw) seja configurável. Assim, aumentam-se as chances que as observação neste ambiente sejam mais fáceis de serem reproduzidas ou repetidas por outros pesquisadores.

Ter um controle completo sobre toda a pilha de sw/hw é desafiador pois frequentemente o ambiente computacional de alto desempenho – um cluster ou supercomputador – é suficientemente específico do ponto de vista de hw e limita a capacidade de customização da pilha de sw (difícil acesso aos direitos de superusuário). Técnicas de virtualização para alto desempenho (NUSSBAUM et al., 2009) aportam um caminho possível para controlar o ambiente, embora nem sempre possíveis em plataformas de PAD. Alternativas em nível de usuário são preferíveis pois, além de permitir independência da configuração, são facilmente postas em prática sem envolver os administradores das plataformas.

Historicamente, NIX (DOLSTRA; JONGE; VISSER, 2004) foi uma das primeiras ferramentas a empregar assinaturas *hash* para ter um controle de versões de sw e suas variantes. Mais recentemente, GNU Guix (COURTÈS; WURMUS, 2015) (<https://guix.gnu.org/>) provê gerenciamento de pacotes utilizando transações, tal qual o conceito visto em banco de dados, para a construção de ambientes reprodutíveis. Tanto NIX quanto GNU Guix exigem que a ferramenta em si seja instalada com permissões de superusuário. Outras ferramentas, tais como Homebrew (HOWELL, 2017) e Spack (GAMBLIN et al., 2015) se diferenciam por ter uma instalação puramente em nível de usuário, tanto para a ferramenta em si quanto para os sw instalados. Ambientes de processamento de alto desempenho com múltiplos usuários normalmente tem demanda de sw e versões de sw diversos. Nesses ambientes, Spack se destaca por ter sido concebido especificamente às demandas de plataformas de alto desempenho.

Este minicurso apresenta a ferramenta Spack, que é um gerenciador de pacotes de sw multi-plataforma que permite compilar e instalar múltiplas versões e configurações de sw. Dentre outras soluções possíveis (discutidas na Seção 3.5), Spack tem a vantagem de oferecer uma sintaxe dita de "especificação" suficientemente simples mas capaz de capturar as diferentes possibilidades de instalação dos pacotes. Escrito em Python, Spack é extensível na medida que receitas para instalação de outras ferramentas possam ser criadas.

3.1.1. Instalação do Spack

Para instalar Spack, na linha de comando *bash*:

SH

```
git clone https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
spack --help
```

3.1.2. Organização do documento

O texto deste minicurso está organizado em três partes, seguindo como base o tutorial da equipe mantenedora do Spack no SC'20 (GAMBLIN et al., 2015) e disponível em

inglês no site <https://spack-tutorial.readthedocs.io/en/latest/>. A Seção 3.2 apresenta o processo de configuração de ambientes isolados de instalação, para um uso inicial da ferramenta spack. A Seção 3.3 lista os comandos necessários para a configuração e utilização de ambientes reprodutíveis através da possibilidade de criação de containers. A Seção 3.4 detalha como se emprega a linguagem de programação Python para se definir receitas de instalação. Utilizaremos como exemplo a ferramenta `pajeng` (SCHNORR, 2021). Enfim, a Seção 3.5 discute outras ferramentas equivalentes ou semelhantes à Spack e finaliza o minicurso com a principal mensagem a ser levada como conhecimento.

3.2. Configuração de ambientes Spack isolados

Após a instalação do spack demonstrada na introdução, todos os comandos estão disponíveis pela invocação do executável `spack`. Veremos os comandos para a criação de ambientes isolados, para a instalação de pacotes, para a configuração de compiladores. Veremos no final como funcionam os arquivos de configuração e como estes podem ser compartilhados de maneira que outras pessoas possam reproduzir a pilha de software em outra máquina.

3.2.1. Criação de ambientes para isolamento da instalação de pacotes

Os softwares instalados com spack podem ser isolados em ambientes. Cada ambiente possui seus pacotes, configurações, enfim, os comandos serão executados internamente. A principal vantagem de sua utilização é a possibilidade de ter vários ambientes distintos, independentes, que podem ser compartilhados. Mesmo assim, seu uso é opcional, já que pacotes podem ser instalados no ambiente padrão de cada usuário. A diretiva `env` permite gerenciar ambientes. Por exemplo, para a criação de um novo ambiente chamado *meuambiente* o seguinte comando pode ser utilizado:

SH

```
spack env create meuambiente
```

Informações de ambientes spack podem ser gravadas e compartilhadas com arquivos `.lock` ou `.yaml`. Para a criação de um ambiente descrito por um arquivo no formato `yaml`, o seguinte comando pode ser utilizado:

SH

```
spack env create meuambiente arquivo.yaml
spack install
```

Quando for necessário trocar ou acessar um ambiente nomeado, a diretiva `env activate` é utilizada. Por exemplo, para ativar o ambiente chamado *experimentos* pode-se utilizar o seguinte comando:

SH

```
spack env activate experimentos
```

Para desativar e sair de um ambiente, utiliza-se a diretiva `deactivate`:

SH

```
spack env deactivate
```

3.2.2. Instalação de pacotes e suas dependências

A grande vantagem da utilização do spack é a instalação de pacotes em nível de usuário com múltiplas opções por pacote e a instalação automática de dependências. Para a instalação de um pacote com spack, utiliza-se a diretiva `install` com o nome do pacote. Por exemplo, para instalar o pacote `zlib`:

SH

```
spack install zlib
```

O pacote será baixado, configurado e compilado utilizando o compilador padrão detectado pelo spack. Caso o sistema tenha múltiplos compiladores pode-se adicionar `%compilador` após o nome do pacote para escolher o compilador a ser utilizado. Por exemplo, caso seja desejado utilizar o compilador `clang`:

SH

```
spack install zlib %clang
```

Os pacotes ainda podem possuir várias opções de compilação e instalação. Cada possibilidade de instalação é chamada de variante. Cada variante diferente possui um *hash* diferente. As variantes para cada pacote podem ser listadas com `info`. O seguinte comando pode ser utilizado para verificar as variantes do `zlib`:

SH

```
spack info zlib
```

Por exemplo, o pacote `zlib` pode ser compilado como uma biblioteca compartilhada ou estática. O padrão é compartilhada. Para ativar uma variante utiliza-se + (o símbolo mais) e o nome da variante. Para negar utiliza-se ~ (o símbolo til). Caso deseja-se que o pacote `zlib` seja compilado como uma biblioteca estática podemos negar a opção `shared` utilizando portanto o símbolo ~:

SH

```
spack install zlib~shared
```

Ainda é possível alterar variáveis de ambiente para a instalação dos pacotes. Para tal, deve-se passar o nome da variável após o pacote com a declaração desejada. Por

exemplo, no comando a seguir, pode-se adicionar o parâmetro `-fPIC` na variável `CFLAG` fazendo que o pacote e toda sua pilha de dependências seja compilada com essa variável de ambiente.

SH

```
spack install zlib cflags="-fPIC"
```

As versões dos pacotes disponíveis nos repositórios base do spack podem ser visualizadas utilizando a diretiva `versions` seguido do nome do pacote. Para instalar uma versão específica utilizamos `@` com a versão. Assim, para mostrar as versões do pacote `zlib` e para instalar a versão `1.2.10`, utiliza-se os seguintes comandos:

SH

```
spack versions zlib
spack install zlib@1.2.10
```

Os pacotes podem ter várias dependências. O comando `spec` é utilizado para mostrar todas as dependências de um pacote com as opções a serem concretizadas. Durante a instalação de um pacote, pode-se realizar as mesmas customizações de versões e variantes na instalação de dependências. Para isto, basta especificar a dependência com `^` e as opções desejadas. Os comandos abaixo demonstram a verificação das dependências do pacote `tcl` e, em seguida, a instalação utilizando a versão `1.2.8` de sua dependência `zlib`. Ambos os pacotes serão compilados utilizando o compilador `clang`:

SH

```
spack spec tcl
spack install tcl ^zlib @1.2.8 %clang
```

Para mostrar todos os pacotes instalados pode-se utilizar a diretiva `find`. Esta ainda tem as opções: `-d`, para mostrar as dependências; `-v`, para mostrar as opções de variantes; e `-l`, para mostrar a identificação única (*hash*) de cada instalação. Para ver todas as instalações do `zlib` é utilizado, por exemplo, o seguinte comando:

SH

```
spack find -d -v -l zlib
```

Existem várias formas para utilizar um pacote instalado. Em uma primeira possibilidade, pode-se utilizar a opção `load` para carregar e atualizar as variáveis de ambiente (`PATH`, `LD_LIBRARY_PATH` e outras que possam ser considerada necessárias) com os locais de instalação do pacote e suas dependências. Outra possibilidade é criar um diretório com a estrutura tradicional de instalação de pacotes e as dependências necessárias (`bin/`, `lib/`, `include/`) utilizando a diretiva `view` e a opção `soft`. Os comandos a seguir mostram o emprego destas duas maneiras com o pacote `zlib`, sendo que o segundo o faz no diretório *pasta*.

SH

```
spack load zlib@1.2.8
spack view soft pasta zlib@1.2.8
```

Para remover um pacote utiliza-se a diretiva `uninstall`. Por exemplo, para remover o pacote `tcl`:

SH

```
spack uninstall tcl
```

Caso o pacote seja uma dependência de outros pacotes, deve-se desinstalá-lo utilizando a opção `--dependents`. Assim, todos as dependências também serão desinstaladas em cascata. Caso múltiplas variantes estejam instaladas, pode-se remover todos as instalações de um pacote com a opção `--all`. No caso para remover todas as variantes do `zlib` e todos os pacotes que são dependentes dele pode-se utilizar o seguinte comando:

SH

```
spack uninstall --all --dependents zlib
```

3.2.3. Controlando a coexistência de diversos compiladores

Uma das possibilidades do spack é o emprego de diversos compiladores. Todos os comandos de instalação permanecem os mesmos, entretanto, diferentes compiladores e versões podem ser utilizados para a instalação de toda a pilha de *software*. Para verificar os compiladores disponíveis no spack, pode-se utilizar a diretiva `compilers`:

SH

```
spack compilers
```

Caso seu sistema possua compiladores que ainda não foram encontrados, pode-se utilizar a diretiva `compiler find` para o spack tentar localizá-los automaticamente.

SH

```
spack compiler find
```

Quando a localização automática falhar, provavelmente porque os compiladores não estão presentes no `PATH`, pode-se adicioná-los manualmente com a diretiva `compiler add` e o caminho absoluto para o compilador:

SH

```
spack compiler add /local/compilador
```

Após os compiladores serem adicionados, pode-se fazer uso na instalação dos pacotes com a opção `%` e a especificação do compilador. Esta especificação pode conter a versão do compilador caso mais de uma versão esteja disponível.

3.2.4. Arquivos de configuração e opções

O spack possui diversos arquivos de configurações. Estes arquivos podem estar localizados em diretórios diferentes para escopos de trabalho distintos. Quando diferentes configurações são encontradas, a ordem de precedência é a listagem da seguinte tabela:

Escopo de Trabalho	**Local ou Diretório**
Linha de Comando	Informado diretamente pelo usuário
Diretório	Especificado com `--config-scope`
Usuário	`~/.spack/`
Site	`$SPACK_ROOT/etc/spack/`
Sistema	`/etc/spack/`
Padrão	`$SPACK_ROOT/etc/spack/defaults/`

Os vários arquivos de configuração do spack estão todos no formato `.yaml` e são brevemente apresentados na tabela a seguir.

Arquivo	**Objetivo e opções de configuração**
compilers.yaml	Compiladores, locais, flags padrões de compilação
packages.yaml	Pacotes e variantes, instalações locais do sistema
config.yaml	Funcionamento do spack
mirrors.yaml	Espelhos de onde se baixam os fontes
modules.yaml	Módulos do spack
repos.yaml	Listagem de repositórios de softwares alternativos ao oficial

3.2.5. Compartilhando configurações

Um dos recursos disponíveis no spack é o compartilhamento de configurações ou ambientes com todas as informações sobre os pacotes instalados. Para conseguir o arquivo que contenha toda estas descrições, pode-se utilizar a diretiva `cd -e meuambiente` para se deslocar ao diretório do ambiente. Neste diretório encontram-se os arquivos `spack.yaml` e `spack.lock`. O arquivo `spack.yaml` registra os pacotes a serem instalados e suas opções, entretando tal listagem não encontra-se finalmente concretizadas (opções como versão podem mudar em plataformas diferentes ou versões do spack diferentes). O arquivo `spack.lock` descreve toda a pilha de *software* efetivamente concretizada, versões utilizadas, configurações e inclui o ambiente utilizado. Desta forma, o arquivo `spack.lock` só poderá ser utilizado em ambientes que possuem as mesmas configurações (CPU, compiladores, etc). Estes arquivos podem ser compartilhados para outros usuários e sistemas que terão uma réplica da pilha de *software*. Esta opção auxilia na reprodutibilidade dos experimentos. O comando a seguir ilustra como encontrar estes arquivos para o ambiente `meuambiente`:

SH

```
spack cd -e meuambiente
ls
```

A leitura destes arquivos ocorre com o comando `env create` como discutindo anteriormente.

3.3. Utilização de ambientes reprodutíveis

Além da possibiliadde de criar ambientes com as configurações específicas de pacotes necessários para a execução de alguma aplicação – facilitando assim a reprodutibilidade de experimentos – também é possível criar containers a partir do Spack. Containers vêm ganhando espaço com a comunidade científica em HPC, uma vez que eles provêm ambientes portáteis para a reprodução mais fiel de experimentos, e podem ser executados em qualquer recurso computacional a partir de um único arquivo de imagem (KURTZER; SOCHAT; BAUER, 2017). Alguns estudos (TORREZ; RANDLES; PRIEDHORSKY, 2019) (ALLES; CARISSIMI; SCHNORR, 2018) demonstram que containers tem uma interferência muito pequena (e por vezes nula) no tempo de execução de aplicações paralelas que são computacionalmente intensivas. Apresentamos a seguir a criação, a partir de ambientes do spack, de containers Docker e Singularity, seguido de um detalhamento de configurações adicionais que influenciam nestes processos.

3.3.1. Criando containers Docker

Uma vez criado um ambiente, pode-se empregar a diretiva `containerize` para transformá-lo em um container. Com esta diretiva, o Spack cria o arquivo de configuração inicial para a construção da imagem base do container. O Spack suporta a criação de arquivos de configuração para containers Docker (MERKEL, 2014) e Singularity (KURTZER; SOCHAT; BAUER, 2017), sendo este com maior foco para uso em processamento de alto desempenho. Mesmo assim, como podemos criar um container Singularity a partir de uma imagem Docker, este material se inicia com a geração de um Dockerfile.

Para criar o Dockerfile, devemos acessar o diretório do ambiente que pretendemos transformar em um container. Os ambientes spack ficam normalmente no diretório `./var/spack/environments/` a partir de onde o spack foi instalado. Assumindo que temos o ambiente nomeado `erad` e queremos transformá-lo em um container, basta nos deslocarmos ao diretório do ambiente e lançar a diretiva `containerize`, assim:

SH

```
cd ~/spack/var/spack/environments/erad
spack containerize > Dockerfile
```

A partir do arquivo `Dockerfile` gerado, é possível criar uma imagem Docker seguindo os comando padrões da plataforma:

SH

```
docker build -t erad:1.0 .
docker image list
```

Feito isso, a imagem deve estar criada e disponível para utilização.

3.3.2. Criando containers Singularity

Containers Docker acabam não sendo os mais recomendados para uso em processamento de alto desempenho (PAD). Outros orquestradores de container, como Singularity, são mais indicados para executar aplicações em supercomputação por se integrarem mais facilmente aos gerenciadores de trabalhos (GAMBLIN et al., 2015).

Podemos criar containers Singularity (representados normalmente por arquivos com a extensão `.sif`) tanto a partir de uma imagem Docker pré-existente, quanto diretamente de um ambiente existente no spack. Para ambas as opções, devemos nos certificar que o Singularity está um uma versão igual ou maior a `3.7.7`. Para converter uma imagem Docker em Singularity, podemos executar o comando:

SH

```
sudo singularity build erad.sif docker-daemon://erad:1.0
```

Já para criar uma imagem Singularity diretamente do Spack, precisamos alterar o arquivo `spack.yaml` que contém as configurações do ambiente a ser transformado em container. No marcador `format`, devemos especificar Singularity, assim:

YAML

```
spack:
  specs: [zlib]

  container:
    format: singularity
```

Em seguida, gerar o arquivo de configurações `.def` do singularity e enfim gerar a imagem `.sif` do container com os comandos:

SH

```
spack containerize > erad.def
sudo singularity build erad.sif erad.def
```

3.3.3. Configurações adicionais para containers

É possível também adicionar configurações extras ao arquivo `spack.yaml` do ambiente a ser transformado em container. Por exemplo, é possível especificar a imagem base a ser utilizada, rótulos e outras informações. No exemplo de código abaixo, a subárvore `container:` traz as especificações a serem consideradas na criação do container.

YAML

```yaml
spack:
  specs: [zlib]

  container:
    format: docker

    images:
      os: "ubuntu:18.04"
      spack: develop

    strip: true

    os_packages:
      final:
      - gcc
      - libgomp

    labels:
      mpi: "openmpi"
```

No exemplo do código acima, determinamos com o marcador `format` se o container será do tipo Docker ou Singularity. Já no marcador `image` podemos definir qual versão de sistema operacional desejamos usar como base, e qual versão do repositório spack também desejamos usar. Neste exemplo, definimos a versão 18.04 do Ubuntu, com o spack em sua versão de desenvolvimento. Também é possível usar imagens personalizadas como base, como veremos no exemplo a seguir. Por fim, no marcador `os_packages` podemos definir pacotes de sistema que desejamos que sejam instalados no sistema operacional da imagem para serem utilizados na execução do container. Neste exemplo, estamos adicionando a instalação da biblioteca `libgomp` (para execução de aplicações OpenMP). O marcador `labels` permite definir rótulos para a imagem final diretamente pela criação do Dockerfile. Com esse novo arquivo no formato `yaml`, basta reexecutar os comando de criação do Dockerfile e da imagem, como previamente demonstrado.

Para criar uma imagem a partir de uma imagem base externa, deve-se usar a tag `images:build` e `images:final`, com o identificador da imagem a ser utilizada, como no exemplo abaixo:

YAML

```yaml
spack:
  specs:
  - gromacs@2019.4+cuda build_type=Release
  - mpich
  - fftw precision=float
  packages:
    cuda:
      buildable: False
      externals:
      - spec: cuda%gcc
        prefix: /usr/local/cuda

  # Criando o container com imagens base externas.
  container:
    images:
      build: custom/cuda-10.1-ubuntu18.04:latest
      final: nvidia/cuda:10.1-base-ubuntu18.04
```

3.4. Criação de pacotes

Os pacotes Spack são escritos em Python. A criação de um novo pacote resume-se a criação de um arquivo chamado `package.py` contendo as informações básicas do pacote tais como descrição, versões, dependências e sistema de construção como Autotools ou CMake, por exemplo.

3.4.1. Passos iniciais

O passo inicial para criação de um novo pacote deve especificar a descrição, URL para *download* e depêndencias obrigatórias. A diretiva `create` através do comando `spack create` facilita a criação de um novo pacote preparando um diretório para o mesmo e um esqueleto do arquivo `package.py` a ser preenchido pelo usuário. Aqui usaremos como exemplo a criação de um pacote Spack para a ferramenta `pajeng`:

SH

```sh
spack create -n pajeng -t cmake https://github.com/schnorr/
   pajeng/archive/1.3.6.tar.gz
```

Em seguida, podemos preencher o `package.py` adicionando as informações básicas para que o `pajeng` possa ser construído e instalado. A opção `-t cmake` utilizada no comando acima permite criar um esqueleto preparado para *softwares* construídos com o CMake, como é o caso do `pajeng`.

As linhas 4 a 8 do código abaixo descrevem informações básicas como descrição e página do `pajeng`, tais informações serão exibidas através da diretiva `info` ao executar o comando `spack info pajeng`. A linha 10 aponta os mantenedores do pacote enquanto a linha 12 define uma versão que instalará o `pajeng` no *release* 1.3.6. Nesta

mesma linha, utilizamos o parâmetro `preferred` para designá-la como versão preferencial. Por fim, as linhas 18 a 20 declaram as dependências requeridas para compilação do `pajeng`. Estas dependências serão automaticamente instaladas pelo spack e seus respectivos locais de instalação serão fornecidos de maneira transparente ao CMake do `pajeng`. Também é possível instalar um pacote a partir do código fonte disponibilizado em um repositório git, svn ou hg conforme ilustrado na linha 15. Parâmetros adicionais como `branch`, `commit` e `tag` permitem especificar versões diferentes do código fonte no escopo do repositório informado.

PYTHON

```
1  from spack import *

3  class Pajeng(CMakePackage):
4    """PajeNG is a re-implementation of the well-known Paje
      visualization tool for the analysis of execution traces."""

6    homepage = "https://github.com/schnorr/pajeng"
7    git = "https://github.com/schnorr/pajeng.git"
8    url = "https://github.com/schnorr/pajeng/archive/1.3.6.tar.gz"

10   maintainers = ['viniciusvgp', 'schnorr']

12   version('1.3.6',
13           sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f9
       7e01c7839ffea8b9d0b3',
14           preferred = True)
15   version('develop',
16           git = 'https://github.com/schnorr/pajeng.git')

18   depends_on('boost')
19   depends_on('flex')
20   depends_on('bison')
```

O pacote recém criado pode ser instalado com:

SH

```
spack install pajeng
```

3.4.2. Alterações em pacotes existentes

Alterações em pacotes criados previamente podem ser feitas com a diretiva `edit` como no comando `spack edit pajeng`. A título de exemplo, adicionaremos as demais versões do `pajeng`. Uma maneira simples de se obter todas as versões disponíveis para um dado pacote é executar o comando `spack checksum` e adicionar sua saída ao conteúdo do `package.py`. Por meio da `url` base informada com `spack create` quando da criação do pacote, será feita uma busca pelos números de versões lançadas e das respectivas somas de verificação SHA-256.

SH

```
spack checksum -b pajeng
```

Como resultado do comando acima, obtemos as seguintes linhas a serem adicionadas ao arquivo do pacote:

PYTHON

```
version('1.3.6', sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba
    01172f97e01c7839ffea8b9d0b3')
version('1.3.5', sha256 = 'ea8ca02484de4091dcf57289724876ec17dd9
    8e3a032dc609b7ea020ca2629eb')
version('1.3.4', sha256 = '284e9a590a2861251e808542663bf1b77bc2c
    99650a1fbf945cd5bab65402f9e')
version('1.3.3', sha256 = '42cf44003d238fd5c4ab512bdeb445fc12f7e
    3bd3f0526b389f080c84b83b19f')
version('1.3.2', sha256 = '97154415a22f9b7f83516e988ea664b399037
    7d69fca859275ca48d7bfad0932')
version('1.3.1', sha256 = '4bc3764aaa7e79da9a81f40c0593b646007b6
    89e4ac20886d06f271ce0fa0a60')
version('1.3',   sha256 = '781b8be935e10b65470207f4f179bb1196aa6
    740547f9f1af0cb1c0193f11c6f')
version('1.1',   sha256 = '986d03e6deed20a3b9d0e076b1be9053c1bc8
    6c8b41ca36cce3ba3b22dc6abca')
version('1.0',   sha256 = '4d98d1a78669290d0a2e6bfe07a1eb4ab96bd
    05e5ef78da96d2c3cf03b023aa0')
```

3.4.3. Dependências e versões

Na seção 3.4.1, demostramos como adicionar as depêndencias mínimas para que se possa construir o `pajeng`. Entretanto, diferentes versões de um dado pacote podem ter dependências distintas e bastante específicas. Como exemplos, usaremos tanto versões mais antigas do `pajeng` que dependem das bibliotecas `qt` na versão 4.x e `glut` quanto a versão em desenvolvimento que depende da biblioteca `fmt`. Para tratar o primeiro caso, podemos especificar novas dependências que serão aplicadas apenas a versões do `pajeng` iguais ou anteriores a 1.3.2. Note que é possível selecionar versões específicas de uma dependência. Neste exemplo, é requerida uma versão do `qt` inferior a 4.999 e com a variante `opengl` habilitada. No segundo caso, adicionamos a dependência `fmt` que será aplicada apenas quando solicitada a versão `develop`. As linhas a serem adicionadas são:

PYTHON

```
depends_on('qt@:4.999+opengl', when='@:1.3.2')
depends_on('freeglut', when='@:1.3.2')
depends_on('fmt', when='@develop')
```

3.4.4. Variantes

Pacotes spack podem ter variantes que permitem que o usuário possa habilitar ou desabilitar funcionalidades. Para ilustrar este recurso, vejamos um exemplo de variantes para

habilitar a ligação estática e a documentação e para desabilitar a construção da biblioteca `libpaje` e das ferramentas auxiliares:

PYTHON

```
variant('static',
        default = False,
        description = "Build as static library")
variant('doc',
        default = False,
        description = "The Paje Trace File documentation")
variant('lib',
        default = True,
        description = "Build libpaje")
variant('tools',
        default = True,
        description = "Build auxiliary tools")

def cmake_args(self):
  args = [
    self.define_from_variant('STATIC_LINKING', 'static'),
    self.define_from_variant('PAJE_DOC', 'doc'),
    self.define_from_variant('PAJE_LIBRARY', 'lib'),
    self.define_from_variant('PAJE_TOOLS', 'tools')
  ]
  return args
```

Note que as variantes podem ser marcadas como habilitadas ou desabilitadas por padrão através do parâmetro `default`. Como as novas variantes implicam alterar a configuração padão de construção do `pajeng`, também se faz necessário sobreescrever o método `cmake_args` para levar em conta a ativação ou desativação das variáveis relacionadas às funcionalidades ativadas ou não pelo usuário no momento da instalação.

3.4.5. Conflitos

Em alguns casos pode ser interessante declarar conflitos entre as diversas opções a serem aplicadas quando da construção de um pacote. Estes conflitos podem refletir incompatibilidades com dependências ou compiladores, *bugs*, ou simplesmente configurações contraditórias do próprio pacote. No caso do `pajeng`, definimos anteriormente duas variantes `lib` e `tools`. Enquanto requisitar a instalação `pajeng+lib~tools` é perfeitamente válido, fazer o contrário, isto é `pajeng~lib+tools`, é inconsitente visto que as ferramentas auxiliares (`tools`) dependem da biblioteca (`lib`). Para lidar com estes casos, podemos definir conflitos de forma a impedir tal tentativa de instalação:

PYTHON

```
conflicts('+tools',
          when = '~lib',
          msg = "Enable libpaje to compile tools.")
```

Ao solicitar qualquer fórmula de instalação que case, ainda que implicitamente, com a regra especificada como conflito o usuário receberá a mensagem de erro descrita no parâmetro `msg`.

3.4.6. Publicação de pacotes

Existem duas maneiras de tornar público um novo pacote Spack. A primeira, e mais abrangente, é submetê-lo ao repositório oficial. Submissões de novos pacotes podem ser feitas por meio de *pull-requests*. Uma vez que o pacote passe nos testes automatizados, ele estará disponível para instalação por qualquer usuário. Uma versão completa do pacote `pajeng` foi submetida ao repositório oficial do Spack e pode ser encontrada em <https://github.com/spack/spack/blob/develop/var/spack/repos/builtin/packages/pajeng/package.py>.

A segunda maneira de publicar um novo pacote é por meio da criação de repositórios adicionais quem podem ser públicos ou privados. Repositórios adicionais podem ser criados com `spack repo create` e posteriormente compartilhados com demais usuários que poderão adicioná-los em suas instâncias locais executando `spack repo add`. Repositórios adicionais são úteis para *softwares* cujo acesso é restrito ou para pacotes Spack em desenvolvimento que ainda não estão suficientemente maduros para serem submetidos ao repositório oficial. Algumas instituições procuram manter repositórios Spack externos e públicos para divulgar e facilitar a instalação dos *softwares* por elas desenvolvido. Outro caso de uso para repositórios adicionais é quando, por algum motivo, é necessário sobrepor um pacote do repositório oficial.

3.5. Conclusão e Discussão

Este minicurso abordou os conceitos básicos fundamentais para se gerenciar pacotes de *software* em nível de usuário. Facilitando a reprodutibilidade dos experimentos, apresentamos a ferramenta Spack que tem sido bastante utilizando em parques computacionais de alto desempenho. Para saber mais sobre o Spack, referenciamos ao tutorial Spack (GAMBLIN et al., 2015), que porta uma enorme quantidade de diretivas auxiliares para tratar casos mais específicos não abordados neste minicurso.

Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) com a bolsa 141971/2020-7 para o terceiro autor, e dos projetos: FAPERGS ReDaS (19/711-6), MultiGPU (16/354-8) e GreenCloud (16/488-9), do projeto CNPq 447311/2014-0, do projeto CAPES/Brafitec 182/15 e CAPES/Cofecub 899/18, e com apoio do projeto Petrobras (2018/00263-5).

Referências

ALLES, G. R.; CARISSIMI, A.; SCHNORR, L. M. Assessing the computation and communication overhead of linux containers for hpc applications. In: IEEE. *2018 Symposium on High Performance Computing Systems (WSCAD)*. [S.l.], 2018. p. 116–123. páginas 8

COURTÈS, L.; WURMUS, R. Reproducible and user-controlled software environments in hpc with guix. In: HUNOLD, S. et al. (Ed.). *Euro-Par 2015: Parallel Processing Workshops*. Cham: Springer International Publishing, 2015. p. 579–591. ISBN 978-3-319-27308-2. páginas 2

DOLSTRA, E.; JONGE, M. de; VISSER, E. Nix: A safe and policy-free system for software deployment. In: *Proceedings of the 18th USENIX Conference on System Administration*. USA: USENIX Association, 2004. (LISA '04), p. 79–92. páginas 2

GAMBLIN, T. et al. The spack package manager: Bringing order to hpc software chaos. In: IEEE. *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. [S.l.], 2015. p. 1–12. páginas 2, 9, 15

HOWELL, M. *Homebrew, the missing package manager for OS X*. 2017. Disponível em: <http://brew.sh>. páginas 2

KURTZER, G. M.; SOCHAT, V.; BAUER, M. W. Singularity: Scientific containers for mobility of compute. *PloS one*, Public Library of Science San Francisco, CA USA, v. 12, n. 5, p. e0177459, 2017. páginas 8

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, Belltown Media, Houston, TX, v. 2014, n. 239, mar. 2014. ISSN 1075-3583. Disponível em: <http://dl.acm.org/citation.cfm?id=2600239.2600241>. páginas 8

NUSSBAUM, L. et al. Linux-based virtualization for hpc clusters. In: *Montreal Linux Symposium*. [S.l.: s.n.], 2009. páginas 2

SCHNORR, L. M. *PajeNG – Paje Next Generation*. 2021. Disponível em: <https://github.com/schnorr/pajeng>. páginas 3

TORREZ, A.; RANDLES, T.; PRIEDHORSKY, R. Hpc container runtimes have minimal or no performance impact. In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. [S.l.: s.n.], 2019. p. 37–42. páginas 8