Garlic: User Guide

Rodrigo Arias Mallo

Barcelona Supercomputing Center

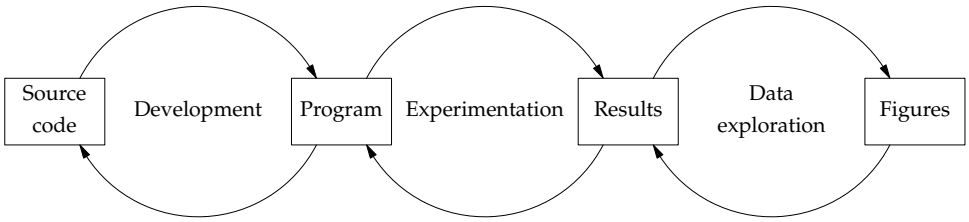
ABSTRACT

This document contains all the information to configure and use the garlic benchmark. All stages from the development to the publication are covered, as well as the introductory steps required to setup the machines.

Generated on 2023-09-14
Git commit: 3a4062ac04be6263c64a481420d8e768c2521b80

1. Introduction

The garlic framework is designed to fulfill all the requirements of an experimenter in all the steps up to publication. The experience gained while using it suggests that we move along three stages despicted in the following diagram:



In the development phase the experimenter changes the source code in order to introduce new features or fix bugs. Once the program is considered functional, the next phase is the experimentation, where several experiment configurations are tested to evaluate the program. It is common that some problems are spotted during this phase, which lead the experimenter to go back to the development phase and change the source code.

Finally, when the experiment is considered completed, the experimenter moves to the next phase, which envolves the exploration of the data generated by the experiment. During this phase, it is common to generate results in the form of plots or tables which provide a clear insight in those quantities of interest. It is also common that after looking at the figures, some changes in the experiment configuration need to be introduced (or even in the source code of the program).

Therefore, the experimenter may move forward and backwards along three phases several times. The garlic framework provides support for all the three stages (with different degrees of madurity).

1.1. Machines and clusters

Our current setup employs multiple machines to build and execute the experiments. Each cluster and node has it's own name and will be different in other clusters. Therefore, instead of using the names of the machines we use machine classes to generalize our setup. Those machine clases currently correspond to a physical

machine each:

- **Builder** (xeon07): runs the nix-daemon and performs the builds in /nix. Requires root access to setup the *nix-daemon* with multiple users.
- **Target** (MareNostrum 4 compute nodes): the nodes where the experiments are executed. It doesn't need to have /nix installed or root access.
- **Login** (MareNostrum 4 login nodes): used to allocate resources and run jobs. It doesn't need to have /nix installed or root access.
- **Laptop** (where the keyboard is attached, can be anything): used to connect to the other machines. No root access is required or /nix, but needs to be able to connect to the builder.

The machines don't need to be different of each others, as one machine can implement several classes. For example the laptop can act as the builder too but is not recommended. Or the login machine can also perform the builds, but is not possible yet in our setup.

1.2. Reproducibility

An effort to facilitate the reproducibility of the experiments has been done, with varying degrees of success. The names of the different levels of reproducibility have not been yet standarized, so we define our own to avoid any confusion. We define three levels of reproducibility based on the people and the machine involved:

- R0: The *same* people on the *same* machine obtain the same result
- R1: *Different* people on the *same* machine obtain the same result
- R2: *Different* people on a *different* machine obtain the same result

The garlic framework distinguishes two types of results: the result of *building a derivation* (usually building a binary or a library from the sources) and the results of the *execution of an experiment* (typically those are the measurements performed during the execution of the program of study).

For those two types, the meaning of *same result* is different. In the case of building a binary, we define the same result if it is bit-by-bit identical. In the packages provided by nixos is usually the case except some rare cases. One example is that during the build process, a directory is listed by the order of the inodes, giving a random order which is different between builds. These problems are tracked by the r13y ⟨`https://r13y.com/`⟩ project. About 99% of the derivations of the minimal package set achieve the R2 property.

On the other hand, the results of the experiments are always bit-by-bit different. So we change the definition to state that they are the same if the conclusions that can be obtained are the same. In particular, we assume that the results are within the confidence interval. With this definition, all experiments are currently R1. The reproducibility level R2 is not posible yet as the software is compiled to support only the target machine, with an specific interconnection.

2. Preliminary steps

The peculiarities of our setup require that users perform some actions to use the garlic framework. The content of this section is only intended for the users of our machines, but can serve as reference in other machines.

The names of the machine classes are used in the command line prompt instead of the actual name of the machine, to indicate that the command needs to be executed in the stated machine class, for example:

```
builder% echo hi
hi
```

When the machine class is not important, it is ignored and only the `%` prompt appears.

2.1. Configure your laptop

To easily connect to the builder (xeon07) in one step, configure the SSH client to perform a jump over the Cobi login node. The *ProxyJump* directive is only available in version 7.3 and upwards. Add the following lines in the `~/.ssh/config` file of your laptop:

```
Host cobi
      HostName ssflogin.bsc.es
      User your-username-here

Host xeon07
      ProxyJump cobi
      HostName xeon07
      User your-username-here
```

You should be able to connect to the builder typing:

```
laptop$ ssh xeon07
```

To spot any problems try with the `-v` option to enable verbose output.

2.2. Configure the builder (xeon07)

In order to use nix you would need to be able to download the sources from Internet. Usually the download requires the ports 22, 80 and 443 to be open for outgoing traffic.

Check that you have network access in xeon07 provided by the environment variables *http_proxy* and *https_proxy*. Try to fetch a webpage with curl, to ensure the proxy is working:

```
xeon07$ curl x.com
x
```

2.2.1. Create a new SSH key

There is one DSA key in your current home called "cluster" that is no longer supported in recent SSH versions and should not be used. Before removing it, create a new one without password protection leaving the passphrase empty (in case that you don't have one already created) by running:

```
xeon07$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.
Your public key has been saved in ~/.ssh/id_rsa.pub.
...
```

By default it will create the public key at `~/.ssh/id_rsa.pub`. Then add the newly created key to the authorized keys, so you can connect to other nodes of the Cobi cluster:

```
xeon07$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Finally, delete the old "cluster" key:

```
xeon07$ rm ~/.ssh/cluster ~/.ssh/cluster.pub
```

And remove the section in the configuration `~/.ssh/config` where the key was assigned to be used in all hosts along with the `StrictHostKeyChecking=no` option. Remove the following lines (if they exist):

```
Host *
    IdentityFile ~/.ssh/cluster
    StrictHostKeyChecking=no
```

By default, the SSH client already searchs for a keypair called `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`, so there is no need to manually specify them.

You should be able to access the login node with your new key by using:

```
xeon07$ ssh ssfhead
```

2.2.2. Authorize access to the repository

The sources of BSC packages are usually downloaded directly from the PM git server, so you must be able to access all repositories without a password prompt.

Most repositories are open to read for logged in users, but there are some exceptions (for example the nanos6 repository) where you must have explicitly granted read access.

Copy the contents of your public SSH key in `~/.ssh/id_rsa.pub` and paste it in GitLab at

```
https://pm.bsc.es/gitlab/profile/keys
```

Finally verify the SSH connection to the server works and you get a greeting from the GitLab server with your username:

```
xeon07$ ssh git@bscpm03.bsc.es
PTY allocation request failed on channel 0
Welcome to GitLab, @rarias!
Connection to bscpm03.bsc.es closed.
```

Verify that you can access the nanos6 repository (otherwise you first need to ask to be granted read access), at:

```
https://pm.bsc.es/gitlab/nanos6/nanos6
```

Finally, you should be able to download the nanos6 git repository without any password interaction by running:

```
xeon07$ git clone git@bscpm03.bsc.es:nanos6/nanos6.git
```

Which will create the nanos6 directory.

2.2.3. Authorize access to MareNostrum 4

You will also need to access MareNostrum 4 from the xeon07 machine, in order to run experiments. Add the following lines to the `~/.ssh/config` file and set your user name:

```
Host mn0 mn1 mn2
    User <your user name in MN4>
```

Then copy your SSH key to MareNostrum 4 (it will ask you for your login password):

```
xeon07$ ssh-copy-id -i ~/.ssh/id_rsa.pub mn1
```

Finally, ensure that you can connect without a password:

```
xeon07$ ssh mn1
...
login1$
```

2.2.4. Clone the bscpkgs repository

Once you have Internet and you have granted access to the PM GitLab repositories you can begin building software with nix. First ensure that the nix binaries are available from your shell in xeon07:

```
xeon07$ nix --version
nix (Nix) 2.3.6
```

Now you are ready to build and install packages with nix. Clone the bscpkgs repository:

```
xeon07$ git clone git@bscpm03.bsc.es:rarias/bscpkgs.git
```

Nix looks in the current folder for a file named `default.nix` for packages, so go to the bscpkgs directory:

```
xeon07$ cd bscpkgs
```

Now you should be able to build nanos6 (which is probably already compiled):

```
xeon07$ nix-build -A bsc.nanos6
...
/nix/store/...2cm1ldx9smb552sf6r1-nanos6-2.4-6f10a32
```

The installation is placed in the nix store (with the path stated in the last line of the build process), with the `result` symbolic link pointing to the same location:

```
xeon07$ readlink result
/nix/store/...2cm1ldx9smb552sf6r1-nanos6-2.4-6f10a32
```

2.2.5. Configure garlic

In order to launch experiments in the *target* machine, it is required to configure nix to allow a directory to be available during the build process, where the results will be stored before being copied in the nix store. Create a new `garlic` directory in your personal cache directory and copy the full path:

```
xeon07$ mkdir -p ~/.cache/garlic
xeon07$ readlink -f ~/.cache/garlic
/home/Computational/rarias/.cache/garlic
```

Then create the nix configuration directory (if it has not already been created):

```
xeon07$ mkdir -p ~/.config/nix
```

And add the following line in the `~/.config/nix/nix.conf` file, replacing it with the path you copied before:

```
extra-sandbox-paths = /garlic=/home/Computational/rarias/.cache/garlic
```

This option creates a virtual directory called `/garlic` inside the build environment, whose contents are the ones you specify at the right hand side of the equal sign (in this case the `~/.cache/garlic` directory). It will be used to allow the results of the experiments to be passed to nix from the *target* machine.

2.2.6. Run the garlic daemon (optional)

The garlic benchmark has a daemon which can be used to automatically launch the experiments in the *target* machine on demand, when they are required to build other derivations, so they can be launched without user interaction. The daemon creates some FIFO pipes to communicate with the build environment, and must be running to be able to run the experiments. To execute it, go to the `bscpkgs/garlic` directory and run

```
xeon07$ nix-shell
nix-shell$
```

to enter the nix shell (or specify the path to the `garlic/shell.nix` file as argument). Then, run the daemon inside the nix shell:

```
nix-shell$ garlicd
garlicd: Waiting for experiments ...
```

Notice that the daemon stays running in the foreground, waiting for experiments. At this moment, it can only process one experiment at a time.

2.3. Configure the login and target (MareNostrum 4)

In order to execute the programs in MareNostrum 4, you first need load some utilities in the PATH. Add to the end of the file `~/.bashrc` in MareNostrum 4 the following line:

```
export PATH=/gpfs/projects/bsc15/nix/bin:$PATH
```

Then logout and login again (our source the `~/.bashrc` file) and check that now you have the `nix-develop` command available:

```
login1$ which nix-develop
/gpfs/projects/bsc15/nix/bin/nix-develop
```

The new utilities are available both in the login nodes and in the compute (target) nodes, as they share the file system over the network.

3. Development

During the development phase, a functional program is produced by modifying its source code. This process is generally cyclic: the developer needs to compile, debug and correct mistakes. We want to minimize the delay times, so the programs can be executed as soon as needed, but under a controlled environment so that the same behavior occurs during the experimentation phase.

In particular, we want that several developers can reproduce the same development environment so they can debug each other programs when reporting bugs. Therefore, the environment must be carefully controlled to avoid non-reproducible scenarios.

The current development environment provides an isolated shell with a clean environment, which runs in a new mount namespace where access to the filesystem is restricted. Only the project directory and the nix store are available (with some other exceptions), to ensure that you cannot accidentally link with the wrong library or modify the build process with a forgotten environment variable in the `~/.bashrc` file.

3.1. Getting the development tools

To create a development environment, first copy or download the sources of your program (not the dependencies) in a new directory placed in the target machine (MareNostrum 4).

The default environment contains packages commonly used to develop programs, listed in the *garlic/index.nix* file:

```
develop = let
  commonPackages = with self; [
    coreutils htop procps-ng vim which strace
    tmux gdb kakoune universal-ctags bashInteractive
    glibcLocales ncurses git screen curl
    # Add more nixpkgs packages here...
  ];
  bscPackages = with bsc; [
    slurm clangOmpss2 icc mcxx perf tampi impi
    # Add more bsc packages here...
  ];
  ...
```

If you need additional packages, add them to the list, so that they become available in the environment. Those may include any dependency required to build your program.

Then use the build machine (xeon07) to build the *garlic.develop* derivation:

```
build% nix-build -A garlic.develop
...
build% grep ln result
ln -fs /gpfs/projects/.../bin/stage1 .nix-develop
```

Copy the *ln* command and run it in the target machine (MareNostrum 4), inside the new directory used for your program development, to create the link *.nix-develop* (which is used to remember your environment). Several environments can be stored in different directories using this method, with different packages in each environment. You will need to rebuild the *garlic.develop* derivation and update the *.nix-develop* link after the package list is changed. Once the environment link is created, there is no need to repeat these steps again.

Before entering the environment, you will need to access the required resources for your program, which may include several compute nodes.

3.2. Allocating resources for development

Our target machine (MareNostrum 4) provides an interactive shell, that can be requested with the number of computational resources required for development. To do so, connect to the login node and allocate an interactive session:

```
% ssh mn1
login% salloc ...
target%
```

This operation may take some minutes to complete depending on the load of the cluster. But once the session is ready, any subsequent execution of programs will be immediate.

3.3. Accessing the developement environment

The utility program *nix-develop* has been designed to access the development environment of the current directory, by looking for the *.nix-develop* file. It creates a namespace where the required packages are installed and ready to be used. Now you can access the newly created environment by running:

```
target% nix-develop
develop%
```

The spawned shell contains all the packages pre-defined in the *garlic.develop* derivation, and can now be accessed by typing the name of the commands.

```
develop% which gcc
/nix/store/azayfhqyg9...s8aqfmy-gcc-wrapper-9.3.0/bin/gcc
develop% which gdb
/nix/store/1c833b2y8j...pnjn2nv9d46zv44dk-gdb-9.2/bin/gdb
```

If you need additional packages, you can add them in the *garlic/index.nix* file as mentioned previously. To keep the same current resources, so you don't need to wait again for the resources to be allocated, exit only from the development shell:

```
develop% exit
target%
```

Then update the *.nix-develop* link and enter into the new develop environment:

```
target% nix-develop
develop%
```

3.4. Execution

The allocated shell can only execute tasks in the current node, which may be enough for some tests. To do so, you can directly run your program as:

```
develop$ ./program
```

If you need to run a multi-node program, typically using MPI communications, then you can do so by using srun. Notice that you need to allocate several nodes when calling salloc previously. The srun command will execute the given program **outside** the development environment if executed as-is. So we re-enter the develop environment by calling nix-develop as a wrapper of the program:

```
develop$ srun nix-develop ./program
```

3.5. Debugging

The debugger can be used to directly execute the program if is executed in only one node by using:

```
develop$ gdb ./program
```

Or it can be attached to an already running program by using its PID. You will need to first connect to the node running it (say target2), and run gdb inside the nix-develop environment. Use *squeue* to see the compute nodes running your program:

```
login$ ssh target2
target2$ cd project-develop
target2$ nix-develop
develop$ gdb -p $pid
```

You can repeat this step to control the execution of programs running in different nodes simultaneously.

In those cases where the program crashes before being able to attach the debugger, enable the generation of core dumps:

```
develop$ ulimit -c unlimited
```

And rerun the program, which will generate a core file that can be opened by gdb and contains the state of the memory when the crash happened. Beware that the core dump file can be very large, depending on the memory used by your program at the crash.

3.6. Git branch name convention

The garlic benchmark imposes a set of requirements to be meet for each application in order to coordinate the execution of the benchmark and the gathering process of the results.

Each application must be available in a git repository so it can be included into the garlic benchmark. The different combinations of programming models and communication schemes should be each placed in one git branch, which are referred to as *benchmark branches*. At least one benchmark branch should exist and they all must begin with the prefix `garlic/` (other branches will be ignored).

The branch name is formed by adding keywords separated by the "+" character. The keywords must follow the given order and can only appear zero or once each. At least one keyword must be included. The following keywords are available:

`mpi` A significant fraction of the communications uses only the standard MPI (without extensions like TAMPI).

`tampi` A significant fraction of the communications uses TAMPI.

`send` A significant part of the MPI communication uses the blocking family of methods (*MPI_Send*, *MPI_Recv*, *MPI_Gather*...).

`isend` A significant part of the MPI communication uses the non-blocking family of methods (*MPI_Isend*, *MPI_Irecv*, *MPI_Igather*...).

`rma` A significant part of the MPI communication uses remote memory access (one-sided) methods (*MPI_Get*, *MPI_Put*...).

`seq` The complete execution is sequential in each process (one thread per process).

`omp` A significant fraction of the execution uses the OpenMP programming model.

`oss` A significant fraction of the execution uses the OmpSs-2 programming model.

`task` A significant part of the execution involves the use of the tasking model.

`taskfor` A significant part of the execution uses the taskfor construct.

`fork` A significant part of the execution uses the fork-join model (including hybrid programming techniques with parallel computations and sequential communications).

`simd` A significant part of the computation has been optimized to use SIMD instructions.

In the Appendix A ⟨`see below`⟩ there is a flowchart to help the decision process of the branch name. Additional user defined keywords may be added at the end using the separator "+" as well. User keywords must consist of capital alphanumeric characters only and be kept short. These additional keywords must be different (case insensitive) to the already defined above. Some examples:

```
garlic/mpi+send+seq
garlic/mpi+send+omp+fork
garlic/mpi+isend+oss+task
garlic/tampi+isend+oss+task
garlic/tampi+isend+oss+task+COLOR
garlic/tampi+isend+oss+task+COLOR+BTREE
```

3.7. Initialization time

It is common for programs to have an initialization phase prior to the execution of the main computation task which is the objective of the study. The initialization phase is usually not considered when taking measurements, but the time it takes to complete can limit seriously the amount of information that can be extracted from the computation phase. As an example, if the computation phase is in the order of seconds, but the initialization phase takes several minutes, the number of runs would need to be set low, as the units could exceed the time limits. Also, the experimenter may be reluctant to modify the experiments to test other parameters, as the waiting time for the results is unavoidably large.

To prevent this problem the programs must reduce the time of the initialization phase to be no larger than the computation time. To do so, the initialization phase can be optimized either with parallelization, or it can be modified to store the result of the initialization to the disk to be later at the computation phase. In the garlic framework an experiment can have a dependency over the results of another experiment (the results of the initialization). The initialization results will be cached if the derivation is kept invariant, when modifying the computation phase parameters.

3.8. Measurement of the execution time

The programs must measure the wall time of the computation phase following a set of rules. The way in which the wall time is measured is very important to get accurate results. The measured time must be implemented by using a monotonic clock which is able to correct the drift of the oscillator of the internal clock due to changes in temperature. This clock must be measured in C and C++ with:

```
clock_gettime(CLOCK_MONOTONIC, &ts);
```

A helper function can be used the approximate value of the clock in a double precision float, in seconds:

```
double get_time()
{
    struct timespec tv;
    if(clock_gettime(CLOCK_MONOTONIC, &tv) != 0)
    {
        perror("clock_gettime failed");
        exit(EXIT_FAILURE);
    }
    return (double)(ts.tv_sec) +
        (double)ts.tv_nsec * 1.0e-9;
}
```

The start and end points must be measured after the synchronization of all the

processes and threads, so the complete computation work can be bounded to fit inside the measured interval. An example for a MPI program:

```
double start, end, delta_time;
MPI_Barrier();
start = get_time();
run_simulation();
MPI_Barrier();
end = get_time();
delta_time = end - start;
```

3.9. Format of the execution time

The measured execution time must be printed to the standard output (stdout) in scientific notation with at least 7 significative digits. The following the printf format (or the strict equivalent in other languages) must be used:

```
printf("time %e\n", delta_time);
```

The line must be printed alone and only once: for MPI programs, only one process shall print the time:

```
if(rank == 0) printf("time %e\n", delta_time);
```

Other lines can be printed in the stdout, but without the *time* prefix, so that the following pipe can be used to capture the line:

```
% ./app | grep "^time"
1.234567e-01
```

Ensure that your program follows this convention by testing it with the above *grep* filter; otherwise the results will fail to be parsed when building the dataset with the execution time.

4. Experimentation

During the experimentation, a program is studied by running it and measuring some properties. The experimenter is in charge of the experiment design, which is typically controlled by a single *nix* file placed in the `garlic/exp` subdirectory. Experiments are formed by several *experimental units* or simply *units*. A unit is the result of each unique configuration of the experiment (typically involves the cartesian product of all factors) and consists of several shell scripts executed sequentially to setup the *execution environment*, which finally launch the actual program being analyzed. The scripts that prepare the environment and the program itself are called the *stages* of the execution and altogether form the *execution pipeline* or simply the *pipeline*. The experimenter must know with very good details all the stages involved in the pipeline, as they have a large impact on the execution.

Additionally, the execution time is impacted by the target machine in which the experiments run. The software used for the benchmark is carefully configured and tuned for the hardware used in the execution; in particular, the experiments are designed to run in MareNostrum 4 cluster with the SLURM workload manager and the Omni-Path interconnection network. In the future we plan to add support for other clusters in order to execute the experiments in other machines.

4.1. Isolation

The benchmark is designed so that both the compilation of every software package and the execution of the experiment is performed under strict conditions. We can ensure that two executions of the same experiment are actually running the same program in the same software environment.

All the software used by an experiment is included in the *nix store* which is, by convention, located at the `/nix` directory. Unfortunately, it is common for libraries to try to load software from other paths like `/usr` or `/lib`. It is also common that configuration files are loaded from `/etc` and from the home directory of the user that runs the experiment. Additionally, some environment variables are recognized by the libraries used in the experiment, which change their behavior. As we cannot control the software and configuration files in those directories, we couldn't guarantee that the execution behaves as intended.

In order to avoid this problem, we create a *sandbox* where only the files in the nix store are available (with some other exceptions). Therefore, even if the libraries try to access any path outside the nix store, they will find that the files are not there anymore. Additionally, the environment variables are cleared before entering the environment (with some exceptions as well).

4.2. Execution pipeline

Several predefined stages form the *standard* execution pipeline and are defined in the *stdPipeline* array. The standard pipeline prepares the resources and the environment to run a program (usually in parallel) in the compute nodes. It is divided in two main parts: connecting to the target machine to submit a job and executing the job. Finally, the complete execution pipeline ends by running the actual program, which is not part of the standard pipeline, as should be defined differently for each program.

4.2.1. Job submission

Some stages are involved in the job submission: the *trebuchet* stage connects via *ssh* to the target machine and executes the next stage there. Once in the target machine, the *runexp* stage computes the output path to store the experiment results, using the user in the target machine and changes the working directory there. In MareNostrum 4 the output path is at `/gpfs/projects/bsc15/garlic/$user/out`. Then the*isolate* stage is executed to enter the sandbox and the *experiment* stage begins, which creates a directory to store the experiment output, and launches several *unit* stages.

Each unit executes a *sbatch* stage which runs the *sbatch(1)* program with a job script that simply calls the next stage. The sbatch program internally reads the `/etc/slurm/slurm.conf` file from outside the sandbox, so we must explicitly allow this file to be available, as well as the *munge* socket used for authentication by the SLURM daemon. Once the jobs are submitted to SLURM, the experiment stage ends and the trebuchet finishes the execution. The jobs will be queued for execution without any other intervention from the user.

The rationale behind running sbatch from the sandbox is because the options provided in environment variables override the options from the job script. Therefore, we avoid this problem by running sbatch from the sandbox, where the interfering environment variables are removed. The sbatch program is also provided in the *nix store*, with a version compatible with the SLURM daemon running in the target machine.

4.2.2. Job execution

Once an unit job has been selected for execution, SLURM allocates the resources (usually several nodes) and then selects one of the nodes to run the job script: it is not executed in parallel yet. The job script runs from a child process forked from on of the SLURM daemon processes, which are outside the sandbox. Therefore, we first run the *isolate* stage to enter the sandbox again.

The next stage is called *control* and determines if enough data has been generated by the experiment unit or if it should continue repeating the execution. At the current time, it is only implemented as a simple loop that runs the next stage a fixed amount of times (by default, it is repeated 30 times).

The following stage is *srun* which launches several copies of the next stage to run in parallel (when using more than one task). Runs one copy per task, effectively creating one process per task. The CPUs affinity is configured by the parameter *--cpu-bind* and is important to set it correctly (see more details in the *srun(1)* manual). Appending the *verbose* value to the cpu bind option causes srun to print the assigned affinity of each task, which is very valuable when examining the execution log.

The mechanism by which srun executes multiple processes is the same used by sbatch, it forks from a SLURM daemon running in the computing nodes. Therefore, the execution begins outside the sandbox. The next stage is *isolate* which enters again the sandbox in every task. All remaining stages are running now in parallel.

4.2.3. The program

At this point in the execution, the standard pipeline has been completely executed, and we are ready to run the actual program that is the matter of the experiment. Usually, programs require some arguments to be passed in the command line. The *exec* stage sets the arguments (and optionally some environment variables) and executes the last stage, the *program*.

The experimenters are required to define these last stages, as they define the specific way in which the program must be executed. Additional stages may be included before or after the program run, so they can perform additional steps.

4.2.4. Stage overview

The complete execution pipeline using the standard pipeline is shown in the Table 1. Some properties are also reflected about the execution stages.

Stage	Where	Safe	Copies	User	Std
trebuchet	*	no	no	yes	yes
runexp	login	no	no	no	yes
isolate	login	no	no	no	yes
experiment	login	yes	no	no	yes
unit	login	yes	no	no	yes
sbatch	login	yes	no	no	yes
isolate	target	no	no	no	yes
control	target	yes	no	no	yes
srun	target	yes	no	no	yes
isolate	target	no	yes	no	yes
exec	target	yes	yes	no	no
program	target	yes	yes	no	no

Table 1: The stages of a complete execution pipeline. The *where* column determines where the stage is running, *safe* states if the stage begins the execution inside the sandbox, *user* if it can be executed directly by the user, *copies* if there are several instances running in parallel and *std* if is part of the standard execution pipeline.

4.3. Writing the experiment

The experiments are generally written in the *nix* language as it provides very easy management for the packages an their customization. An experiment file is formed by several parts, which produce the execution pipeline when built. The experiment file describes a function (which is typical in nix) and takes as argument an attribute set with some common packages, tools and options:

```
{ stdenv, lib, bsc, stdexp, targetMachine, stages, garlicTools }:
```

The *bsc* attribute contains all the BSC and nixpkgs packages, as defined in the overlay. The *stdexp* contains some useful tools and functions to build the experiments, like the standard execution pipeline, so you don't need to redefine the stages in every experiment. The configuration of the target machine is specified in the *targetMachine* attribute which includes information like the number of CPUs per node or the cache line length. It is used to define the experiments in such a way that they are not tailored to an specific machine hardware (sometimes this is not posible). All the execution stages are available in the *stages* attribute which are used when some extra stage is required. And finally, the *garlicTools* attribute provide some functions to aid common tasks when defining the experiment configuration

4.3.1. Experiment configuration

The next step is to define some variables in a `let ... in ... ;` construct, to be used later. The first one, is the variable configuration of the experiment called *varConf*, which include all the factors that will be changed. All the attributes of this set *must* be arrays, even if they only contain one element:

```
varConf = {
  blocks = [ 1 2 4 ];
  nodes = [ 1 ];
};
```

In this example, the variable *blocks* will be set to the values 1, 2 and 4; while *nodes* will remain set to 1 always. These variables are used later to build the experiment configuration. The *varConf* is later converted to a list of attribute sets, where every attribute contains only one value, covering all the combinations (the Cartesian product is computed):

```
[ { blocks = 1; nodes = 1; }
  { blocks = 2; nodes = 1; }
  { blocks = 4; nodes = 1; } ]
```

These configurations are then passed to the *genConf* function one at a time, which is the central part of the description of the experiment:

```
genConf = var: fix (self: targetMachine.config // {
  expName = "example";
  unitName = self.expName + "-b" + toString self.blocks;
  blocks = var.blocks;
  cpusPerTask = 1;
  tasksPerNode = self.hw.socketsPerNode;
  nodes = var.nodes;
});
```

It takes as input *one* configuration from the Cartesian product, for example:

```
{ blocks = 2; nodes = 1; }
```

And returns the complete configuration for that input, which usually expand the input configuration with some derived variables along with other constant parameters. The return value can be inspected by calling the function in the interactive *nix repl* session:

```
nix-repl> genConf { blocks = 2; nodes = 1; }
{
  blocks = 2;
  cpusPerTask = 1;
  expName = "example";
  hw = { ... };
  march = "skylake-avx512";
  mtune = "skylake-avx512";
  name = "mn4";
  nixPrefix = "/gpfs/projects/bsc15/nix";
  nodes = 1;
  sshHost = "mn1";
  tasksPerNode = 2;
  unitName = "example-b2";
}
```

Some configuration parameters were added by *targetMachine.config*, such as the *nixPrefix*, *sshHost* or the *hw* attribute set, which are specific for the cluster they experiment is going to run. Also, the *unitName* got assigned the proper name based on the number of blocks, but the number of tasks per node were assigned based on the hardware description of the target machine.

By following this rule, the experiments can easily be ported to machines with other hardware characteristics, and we only need to define the hardware details once. Then all the experiments will be updated based on those details.

4.3.2. Adding the stages

Once the configuration is ready, it will be passed to each stage of the execution pipeline which will take the parameters it needs. The connection between the parameters and how they are passed to each stage is done either by convention or manually. There is a list of parameters that are recognized by the standard pipeline stages. For example the attribute *nodes*, it is recognized as the number of nodes in the standard *sbatch* stage when allocating resources:

Stage	Attribute	Std	Req	Description
*	nixPrefix	yes	yes	Path to the nix store in the target
unit	expName	yes	yes	Name of the experiment
unit	unitName	yes	yes	Name of the unit
control	loops	yes	yes	Number of runs of each unit
sbatch	cpusPerTask	yes	yes	Number of CPUs per task (process)
sbatch	jobName	yes	yes	Name of the job
sbatch	nodes	yes	yes	Number of nodes allocated
sbatch	ntasksPerNode	yes	yes	Number of tasks (processes) per node
sbatch	qos	yes	no	Name of the QoS queue
sbatch	reservation	yes	no	Name of the reservation
sbatch	time	yes	no	Maximum allocated time (string)
exec	argv	no	no	Array of arguments to execve
exec	env	no	no	Environment variable settings
exec	pre	no	no	Code before the execution
exec	post	no	no	Code after the execution

Table 2: The attributes recognized by the stages in the execution pipeline. The column *std* indicates if they are part of the standard execution pipeline. Some attributes are required as indicated by the *req* column.

Other attribute names can be used to specify custom information used in additional stages. The two most common stages required to complete the pipeline are the *exec* and the *program*. Let see an example of*exec*:

```
exec = {nextStage, conf, ...}: stages.exec {
  inherit nextStage;
  argv = [ "--blocks" conf.blocks ];
};
```

The *exec* stage is defined as a function that uses the predefined *stages.exec* stage, which accepts the *argv* array, and sets the argv of the program. In our case, we fill the *argv* array by setting the *--blocks* parameter to the number of blocks, specified in the configuration in the attribute *blocks*. The name of this attribute can be freely choosen, as long as the *exec* stage refers to it properly. The *nextStage* attribute is mandatory in all stages, and is automatically set when building the pipeline.

The last step is to configure the actual program to be executed, which can be specified as another stage:

```
program = {nextStage, conf, ...}: bsc.apps.example;
```

Notice that this function only returns the *bsc.apps.example* derivation, which will be translated to the path where the example program is installed. If the program is located inside a directory (typically *bin*), it must define the attribute *programPath* in the *bsc.apps.example* derivation, which points to the executable program. An example:

```
stdenv.mkDerivation {
  ...
  programPath = "/bin/example";
  ...
};
```

4.3.3. Building the pipeline

With the *exec* and *program* stages defined and the ones provided by the standard pipeline, the complete execution pipeline can be formed. To do so, the stages are placed in an array, in the order they will be executed:

```
pipeline = stdexp.stdPipeline ++ [ exec program ];
```

The attribute *stdexp.stdPipeline* contains the standard pipeline stages, and we only append our two defined stages *exec* and *program*. The*pipeline* is an array of functions, and must be transformed in something that can be executed in the target machine. For that purpose, the *stdexp* provides the *genExperiment* function, which takes the *pipeline* array and the list of configurations and builds the execution pipeline:

```
stdexp.genExperiment { inherit configs pipeline; }
```

The complete example experiment can be shown here:

```
{ stdenv, lib, stdexp, bsc, targetMachine, stages }:
with lib;
let
  # Initial variable configuration
  varConf = {
    blocks = [ 1 2 4 ];
    nodes = [ 1 ];
  };
  # Generate the complete configuration for each unit
  genConf = c: targetMachine.config // rec {
    expName = "example";
    unitName = "${expName}-b${toString blocks}";
    inherit (targetMachine.config) hw;
    inherit (c) blocks nodes;
    loops = 30;
    ntasksPerNode = hw.socketPerNode;
    cpusPerTask = hw.cpusPerSocket;
    jobName = unitName;
  };
  # Compute the array of configurations
  configs = stdexp.buildConfigs {
    inherit varConf genConf;
  };
  exec = {nextStage, conf, ...}: stages.exec {
    inherit nextStage;
    argv = [ "--blocks" conf.blocks ];
  };
  program = {nextStage, conf, ...}: bsc.garlic.apps.example;
  pipeline = stdexp.stdPipeline ++ [ exec program ];
in
  stdexp.genExperiment { inherit configs pipeline; }
```

4.3.4. Adding the experiment to the index

The experiment file must be located in a named directory inside the *garlic/exp* directory. The name is usually the program name. Once the experiment is placed in a nix file, it must be added to the index of experiments, so it can be build. The index is hyerarchically organized as attribute sets, with *exp* containing all the experiments; *exp.example* the experiments of the *example* program; and *exp.example.test1* referring to the *test1* experiment of the *example* program. Additional attributes can be added, like *exp.example.test1.variantA* to handle more details.

For this example we are going to use the attribute path *exp.example.test* and add it to the index, in the *garlic/exp/index.nix* file. We append to the end of the attribute set, the following definition:

```
...
  example = {
    test = callPackage ./example/test.nix { };
  };
}
```

The experiment can now be built with:

```
builder% nix-build -A exp.example.test
```

4.4. Recommendations

The complete results generally take a long time to be finished, so it is advisable to design the experiments iteratively, in order to quickly obtain some feedback. Some recommendations: Start with one unit only. Set the number of runs low (say 5) but more than one. Use a small problem size, so the execution time is low. Set the time limit low, so deadlocks are caught early.

As soon as the first runs are complete, examine the results and test that everything looks good. You would likely want to check: The resources where assigned as intended (nodes and CPU affinity). No errors or warnings: look at stderr and stdout logs. If a deadlock happens, it will run out of the time limit.

As you gain confidence over that the execution went as planned, begin increasing the problem size, the number of runs, the time limit and lastly the number of units. The rationale is that each unit that is shared among experiments gets assigned the same hash. Therefore, you can iteratively add more units to an experiment, and if they are already executed (and the results were generated) is reused.

5. Post-processing

After the correct execution of an experiment the results are stored for further investigation. Typically the time of the execution or other quantities are measured and presented later in a figure (generally a plot or a table). The *postprocess pipeline* consists of all the steps required to create a set of figures from the results. Similarly to the execution pipeline where several stages run sequentially,[1] the postprocess pipeline is also formed by multiple stages executed in order.

The rationale behind dividing execution and postprocess is that usually the experiments are costly to run (they take a long time to complete) while generating a figure require less time. Refining the figures multiple times reusing the same experimental results doesn't require the execution of the complete experiment, so the experimenter can try multiple ways to present the data without waiting a large delay.

5.1. Results

The results are generated in the same *target* machine where the experiment is executed and are stored in the garlic `out` directory, organized into a tree structure following the experiment name, the unit name and the run number (governed by the *control* stage):

```
|-- 6lp88vlj7m8hvvhpfz25p5mvvg7ycflb-experiment
|   |-- 8lpmmfix52a8v7kfzkzih655awchl9f1-unit
|   |   |-- 1
|   |   |   |-- stderr.log
|   |   |   |-- stdout.log
|   |   |   |-- ...
|   |   |-- 2
...
```

In order to provide an easier access to the results, an index is also created by taking the *expName* and *unitName* attributes (defined in the experiment configuration) and linking them to the appropriate experiment and unit directories. These links are overwritten by the last experiment with the same names so they are only valid for the last execution. The out and index directories are placed into a per-user directory, as we cannot guarantee the complete execution of each unit when multiple users share units.

The messages printed to *stdout* and *stderr* are stored in the log files with the same name inside each run directory. Additional data is sometimes generated by the experiments, and is found in each run directory. As the generated data can be very large, is ignored by default when fetching the results.

5.2. Fetching the results

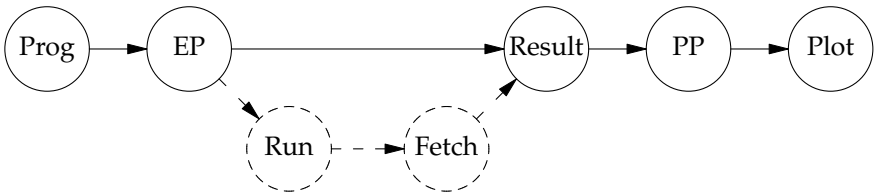
Consider a program of interest for which an experiment has been designed to measure some properties that the experimenter wants to present in a visual plot. When the experiment is launched, the execution pipeline (EP) is completely executed and it will generate some results. In this escenario, the execution pipeline depends on the program—any changes in the program will cause nix to build the pipeline again using the updated program. The results will also depend on the execution pipeline as well as the postprocess pipeline (PP) and the plot on the results. This chain of dependencies can be shown in the following dependency graph:

Ideally, the dependencies should be handled by nix, so it can detect any change

[1] Rodrigo Arias Mallo, *Garlic: the execution pipeline* (2020).

and rebuild the necessary parts automatically. Unfortunately, nix is not able to build the result as a derivation directly, as it requires access to the *target* machine with several user accounts. In order to let several users reuse the same results from a shared cache, we would like to use the *nix store*.

To generate the results from the experiment, we add some extra steps that must be executed manually:



The run and fetch steps are provided by the helper tool *garlic(1)*, which launches the experiment using the user credentials at the *target* machine and then fetches the results, placing them in a directory known by nix. When the result derivation needs to be built, nix will look in this directory for the results of the execution. If the directory is not found, a message is printed to suggest the user to launch the experiment and the build process is stopped. When the result is successfully built by any user, is stored in the *nix store* and it won't need to be rebuilt again until the experiment changes, as the hash only depends on the experiment and not on the contents of the results.

Notice that this mechanism violates the deterministic nature of the nix store, as from a given input (the experiment) we can generate different outputs (each result from different executions). We knowingly relaxed this restriction by providing a guarantee that the results are equivalent and there is no need to execute an experiment more than once.

To force the execution of an experiment you can use the *rev* attribute which is a number assigned to each experiment and can be incremented to create copies that only differs on that number. The experiment hash will change but the experiment will be the same, as long as the revision number is ignored along the execution stages.

5.3. Postprocess stages

Once the results are completely generated in the *target* machine there are several stages required to build a set of figures:

fetch— waits until all the experiment units are completed and then executes the next stage. This stage is performed by the *garlic(1)* tool using the *-F* option and also reports the current state of the execution.

store— copies from the *target* machine into the nix store all log files generated by the experiment, keeping the same directory structure. It tracks the execution state of each unit and only copies the results once the experiment is complete. Other files are ignored as they are often very large and not required for the subsequent stages.

timetable— converts the results of the experiment into a NDJSON file with one line per run for each unit. Each line is a valid JSON object, containing the *exp*, *unit* and *run* keys and the unit configuration (as a JSON object) in the *config* key. The execution time is captured from the standard output and is added in the *time* key.

merge— one or more timetable datasets are joined, by simply concatenating them. This step allows building one dataset to compare multiple experiments in the same figure.

rPlot— one ot more figures are generated by a single R script[2] which takes as

[2] Winston Chang, *R Graphics Cookbook: Practical Recipes for Visualizing Data*, O'Reilly

input the previously generated dataset. The path of the dataset is recorded in the figure as well, which contains enough information to determine all the stages in the execution and postprocess pipelines.

5.4. Current setup

As of this moment, the *build* machine which contains the nix store is *xeon07* and the *target* machine used to run the experiments is Mare Nostrum 4 with the *output* directory placed at `/gpfs/projects/bsc15/garlic`. By default, the experiment results are never deleted from the *target* so you may want to remove the ones already stored in the nix store to free space.

Media (2020). 2nd edition.

Appendix A: Branch name diagram