OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

Bachelor Thesis

Automated performance analysis of software stacks for distributed systems

Fabian Schröder
`fabian.schroeder@st.ovgu.de`

September 13, 2023

First Reviewer:
Prof. Dr. Michael Kuhn

Second Reviewer:
M.Sc. Michael Blesel

Abstract

The performance of clusters is influenced by both the hardware and the software it uses. For key functionalities such as a compiler and libraries such as MPI and BLAS, there are a large number of different implementations that might achieve different perforamnces depending on the exact cluster setup. Finding well-performing implementations is therefore important to maximise the efficiency with which the available resources are used in terms of program runtime, throughput, etc.

HPC-Benchmarks-2, an extension of HPC-Benchmarks, is a configurable tool designed to find a software stack consisting of a compiler and an implementation of MPI and BLAS on a SLURM-based cluster, using spack to manage the different software configurations. The stack is built during in a slurm job by iteratively running benchmarks, analysing the results obtained and selecting the best software implementation of each of the three types. The comparison is based on an estimate of the performance improvement each implementation could offer, using only selected performance metrics and weights to quantify their assumed importance.

The program largely works as intended, although there are some software incompatibility issues. The results suggest that a combination of the specifications gcc@13.2.0 (compiler), mpich@4.1.2 netmod=tcp (MPI) and openblas@0.3.23 (BLAS) form the best software stack for the cluster on which the tests were performed. The choice of weights and metrics was mixed in the extent to which they accurately matched and emphasized the performance differences between the software implementations.

Contents

Chapter 1.

Introduction

High performance computing has had a profound impact on our world and will continue to do so in a wide range of areas, from modelling global climate to analysing gene sequences and many more. A common approach used in scientific institutions and industry is clusters, computers that are interconnected to combine their resources, providing a cost-effective way to create large amounts of computing power.[Bauke and Mertens, 2006, chapter Introduction]

However, the performance of a cluster doesn't just depend on the hardware used, but also on the software running on it [Leiserson et al., 2020]. Software implementations can differ in the way their algorithms work or their optimisations, resulting in performance differences. Finding the best software is therefore an important aspect of maximising the computing power of a cluster.

Clusters use a variety of software in a so-called software stack, which makes optimisation difficult due to the number of possible configurations. Until now, there have been ways to analyse and rank the performance of clusters (which is influenced by both software and hardware), and to analyse the performance of specific pieces of software (e.g. compilers, MPI) on specific hardware, but no way to automatically optimise the selection process of the pieces of the software stack for a given cluster. Therefore, finding an approach to do so would provide a user-friendly way to improve the efficiency with which the available computing resources are used and the throughput of the system.

This thesis presents HPC-Benchmarks-2 [Schröder, Fabian, 2023], a tool designed to maximise the performance of a cluster by building a software stack consisting of a compiler, an implementation of MPI and a BLAS implementation, and analysing their results in a selection of benchmarks. It is a fork of HPC-Benchmarks [Lafontaine and Cirkov, 2022] , a program to run and analyze benchmarks on clusters, and uses much of its basic functionality.

The thesis will be structured in the following way:

- **Background** introduces the basic aspects necessary to understand the rest of the thesis and the tools that will be used to implement the stack optimisation.
- **Related Work** covers some approaches to comparing the performance of clusters or different types of software.
- **Benchmarks and Packages** discusses the benchmarks that will be used to collect data, general categories of software that will make up our stack, and specific implementations that might be tested.
- **Design and Implementation** details the general planning of the optimisation, the changes required to the existing tools and the implementation of the proposed solution.

- **Evaluation** covers the precise test environment, how the program is run, its results and a discussion of the findings.
- **Conclusion and Future Work** summarises the thesis and lists some aspects that can be improved upon.

Chapter 2.

Background

2.1. Clusters

Clusters are a common architecture in high performance computing. They consist of a number of interconnected computers that can work together on a problem. [Baker and Buyya, 1999] This is done for a number of reasons, including cost and scalability advantages and greater configuration flexibility compared to dedicated hardware.[Sterling, 2011] The exact structure may vary from cluster to cluster, but a simple one can be seen in Figure 2.1.

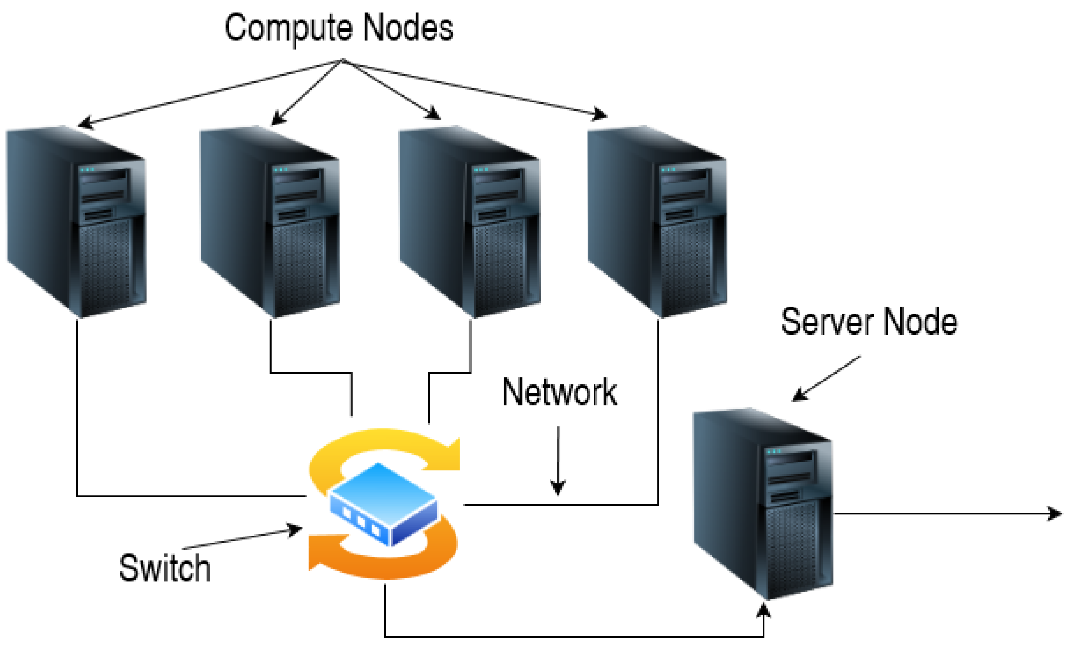


Figure 2.1.: A simple cluster consisting of four compute nodes connected by a network and a switch and a server node that allows external access.The figure is on an image from [Bauke and Mertens, 2006]

The individual computers, also called nodes, consist of several important components. These are as follows[Sterling, 2011]:

- At least one processor, usually with more than one core. The most popular processor architecture for clusters is x86.
- Main memory and a cache hierarchy.

- A motherboard that enables communication within the node, the BIOS and all the node's control functions.
- A network interface controller required to connect nodes on a network.
- Sometimes, but not always, nodes have secondary storage.

Sometimes a cluster includes nodes that perform more specific tasks, such as user login or system administration. [Sterling, 2011]

These nodes are connected by a private System Area Network (typically Ethernet), which includes the nodes' Network Interface Controllers, the channels or cables that run between them, and network switches to route traffic. The network topology may vary from cluster to cluster. [Sterling, 2011]

The software aspect of a cluster covers two important areas. The first is the application programming environment, which is basically the way in which parallel computing is achieved on the cluster. Today, this is typically done using message passing and the Message Passing Interface (MPI), which allows processes on different nodes to communicate with each other. [Sterling, 2011]. Other options include data-parallel programming. [Baker, 2000] The second aspect is resource management and scheduling [Baker and Buyya, 1999], which will be discussed in more detail in the following section.

2.2. slurm

Using the hardware resources of a cluster involves additional overheads granting access to users, allowing them to track the status of their tasks etc. This requires a resource management and scheduling tool. Resource management involves the allocation of compute resources and authentication, while scheduling includes queuing of tasks and resource allocations. [Baker and Buyya, 1999]

Slurm is such a tool and has been designed with high scalability and simplicity in mind. It's three key functions are as follows [Yoo et al., 2003]:

- It allocates access to the available resources to users for some duration of time. This allows a user to request and receive nodes from the cluster to run a program or workload.
- It provides a framework that allows the user to launch, monitor and execute the desired work on the allocated nodes. For example, if the user wants to run a benchmark on the cluster, slurm allows them to run it and check its status.
- It uses a queue of pending work to arbitrate conflicting resource requests. This means that if there are multiple requests that can't be granted at the same time (e.g. two users requesting 60% of all nodes in a cluster), they can be held in the queue to be granted at a later time (e.g. when one of the users' allocations expires).

The architecture of slurm is shown in figure 2.2. A centralised manager, slurmctld, is used to monitor resources and work. Optionally, there may be a secondary slurmctld that can continue to perform these functions if the primary slurmctld fails. Each node has a slurmd daemon that cyclically listens for work, executes it and reports its status. They communicate in a hierarchy. There may also be a slurmdbd or slurm DataBase Daemon to facilitate the communication of accounting information from multiple clusters to a single database. [Yoo et al., 2003]

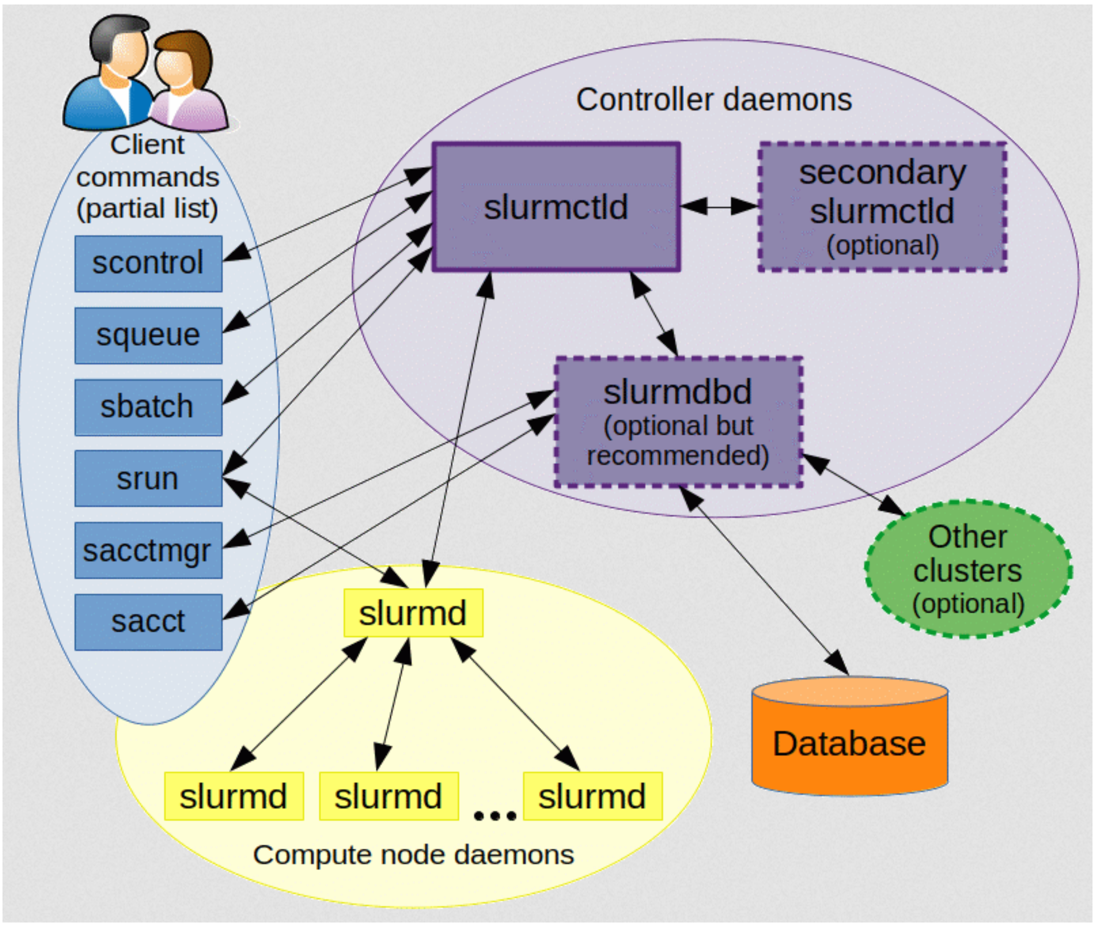

Controller daemons
Client commands (partial list)
scontrol
squeue
sbatch
srun
sacctmgr
sacct
slurmctld
secondary slurmctld (optional)
slurmdbd (optional but recommended)
Other clusters (optional)
slurmd
Database
slurmd
slurmd
slurmd
Compute node daemons

Figure 2.2.: Graphical representation of slurms architecture. Image taken from [SCHEDMD, 2021]

The work to be executed are called jobs, each identified with their own JobID, and a single job may contain several tasks, possibly running in parallel. These are called job steps. This structure can be seen in figure 2.3.

Slurm also has a large number of number of commands, each with its own arguments. Only a few of these will be used in this thesis, and their use will be demonstrated using an example example job script in the listing 2.1. The information is based on the slurm manual [SCHEDMD, 2023], which also contains additional information that will be omitted here for the sake of brevity.

The script creates a file called test.txt, sleeps for 10 seconds to prolong its run time and allow us to check its status, echoes the text "Hellow world!" into the file created earlier end finishes after echoing SLURM_JOB_NODELIST. Using this script, we will have a look at some of slurm's commands and their respective arguments. The submission and output of the script can be seen in listing 2.2.

```
[fschroed@ants test_dir]$ sbatch batch.sh
Submitted batch job 765184

[fschroed@ants test_dir]$ squeue -j 765184
    JOBID PARTITION     NAME     USER ST       TIME  NODES
       ↪ NODELIST(REASON)
```

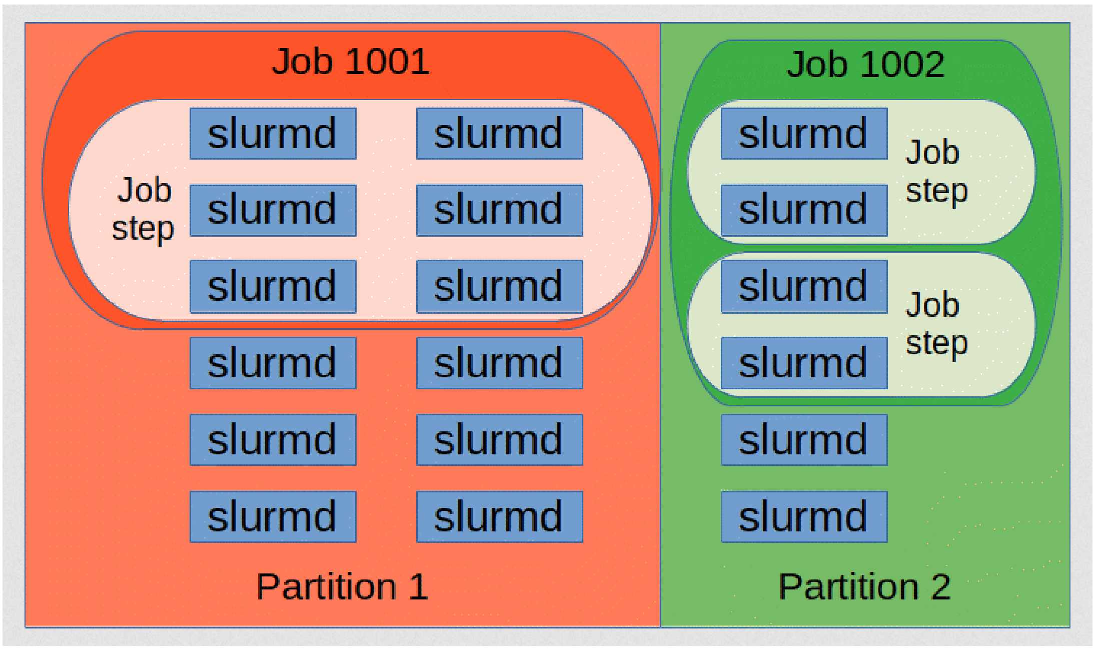


Figure 2.3.: The structure of two jobs running on different partitions. Image taken from [SCHEDMD, 2021]

```
    765184 vl-parcio slurm-ex fschroed  R       0:09      2
       ↪ ant[13-14]

[fschroed@ants test_dir]$ cat test.txt
1: Hello world!
2: Hello world!
0: Hello world!
3: Hello world!

[fschroed@ants test_dir]$ cat out.txt
ant[13-14]
```

Listing 2.2: Submission of the job script, tracking the job's status with squeue and the output of the script

- `sbatch batch.sh` lets us submit the script as a job to the slurmctld, which will try to allocate the requested resources, schedule the job amd return its JobID. Arguments for this command can be provided by including them when invoking sbatch or as a part of the script itself, which is the case in this instance. Listing 2.1 shows them at the beginning of the script, preceded by `#SBATCH`. The meaning of the arguments is as follows:
  - `--job-name=slurm-example` sets the name of the job to "slurm-example".
  - `--partition=vl-parcio` allows us to specify the partition our job should run on, "vl-parcio" in this case.
  - `--ntasks=4` specifies that we need a maximum of 4 (parallel) tasks, meaning 4 processes unless we include the `--ncpus-per-task` argument to the number of threads.

```
 1  #!/bin/bash
 2  #SBATCH --job-name=slurm-example
 3  #SBATCH --partition=vl-parcio
 4  #SBATCH --ntasks=4
 5  #SBATCH --nodes=2
 6  #SBATCH --ntasks-per-node=2
 7  #SBATCH --output=out.txt
 8  #SBATCH --error=err.txt
 9
10  touch test.txt
11  sleep 10s
12  srun -n 4 -l echo "Hello world!" >> test.txt
13  echo $SLURM_JOB_NODELIST
```

Listing 2.1: A simple script used to demonstrate the submission and execution of slum jobs

  - `--nodes=2` requests a minimum of 2 nodes for our job.
  - `--ntasks-per-node=2` sets the maximum number of tasks per node to 2. Since we have requested 4 tasks on 2 nodes, we can expect that 2 tasks will be running per node.
  - `--output=out.txt` specifies were the standard output should be directed, in this case out.txt. In this example, list of nodes allocated for the job have been echoed into the file.
  - `--error=err.txt` specifies were the standard error should be directed, in this case err.txt.

- `srun` can be used to start job steps, which might involve parallel work. It's invoked in line 12 of the script, using the command line argument `-n` to specify the number of tasks to echo a short text into a file. Due to the `-l` flag, the standard output of every task is preceded by its number. Since this echoing is done in parallel by every task, the file now contains the 4 lines that can be seen in 2.2.
- `squeue` shows information about the job queue. Here it has been invoked with the -j flag, which allows us to track the status of our job by providing its ID. The partition, job name and number of nodes matches the arguments we have specified in the script. Additionally, we can see the current status of the job (R, ie. running), and the time since its execution started.

2.3. spack

For scientific applications to work correctly, specific versions of compilers, message passing and additional dependency libraries are required. Due to the large number of possible configurations, a tool to manage this complexity would be extremely helpful. Spack was designed to do just that. This section is based on [Gamblin et al., 2015].

Spack, short for Supercomputing Package manager, allows parametric building of packages and dependencies. It provides "[composable] packages, explicitly parameterised by version, platform, compiler, options, and dependencies" [Gamblin et al., 2015], where package refers to a Python script used to build specific software. These scripts contain all the instructions needed to install the software, along with additional information ranging from different versions available to the package's dependencies. When installing packages, spack examines this information to construct a directed acyclic graph of its dependencies. The DAG of the mpileaks package can be seen in figure 2.4.

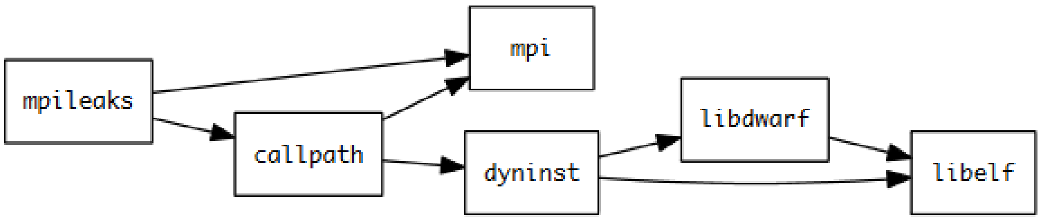


Figure 2.4.: DAG of the mpileaks package. Image taken from [Gamblin et al., 2015]

Some libraries are interchangeable and share a common interface, meaning that an application buildable with one of these packages is likely to be buildable with another. Spack supports this interchangeability by grouping these packages together as a virtual dependency, which does not refer to a specific implementation, but to a library interface [Gamblin et al., 2015] . Different MPI implementations, such as OpenMPI and MPICH, are a good example of this; they are grouped together in the MPI virtual dependency, which is also shown in the figure 2.4.

To work with these possible configurations, spack uses a new and recursive syntax. The individual build configurations are called specs. Some of its main features are [Gamblin et al., 2015]

- @ Specifies the desired version of a package.
- % Specifies the compiler to use for the package. The compiler version can also be specified with the symbol.
- Compiler flags and optional components can also be added. This results in different variants of the package.
- ^ Describes dependencies and provides a way to use a specific configuration of the required package.

These options impose constraints on the software configurations and give the user the freedom to specify them to the desired degree. Spack can then internally fill in the gaps and concretise the complete configurations (i.e. select dependencies, compilers, version etc.) needed to install the packages [Gamblin et al., 2015] . Continuing with the mpileaks example, configurations of varying complexity can be seen in the table 2.1

All of this leads to two additional points of note, which will be discussed now. Different subgraphs of a configuration's DAG may be identical, meaning that they may share some of their dependencies. In such cases, spack will reuse them instead of creating identical copies. The other important aspect is spack's use of hashes. Packages and their configurations must be

spec	Meaning
mpileaks	mpileaks package, no constraints.
mpileaks@1.1.2	mpileaks package, version 1.1.2.
mpileaks@1.1.2 %gcc	mpileaks package, version 1.1.2, built with gcc at the default version.
mpileaks@1.1.2 %intel@14.1 +debug	mpileaks package, version 1.1.2, built with Intel compiler version 14.1, with the “debug” build option.
mpileaks@1.1.2 =bgq	mpileaks package, version 1.1.2, built for the Blue Gene/Q platform (BG/Q).
mpileaks@1.1.2 ^mvapich2@1.9	mpileaks package version 1.1.2, using mvapich2 version 1.9 for MPI
mpileaks @1.2:1.4 %gcc@4.7.5 -debug =bgq ^callpath @1.1 %gcc@4.7.2 ^openmpi @1.4.7	mpileaks at any version between 1.2 and 1.4 (inclusive), built with gcc 4.7.5, without the debug option, for BG/Q, linked with callpath version 1.1 and building callpath with gcc version 4.7.2, linked with openmpi version 1.4.7

Table 2.1.: Different speaks and their respective meaning. The table is based on a table from [Gamblin et al., 2015]

uniquely identifiable, but since the concrete specs that define them can become impractically long and complex, hashes of them are used instead [Gamblin et al., 2015].

Spack provides a large number of commands, but only some of them are important for this thesis. These will be covered now, together with the output they generate. Additional info is available in the command reference [Spack, 2023]

`spack info mpileaks` shows the information available for the mpileaks package, including a short description, available versions and the package’s dependencies (listing 2.3)

```
[fschroed@ants ~]$ spack info mpileaks
AutotoolsPackage:   mpileaks

Description:
    Tool to detect and report MPI objects like MPI_Requests and
    MPI_Datatypes.

Homepage: https://github.com/LLNL/mpileaks

Preferred version:
    1.0
       ↪ https://github.com/LLNL/mpileaks/releases/download/v1.0/
       ↪ mpileaks-1.0.tar.gz

Safe versions:
    1.0
       ↪ https://github.com/LLNL/mpileaks/releases/download/v1.0/
       ↪ mpileaks-1.0.tar.gz
```

```
Deprecated versions:
    None

Variants:
    Name [Default]    When    Allowed values          Description
    ==============    ====    ====================
       ↪ ==============================================

    stackstart [0]    --      [omitted for brevity]     Specify
       ↪ the number of stack frames to truncate

Build Dependencies:
    adept-utils  callpath  gnuconfig  mpi

Link Dependencies:
    adept-utils  callpath  mpi

Run Dependencies:
    None
```

Listing 2.3: output of the command `spack info mpileaks`

`spack install mpileaks` lets us install the package and its dependencies (listing 2.4).

```
[fschroed@ants HPC-Benchmarks]$ spack install mpileaks
[output for the depedencies; removed to save space]
==> Installing mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z
==> No binary for mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z
   ↪ found: installing from source
==> Using cached archive:
   ↪ /home/fschroed/spack/var/spack/cache/_source-cache/archive
   ↪ /2e/2e34cc4505556d1c1f085758e26f2f8eea0972db9382f051b2dcfb1d
   ↪ 7d9e1825.tar.gz
==> No patches needed for mpileaks
==> mpileaks: Executing phase: 'autoreconf'
==> mpileaks: Executing phase: 'configure'
==> mpileaks: Executing phase: 'build'
==> mpileaks: Executing phase: 'install'
==> mpileaks: Successfully installed
   ↪ mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z
  Fetch: 0.00s.  Build: 12.49s.  Total: 12.50s.
[+] /home/fschroed/spack/opt/spack/linux-centos8-x86_64_v3/
   ↪ gcc-12.1.0/mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z
```

Listing 2.4: Installation process of mpileaks

`spack find mpileaks` can now be used to look for installed configurations of mpileaks, grouped according to the OS, architecture and compiler. Using `-l`, we can see the shortened hash of the package, and the intallation path can be accessed by using the `--paths` argument. Additionally, we can also use this hash as an argument for spack find instead of the spec (listing 2.5).

```
[fschroed@ants ~]$ spack find mpileaks
-- linux-centos8-x86_64_v3 / gcc@12.1.0 -------------------------
mpileaks@1.0
==> 1 installed package

[fschroed@ants ~]$ spack find --paths mpileaks
-- linux-centos8-x86_64_v3 / gcc@12.1.0 -------------------------
mpileaks@1.0
   ↪ /home/fschroed/spack/opt/spack/linux-centos8-x86_64_v3
   ↪ /gcc-12.1.0/mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z
==> 1 installed package

[fschroed@ants ~]$ spack find -l mpileaks
-- linux-centos8-x86_64_v3 / gcc@12.1.0 -------------------------
4zqzqvj mpileaks@1.0
==> 1 installed package

[fschroed@ants ~]$ spack find /4zqzqvj
-- linux-centos8-x86_64_v3 / gcc@12.1.0 -------------------------
mpileaks@1.0
==> 1 installed package
```

Listing 2.5: Usage of the "spack find" command and some command line arguments; –paths shows the installation location of the packages, -l adds the shortened hash of the package to the output, and the last command shows how a hash can be instead of a spec to find installed packages

`spack cd -i mpileaks` lets us change the directory to the path were mpileaks is installed. As we can see, the path matches the one we found with `spack find --paths mpileaks`. This command is useful if we need to access the binary of the package. `spack uninstall /4zqzqvj` lets us uninstall the package (listing 2.6).

```
[fschroed@ants ~]$ spack cd -i mpileaks
[fschroed@ants mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z]$ pwd
/home/fschroed/spack/opt/spack/linux-centos8-x86_64_v3/gcc-12.1.0/
   ↪ mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z

[fschroed@ants mpileaks-1.0-4zqzqvjp5jiooejyh3cgpeczedkpf75z]$
   ↪ spack uninstall mpileaks
==> The following packages will be uninstalled:

    -- linux-centos8-x86_64_v3 / gcc@12.1.0
       ↪ -------------------------
    4zqzqvj mpileaks@1.0

==> Do you want to proceed? [y/N] y
==> Successfully uninstalled mpileaks@1.0%gcc@12.1.0 stackstart=0
   ↪ arch=linux-centos8-x86_64_v3/4zqzqvj


-Verzicht auf Details zu den Pfaden, concretization
```

Listing 2.6: Demonstration of "spack cd" and "spack uninstall"

2.4. HPC-Benchmarks

This chapter is based on the documentation of HPC-Benchmarks [Lafontaine and Cirkov, 2022], which, to the best of my knowledge, hasn't been made public.

HPC-Benchmarks [Lafontaine and Cirkov, 2022] is a tool written in python designed to help manage various aspects involved in managing and running benchmarks on high performance computing systems. Users can create configuration files like the one shown in listing 2.7 for the currently supported benchmarks HPL, HPCG and OSU-Micro-Benchmarks. The name of these files needs to follow the pattern <benchmark>_cfg_<name>.txt (e.g. hpl_cfg_1.txt).

```
--------------------meta settings--------------------
2                                       [iterations]
avg                                     [aggregation: e.g. avg,
   ↪ max]
------------------base dependencies------------------
3.1                                     [HPCG Version]
gcc                                     [Compiler]
12.1.0                                  [Compiler Version]
openmpi                                 [MPI Implementierung]
4.1.3                                   [MPI Version]
gcc                                     [MPI Compiler]
12.1.0                                  [MPI Version]
-----------------benchmark parameters-----------------
120 120 120                             [problem size]
60                                      [time target in s]
-------------------SLURM parameters-------------------
                                        [Partition]
                                        [Nodes]
                                        [Prozesse]
                                        [Prozesse/Tasks per Node]
                                        [CPUs pro Prozess/Task]
                                        [RAM in MB/CPU]
                                        [Startpunkt, YYYY-MM-DD
                                           ↪ HH:MM:SS]
                                        [Zeitlimit, DD-HH:MM:SS]
                                        [Email an User]
                                        [Email-Trigger, z.B.
                                           ↪ END,FAIL,TIME_LIMIT_80]
                                        [Ziel/File fuer Output]
                                        [Ziel/File fuer Fehler]
```

Listing 2.7: A configuration for the HPCG benchmark. Included are meta settings for the execution of the benchmark, the desired spec split into packages, compilers and versions, the parameters for running the benchmark, and the slurm parameters for running it.

Based on these configuration files, HPC-Benchmarks allows the user to install and manage different specs for the benchmarking software using spack, and to run them with different problem sizes and resource allocations using slurm. In addition, several checks are included to avoid possible errors such as installing packages that are already installed or running specs that aren't installed.

HPC-benchmarks can be used interactively or through the command line interface, the latter being the one we will focus on.

To install benchmarks, the main Python file sb.py must be run with the `-i` flag. The user can now specify the name of the benchmark to be installed and the `<name>` part of the configuration file mentioned earlier. If these specifications aren't already installed and pass a syntax check, they will be added as job steps to an installation script, which will then be submitted to slurm. This can be seen in the listing 2.8.

```
[fschroed@ants HPC-Benchmarks]$ python3 sb.py -i hpl 1,2

loading hpl
[============]

Submitted batch job 765188
```

Listing 2.8: Installing the hpl configurations 1 and 2 via the command line. The JobID of the installation script is part of the output

Running benchmarks works in a similar way, as can be seen in listing 2.9. Inside of the `projects` folder, a new folder is generated for this benchmark run containing the individual benchmark job scripts and a central script, `batch.sh`, that will submit the other scripts one after another to slurm. Before finishing, `batch.sh` launches a last job to plot some of the data gathered from the benchmarks. All of the results will be in another folder, called `<benchmark>_res@<time>_<date>` instead of `<benchmark>_run@<time>_<date>`. A shortened example of `batch.sh` can be seen in listing 2.10.

```
[fschroed@ants HPC-Benchmarks]$ python3 sb.py -r hpl 1,2

loading hpl
[============]
script building completed:
/home/fschroed/bm-tool/HPC-Benchmarks/projects/hpl_run@
   ↪ 230721_16072023/batch.sh



Submitted batch job 76519
```

Listing 2.9: Submitting the hpl configurations 1 and 2 to the slurmctld via the command line. The output includes the location of the submitted script and its JobID

```
id0=$(sbatch <project_path>/cfg_to_dat_hpl_cfg_1.sh)
id1=$(sbatch --dependency=afterany:${id0##* }
   ↪ <project_path>/1\#hpl_cfg_1.sh)
id2=$(sbatch --dependency=afterany:${id1##* }
   ↪ <project_path>/2\#hpl_cfg_1.sh)
id3=$(sbatch --dependency=afterany:${id2##* }
   ↪ <project_path>/cfg_to_dat_hpl_cfg_2.sh)
id4=$(sbatch --dependency=afterany:${id3##* }
   ↪ <project_path>/1\#hpl_cfg_2.sh)
id5=$(sbatch --dependency=afterany:${id4##* }
   ↪ <project_path>/2\#hpl_cfg_2.sh)

source
   ↪ /home/fschroed/forked-bm-tool/spack/share/spack/setup-env.sh
spack load python /kfmsq4afjpppkt4xqt3gpqqt4fldypxi
sbatch --dependency=afterany:${id5##* } <project_path>/plot.sh
```

Listing 2.10: Shortened example of a `batch.sh` script with the actual path to the scripts being replaced to save space. It submits `cfg_to_dat_hpl_cfg_<cfg>.sh` scripts to set the benchmark parameters in the respective `.dat` file and the scripts for each individual benchmark run. These are chained together with slurm dependencies to ensure that the next job starts running only once the previous one terminated. The final submitted and running job is the one that plots the results

Chapter 3.

Related Work

During my research for this thesis, I didn't find much work that was really closely related to the topic at hand, i.e. the performance evaluation of different types of software for the purpose of constructing an efficient software stack. Most of the sources I came across dealt with one of two things; some of them compared the performance of different software implementations (e.g. different MPI variants), and others tried to quantify the performance of a system/cluster as affected by both the software and the hardware.

In terms of evaluating the performance of a system, HPL [Dongarra et al., 2003] and HPCG [Heroux et al., 2013] , two different but related benchmarks, may be a good place to start. Both attempt to quantify the available computing resources by solving random linear systems of configurable size, and calculating and reporting the floating-point operations per second achieved. This is useful for this thesis as a way of comparing the performance of different software stacks. Although HPL and HPCG are similar, they have certain differences in their algorithms and the resulting limitations of their measurements. These are discussed in more detail in the relevant sections of the next chapter.

Performance evaluation of different types of software is also something that has been done in other literature. For example, different compilers have been compared in "An Evaluation of the Fujitsu A64FX for HPC Applications" [Poenaru et al., 2021], where a variety of benchmarks were used to evaluate them on Fujitsu hardware. MPI implementations have been tested in "Software Architecture and Performance Comparison of MPI/Pro and MPICH" [Dimitrov and Skjellum, 2003] where their achieved bandwidths and Gflops are compared in the LINPACK benchmark. Work like this allows us to get an idea of how implementations of this type of software can be compared and software implementations, what differences might be expected, etc.

In addition, the paper "How well can simple metrics represent the performance of HPC applications? [Carrington et al., 2005] provides valuable insights into the reliability of using synthetic benchmark results to estimate the performance of real applications on a computer system. This is done by measuring the execution time of a selection of applications on reference systems, and estimating the execution time of the applications on the system being evaluated based on a ratio of the benchmark results for the current and reference systems. The error and variance are calculated after the applications have been run on the other system. It shows that a combination of different metrics gives more accurate results than single metrics. Although this paper doesn't focus on the software aspects of system performance, it can still be a very useful resource as it provides an approach to estimating performance and shows the importance of the choice of metrics.

Chapter 4.

Benchmarks and Packages

4.1. General thoughts

In order to find an optimal software stack through the use of benchmarks, three fundamental issues need to be addressed. First, it is necessary to decide on the relevant performance metrics of the cluster. Second, the elements of the software stack need to be defined. Third, benchmarks need to be selected that use the software stack and provide the agreed upon performance metrics.

Over the years, many different ways of evaluating the performance of HPC systems have been proposed or used. One of these is their performance on the LINPack benchmark, which measures the number of floating point operations per second (flops) and forms the basis of the TOP500, a ranking of the most powerful computer systems [Dongarra et al., 2003]. Although this is a simple and straightforward method, it can actually have a negative correlation with system performance in real applications [Gustafson and Todi, 1998]. The inclusion of additional metrics is therefore important, and it has been shown that a combination of flops and data on the memory access and communication performance can be used to predict a system's performance with reasonable accuracy [Carrington et al., 2005]. For this reason, these metrics will be used to compare the different software stacks.

In order to keep track of the software stack and manage the different installations of the benchmarks, spack is used as it seems to be the most practical option for this task. As a result, the software selection is limited to packages available through it and will be discussed in more detail in the following sections.

In this thesis, the software stack consists of a few selected parts, namely a compiler, an implementation of mpi, and an implementation of BLAS, a mathematical library commonly used in scientific applications. LAPACK and FFT were also considered as possible candidates, but were excluded due to a lack of working benchmarks to test them. LAPACK implementations were also almost exclusively bundled with BLAS, making it impossible to test them separately.

The selection of benchmarks includes HPL, HPCG, OSU and HPCC, all of which will be covered in the next section. Another possible option was Cbench, which combines several benchmarks, but it was ultimately excluded because it couldn't be installed with the selected compilers. The dependencies listed for each benchmark have been gathered though `spack info`.

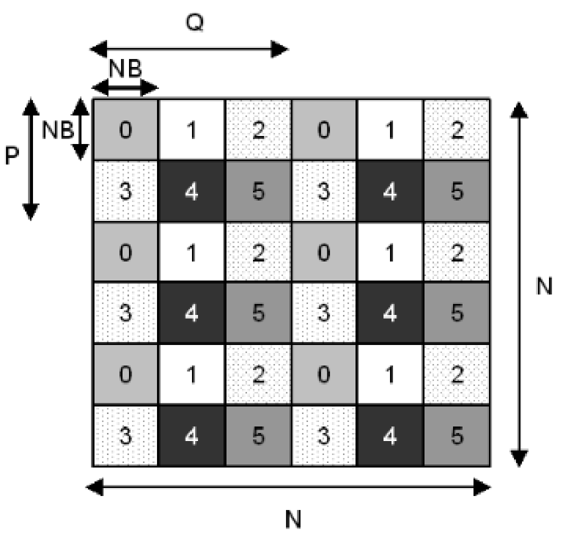


Figure 4.1.: Graphical representation of HPLs paramters and an example of a P by Q partition of the resulting matrix [Farreras et al., 2009]

4.2. Benchmarks

4.2.1. HPL

This section is based on [Dongarra et al., 2003].

HPL is a portable implementation of the HPLinpack Benchmark. It generates and solves a dense random linear system of equations of a given order n, as can be seen in equation 4.1

$$Ax = b; A \in R^{n \times n}; x, b \in R^n \tag{4.1}$$

This involves the computation of the LU factorization with row partial pivoting like in equation 4.2, after which the solution x can be calculated by solving equation 4.3

$$P_r = [A, b] = [[LU], y]P_r, L, U \in R^{n \times n}, y \in R^n \tag{4.2}$$

$$Ux = y \tag{4.3}$$

HPL measures the number of floating points operations per second (flops) [Dongarra et al., 2003], and since the amount of data per operation is rather small, HPL can nearly reach the theoretical performance limit of a system. However, this behaviour is hardly representative of real applications, which tend to be more memory bound [Marjanović et al., 2015].

It can be configured using several parameters, the most important of which can be seen in figure 4.1. These are [Dongarra et al., 2003]:

- N: size of the coefficient matrix
- NB: blocking factor, size of the blocks that the matrix is partitioned into

- P and Q: Dimensions of the grid that is used to ciclicly spread the blocks over the processes

HPL depends on MPI and BLAS.

4.2.2. HPCG

This section is based on [Heroux et al., 2013].

The High Performance Conjugate Gradient benchmark, or HPCG for short, is another benchmark measuring flops. It generates a sparse linear system which it then solves through using domain decomposition and a conjugate gradient method [Heroux et al., 2013]. The linear system has the size $(n_x * np_x) \times (n_y * np_y) \times (n_z * np_z)$, with np_x,np_y, and np_z being generated during the setup phase of the benchmark [Heroux et al., 2013]. Users can configure HPCG using the following parameters:

- n_x,n_y, and n_z describe the size of the local grid can be used to modify the problem size [Heroux et al., 2013]. Only multiples of 8 are allowed [AMD, 2023]
- The fourth paramter is the desired run time [AMD, 2023]

Compared to HPL, it better represents a system's performance for real applications. It's performance is also heavily reliant on the memory bandwidth [Marjanović et al., 2015].

HPCG depends on MPI.

4.2.3. OSU

OSU-Micro-Benchmarks is a collection of small tests designed to measure different aspects of MPI performance (bandwidth or latency in most cases). It offers a large variety of micro-benchmarks, covering both point-to-point and collective communication for messages of different sizes and for a large number of MPI operations [MVAPICH, 2023].

Latency describes the time taken to complete communication, and bandwidth is a measure of how much data is transferred in a given period of time. Since communication between processes is important for high performance computing, these metrics should definitely be included.

OSU depends on MPI.

4.2.4. HPCC

This section is based on [Luszczek et al., 2005].

The HPC Challenge benchmark suite, or HPCC for short, is a collection of the following benchmarks:

- HPL has already been covered.
- STREAM measures the sustainable memory bandwidth in Gbyte/s and the computation rate for four operations: COPY $c \leftarrow a$, SCALE $b \leftarrow \alpha c$, ADD $c \leftarrow a + b$ and TRIAD $a \leftarrow b + \alpha c$.

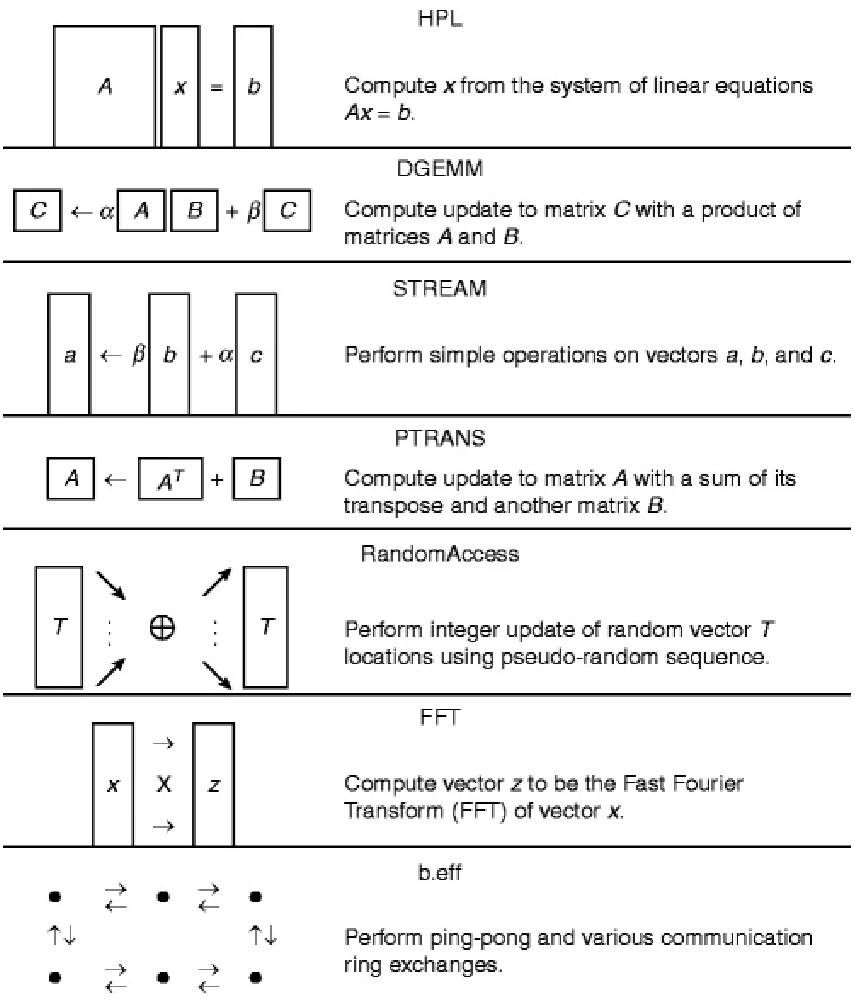

HPL
A
x
=
b
Compute x from the system of linear equations Ax = b.
DGEMM
C ← α A B + β C
Compute update to matrix C with a product of matrices A and B.
STREAM
a ← β b + α c
Perform simple operations on vectors a, b, and c.
PTRANS
A ← A^T + B
Compute update to matrix A with a sum of its transpose and another matrix B.
RandomAccess
T ⊕ T
Perform integer update of random vector T locations using pseudo-random sequence.
FFT
x X z
Compute vector z to be the Fast Fourier Transform (FFT) of vector x.
b.eff
Perform ping-pong and various communication ring exchanges.

Figure 4.2.: Graphical representation of the individual benchmarks included in [Dongarra and Luszczek, 2011]

- RandomAccess measures the rate at which integers are updates in random memory locations. This is also known as GUPS.
- PTRANS tests the communications of a system by simultaneously exchanging large messages between processors.
- FFTE is another benchmark that measures FLOPS, this time through a one-dimensional Discrete Fourier Transformation.
- DGEMM measures FLOPS through double precision matrix-matric multiplication $C \leftarrow \beta C + \alpha AB$, with $A, B, C \in R^{n \times n}; \alpha, \beta \in R^n$.
- b_eff measures latency and bandwidth for single-process-pairs and collectively for all processes at the same time in a ring structure.

Several of these benchmarks are executed in different scenarios, namely as a single and star configurations. "Single" in this case means that only one process is working, and "star" or "global" means that all process are executing the task [Luszczek et al., 2005] .

HPCC requires both MPI and BLAS.

4.3. Packages

Spack provides various packages that we can use as a compiler, MPI implementation and BLAS implementation. These may differ in the way their algorithms are designed or how they are optimised. For example, hardware vendors often provide their own implementations of certain functions or libraries that are designed to work particularly well with their particular hardware [Köhler and Saak, 2013]. Accordingly, their performance may vary depending on the exact implementation and system, so testing several different is a key aspect of finding an optimised software stack. The following subsections cover the three package types that make up the software stack of this thesis, each with featuring some of the available implementations. For the sake of space and to keep the focus on the optimization, the individual implementations will be described rather sparingly based on the information available for the package via `spack info`.

4.3.1. Compilers

To install software, spack needs to convert the source code into an executable program. This requires a compiler [Grune et al., 2012] . Since the performance of the program can vary depending on the compiler, it is important to test several and see which one gives the best results. Spack doesn't provide an easy way to view all installable compilers, so a manual search for possible candidates is required. Testing has shown that the compiler must work with both C/C++ and Fortran code, which further limits the choice. For the For the purposes of this thesis, we will focus on a handful of promising candidates for our cluster. These are:

- GCC: the standard GNU Compiler Collection
- AOCC: AMD's optimized c/c++ compiler
- clang: LLVM's c/c++ compiler
- intel-oneapi-compilers: collection of compilers from intel

Other possible options include arm and fujitsu. These were left out because they were designed to work with entirely different architectures.

4.3.2. MPI

Because a cluster may use processes running on different connected nodes to perform tasks, they need to be able to communicate with each other. This is typically done using message passing in the form of MPI (Message Passing Interface), an API that defines the syntax and semantics of a semantics of a software library that allows data to be exchanged between processes. between processes [Gropp, 2011]. There are many different implementations of this API, so it seems useful to select some of the more popular ones for testing:

- openmpi: open source MPI implementation from the Open MPI Project
- mvapich2: high performance MPI implementation for clusters
- intel-oneapi-mpi: implementation of the MPICH specification made by Intel
- mpich: high performance and portable MPI implementation

4.3.3. BLAS

BLAS, which stands for Basic Linear Algebra Pack, is an interface that provides common, basic mathematical operations. The current interface, Level 3 BLAS, supports vector-vector, matrix-vector and matrix-matrix operations [van de Geijn and Goto, 2011]. As these are extremely common types of computation, finding an implementation that works well for the implementation that works well for the given hardware is very important. Available implementations include:

- amdblis: BLAS-like dense linear algebra library implementation from AMD
- atlas: Automatically Tuned Linear Algebra Software, contains optimized implementations of BLAS
- cblast: OpenCL BLAS library
- flexiblas: "BLAS and LAPACK wrapper library with runtime exchangable backends" (as written in the output of `spack info flexiblas`)
- intel-oneapi-mkl: math kernel library from Intel
- openblas: optimized implementation of BLAS

Chapter 5.

Design and Implementation

5.1. Planning

With a rough selection of benchmarks and packages made, it is now time to plan the implementation of our tests. Finding a good performing software stack can be achieved in a number of ways. The most obvious options are to generate and test all potential stacks, or to narrow down the selection to a few promising candidates. The first option quickly becomes impractical due to the sheer number of possible stacks to install and test (the number of compilers times the number of MPI implementations times the number of BLAS implementations), so the second option is chosen.

The search for the software stack can be seen as an optimisation problem with the goal of finding the best performing one. Since only a few possible options are tested, there's no guarantee that the optimal solution will be found, but it's still possible to find something reasonably close. This can be done by using the greedy algorithm, which makes locally optimal choices to approximate the globally optimal one [Vince, 2002].

For our problem at hand, we can use the selection of the different packages in the stack (i.e. the compiler, MPI and BLAS) as local choices, and the best performing stack as the globally optimal choice. Using this approach, we are able to build the stack one package at a time, testing and selecting one of the available options. The order in which we select the package types can also affect the results, and should be specified here. The compiler is the obvious first choice, because it is required for the installation of each benchmark and package, and because of the way spack concretes the specs to be installed; adding the compiler immediately after the benchmark in the specs will guarantee that all other dependencies will also be installed using that compiler, which is what we want, so selecting it first is the natural choice. MPI will be the second choice, both because it is used in all the benchmarks, and because it is likely to have more impact than the last package type to be tested, BLAS.

The three packages differ in the benchmarks they are used in and the impact they might have on the result, so while it would be possible to run the same tests for each of them, tailoring them to the package is the more promising approach. This is covered in section 5.3.

To install and run the different benchmark configurations, HPC-Benchmarks will be used. As stated in the introduction, it can be used to install and run different configurations of each benchmark. The program's options allows us to install or run several configurations of a benchmark using just one command, which will create a batch script that launches additional jobsteps or job to do the installing/running (see listing 2.10 as an example). This forms a solid base for the optimization of the software stack. To complete the task, a few additional steps are necessary; configuration files need to be generated based on the current packages of the stack and the package we want to test before the installation can proceed, and after the benchmarks have finished the results need to be analyzed so that the best package can be chosen. This leads us to the following rough approach, as shown in listing 5.1:

```
optimization():
    for package in stack:
        generate_configs()
        install_benchmarks()
        #wait for the installation process to finish
        run_bechmarks()
        #wait untill the benchmark runs are finished
        analyze_()
        pick_best_package()
```

Listing 5.1: Rough workflow for the optimization process

The missing functionality could be implemented in a number of different ways, with varying degrees of involvement with HPC benchmarks. The program has no way of checking the status of a slurm job that is installing or running benchmarks. This requires the use of the sacct or squeue slurm commands, which means that shell commands have to be used in some way. Because of this, the first approach I considered and started working on was to create a batch script to do all the tasks in listing 5.1. However, this quickly proved to be rather cumbersome and difficult to work with, which led me to another approach that I ended up using for the rest of this thesis. Instead of writing a separate program to use HPC-Benchmarks for some tasks, I decided to add the required functionality directly to HPC-Benchmarks. Not only did this decision lead to a simpler solution, as there was no need for a complex helper program like the batch script would have been, but it also made it possible to work directly with the internal functions and data of the original code, which otherwise couldn't be accessed.

In order to perform the planned optimisation, the program needs to keep track of some data, the most important of which are the stack and the packages to be tested. HPC-Benchmark kept track of its data with a few global variables for things like the path to spack and the python file being executed. All the information about the benchmark configurations and their (meta) settings is kept in a large multi-dimensional list where the data can be accessed via its indices, slicing, etc. While this approach works and has less overhead than other possible alternatives, it did make the code and workflow of the program quite difficult to follow at times. To avoid any unnecessary changes to this system that might introduce new bugs in the future, I decided to leave these aspects as they were and use a new global variable for stack optimisation. This variable contains a dictionary, a set of key and value pairs, of data. A dictionary was chosen for two main reasons:

- This made the code more descriptive and easier to work with. For example, instead of iterating over indices where the data they represent may be unclear, a dictionary allows iteration over meaningful keys in the structure (such as the packages in the stack).
- The data in a dictionary can easily be stored as and read from json files by using the json package that's build into python. This presents a useful way of configuring the stack optimization without having to define a new syntax and implement functions to parse these configs, which is what HPC-Benchmarks does.

5.2. Performance comparison

To find the best packages for the software stack, it's necessary to compare and rank them using the performance metrics we've chosen. However, this is problematic because the importance of these metrics is both subjective and dependent on the workload [Carrington et al., 2005]. The lack of concrete applications for which we are optimising the stack further complicates the issue and makes a more flexible approach, where a user is able to configure the impact of each metric on the comparison, more appealing.

As mentioned previously, a combination of metrics has been shown to provide a reasonably accurate prediction of cluster performance for real applications. However, the more accurate predictions are based not only on benchmark results, but also on operation counts. We do not have access to these. Simpler methods estimate the runtime of an application on a system based on the runtime of the same application on a reference system and the ratio of the performance of the two systems in a benchmark [Carrington et al., 2005].

This approach cannot be used directly in this thesis because there is no reference system either, so we are missing two of the three values. However, to compare the performance of two spack packages, we can calculate the ratio of their estimated runtimes, in which case the two missing values cancel each other out, and we are left with an estimated speedup of one package over another, based solely on their respective performance in a benchmark. However, not all metrics are equal; in some cases a higher score is better (e.g. floating point operations per second), in other cases a lower score is preferable (e.g. communication latency). This leads to two similar equations for calculating the speedup $S(X_1, X_2)$ of package X_1 over package X_2 based on their respective benchmark performance $R(X)$, equation 5.1 for the first case and equation 5.2 for the second.

$$S(X_1, X_2) = \frac{R(X_1)}{R(X_2)} \tag{5.1}$$

$$S(X_1, X_2) = \frac{R(X_2)}{R(X_1)} \tag{5.2}$$

These equations make it possible to compare two packages based on a single benchmark result. However, as noted above, a combination of performance metrics is a better predictor than any one metric alone. One attempt to weigh such metrics is the IDC balanced rating, which uses fixed weights to rank HPC systems based on a score calculated from their processor performance, memory capacity, and scalability [HPCwire, 2001]. While the general idea is appropriate for the purpose of this work, it may not be flexible enough for different workloads. Instead, our performance comparison will use adjustable weights that can be configured by the user. Combining this idea with the speedup equations, we can create a way to compare two packages based on the speedups S_i (equations 5.1 and 5.2) estimated using the performance metrics R_i and their respective weights w_i:

$$Score(X_1, X_2, w) = \frac{\sum_{i=0}^{n-1} (S_i(X_1, X_2) * w_i)}{\sum_{j=0}^{n-1} w_j} \tag{5.3}$$

By normalising this score by dividing it by the sum of the weights, we can easily determine which package is better based on the estimates made: If the score is greater than 1, X_1 is better, otherwise X_2 is more promising.

5.3. Benchmark and Data selection for each package

With the general approach and performance comparison in place, it is now time to select the benchmarks and data that will be used to evaluate the performance of the of the selected implementations of each package type. This is is necessary for several reasons:

- Not every package type is used in all benchmarks. BLAS is only used in HPCC and HPL, so it is unnecessary to run HPCG and OSU for every implementation of it.

- The output of the benchmarks contains extensive data covering a variety of performance metrics and information regarding the benchmark run, making it impractical to include them all. It is therefore important to focus on the most promising.

- Running benchmarks takes a lot of time, and leaving out ones that are less important for the performance evaluation allows us to increase the problem sizes and to run the chosen benchmarks multiple times, which reduces the impact of random variations have on the more relevant metrics.

The weights of each metric will also be covered in the following subsections. Choosing the optimal values is probably impossible without a lot of additional context, such as the applications the stack is being optimised for, so the values need to be estimated.

5.3.1. Compiler

Out of the three package categories in our stack, the compiler has the largest amount of potential data to analyse as it is used to install and therefore affects the performance of each benchmark and package. Since the software stack is intended to be used for general purpose applications, it makes sense to include the Gflops that each compiler achieves on the closest approximation of such an application that we have access to, which is HPCG. A base weight of 1 is used.

Other important metrics that should be covered are the memory bandwidth and memory access of the system. Since the impact of BLAS and MPI on these can probably be neglected, the compiler provides the best way to include them. In addition, the Fujitsu compiler has been shown to achieve significantly better memory bandwidth on Fujitsu hardware than other compilers [Poenaru et al., 2021], and similar results may be possible for other compiler/hardware combinations. HPCC provides these metrics via the STREAM and RandomAccess benchmarks respectively, both as individual values and for a star configuration. Given the nature of the cluster, the star variants are chosen. Following the approach of [Poenaru et al., 2021], we will focus on the triad version of STREAM, given that STREAM can be used to predict the performance of the HPCG benchmark [Marjanović et al., 2015] and that GUPS has given more accurate predictions than the STREAM benchmark [Carrington et al., 2005], a weight of 2 and 3 respectively is chosen.

The table 5.1 gives a summary of our selection.

Benchmark	Metric	Weight
HPCG	Gflops	1
HPCC	StarStream Triad Gb/s	2
HPCC	StarRandomAccess GUPS	3

Table 5.1.: Benchmarks, metrics and weights for the compiler

Benchmark	Metric	Weight
HPCG	Gflops	2
osu_latency	Microseconds for s=8	10
osu_latency	Microseconds for s=16384	4
osu_latency	Microseconds for s=4194304	2
osu_bw	Gb/s for s=8	10
osu_bw	Gb/s for s=16384	4
osu_bw	Gb/s for s=4194304	2

Table 5.2.: Benchmarks, metrics and weights for the MPI implementations. s= denotes the message sizes

5.3.2. MPI

MPI is necessary for the processes to communicate with each other, so latency and bandwidth are obvious choices for the important data. OSU is used to measure these. Since OSU measures them for different data sizes [MVAPICH, 2023] the focus should be on the most common ones. However, the frequency of messages of a given size depends on the application, and without knowing the specific application for which the MPI implementation is to be optimised, we will again have to use the closest approximation we have, which is HPCG. Exact data is hard to come by, but it seems that short messages are the most common, with spikes in frequency for a few other message sizes [Ferreira and Levy, 2020]. In order to cover a wide range, message sizes 8, 16384 and 4194304 were chosen with weights of 10, 4 and 1 respectively, for both the latency and the bandwidth.

As the OSU benchmark runs in a few seconds, we can include another benchmark to extend the testing without running into time problems. Including not only the basic performance metrics, but also those that cover the efficiency of the MPI implementation during the course of computationally intensive work seems promising, so the HPCG benchmark was also chosen. This is similar to the approach used in [Dimitrov and Skjellum, 2003] but with a more relevant benchmark to measure the Gflops (HPCG instead of LINPack/HPL).

This is summarised in the table 5.2

5.3.3. BLAS

Compared to MPI and the compiler, BLAS is hardly used in our general benchmark selection. It’s not used in OSU or HPCG, so all data is based on HPCC results (which includes HPL).

Benchmark	Metric	Weight
HPCC	HPL Tflops	1
HPCC	SingleDGEMM Gflops	2
HPCC	StarDGEMM Gflops	3

Table 5.3.: Benchmarks, metrics and weights for the blas implementations

Information on the individual benchmarks included in HPCC has been hard to find, and to the best of my knowledge BLAS is only used in HPL and DGEMM [Krommydas, 2023], the letter of which includes a single and a star variant. The performance analysis for the BLAS implementation will therefore have to be limited to these metrics. Since HPL is a rather imprecise predictor [Carrington et al., 2005], it will be weighted by 1, as will SingleDGEMM, since it is local. StarDGEMM gets twice the weight.

This information can also be found in the table 5.3.

5.4. General changes to HPC-Benchmarks

While the core functionality of HPC-Benchmarks [Lafontaine and Cirkov, 2022] largely worked as intended, a number of changes had to be made to lay the groundwork for the planned automated search for a good software stack. These changes are somewhat difficult to document due to the complex and interconnected structure of the code; trying to explain them in detail with the context necessary to understand the problems together with their causes and solutions would be too long winded. Due to this, they will be covered in a shorter and more general manner that doesn't require deep knowledge of HPC-Benchmarks' code. These changes can be accessed through githubs compare function though, using the latest commits of the original project and HPC-Benchmarks-2.

The benchmarks could only be configured through the choice of packages/dependencies, compilers and versions, but not flags. This made it impossible to select specific package variants. To change this, support for flags has been added and the way the specs are written has been changed from a continuous string to one including spaces (e.g. `hpcc %gcc` instead of `hpcc%gcc`). This also required changes to the way specs were validated.

Certain packages were not compatible with the spec validation, because it has been build under the assumption, that all software installed via spack must be available as a package of the respective name. This however isn't the case. The intel compilers dpcpp, intel, and oneapi for example aren't available as separate packages, but combined in the "intel-oneapi-compilers" package. To solve this problem, a list of exceptions has been added which won't be validated like the other packages.

In addition, the benchmark parameters were read from a line with a hardcoded index, line 12 for example in the case of hpcg. This prevented the user from adding or removing lines in the "base dependencies" block of the config files, lest that the benchmarks were run with wrong parameters. To fix this issue, they are now read from the lines following the one containing the string "benchmark parameters".

To run the benchmarks, they were first loaded via `spack load` with the respective spec. This is only works, if the spec matches exactly one installed configuration, otherwise an error occurs because spack cannot determine which of the available configuration to load. For example, if the specs `hpl %gcc` and `hpl %aocc` are installed, `hpl` would match both, and loading it would fail. To fix this, the `--first` option has been added to the `spack load` command to ensure that the one of the available configurations is loaded (the first one).

A similar problem occurs, when HPC-Benchmarks searches for the installation paths of its benchmarks. It extracts the path from the output of `spack find --paths` starting at the index of the first occurence of the substring "/home" and removes all trailing spaces using the ".strip()" method. If multiple installed configurations match the spec, the substring following the paths will contain characters that wont be stripped, leading to the program reading a wrong and nonexistant path. This has been fixed by splitting the substring following "/home" at the "\n" character, at which point the correct path can be accessed at index 0 of the resulting list.

Running benchmarks worked by generating a batch script for each benchmark run and one to plot the results. Another script submitted these scripts as separate jobs and chained them together through the use of slurm dependencies to ensure that they were run in sequence. This script however terminated right after submitting all other scripts and was the only one whose jobID was returned to the user, which made keeping track of the status of a benchmark run more difficult. To address this issue, the "–wait" option was added to the plot script submission. This ensures that the main script only terminates once the plot script and therefor every other benchmarking script has finished.

As the original benchmark selection didn't include hpcc, it had to be added to the general workflow and script building pipeline.

The last minor change was a bug fix for hpcg. While the benchmark was provided with correct parameters, they didn't actually seem to be used when it was run. This appeared to be the case due to the fact that the benchmark was loaded and run while in the directory where the results were supposed to end up. Modifying the script generation to ensure that the job changes to the directory containing both the executable hpcg and its configuration file solved the problem.

5.5. New Functionality

Besides the previously mentioned changes to the existing workflow of HPC-Benchmarks, new features had to be included to enable the stack optimisation. These will be covered in this section.

5.5.1. folder and file structure

HPC-Benchmarks2 keeps file structure of the original HPC-Benchmmark untouched, and uses some additional folders to manage its data. These additions can be seen in figure 5.1.

Inside of the main folder HPC-Benchmarks-2, a new directory called `stacks` has been created that contains all of the stack configurations and optimization output. The individual configs are named according to the pattern `<stack>.json`. When running the tool for such a configuration, a new directory will be created named after the stack. This folder contains the output generated by slurm, `<stack>.out` and `<stack>.err`, a log file where the extracted benchmark results, package comparisons (both in the form of tables) and the final stack are printed, a directory

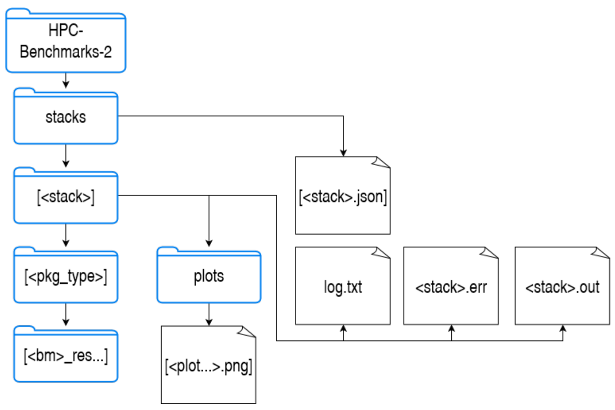


Figure 5.1.: Additional folders and files for HPC-Benchmarks-2. <x> denotes a pattern like "stack1" or "MPI", the square brackets indicate multiple files/directories

`plots` and one for each package type that has been tested, which in this thesis are `Compiler`,`MPI`, and `BLAS`. `plots` contains the graphs of the benchmark results in the form of a list of `.png` files, each named along the pattern of `plot\_pkg\_<bm>\_<i>.png` with `pkg` being the tested package type, `bm` the benchmark, and `i` as the index of the graphed result in the configuration. The images show a group of bar charts for each package, with one bar for each benchmark iteration and the average result.

The folders named after the packages contain copies of the output directories created by the benchmark run functionality of the original program.

5.5.2. Config Structure

As mentioned in the previously, the configurations for HPC-Benchmarks-2 are individual json files stored in the `stacks` directory. These files are read with python and converted into dictionaries. Since this data type consists of (key,value) pairs, those pairs will be discussed one after another using `stack1.json` as an example. This configuration is the template available at the github page [Schröder, Fabian, 2023] and has been used for the optimization in this thesis, making it the perfect candidate for this demonstration.

The config itself is rather long, which makes covering it in its entirety impractical. Due to this, it will be split into its (key,value) pairs, and redundant information will be left out.

The config begins with the list of benchmarks that the program is supposed to run for each package type. Testing the compiler for example requires both HPCC and HPCG, as can be seen in listing 5.2.

```
    "bms_for_package":{
        "Compiler":["hpcc","hpcg"],
        "MPI":["osu_latency","osu_bw","hpcg"],
        "BLAS":["hpcc"]
    },
```

Listing 5.2: List of benchmarks that should be run for the individual package types

The next section (listing 5.3) covers the data collected from the benchmark runs for each package of the given type and its weights. The package types act as a dictionary, containing pairs consisting of the benchmark and a list of lists containing information about the data selection. The sub-lists contain a name of the metric to be displayed in the generated plots and tables (see the figures and tables in section 6.4), a string to find the result in the benchmark output and the weight of the metric. The benchmark output files contain the data in the format of lines following the pattern `<substring><value>`, and by adding the substrings to the config we can get the desired metric by searching for the substring and reading the data between its end and the end of the line.

```
    "weights":{
        "Compiler":{
            "hpcc":[["StarSTREAM Triad
               ↪ Gb/s","StarSTREAM_Triad=",2],["StarRA
               ↪ GUPS","StarRandomAccess_GUPs=",3]],
            "hpcg":[["HPCG Gflops","Final Summary::HPCG result is
               ↪ VALID with a GFLOP/s rating of=",1]]
        },
        [...]
    },
```

Listing 5.3: Name of the values, the substring used to cut the value out of the output, and its weight for the package type Compiler and its respective benchmarks

The third part covers the data necessary to build the configuration files (see listing 2.7 for an example of a HPCG configuration) for the installation and run functionality of the original HPC-Benchmarks. The lines in these files follow a strict pattern of `<parameter>    [<description>]` (the square brackets are literal this time and don't denote a list), which is why I decided to use `[<description>]` as the keys and `<parameter>` as the values of these sections, since it made generating the configurations much easier. This approach worked for the sections "meta settings", "benchmark parameters" and "SLURM parameters", but not for the "dependency" section since it needs to be contructed from the packages to be tested and the packages currently in the stack. Instead, only the desired benchmark version can be added here 5.4.

```
    "config":{
        "meta settings":{
            "[iterations]":"2",
            "[aggregation: e.g. avg, max]":"avg"
        },
        "bm_version":{
            "hpl":"2.3",
            [...]
        },
        "benchmark parameters":{
            "hpl":{
                "[0]":"HPL.out",
                [...]
            },
            [...]
        },
        "SLURM parameters":{
            "[Partition]":"vl-parcio",
            [...]
        }
    }
```

Listing 5.4: Data for the construction of the HPC-Benchmark configuration files is located in this section

The next section covers the stack (listing 5.5). It has an entry for each of the three package types we are trying to find the best candidate for, and this is where they are added to the dictionary after the configuration has been loaded. The stack is used to generate the configuration files for HPC-Benchmarks to ensure that the best compiler and MPI implementation will be used if we want to test the blas packages for example. Additionally, it will also be printed to the log file as the last action of the program.

```
    "stack":{
        "Compiler":"",
        "MPI":"",
        "BLAS":""
    },
```

Listing 5.5: Section containing space for the components of the stack

The last section (listing 5.6) of the configuration covers the packages that are to be tested. These are grouped according to their types (compiler, MPI and BLAS). Each package is listed under a name that is used to identify it in the plots and tables the program creates (see section 6.4, and in the names of the generated configs for the installation and running of the benchmark. These packages consist of the spack elements name (of the package), version and flag (if the package isn't a compiler), which will be used to construct the respective specs. In the MPI example above for instance, "mpi1" represents the spec `mpich@4.1.2 netmod=ofi`.

```
    "packages":{
        "Compiler":{
            "cmp1":{
                "Name":"gcc",
                "Version":"13.2.0"
            },
            [...]
        },
        "MPI":{
            "mpi1":{
                "Name":"mpich",
                "Version":"4.1.2",
                "Flags":"netmod=ofi"
            },
            [...]
        },
        [...]
    }
```

Listing 5.6: List of packages to be tested, grouped by category

5.5.3. code

The main workflow of the stack optimization has been added directly to the pre-existing function `cl_args()`, which parses the arguments given to the program via the command line and executes the code block corresponding to the chosen flags. A shortened demonstration of this can be seen in listing 5.7. From there, several functions are called to do a variety of tasks. Due to this, the main workflow will be covered first, after which some of the more important functions will be discussed.

```
def cl_arg():
    [...]
    parser = argparse.ArgumentParser(formatter_class =
       ↪ RawTextHelpFormatter)
    parser.add_argument('-i','--install',nargs='+',type=str,help
       ↪ = ''+[...]
    [...]
    args= parser.parse_args()
    [..]
    #Install Benchmarks
    if args.install:
        [...]
    [...]
    if args.optimize:
        #begin optimisation
```

Listing 5.7: cl_args function of HPC-Benchmarks. If the `-o` flag is used together with an argument (i.e. the stack that is suppoed to be optimised) when running the program (e.g. `python3 sb.py -o stack1`, the optimisation will begin.

Main functionality

The general workflow of the optimisation is shown in the figure 5.2. It can be started with the command `python3 sb.py -o <stack>`, where `<stack>` is the configuration file (without the .json substring) to use. The program will set the optimisation up by creating the necessary folders, verify the existence of the configuration file etc. A batch script containing the command `python3 sb.py -e <stack>` is then generated using the function `generate_stack_script` and submitted with a job name of `<stack>_building` using a python subprocess. The original call of `python3 sb.py -o <stack>` terminates here; the rest of the optimisation is run inside the slurm job that calls `python3 sb.py -e <stack>`. This is done to ensure that the time-consuming optimisation is done inside a slurm job, which will continue to run even if the user logs out after initiating it. The stack is then loaded into the `STACK` variable from the appropriate .json file, at which point the stack optimisation begins.

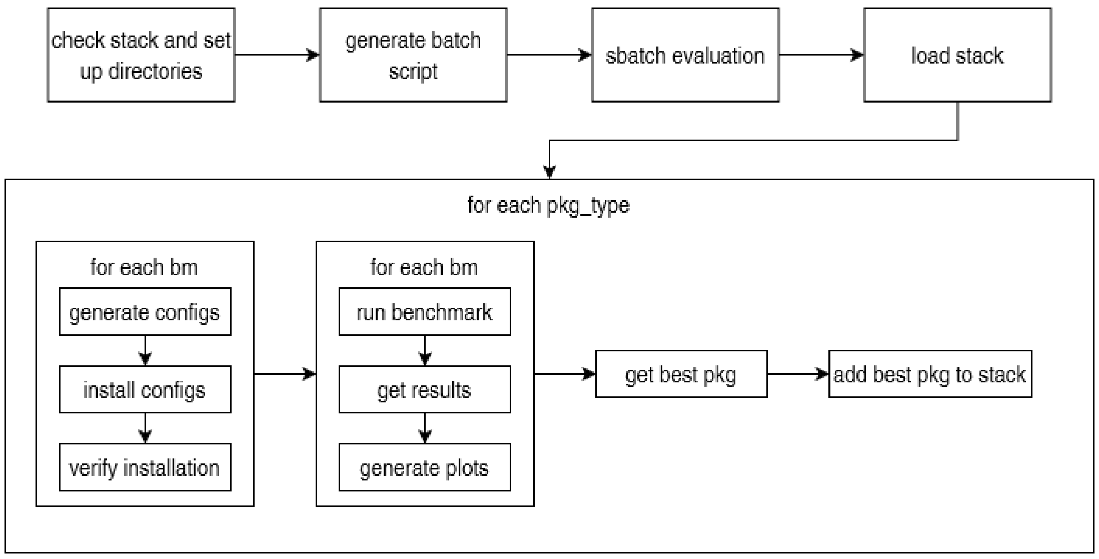


Figure 5.2.: General workflow of the stack optimization. `pkg_type` describes the categories of packages in the config (i.e. Compiler, MPI and BLAS), the `bm`s are the benchmarks that the program is supposed to install and run for the given `pkg_type`, and `pkg` refers to the individual packages of current `pkg_type`

The program iterates over every package type in `STACK["stack"]` to find the best package of the that type. First, the packages of this type are copied into `STACK["testing"]`, and two benchmark lists are created that include the benchmarks that needs to be installed and run, respectively. The reason for this split is OSU, since it is internally referred to as `osu_bw`, `osu_latency` etc when we want to run it, and just as `osu` in all other cases.

For every benchmark that needs to be installed, HPC-Benchmark configs are generated (based on the current elements of the stack and the package type that is currently tested), installed via `python3 sb.py -i <bm> [<config1,...>]`, and then filtered to exclude the candidates that failed to install. This is done to avoid unnecessary work; for example, if the first compiler failed to install HPCC, it can't be tested and therefore needs to be excluded, which would make the attempt to install HPCG with it pointless.

Once this is done, it is time to run the remaining configs for each benchmark through a subshell, using `python3 sb.py -r <bm> <config1,...>`. These results are added to `STACK["testing"][pkg][bm]` as a list of lists containing the performance metrics for the respective benchmark run, after which the average values are calculated and appended to the aforementioned entry. Once this has been done for every package, the results are plotted grouped together in a bar chart, with one group for every package and one bar for the result of every iteration and the average values. The last step is to find the best package out of the available ones, which will then be added to the stack. Afterwards, the next pkg_type will be tested.

`generate_stack_script(stack)`

`generate_stack_script` creates the batch script `optimization_script.sh` that will be submitted to slurm. The resulting job is responsible for the rest of the workflow that's part of the stack optimisation. The first part of the slurm parameters is indirectly taken from the file `/configs/config.txt`, which was part of the original HPC-Benchmarks and formed the basis of batch.sh (see listing 2.10). The second part of the parameters sets the job name and output locations. The last line of the script starts the actual optimisation functionality of HPC-Benchmarks-2

```
def generate_stack_script(stack):
    with open(LOC+"/optimization_script.sh","w") as script:
        batchtxt=write_slurm_params(cfg_profiles[0][0],3)
        batchtxt+='#SBATCH --job-name={}_building\n'.format(stack)
        batchtxt+='#SBATCH
           ↪ --output={}\n'.format(LOC+"/stacks/{}/{}.out"
           ↪ .format(stack,stack))
        batchtxt+='#SBATCH
           ↪ --error={}\n'.format(LOC+"/stacks/{}/{}.err"
           ↪ .format(stack,stack))
        batchtxt+="\n"
        batchtxt+="python3 {}/sb.py -e {}\n".format(LOC,stack)
        script.write(batchtxt)
    return LOC+"/optimization_script.sh"
```

Listing 5.8: generate_stack_script creates the slurm script used to run the optimisation

generate_configs(bm,package_type)

generate_configs(bm,package_type) (listing 5.9) generates the configurations for the benchmark bm for all packages in STACK["testing"], which are of type package_type. The first and last section are directly created from their respective entry in the stack configuration, while a benchmark specific function (generate_<bm>_config) is called to write the base dependencies and benchmark parameters sections, since these differ from benchmark to benchmark.

```
def generate_configs(bm,package_type):
    config_dir=LOC+"/configs/"+bm
    for spec in STACK["testing"]:
        cfg=""
        cfg+="----------meta settings----------\n"
        for key in STACK["config"]["meta settings"]:
            value=STACK["config"]["meta settings"][key]
            cfg+=generate_config_line(key,value)

        for part in ["dependencies","parameters"]:
            if part=="dependencies":
                cfg+="----------base dependencies----------\n"
            elif part=="parameters" and not bm=="osu":
                cfg+="----------benchmark parameters----------\n"

            [cfg+=generate_<bm>_config(package_type,spec,part)]

        cfg+="----------SLURM parameters----------\n"
        for line in STACK["config"]["SLURM parameters"]:
            cfg+=generate_config_line(line,STACK["config"]["SLURM
               ↪ parameters"][line])

        with open("{}/{}_cfg_{}.txt".format(config_dir,bm,spec),
           ↪ "w") as cfg_file:
            cfg_file.write(cfg)
```

Listing 5.9: generate_configs builds the configuration files for HPC-Benchmarks. The `for` ↪ `part in ["dependencies","parameters"]` loop had to be shortened here. In the actual code, the config generation function called depends on the current benchmark, e.g. `generate_hpl_config`, `generate_osu_config` etc.

`wait_for_job(JobID)`

`wait_for_job(JobID)` (listing 5.10) is used after the installation/execution of benchmark configurations has begun to ensure that these slurm jobs are finished before the next step of the optimisation begins. It takes the JobID of the respective job, checks its status every 5 seconds and thus stops the script from progressing, and returns once the job ID is no longer listed in the list of currently scheduled jobs.

```
def wait_for_job(JobID):
    while subprocess.run("squeue -h", stdout=subprocess.PIPE,
       ↪ stderr=subprocess.STDOUT,
       ↪ shell=True).stdout.decode().find(JobID)>-1:
        time.sleep(5)
```

Listing 5.10: wait_for_job periodically tests, whether the job is still in the list of scheduled jobs. Once this isn't the case anymore (because the job finished, failed etc.), the function terminates.

`remove_uninstalled_spec(unavailable_config)`

Once the benchmark installation finishes, `remove_uninstalled_spec` (listing 5.11 is called to remove the packages from `STACK["testing"]` that didn't result in installable configuration files. To do this, some of the internal functions of HPC-Benchmarks are used. The variable `cfg_profiles` keeps track of all profiles (i.e. configs) and information related to them (e.g. their spec, benchmark and slurm parameters); it is a list with one list (containing the respective profiles) for every benchmark as elements. The profile list that matches the current benchmark can be filtered to find all configurations without an installation path (meaning they are not installed). Any package in `STACK["testing"]` with a config that list is not installed and can therefor be removed/pop from `STACK["testing"]`.

```
#profiles=cfg_profiles[tag_id_switcher(bm)]
#unavailable_configs=list(filter(lambda p: p[0][2]=="no path
   ↪ found!",profiles))

def remove_uninstalled_spec(unavailable_config):
    generated_configs=list(STACK["testing"].keys())
    pkg=unavailable_config[0][0].split("_cfg_")[1].split(".txt")[0]
    if pkg in generated_configs:
        write_to_log("installation for config {}
           ↪ failed".format(pkg))
        STACK["testing"].pop(pkg,None)
```

Listing 5.11: remove_uninstalled_spec is used to remove the package with a configuration that couldn't be installed. The first two lines are only included to provide context for the unavailable_config variable.

analyze_results(res_dir,pkg_type,bm,pkg)

analyze_results(res_dir,pkg_type,bm,pkg) (see listing 5.12) is used to gather the data from the output files in directory <res_dir> of benchmark <bm> for package <pkg> of type <pkg> (e.g. the HPCG results of the MPI implementation mpi1) . This is done by creating a list of lines containing the desired data from STACK["weights"][pkg_type][bm], i.e. element[1] of each element/sublist in the list from the configuration file. This list is then iterated over for all of the output, and the value is cut from the output file using the function read_val_from_result(res,str), with res being the string contained in the output and str being the list element. This functions splits res at the string str, and splits the second string in the resulting list at a line break. Using this method, the value can be cut out of the file and returned as a float. anaylze_results returns a two dimensional list, consisting of a list with the cut values for each repeated benchmark execution.

```
def analyze_results(res_dir,pkg_type,bm,pkg):
    res_list=[]
    benchmark=bm
    lines=[]

    if (bm.find("osu")>-1):
        benchmark="osu"

    lines=[elem[1] for elem in STACK["weights"][pkg_type][bm]]

    for i in range(int(STACK["config"]["meta
      ↪ settings"]["[iterations]"])):
        res_list.append([])
        with open(res_dir+"/{}#{}_cfg_{}.out".format(i+1,
           ↪ benchmark,pkg),"r") as file:
            res=file.read()
            for str in lines:
                res_list[i].append(read_val_from_result(res,str))

return res_list

def read_val_from_result(file,s):
    return float(file.split(s)[1].split("\n")[0])
```

Listing 5.12: Shortened Version of the function analyze_results and read_val_from_result. The strings necessary to cut the desired values out of the output files are taken from the config and iterated over for every iteration of the benchmark run and their respective output files.

get_averages(pkg,bm)

get_averages(pkg,bm) calculates the average values for each value in STACK["testing"][pkg][bm] over all benchmarking run iterations and returns them as a list (see listing 5.13).

```
def get_averages(pkg,bm):
    cur_list=STACK["testing"][pkg][bm]
    res=list(map(lambda i:functools.reduce(lambda a,b:a+b[i],
       ↪ cur_list,0)/len(cur_list),[i for i in
       ↪ range(len(cur_list[0]))]))
    return res
```

Listing 5.13: The function get_averages calculates the average values of the results contained in STACK["testing"][pkg][bm] and returns these. They are appended to STACK["testing"][pkg][bm] in the main workflow

best_pkg(pkg_type)

best_pkg(pkg_type) is used to determine the best available package of type <pkg_type>. This is done by first creating a weight list, consisting of a list of weights for every benchmark that used to test <pkg_type>, and by calculating the sum of all weights. Once this is done, the function iterates over all available packages in STACK["testing"]; the first package becomes the best package, and every following package is tested against the current package. This test is done by using the score function defined in section 5.2, which is implemented as follows.

The lists of average values of the results for each package and benchmark are stored in STACK["testing"][pkg][bm][iterations]. To calculate the score, we can iterate over all of the respective benchmarks and values in the lists for the two packages we want to compare, calculate the sums of their weighted ratios and divide it by the sum of weights. If this value is greater than 1, the other package becomes the best package and the loop continues until all packages have been tested. Afterwards, the name of the best package is returned.

```
def best_pkg(pkg_type):
    #weights=[bm_weight_list for every benchmark used to tesk
       ↪ pkg_type]
    #total_weight=sum of of all absolute weights
    for pkg in STACK["testing"]:
        if best_pkg=={}:
            best_pkg=pkg
        else:
            score=0
            bms=[benchmark for benchmark in
               ↪ STACK["bms_for_package"][pkg_type]]
            for index,bm in enumerate(bms):
                for i in range(len(weights[index])):
                    val_best_pkg=STACK["testing"][best_pkg][bm]
                       ↪ [iterations][i]
                    val_pkg=STACK["testing"][pkg][bm]
                       ↪ [iterations][i]
                    if weights[index][i]<0:
                        ratio=val_best_pkg/val_pkg
                    else:
                        ratio=val_pkg/val_best_pkg
                    w_ratio=ratio*abs(weights[index][i])
                    score+=w_ratio
            score=score/total_weight
            if score>1:
                best_pkg=pkg
    return best_pkg
```

Listing 5.14: Condensed Version of the function best_pkg

graph(l,col,xlabel,ylabel,title,figname)

graph(l,col,xlabel,ylabel,title,figname) (see 5.15) creates a figure containing the results of one execution of the command python3 sb.py -r <bm> <config1,...> for one of the extracted values, e.g. the Gflops achieved for HPCG. <l> is the list of containing data that should be grouped together in a bar chart and follows the pattern of [<pkg_name>,<value1>,<value2>,..<average_value>], <col> refers to the individual bars [<bm> run <i>,..,average result], etc. These values are then plotted and saved as <figname>. Figures generated by this function can be seen in section 6.4.

```
def graph(l,col,xlabel,ylabel,title,figname):
    matplotlib.rcParams.update({'font.size': 22})
    df = pd.DataFrame(l,columns=col)

    ax=df.plot(x=col[0],kind='bar',stacked=False,title=title,rot=0,
       ↪ figsize=(20, 12))
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend(bbox_to_anchor=(1,1), loc="upper
       ↪ left",fontsize='small')

    ax.figure.savefig(figname,bbox_inches='tight')
```

Listing 5.15: graph is used to create the plots

Chapter 6. Evaluation

6.1. Final package selection

Installing and running every possible benchmark configuration based on the preliminary package selection in 4.3 would be incredibly time consuming. Due to the fact that the job duration on the nodes running the benchmarks is limited to 6 hours, a large number of benchmark configurations to be tested would require shorter individual benchmark runs (and thus smaller problem sizes) and fewer iterations per configuration.

To address this issue, the packages were first tested using the regular HPC-Benchmarks features to see if they worked at all. Unfortunately, a large number of packages did not produce results, either because the benchmark installations failed or because the benchmarks failed to run correctly. This ran counter to the expectation that spack and its concretisation could reliably produce working package installations that didn't require fine-tuning through the use of variants and trial and error. As a result, it highlighted the importance of making sure that packages are set up correctly for the system they are being used on. Because of this, the decision was made to focus on a smaller number of packages that did work and increase their problem size/number of iterations to get more reliable results.

`spack install` also gave different results for the same packages, depending on a number of factors, some of which are unknown to me. For example, in some cases benchmark installations failed when run in a batch script on compute nodes, but worked when the some command was run on the login node.

For the compilers, neither intel-oneapi-compilers nor llvm were able to compile all the benchmarks, but GCC and AOCC did.

Of all the MPI implementations tested, only mpich worked on the cluster on which the tests were run. None of the others worked correctly out of the box with HPC-Benchmarks' script-building pipeline. This was a problem because testing only a single implementation makes it difficult to demonstrate stack optimisation. To remedy this, different variants of mpich were used instead, each with a different networking method. ofi and tcp ran successfully, mxm led to an installation error because the required mxm package doesn't seem to be fully implemented in spack, and ucx ran without terminating. The ofi and tcp variants are tested accordingly.

The BLAS implementations caused the least problems. All but ATLAS seemed to work well. To continue the pattern of a reduced number of packages, the selection is limited to the three implementations OpenBLAS, AMDBLIS and flexiblas.

Name/Variant	Version	Package Type	Name in config
HPCG	3.1	benchmark	
HPCC	1.5.0	benchmark	
OSU	7.2	benchmark	
GCC	13.2.0	compiler	cmp1
AOCC	4.0.0	compiler	cmp2
mpich netmod=ofi	4.1.2	MPI	mpi1
mpich netmod=tcp	4.1.2	MPI	mpi2
openblas	0.3.23	BLAS	blas1
flexiblas	3.3.0	BLAS	blas2
amdblis	4.0	BLAS	blas3

Table 6.1.: Table containing the software that will be used for the testing, with one column each for the package name (or variant in the case of mpich), the version, and the package category. Included for the packages that will be tested is also the name they have recieved in the configuration file.

The final package selection that resulted from this process of elimination can be seen in table 6.1. The software versions are the ones that `spack info` lists as preferred, which is usually the latest. For the individual packages that are going to be tested, their name in the configuration file will also be included and used as a shorthand, as the specs may be too long to write out each time the packages are referenced. The names are derived from the package type ("cmp" for compilers, "mpi" and "blas" are self-explanatory) and a number to distinguish between implementations.

6.2. Parameters

To run the benchmarks, two nodes of the ovgu cluster have been used, both running CentOS 8. Each of these contains one AMD Epyc 7443 processor with 2,85 GHz for all of their 24 cores. The RAM size is 128 GB, and they are connected via 100Gb Ethernet. Another one of these nodes is used to run the control script that submits the benchmark runs during the course of the optimization.

During the tests, slurm version 20.11.9 and spack 0.21.0.dev0 have been used.

To utilise the listed resources, the individual benchmarking scripts request 2 nodes, 48 tasks and 24 tasks per node. Since the job duration is limited to a maximum of 6 hours on the chosen nodes, the problem sizes will have to be adjusted accordingly. The recommended problem size for HPCG should fill at least 25% of the main memory [HPCG, 2022] , which is $2 * 128GB = 256GB$ in this case. After running some tests to find out the memory usage of different problem sizes (as can be seen in table 6.2), "144 120 120" will be chosen since it's the smallest problem size that's still large enough to use the recommended amount of memory. The parameter selection for HPL has been driven by practical concerns over the run time of the benchmark. The 48 tasks have dictate the size of the process grid, which results in

Problem size	Main memory used
144 144 144	102.499
144 144 120	85.4193
144 120 120	71.1857
120 120 120	59.324

Table 6.2.: Experiments with different problem sizes to learn about the memory usage of HPCG.

Memory	Problem size	Run time
8gb	40704	00:11:51
16gb	57600	00:13:33
32gb	81792	00:17:34
64gb	115584	00:26:25

Table 6.3.: Experiments with different problem sizes for HPL. The first column is the amount of memory entered into [Advanced Clustering Technologies, 2022], the second column contains the calculated problem size, and the third column contains the time needed to run the benchmark with the respective problem size.

the values $P = 6$ and $Q = 8$ based on the results of a configuration generating tool for HPL [Advanced Clustering Technologies, 2022] . Using the same website, configurations for several different amounts of main memory have been generated and tested to find one that achieves a balanced run time of between 15 and 20 minutes, the results of which can be seen in table 6.3. N=81792 was the only one that finished in this time window, which is why it will be chosen moving forward.

To summarize the parameters in a more readable format, they will be listed in the table 6.4 for HPCG and table 6.5 for HPL; HPCC uses the exact same parameters as HPL, with the additional ones (e.g. additional problem sizes for PTRANS) set to 0 and thus ignored.

6.3. Running the program

HPC-Benchmarks-2 is available at github [Schröder, Fabian, 2023]. To run the program and use all of its functionality it is necessary to have a few python modules installed (as well as spack and slurm of course). These are `tabulate`, which is necessary to generate the result and comparison tables that will be printed in the log file, and both `pandas` and `matplotlib` to plot the results.

Once the repository has been cloned, the configurations in the `stack` directory could be modified to include the desired packages and choose appropriate problem sizes and resources. Since the config template `stack1.json` has been written for use in this thesis, this step can be skipped.

Problem size (n_x,n_y,n_z)	time target
144 120 120	300

Table 6.4.: Parameters for HPCG; the time target is the minimum time that the benchmark should run

Problem size N	blocking factor NB	process grid P*Q
81792	192	6*8

Table 6.5.: Parameters used for HPL and HPCG

The optimization can be started using the by launching `sb.py` with the `-o` flag and the desired configuration without the .json suffix. Assuming that we are in the HPC-Benchmarks-2 directory and want to use the preexisting configuration, the command is `python3 sb.py -o stack1`. As part of HPC-Benchmarks' standard functions, a spack installation will be searched if none is listed in the `HPC-Benchmarks-2/configs/config.txt` file. Since the repository is freshly cloned, this is necessary here. This step can be seen in listing 6.1

```
[fschroed@ants HPC-Benchmarks-2]$ python3 sb.py -o stack1

<warning> there's no specified path for spack!

<info>    these possibly valid options were detected:
          [1.] /home/fschroed/forked-bm-tool/spack/bin/spack
          [2.] /home/fschroed/spackcopy/spack/bin/spack
          search mode: ~ [local]

- - - how to proceed? - - -
(1) install spack to a specific location
(2) terminate
(3) proceed with an alternative (will be saved to config!)

<input> 3

- - - which one? - - -
1 <=> first one & up to 2 options

<input> 1
```

Listing 6.1: Starting HPC-Benchmarks-2 for the first time

Once this has been done, the optimisation functionality can begin (listing 6.2. If the stack exists and no results in the form of a directory exist for this configuration, the directory will be created, its path printed to the console and the optimization script will be written and submitted to slurm with the name `<stack>\_building`, which means `stack1\_building` in this case. Its job ID will be returned and can be used to keep track of the optimization; once it is done, the optimization has finished. The script itself is used to execute the command `python3 sb.py -e stack1`.

```
evaluating...
creating directory /home/fschroed/HPC-Benchmarks-2/stacks/stack1
Submitted batch job 768688
```

Listing 6.2: Once a path to the local spack installation has been entered, the output directory for the optimisation can be created and the optimisation script submitted

While active, the program will create more jobs to install and run the benchmarks. This can be seen in the following snapshots of the current job queue (listing 6.3):

```
[fschroed@ants HPC-Benchmarks-2]$ squeue -u fschroed
             JOBID PARTITION     NAME     USER ST       TIME
                ↪ NODES NODELIST(REASON)
            768688    parcio stack1_b fschroed  R       0:02
               ↪ 1 ant13
[fschroed@ants HPC-Benchmarks-2]$ squeue -u fschroed
             JOBID PARTITION     NAME     USER ST       TIME
                ↪ NODES NODELIST(REASON)
            768689    parcio install. fschroed  R       0:01
               ↪ 1 ant13
            768688    parcio stack1_b fschroed  R       0:27
               ↪ 1 ant13
[fschroed@ants HPC-Benchmarks-2]$ squeue -u fschroed
             JOBID PARTITION     NAME     USER ST       TIME
                ↪ NODES NODELIST(REASON)
            768688    parcio stack1_b fschroed  R       1:47
               ↪ 1 ant13
[fschroed@ants HPC-Benchmarks-2]$ squeue -u fschroed
             JOBID PARTITION     NAME     USER ST       TIME
                ↪ NODES NODELIST(REASON)
            768698    parcio hpcg_plo fschroed PD       0:00
               ↪ 1 (Dependency)
            768695    parcio cfg_to_d fschroed PD       0:00
               ↪ 1 (Dependency)
            768691    parcio hpcg_run fschroed  R       0:18
               ↪ 1 ant13
            768688    parcio stack1_b fschroed  R       2:08
               ↪ 1 ant13
            768697 vl-parcio 2#hpcg_c fschroed PD       0:00
               ↪ 2 (Dependency)
            768696 vl-parcio 1#hpcg_c fschroed PD       0:00
               ↪ 2 (Dependency)
            768694 vl-parcio 2#hpcg_c fschroed PD       0:00
               ↪ 2 (Dependency)
            768693 vl-parcio 1#hpcg_c fschroed  R       0:15
               ↪ 2 ant[14-15]
```

Listing 6.3: Checking the status of the optimisation job using squeue at different times. The first configs are installed, afterwards the benchmarking process will begin. All of the jobs in the last call of squeue have been submitted by the batch.sh script that was generated by HPC-Benchmarks (see listing 2.10)

Additionally, the log file in the folder of the stack contains more information about the current status of the optimisation, intermediate results etc. Sadly, the output can't fit in this document due to its size without reducing its size to make it too hard to read, so a list of the most important aspects will have to suffice. The following information is amongst the output generated for every iteration/package type of the optimisation:

- The benchmarks that will installed and run
- Information about the benchmark installation and run jobs (jobID, start time, duration)
- The list of current configs and those that were removed due to installation problems
- One table containing the gathered data of all benchmark iterations and their average value for every tested package and benchmark
- One table for every comparison of the package selection together with the score achieved by the package (see the tables in section 6.4; they are LaTeX Versions of the ones in the log file)
- The selected package of the given type

The last part of the output is a list of all selected packages, which form the stack that the optimisation process calculated to be the best one. In this test, the result is the one shown in listing 6.4

```
The final stack consists of the following packages:
Compiler: gcc@13.2.0
MPI: mpich@4.1.2 netmod=tcp
BLAS: openblas@0.3.23
```

Listing 6.4: The last lines in the log file after the optimisation successfully terminates contain the final stack

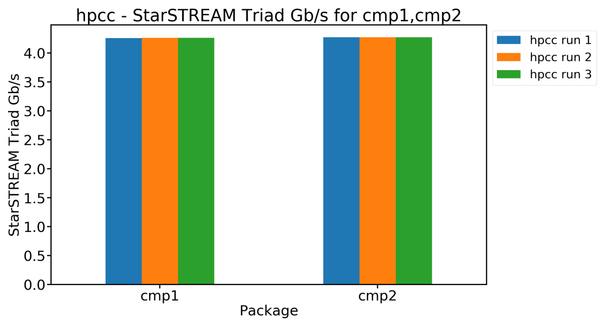
6.4. Results

6.4.1. Compiler

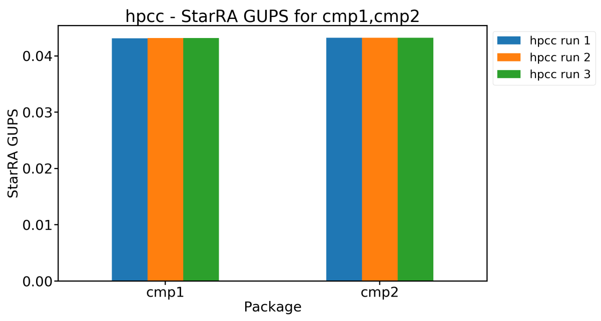
data	StarSTREAM Triad Gb/s	StarRA GUPS	HPCG Gflops
cmp2	4.27288	0.0432303	3.39215
cmp1	4.26108	0.0431592	5.90837
ratio	1.00277	1.00165	0.574125
weights	2	3	1
weighted ratio	2.00554	3.00495	0.574125

Table 6.6.: Comparison of the compilers cmp2 (aocc@4.0.0) and cmp1 (gcc@13.2.0). cmp2 achieved a score of 0.9307682502996789, making it worse than cmp1

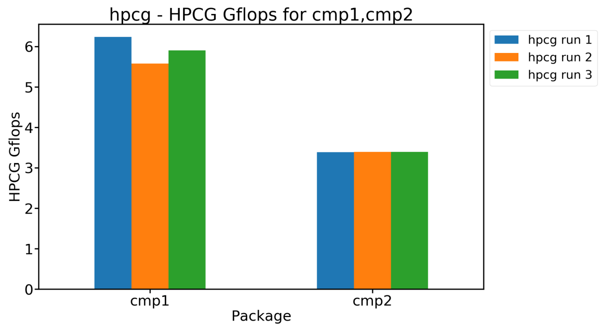
Both compilers, gcc@13.2.0 (cmp1) and aocc@4.0.0 (cmp2), achieved nearly identical memory bandwidths for the TRIAD operation of StarSTREAM, with an average of 4.26 Gb/s and 4.27 Gb/s respectively, as can be seen in figure 6.1a. In addition, the individual benchmark runs produced consistent results without too much variation. The same is true for the GUPS measured with the StarRandomAccess benchmark, where both compilers achieved around 0.043 as shown in figure 6.1b. The results for HPCG, however, are quite different, with AOCC only achieving around 57% of GCC's performance and less consistent results overall (diagram 6.1c).



(a) Gb/s of memory bandwidth achieved in the Triad operation of StarSTREAM



(b) GUPS achieved in the StarRandomAccess benchmark



(c) Gflops achieved in HPCG

Figure 6.1.: Results of the Compilers gcc@13.2.0 (cmp1) and aocc@4.0.0 (cmp2)

This quite surprising; despite the fact that AOCC is supposed to be optimized for AMD hardware, it failed to yield real advantages in its performance in the factors that I considered to be most impactful on the performance, and achieved a comparatively bad performance in the one that was judged to be least important. Due to this, the best compiler was effectively solely chosen based on the metric with the lowest weight, and it was the one that wasn't specifically designed to work with the available hardware resources. The results from the tests of the Fujitsu compiler on Fujitsu hardware [Poenaru et al., 2021] lead me to the expectation, that AOCC would probably better and that the memory bandwidths would show significant differences. The result is consistent though with what one would expect judging purely by the plots, so the weights didn't lead to any surprises in that regard.

Based on these results, the compiler might have less impact on the memory bandwidth and memory access than assumed, which means the selection of metrics and their respective weight didn't match the actual performance differences. It's hard to say how much of an effect the limitation of the testing had on that; it's certainly possible that the inclusion of more compilers or the use of different cluster setups might to lead to results that fit the initial assumptions better.

Also, it stands to reason that GCC produces better optimised code than AOCC, which would explain why the HPCG results is so different despite the other results being nearly identical.

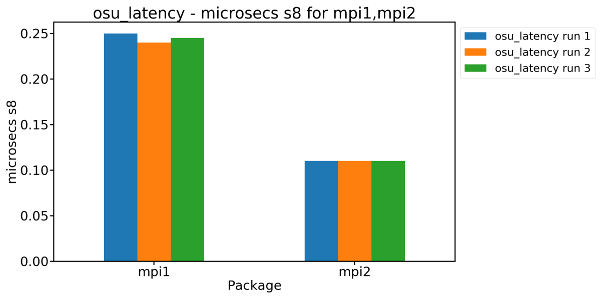
6.4.2. MPI

The testing of the mpich@4.1.2 variants with the netmods ofi (mpi1) and tcp (mpi2) involved the benchmarks OSU bandwidth, OSU latency and HPCG. mpi2 showed consistent advantages in

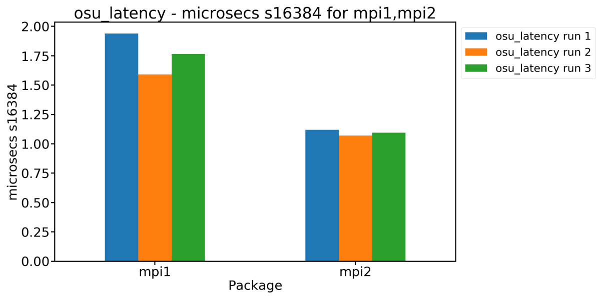
data	μs 1	μs 2	μs 3	Gb/s 1	Gb/s 2	Gb/s 3	HPCG Gflops
mpi2	0.11	1.095	527.785	83.925	15245.5	8624.33	6.58365
mpi1	0.245	1.765	631.04	42.19	15262.9	10944.7	5.91896
ratio	2.22727	1.61187	1.19564	1.98922	0.998861	0.787989	1.1123
weights	-10	-4	-2	10	4	2	2
weighted ratio	22.2727	6.44749	2.39128	19.8922	3.99545	1.57598	2.2246

Table 6.7.: Comparison of the MPI implementations mpich@4.1.2 netmod=ofi (mpi1) and mpich@4.1.2 netmod=tcp (mpi2). The score for mpi2 is 1.7294019723296077, making it the better choice. Due a lack of space, the table headers had to be significantly shortened. μs denotes the latency in microseconds, Gb/s denotes the achieved bandwidth, and the number following each instance of "Gb/s" and μs denotes the respective message size (1->message size 8, 2->message size 16384 ,3->message size 4194304

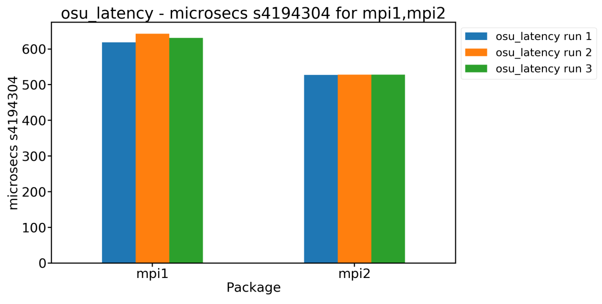
regards to the achieved latency, with 0.11 microseconds compared to mpi2's 0.245 microseconds for a message size of 8 (figure 6.2a), around 1.1 μs to mpi2's 1.765 μs for a message size of 16384 (figure 6.2b), and roughly 528 μs for the message size 4194304 (figure 6.2c). In all of these cases, mpi2 performed better, although the relative improvement drops from a factor of about 2.2 to about 1.2 from the smallest to the largest message size.



(a) Latency achieved in the OSU benchmark for the message size 8



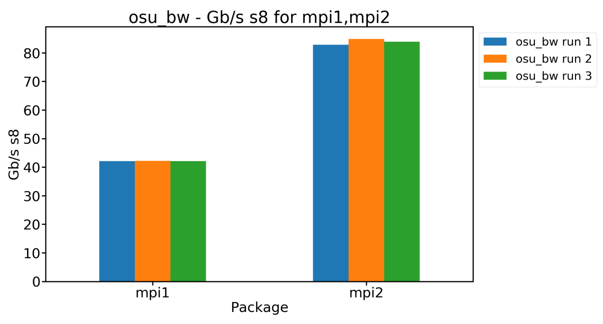
(b) Latency achieved OSU for the message size 16384



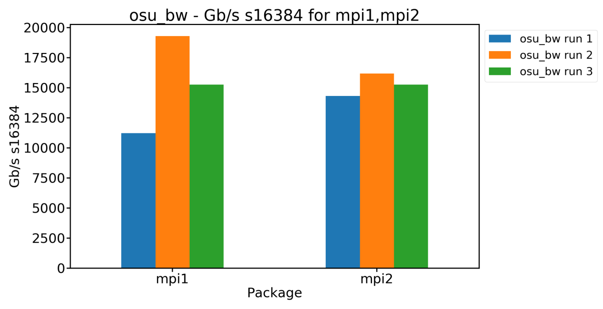
(c) Latency achieved in the OSU benchmark for the message size 4194304

Figure 6.2.: OSU Latency results of the MPI implementations mpich@4.1.2 netmod=ofi (mpi1) and mpich@4.1.2 netmod=tcp (mpi2)

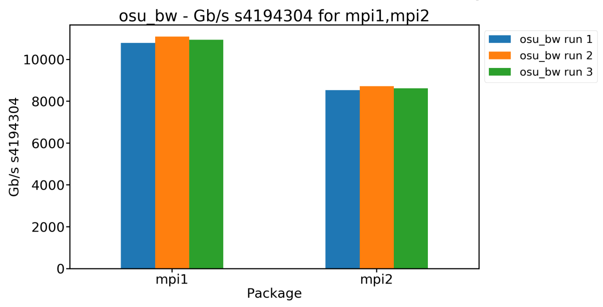
The bandwidth results show a similar pattern, with mpi2 starting of with approximately twice the bandwidth at 83.925 Gb/s compared to mpi1's 42.19 Gb/s (figure 6.3a). In this case however, the performance of the tcp variant dips below that of the ofi one, with nearly identical results for messages of the size 16384 at 15245.5 Gb/s and 15262.9 Gb/s respectively (figure 6.3b). Judging



(a) Bandwidths achieved in the OSU benchmark for the message size 8



(b) Bandwidths achieved in the OSU benchmark for the message size 16384



(c) Bandwidths achieved in the OSU benchmark for the message size 4194304

Figure 6.3.: OSU Bandwidths results of the MPI implementations mpich@4.1.2 netmod=ofi (mpi1) and mpich@4.1.2 netmod=tcp (mpi2)

from figure 6.3b, the result shows quite some variance, especially for mpi1, possibly due to network load caused by other users on the cluster. Repeating these measurements might yield a more accurate result. For the largest message size, mpi2 only achieved around 79% of mpi1's bandwidth, with 8624.33 Gb/s and 10944.7 Gb/s respectively (figure 6.3c).

The HPCG results show that the tcp variant achieved about 11% more Gflops than the ofi variant, with about 6,584 for the former and 5,919 for the latter (figure 6.4). This makes sense, as mpi2 was largely superior in the other metrics, especially for the smaller, more common message sizes. Overall, mpi2 achieved a score of around 1.73 compared to mpi1, making it the clearly better package (table 6.7).

The overall selection of metrics and weights appears to work better than the one used for the

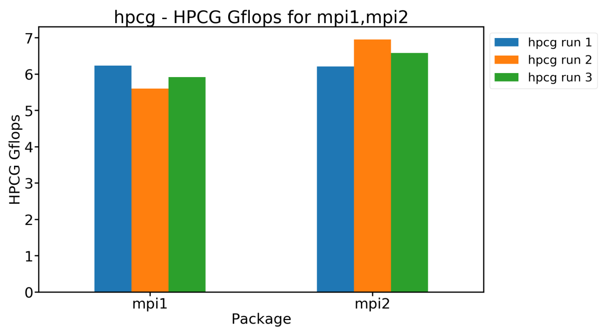
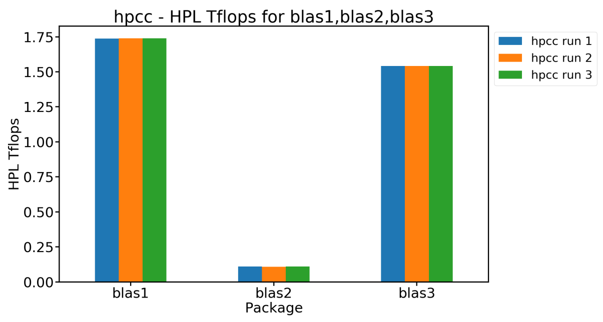


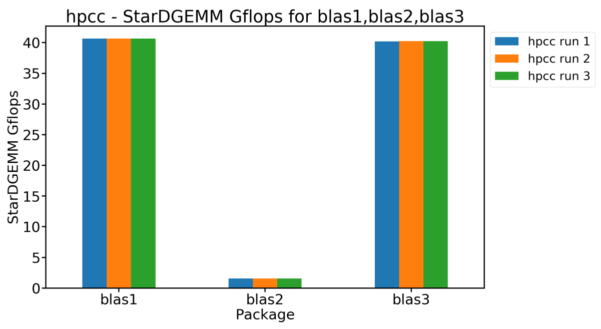
Figure 6.4.: HPCG results in Gflops of the MPI implementations mpich@4.1.2 netmod=ofi (mpi1) and mpich@4.1.2 netmod=tcp (mpi2)

compiler; the two mpich variants show clear differences in the perfomance data that was used to pick the better package, and the importance of these differences seems to be reasonably well quantified with their respective weights. These results paint a coherent picture that falls in line with the assumption, that smaller message sizes are more important for the overall performance than large ones.

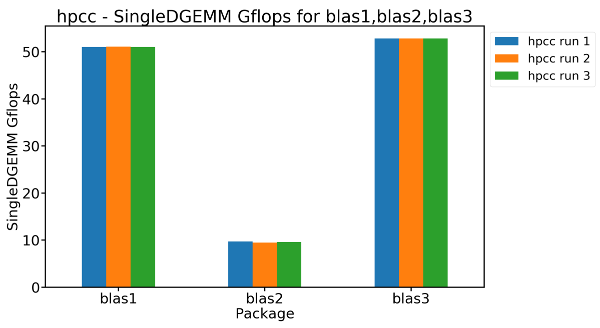
6.4.3. BLAS



(a) Gflops achieved in the HPL Benchmark of HPC-Challenge



(b) Gflops achieved in the StarDGEMM Benchmark of HPC-Challenge



(c) Gflops achieved in the SingleDGEMM Benchmark of HPC-Challenge

Figure 6.5.: Results of the BLAS implementations openblas@0.3.23 (blas1), flexiblas@3.3.0 (blas2) and amdblis@4.0 (blas3)

openblas@0.3.23 (blas1) and amdblis@4.0 (blas3) achieved fairly comparable results in StarDGEMM with around 40.68 and 40.23 Gflops respectively (figure 6.5b), the same is true for SingleDGEMM with 51.06 and 52.84 Gflops (figure 6.5c). The difference between the results for these two benchmarks and packages is between about 1% and 3.5%, so quite small. The biggest difference is in HPL Tflops, with blas3 achieving only 1.54 compared to blas1's 1.74 Tflops (figure 6.5a). In this case, blas3 was about 11% worse. The final score for blas3 compared to blas1 is around 0.99, making blas1 the better package by a very small margin.

However, both openblas@0.3.23 and amdblis@4.0 are far superior to flexiblas@3.3.0 (blas2). Comparing blas2 and blas1, blas2 achieves only 6% of blas1's HPL Tflops, 19% of its SingleDGEMM Gflops and 4% of its StarDGEMM Gflops (table 6.8), making it by far the worst BLAS implementation tested with a score of 0.09 relative to blas1. While the comparison between blas1 and blas3 might have been decided by random variation due to the small difference, this can't be the case here.

data	HPL Tflops	StarDGEMM Gflops	SingleDGEMM Gflops
blas2	0.109351	1.57581	9.5862
blas1	1.73834	40.677	51.0628
ratio	0.0629057	0.0387396	0.187733
weights	1	3	2
weighted ratio	0.0629057	0.116219	0.375467

Table 6.8.: Comparison of the BLAS implementations blas2 (flexiblas@3.3.0) and blas1 (openblas@0.3.23). The final score is 0.092431911629186

data	HPL Tflops	StarDGEMM Gflops	SingleDGEMM Gflops
blas3	1.5416	40.2314	52.8418
blas1	1.73834	40.677	51.0628
ratio	0.886823	0.989045	1.03484
weights	1	3	2
weighted ratio	0.886823	2.96714	2.06968

Table 6.9.: Comparison of the BLAS implementations blas3 (amdblis@4.0) and blas1 (openblas@0.3.23). blas3 reached a score of 0.9872726978314716.

The low performance of blas2 is obvious in the plots, as is the slight superiority of blas1 compared to blas3.

The selection of metrics and weights appears once again be suboptimal. While they do clearly show and emphasize, that flexiblas@3.3.0 is the worst performing BLAS implementation tested, they didn't work too well for the other two. The Gflops achieved in the StarDGEMM and SingleDGEMM benchmarks were assumed to be the more important metrics, but the performance differences between blas3 and blas1 was only around 3% for SingleDGEMM, and 1% for StarDGEMM. Despite the fact that is was assumed to be least important, the deciding metric was once again the Tflops achieved in the HPL benchmark, as was the case for the selection of the compilers.

6.5. Discussion

Overall, the main task of finding and implementing a way to find well performing software stacks on a cluster has been achieved, though the results are mixed. On the one hand, the general workflow of the stack optimisation works as intended, and the configurability of the analysis is a nice feature. The results are presented in an appropriate manner, with the log file containing tables of the benchmark results and the comparison of the individual packages, and the graphs providing a well-labelled and understandable representation of the performance of the packages in the selected metrics. The ability to run most of the program as a background slurm job is another good and user-friendly feature, suitable for the long duration that all installations and benchmark runs can take.

Package selection, on the other hand, proved to be a much bigger problem than expected, and the number of packages that failed to produce usable results was staggering. This is obviously annoying in the case of compilers, but the problems with MPI were even worse. This type of package is extremely important for high performance computing and the fact that only one worked is worrying. In addition, the fact that one of the variants ran but didn't exit introduced a new type of bug that might be difficult to fix, unless the runtime of each benchmark run is capped via slurm, in which case variants that work extremely slowly might also inadvertently exit. Uninstallable specs could be filtered out, but if mpirun doesn't terminate, then the rest of the job scripts can't be executed either, meaning that HPC-Benchmarks can't copy the output to the correct directory, and so those results can't be analysed. A time limit on the job could have been used to kill it after a while so that the next one in the dependency chain could start, but this still wouldn't have solved the output location problem.

The assumption that spack and its concretiser could reliably produce working software configurations on the cluster without further adjustment was incorrect in this case, which had a negative impact on the result. In addition, the HPC-Benchmarks-2 code was written under the assumption that the original HPC-Benchmarks in combination with spack could actually run the benchmarks with different MPI implementations, but this simply wasn't the case. Even openmpi, which is the standard MPI implementation included in all existing configurations of the tool, didn't work correctly with the HPC-Benchmarks scripting pipeline, even though it was developed for use on the same cluster as the one used to run the program in this thesis.

The pre-selection and seperate testing of the packages was a useful step in this thesis, but overall probably unnecessary for other users, as long as they keep the previously mentioned MPI problem in mind. This is due to the fact, that packages that failed to result in installable configurations were automatically detected and removed from consideration as part of the main workflow of HPC-Benchmarks-2. The process can be time consuming though, since the installation still needs to begin and fail first.

The implemented comparison and package evaluation was also mixed, although it's difficult to imagine much better results given the very limited amount of information on which the optimisation is based. Without any reference results or ways to measure the metrics beyond a handful of benchmarks, there's not much that can be done to find the best package beyond trying to judge them based on their relative results. The weights chosen didn't always emphasise the metrics where the packages really differed, which led to the least emphasised metric, HPL Tflops, having an outsized influence on the result of the compiler and blasters selection; this didn't occure for the MPI implementations. The final scores ended up being quite similar for the cmp1/cmp2 and blas3,blas1 pairs, with a difference of about 7%-8%. Without a way of estimating the error or variance in the chosen approach, it is difficult to say whether the choices were the right ones or not, which makes the overall reliability of these results questionable. Further tests, using more packages and different clusters, might help answer that question.

The software stack that appears to be the most promising on for the cluster consists of the compiler gcc@13 .2.0, the MPI implementation mpich@4.1.2 netmod = tcp, and the BLAS implementation openblas@0 .3.2. This matches what one might expect just by looking at the (graphed) results.

Chapter 7.

Conclusion and Future Work

7.1. Conclusion

This paper presents HPC-Benchmarks-2, a tool for building an efficient software stack consisting of a compiler, an implementation of MPI and an implementation of BLAS on a cluster based on data collected by running selected benchmarks with spack. It is based on and is an extension of HPC-Benchmarks [Lafontaine and Cirkov, 2022].

The optimisation was based on an algorithm that attempts to build the stack step by step by selecting one package from the configuration for each of the three types. A number of performance metrics were selected and weighted to quantify their perceived importance for each package type. To compare two packages, a ratio of each metric is calculated as an estimate of the speedup of one package over the other, and multiplied by its respective weight. If the sum of these weighted ratios is greater than the total weight, the package is considered to have better performance than the other.

On the setup used, the compiler gcc@13.2.0 narrowly beat aocc@4.0.0. mpich@4.1.2 netmod=tcp appeared to be superior to mpich@4.1.2 netmod=tcp, and of the three tested BLAS implementations openblas@0.3.23, flexiblas@3.3.0 and amdblis@4.0, openblas performed slightly better than amdblis, with flexiblas trailing far behind both.

Considering the scope of the thesis and the foundation on which it is built, the program seems to work reasonably well. The configurability via the json files is both functional and flexible, and starting/running the optimisation as a slurm job and tracking it via the output in the log file is a suitable and user-friendly approach given the duration of the whole process.

However, the packages have caused far greater problems than anticipated. The original hope was that one could use spack to browse the available packages, add them to the configuration and run the tests without having to manually configure the packages to work on the system, but this wasn't the case and led to a somewhat lengthy pre-selection process. As long as the MPI implementation works with the scripts generated by HPC-Benchmarks, this pre-selection shouldn't be necessary though, assuming that the failure to test some the packages is acceptable.

The performance analysis aspect worked largely as intended, in that the process reliably produced understandable results for selecting one of each package type for the stack based on a tailored selection of metrics. The weights used in this work lead to selections that are in line with what would be expected if one were to look at the graphed results, but at least for the compiler and BLAS they didn't emphasise the most significant differences that ended up being the deciding factors; the weights for MPI were better chosen in this respect.

7.2. Future work

At the current state of the program, several improvements could be made that would result in more accurate and less error-prone results.

Extending the scripting process of the original HPC-Benchmarks to differentiate between the MPI implementations, and tailoring the installation and execution of the benchmarks to them, may offer a way to reduce some of the compatibility problems encountered in the course of this thesis.

As discussed earlier, the choice of metrics and their respective weights was somewhat problematic. It's not clear if they were chosen incorrectly, or if the relative lack of performance differences for some of the metrics was just coincidental to this setup and the low number of tested implementations and not necessarily representative of other clusters and packages. Further testing on different hardware and using more packages may be required to resolve this issue. The same goes for the weights; experimenting to find the optimal ones is also recommended, as well as finding out how well the weights work on other setups.

At present, the program is written to assume that all individual benchmark runs will complete without error. While this worked well enough for this thesis, extending the code to deal with the possibility that some of the runs might fail to produce data for whatever reason would be a good way to avoid possible runtime problems.

Since the results of each benchmark run are stored in the folders, it might also be a good idea to offer the option of continuing a previous optimisation attempt based on the existing data. This would avoid the need to run the program in one go, which could take hours, and could act as a failsafe in case of an error; for example, if a compiler has been selected and the program crashes during the MPI selection, one could start a new optimisation attempt and use the existing data to select a compiler instead of starting from scratch.

Originally, the software stack was supposed to include a few more libraries (FFTW and LAPACK), which didn't end up being part of the tests due to a lack of working benchmarks to test them. Modifying the program to test other packages, such as those mentioned above, is an obvious step that could be taken to extend the selected software stack. This will involve finding other benchmarks and integrating them into the HPC-Benchmarks framework. cbench is an obvious candidate for this, if it can be made to work properly, and it might be possible to create new packages for the stack that include benchmarks that aren't currently covered.

Future users might want to clear up the text that is being written to the log files. This function has been used during the development of the code to identify and fix bugs, but it is probably not needed anymore. Some of these things were cleaned up once the coding was done, but there might still be room for improvement.

Bibliography

[Advanced Clustering Technologies, 2022] Advanced Clustering Technologies (2022). How do I tune my HPL.dat file? `https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/`. Accessed: 2023-08-10. (Cited on page 41)

[AMD, 2023] AMD (2023). HPCG 3.0. `https://www.amd.com/en/technologies/infinity-hub/hpcg`. Accessed: 2010-09-30. (Cited on page 18)

[Baker, 2000] Baker, M. (2000). Cluster Computing White Paper. *CoRR*, cs.DC/0004014. (Cited on page 4)

[Baker and Buyya, 1999] Baker, M. and Buyya, R. (1999). Cluster computing at a glance. *High Performance Cluster Computing: Architectures and Systems*, 1:3–47. (Cited on pages 3 and 4)

[Bauke and Mertens, 2006] Bauke, H. and Mertens, S. (2006). *Cluster-Design*, pages 51–73. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on pages 1 and 3)

[Carrington et al., 2005] Carrington, L., Laurenzano, M., Snavely, A., Campbell, R. L., and Davis, L. P. (2005). How well can simple metrics represent the performance of HPC applications? In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 48–48. (Cited on pages 15, 16, 24, 25, and 27)

[Dimitrov and Skjellum, 2003] Dimitrov, R. and Skjellum, A. (2003). Software Architecture and Performance Comparison of MPI/Pro and MPICH. In Sloot, P. M. A., Abramson, D., Bogdanov, A. V., Gorbachev, Y. E., Dongarra, J. J., and Zomaya, A. Y., editors, *Computational Science — ICCS 2003*, pages 307–315, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on pages 15 and 26)

[Dongarra and Luszczek, 2011] Dongarra, J. and Luszczek, P. (2011). *HPC Challenge Benchmark*, pages 844–850. Springer US, Boston, MA. (Cited on page 19)

[Dongarra et al., 2003] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820. (Cited on pages 15, 16, and 17)

[Farreras et al., 2009] Farreras, M., Marjanovic, V., Ayguade, E., and Labarta, J. (2009). Gaining asynchrony by using hybrid UPC/SMPSs. (Cited on page 17)

[Ferreira and Levy, 2020] Ferreira, K. B. and Levy, S. (2020). Evaluating MPI Message Size Summary Statistics. In *Proceedings of the 27th European MPI Users' Group Meeting*, EuroMPI/USA '20, page 61–70, New York, NY, USA. Association for Computing Machinery. (Cited on page 26)

[Gamblin et al., 2015] Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. (2015). The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA. Association for Computing Machinery. (Cited on pages 7, 8, and 9)

[Gropp, 2011] Gropp, W. (2011). *MPI (Message Passing Interface)*, pages 1184–1190. Springer US, Boston, MA. (Cited on page 20)

[Grune et al., 2012] Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., and Langendoen, K. (2012). *Introduction*, pages 1–51. Springer New York, New York, NY. (Cited on page 20)

[Gustafson and Todi, 1998] Gustafson, J. and Todi, R. (1998). Conventional benchmarks as a sample of the performance spectrum. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 514–523 vol.7. (Cited on page 16)

[Heroux et al., 2013] Heroux, M. A., Dongarra, J., and Luszczek, P. (2013). HPCG Benchmark Technical Specification. (Cited on pages 15 and 18)

[HPCG, 2022] HPCG (2022). Faq. `https://www.hpcg-benchmark.org/faq/index.html`. Accessed: 2023-09-09. (Cited on page 40)

[HPCwire, 2001] HPCwire (2001). IDC AND HPC USER FORUM LAUNCH NEW BENCHMARK RATING SYSTEM. `https://www.hpcwire.com/2001/06/01/idc-and-hpc-user-forum-launch-new-benchmark-rating-system/`. Accessed: 2010-08-12. (Cited on page 24)

[Köhler and Saak, 2013] Köhler, M. and Saak, J. (2013). Flexiblas-a flexible blas library with runtime exchangeable backends. *LAPACK Working Notes*. (Cited on page 20)

[Krommydas, 2023] Krommydas, K. (2023). Benchmarking GEMM on Intel® Architecture Processors. `https://www.intel.com/content/www/us/en/developer/articles/technical/benchmarking-gemm-with-intel-mkl-and-blis-on-intel-processors.html`. Accessed: 2023-08-08. (Cited on page 27)

[Lafontaine and Cirkov, 2022] Lafontaine, J. and Cirkov, T. (2022). HPC-Benchmarks. `https://github.com/Azera5/HPC-Benchmarks`. Accessed: 2023-04-05. (Cited on pages 1, 12, 27, and 51)

[Leiserson et al., 2020] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., and Schardl, T. B. (2020). There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495):eaam9744. (Cited on page 1)

[Luszczek et al., 2005] Luszczek, P., Dongarra, J. J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., and Takahashi, D. (2005). Introduction to the HPC Challenge Benchmark Suite. (Cited on pages 18 and 19)

[Marjanović et al., 2015] Marjanović, V., Gracia, J., and Glass, C. W. (2015). Performance Modeling of the HPCG Benchmark. In Jarvis, S. A., Wright, S. A., and Hammond, S. D., editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 172–192, Cham. Springer International Publishing. (Cited on pages 17, 18, and 25)

[MVAPICH, 2023] MVAPICH (2023). https://mvapich.cse.ohio-state.edu/static/media/mvapich/README-OMB.txt. `https://mvapich.cse.ohio-state.edu/static/media/mvapich/README-OMB.txt`. Accessed: 2023-09-10. (Cited on pages 18 and 26)

[Poenaru et al., 2021] Poenaru, A., Deakin, T., McIntosh-Smith, S., Hammond, S. D., and Younge, A. J. (2021). An evaluation of the Fujitsu A64FX for HPC applications. In *Presentation in AHUG ISC 21 Workshop*. (Cited on pages 15, 25, and 45)

[SCHEDMD, 2021] SCHEDMD (2021). Quick Start User Guide. `https://slurm.schedmd.com/quickstart.html`. Accessed: 2023-06-20. (Cited on pages 5 and 6)

[SCHEDMD, 2023] SCHEDMD (2023). Man Pages. `https://slurm.schedmd.com/man_index.html`. Accessed: 2023-06-02. (Cited on page 5)

[Schröder, Fabian, 2023] Schröder, Fabian (2023). HPC-Benchmarks-2. `https://github.com/fjs-acc/HPC-Benchmarks-2`. Accessed: 2023-09-13. (Cited on pages 1, 29, and 41)

[Spack, 2023] Spack (2023). Command Reference. `https://spack.readthedocs.io/en/latest/command_index.html`. Accessed: 2023-09-13. (Cited on page 9)

[Sterling, 2011] Sterling, T. L. (2011). *Clusters*, pages 289–297. Springer US, Boston, MA. (Cited on pages 3 and 4)

[van de Geijn and Goto, 2011] van de Geijn, R. and Goto, K. (2011). *BLAS (Basic Linear Algebra Subprograms)*, pages 157–164. Springer US, Boston, MA. (Cited on page 21)

[Vince, 2002] Vince, A. (2002). A framework for the greedy algorithm. *Discrete Applied Mathematics*, 121(1):247–260. (Cited on page 22)

[Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. In Feitelson, D., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 4)

Statement of Authorship

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, September 13, 2023

Signature