Seminar Report

Application and System Benchmarks

Johannes Michael Richter

MatrNr: 21555844

Supervisor: Marcus Merz

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2023

Abstract

This report will provide an introduction and presentation into the concept of Application Benchmarks in the context of High-Performance Computing (HPC). It is written in the scope of the course High-Performance Computing System Administration conducted by the univerity of Göttingen and the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG). HPC gains in importance year by year and there is no end in sight. It refers to the usage of massive computing resources in form of super-computers and clusters of servers to solve complex problems. Benchmarks, in HPC context, are designed to provide an analysis of the performance of this massive computing resources and all its aspects which depends on highly parallelized programs. The main purpose of this report is to describe and interpret the function, the deployment and the execution of benchmarks in a HPC environment and also evaluate instructions and results for those. The two benchmarks Gadget and GPAW will be presented in the course of this report.

Contents

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
SCC	Scientific Compute Cluster
GWDG	Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
FLOPS	Floating Point Operations per second
IOPS	Input/Output operations per second
CPU	Central Processing Unit
GPU	Graphics processing unit
RAM	Random access memory
MPI	Message Passing Interface
UEABS	Unified European Applications Benchmark Suite
SSH	Secure Shell
GÖNET	Göttinger Wissenschaftsnetz
HDF5	Hierarchical Data Format
FFTW	Fastest Fourier Transform in the West
GSL	The GNU scientific library
GNU	GNU's Not Unix
HWLOC	Hardware locality library
MB	megabyte
DFT	Density functional theory
TD-DFT	Time-dependent density functional theory
PAW	Projected Augmented Wave
BLAS	Basic Linear Algebra Subprograms
BLACS	Basic Linear Algebra Communication Subprograms
LAPACK	Linear Algebra PACKage
ScaLAPACK	Scalable Linear Algebra PACKage
Libxc	Library of exchange-correlation functions for density-functional theory
ASE	Atomic Simulation Environment

1 Introduction

The goal of this course High-Performance Computing System Administration was to learn about the various concepts of High-Performance Computing and the work a system-administrator has to do in this context. In detail, each of the participants did choose on of the concepts to work on. In my case, I did choose the topic Application and System Benchmarks. The objective was to learn about benchmarking and to try to install and test Application Benchmarks on a HPC system. In this case this system is the Scientific Compute Cluster (SCC) which is a local Tier-2 HPC system of the GWDG that is used by researchers of the Max-Planck-Institute and the university of Göttingen and in our case also students of the university and employees of the GWDG.

HPC makes use of massive computing resources which are realized in form of super-computers and clusters of servers where the hardware is highly efficient interconnected and supports highly parallel running applications. They exceed the capabilities of normal desktop PCs by a factor of one million. HPC systems become more and more important in our times with the availability of unimaginable amounts of information (Internet, Social Media) and data which has to be processed in acceptable time. This is most notably important for research and large-sized companies like Google or Microsoft.

Benchmarks in general, are applications specifically designed to characterize the performance of a system, in this case the SCC system. These results can then be used to rank and compare different systems to improve the own system, identify problems and then monitor the process of fixing these problems. A famous example of such a ranking would be the Top500 list. Usually, a benchmark can not characterize all aspects of a system but only single ones like Floating Point Operations per second (FLOPS), execution time, et cetera. Benchmark-Suites solve this problem by gathering multiple benchmarks that all target different aspects of the system. There exist numerous types of benchmarks like Algorithmic benchmarks, Parallel benchmarks which are especially used in HPC and many more.
Before one can build and execute a benchmark, it is important to understand the purpose of the benchmark, how many dependencies do exist, the various parameters of execution and whether and how results can be verified or not. The executions of the benchmarks will be reviewed in a regression testing style.

In the main part of the report, benchmark deployments of the two benchmarks Gadget and GPAW will be described as well as general prerequisites like the module system and spack, SLURM and sbatch. The module system and spack provide tools for simplified installations of for example dependencies and even a whole benchmark system. SLURM and sbatch aid with regard to the execution of benchmarks via allocating a program in the SCC system by delivering multiple Message Passing Interface (MPI) processes to various computing nodes and its Central Processing Unit (CPU)s.
Finally, the outputs and results of the benchmarks will be presented, evaluated and interpreted with regard to purpose and expected behaviour. I will also evaluate given instructions and tools for compilation of the programs and the verification of their results. In the end, I will conclude the main results and my thoughts on the whole course on its own.

2 High Performance Computing

2.1 Summary

High Performance Computing refers to the use of powerful computing resources to solve complex computational problems that require massive amounts of processing power and data storage. HPC involves the use of supercomputers, clusters of servers, and specialized software to achieve high speeds and efficiency in performing computations. HPC systems are designed to handle large-scale simulations, data analysis, and modeling tasks that would be impossible or prohibitively expensive to perform with traditional computing resources. HPC system are more than a million times faster than the most powerful desktop PC's, laptops or servers.
Applications of HPC include scientific simulations, DNA sequencing and machine learning. The sequencing of the human genome for example took 13 years on the first attempt but with HPC this task can be done in less a day. [IBM]
HPC systems are typically built with high-performance processors, high-speed interconnects, and large amounts of memory and storage. HPC applications require specialized software that is optimized for parallel processing and can take advantage of the system's architecture. An example for such software is MPI which is contained in many programs that are designed for HPC.
HPC systems require specialized knowledge to operate and maintain, including system administration, programming, and algorithm design. This course was created o learn about the aspect of system administration. These systems also consume significant amounts of energy and resources, which has led to efforts to develop more energy-efficient and sustainable HPC systems. There is for example the Green500 [TOP22a] which ranks the supercomputers of the Top500, which will be explained in Section 3: The Concept of Benchmarking , in terms of energy efficiency.
In recent years, there has been a trend towards making HPC resources more accessible to a wider range of users, including small and medium-sized businesses and academic researchers. Cloud-based HPC services and web-based interfaces have made it easier to access and use HPC resources without requiring specialized knowledge or infrastructure.
Overall, HPC is a critical tool for advancing scientific research, engineering, and innovation, and is expected to continue to play a important role in our future in terms of technology and society. The main terms in this context are Artificial Intelligence, Social Media and massive amounts of available data in various research fields.

2.2 Local System

It was my task to install and test benchmarks on the SCC which belongs to the GWDG and is located in Göttingen. It provides the integration of multiple systems like CPU and GPU-Clusters and supports individual applications. This leads to a heterogeneous environment which requires detailed knowledge about the system and specialised scripts for in my case the execution of benchmarks. The system consists of 7 racks, 4 of them water-cooled, 2 Graphics processing unit (GPU)-nodes are air-cooled and one CPU-rack is air-cooled. Overall, there are 18.376 CPU Cores and 99TB RAM distributed over 410 compute nodes. The Interconnect consists of 56GBit/s FDR Infiniband and 100 GBit/s Omni-Paths. [23]

3 The Concept of Benchmarking

As already mentioned in Section 1 Introduction, benchmarks are in particular designed to analyze the performance of various aspects of a HPC system from which some important ones are listed below.

1. Processing speed in FLOPS
2. Execution time
3. Throughput in Input/Output operations per second (IOPS)
4. Power Consumption
5. Environmental Impact

In contrast to measuring instructions per second, FLOPS are a more accurate type of measurement which benchmarks the performance of the CPU. It is used for everything involving floating-point operations and handling very large or very small real numbers which means for example scientific computational research and in our case benchmarking. FLOPS for a HPC system can be calculated with the formula below.

$FLOPS = racks \times \frac{nodes}{racks} \times \frac{sockets}{nodes} \times \frac{cores}{sockets} \times \frac{cycles}{second} \times \frac{FLOPs}{cycle}$
[Wik23]
Throughput or its synonyms "bandwidth" or "transfer rate" is measured in IOPS which analyses the performance of the storage system by counting the Input and Output operations. Power consumption and environmental impact are important aspects of HPC especially in times of climate change and the problems of availability of resources because HPC systems consume large amounts of energy and cost large amount of money.

By the analysis of all these different aspects, HPC systems can be ranked and then compared with each other. In the case of processing speed in FLOPS there is the famous TOP500 project that ranks and maintains a list of the 500 most potent non-distributed HPC systems in this aspect. This list is updated twice a year since June 1993. All systems are evaluated by their performance on the LINPACK benchmark. This benchmark was released in 1973 and solves dense system of linear equations by measuring the performance in 64bit-FLOPS. It is very suitable because the problem size is scalable and the problem itself is very regular. A similar project which aims to analyze throughput in HPC storage systems is the IO500. In this benchmark a score is constructed from multiple measurement. For example the bandwidth with read/write operations and the metadata with create, stat and delete operations.[TOP22b]
There a multiple types of benchmarks for example Algorithmic benchmarks with the already mentioned LINPACK benchmark, Microbenchmarks that analyse small parts of the system like bandwidth, latency or communication, Database benchmarks which analyze throughput and response time of databases. Most importantly to mention are Parallel Benchmarks that are used in HPC systems because they can be executed on multiple Nodes, should be able to be scaled according to that and for that make use of MPI. [19]

Benchmarks are limited by the fact that they usually cover just one aspect of a system. If one want to analyze multiple aspects of their system one can make use of benchmark suites. These contain numerous benchmarks which can measure different aspects of a system. In my case, I used the benchmark suite Unified European Applications Benchmark Suite (UEABS) [Lio22] which currently contains 13 Application Benchmarks that are taken from the pre-existing benchmark suites PRACE and DEISA. The goal was to provide a Benchmark Suite which contains currently relevant and publicly available application codes and datasets of sizes that can be realistically run on large HPC systems. Each application code has multiple test cases, in the most cases 2. One Testcase A, designed for Tier-1 systems, that can be run on 1,000 x86 cores, or equivalent and a Testcase B, designed for Tier-0 systems, that can be executed on up to around 10,000 x86 cores, or equivalent. In my case, I worked on the SCC system so I worked with the first or the first two test-cases designed for lower tier systems.

To complete the introduction of benchmarks, I provide a list of the most important properties which all benchmarks should provide.

1. Relevance
2. Representativeness
3. Equity
4. Repeatability
5. Cost-effectiveness
6. Scalability
7. Transparency

[DB19]
In the presentation of the benchmarks I installed and tested, I will try to check them with regard to this properties. Relevance means that the most important aspects of the system should be analysed by the benchmark. With Representativeness, the results of benchmark should be accepted by academia and the industry, so companies can safely test their products and researchers can compare their findings and projects like Top500 are possible. With respect to this, most of the systems should be fairly comparable which means equity. The results should be repeatable and verifiable and very importantly, they should be transparent and be understandable by the user. Scalability is one of the most important aspect with regard to HPC because to be executable on this systems they must be scalable with regard to the amount of computing nodes and MPI processes. Cost-effectiveness is similar important like Power Consumption and Environmental Impact especially in today's times.

4 Challenges and possible Approaches

4.1 Upcoming Challenges

First of all, the most important things when deploying software like benchmarks in a system like the SCC, is to understand the benchmark, learn about the dependencies and then comprehend in which ways the software can be executed. This is necessary to be able to install the software in the right way and correctly execute it. If one is new in this field like me and even if not, the developers of the software should always provide adequate documentation so one is able to understand all these aspects. If this is not the case and for example the exact versions of essential dependencies are not given, installing such software can quickly become very difficult. Another examples would be if there are too few instructions of the execution of a software and one has to figure out the right configuration on themselves. To overcome such problems or handle updates of a given software, regression testing is the method of choice.

4.2 Regression Testing

Regression testing is a highly important testing technique in software development to assure that a software after being updated or receiving any form of changes still functions on the system and no additional unintentional bugs occurred. This technique involves the rerunning of previous conducted test cases of the modified software to check whether some problems occur in terms of system stability, execution of the program or the quality of the provided results. There exist automated tools that can do this testing in a automatized way rather than manual testing. This is very efficient given that it saves time, minimizes human errors and provides reliable results. Additionally, many test cases can be performed in parallel. It is indispensable to run regression tests during the life cycle of development and every time before updates of the software are about to be pushed.

4.3 Approaches

Nonetheless, I conducted these test cases, which in this case would be the testing of some benchmarks on the SCC system with various changing parameters, rather in the manual way than in a automatized way. In this way I had the overview of all the parameters for example different versions of dependencies or varying configurations of the *sbatch* script which will be explained in section 5.1.3 and could learn about them. The running of test cases will involve the execution of the benchmarks and validation of the results, the performance and integrity of the programs.

5 Benchmark Deployment

5.1 Prerequisites

First of all, before one can start to work in an environment attached to HPC, there are some systems worth to know which facilitate the work to do, in this case the deployment of benchmarks in the SCC system. These systems are listed below:

1. Module system
2. Spack
3. SLURM and sbatch

However, before one can work on the SCC one has to get access to the system. In our case every participant of the course got the necessary information, like a Secure Shell (SSH) key and an account name, to access the cluster via SSH where one can connect from home via a proxy to connect to SCC over *login.gwdg.de* or directly if beforehand one is working on a device in Göttinger Wissenschaftsnetz (GÖNET).

5.1.1 Module system

Effectively, on a HPC system there is more software installed then any user could ever utilize. Additionally to that, every program might have various dependencies and and there might be programs whose dependencies could mutually exclude each other. Furthermore, each software has different settings according to `$PATH` and `$LD_LIBRARY_PATH`. Also, different users may want to execute the same software but in differentiating versions which usually cannot be installed or used at the same time.
Because of these reasons, the module system has a vast amount of software preinstalled on the system and the user can load and unload desired programs in an encapsulated user environment with `module load` and `module unload`. Additionally, already installed packages can be listed with `module list` and further information about a package can be displayed with `module info package` among various other commands retrievable with `module -help`. These modules should not be confused with modules like in python.

5.1.2 Spack

Spack is a tremendously useful package manager especially developed for supercomputers in Linux and macOS and in its functionality similar to the module system. According to github, *spack* consists to 98.2% of python scripts. However, with *Spack* one can easily install desired software which is not preinstalled on the system. This massively diminish the work one has to do because software can be really complex and difficult to install. *Spack* takes the job and does everything for the user, including the installment of all the dependencies. One can retrieve information of the package `spack info package` and install it with `spack install package`. The user can specify various parameters with regard to the installation of software. It is possible to specify the package by customizing version, compilers,compiler flags, CPU architecture and dependencies. Then, similarly to `module`, one can load and unload the installed software into an encapsulated user environment with `spack load package` and `spack unload package`. [End22]

5.1.3 SLURM and sbatch

When the user logs in via SSH, one is located at a front-end node from which for example a benchmark program can be delivered to various compute nodes visualized in Figure 1.
SLURM is a workload manager/job scheduler system that takes requests for computation from various user's front-end nodes and distributes each requests to the computation nodes, where parallel programs can be computed. Each request is prioritized by the scheduler depending for example on requested number of nodes and requested memory

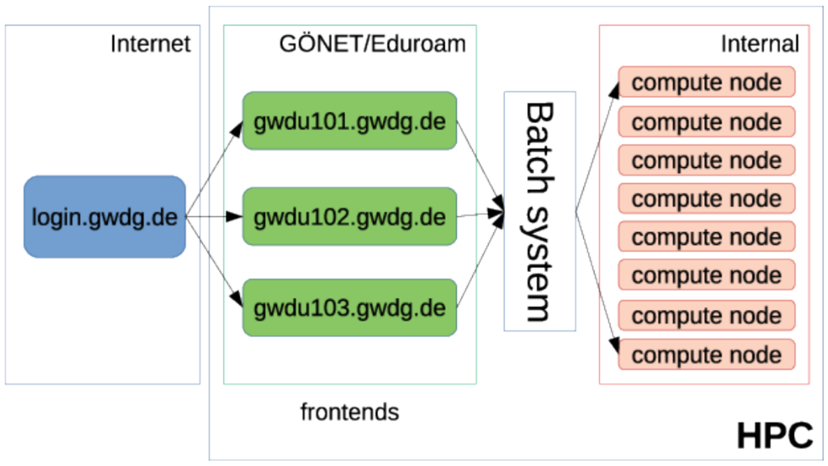


Figure 1: Simplified visualization of a cluster.
Introduction to Slurm [Kel+22]

size. It is then queued among all other requests and eventually begins to compute on the nodes.
The user can submit requests with `srun` to the batch-system `slurm` where one can specify the configuration and distribution of HPC resources for computation. To make lifes easier it is recommended to submit a `srun` request via a sbatch script which is shown below.

```
#!/bin/bash
### selected partition [medium, fat, fat+, gpu]
#SBATCH -p medium
### for parallel job on multiple nodes
#SBATCH --nodes=5
### for MPI, only one process per CPU
#SBATCH --cpus-per-task=1
### no hyper-threading
#SBATCH --ntasks-per-core=1
### number of processes per node
#SBATCH --ntasks-per-node=10
#SBATCH -o job-%J.out           // generating outputfile

module purge
module load openmpi
module load spack-user
source $SPACK_USER_ROOT/share/spack/setup-env.sh

srun benchmark
```

Listing 1: Example sbatch script for SLURM

Parameters for `srun` are given in lines starting with the phrase `#SBATCH`. These commands have to be at the beginning of the script otherwise they will not be read in. All parameters can be accessed with `$ srun - -help` The script is then submitted via `$ sbatch srun_script.sh`. Submitting a job directly via `srun` in the command line forces the user to be logged in as long as the job is running but there are jobs that might run for days. The solution is such a script which, as soon as submitted, is queued and the user is free to log out. As soon as the job is submitted, there are commands available to get information about this job.

1. `$ squeue -u <user_id>`:
   returns information about the queue status like expected starting time
2. `$ scancel <job_id>`:
   will abort a request
3. `$ sacct`:
   information about past and current jobs are accessed

5.2 Gadget

The benchmark *Gadget* provides information about the scalability and performance of the *gadget-4*-code, made for the astrophysical community. The code, written in C++, packaged in the UEABS as tarballs and is accessible on the respective GitHub repository at `https://repository.prace-ri.eu/git/UEABS/ueabs/-/tree/master/gadget`. It uses MPI and the libraries Hierarchical Data Format (HDF5), The GNU scientific library (GSL) and Fastest Fourier Transform in the West (FFTW).

5.2.1 Gadget-4

Gadget-4, the program to be benchmarked, calculates N-body/hydro-dynamical cosmological simulations in general by being able to process plain Newtonian dynamics, or cosmological integrations in arbitrary cosmologies. It is realized as massively parallel code written in C++ and by that perfectly suitable for HPC. That means it is possible to run the code parallel on multiple nodes and with that should also scale this way. The code was mainly written by Volker Springel from the Max-Planck-Institute for Astrophysics in Garching.
The acronym *Gadget* stands for (**GA** laxies with **D** ark matter and **G** as int **E** rac **T**) and points to the former purpose of the code to study galaxies collisions and is able to run collision-less simulations and smoothed particle hydrodynamics on massive HPC systems. All processes of the program communicate by means of MPI and shared-memory access on multi-core nodes and its compatibility has been confirmed on a large number of Linux/UNIX-based systems. [Spr21]

5.2.2 Dependencies and Compilation

First of all, the corresponding script for the build and compile process of the main code and the test cases, described in this section, is located in the appendix A. Before one can start to build and compile the benchmark, some requirements must be fulfilled.

1. GNU gcc 4.x or later
2. Python

3. OpenMPI/4.0.3
4. HDF5-1.12.1
5. FFTW-3.3.8
6. GSL-2.7

The versions restrictions are highly important. To deviate from these might result in a broken build or it simply will not compile. The GNU's Not Unix (GNU) gcc compiler is needed to compile the program that is written in C++ alongside of Python that aids the building process of the program. OpenMPI is responsible for essential virtual topology, synchronization, and communication functionality between a set of processes that will be mapped to compute nodes and by that enables parallel computing of the benchmark. FFTW is necessary for simulations requiring the TreePM algorithm. To read or write snapshots in the HDF5-format, which we will see in the output files of the benchmark in the results section 6.1 the correspondent library is needed. At last, the library GSL permits for a few cosmological integrations at the start-up of the program.
At first, OpenMPI 4.0.3 has to be installed and loaded with *Spack* whereas compatible versions of FFTW, GSL and HDF5 are available in the module system and can simply be loaded by using it. This has to be done before we can start to build the Test Case A executable. Now, the files *gadget4-case-A.tar.gz* and *gadget4-benchmarks.tar.gz* are needed to build the executable by simply cloning the UEABS repository. Decompressing the *gadget4-benchmarks.tar.gz* gives us the master folder **gadget4-benchmarks**. In this folder the files *Makefile.systype* and *Makefile* must be modified. In *Makefile.systype* the line `SYSTYPE="cascadelake-openmpi"` is added, under this name, paths and compilers will be added tailored to the SCC system. Then, the *Makefile* is modified under `#define available Systems#`.

```
##########################
#define available Systems#
##########################
ifeq ($(SYSTYPE),"cascadelake-openmpi")
include buildsystem/Makefile.comp.cascadelake-openmpi
include buildsystem/Makefile.path.cascadelake-openmpi
endif
```

Listing 2: Gadget Makefile: modified part

Next, the two files in **buildsystem** have to be created. *Makefile.comp.cascadelake-openmpi* can be copied from the existing file *Makefile.comp.gcc*.

```
CPP      =  mpicxx  -std=c++11 # sets the C++-compiler
OPTIMIZE =  -ggdb -O3 -march=native  -Wall -Wno-format-security

ifeq (EXPLICIT_VECTORIZATION,$(findstring EXPLICIT_VECTORIZATION,
                               $(CONFIGVARS)))
# enables generation of AVX instructions (used through vectorclass)
CFLAGS_VECTOR += -mavx -fabi-version=0
CPV      =  $(CPP)
else
CFLAGS_VECTOR =
CPV      =  $(CPP)
endif
```

Listing 3: Gadget Makefile.comp.cascadelake-openmpi

Here, `mpicxx` is the wrapper for the MPI C++-compiler, in this case from OpenMPI 4.0.3. I discovered that version and MPI software per se are highly relevant. Using Intel-MPI will cause compiler errors and using OpenMPI 4.1.1 by utilizing `module load openmpi` will indeed lead to successful compiling and building but in the end in execution will end up in a `'MPI INIT FATAL ERROR'`.
Furthermore, in *Makefile.path.cascadelake-openmpi* the paths of the dependencies have to be configured via environment variables. Here we need the sub-folders **lib** and **include** for each dependency, Hardware locality library (HWLOC) is a tool which allows the program to run on and enabling individual cores but we do not need it for the benchmark.

```
GSL_INCL   = -I$(GSL_ROOT)/include
GSL_LIBS   = -L$(GSL_ROOT)/lib
FFTW_INCL  = -I$(FFTW_ROOT)/include
FFTW_LIBS  = -L$(FFTW_ROOT)/lib
HDF5_INCL  = -I$(HDF5_ROOT)/include
HDF5_LIBS  = -L$(HDF5_ROOT)/lib
#HWLOC_INCL = -I$(LIB_DIR)/hwloc/build/include
#HWLOC_LIBS = -L$(LIB_DIR)/hwloc/build/lib
```

Listing 4: Gadget Makefile.path.cascadelake-openmpi

Finally, the *gadget4-case-A.tar.gz* is uncompressed with a *Config.sh* in the resulting folder which is needed for compilation. The executable is generated via the compiling process started with:
`make CONFIG=../gadget4-case-A/Config.sh EXEC=../gadget4-case-A/gadget4-exe`.
One can now execute the benchmark with `srun` and the `param.txt` as input. Details about this topics are given in the following chapter about results and outputs of the gadget benchmark in section 6.1.

5.3 Gpaw

GPAW is an acronym for "A Projected Augmented Wave code". It is a package of programs designed to calculate electronic structures by means of Density functional theory (DFT) and Time-dependent density functional theory (TD-DFT). DFT is used for analysis of ground state properties such as energetics and equilibrium geometries and TD-DFT enables the calculation of excited state properties such as optical spectra. As seen in the acronym, GPAW uses Projected Augmented Wave (PAW) to exclude the main electrons which allows to do the calculations solely with soft pseudo valence wave functions. These functions do not need to be normalized which allows more efficient calculations. Since we are in the field of HPC it is essential to parallelize the problem. This is possible because in DFT and TD-DFT it is possible to parallelize over electronic states. The program is written in Pyton and C++ and parallelized with MPI. [Lus22]

5.3.1 Dependencies and Compilation

Aside from Python 3.6-3.9, MPI and a suitable C++-compiler, the dependencies of the newest version of GPAW are:

1. Basic Linear Algebra Subprograms (BLAS)
2. Linear Algebra PACKage (LAPACK)
3. Basic Linear Algebra Communication Subprograms (BLACS)
4. Scalable Linear Algebra PACKage (ScaLAPACK)
5. FFTW
6. Library of exchange-correlation functions for density-functional theory (Libxc)
7. Atomic Simulation Environment (ASE)

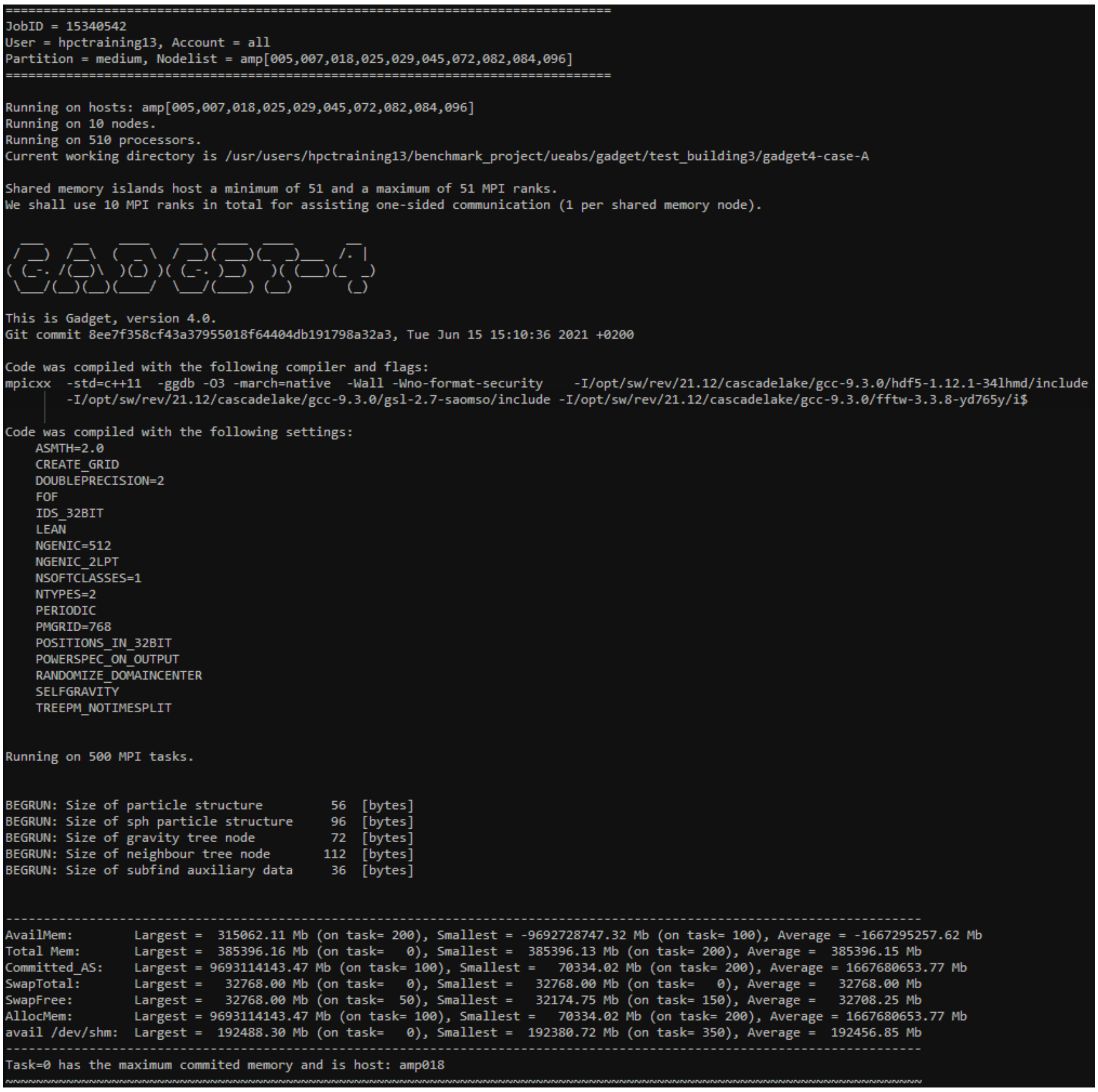
Additionally, the Python packages NumPy ≥ 1.9 and SciPy ≥ 0.14 are necessary. This is a rather long list of dependencies and the manual installation would be a big task, but luckily *Spack* contains the package *py-gpaw* which exactly is the desired program. Hence, the user just has to install and load the program via *Spack* with `spack install py-gpaw` and `spack load py-gpaw`.

6 Execution and Results

In this section, the results, e.g. outputs of 2 out of the 4 benchmarks I deployed, are presented in different test cases and computing configurations. I will provide details about the corresponding output files of the benchmarks and the results of the benchmarks which are execution times, Input/Output performance and verification data.

6.1 Gadget

I used a sbatch script tailored to the given program to successfully execute the gadget benchmark. At first, we have the corresponding `#SBATCH` lines that define the `srun` parameters. The most important one is to provide sufficient Random access memory (RAM) memory which in this case are 2000 megabyte (MB) defined in the *param.txt*. An example script is given in Listing 6. If not explicitly given as a srun-parameter, the benchmark will fail because there is no sufficient shared-memory for the program. The first command after the srun parameters is a `module purge` to provide a clean environment.

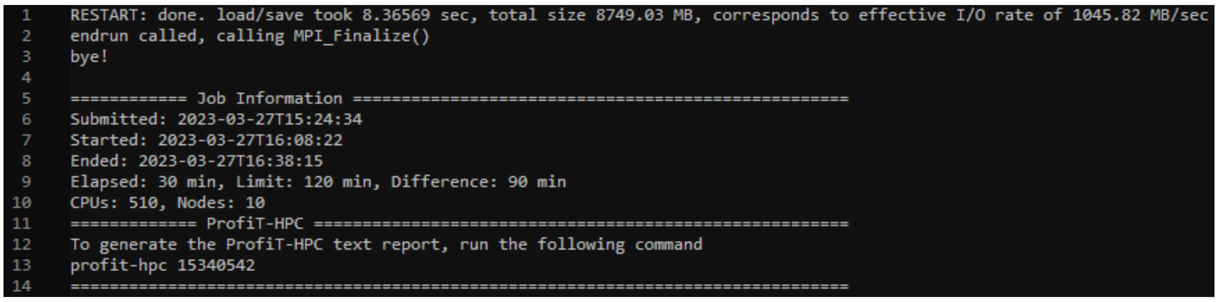


```
===============================================================================
JobID = 15340542
User = hpctraining13, Account = all
Partition = medium, Nodelist = amp[005,007,018,025,029,045,072,082,084,096]
===============================================================================

Running on hosts: amp[005,007,018,025,029,045,072,082,084,096]
Running on 10 nodes.
Running on 510 processors.
Current working directory is /usr/users/hpctraining13/benchmark_project/ueabs/gadget/test_building3/gadget4-case-A

Shared memory islands host a minimum of 51 and a maximum of 51 MPI ranks.
We shall use 10 MPI ranks in total for assisting one-sided communication (1 per shared memory node).

  ___    __    ____    ___  ____  ____       __
 / __)  /__\  (  _ \  / __)( ___)(_  _)___  /. |
( (_-. /(__)\  )(_) )( (_-. )__)   )( (___)(_  _)
 \___/(__)(__)(____/  \___/(____) (__)       (_)

This is Gadget, version 4.0.
Git commit 8ee7f358cf43a37955018f64404db191798a32a3, Tue Jun 15 15:10:36 2021 +0200

Code was compiled with the following compiler and flags:
mpicxx  -std=c++11  -ggdb -O3 -march=native  -Wall -Wno-format-security    -I/opt/sw/rev/21.12/cascadelake/gcc-9.3.0/hdf5-1.12.1-34lhmd/include
        -I/opt/sw/rev/21.12/cascadelake/gcc-9.3.0/gsl-2.7-saomso/include -I/opt/sw/rev/21.12/cascadelake/gcc-9.3.0/fftw-3.3.8-yd765y/i$

Code was compiled with the following settings:
    ASMTH=2.0
    CREATE_GRID
    DOUBLEPRECISION=2
    FOF
    IDS_32BIT
    LEAN
    NGENIC=512
    NGENIC_2LPT
    NSOFTCLASSES=1
    NTYPES=2
    PERIODIC
    PMGRID=768
    POSITIONS_IN_32BIT
    POWERSPEC_ON_OUTPUT
    RANDOMIZE_DOMAINCENTER
    SELFGRAVITY
    TREEPM_NOTIMESPLIT


Running on 500 MPI tasks.


BEGRUN: Size of particle structure        56  [bytes]
BEGRUN: Size of sph particle structure    96  [bytes]
BEGRUN: Size of gravity tree node         72  [bytes]
BEGRUN: Size of neighbour tree node      112  [bytes]
BEGRUN: Size of subfind auxiliary data    36  [bytes]


-------------------------------------------------------------------------------------------------------------------------
AvailMem:        Largest =  315062.11 Mb (on task= 200), Smallest = -9692728747.32 Mb (on task= 100), Average = -1667295257.62 Mb
Total Mem:       Largest =  385396.16 Mb (on task=   0), Smallest =  385396.13 Mb (on task= 200), Average =  385396.15 Mb
Committed_AS:    Largest = 9693114143.47 Mb (on task= 100), Smallest =   70334.02 Mb (on task= 200), Average = 1667680653.77 Mb
SwapTotal:       Largest =   32768.00 Mb (on task=   0), Smallest =   32768.00 Mb (on task=   0), Average =   32768.00 Mb
SwapFree:        Largest =   32768.00 Mb (on task=  50), Smallest =   32174.75 Mb (on task= 150), Average =   32708.25 Mb
AllocMem:        Largest = 9693114143.47 Mb (on task= 100), Smallest =   70334.02 Mb (on task= 200), Average = 1667680653.77 Mb
avail /dev/shm:  Largest =  192488.30 Mb (on task=   0), Smallest =  192380.72 Mb (on task= 350), Average =  192456.85 Mb
-------------------------------------------------------------------------------------------------------------------------
Task=0 has the maximum commited memory and is host: amp018
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 2: Gadget Output: Technical Details.

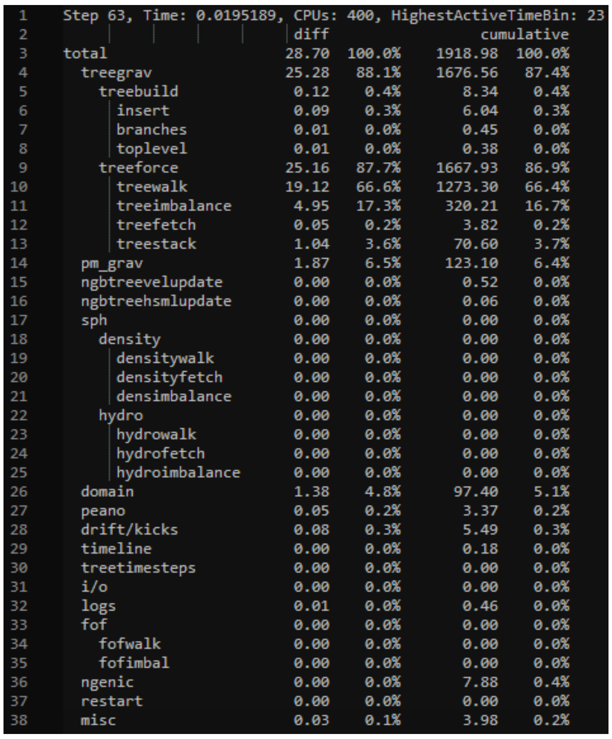
For every nodes that is allocated for computation the benchmark reserves one MPI process for node-to-node communication. That means, in this example, if one wants to compute the benchmark on 10 nodes with effectively 500 MPI process which are calculating, one needs to reserve 51 tasks per node. The following results belong to Testcase A provided

in the UEABS where also instructions for the compiling process were provided. This test-case simulates structures in the universe over time in a small box of linear length, involving dark matter and gravity. Sadly, and I must criticise this, there were flaws in the description and also a wrong *param.txt* for Testcase B so I couldn't test this one. Additionally, there were some typos which irritated me.
In Figure 2 the first page of the benchmarks output is shown. It displays used compiler flags, compiler settings, information about the srun settings and an overview about allocated memory. This summary is followed by a list of the parameter settings of the benchmark itself and then calculation summaries begin. The steps of the calculations are summarized in sync-points. It is possible to interrupt and restart the benchmark if the user wants to. Snapshots are created to realize this and in the end there is a small I/O performance that evaluates loading and saving of this snapshots and the final running time, showed in Figure 3 below.



```
RESTART: done. load/save took 8.36569 sec, total size 8749.03 MB, corresponds to effective I/O rate of 1045.82 MB/sec
endrun called, calling MPI_Finalize()
bye!

=========== Job Information ================================================
Submitted: 2023-03-27T15:24:34
Started: 2023-03-27T16:08:22
Ended: 2023-03-27T16:38:15
Elapsed: 30 min, Limit: 120 min, Difference: 90 min
CPUs: 510, Nodes: 10
============ ProfiT-HPC ====================================================
To generate the ProfiT-HPC text report, run the following command
profit-hpc 15340542
============================================================================
```

Figure 3: Gadget Output: The last lines.



```
Step 63, Time: 0.0195189, CPUs: 400, HighestActiveTimeBin: 23
          |    |    |    |    |diff                  cumulative
total                      28.70  100.0%    1918.98  100.0%
  treegrav                 25.28   88.1%    1676.56   87.4%
    treebuild               0.12    0.4%       8.34    0.4%
      |insert               0.09    0.3%       6.04    0.3%
      |branches             0.01    0.0%       0.45    0.0%
      |toplevel             0.01    0.0%       0.38    0.0%
    treeforce              25.16   87.7%    1667.93   86.9%
      |treewalk            19.12   66.6%    1273.30   66.4%
      |treeimbalance        4.95   17.3%     320.21   16.7%
      |treefetch            0.05    0.2%       3.82    0.2%
      |treestack            1.04    3.6%      70.60    3.7%
  pm_grav                   1.87    6.5%     123.10    6.4%
  ngbtreevelupdate          0.00    0.0%       0.52    0.0%
  ngbtreehsmlupdate         0.00    0.0%       0.06    0.0%
  sph                       0.00    0.0%       0.00    0.0%
    density                 0.00    0.0%       0.00    0.0%
      |densitywalk          0.00    0.0%       0.00    0.0%
      |densityfetch         0.00    0.0%       0.00    0.0%
      |densimbalance        0.00    0.0%       0.00    0.0%
    hydro                   0.00    0.0%       0.00    0.0%
      |hydrowalk            0.00    0.0%       0.00    0.0%
      |hydrofetch           0.00    0.0%       0.00    0.0%
      |hydroimbalance       0.00    0.0%       0.00    0.0%
  domain                    1.38    4.8%      97.40    5.1%
  peano                     0.05    0.2%       3.37    0.2%
  drift/kicks               0.08    0.3%       5.49    0.3%
  timeline                  0.00    0.0%       0.18    0.0%
  treetimesteps             0.00    0.0%       0.00    0.0%
  i/o                       0.00    0.0%       0.00    0.0%
  logs                      0.01    0.0%       0.46    0.0%
  fof                       0.00    0.0%       0.00    0.0%
    fofwalk                 0.00    0.0%       0.00    0.0%
    fofimbal                0.00    0.0%       0.00    0.0%
  ngenic                    0.00    0.0%       7.88    0.4%
  restart                   0.00    0.0%       0.00    0.0%
  misc                      0.03    0.1%       3.98    0.2%
```

Figure 4: Gadget Output: cpu.txt

In addition to that, there is a whole directory containing various files about different aspects of the output provided by the program. These files can be used by experts to verify the results of the benchmark. In the UEABS was no information provided to verify the results for me, as I am not an expert for astrophysics. Figure 4 shows the *cpu.txt* that displays which methods of the program used how many capacities of the CPU. It becomes clear that treegraves needs the most of the computational power. It calculates and simulates gravitation in this Testcase A. In the following Figure 5 I plotted the running times for different amounts of MPI processes. One can observe a flattening curve and can apply the elbow criterion where at around 250 to 300 MPI processes the benchmark does not scale was well as in the beginning.

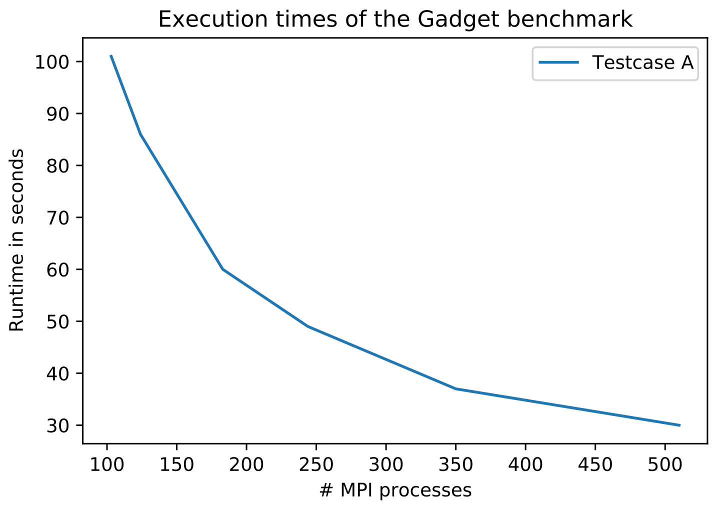


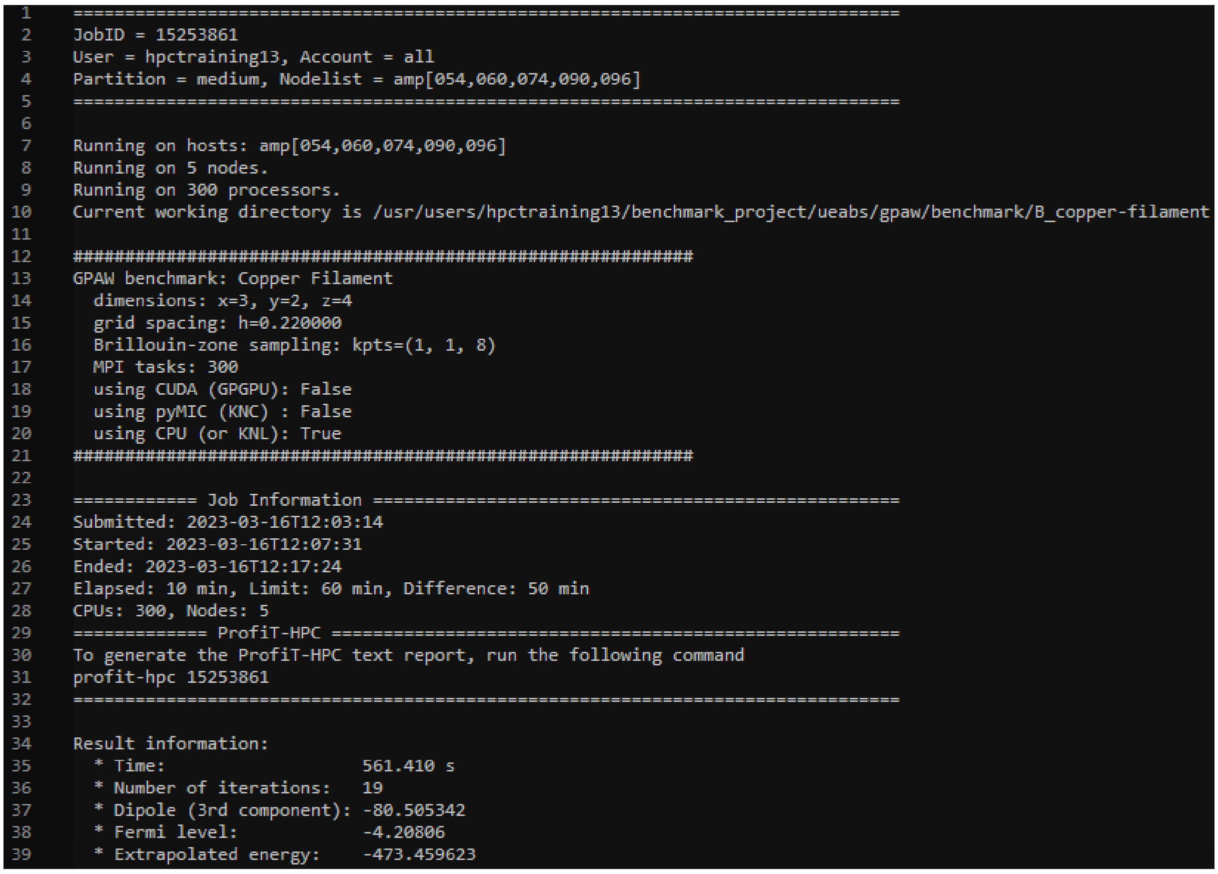
Figure 5: Gadget Output: All runs output

6.2 Gpaw

The UEABS provides three test-cases to run the GPAW-benchmark on.

1. Case A (small): Carbon nanotube: scales up to 10 nodes and/or 100 MPI tasks
2. Case B (medium): Copper filament: scales up to 100 nodes and/or 1000 MPI tasks
3. Case C (large): Silicon cluster: scales up to 1000 nodes and/or 10000 MPI tasks

I ran the first two test-cases but chose to not try the last test-case because the requirements seemed not to be realizable with the SCC system . For verification, for each test-case four parameters are given to check the results of the benchmark e.g Number of iterations, Dipole (3rd component), Fermi level and Extrapolated energy. This comes pretty handy because one can check the results and do not necessarily be an expert for physics. Listing 7 displays an example sbatch script to run the GPAW-benchmark. A script is provided to extract this parameters from the output-file which is an advantage over the Gadget benchmark . An example is given in Figure 6.



```
================================================================================
JobID = 15253861
User = hpctraining13, Account = all
Partition = medium, Nodelist = amp[054,060,074,090,096]
================================================================================

Running on hosts: amp[054,060,074,090,096]
Running on 5 nodes.
Running on 300 processors.
Current working directory is /usr/users/hpctraining13/benchmark_project/ueabs/gpaw/benchmark/B_copper-filament

############################################################
GPAW benchmark: Copper Filament
  dimensions: x=3, y=2, z=4
  grid spacing: h=0.220000
  Brillouin-zone sampling: kpts=(1, 1, 8)
  MPI tasks: 300
  using CUDA (GPGPU): False
  using pyMIC (KNC) : False
  using CPU (or KNL): True
############################################################

============ Job Information ==============================================
Submitted: 2023-03-16T12:03:14
Started: 2023-03-16T12:07:31
Ended: 2023-03-16T12:17:24
Elapsed: 10 min, Limit: 60 min, Difference: 50 min
CPUs: 300, Nodes: 5
============ ProfiT-HPC ===================================================
To generate the ProfiT-HPC text report, run the following command
profit-hpc 15253861
================================================================================

Result information:
  * Time:                    561.410 s
  * Number of iterations:    19
  * Dipole (3rd component): -80.505342
  * Fermi level:             -4.20806
  * Extrapolated energy:     -473.459623
```

Figure 6: GPAW Output: Filtered Output

There exists also the full output of the benchmark which consists of compilation and calculation details and also provides a pseudo-3D visualisation of the simulated molecule. A visualization of all my runs on both test-cases is given on the following Figure 7.

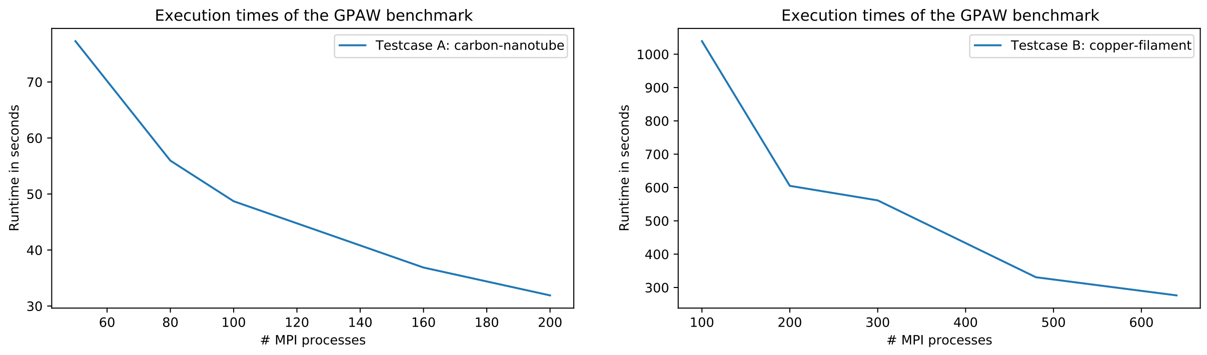


Figure 7: GPAW Output: All runs output

Similar to Figure 5, we can here also observe a flattening curve. Testcase A should scale up to 100 MPI processes but we can see that it scales further than that. Testcase B should scale up to 1000 MPI processes but it seems that the curve flattens at around 500 MPI jobs but that is just an assumption because I did not test with up to 1000 MPI processes.

7 Conclusion

All things considered, I think the course worked out very well, especially considering that it was its first iteration. In my opinion, the importance and relevance of HPC in the current time is very clear and omnipresent. Primarily with the continuous emergence of Artificial Intelligence, the enormous growing amount of data in the internet and examples in biology like DNA where also always new and massive amounts of data are emerging.
With this in mind, also benchmarks inherit this importance because they are indispensable for maintenance and analysis of a HPC systems performance and integrity. This monitoring of the systems performance is important because a HPC cluster runs 24 hours a day , 7 days a week and is often used to capacity, which leads to abrasion of the hardware.
Regarding the installation and management of software in HPC, the module system and Spack are helpful tools and essential with respect to the time one has to invest as shown in 5.2.2 and 5.3.1. Both Gadget and GPAW were interesting benchmarks to work with in the context of deployment and execution and their purposes. I also would have like to present code_saturn and cp2k, both benchmarks I also worked with and tested, but it did not fit in the scope of this report. I focused on the presented benchmarks because Gadget had to be compiled manually and provided very interesting outputs and GPAW could be build easily by means of Spack and provided also interesting but especially user-friendly outputs and verification tools. A downside of Gadget was, that verification was not really possible because I am not an expert in astrophysics, whereas GPAW solved this with simple verification parameters. The UEABS was a very helpful tool to investigate a range of benchmarks but was also a bit overwhelming in its scope. It provided test-cases and helpful instructions for compilation and execution of the programs but in case of Gadget there were confusing typos and faults in the instructions and also the script to extract results from the output file did not function. On the other hand, the execution of GPAW worked out very well, instructions and a given script for validation and extraction of results functioned as well. The SCC system also worked as intended both in its utilization environment and its HPC performance. The evaluation of the results of the benchmarks confirms this observation.
Finally, all of the aspects, the course, the SCC system and the benchmarks I presented, worked out as intended and granted meaningful insight into the massive topic HPC and confirmed its major importance.

References

[19] *Benchmarks.* 2019. URL: `https://hpc-wiki.info/hpc/Benchmarks`.

[23] *Scientific Compute Cluster (SCC).* 2023. URL: `https://hpc-neu.gwdg.de/hpc/systems/scc/`.

[DB19] Wei Dai and Daniel Berleant. "Benchmarking Contemporary Deep Learning Hardware and Frameworks: a Survey of Qualitative Metrics". In: 12-14 (Dec. 2019). URL: `https://dberleant.github.io/papers/Benchmarking%20ContemporaryDeepLearningHardwareAndFrameworks.pdf`.

[End22] Laura Endter. *Spack.* 2022. URL: `https://hps.vi4io.org/_media/teaching/summer_term_2022/pchpc_spack_slides.pdf`.

[IBM] IBM. *What is High Performance Computing?* URL: `https://www.ibm.com/de-de/topics/hpc`.

[Kel+22] Ruben Kellner et al. *High-Performance System Administration Introduction to Slurm.* 2022. URL: `http://hps.vi4io.org/_media/teaching/autumn_term_2022/hpcsa-slurm.pdf`.

[Lio22] Walter Lioen. *Unified European Applications Benchmark Suite.* 2022. URL: `https://repository.prace-ri.eu/git/UEABS/ueabs`.

[Lus22] Kurt Lust. *GPAW - A Projected Augmented Wave code.* 2022. URL: `https://repository.prace-ri.eu/git/UEABS/ueabs/-/tree/master/gpaw`.

[Spr21] Volker Springel. *Introduction to GADGET-4.* 2021. URL: `https://gitlab.mpcdf.mpg.de/vrs/gadget4/-/blob/master/documentation/01_index.md`.

[TOP22a] TOP500.org. *GREEN500.* 2022. URL: `https://www.top500.org/lists/green500/`.

[TOP22b] TOP500.org. *TOP500.* 2022. URL: `https://www.top500.org/`.

[Wik23] Wikipedia. *FLOPS.* 2023. URL: `https://en.wikipedia.org/wiki/FLOPS`.

A Code samples

```bash
#!/bin/bash
module purge
module load fftw/3.3.8
module load gsl
module load hdf5
module unload openmpi
module load spack-user
source $SPACK_USER_ROOT/share/spack/setup-env.sh
spack load openmpi@4.0.3
tar xvf gadget4-benchmarks.tar.gz
tar xvf gadget4-case-A.tar.gz
cd gadget4-benchmarks
cat Template-Makefile.systype > Makefile.systype
sed -i '17 s/.*/SYSTYPE='cascadelake-openmpi'/' Makefile.systype
sed -i 's/^PYTHON.*/PYTHON   = python/g' Makefile
sed -i '112 i \\nifeq ($(SYSTYPE),'cascadelake-openmpi')\ninclude \
buildsystem/Makefile.comp.cascadelake-openmpi\ninclude \
buildsystem/Makefile.path.cascadelake-openmpi\nendif' Makefile
cd buildsystem
cat Makefile.comp.gcc > Makefile.comp.cascadelake-openmpi
echo -e 'GSL_INCL   = -I\$(GSL_ROOT)/include\nGSL_LIBS   = \
-L\$(GSL_ROOT)/lib\nFFTW_INCL  = -I\$(FFTW_ROOT)/include\nFFTW_LIBS  \
= -L\$(FFTW_ROOT)/lib\nHDF5_INCL  = -I\$(HDF5_ROOT)/include\nHDF5_LIBS\
= -L\$(HDF5_ROOT)/lib\n#HWLOC_INCL = -I\$(LIB_DIR)/hwloc/build/include\n\
#HWLOC_LIBS = -L\$(LIB_DIR)/hwloc/build/lib' \
> Makefile.path.cascadelake-openmpi
cd ..
make CONFIG=../gadget4-case-A/Config.sh EXEC=../gadget4-case-A/gadget4-exe
cd ..
cd gadget4-case-A
echo -e '#!/bin/bash\n#SBATCH -p medium\n#SBATCH --time=02:00:00\n\
#SBATCH --job-name=DM_L50-N512\n#SBATCH --output=g_%j.out\n#SBATCH \
--error=g_%j.error\n#SBATCH --nodes=4\n#SBATCH --ntasks-per-node=31\n\
#SBATCH --cpus-per-task=1\n#SBATCH --mem-per-cpu=2000\nmodule purge\n\
module load fftw/3.3.8\nmodule load gsl\nmodule load hdf5\nmodule \
unload openmpi\nmodule load spack-user\nsource \
$SPACK_USER_ROOT/share/spack/setup-env.sh\nspack load \
openmpi@4.0.3\necho\necho "Running on hosts: $SLURM_NODELIST"\necho \
"Running on $SLURM_NNODES nodes."\necho "Running on $SLURM_NPROCS \
processors."\necho "Current working directory is `pwd`"\necho\nsrun \
./gadget4-exe param.txt' > slurm_script.sh
```

Listing 5: Gadget-Benchmark: Build script

```bash
#!/bin/bash
#SBATCH -p medium
#SBATCH --time=02:00:00
#SBATCH --job-name=DM_L50-N512
#SBATCH --output=g_%j.out
#SBATCH --error=g_%j.error
#SBATCH --nodes=10
#SBATCH --ntasks-per-node=51
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2000
module purge
module load fftw/3.3.8
module load gsl
module load hdf5
module unload openmpi
module load spack-user
source $SPACK_USER_ROOT/share/spack/setup-env.sh
spack load openmpi@4.0.3
echo
echo "Running on hosts: $SLURM_NODELIST"
echo "Running on $SLURM_NNODES nodes."
echo "Running on $SLURM_NPROCS processors."
echo "Current working directory is `pwd`"
echo
srun ./gadget4-exe param.txt
```

Listing 6: Gadget-Benchmark: Example sbatch script

```bash
#!/bin/bash
#SBATCH -p medium
#SBATCH --output=gpaw_%j.out
#SBATCH --error=gpaw_%j.error
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=40

module purge
module load spack-user
source $SPACK_USER_ROOT/share/spack/setup-env.sh
spack load py-gpaw

echo
echo "Running on hosts: $SLURM_NODELIST"
echo "Running on $SLURM_NNODES nodes."
echo "Running on $SLURM_NPROCS processors."
echo "Current working directory is `pwd`"
echo

srun gpaw python input.py
```

Listing 7: GPAW-Benchmark: Example sbatch script