Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Report

Student Cluster Competition 2020

written by

Julius Plehn, Johannes Coym, Ruben Felgenhauer, Daniel Bremer, Roland Fredenhagen, Lina Meyer, Yannik Könneker, Felix Maurer

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Betreuer: Jannek Squar, Michael Kuhn

Hamburg, July 8, 2021

Contents

1 Introduction

1.1 SCC2020 - The COVID-19 Special Edition

Author: Daniel Bremer

The Student Cluster Competition is a yearly competition held by the HPC-AI Advisory Council. Normally it is held over three days at the International Super Computing Conference in Frankfurt am Main with physical attendance of each team. Teams need to acquire sponsors for hardware to build a cluster to run given tasks. Such tasks contain benchmarks like the HPL, HPCC and HPCG, but also different "real world" scientific applications like TensorFlow, ElmerFEM or Gromacs. Furthermore there are coding challenges that require the teams to implement code themselves or secret applications which are presented to the teams in the morning of a competition day and teams have time until the end of the day to get the application to run and to tune the results.

The competition runs with only two big rule: Your cluster cannot consume more than 3kW of electrical power at any point in time and every cluster component must be powered on over the whole span of the competition.
This means that the hardware has to be chosen carefully or that it has to be limited in power draw to be under the power limit. The latter can be done by limiting clock speeds or setting power targets. Also low level tuning like undervolting can be an option.

Due to COVID-19 the 2020 edition was changed very much, regarding participation and cluster hardware. Initially the UHH team planned on using NEC's SX-Aurora TSUBASA vector engine cards as accelerators. As the ISC was held online, SCC was moved to a virtual competition, too. The own hardware criteria was omitted and the competition was moved to NSCC's Aspire 1 cluster.
Previously teams had time before the competition to set up their systems and to get familiar with the hardware. This was not possible this year. Because of this the competition duration was extended to 17 days, not only for 10 hours per day (while the exhibition center is open), but for 24 hours per day. In this time it was possible to get familiar with the cluster, build applications, queue runs into the shared queue, etc. on the Aspire 1 cluster.

With the rule changes, there was also a dedication to the Corona Virus Disease happening with the introduction of the Tinker-HP and Gromacs challenges. These applications can help with analysing virus proteins and therefore fighting the virus. With optimizations that students can find while working with the applications the world wide development or an antidote could be boosted, as well as in future research.

2 NSCC Aspire 1

Author: Daniel Bremer

2.1 Cluster Configuration

This years competition was held on the NSCC Aspire 1 cluster located in Singapore. The computer system is a Peta-FLOPS capable system, spanning 1288 CPU nodes and 128 Tesla K40 equipped nodes, as well as 6 DGX-1 nodes featuring 8 NVIDIA V100 GPUs. All of the system is connected to a 13 PB storage system [30].

2.1.1 Hardware

Hardware configuration of CPU nodes:

CPU	Intel E5-2690v3 (12 cores)
Sockets	2
Memory	128 GB DDR4
OS	RHEL 6
Network	Infiniband EDR (100 Gbps)

Additionally there were 10 nodes configured for large memory jobs - 1 nodes was equipped with 10 TB, 4 nodes with 2 TB and 5 nodes with 5 TB.
GPU nodes were equipped like follows:

CPU	Intel Xeon E5-2698 v4 (20 cores)
Sockets	2
Memory	512 GB DDR4
GPU	NVIDIA V100 (8 per node)
OS	Ubuntu 18.01
Network	Infiniband EDR (100 Gbps)

2.1.2 System Architecture

Aspire 1 is designed to implement a Fat Tree with full bisectional bandwidth. This design enables high connectivity inside the cluster, guaranteed by the Fat Tree, while half of the nodes can communicate simultaneously with their full bandwidth [30].

Attached to the system are two distributed filesystems, a GPFS for `/home`, to which file quotas apply, and a Lustre filesystem mounting a `/scratch` directory without quotas. In total the filesystems are capable of storing 13 PB of data [1]s3SAH17NÄ? .

For job management, the PBS Professional scheduler is installed on the system.

2.2 Challenges

With Aspire 1 we have faced multiple challenges. These ranged from not knowing the system to limited access to the nodes themselves. But we knew that because each team had the same hardware, this year software optimization was key to success.

2.2.1 User Management on User Level

For cluster access we received one account for the whole team. While it is possible to work like if every team member sticks to a specific folder in the user's home, we wanted to make sure that no corruption is possible. For this we implemented a bash hack to create virtual users that each of the team members could use.

```
function loaduserenv() {
    export HOME=/home/projects/50000010/$1
    cd /home/projects/50000010/$1
    source .bashrc
}
```

Listing 2.1: `.bashrc` to enable multiple users in one unix user

A `.bashrc` was created for the UNIX user, that introduced a function `loaduserenv`. This function then modified the `$HOME` variable and set it to a user home in a defined location. In our case, we used the /project mount to store user homes, as it was the only location big enough to store all virtual user home directories and was mounted with a distributed file system.
Users could edit their ssh-config to contain the line `RemoteCommand loaduserenv daniel; /bin/bash` (e. g. for the user "daniel"), which then loaded their virtual environment and set up the system.

2.2.2 Heterogeneous System

Another challenge was given by the cluster architecture. While we learned working on a homogeneous cluster with mistral and the WR cluster, the Aspire 1 installation was a heterogeneous system. The login nodes and CPU cluster nodes were running RHEL 6, while DGX nodes ran on Ubuntu 18.04. Not only did we encounter different operating systems, but with these came version inconsistencies of libraries and tools. This could be noticed heavily in the Coding Challenge, as a discussion in the Slack channel showed that the results heavily depended on the used `libm` version and the checking program failed on versions later than 2.12, as the results deviated massively.
Also with old operating systems came old compilers, so building a GCC version 9 required multiple steps, first building version 7 with `binutils` and then building the target version 9.
While cross-compiling would have been an option, we have decided to directly build on the target systems to take potential advantage of newer libs, when using the Ubuntu nodes.

2.2.3 File Systems

Aspire 1 utilizes three mounts for user data: `/scratch`, `/home` and `/project`.
While `/scratch` allowed for unlimited storage for 30 days, unfortunately if was running on a Lustre 2.13 installation and was mounted with default parameters - this lead to a

broken Spack, as Spack requires a file system that supports locking.
`/home` would have been easy to work with, but unfortunately the quota was set to 50 Gb, which was neither enough for virtual user home directories, CUDA libraries with different versions and a complete Spack stack.
Because of these limitations, we mostly used `/project`, where the virtual user directories resided, as well as our Spack stack.

2.2.4 No `root` Access

A huge benefit of a own cluster is the ability to use root whenever needed or to change hardware settings directly in the BIOS (e. g. setting power consumption behaviours or thermal profiles). With root is is possible to set power target, enable/disable hyper-threading and other hardware tweaks. As this year the power consumption played no role in the competition these barriers did not impact us, but in other cases, root would have been used to update libraries or other software, which then had to be installed in user space, whenever possible.

2.2.5 Brute Force Node Allocation

This section is quite special, but might give some helpful ideas, if the SCC should run again on shared hardware.
During the competition we have noticed that queue length were growing each day and it got impossible to work on some applications, as they required trial and error (e. g. BERT/SQuAD). For the last days, we have implemented a obscure hack that allowed us to work on a GPU node. The used loopholes partially are possible, because PBS was used as a job scheduler.
PBS allows allocation of interactive jobs. While running such an interactive job, resource allocations were not enforces, meaning it was possible to allocate a node with 1 CPU, but run tasks on all CPUs. For us, we have utilized this to allocate jobs that allocated 1 GPU on a node and 10 CPUs. As one GPU was allocated, no of the other team's jobs were starting on this node, as they all were written for 4 GPUs.
To make sure we always have a node available, we were using TMUX terminals, which ran a loop on the allocation with an interactive flag. While the interactive shell is still open, the interactive job is not killed (unless by the time limit) and the node can be SSHed to. When detaching the TMUX terminals, these ran until terminated by a system restart.
Such interactive jobs in theory also could be used with MPI, allocate 4 nodes, build the MPI environment manually and run the application - a scheduler only automates this process. This might even work stealthily when starting TMUX terminals on each allocated nodes (imagine a dev queue with fast queue times - allocate a 5 min job and start a TMUX on the node with the environment), but in this case you actively would manipulate other team's runs, as their machines do not receive full performance when your applications are also running on them, and nobody likes cheating retards.

3 Spack - WR's favorite

3.1 Spack

In order to enable us to configure and build applications and dependencies in many different variants we utilized Spack[1]. Spack was created by Todd Gamblin at the Lawrence Livermore National Laboratory. It eases the use of handling multiple versions of the same software and their dependencies e.g. by altering environments and abstracting version and system specific differences. Build instructions are written in Python and in general follow a simple schema. The user has the choice to specify an available version and to select specific variants. For example the following instruction compiles GCC version 9.2 with support for Fortran: `spack install gcc@9.2.0 languages=fortran`. This package is then installed at the user level and no inference with system packages is to be expected. Depending on other installed packages this variant can then be loaded by executing `spack load gcc`

Spack is used heavily within the WR working group as well in previous student cluster competitions. With over 4800 available packages Spack proved to be a solid foundation for our applications. For our purposes we maintained a custom Spack fork[2] which contains package contributions and configuration changes. To a large extent the available packages were sufficient. However, the HPCG benchmark has been updated and a ChaNGa and Tinker-HP package has been added.

3.1.1 Custom Package

In the following subsection a brief introduction into the development of a Spack package is given. An application that initially was not available was ChaNGa and more details can be found in Chapter 8. In Listing 3.1 an excerpt of this package is shown. This package uses the build rules defined by the `AutotoolsPackage` in line 3 and therefor inherits the phases `autoreconf`, `configure`, `build` and `install` from this given build system. The `autoreconf` phase is only executed if the `configure` script is missing or if the package author specifically requests the regeneration of this file. However, it is recommended to apply additional patches to the configure script directly, as we have done in line 9, if required.
In line 8 the SHA256 hash of ChaNGa 3.4 is defined. Given this hash and a version resolution build into Spack, the url in line 5 is used to download and extract this version

[1]`https://github.com/spack/spack`
[2]`https://git.wr.informatik.uni-hamburg.de/scc/spack`

into a temporary directory. In the next step a dependency on Charm++ is defined. Similar to the GCC example earlier, the variant *build-target* is set to ChaNGa. Requiring a dependency makes Spack setup environment variables like *LD_LIBRARY_PATH* so that other packages can find dependent libraries and other files.
In line 13 and 19 the behavior of the configure and install step is altered. First of all, a required variable *STRUCT_DIR* is set to the path of the utilities that were downloaded in line 11. Finally, in the install step, a new directory is created whereas the *prefix* is the final directory in user space and two binaries are moved to their final destination, while maintaining file permissions.

```
 1  from spack import *
 2
 3  class Changa(AutotoolsPackage):
 4      homepage =
           ↪ "http://faculty.washington.edu/trq/hpcc/tools/changa.html"
 5      url      =
           ↪ "https://github.com/N-BodyShop/changa/archive/v3.4.tar.gz"
 6      git      = "https://github.com/N-BodyShop/changa.git"
 7
 8      version('3.4',
           ↪ sha256='c2bceb6ac00025dfd704bb6960bc17c6df7c746872185845d1e75f47e6ce2a9
 9      patch("fix_configure_path.patch")
10
11      resource(
12          name="utility",
13          url="https://github.com/N-BodyShop/utility/archive/v3.4.tar.gz",
14          sha256="19f9f09023ce9d642e848a36948788fb29cd7deb8e9346cdaac4c945f1416667"
15          placement="utility"
16      )
17
18      depends_on("charmpp build-target=ChaNGa")
19
20      def configure_args(self):
21          args = []
22          args.append("STRUCT_DIR={0}/utility/structures"
23                      .format(self.stage.source_path))
24          return args
25
26      def install(self, spec, prefix):
27          with working_dir(self.build_directory):
28              mkdirp(prefix.bin)
29              install('ChaNGa', prefix.bin)
30              install('charmrun', prefix.bin)
```

Listing 3.1: ChaNGa package

3.2 Spack Chaining

Spack Chaining can be used to maintain multiple independent Spack installations which rely on each other and use another if they fulfill a required dependency. We had several reasons to go with this feature. First of all, we wanted to be able to work independently on our own Spack environments which were shared by the same cluster user account. Additionally, we tried to build a solid foundation of proven packages with good performance characteristics. As can be seen in Chapter 4, especially regarding MPI variants, the best library for an application is a great concern for the performance in general, while each variant introduces challenges by itself. Among others, GCC 9.3, Charm++, various MPI implementations, CUDA and HDF5 where provided. On the individuals Spack version this upstream tree has been configured and other package installations are able to install upon this base installation.

To make use of this feature the configuration in *etc/spack/defaults/upstreams.yaml* has to be set like it can be seen in Listing 3.2. Every dependency lookup or queries like `spack find` will try to fulfill their request by using their own Spack instance first. Otherwise other instances are used in order of appearance in this configuration. For the most part this worked very well for us. The only down sight we encountered was, that packages can easily break if an upstream package is removed like it happened to us with several last-minute MPI variant changes.

```
upstreams:
  scc20-base:
    install_tree: /home/users/industry/isc2020/iscst10/spack/opt/spack
```

Listing 3.2: Spack Chaining

4 MPI Benchmarks

Author: Johannes Coym

As all of our applications had different requirements for their MPI variant we first did a large scale benchmark on several aspects of all of the MPI variants we had available.

4.1 MPI Variants

The MPI variants that were tested were the two MPI variants which were provided on the Aspire cluster of the NSCC, as well as six of our own MPI installations, consisting of mpich, mvapich2, Intel MPI and three different OpenMPI variants. All of our variants were installed in Spack using the following install commands.

```
#!/usr/bin/env bash
set -e

# Clone spack repo and checkout SCC20 branch
git clone https://git.wr.informatik.uni-hamburg.de/scc/spack.git
cd spack
git checkout scc20

# Activate Spack stack
cd ..
. spack/share/spack/setup-env.sh

# Install gcc 9.3.0 and add it as a compiler
spack compiler find
spack install gcc@9.3.0+binutils
spack load gcc@9.3.0
spack compiler find

spack install mvapich2 fabrics=mrail
spack install openmpi fabrics=ucx
spack install openmpi fabrics=mxm ~cuda
spack install openmpi@4.0.3 fabrics=ucx ~cuda
spack install intel-mpi@2019.6
spack install mpich netmod=ucx
```

Listing 4.1: Install all MPI variants with spack

4.2 OSU Benchmark

As the benchmark to test our MPI performance we chose the OSU micro-benchmarks which have several benchmarks testing the performance and latency of MPI solutions. The micro-benchmarks we tested with all of our MPI variants were latency, bandwidth, bi-directional bandwidth, multiple bandwidth, multiple message rate, barrier latency, allreduce latency and allgather latency. Using these benchmarks we then had tests for multiple MPI operations with different block sizes, so we could choose the best MPI variant for our applications.

4.3 Results

The first benchmark we take a look at is the bi-directional bandwidth benchmark and as Figure 4.1 shows, the NSCC preinstalled MPI variants had the lowest bandwidth for most of the tested block sizes. The best variants for most block sizes were OpenMPI with UCX fabrics, in both tested OpenMPI versions, 3 and 4. With 8 KiB block sizes both of these variants had a significant drop in performance, but that was recovered quickly with 16 KiB. For really large block sizes, especially 4 MiB, all MPI variants had significant drops in performance, except the two Intel MPI variants as well as mpich, which all had a pretty consistent speed from 2 MiB to 4 MiB.

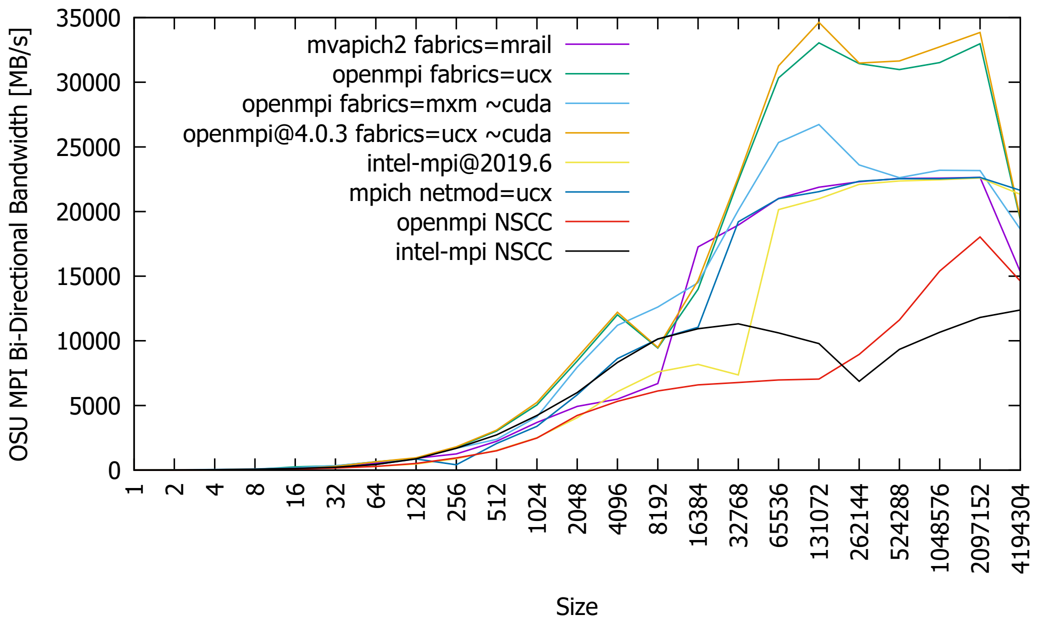


Figure 4.1: Bi-directional MPI Bandwidth

As the following table shows, OpenMPI also has much better latencies for barriers in all variants except the one preinstalled on the Aspire cluster. The preinstalled MPI variants were also much slower in the other benchmarks, so for a better overview they won't be included in the following diagrams.

MPI variant	Barrier Latency in us
mvapich2 fabrics=mrail	1.27
openmpi fabrics=ucx	0.57
openmpi fabrics=mxm ~cuda	0.43
openmpi@4.0.3 fabrics=ucx ~cuda	0.54
intel-mpi@2019.6	2.08
mpich netmod=ucx	1.20
openmpi NSCC	1.85
intel-mpi NSCC	1.69

In Figure 4.2 you can see that the latency during data transfers is pretty similar for all OpenMPI variants on the one hand and Intel MPI, mpich and mvapich2 on the other hand. OpenMPI has a significantly better latency consistently for up to 2 MiB blocks, but like with the bandwidth, in the latency test it is worse than Intel MPI, mpich and mvapich2 with 4 MiB blocks.

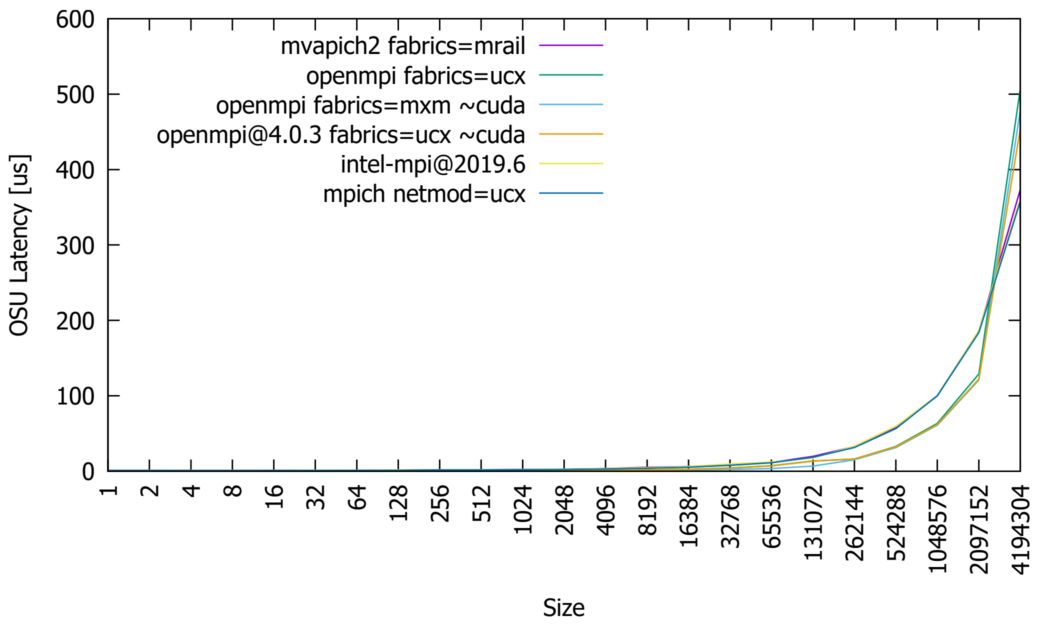


Figure 4.2: MPI Latency

The last figure shows the message rate of the MPI variants and two of the OpenMPI Variants are quite inconsistent with smaller block sizes, but most of the time the message rate of OpenMPI is significantly better than with the other MPI implementations. Only for smaller block sizes mpich is the fastest variant, closely followed by OpenMPI with MXM fabrics.

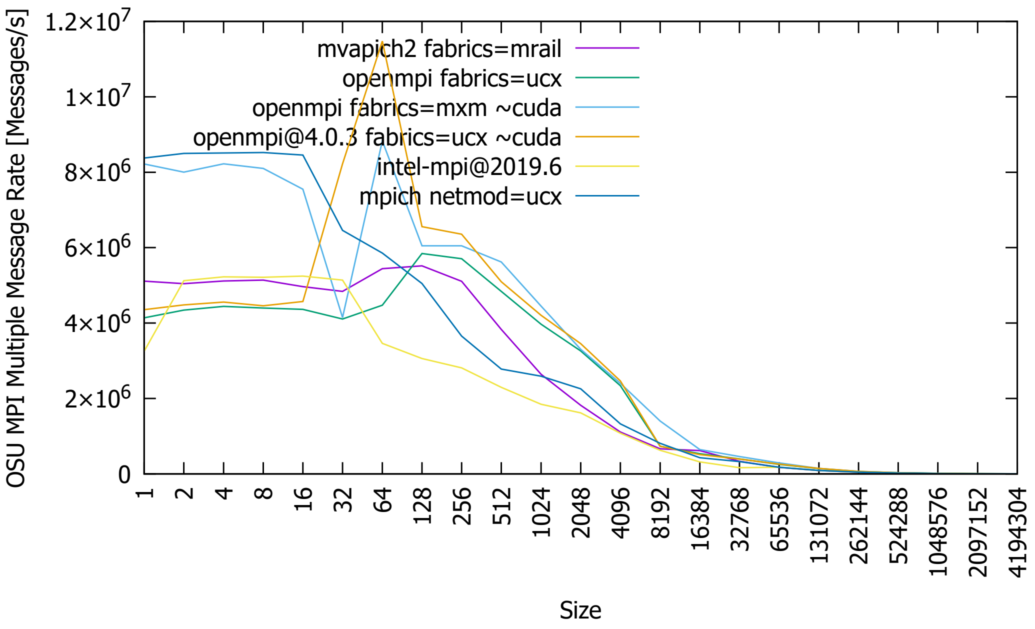


Figure 4.3: MPI Multiple Message Rate

With all of the benchmarks analyzed there was not one MPI solution which was best for all cases but this provides a pretty good help to select a fitting MPI variant. Without specific requirements OpenMPI was a solid option most of the time, but for example with Tinker-HP we found mpich to be the best option, so these benchmarks are only an assistance for the selection.

5 Tinker-HP

Author: Johannes Coym

Tinker-HP is a tool for long polarizable[1] molecular dynamics simulations and to polarizable Quantum Mechanics(QM)/Molecular Mechanics(MM). It is based on Tinker but Tinker-HP adds the ability to use with long molecular dynamics simulations. Also, unlike the original Tinker, Tinker-HP is only CPU-based and optimized to be run on a great number of nodes and CPU cores.

5.1 Architecture

Compiling Tinker-HP results in five applications, called `analyze`, `bar`, `dynamic`, `testgrad` and `minimize`. The `dynamic` application is of special interest, as this is the only one that was used in the competition. The others are used to modify or analyze the grid or the simulation in general, whereas `dynamic` is used to run the molecular dynamics simulation. Using the regular MPI call, a run for the competition could look like the following: `mpirun -np 128 dynamic papain 1000 2.0 1.0 2 300`. In this case, the simulation expects the papain geometry data to be available in *papain.xyz*, which is a widely used chemical file format and the simulation setup in *papain.key*. The simulation itself is influenced by various parameters and an overview is given in the official tinker guide[2]. Additionally, 1000 MD steps, 2 femtoseconds for each step, 1 picosecond between writing of geometry and NVT for the statistical ensemble are selected, while the simulated temperature is 300 kelvin.

While we were not allowed to change the command line parameters, it was indeed possible to change parameters within the key file. It turned out, that as long as the temperature oscillates at around 300 kelvin the results where valid. Due to time constraints, we were not able to gain performance improvements at this stage.

5.2 Challenge

For the competition we recieved three different input files which we needed to execute while minimizing the execution time one step took on average. The first input file was `papain` which consists of 160.233 atoms and represents an enzyme present in papaya. The second input file, `protease_dimer`, is of a similar size with 168.076 atoms and represents an enzyme that can split proteins or peptides. The last input file was `stmv`,

[1]Property of particles to change their charge in response to an electric field
[2]`https://dasher.wustl.edu/tinker/downloads/tinker-guide.pdf`

which is short for Satellite Tobacco Mosaic Virus, which is a specific virus that worsens the symptoms of the Tobacco mosaic virus. It is by far the largest of the input files with 1.066.624 atoms, which set us in front of a small problem. As the higher number of atoms also uses far more RAM, it was impossible to run `stmv` as a small scale test on a single node, we needed at least 2 nodes to run it. Because of the higher size we were also only required to run 100 steps, opposed to the 1000 steps for the first two input files.

5.3 Installation

For the installation we created our own Spack package which used Intel MKL as the math library since it is the only directly supported math library by Tinker-HP. We also installed several versions of Tinker-HP, using several different versions of OpenMPI, MPICH, mvapich2 and Intel MPI with our package to select the quickest variant. In the end we selected MPICH as our MPI variant as it provided consistently the best results of all MPI variants. In the following is the full install script using our Spack package and GCC 9.3.

```bash
#!/usr/bin/env bash

set -e

# Clone spack repo and checkout SCC20 branch
git clone https://git.wr.informatik.uni-hamburg.de/scc/spack.git
cd spack
git checkout scc20

# Activate Spack stack
cd ..
. spack/share/spack/setup-env.sh

# Install gcc 9.3.0 and add it as a compiler
spack compiler find
spack install gcc@9.3.0+binutils
spack load gcc@9.3.0
spack compiler find

# Install tinker-hp using mpich
spack install tinker-hp ^mpich device=ch4 netmod=ucx
```

Listing 5.1: Install Tinker-HP with spack

5.4 MPI Performance

In order to select the best performing MPI Implementation we created an automatic benchmark application which creates various jobs for specific tinker benchmarks. The very first benchmarks were run on Mistral which uses SLURM as a job scheduler and therefore various changes had to be made to support the PBS Scheduler [3]. The output of these runs were visualized by creating plots using a small Python application[4]. An example output can be seen in Figure 5.1, where the Spack hash of `lk7p5de` translates to the use of MPICH using the ch4 device and netmod UCX.

[3] `https://git.wr.informatik.uni-hamburg.de/scc/2020/src/branch/master/Tinker-HP/tinker-examples/example/benchmark_dhfr_pbs.sh`

[4] `https://git.wr.informatik.uni-hamburg.de/scc/2020/src/branch/master/Tinker-HP/benchmark-visualizer/plot.py`

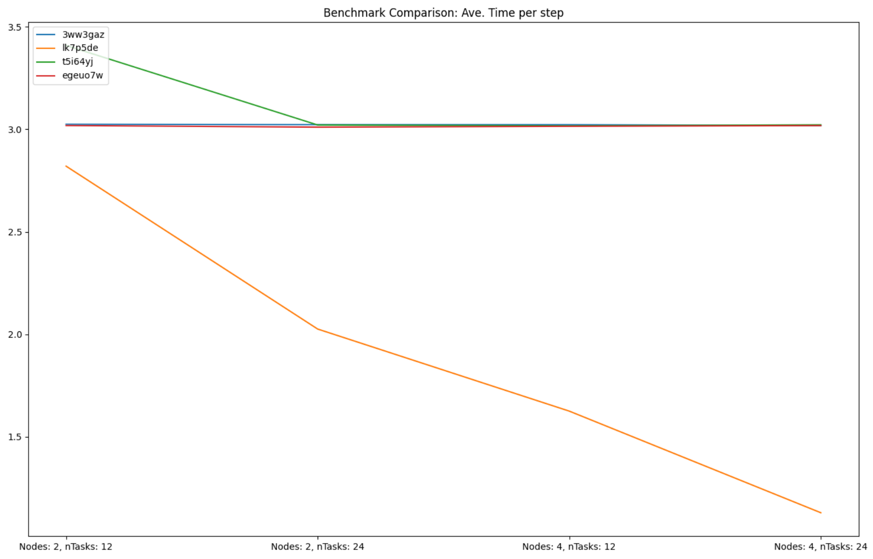


Figure 5.1: Tinker-HP performance using different MPI Implementations

In the end, the overhead of running benchmarks for many implementations using several runs turned out to be of limited use on a cluster with long queue times. The insights into differences of several MPI implementations were interesting, nevertheless.

5.5 Running

While there were many potential improvements to be made in the simulation setup we focused on the number of MPI processes, OpenMP threads and the compilation in general. As we already discussed the changes during compilation in the previous chapter, only the number of MPI processes and OpenMP threads remain. For general testing of these parameters we only used the `stmv` input on 4 nodes worth our MPICH installatiion which we found worked best. On this ground we then ran the following five different configurations (process numbers per node):

MPI/OMP Configuration	Average time per step
24 procs / 24 threads	10.56s
24 procs / 48 threads	7.99s
12 procs / 12 threads	13.79s
12 procs / 24 threads	13.59s
12 procs / 48 threads	13.73s

As the benchmark results show, the variants with 12 MPI processes per node are significantly slower than the variants with one process per core and with one process per node OpenMP can make good use of Hyperthreading which can also reduce the runtime significantly. For the other input files we didn't even consider the 12 process variants but found out that OpenMP only has a positive effect on `stmv`, which is why we ran the other two input files without OpenMP.

Input + MPI/OMP Configuration	Average time per step
protease_dimer 24 procs / 24 threads	1.2464s
protease_dimer 24 procs / 48 threads	1.2531s
papain 24 procs / 24 threads	1.1037s
papain 24 procs / 48 threads	1.1312s

5.6 Visualization

For the visualization we were only required to visualize `papain` and `protease_dimer`, but the runs which we were supposed to visualize should run with 10000 steps for higher accuracy. This run then created .arc files which were used as an input in VMD to visualize both proteins and were then posted on our twitter page[5] as a short video showing all sides of the protein.

In the following figures you can see single still images of the short videos.

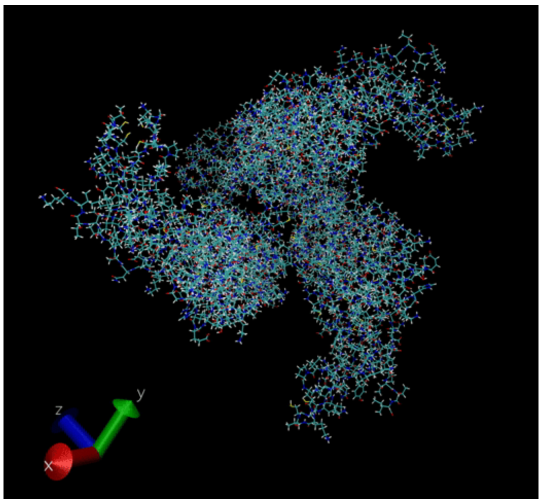


Figure 5.2: Visualization of papain

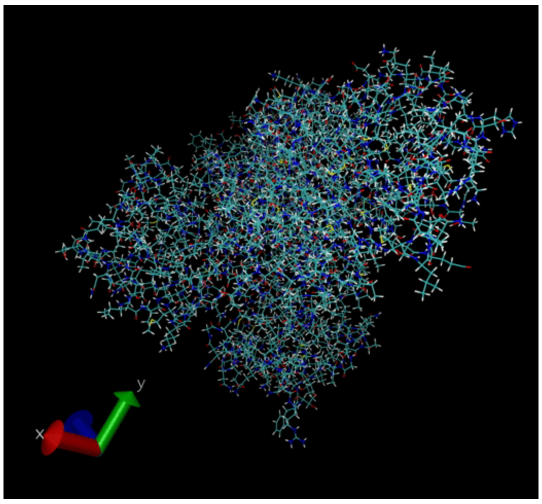


Figure 5.3: Visualization of protease_dimer

[5]`https://twitter.com/UHH_ISC_SCC/status/1272519064624259079`

5.7 Benchmark

As we already mentioned we used the same binary for all of the three inputs in our final runs, but with `stmv` we did use OpenMP, which we did not use with `papain` and `protease_dimer`. For `stmv` we also improved our time from the tests to 6.9915 seconds per step. We initially tested `stmv` with 1000 steps, but our time improved as we only ran the 100 steps that were required for `stmv` in our final run. For `papain` and `protease_dimer` the before mentioned 1.1037 seconds per step and 1.2464 seconds per step were also the best runs which we submitted in the end.

5.8 Recapitulation

Comparing our results with the performance of the winning team we can see room for improvements. We mainly relied on the benchmarks of different MPI implementations whereas changes to the parameters of the actual models could lead to more improvements. Those experiments probably could have been made early on on Mistral and those insights might be of more use on a different system then the very specific insights of MPI implementations. In the end we at least got a point for innovation.

6 Charm++

Author: Ruben Felgenhauer & Felix Maurer & Yannik Könneker

The Charm++ project was initiated and is to this day actively maintained by researchers of the HPC research group of the University of Ilinois. While the first official release of Charm++ dates back to 1993, developement has been conducted since the late 1980's. It gives an alternative approach to managing computing units (i.e. processes) of parallel C++ programs by using a dynamic way of allocating processing ressources at runtime. To achieve this, Charm++ introduces the concept of a "chare" to generalize the idea of processes, omitting the limitation of allocating each computing unit to one exact unit of hardware (i.e node).
Amongst other key features, we will give a thorough overview of chares and how they interact with one another in Section 6.1.
Charm++ utilizes a custom precompiler to generate compilable C++ code. Much like for example MPI's `mpirun`, Charm++ executables are run via a framework-specific `charmrun` binary. Section 6.3 and Section 6.4 cover these two topics.
Additional information on what our workflow with Charm++ looked like can be found in Chapter 7 and Chapter 8.

6.1 Key Features

As mentioned before the Charm++ software stack has been under development for a significant amount of time. Therefore it should not come as a surprise it encompasses a rich variety of features.
To understand the nature of Charm++ and what makes it different to alternative multi-process parallelization frameworks (e.g. MPI), we will now cover some of its main concepts. A more thorough description and guidance on usage for the following concepts can also be found at the official Charm++ documentation. [9]

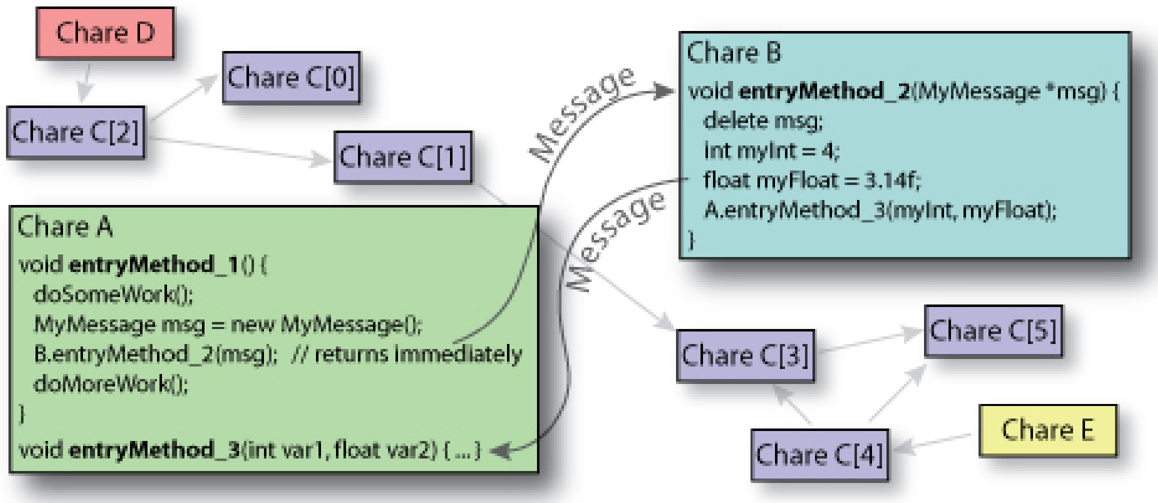
6.1.1 Chares

The Charm++ parallelization paradigm operates at a conceptually high level. You can see this high-level approach when looking at the main object used in Charm++ applications: chares.
At the most basic level, a chare is a C++ object, which represents a certain state via member variables and handles communication to other chares. Chares are dynamically distributed and managed at runtime via Charm++'s load balancer system.
This means that, while from the programmer's perspective chares are somewhat similar

to processes or threads, they are much more flexible. Unlike the latter two, chares can be moved to other computing units at will using built in or custom load balancers and are not bound to certain memory locations at initialization.
Usually you want to have more chares than logical processing resources. This enables the load balancer to move chares from highly utilized to lesser utilized nodes and thus get a better workload distribution overall. This process is called overcommitment. How these chare objects are distributed and how they communicate with each other on a lower level, e.g. via MPI or POSIX-threads, is customizable. We will now describe how Charm++'s API for inter-chare communication works.

6.1.2 Remote Method Invocation

To understand how chares communicate with each other, we will first discuss entry methods. Entry methods are to chares what member functions are to objects in Object Oriented Programming: They define an interface that can be accessed from other chares. To initialize chare objects at least one constructor entry method must be defined. Calling an entry method on another chare invokes communication by sending a message to the target chare. Since Charm++ supports asynchronous communication only, entry methods must return immediately upon invocation and thus cannot have actual return values. This means that in order to get return values you need to send a callback from the originally targeted chare to you.
Figure 6.1 is a visualization of what such a process can look like.



```
Chare A
void entryMethod_1() {
  doSomeWork();
  MyMessage msg = new MyMessage();
  B.entryMethod_2(msg);  // returns immediately
  doMoreWork();
}
void entryMethod_3(int var1, float var2) { ... }
```

```
Chare B
void entryMethod_2(MyMessage *msg) {
  delete msg;
  int myInt = 4;
  float myFloat = 3.14f;
  A.entryMethod_3(myInt, myFloat);
}
```

Figure 6.1: Charm++ remote message invocation visualized [10]

As mentioned earlier, a chare's location in memory can change across nodes throughout execution. This is why in practice, you usually create local proxy objects (provided by Charm++) that keep track of their corresponding chare's location at all times. This creates an easier interface to work with for the programmer, as you can abstract from chare objects being stored in a certain location. Figure 6.2 shows the global object space and how the objects or chares are distributed on multiple nodes.

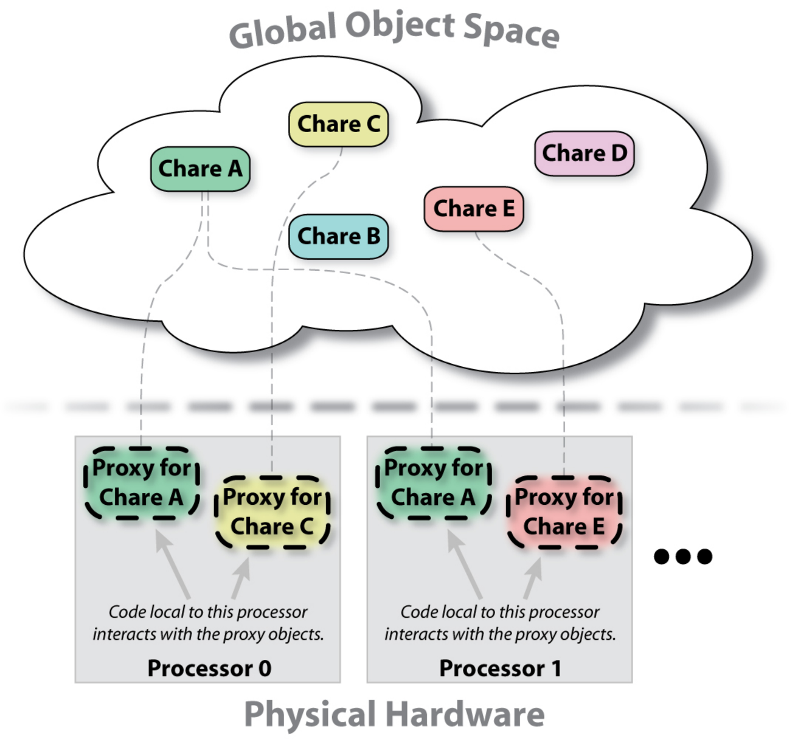


Figure 6.2: Charm++ Proxies visualized [10]

6.1.3 Chare Arrays

Because chares are C++ objects, they can be treated as such. This makes it possible to store them in datastructures, such as special Charm++ ones that require declaration in the respective `.ci` file.
Most commonly used are chare arrays, which interface similar to data structures of the C++ Standard Template Library, i.e. define `operator[]`. While being seamlessly integratable into iterative C++ code, they are in practice often not coherently stored on the same computing unit.
This is due to the fact that objects of chare collections get spread across all available resources just like individual chares would. While this should not negatively impact performance, you can instead use either of the following two to force a certain pattern of distribution upon the Charm++ runtime system:

- Chare Groups: Bijective mapping of chare objects to physical processors.
- Chare NodeGroups: Bijective mapping of chare objects to physical computing nodes.

6.1.4 Reductions

A central concept in distributed computation are reductions, the operation to conclude a solution out of multiple different processing units. Charm++ has it's own asynchronous implementation of reductions. This implementation consist of three parts: a method in a chare that contributes data to the reduction, a function that calculates or reduces the data, and lastly a receiving method that receives the reduced data. Reductions are calculated multiple times, first locally on the same process, then on the same node, then with its neighbours and so on. Once every chare has contributed it's data the final reduction result gets sent to the receiving chare.

6.1.5 Pack/Unpack

The Pack/Unpack (PUP) framework is used to serialize objects into a generic way. This enables sending and unpacking/rebuilding the object on another processor. This is essential for load balancing, fault tolerance and makes checkpoint creation and restarting easier.

6.1.6 Load Balancer

One of Charm++'s biggest strengths are its dynamic load balancers that can move chares between different nodes or processing units to help utilize resources better. Load balancing can happen at creation time or during runtime. Here are a few examples of creation time load balancers:

- random: randomly assigns chares to processing units
- neighbor: chares can be exchanged only between neighbours
- spray: reduces total average load and spreads chares to lesser utilized processing units
- workstealing: idle processing units steal random chares

Runtime load balancers can be categorized into centralized, distributed and hierarchical balancers. Centralized load balancers collect the workload of all processing units and calculate if and where to move chares. Distributed load balancers work on a per node basis, only exchanging chares with neighbours. Hierarchical load balancers balance between these hierarchies: logical processing unit, physical processing unit or node, node groups and root. Balancing only happens on each hierarchy.
Load balancing can happen at synchronized states, periodically or manually. It is possible to write custom load balancers.

6.1.7 Structured Dagger

Because the Charm++ scheduler can interrupt chares it is possible that messages for a given chare are not calculated one after another but in a random order. This behavior can lead to inconsistent data. Structured Dagger (SDAG) introduces the “serial” keyword which is on a conceptually level similar to the “atomic” keyword from C++. If you enclose code within a “serial”, the code cannot be interrupted by the Charm++ scheduler. With the “when” keyword it is possible to store messages and wait until it is allowed for the messages to be calculated.

6.1.8 Adaptive MPI

Another feature of Charm++ is Adaptive MPI (AMPI) [22]: An implementation of the MPI standard that uses Charm++ worker threads instead of MPI processes. Therefore, AMPI can not only be seen as a replacement to other MPI providers like MPICH, MVAPICH2 or OpenMPI, but also rather as a competitor to hybrid approaches that use both MPI and OpenMP [31], or using MPI-3 Shared Memory [5]. Such hybrid approaches are popular in environment in which the data that is used by different ranks is not disjoint and overall too large to fit into a node’s main memory. A performance evaluation of AMPI can be found in a paper from Huang et al. in 2006 [23].
Note that an in-depth comparison between AMPI and its alternative remains to be done and that we did not use AMPI during the SCC. Also note that this is strictly separated from using MPI as a Charm++ backend, which is described in Section 6.2.

6.1.9 Complilation Process of Charm++ executables

As is usual for C++ Programs, a separation of header and implementation files is advised when working with Charm++. This, however, needs to be supplemented by a `.ci` interface file which specifies Charm++ internals, i.e. contains declarations and definition of chares and chare data structures.
This `.ci` file uses a syntax similar to C-style and must contain include directives for all headers your program wishes to use. This is due to the fact that solely this one `.ci` file is fed to the `charmc` compiler.
The complete compilation process of a Charm++ program is depicted in Figure 6.3.

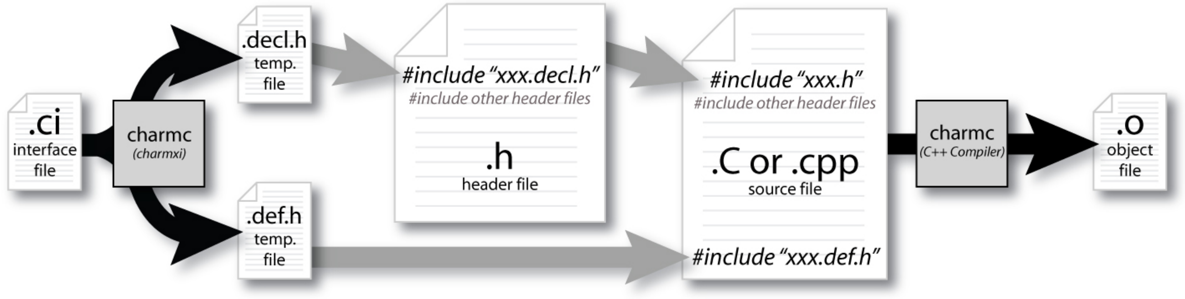


Figure 6.3: Compilation process of a Charm++ program visualized [11]

As you can see, `charmc` can be divided into two parts: First, a precompiler generates temporary header files written in standard C++. It then uses its second mode of operation (i.e as a C++ compiler of choice) to compile the precompiled C++ code.

6.2 Dependencies

Charm++ can utilize different backends: The default is the built-in `netlrts` which uses `ssh`, but it can also use MPI or UCX among others (note that e.g. MPI can also use UCX as a backend). Using MPI as a backend will modify slightly how Charm++ applications are started which is further elaborated in Section 6.4.

6.3 Building Charm++

Charm++ can be installed with Spack as shown in Listing 6.1. As already mentioned in Section 6.2, `netlrts` is the default backend and therefore this argument can also be omitted. Alternatively, one can specify the MPI backend with `backend=mpi ^provider`, where `provider` is an MPI implementation, e.g. `mpich`. A performance evaluation of different MPI providers can be found in Chapter 4. In our final submissions for both the ChaNGa Challenge and the Coding Challenge, we exclusively used the MPI backend. The exact installation directives can be found in Section 7.3.1 for the Coding Challenge, and in Section 8.2.8 for the ChaNGa Challenge, where it has been used as a dependency.

```
$ spack install charmpp backend=netlrts
```

Listing 6.1: Installing Charm++ with the `netlrts` backend

6.4 Running Charm++

Charm++ applications can be started with the `charmrun` binary which is comparable to the `mpiexec` command in MPI environments. Using a standard `netlrts` installation, the general form to start an application can be found in Listing 6.2.

```
$ spack load charmpp
$ charmrun [arguments] executable [arguments]
```

Listing 6.2: Starting a Charm++ application using the `netlrts` backend

The first list of arguments is used by `charmrun`, while the second list of arguments is passed to the specified executable. Charm++ arguments have the form `++argument` or `++argument value` and can be found in the documentation[1].

Notable examples are

- `++pN`, where `N` is an integer, which specifies the exact number of worker threads,
- `++ppn N`, where `N` is an integer, which specifies the number of worker threads per node,
- `++mpiexec` and `++mpiexec-no-n` which allow using `mpiexec`,
- `++remote-shell` which allows starting applications over a specified command which will be executed in a remote shell,
- `++nodelist file` which specifies a file that describes groups of nodes (see the documentation[2]) to be used for the job. The parameter `++nodegroup` can be used to select a named group of nodes from this file and the parameter `++numHosts` can be used to specify how many nodes to assign.

If `charmrun` is used together with `mpiexec`, it will accept all MPI arguments and directly pass it to the `mpiexec` command. This includes arguments like `-ppn` and `-n`, although the latter should be set by `charmrun` automatically if `++mpiexec` is used instead of `++mpiexec-no-n`. On a system that uses SLURM as a batch scheduling system, this should distribute the MPI processes without any further effort. However, on the NSCC Aspire 1 cluster which uses PBS, this proved to be challenging.

[1]`https://charm.readthedocs.io/en/latest/charm++/manual.html#running-charm-programs`
[2]`https://charm.readthedocs.io/en/latest/charm++/manual.html#nodelist-file`

7 Coding Challenge

Author: Felix Maurer & Yannik Könneker

In this year's Coding Challenge we had to program a function that simulates particles moving in a predefined manner within a two-dimensional space (bounding box). The bounding box is divided into a grid of cells. The particles will be moving around in the bounding box with <x,y> double precision floating point coordinates and belong to one of the cells based on their coordinates. In each cell, the particles are stored as a vector. The size of each cell is 1.0×1.0, making the entire bounding box have a size of $n \cdot n$, where n is the number of cells per dimension. We had a simulation size of 35×35 cells and 10.000 iterations.
The entire simulation is written using the Charm++ parallel programming model as described in Chapter 6 and uses the power of the adaptive Charm++ runtime system to parallelize, automatically overlap computation and communication and balance imbalanced load. The assignment was to write a serial - non-parallel - part of the program that moves each particle and then based on the movement of each particle, sends it to an adjacent cell. A lot of the program was already prewritten like the parallelization of the program and the simulation steps. In addition there was a bonus question, in which we had to return the cells with maximum and minimum number of particles and the values of the maximum and minimum particles across the bounding box using Charm++ custom reduction functions.

7.1 Program Architecture

The program sequence is rather simple. The main function reads all parameters, creates the given number of cells as chares, lets the cells generate particles and then starts the calculation. If the termination criteria is met, all cells compare their output to a given pre-calculated output and reduce their data with a Charm++ reduction function to the main class. Available parameters are

- the grid size,
- the number of particles per cell,
- the number of iterations,
- the per cell particle ratio of red, green and blue particles in their respective areas (top, bottom, diagonal, center),
- a factor for reducing the particles' velocity,

- a boolean to enable log output,
- frequency of the automatic load balancer.

The calculation occurs iteratively in every cell individually. First all particle positions get updated, then particles get send to and received from neighbouring cells. Every fifth iteration a reduction happens, printing out how many particles have been moved. At the end of the iteration the cells synchronize to enable the automatic load balancing.

We will now focus on the five most importan files the program consists of.

Methods inside the `main.cpp` file read all parameters, generate cells and start the computation. In the end it collects all data and prints out the results.

`cell.cpp` contains definitions of its own dimensions and a vector of particles. Every cell has a size of 1.0×1.0 and a starting and an end point for both dimensions. The cell at index [3][2] for example only contains particles that have their x coordinate between 3.0 and 4.0 and their y coordinate between 2.0 and 3.0. There are also methods for

- generating particles with given parameters,
- updating the position of a given particle,
- sendinh and receiving particles,
- calculating the number of particles in a cell,
- and verification of the results after finishing the calculations.

The file `particle.h` contains definitions for particles. Every particle has an ID to make comparing results easy. They also have x and y coordinates and a string containing the color used for visual output.

The `particleSimulation.ci` file contains all global variables, entry methods for communication and the computating loop.

The exercise.cpp was almost empty because as part of the main task we were supposed to write a method that sorts particles to their respective new cell after moving. For the bonus question we had to write three new methods. The first method is named `contributeToReduction()` in which the data is being packaged in a Charm++ reduction message and then send to then main proxy. The second method is a Charm++ reduction method named `calculateMaxMin(int nMsg, CkReductionMsg **msgs)` that iterates through all received reduction messages and calculates the cells with the biggest and smallest number of particles inside them. The third method belongs to the main chare and receives the finished reduction message, "unpacks" it and then prints out the result.

7.2 Implementation

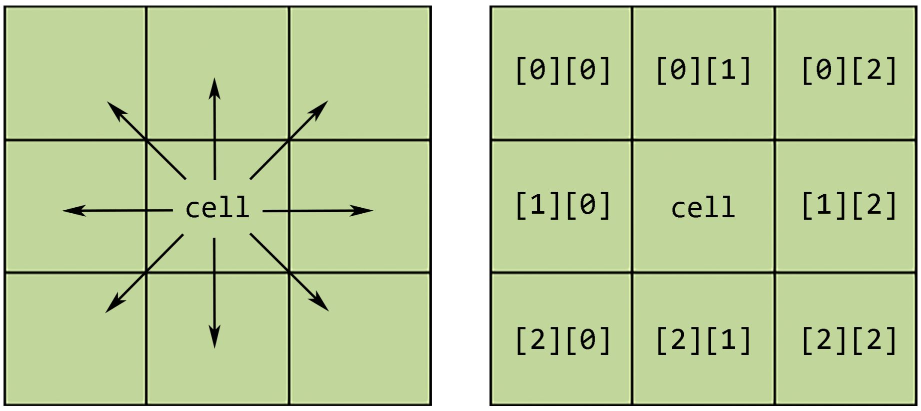
7.2.1 Main Exercise

The gist of our approach is that the particle's position gets updated and then moved into one of eight vectors corresponding to one of the eight neighbouring cells. After that the particles are sent away to their new cell using a predefined method.

```
 1  #include <string>
 2  #include <limits>
 3  using namespace std;
 4
 5  #include "particleSimulation.decl.h"
 6  #include "custom_rand_gen.h"
 7  /*readonly*/ extern CProxy_Main mainProxy;
 8  /*readonly*/ extern CProxy_Cell cellProxy;
 9  /*readonly*/ extern int particlesPerCell;
10  /*readonly*/ extern int numCellsPerDim;
11  /*readonly*/ extern int iterations;
12  /*readonly*/ extern int lbFreq;
13  /*readonly*/ extern int reductionFreq;
14  /*readonly*/ extern double boxMax;
15  /*readonly*/ extern double boxMin;
16  /*readonly*/ extern double cellDim;
17
18  #include "cell.h"
19  #include "main.h"
20
21  //change the position of the particles and send messages to neighbors
       ↪ with their incoming particles
22  void Cell::updateParticles(int iter) {
23      // movedirection bits: 0 0 0 0 left right up down
24      std::vector<Particle> outgoing[3][3];
25      char movedirection = 0x0;
```

Listing 7.1: Includes and definitions

In cell.cpp the particles are stored inside a vector called particles. First we defined a 3×3 vector "outgoing" to map the neighbours in the two dimensional cell chare-array. Each index contains a particle vector because the actual number of particles that leave the current cells coordinates is dynamic. The current cell would be in the middle of the outgoing array but since no particles have to move, the particles won't be put into the middle vector. The graphics in Figure 7.1 visualize the movement of the particles. In line 25 a char "movedirection" is defined that stores where the particle is supposed to go. For that we designed a four bit pattern where if the first bit is a one the particle will move to one of the left neighbouring cells, if the second bit is a one the particle will move to one of the right neighbouring cells. Bit three and four describe the same pattern just for up and down movement. For example 1001 would mean that the particle will be sent to the lower left neighbour. The pattern 1100 would be impossible since no particle can move the the left and the right at the same time. This redundancy enabled this



(a) Visualisation of the particles moving out of the current cells bounds (b) Visualisation of the outgoing array

Figure 7.1: Visualisation of the particle movement from a cell's perspective

simplistic system.

```cpp
for (vector<Particle>::iterator particle = particles.begin(); particle != particles.end();) {
    perturb(&(*particle));

    // check for cell changing
    if (particle->x < startX) {
        movedirection |= 0b1000;
    } else if (particle->x > endX) {
        movedirection |= 0b0100;
    }

    if (particle->y < startY) {
        movedirection |= 0b0010;
    } else if (particle->y > endY) {
        movedirection |= 0b0001;
    }

    // fix x/y values that would be out of boundaries (0.0 < x/y < # of cells)
    if(particle->x < 0)
        particle->x += numCellsPerDim;
    if(particle->x > numCellsPerDim * cellDim)
        particle->x -= numCellsPerDim*cellDim;
    if(particle->y < 0)
        particle->y += numCellsPerDim;
    if(particle->y > numCellsPerDim * cellDim)
        particle->y -= numCellsPerDim*cellDim;

    switch(movedirection) {
        case 0b0001:                                // move particle down
            outgoing[1][2].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b0010:                               // move particle up
            outgoing[1][0].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b0100:                               // move particle
           ↪ right
            outgoing[2][1].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b0101:                               // move particle
           ↪ right-down
            outgoing[2][2].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b0110:                               // move particle
           ↪ right-up
            outgoing[2][0].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b1000:                               // move particle
           ↪ left
            outgoing[0][1].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b1001:                               // move particle
           ↪ left-down
            outgoing[0][2].push_back(*particle);
            particles.erase(particle);
            break;
        case 0b1010:                               // move particle
           ↪ left-up
            outgoing[0][0].push_back(*particle);
            particles.erase(particle);
            break;
        default:
            particle++;
            break;
        //IMPOSSIBLE 0b0011 0b0111 0b1011 0b1100 0b1101 0b1111
    }
    movedirection = 0x0;
}
```

Listing 7.2: For-Loop

In Listing 7.2 you can see the loop that iterates over all particle's within the current cell. First the particles position gets updated by a predefined method called "perturb". Next we decide if the particle is still within the current cells boundaries or if the particle has to be sent away. To determine in which cell the particle will be in the next iteration we simply compare the x and y values of the particle with the values of the cell and, if necessary, set the bit pattern to the corresponding direction. In the lines 18 to 25 we fix the coordinates should the particle go out of bounds of the cell array and, because it is a bounding box, move the particle to the opposite side of the array. Next up is a switch removing the current particle from the cell and putting it into the correct outgoing vector. Nothing happens if the particle stays within the current cell. Afterwards the "movedirection" gets reset to 0 to be ready for the next particle.

```
for(int i = 0; i < 9; i++) {
    int x = (i >= 3) ? i % 3 : i;
    int y = i / 3;
    int newX = thisIndex.x + numCellsPerDim + x - 1;
    int newY = thisIndex.y + numCellsPerDim + y - 1;
    if(x == 1 && y == 1) continue;
    sendParticles((newX >= numCellsPerDim) ? newX % numCellsPerDim :
       ↪ newX,
                 (newY >= numCellsPerDim) ? newY % numCellsPerDim :
                    ↪ newY,
                  iter,
                  outgoing[x][y]);
}
```

Listing 7.3: Sending particles

After every particle has been updated and put into the respective outgoing vector another for-loop sends every particle to it's respective new cell. The for-loop iterates "i" from zero to eight. First the x and y of the outgoing array are calculated from the iterator "i". If x and y point to the middle ([1][1]) of the outgoing array the sending is skipped since the vector is empty. Next up the new cell's coordinate is calculated by adding x - 1 to the current cell's x-coordinate. It is x-1 because if x is 0 it becomes -1 therefore making the targets x-coordinate lower by one. Same goes for the y-coordinate. Because the array is a bounding box we need to add or subtract the maximum value that a coordinate can have. To streamline this process we always add the maximum coordinate and check in the "sendParticles()" if it's too big, if it is, we use the module operator to subtract the maximum coordinate.

7.2.2 Bonus Exercise

This chapter covers reductions as described in Section 6.1.4.

```
extern CkReduction::reducerType minMaxType;

void Cell::contributeToReduction() {
    int data[] = {thisIndex.x, thisIndex.y, (int) particles.size()};
    CkCallback
       ↪ cbMinMax(CkIndex_Main::receiveMinMaxReductionData(NULL),mainProxy);

    contribute(3*sizeof(int), data, minMaxType, cbMinMax);
}
```

Listing 7.4: Reduction contribution

The first method we had to define is a contribution method for a Charm++ reduction. This method gives its data to the custom reduction. First we define an integer array containing the cells coordinates and the number of particles in it. Next we define a CkCallback named “cbMinMax” which stores the method that the reducing chare executes and the address of the reducing chare. In the last step we contribute the size of the input, the input, a pointer to the reduction function and the CkCallback.

```
int outdata[] = {0, 0, std::numeric_limits<int>::max(), 0, 0, 0};

CkReductionMsg *calculateMaxMin(int nMsg, CkReductionMsg **msgs){
    int *data;
    for(int i = 0; i < nMsg; i++) {
        data = (int *) msgs[i]->getData();
        if(data[2] < outdata[2]){
            outdata[0] = data[0];
            outdata[1] = data[1];
            outdata[2] = data[2];
        }
        if(data[2] > outdata[5]){
            outdata[3] = data[0];
            outdata[4] = data[1];
            outdata[5] = data[2];
        }
    }
    return CkReductionMsg::buildNew(6*sizeof(int), outdata);
}
```

Listing 7.5: Reduction Method

This is the reduction method which takes the number of contributions or reduction messages received and an array of reduction messages and returns a reductions message itself. The reduction process is rather simple, we simply iterate over every data input and store the biggest and smallest amount of particles in an array. The array’s data is described like this: {x of lowest, y of lowest, lowest particle amount, x of highest, y of highest, highest particle amount}. We chose the maximum integer value for the lowest particle amount to ensure that in comparison every amount of particles is lower. After comparing the particle amounts a new reduction message is sent containing the reduced values.

```
void Main::receiveMinMaxReductionData(CkReductionMsg *data) {
    int *output = (int *) data->getData();
    minCellX = output[0];
    minCellY = output[1];
    minParticles = output[2];
    maxCellX = output[3];
    maxCellY = output[4];
    maxParticles = output[5];

    CmiPrintf("Max Particles:%d, Cell with Max Particles: (%d,
       ↪ %d)\n", maxParticles, maxCellX, maxCellY);
    CmiPrintf("Min Particles:%d, Cell with Min Particles: (%d,
    ↪ %d)\n", minParticles, minCellX, minCellY);
    readyToOutput();
}
```
Listing 7.6: Receive reduction

Once all chares contributed to the reduction, this method receives the result, maps the array indices to the predefined variables and prints them out.

7.2.3 Difficulties and Optimization Ideas

One of the main goals of all tasks at the Student Cluster Competition is to be time efficient. In case of the Coding Challenge this means optimizing the source code itself, which is what we did.
Here is what we tried. Of course, some ideas came out more successful than others.

Using '%' (modulo) as little as possible

During our performance analysis, we came to the conclusion that the rather expensive operation of modulation can take up a significant amount of time. We thus reduced the usage of the C-Style '%'-operator, which accounted for a notable performance gain.

std::cos is a limiting factor

You might have wondered what the ominous 'perturb' function (called in the beginning of `Cell::updateParticle()`) is doing. It basically uses expensive cosine operations to stretch the program's runtime.
This is why we planned to replace all usages of `std::cos` by a custom function which reads equally accurate cosine values from a precalculated table.
This endeavor failed because of the high precision the program needs to stay congruent to its control values. Rough estimates concluded that we would require at least 600 GB of memory filled with precalculated cosine values, which would almost certainly have meant a loss of performance.

Opting away from std::vector

One big problem which prevented us from using thread-parallel frameworks like OpenMP to further boost the simulation's performance was the fact that the code handed to us uses predominantly `std::vectors` to store data. This data structure is not thread safe. We came to the conclusion that rewriting the entire program to use a thread safe alternative would not be worth the effort.

Using Intel C Compiler

We tried getting Charm++ compile with the ICC but unfortunatly a few libraries were out of date and could not be compiled in a timely manner.

Trying different load balancers

Because the program didn't run efficiently enough to finish we didn't get to try different load balancers.

7.3 Application

7.3.1 Building Options

As for all applications that were part of this year's Student Cluster Competition, we used *spack* as our package manager of choice for a structured apporach of comparing different multiple different implementations of compilers and software dependencies in the Coding Challenge.
Spack allows for an easy side-by-side installation of mutliple instances of the same package with varying build parameters, which further simplified the aforementioned comparison.

For example, the shell command

```
spack install charmpp backend=mpi ^openmpi@3.1.5\%gcc@9.3.0
```

creates a new unique installation of *Charm++*, which uses a specific MPI instance as their communications backbone: OpenMPI 3.1.5 compiled using GCC 9.3.0.

Overall we tried out:

- OpenMPI, MPICH and MVAPICH
- GCC (9.3.0, 4.7.0, 7.4.0)
- with/without MXM

7.3.2 Execution Options

The job scheduler in use at the *Aspire* HPC cluster is *PBS*.
Much like other scheduling systems, *PBS* lets you specify the exact resources (nodes, cores, time) you wish to utilize for running your job.
In addition to that, it is neccessary to give an upper bound on the estimated runtime on your program, i.e. once this so-called "walltime" is reached, your job is terminated.

This proved to be fatal to measuring actual times for a full execution of the `testbench`

at hand using our implementation, as allocating such quantities of time (see Section 7.5) would lead to being queued for inacceptable timespans.
We thus had to either find optimizations that would lead to significant performance increases (factor 200 would have been sufficient) or accept what we could get and see how far that would take us. While unfortunately not succeeding at the former, you can find results of the latter pursuit below.

7.3.3 Job-Script

```
 1 #!/bin/bash
 2 
 3 #PBS -N CC_mvapich_2x24cpu_bench
 4 #PBS -q normal
 5 #PBS -l select=2:ncpus=24:mpiprocs=12
 6 #PBS -l walltime=1:00:00
 7 #PBS -P 50000010
 8 
 9 # set UCX High Bandwidth device
10 #export UCX_NET_DEVICES=mlx5_0:1
11 
12 
13 # ENV SETUP
14 . spack/share/spack/setup-env.sh
15 spack load gcc@9.3.0
16 spack compiler find
17 spack load /bg3sjge
18 spack load /u7sey73
19 
20 #go to cc dir
21 cd CC_MVAPICH
22 
23 #execute cc
24 make clean
25 make testbench > CC_mvapich_2x24cpu_bench.out
```

Listing 7.7: PBS jobscript executing the Coding Challenge’s testbench

This is what the average job script we used looked like.
In lines 3 to 6, you can find macros that are interpreted by *PBS* to allocate ressources accordingly. In this case, we used two nodes running 12 MPI processes each, meaning a total number of 24 CPUs are requested.
In the “ENV SETUP” section of the script, we utilized the *spack* package manager to

make sure the right compiler version and the correct installations of MVAPICH and *CHARM++* are being loaded: The two hashes correspond to specific *spack* installations.

7.4 Results

There were two given modes of execution for this year's Coding Challenge: `test` and `testbench`.
As the first one of these two does not judge efficiency given that it runs for merely a few seconds, it was (as the name suggests) mainly used as a tool to deliver a proof of concept of your implementation.
Our solutions to both main and bonus exercise behave as expected during this `test` scenario.
`Testbench` gives a functionally identical, albeit significantly larger simulation than `test`. While not encountering any errors during execution of this performance benchmark, it failed by timing out in every attempt regardless of the MPI implementation we were using.
We thus conclude our implementation to be correct but too imperformant to be a contender of winning this year's Coding Challenge.

7.5 Performance Analysis

Because of the execution time limit on NSCC the simulation always got interrupted in the 55th iteration. Out of 10.000 that is [1]. We tried executing our code on Mistral and used about 1.700 cores and still only managed to get through 1% of all iterations before timing out after eight hours.

[1]:(

8 ChaNGa

Author: Ruben Felgenhauer

8.1 Background

ChaNGa (Charm N-body GrAvity solver)[1] is a collisionless n-body cosmology simulation application which uses the Charm++ framework which is already described in detail in Chapter 6. "It can perform cosmological simulations with periodic boundary conditions in comoving coordinates or simulations of isolated stellar systems. It also can include hydrodynamics using the Smooth Particle Hydrodynamics (SPH) technique. It uses a Barnes-Hut tree to calculate gravity, with hexadecapole expansion of nodes and Ewald summation for periodic forces. Timestepping is done with a leapfrog integrator with individual timesteps for each particle" [8]. The Barnes-Hut algorithm is further explained in Section 8.2.1.

ChaNGa has first been mentioned in a paper from Jetley et al. in 2008 [25], where it was described as "recently released", while the first commit from ChaNGa's git repository on GitHub[2] has been created in 2002.

In their paper, Jetley et al. state, that "[c]osmological simulators are an important component in the study of the formation of galaxies and large scale structures, and can help answer many important questions about the universe". Consequentially, ChaNGa's development was initiated due to the lack of a cosmological simulation that retains good scalability even on systems with thousands of processors. The developers are an "interdisciplinary group led by Tom Quinn [which] includes faculty and students from the departments of Astronomy and Computer Science and Engineering at the University of Washington" [36] and calls itself the "N-Body Shop".

Furthermore, several techniques are used to improve performance: The overlap between computation and communication is maximized so that communication tasks don't have to wait for pending communication, and "novel load balancing schemes" are used.

8.2 Architecture

Like the Charm++ framework, ChaNGa is written in C++.

[1]`http://faculty.washington.edu/trq/hpcc/tools/changa.html`
[2]`https://github.com/N-BodyShop/changa/`

8.2.1 Barnes-Hut Algorithm

The Barnes-Hut algorithm is an approximation method for n-body simulations which was proposed by Josh Barnes and Piet Hut in 1986 [4]. Its main advantage is its low computational complexity of $\mathcal{O}(n\ \log(n))$ as opposed to traditional methods which calculate the total force that acts on each of the n particles by summing over the forces that are induced by all other $n-1$ particles

$$\vec{F}_i = \sum_{j=1;\ j\neq i}^{n} \vec{F}_{i,j},$$

where the individual forces may be

$$\vec{F}_{i,j} = \Gamma\ m_i m_j\ \frac{\vec{r}_i - \vec{r}_j}{\|\vec{r}_i - \vec{r}_j\|^3}$$

in a simulation that only considers gravity, where Γ is the gravitational constant, m_i is the mass and $\vec{r}_i$ is the position of the i-th particle, yielding a computational complexity of

$$\frac{n(n-1)}{2} \in \mathcal{O}(n^2).$$

The factor of $\frac{1}{2}$ arises from Newton's third law of motion:

$$\vec{F}_{i,j} = -\vec{F}_{j,i}.$$

The algorithm is especially interesting in the field of Astrophysics, where the structure of the simulated objects is far from homogeneous, but rather has a hierarchical nature: Stars (or solar systems) form galaxies, which form galaxy clusters, which then again form superclusters. Inside a galaxy cluster, the gravitational force between two galaxies might be noticeable, even if they have a large distance between them, but the force between two stars in those separate galaxies will be negligible in comparison. Likewise, the gravitational force of a distant galaxy on a star might be in the same order of magnitude as the influence from a star in the same galaxy. Therefore, when simulating the motion of a single star, one has to incorporate the effects of all nearby stars directly, while all the stars of a distant galaxy can be regarded as one object; they can get reduced to the galaxy's mass and its center of mass.

This approximation can only done if the ratio

$$\theta = \frac{d}{r}$$

between the diameter d and the distance r of a group as seen from a particle is small, which is called the Multipole-Acceptance-Criterion. This is guaranteed by partitioning the three-dimensional space into octants until every sector has at most one element and then organizing this structure in a tree, a so-called “Octree”, where every leaf-node represents a particle, and every inner node saves the accumulated mass and center of mass of its children. This is demonstrated in Figure 8.1 for the two-dimensional case.

Calculating the force on each of the n particles can be done by traversing this tree, which has a depth of $\log(n)$ while summing up the gravitational forces, which finally yields the computational complexity of $\mathcal{O}(n\ \log(n))$.

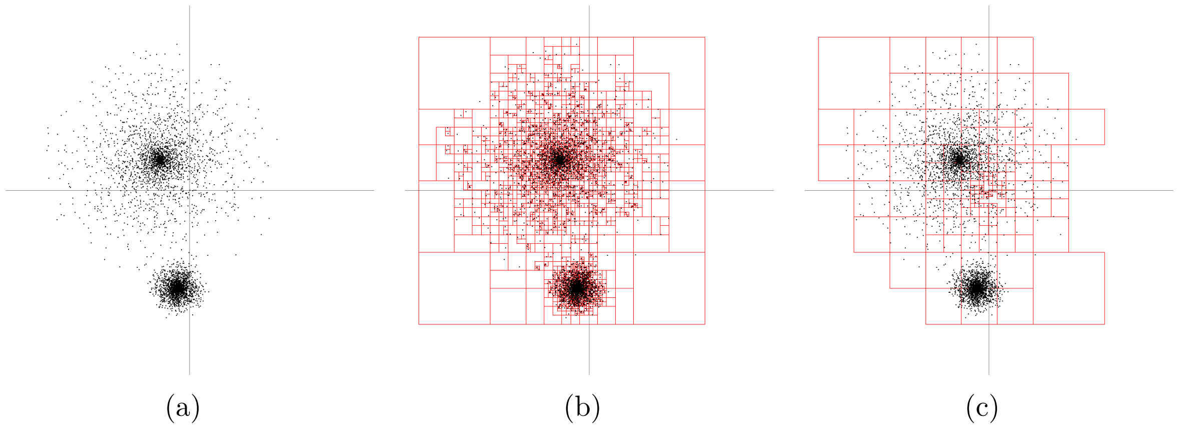


Figure 8.1: Example of 2D Barnes-Hut tree (quad tree). (a) shows a particle distribution that resembles two neighbouring galaxies [12]. (b) shows the complete tree [13]. (c) shows only the cells that are considered for the force on a particle in the origin [14].

8.2.2 Smooth Particle Hydrodynamics?

8.2.3 Comoving Coordinates?

8.2.4 Octree?

8.2.5 Ewald Summation?

8.2.6 Leapfrog Integrator?

8.2.7 Dependencies

ChaNGa depends on Charm++ which has to be installed with ChaNGa set as its build-target as shown in Section 8.2.8. Furthermore, as has already been described in Section 6.2, Charm++ can utilise different backends: For ChaNGa, we only considered the default `netlrts` backend and the MPI backend. Finally, ChaNGa can also be used with CUDA. To ease the use of different configurations a new Spack package has been created which can be found in the official branch[3].

[3]`https://github.com/Julius-Plehn/spack/blob/develop/var/spack/repos/builtin/packages/changa/package.py`

8.2.8 Building

As you can see in Listing 8.1, we installed ChaNGa with Spack and used the MPI backend of Charm++ and OpenMPI with UCX as a backend. We did not use the `netlrts` backend, because we could not manage to enable `charmrun` to find the nodes allocated by the batch scheduling system (both with SLURM on Mistral and PBS on the NSCC Aspire 1), therefore being limited to only one node. The reason for us to use the specified OpenMPI as an MPI provider lies in its performance: In our benchmarks (see Section 8.6.1) this option turned out to be the fastest.

```bash
#!/usr/bin/env bash

set -e

# Clone spack repo and checkout SCC20 branch
git clone https://git.wr.informatik.uni-hamburg.de/scc/spack.git
cd spack
git checkout scc20

# Activate Spack stack
cd ..
. spack/share/spack/setup-env.sh

# Install gcc 9.3.0 and add it as a compiler
spack compiler find
spack install gcc@9.3.0+binutils
spack load gcc@9.3.0
spack compiler find

# Install python3 for the ChaNGa package
spack install python
spack load python

# Install ChaNGa with Charm++, OpenMPI 3.1.5 and UCX
spack clean --all
spack install changa ^charmpp build-target=ChaNGa backend=mpi
    ↪ ^openmpi@3.1.5 fabrics=ucx
```

Listing 8.1: Install ChaNGa with spack

8.3 Assignment

For the SCC, the teams were given the parameters and data-set to “run a cosmological simulation of a galaxy cluster with an unprecedented resolution” [21]. The challenge was split in two: Firstly, the goal was to run this simulation with the shortest runtime possible, using up to 4 nodes of the NSCC Aspire 1 cluster, where the “Big Step” time was monitored from the standard output as a performance indicator, and secondly, the visualisation output of ChaNGa had to be submitted as well which is a series of frames in the Portable Pixmap (PPM) format which were afterwards converted to a video. It was not necessary to generate the visualisation in the same run that was used for the performance benchmark, since the visualisation was decreasing the performance [7]. The performance evaluation can be found in Section 8.6.

8.4 Running

As has already mentioned in Section 8.2.8, we built ChaNGa with Spack using the MPI backend of Charm++. The command that we used for the performance benchmark can be found in Listing 8.2. This command was executed via a PBS script that allocated 4 nodes with 24 CPUs each, resulting in the 96 MPI processes that can be found in the listing. The file "`h1.768.vis.param`" contains the launch configuration. They are accompanied by a file called "`h1.cosmo50PLK.768_dm.002016`" which is about 2.3 GiB in size and contains the initial conditions of the simulation, and a file called "`h1.vis.768_dm.director`" which contains parameters for the visual output. In the following, we will refer to this Benchmark and group of files as the "`h1` Benchmark". For the performance run, we commented out the `dDumpFrameStep` parameter from the param file which deactivates the visual output. As can be found in Section 8.6.1, this took 01:07:35 of walltime. For the visualization run, we only used 2 nodes instead of 4, because the runtime performance wasn't critical and this decreased the waiting time for the jobs to be scheduled significantly, since PBS was configured so that it prefers jobs that require at most 2 nodes. This run took 02:06:28 of walltime.

```
$ spack load changa
$ charmrun +p96 ChaNGa ./h1.768.vis.param
```

Listing 8.2: Starting ChaNGa's performance benchmark

8.5 Parameters / Tuning

For benchmarking, we built ChaNGa with Charm++ which we linked against the following MPI implementations:

- OpenMPI 4.0.3 with UCX
- OpenMPI 3.1.5 with UCX
- OpenMPI 3.1.5 with MXM
- MPICH 3.3.2
- MVAPICH 2.3.3

Since executing the `h1` Benchmark took over an hour on the NSCC Aspire 1, we used a benchmark that we are going to refer to as the "`dwf1` Benchmark" which "[...] is a 5 million particle zoom-in simulation. It is cosmological, but the particle sampling focuses on a single halo of roughly [10^{11}] solar masses" [6]. Since the runtime of this was in the order of several minutes (the fastest being little over 2 minutes), we could perform over 300 benchmarks. An in-depth look into our benchmark results can be found in Section 8.6. For the final run, we also made sure to enable process pinning to CPU threads to keep the scheduler from moving threads around over the cores which can be done by setting the environment variable `KMP_AFFINITY=verbose,granularity=thread,compact,1`.

8.6 Results

8.6.1 Benchmark

Our benchmarks can be split into three phases:

In the first phase we ran ChaNGa with all combinations of the MPI backends as listed in Section 8.5 together with 1, 2 and 4 nodes and 1, 2, 4, 6, 8, 12, 16, 20 and 24 processes per node. The results are not given here for space reasons and showed that the fastest 7 runs were all using 4 nodes, of which the fastest was using 12 processes per node. We concluded that ChaNGa's performance scaled well with the number of used nodes (given only the tiny amount of at most 4 nodes) and that it was relatively safe to assume that we would get the best performance using either 12 or 24 processes. However, this phase also contained a lot of invalid results, e.g. all runs that used OpenMPI, because our job script contained launch parameters that its `mpiexec` could not understand, but the runs that used MVAPICH2 were also failing seemingly randomly.

In the second phase, we only concluded benchmarks that were using 4 nodes and 12 or 24 processes per node. The runs that used OpenMPI were still entirely failing even after removing the unrecognised parameters which only revealed further unrecognised options. Fortunately, all runs that used MPICH or MVAPICH2 were successful. The results can be found in Table 8.1. As one can read from the table, MPICH was consistently faster and the quickest run was using 24 processes per node.

In the third phase we moved to the `h1` Benchmark and tested only MPICH and and MVAPICH2, because we were not confident that we could initiate a successful run using OpenMPI and the batch scheduling queue was getting increasingly longer which resulted in very long wait times. For the same reason, we also used only 1 and 2 nodes to get an approximation for the runtime of the final runs by assuming a sub-linear speedup. Unfortunately, out of 8 runs, only one run was successful: The runtime using MPICH and 2 nodes with 12 processes each was 01:57:17 from which we concluded that we had to account for at least 58 minutes of runtime for the final runs.

Lastly, we decided against using MVAPICH2 for the final runs because of its unreliability and only did a run with MPICH. Afterwards, we also managed to successfully run ChaNGa with OpenMPI 3.1.5 using UCX, which ended up having a shorter runtime, so this run was then included into our final submission. An overview over these two runs, sorted by runtime, can be found in Table 8.2.

Table 8.1: Runtimes of ChaNGa on the `dfw1` Benchmark

Runtime [s]	Runtime [m:s]	Big Step Time[3] [s]	Nodes	PPN	MPI impl.	Fabrics
121.689	2m1.689s	10.3037	4	24	mpich@3.3.2	none
123.307	2m3.307s	10.4831	4	24	mpich@3.3.2	none
124.969	2m4.969s	10.4041	4	12	mpich@3.3.2	none
125.644	2m5.644s	10.3132	4	12	mpich@3.3.2	none
125.792	2m5.792s	10.3139	4	12	mpich@3.3.2	none
130.778	2m10.778s	10.7839	4	24	mpich@3.3.2	none
134.690	2m14.690s	10.4989	4	24	mvapich2@2.3.3	mxm
138.957	2m18.957s	10.3471	4	24	mvapich2@2.3.3	mxm
138.967	2m18.967s	10.6538	4	24	mvapich2@2.3.3	mxm
301.454	5m1.454s	22.2244	4	12	mvapich2@2.3.3	mxm
309.449	5m9.449s	23.7092	4	12	mvapich2@2.3.3	mxm
312.796	5m12.796s	24.1721	4	12	mvapich2@2.3.3	mxm

Table 8.2: Runtimes of ChaNGa on the `h1` Benchmark

Runtime [hh:mm:ss]	Big Step Time[3] [s]	Nodes	PPN	MPI impl.	Fabrics
01:07:35	258.024663	4	24	openmpi@3.1.5	ucx
01:10:53	270.125498	4	24	mpich@3.3.2	none

8.6.2 Visualisation

The visualization of the `h1` data-set is a 3 second long video which can be found at the team's twitter page[4].

[3]This specifies the median Big Step Time from the Standard Output
[4]`https://twitter.com/UHH_ISC_SCC/status/1272519064624259079`

9 ElmerFEM/ICE

Author: Daniel Bremer

9.1 ElmerFEM

Elmer is a Finite Element Solver developed by IT Center for Science (CSC) in Finland. With the software suite it is possible to solve "physical models of fluid dynamics, structural mechanics, electromagnetics, heat transfer and acoustics" [15] and also geological models and ice sheet movements. Inputs for the program usually are a mesh that describes the object that is simulated, and a solver, e. g. Navier-Stokes, that is used for calculation of the problem. Tuning to arbitrary problems can be done by providing a custom solver, best fitted to any problem.
The software suite delivers three main programs: `ElmerSolver` (and `ElmerSolver_mpi`), which is the finite element solver (and its MPI capable counterpart), ElmerGrid, for mesh creation and manipulation, and ElmerGUI, a graphical interface for Elmer.

9.1.1 Elmer/ICE

The main repository on Github[1] features a branch `elmerice` that extends ElmerFEM to model ice sheet movement, glacier and ice flows. It provides a navier-stokes-solver. The underlying Navier-Stokes Equations describe motion of viscous fluid in a set of partial differentials.
Ice, similarly to glass, can be described better as a fluid than as a solid - although a very viscous fluid. This is at least due to the fact that ice is in fact not a "huge block of material", but many grains and impurities of minerals. Also it is incompressible, another property that is shared with fluids.
Flow dynamics can be described with the flow of the ice $\nabla * \sigma + \rho * g = 0$ considering the constant sum of all surface forces and gravity and $\nabla * u = 0$ a constant sum of volumes. Doing this for all 6 surfaces of a block provides a Full Stokes Model.
While the one could approximate ice sheets as shallow ice shelfs, only considering shear stresses in the equations, Elmer/ICE implements a full model without such approximations. Furthermore, it can be coupled with other models, e. g. Bedrock Models that simulate bedrock movement influenced by the mass of ice sheets on top or FISOC, an Ice-Ocean coupler, providing data on ice melting at the ocean borders.

[1]**https://github.com/ElmerCSC/elmerfem**.

9.2 Assignment

With Elmer the task was to simulate the ice sheet movement of the Greenland ice sheet. The model was a coupled model, containing not only the ice sheet, but also oceans at the boundaries of the land/ice mass and the bedrock below the ice. To not only was the ice movement simulated, but also the lift of the bedrock, when the pressure of the ice lifted and temperature influences of the oceans.
Given was a mesh describing the ice sheet, Greenland's bedrock and the surrounding oceans and 4 input files `solversettings.sif`, `header.sif`, `stokes.lua` and `Stokes_steady_vec_lua.sif`. It was allowed to modify the first two files, while the latter ones should remain untouched.
The output files showed a runtime, that was used for performance evaluation. Also it was possible to visualize the results with ParaView.

9.3 Dependencies and Building

ElmerFem was build using Spack. For the SCC a special `scc20` release was to be used, as this one contained the required permafrost-solvers for this challenge which are developed outside of the standard ElmerFEM package on the `elmerice` branch. This release was implemented into Spack and used as expected from there.
When building, the `build_type` was set to "Release" and flag `+lua` was provided. GCC version 9.3.0 was used as the compiler and OpenMPI 3.1.5 for the MPI backend.

```
#!/usr/bin/env bash

set -e

# Clone spack repo and checkout SCC20 branch
git clone https://git.wr.informatik.uni-hamburg.de/scc/spack.git
cd spack
git checkout scc20

# Activate Spack stack
cd ..
. spack/share/spack/setup-env.sh

# Install gcc 9.3.0 and add it as a compiler
spack compiler find
spack install gcc@9.3.0+binutils
spack load gcc@9.3.0
spack compiler find

# Install Elmer/ICE
spack clean --all
spack install elmerfem@scc20 ^charmpp build_type=Release +lua
    ↪ ^openmpi@3.1.5 fabrics=ucx
```

Listing 9.1: Install Elmer/ICE with spack

Other dependencies are a BLAS and LAPACK capable library (we have used OpenBLAS) for linear algebra functions, MUMPS for solving of sparse linear algebraic systems.
Building with OpenMP introduced several problems, as the application crashed when running with OpenMP on Mistral. On the Aspire 1 system, we have build the application for the Ubuntu nodes. On these, only OpenMPI worked reliably, which then was used in the submitted run.

9.4 Running

Before running the program, a mesh has to be generated for the desired number of partitions and the header file has to be modified.
Partitions here are parts of the ice sheet that can be feed to single solvers, e. g. MPI processes. Splitting the mesh is done with the following command: To now use the

```
ElmerGrid 2 2 Input_Mesh -metiskway #grid_partitions -out Output_MESH
```

generated partitions, the header file has to be modified, so that the variable `$MESH` contains the same value as the `out` parameter of the `ElmerGrid` command.
Furthermore it was allowed to change existing parameters in the `solversettings.sif` file to tune the run. We did not make changes here.
Please not that the runs do not converge arbitrary partition sizes. 96 partitions was given to be a guaranteed converging run.

9.4.1 Performance Tuning

To determine scaling of the given task, we have run jobs on Mistral to determine a good combination of number of partitions and whether to use hyperthreading or not.
Comparing runs in Figure 9.1, disabling hyperthreading gives a performance improvement with the best performance being around 96 partitions.

9.5 Visualisation

This visualization (Figure 9.2 was done with ParaView. It shows the velocity of the ice sheets as simulated.

9.6 Results

The final submitted result was as follows.

The result was ran on two DGX nodes with 40 cores each, meaning a total of 80 MPI tasks without hyperthreading. In theory this should have produced the best possible

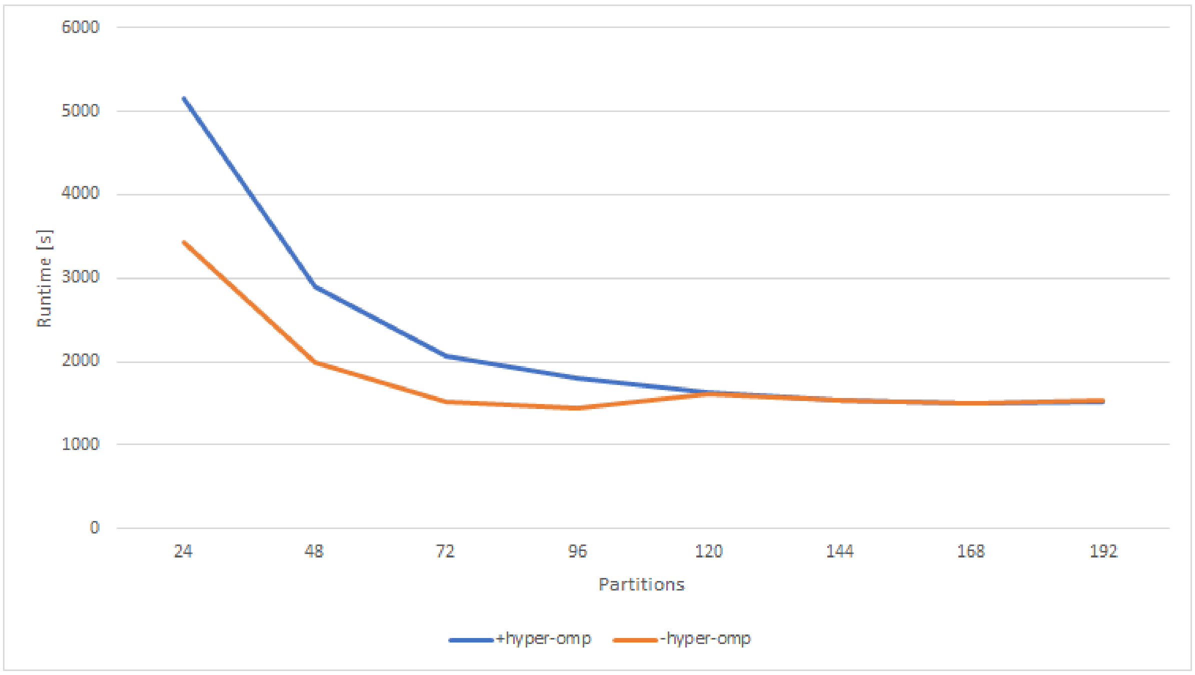


Figure 9.1: Diagram of Elmer runs with different partition sizes and their corresponding runtimes.

result.

With software optimization and changing the source code it would have been possible to gain massive runtime improvements, but we did not find the correct source code lines for this.

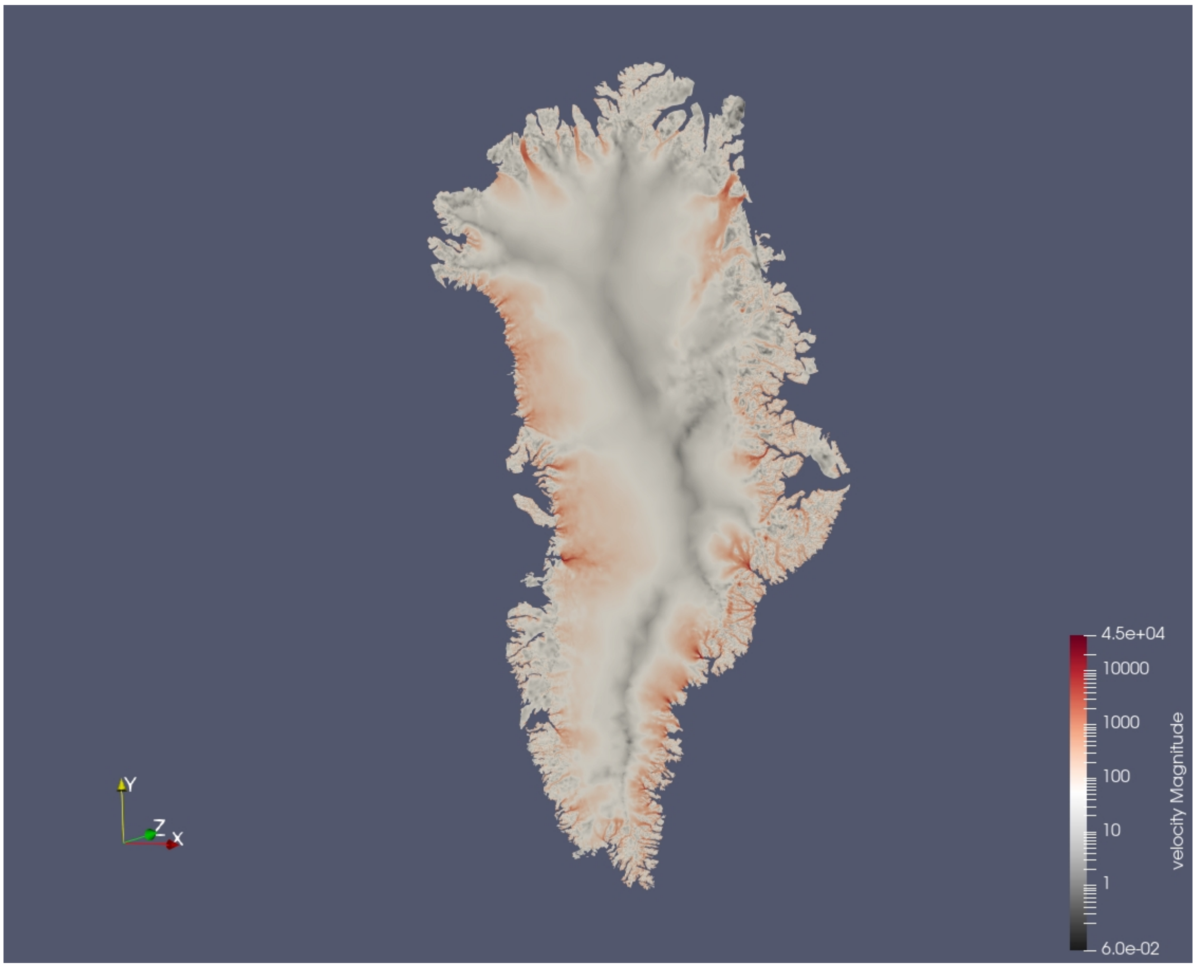


Figure 9.2: Visualization of the simulated velocities of ice sheets on Greenland's landmass

```
SOLVER TOTAL TIME(CPU,REAL):      2714.16     2761.52
```

10 BERT

Author: Daniel Bremer, Lina Meyer

10.1 BERT - Bidirectional Encoder Representations from Transformers

BERT is a language representation model invented by Devlin et al. in 2019 [17]. It is an acronym for Bidirectional Encoder Representations from Transformers, which emphasizes the key techniques BERT uses: a combination of a Transformer and bidirectional training. Bidirectional means that the sentences are not read from the beginning to the end or the other way around, but are processed as a whole [20]. A Transformer or Transfer Learning means a neural network learns to understand texts in general first and afterwards uses it's universal understanding to solve specific natural language processing tasks [37]. One possible task is for example answering questions.
For BERT multiple implementations exists, like ALBERT (A Lite BERT), which reduces parameters and enables better model scaling, or the implementation by the Huggingface team, which was also used in the competition and implements the model directly without model optimizations.
Implementations with TensorFlow and PyTorch are available, which implement the neural network methods for the implementations.

10.1.1 SQuAD - Stanford Question Answering Dataset

The Stanford Question Answering Dataset is a dataset used to test reading comprehension. The dataset consists of questions based on Wikipedia articles which either have an answer, are unanswerable or can not be answered directly, but with a block of text from a text passage in an article.
SQuAD exists in multiple versions. Version 1.1, which we have used in this competition, features over 100000 questions with corresponding answers on a set of over 500 articles. Version 2.0 combines the existing questions from version 1.1 with 50000 more, unanswerable questions. Comparing human performance to the performance achieved by nets answering, nets based on the ALBERT implementation of BERT achieve super-human scores with SQuAD 2.0 [32].

10.2 Background

Since computers exist humans try make them understand our natural language. That is a difficult task as some words have several meanings, sentences have complex structures

and there is infinite number of combinations to link words, so that there is no easy way to create rules about how to read texts [38]. This is where machine learning comes into play. The first feed forward neural net for natural language processing (NLP) was built in 2001 [18]. Since then, neural networks were developed in a lot of directions. The most significant areas for BERT are context and transfer learning. That is why we will take a closer look at them.
There are some models that use context and some that do not. As the word says contextual models do not just look at one word, but also at the surrounding words. So that a contextual model would recognize the different meaning of “bank” in “river bank” and “bank deposit” while a non-contextual would not. BERT of course belongs to the contextual models and is even a bidirectional one, so the words before as well as the words after the word we are looking at are taken into account [3].
The second field are Transformers, which were mostly used in the field of image recognition at first. For example researchers trained nets on ImageNet and then used the trained network to distinct between a certain set of pictures. Transformers are deep learning models, which basically take an sequential data input and transform it to an output. In the field of NLP Transformers are first used in 2017 by researchers at Google and the University of Toronto [37]. Then this strategy was also integrated in BERT. When it was published BERT got the highest f1-score on the SQuAD leaderboard [33].

10.3 Architecture

BERT’s training has two stages: pre-training and fine-tuning. On the first stage BERT develops a general language model using unsupervised learning. The net learns on huge text collections like Wikipedia.
The techniques used to “teach a language” are masked language modelling (LM) and next sentence prediction. For Masked LM 15 % of all words in the text are being masked and the net’s task is it to predict the this word by reading every other word in that sentence. The net’s output is probability of being behind the mask for every existing word. This is what makes BERT deeply bidirectional, because the whole sentence goes into the analysis and not just the part before or after the mask [20].
Next sentence prediction does exactly what the name suggests: predicting the next sentence. More precisely, the network gets two sentences as an input and then has to guess whether they belong together or are just randomly combined. These sentence pairs are build with 50 % probability of being next to one another originally or not [20].
The step of pre-training takes a lot of time and resources. That is why there a some pre-trained BERT networks available for download. They exist in several languages, network sizes and taking into account upper-case or not [16].
For the next step you can either use your self-trained network or one of the pre-built and train it with any specific natural language task, for example sentence classification or in our case question answering. Fine-tuning requires another data set with task specific data like SQuAD. Now the network should learn to mark the answer to a question in a specified text sequence. SQuAD contains more than 100000 question answer pairs we

can “feed” the network with [32]. For this the architecture has to be changed very little, only some pre- and post-processing steps are necessary [3].

10.4 Assignment

This year’s AI challenge consisted of the SQuAD v1.1 benchmark. For training the “BERT-Base, Uncased” model was used.
Several limitations were given to narrow the fields of performance optimization. First of all the HuggingFace implementation for BERT was set as the standard. The model’s architecture was not to be modified, so it was allowed to alter optimizers and learning rate schedulers and other techniques, as long as this did not have deeper changes as a consequence. Also it was disallowed to freeze layers, so with every training the whole model had to be trained.
Later on, even more rules were specified: It was not allowed to add the dev-set into the training data, as well as it was disallowed to integrate any other additional data. Hyper-parameters we allowed globally, so learning rate schedulers, optimizers, drop-out strategies, etc. were free to be used as desired. Also random seeding was enabled, allowing to reduce variance between training runs.
As a result, the f1 score was used for team ranking. The f1 score shows the balance between recall and precision of a model. Recall is given by

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{10.1}$$

which gives an estimation how many actual Positives can be found in positively labeled data, while precision is defined by

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{10.2}$$

and gives information on how many predicted positives are actually positives
f1 the is calculated with the harmonic mean of recall and precision

$$f1 = \left(\frac{Recall^{-1} + Precision^{-1}}{2}\right)^{-1} \quad . \tag{10.3}$$

The f1 score heavily weights False Negatives and False Positives, therefore gives a good estimation of “wrongness” [35] [24].
For the final evaluation, 5 runs had to be handed in, where the top 3 f1 scores were averaged. Predictions were done on the dev-set `dev-v1.1.json`, which is used with `evaluate-v1.1.py`. A unseed json with questions was also given, to be used as a tie-breaker [28].

10.5 Building

Different code bases were possible starting points for this application. We have focussed on the code that is delivered with the NVIDIA Deep Learning Examples[1], an BERT implementation on the basis of TensorFlow 2[2] and the Lambdal implementation [3]. NVIDIA's Deep Learning Examples focus very much on containerized running of applications. For better performance, we have tried to pull all required scripts and programs out of the container to run them natively, but from the source codes, no NVIDIA specific tuning was to be found.
To compare the performance between TensorFlow 1 and TensorFlow 2, we were not satisfied with the outputs of the TF2 implementation. For that reason, and for its native support for Horovod, we have decided to use the Lambdal code base.
To build BERT you have to clone Lambdals repo https://github.com/lambdal/bert and then download the pre-trained BERT-base and the SQuAD data set like in listing Listing 10.1

```
# download bert base
wget https://storage.googleapis.com/bert_models/2018_10_18/
    ↪ uncased_L-12_H-768_A-12.zip
# download squad
wget
    ↪ https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v1.1.json
wget https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v1.1.json
wget
    ↪ https://github.com/allenai/bi-att-flow/blob/master/squad/evaluate-v1.1.py
```

Listing 10.1: Download BERT base and SQuAD

Lambdal's BERT uses Horovod to train the net across multiple GPU's with the help of openmpi [3]. Horovod only works with some versions of OpenMPI for example 3.0.0 and we had to use a special version of GCC, because many versions conflicted on the GPU nodes.

```
# install and load openmpi
spack install openmpi@3.0.0
spack load openmpi@3.0.0
# install and load gcc
spack install gcc@4.9.0 +binutils
spack load gcc@4.9.0
```

Listing 10.2: Build dependencies

For CUDA and CuDNN libraries, we have settled on versions 10.0.130 for CUDA and 7.4.1 for CuDNN. Although CUDA 11 was released shortly before the competition start, the driver version present on the Aspire 1 cluster limited our choice to this specific version, being the most recent compatible with the drivers.

[1]https://github.com/NVIDIA/DeepLearningExamples
[2]https://github.com/kpe/bert-for-tf2

We then decided to run BERT in a python virtual environment which allows better control of installed packages and built all necessary dependencies as shown in listing Listing 10.3. To run TensorFlow on GPU you can use the package `tensorflow-gpu` and version 1.15.0 is needed for `horovod`.

```
# load python
spack load /dtyxxve
# create a virtual environment
python -m venv envname
# activate virtual environment
. ./envname/bin/activate
# install dependencies
pip install numpy
pip install six
pip install mxnet
pip install tensorflow-gpu==1.15.0
HOROVOD_WITH_TENSORFLOW=1 pip install --no-cache-dir
   ↪ horovod[tensorflow]
```

Listing 10.3: Set up python virtual environment with all dependencies

10.6 Running

Running BERT includes two steps: training on SQuAD and evaluating the scores. You can start the training with run_squad.py as shown in listing Listing 10.4 with several parameters. There are a lot of files that have to be specified and their names are self-explaining. The most important parameters we had to adjust were batch size, learning rate and number of epochs. With `do_predict` and `do_train` you can decide whether the net should be trained and BERT should predict answers to the questions.

```
python run_squad.py \
    --vocab_file=$BERT_BASE_DIR/vocab.txt \
    --bert_config_file=$BERT_BASE_DIR/bert_config.json \
    --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
    --do_train=True \
    --train_file=$SQUAD_DIR/train-v1.1.json \
    --do_predict=True \
    --predict_file=$SQUAD_DIR/dev-v1.1.json \
    --train_batch_size=12 \
    --learning_rate=3e-5 \
    --num_train_epochs=2.0 \
    --max_seq_length=256 \
    --doc_stride=128 \
    --output_dir=output-run1/
```

Listing 10.4: Run SQuAD

Parameters that we also altered were `learning_rate` and `num_train_epoch`, which modified learning speed and the number of training epochs on the data. When trying to

increase `train_batch_size` for faster learning runs, we very quickly ran into memory limit issues, which is why this parameter was left untouched.
Evaluation was done by running `evaluate-v1.1.py` against the dev-set and the prediction file acquired from the predictions, when using `-do_predict==True` in `run_squad.py`

10.6.1 Performance Tuning

To reduce variance between runs, one can introduce random shuffling of the input data. This was done by changing the randomiser in `run_squad.py`. At first it only got initialized with `rng = random.Random(12345)`, which made the randomness "static", not being random at all. By switching this with `rng = random.Random(datetime.now())`, a different seed was used with every run, enabling "better" randomness.

10.7 Results

The results handed in were as follows:

Run	Learning Rate	Epochs	Score
1	3e-5	2	f1: 87.97936255542666
2 (random shuffling)	3e-5	2	f1: 88.28141595606796
3	2e-5	3	f1: 87.99612501421532
4	3e-5	3	f1: 88.14213130502958

Run 1 was a run with default parameters, while runs after Run 2 implemented the random shuffling changes, or run parameters.

11 Gromacs

Author: Lina Meyer

Gromacs is a a tool for molecular dynamics simulations. It can be used to simulate millions of atoms with different forces, non-bonded as well as bonded. Gromacs is extremely fast compared to other molecular dynamics simulations due to several performance optimisations and its GPU plus CUDA support. Usually is it used to compute simulations of biochemical molecules, e.g. proteins, lipids and nucleic acids [27].

11.1 Algorithm

The simulation works as follows: The first step is the initialisation of the particles. All of them have to be brought to the right spots and the velocities have to be adjusted. Additionally, the potential of the forces is initialized and after that the forces themselves are calculated. The next step consists of calculating the movement of the atoms by solving the Newton's Equation of Motion with the forces from the previous step. In the end, the particle positions are updated, also taking into account some restraints like boundary conditions. Afterwards the second and third step are repeated as often as the user wants to run the simulation. When all steps are completed, Gromacs outputs end positions, velocities, energies, temperature or other produced data [26]. In figure Figure 11.1 the main steps of this simulation are shown schematically, again.

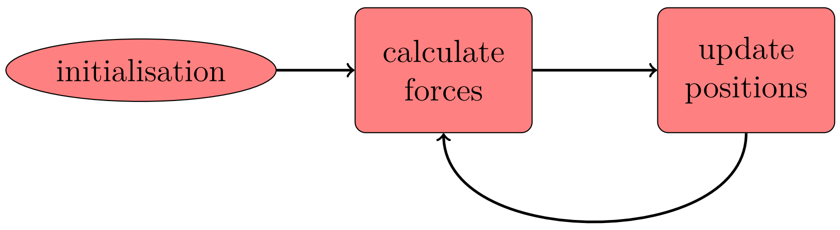


Figure 11.1: Schematic representation of general steps of the algorithm

In Gromacs distinguishes three different kinds of forces:

- Non-bonded
  Particles in non-bounded systems in certain ranges interact with each other, because the forces have a short range.

- PME (Particle Mesh Ewald)
  Non-bonded particles in big ranges interact with each other due to long range forces. The calculations are performed with Fourier Transformations.

- Bonded
  Bonded forces only act on direct neighbors (up to four) of an atom if they are bonded covalently.

Since 2019 Gromacs is able to calculate all of the forces on GPUs, but updating the positions only worked on CPUs. That is why a lot of data had to be transferred from GPU to CPU and back via PCIe. To boost the performance Gromacs' version 2020 features position update on GPUs, so that the data does not have to be transferred anymore. When using multiple GPUs, they communicate via NVLink which is significantly faster than PCIe [19].

11.2 Assignment

The task was is to run Gromacs with two input files: lignocellulose and STMV. Lignocellulose is the woody material plants are made of and can be used for the production of bio fuel [34]. The letters STMV stand for Satellite Tobacco Mosaic Virus, which is a plant virus dependent on the host Tobacco Mosaic Virus [2]. We had to run each of the simulations with 100000 steps and then try to decrease the run time [29]. As a measure for time we used the period the particles were simulated (in nanoseconds) divided by the time needed for running the simulation (in days).

11.3 Dependencies

Gromacs has some optional dependencies: It supports MPI, GPUs via CUDA and Fast Fourier Transformation (FFT). For simulations on a single node Gromacs has a build-in thread-MPI parallelizing the program over multiple cores. If you want to run Gromacs on more machines, you have to build it with external MPI, e.g. OpenMPI or MPICH. For additional parallelization OpenMP is automatically added during compilation. As Gromacs does a lot of Fourier transformations during simulations you need library support for FFT, for example FFTW or MKL. The fallback FFTPACK is only recommended if the performance is unimportant. We chose FFTW as the documentation suggests it is the fastest. On NVIDIA GPUs complex calculations can be done with CUDA [26].

11.4 Building

As you can see in the code in Listing 11.1, we build Gromacs 2020.1 with spack on our own stack. To install Gromacs with CUDA support, we had to use gcc version 7.4.0, because that is the latest compatible version with CUDA 10.0.130. Additionally, we used Open-MPI 3.1.5.

```
#!/usr/bin/env bash

set -e

```

```
# Clone spack repo and checkout SCC20 branch
git clone https://git.wr.informatik.uni-hamburg.de/scc/spack.git
cd spack
git checkout scc20

# Activate Spack stack
cd ..
. spack/share/spack/setup-env.sh

# Install gcc 9.3.0 and add it as a compiler
spack compiler find
spack install gcc@9.3.0+binutils
spack load gcc@9.3.0
spack compiler find

# Install GCC 7.4.0 for CUDA
spack install gcc@7.4.0+binutils%gcc@9.3.0

# Install CUDA and GROMACS
spack install cuda@10.0.130%gcc@7.4.0
spack install gromacs%gcc@7.4.0 +cuda ^openmpi/hxteols ^cuda@10.0.130
```

Listing 11.1: Install Gromacs with spack

11.5 Running

To run the simulation you have to start `gmx_mpi mdrun` with an input file in tpr format containing structure, topology and molecular dynamics parameters. Additionally, there are several parameters that can be adjusted:

`-s`	input file
`-steps`	number of steps, had to be 100000
`-nstlist`	number of steps after the neighbourhood is updated again
`-nb`	non-bonded forces, on gpu or cpu
`-bonded`	bonded forces, on gpu or cpu
`-pme`	pme forces, on gpu or cpu
`-ntmpi`	number of MPI processes
`-ntomp`	number of OMP threads

To run Gromacs we used the dgx nodes, because they had GPUs. In our runs we set `-nb` and `-bonded` to `gpu` to run all forces on GPU instead of CPU. `-pme gpu` was not working on the dgx nodes. We also had to set some environment variables to activate all GPU features of the Gromacs 2020.1 version. To get the optimal results we varied the `-nstlist` parameter. In the code below you can see an example run:

```
export GMX_GPU_DD_COMMS=true
export GMX_GPU_PME_PP_COMMS=true
export GMX_FORCE_UPDATE_DEFAULT_GPU=true
```

```
gmx_mpi mdrun -s lignocellulose-rf.tpr -nsteps 100000 -nstlist 75
    ↪ -ntomp 40 -nb gpu -bonded gpu
```

Listing 11.2: Running Gromacs

Gromacs automatically used 40 OMP threads, as the node has 40 cores, and one MPI process. Unfortunately, we could not manage to use more MPI processes, which would have led to a huge performance boost [26]. We tried to build Gromacs with external MPI (mpich and multiple versions of open mpi) and also with the provided internal MPI setting the cmake option `DGMX_MPI=on`, but both options did not work on the NSCC-Cluster.

11.6 Visualisation

The visualization was not a part of the challenge, but we visualized one of the input files just to know what we are working on. In figure Figure 11.2 you can see the structure of lignocellulose.

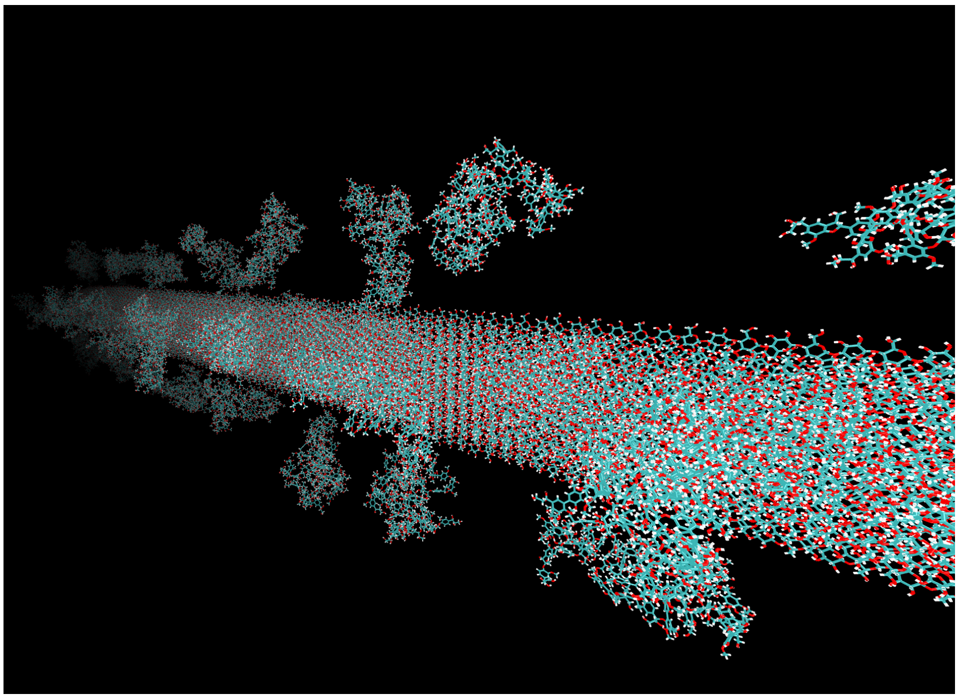


Figure 11.2: Visualisation of lignocellulose made with vmd

11.7 Results

Our main focus while benchmarking Gromacs was the parameter `-nstlist`. In the following diagram the performance measured in ns/day is plotted against the value of `-nstlist`.

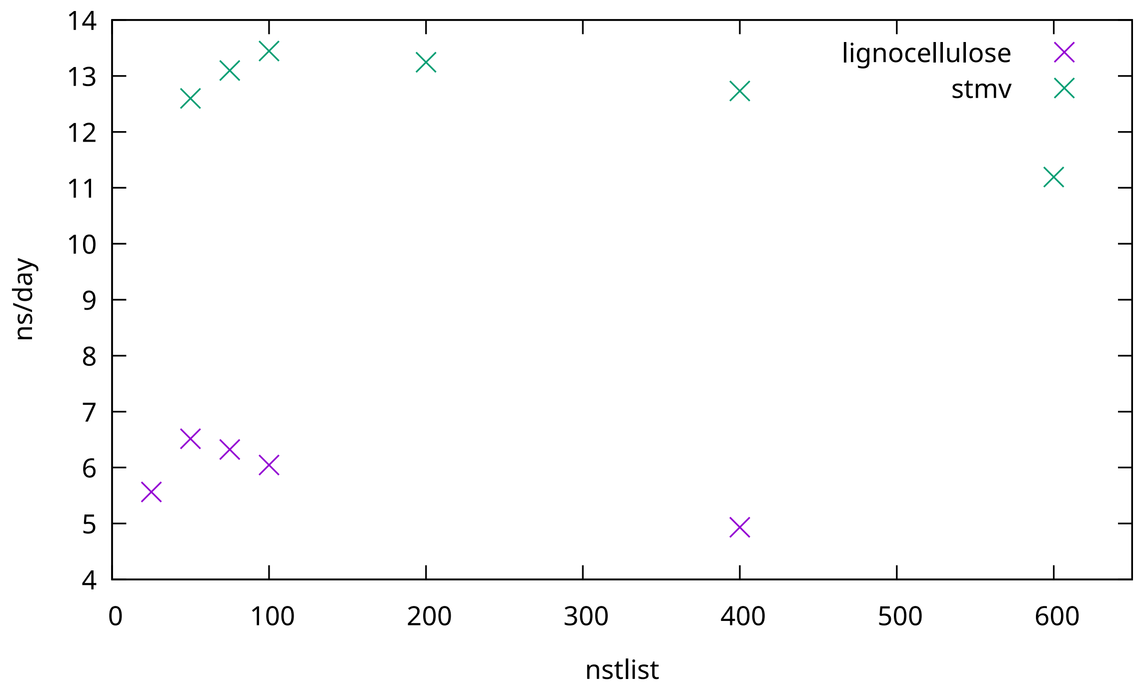


Figure 11.3: Correlation of performance and nstlist parameter

Our best results are:

- lignocellulose: 1 GPU, 40 OpenMP threads, nstlist=50
  - 6.515 nanoseconds per day
  - walltime: 2652.505 seconds
- stmv: 1 GPU, 2 GPU tasks, 40 OpenMP threads, nstlist=100
  - 13.243 nanoseconds per day
  - walltime: 1304.85 seconds

12 Aurora

Author: Roland Fredenhagen

Due to the special circumstances we were unable to use the Aurora Vector Computers in the actual Competition, but before the world fell into chaos there was a workshop at the RWTH Aachen. Introducing the principle of Vector Engines, possible use cases as well as a programming tutorial with a hands on.

12.0.1 Vector Engine

Vector engines are optimized for highly parallel calculation workflows that can the implemented using vector algorithmic. The Vector Engine Processor is built up from a vector register, comparable to the register of a normal CPU only specialized for storing high dimensional vectors and a Vector Processing Unit able to compute these vectors in parallel, as shown in Figure 12.1.

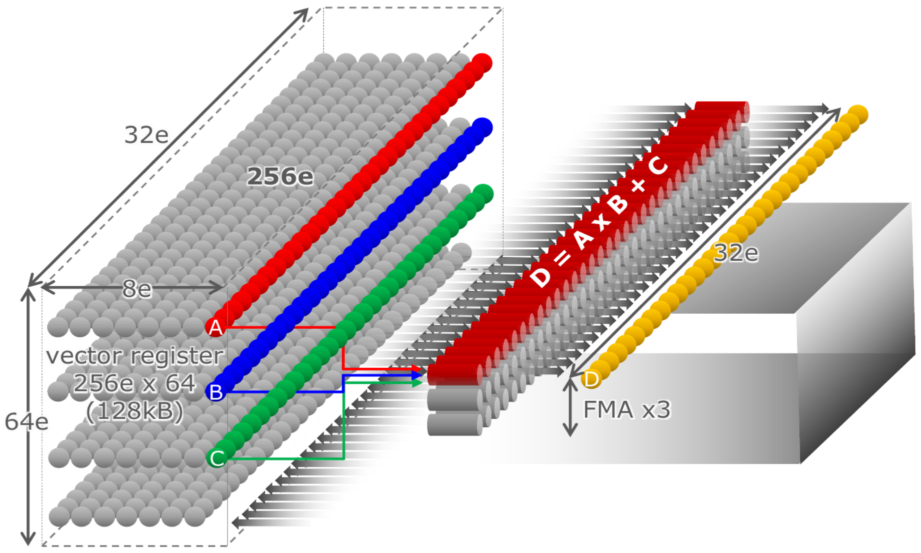


Figure 12.1: Visualisation of vector computation

This is comparable to GPGPUs, but with added benefits as illustrated in Figure 12.2 It avoids PCIe bottleneck as the whole application is executed on the Vector Engine, it has a larger memory of 48GB on the current generation models and can run a standard languages with compilers for FORTRAN or C/C++.
It is also possible to combine scalar an vectorized calculations either using the Vector Engine as an accelerator in an x86 application or the opposite using the x86 CPU to accelerate scalar calculations while running the main application on the Vector Engine.

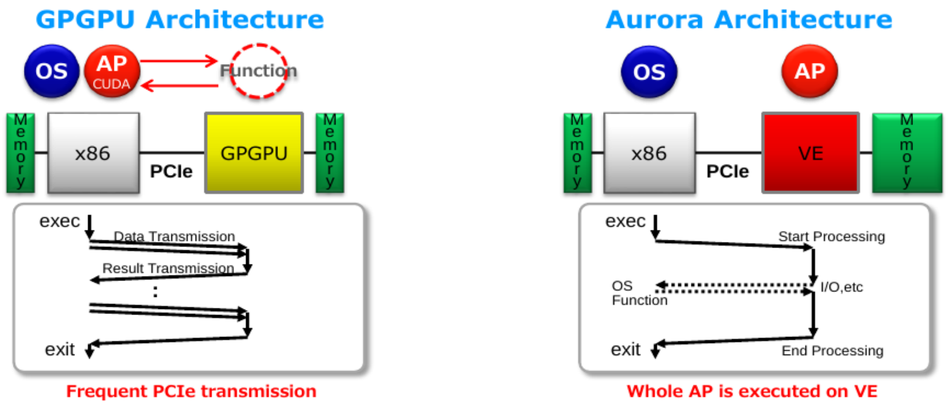


Figure 12.2: GPGPU vs Aurora Architecture Comparison

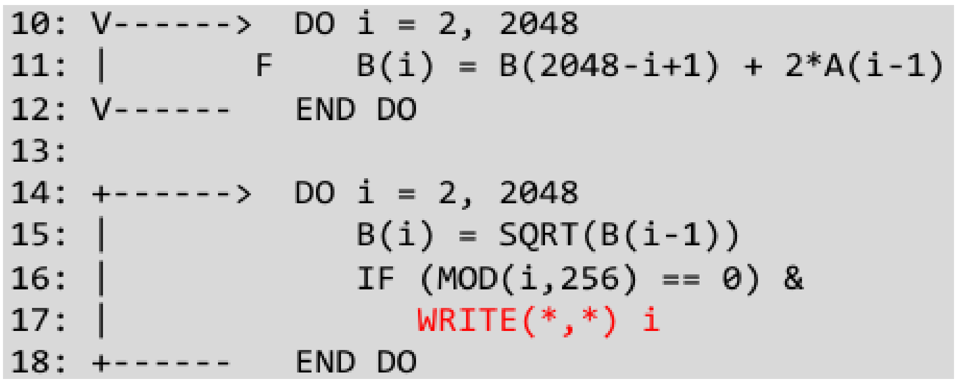
12.0.2 Usage

Tools

Writing applications for Vector Engines is supported with many standard tools with gdb or PROGINF and libraries as OpenMP or MPI.
The vectorization and parellelization can be automated using a Vector Cross Compiler like ncc and can be controlled using compiler directives telling the compiler what should be done at specific places in the source code:

`!NEC\$ vector`	allows vectorization
`!NEC\$ novector`	disallows vectorization
`!NEC\$ vreg(array-name)`	Assigns array "array-name" to vector registers

This can be monitored using Format List File generated by the compiler as shown in Figure 12.3.



```
10: V------>  DO i = 2, 2048
11: |       F    B(i) = B(2048-i+1) + 2*A(i-1)
12: V------   END DO
13:
14: +------>  DO i = 2, 2048
15: |            B(i) = SQRT(B(i-1))
16: |            IF (MOD(i,256) == 0) &
17: |               WRITE(*,*) i
18: +------   END DO
```

Figure 12.3: Format List File: the first loop can be factorized (V), the second is not due to the I/O Operation (+)

This helps identify errors fast, showing in detail what the compiler did at which part of the code.
Examples for loop Transformations:

```
+------>   Loop is not vectorized

V------>   Loop is vectorized

U------>   Loop is unrolled

X------>   Nested loops are interchanged and vectorized
|*----->
```

Examples for special instructions:

`I`	A function call is inlined
`M`	Nested loop is replaced by matrix-multiply routine
`G`	Vector gather memory operation

Programming

For this to work it is important that the structures are vectorizable ie. loop count needs to be knows upon entering the loop otherwise the compiler cannot distribute the loop correctly and data needs to be parallel ie. the order of operation must not matter, that means $A(i) = A(i-1) + B(i)$ cannot be vectorized. On top of that no complicated function calls are allowed only functions that can be inlined. Some data structures as strings ar non-vectorizable as well.
For optimal performance, memory access needs to be taken int account as well, optimally each index only depends on its corresponding cache as shown in Figure 12.4, inefficient memory access as reduction, scatter or gather Figure 12.5, should be avoided or reduced when possible.

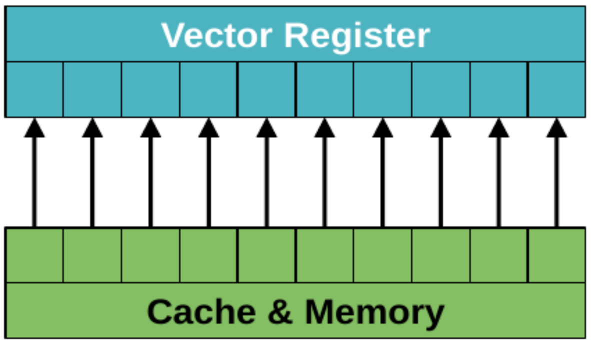


Figure 12.4: Optimal Memory Access (Stride 1) $A(i) = B(i)$

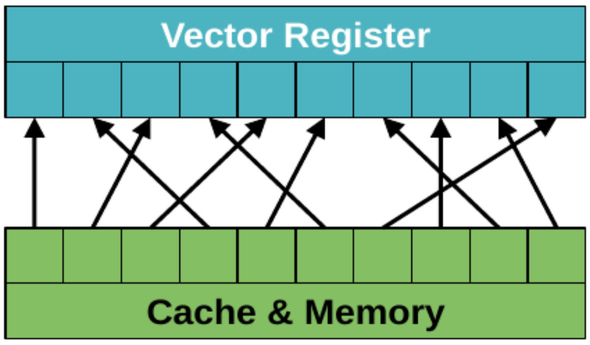


Figure 12.5: Inefficient Random Memory Access (Gather) $A(i) = B(idx(i))$

Bibliography

[1] *AI_System_QuickStart.pdf.* URL: https://help.nscc.sg/wp-content/uploads/AI_System_QuickStart.pdf (visited on 09/20/2020).

[2] Anton Arkhipov et al. *Molecular Dynamics of Viruses.* http://www.ks.uiuc.edu/Research/STMV/. May 13, 2013. (Visited on 09/16/2020).

[3] Stephen A. Balaban. *bert.* Feb. 7, 2019. URL: https://github.com/lambdal/bert (visited on 10/05/2020).

[4] Josh Barnes and Piet Hut. "A hierarchical O(N log N) force-calculation algorithm". In: 324.6096 (Dec. 1986), pp. 446–449. DOI: 10.1038/324446a0.

[5] Mikhail Brinskiy and Mark Lubin. "An introduction to MPI-3 shared memory Programming". In: *Available https: l/software. inteLcomlenus* (2017).

[6] *ChaNGa Benchmarks – N-BodyShop/changa Wiki.* URL: https://github.com/N-BodyShop/changa/wiki/ChaNGa-Benchmarks (visited on 09/21/2020).

[7] *ChaNGa Challenge – HPC-Works – HPCAC Technical Community.* URL: https://hpcadvisorycouncil.atlassian.net/wiki/spaces/HPCWORKS/pages/1182171137/ChaNGa+Challenge (visited on 09/21/2020).

[8] *ChaNGa Wiki on Github.* URL: https://github.com/N-BodyShop/changa/wiki/ChaNGa (visited on 09/19/2020).

[9] *Charm++: Documentation.* URL: https://charm.readthedocs.io/en/latest/charm++/manual.html#basic-charm-programming (visited on 09/28/2020).

[10] *Charm++: Tutorial.* URL: http://charmplusplus.org/tutorial/CharmConcepts.html (visited on 09/28/2020).

[11] *Charm++: Tutorial.* URL: http://charmplusplus.org/tutorial/CharmComponents.html (visited on 09/28/2020).

[12] Wikimedia Commons. *File:Barnes hut partikel.png — Wikimedia Commons, the free media repository.* [Online; accessed 20-September-2020]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Barnes_hut_partikel.png&oldid=462896270.

[13] Wikimedia Commons. *File:Barnes hut tree.png — Wikimedia Commons, the free media repository.* [Online; accessed 20-September-2020]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Barnes_hut_tree.png&oldid=462896306.

[14] Wikimedia Commons. *File:Barnes hut used nodes.png — Wikimedia Commons, the free media repository.* [Online; accessed 20-September-2020]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Barnes_hut_used_nodes.png&oldid=462896326.

[15] CSC. *Elmer - Elmer - CSC Company Site.* URL: https://www.csc.fi/web/elmer (visited on 10/06/2020).

[16] Jacob Devlin et al. *bert.* Mar. 11, 2020. URL: https://github.com/google-research/bert (visited on 10/05/2020).

[17] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* 2019. arXiv: 1810.04805 [cs.CL].

[18] Keith D. Foote. *A Brief History of Natural Language Processing (NLP).* May 22, 2019. URL: https://www.dataversity.net/a-brief-history-of-natural-language-processing-nlp/ (visited on 10/05/2020).

[19] Alex Gray. *Creating Faster Molecular Dynamics Simulations with GROMACS 2020.* Feb. 25, 2020. URL: https://developer.nvidia.com/blog/creating-faster-molecular-dynamics-simulations-with-gromacs-2020/ (visited on 10/07/2020).

[20] Rani Horev. *BERT Explained: State of the art language model for NLP.* Nov. 10, 2018. URL: https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270 (visited on 10/05/2020).

[21] *HPC Advisory Council ISC 2020 Student Cluster Competition – Benchmarking.* URL: http://www.hpcadvisorycouncil.com/events/2020/student-cluster-competition/benchmarking.php (visited on 09/20/2020).

[22] Chao Huang, Orion Lawlor, and L. V. Kalé. "Adaptive MPI". In: *Languages and Compilers for Parallel Computing.* Ed. by Lawrence Rauchwerger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322. ISBN: 978-3-540-24644-2.

[23] Chao Huang et al. "Performance evaluation of adaptive MPI". In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2006, pp. 12–21.

[24] Purva Huilgol. *Accuracy vs. F1-Score.* Aug. 24, 2019. URL: https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2 (visited on 10/05/2020).

[25] P. Jetley et al. "Massively parallel cosmological simulations with ChaNGa". In: *2008 IEEE International Symposium on Parallel and Distributed Processing.* 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536319.

[26] Lindahl et al. *GROMACS 2020.3 Manual.* Version 2020.3. July 2020. DOI: 10.5281/zenodo.3923644. URL: https://doi.org/10.5281/zenodo.3923644.

[27] mabraham. *About GROMACS.* http://www.gromacs.org/About_Gromacs. Sept. 24, 2018. (Visited on 07/11/2020).

[28] Ophir Maor. *AI Challenge - SQuAD 1.1 with BERT-Base Guidelines*. June 9, 2020. URL: https://hpcadvisorycouncil.atlassian.net/wiki/spaces/HPCWORKS/pages/1326612594/AI+Challenge+-+SQuAD+1.1+with+BERT-Base+Guidelines (visited on 10/05/2020).

[29] Ophir Maor. *Gromacs Challenge*. June 9, 2020. URL: https://hpcadvisorycouncil.atlassian.net/wiki/spaces/HPCWORKS/pages/1453719580/Gromacs+Challenge (visited on 10/07/2020).

[30] *NSCC Software/Hardware Information*. URL: https://help.nscc.sg/softwarehardware-information/ (visited on 09/20/2020).

[31] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 2009, pp. 427–436.

[32] Pranav Rajpurkar et al. *The Stanford Question Answering Dataset*. URL: https://rajpurkar.github.io/SQuAD-explorer/ (visited on 10/05/2020).

[33] Pranav Rajpurkar et al. *The Stanford Question Answering Dataset*. URL: https://rajpurkar.github.io/SQuAD-explorer/ (visited on 12/01/2018).

[34] K Sanderson. "Lignocellulose: A chewy problem". In: *Nature* (July 22, 2011).

[35] Koo Ping Shung. *Accuracy, Precision, Recall or F1?* Mar. 15, 2018. URL: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9 (visited on 10/05/2020).

[36] *University of Washington N-Body Shop Home Page*. URL: http://faculty.washington.edu/trq/hpcc/ (visited on 09/19/2020).

[37] Sarthak Vajpayee. *Transformers (State-of-the-art Natural Language Processing)*. Aug. 6, 2020. URL: https://towardsdatascience.com/transformers-state-of-the-art-natural-language-processing-1d84c4c7462b (visited on 10/05/2020).

[38] Diego Lopez Yse. *Your Guide to Natural Language Processing (NLP)*. Jan. 15, 2019. URL: https://towardsdatascience.com/your-guide-to-natural-language-processing-nlp-48ea2511f6e1 (visited on 10/05/2020).