Student Cluster Competition 2015

— Report —

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

Vorgelegt von: Lennart Braun, Jonas Gresens, Jannek Squar, Lars Thoms, Felix Wiedemann
Betreuer: Dr. Michael Kuhn, Anna Fuchs

Hamburg, den 10.1.2016

Abstract

The International Supercomputing Conference (ISC[1]) is an annual global conference and exhibition for High Performance Computing. New and established companies in the High Performance Computing field are able to present their products and discuss the needs of their potential customers. Additionally, several workshops are held to communicate the knowledge gained in research groups. One major event is the announcement of the new TOP500 list of the 500 best performing supercomputers world-wide.

Another event at the ISC is the Student Cluster Competition (SCC) where student teams from different universities from around the world come together and compete for the best performing cluster system. This report summarizes the experiences of the team from the University of Hamburg at the SCC of ISC'15.

[1] http://www.isc-events.com/isc15/

Contents

1 Introduction and Motivation **4**
- 1.1 Task and Rules . . . 5
- 1.2 Bay . . . 5

2 System **6**
- 2.1 Hardware Overview . . . 6
- 2.2 Power Consumption . . . 8
- 2.3 Power Capping . . . 9
- 2.4 Operating System . . . 10
- 2.5 Compiler . . . 12
- 2.6 Filesystem . . . 13
- 2.7 Modules . . . 21

3 Libraries **25**
- 3.1 Math Libraries . . . 25
- 3.2 I/O Libraries . . . 33
- 3.3 MPI . . . 35

4 Benchmarks **41**
- 4.1 HPL . . . 41
- 4.2 HPCC . . . 43
- 4.3 Graph500 . . . 51

5 Applications **53**
- 5.1 PyFR . . . 53
- 5.2 Octopus . . . 60
- 5.3 LAMMPS . . . 64

Bibliography **68**

List of Figures **69**

List of Tables **70**

List of Listings **71**

A Additional Listings **72**

1. Introduction and Motivation

In July 2015, the 4th HPCAC-ISC Student Cluster Competition[1] during the ISC'15 Conference and Exhibition took place in Frankfurt.

(a) ISC 2015

(b) HPC Advisory Council

It was the second time that a team of the University of Hamburg takes part in the SCC. We were participating to gain experience, both technically and personally. We wanted to prove and – of course – improve our skills and were looking forward to working together as a team.

Figure 1.1.: Setup day in our bay

The Student Cluster Challenge was a very good opportunity to get to know the international HPC scene, which is not easily possible for students. It was also a good chance to compare our own methods with the other teams. Additionally, it was very interesting to have the chance to work with this kind of hardware and resources, because they are generally not easily accessible.

[1] http://www.hpcadvisorycouncil.com/events/2015/isc15-student-cluster-competition

1.1. Task and Rules

The 11 teams, each consisting of 6 undergraduate students from all over the world, built a small prepared cluster in a real time competition. They tried to reach best performance results with their systems under certain conditions. There were three categories of awards to be given - three prizes for Overall Winners, the highest LINPACK performance and the Fan Favorite prize.

The power budget was limited to 3 kW on one monitored PDU. Exceeding this budget was penalized down to disqualification from the competition or parts of it. It was not allowed to restart the system or its components and also to change any of them at any time of the competition, save as otherwise permitted. The advisors were not allowed to actively support the team during the day.

Three applications were published some months in advance as well as the used benchmarks were known from the beginning of the competition. One addition application was published on the immediately preceding day.

1.2. Bay

Author: Lars Thoms

Each team is allocated to a bay of $3 \cdot 3$ qm, which means efficient usage of space. All six members with chairs, one table, routers, power strips, cables, nodes, and advertising material have to be accommodated. As you see in Figure 1.2, we used the cardboard packaging of our two rackmounts as a table for two nodes, a head node, both switches (InfiniBand and Ethernet), and a netbook connected to a screen which displays our current power consumption and CPU usage. Furthermore the DKRZ lent us a thermal camera. We used it to show visitors the thermal emission of the nodes.

Figure 1.2.: Our bay

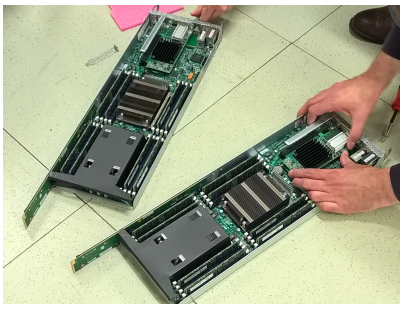
2. System

2.1. Hardware Overview

Author: Lars Thoms

At the Student Cluster Competition our hardware were sponsored by *Bull* (nodes and Ethernet switch) and *Mellanox* (InfiniBand switch and cards). The final setup consists of eight compute nodes, four per rackmount, and each one consists of:

- 2 Xeon E5-2680v3 (12 Cores, 2.5 GHz)
- 128 GB RAM (16 GB modules)
- 240 GB (SAMSUNG MZ7WD240)
- Mellanox ConnectX®-4 VPI adapter card, EDR IB (100 Gb/s) and Gigabit Ethernet, single-port QSFP28, PCIe3.0 x16



(a) Two of our nodes

(b) Front view of our eight nodes

Furthermore, we used a *Mellanox Switch-IB(TM) based EDR InfiniBand 1U Switch with 36 QSFP28 ports* as an InfiniBand switch but we only needed eight of the 32 ports, therefore the power consumption of this hardware was very high. Additionally, we had a 32 port Ethernet switch from Cisco (*Cisco SG300-28*).

As a head node (called microwave), we used small box which contained two Western Digital Green 1 TB as RAID 1, two Gigabit Ethernet slots, an Intel Atom quad core and 16 GB RAM.

2.1.1. Thermal Emissions

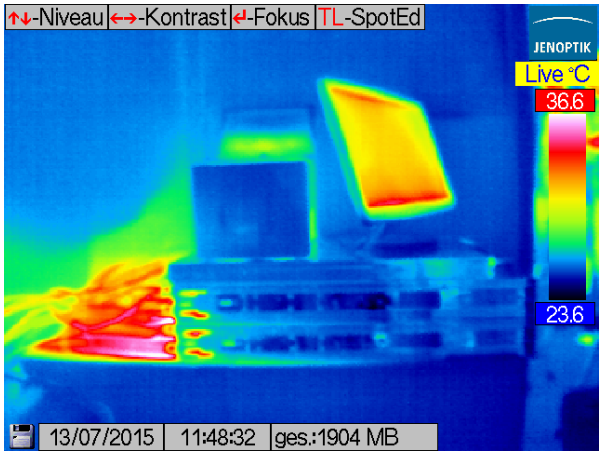
Author: Lars Thoms

Additionally, we also monitored the thermal emissions of our nodes. We borrowed a Janoptik VarioCAM from the DKRZ. The camera has two IP54 connectors (figure 2.1b [1]) and a slot for an SD card (FAT formatted). First of all, we evaluated the temperature of our nodes and calibrated the camera while we took photos (see figures 2.1a and 2.1b). After that we connected the power cord and a VGA to IP54 connector to display emissions on a screen (figure 2.1a). It is important to change the color model on the camera, otherwise you see a greenish image.

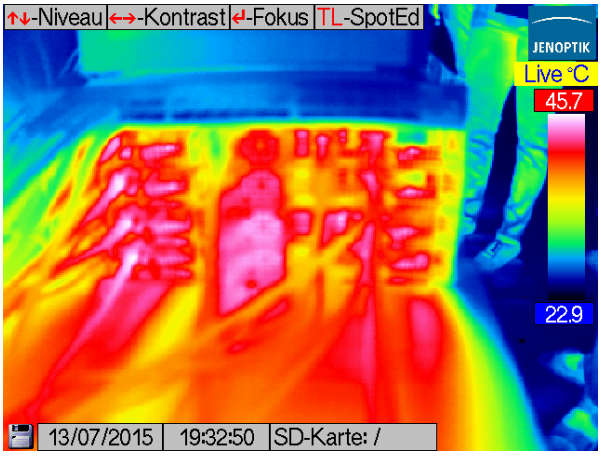
(a) Setup of the camera

(b) IP54 connector

Interestingly, our nodes have never emit more than 45 °C – even under high load (figure 2.1b)! Presumably, our nodes from Bull have a good airflow.



(a) Side view of the nodes



(b) Back view of the nodes

[1] `http://catalogue.techno-test.com/products/images/780/580/81920/produits/VarioCAM_HiRes_384/5-VarioCam_HiRes_inspect_384.jpg`

2.2. Power Consumption

Author: Lars Thoms

The event organizer generated an overall graph which shows the power consumption of every team over time (figure 2.1). But this graph was unusable for measuring our own power consumption, because the sampling was too inaccurate and the resulting image too small to see the exact values.

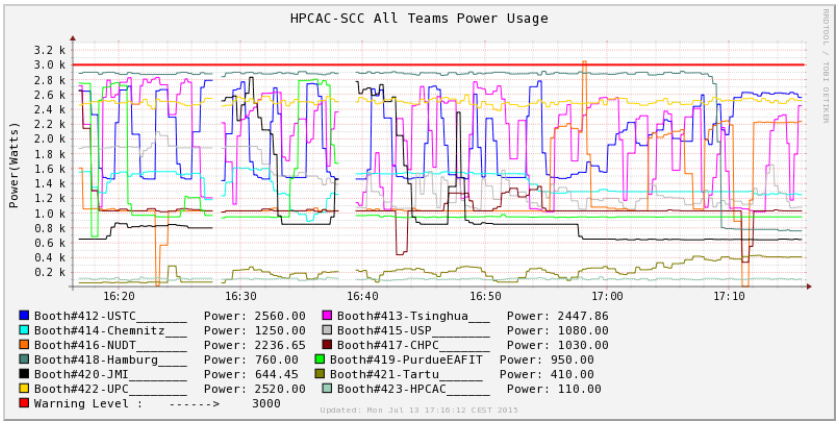


Figure 2.1.: Official power graph

Therefore we used a script to monitor the power consumption, plot the current energy consumption, and notify us if we use more than 3000 Watts. The provided power strip talks SNMP (*Simple Network Management Protocol*) and we operated with a sampling rate of one request per second.

```
1 #!/bin/bash
2
3 while true; do
4     snmpwalk -v1 -c public 192.168.1.248
        ↪ .1.3.6.1.4.1.318.1.1.26.4.3.1.5.1 | awk '/INTEGER:
        ↪ [0-9]+/{ print $4 }' >> power_data
5     sleep 1
6 done
```

After generating the dataset with the power values we used GNU Plot to display a graph (see 2.2). To ensure that GNU Plot continued rendering a looper is required.

```
1 set yrange [2500:3100]
2 set xrange [0:200]
3 set arrow from graph 0,first 3000 to graph 1,first 3000
    ↪ nohead lc rgb "#ff0000" front
4 plot "< tail -200 power_data" using 0:(10 * $1) title
    ↪ "Power [W]" with lines lt rgb "#0000ff"
```

```
pause 0.5; replot; reread;
```

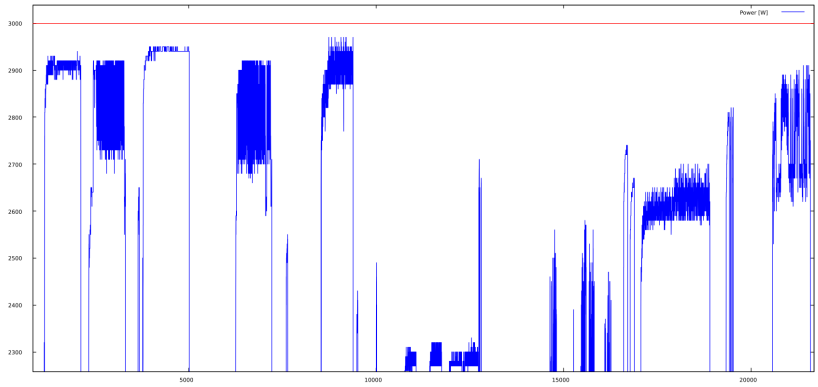


Figure 2.2.: Power consumption of different applications

2.3. Power Capping

Author: Felix Wiedemann

To stay below the 3000 Watt power budget we had to limit the power consumption of our cluster. Fortunately, the Intel processors of our compute nodes can measure and constrain the power consumption of the processor itself and the memory controller. This feature is called *Intel RAPL*.

Since 3.13 Linux exposes an easy-to-use interface for power capping. To read the current power limit we execute the following command on the head node:

```
dsh -a 'cat
   ↪ /sys/class/powercap/intel-rapl:?/con*_1_power_limit_uw'
```

The output of this command is the power consumption of each socket of all compute nodes in Milliwatts.

We used the following script to set the power limit:

```
#!/bin/bash
set -e -o pipefail
trap "echo An error occured :(" ERR
if [[ $# -eq 0 ]] || [[ $1 -lt 60 ]] || [[ $1 -gt 240 ]];
   ↪ then
    echo Usage:
    echo "  $0 <power-budget in Watts>"
    echo Value must be between 80 and 240.
    exit
fi
```

```
10 if [[ $UID -ne 0 ]]; then
11     exec sudo $0 "$@"
12 fi
13 dsh -a echo $(($1*1000000)) '| tee
   ↪ /sys/class/powercap/intel-rapl:?/con*_1_power_limit_uw'
```

2.4. Operating System

Author: Felix Wiedemann

Like our predecessor team we went through a process of evaluating which operating system should to be used. This is an important decision for the cluster setup. Virtually every supercomputer in the TOP500 uses GNU/Linux because Linux is a high performant OS kernel and Free Software, and therefore very suitable for this task. As our whole team has had experience with Linux we had only to choose the Linux distribution. On the one hand we could use a generic Linux distribution, on the other hand we could use one specialized for HPC. To simplify the decision we preselected Ubuntu[2] and Qlustar[3].

Setting up a cluster is a much more complex task than installing some software for a desktop computer as the following facts show. Our cluster consists of nine computers divided into two roles: One head node and eight compute nodes. Typically only the head node has a locally installed OS; compute nodes are booted via network with PXE[4]. We are using InfiniBand as a fast interconnection between the compute nodes which, of course, needs software support. Often custom libraries have to be built and installed, for example MPI and BLAS. If we had chosen Ubuntu we would have to setup all these things by ourself. In contrast Qlustar comes with "batteries included" for the things mentioned above. As we have never worked with Qlustar before but it seemed to ease the initial setup we decided it is worth a try – just like our predecessor team.

2.4.1. Qlustar

Qlustar is a Ubuntu derivative which is specialized for HPC.

It features a handy installer and a GUI application called *QluMan* which greatly simplifies the administration of the cluster. With *QluMan* virtually every option of the cluster can configured: disk setup of the compute nodes, Slurm options, the contents of the base images for the nodes, filesystem structure, and so on. *QluMan* was built for much larger clusters than ours, so it features a hierarchically structure and offers abstractions such as *config sets*. Therefore *QluMan* seems very complex at first, but it is also quite powerful.

Another speciality of Qlustar is the image generation for the compute nodes. With Qlustar we can simply select a set of features we like to have on the compute nodes and

[2] `http://www.ubuntu.com`
[3] `https://qlustar.com/`
[4] `https://en.wikipedia.org/wiki/Preboot_Execution_Environment`

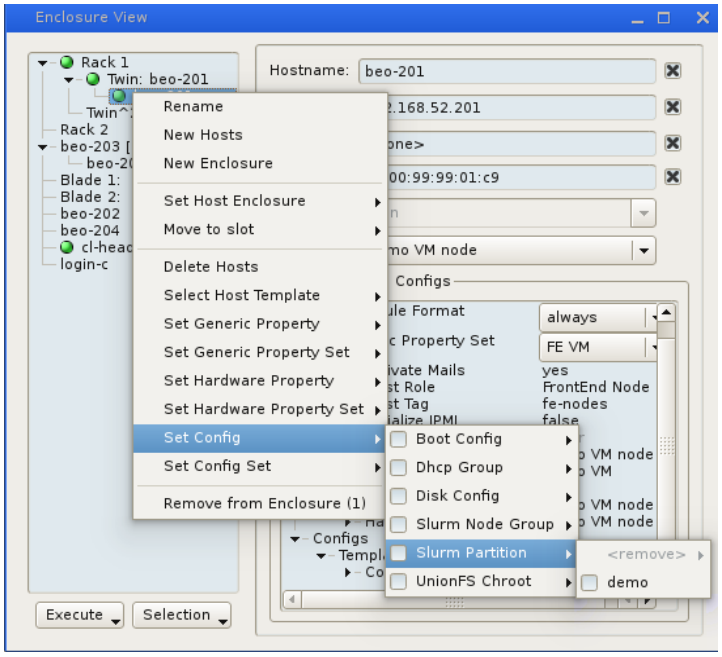


Figure 2.3.: QluMan config sets window

Qlustar generates an image for that which exactly fits our needs.

Installation

The installation of Qlustar is pretty straightforward.

After downloading the disk image, flashing it to a USB drive, and booting it the text installer shows up. The installer does not ask many questions a desktop OS would do but more specific ones for the HPC context. At first we have to setup the disks to use. Here, we set up a RAID 1 consisting of two disks. Afterwards Qlustar asks for the network setup – both Ethernet and InfiniBand, which must be in different IP subnets. The installer also asks for IPMI setup and whether to start OpenSM or not. In the end we can choose which software should be installed. Here we chose to install *Slurm* as task scheduler. After answering all the questions Qlustar is installing and will eventually reboot.

At the first boot we have to login as root and run `qlustar-initial-config` which continues with the initialization of the cluster. Here, we have to give the cluster a name. Then we enter some admin passwords for Nagios, QluMan, and Slurm. After that the cluster has to be rebooted again and is ready for action.

Advantages and Disadvantages of Using Qlustar

Qlustar offers an easy installer and enables us to concentrate on more important topics than setting up the base system for the cluster. It offers many tools and packages which come in handy for HPC systems. With QluMan we can easily modify settings of the

cluster system. Also, Qlustar has reasonable defaults set for cluster computing.

The main downside of Qlustar is that it is a whole new system with which we had to become acquainted. Additionally, QluMan is a quite complex piece of software. Some base concepts of Qlustar are obscure and rather outdated: The authentication via network is handled with *NIS* which is legacy software and has been replaced by *LDAP* on most systems a long time ago. The image generation is also obsolete as it makes use of initrd which has been superseded by initramfs in 2005. These two features are hard debug if it comes to problems because they are out of date since ten years and therefore the information about them on the internet is becoming rare. Generally, Qlustar is harder to debug for us because we are not acquainted with Qlustar as good as with Ubuntu for example.

2.5. Compiler

Author: Felix Wiedemann

There are three widely-used compilers that we can use: Intel ICC, Clang, and GCC. Unfortunately, Clang (mainline) did not support OpenMP as of SCC 15 so that Clang was not usable for at least some applications and benchmarks. We quickly chose to prefer GCC over ICC and Clang because of the following reasons: GCC is the most widely-used compiler today and there are no huge differences in performance between the three candidates. Due to the intensive use of BLAS libraries in the given applications and benchmarks the performance optimization of the compilers would only give a minor speed boost anyway. Probably, the most important aspect for our decision was that GCC was well-known in our group as nearly all of us are using GNU/Linux whose standard compiler is GCC and therefore we already got to know GCC. We used the most recent version of GCC which is 5.1. We provide an installation script in the repository. As GCC 5 uses C11 as standard some programs which are written in older standards did not compile. For those programs we had to use the compiler option `-std=c90`. Apart from that the only other compiler option we are using for compilation is `-O2`.

2.6. Filesystem

Author: Lars Thoms

Generally, the compute nodes do not need a disk or a filesystem. The compute nodes boot via PXE and all necessary data is mounted via NFS. But it can be useful to have a disk based `/tmp` or similar.

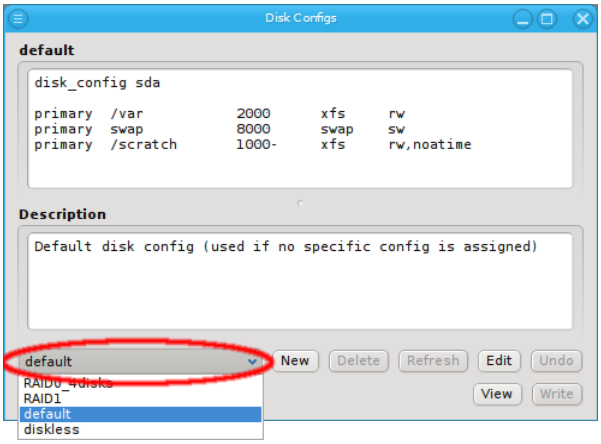


Figure 2.4.: QluMan disk config

QluMan contains a utility called Disk Configs (figure 2.4). It takes care of partitioning the disks on the nodes. Theoretically, the following configuration is valid, but it did not work for us at the SCC.

```
1 disk_config sda
2
3 primary swap 10G swap sw
4 primary /scratch 50G xfs rw
5 primary /var 50G xfs rw,noatime
6 primary /tmp 50G xfs rw,noatime,nosuid,nodev
```

After saving the config into Disk Configs you have to reboot your nodes and one of Qlustar's init-scripts starts partitioning. In fact, this process caused a problem and the nodes remained offline and there was no direct method for trouble shooting. At least we set it back to the default ZFS configuration with a 500 GB vdev mounted at `/dev/SYSTEM/pupo`.

Furthermore, copying files from eight compute nodes to the head node via Gigabit Ethernet is not reasonable, because it is quite slow. If an HPC application uses much IO we need something with very low latency and high bandwidth. Thats why we needed a distributed filesystem which uses the much faster InfiniBand connection instead of the Gigabit Ethernet.

First of all, we tried to use OrangeFS. But there was a problem in combination with HDF5, hence we switched over to BeeGFS.

2.6.1. OrangeFS

Author: Lars Thoms

OrangeFS is a parallel network filesystem which is a special form of a distributed filesystem. It uses TCP/IP or InfiniBand to communicate. Before 2011 it was known as *Parallel Virtual File System*.

This filesystem also offers interfaces for POSIX-IO system calls and MPI-IO libraries like MVAPICH2 and the newest version supports storing metadata to SSDs.

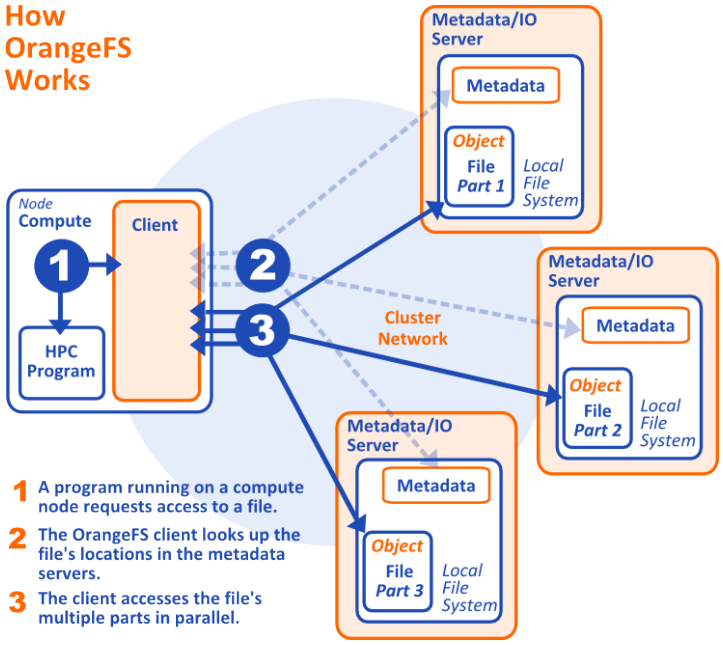


Figure 2.5.: How OrangeFS works[5]

As you can see in the figure OrangeFS consists of IO and metadata servers. It make sense to use the metadata servers on very fast storage devices like SSDs. In our setup it is sufficient to have an IO server on all nodes and one metadata server.

As all of our applications used MPI-IO we did not focus to mount it as a POSIX filesystem.

[5] http://docs.orangefs.com/v_2_9/New_How_OFS_Works.png

How to Build

```
#!/bin/bash
prefix="/tmp/testbuild"

# Compile software
CFLAGS=-std=gnu89 ./configure --prefix="$prefix"
   ↪ --enable-shared
make -j $(nproc)

# Installation
mkdir -p "$prefix"
make install
```

How to Run

We tested our settings on two nodes from *Wissenschaftliches Rechnen* over TCP, because InfiniBand was not available. Both nodes are IO servers, but only `pupo1` was a metadata server.

First of all you have create a config file and then start the IO and metadata server on each node and bind it to an TCP/IP port: we chose 4711.

```
#!/bin/bash
start_pvfs ()
{
    pvfs2-genconfig --quiet --protocol tcp --ioservers
       ↪ "pupo1,pupo2" \
        --metaservers "pupo1" --tcpport 4711 --storage
           ↪ "/tmp/pvfs-${USER}" \
        --logfile "/tmp/pvfs-${USER}.log" --fsname pvfs-fs
           ↪ "${HOME}/pvfs.conf"
    echo "tcp://pupo1:4711/pvfs-fs /pvfs pvfs2
       ↪ defaults,noauto 0 0" > "${HOME}/pvfs2tab"
    sync
    sleep 60
    for server in pupo1 pupo2; do
        ssh ${server} mkdir "/tmp/pvfs-${USER}"
    done
    for server in pupo1 pupo2; do
        ssh ${server} "$(which pvfs2-server)" -f -a
           ↪ "${server}" "${HOME}/pvfs.conf"
    done
    for server in pupo1 pupo2; do
```

```
17          ssh ${server} "$(which pvfs2-server)" -a
              ↪ "${server}" "${HOME}/pvfs.conf"
18      done
19      pvfs2-ping -m /pvfs
20  }
21  stop_pvfs ()
22  {
23      echo 'Stopping servers ...'
24      for server in pupo1 pupo2; do
25          ssh ${server} killall --verbose pvfs2-server
26      done
27      for server in pupo1 pupo2; do
28          ssh ${server} rm -f "/tmp/pvfs-${USER}.log"
29          ssh ${server} rm -rf "/tmp/pvfs-${USER}"
30      done
31      echo 'Removing configuration ...'
32      rm -f "${HOME}/pvfs2tab"
33      rm -f "${HOME}/pvfs.conf"
34  }
```

Problems with HDF5

There were some strange problems in combination with OrangeFS and HDF5. (see section 3.2.1)

2.6.2. BeeGFS

Author: Lars Thoms/Felix Wiedemann

Like OrangeFS BeeGFS is a distributed filesystem. It was developed by Frauenhofer ITWM and is as well known by its old name FhGFS. There are also IO and metadata servers, but the data is striped similar to a RAID and this advantage offers parallel read/write access.

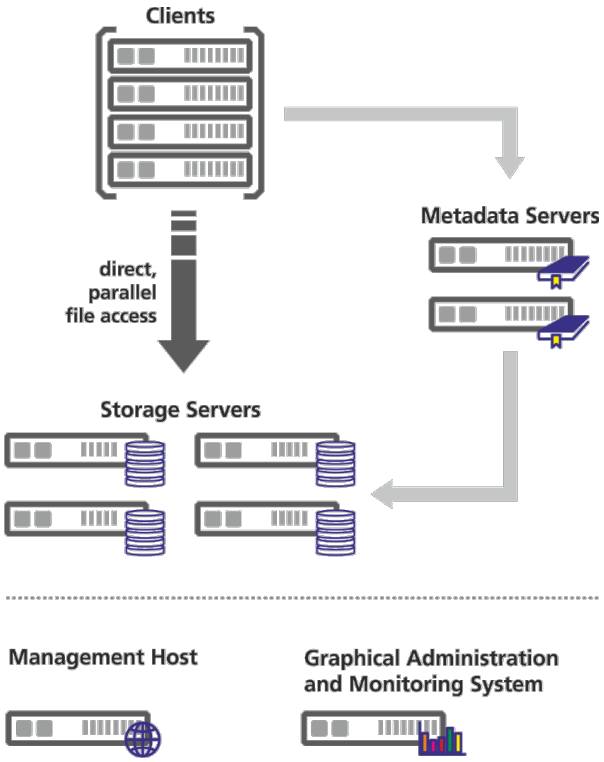


Figure 2.6.: How BeeGFS works[6]

Due to time limitations (changing from OrangeFS to BeeGFS because of HDF5-problems) we simply followed the instructions from the developer of Qlustar without optimizing (see next paragraphs).

Select BeeGFS mgmt Option upon Installation

Check that it is running on the head-node
`$ /etc/init.d/beegfs-mgmtd status`

[6] "An introduction to BeeGFS" by Jan Heichler from November 2014 – v1.0, page 6

Setting up a BeeGFS Server Node

1. Create a Qlustar image with the required modules
   In QluMan, create an image (named for example StorNode) that contains the module beegfs-<version>-server (current version 2014.01). If you want to mount the BeeGFS filesystem also on the storage nodes additionally include the beegfs-<version>-client module. On compute nodes that do not run any BeeGFS meta/storage targets, only the beegfs-<version>-client module is required.

2. Create a boot config with this image and include it in a config set
   Create a new boot config (named for example StorNode) and select the just created image.

3. Create a config set that includes this boot config
   Create a new config set (named for example StorNode) and add the just created boot config (StorNode).

4. Create a disk config for the storage nodes
   Create a new disk config (named for example StorNode) to define the ZFS setup.

Examples Make a BeeGFS FS from the local disks of the compute nodes. A single SSD in the compute nodes is used for a meta and storage target. Copy the definitions of the default boot config and add bmeta bstor as follows.

```
1 ZFS = var, scratch, bmeta, bstor
2 [bmeta]
3 zpool = SYS
4 mountpoint = /beegfs/meta
5 compress = lz4
6 [bstor]
7 zpool = SYS
8 mountpoint = /beegfs/stor
9 compress = lz4
```

5. Add the new disk config to the config set (StorNode)

6. Create a new host template with the above definitions
   Create a new host template (named for example StorNode) and select the just created config set StorNode. Use (or create) a sensible generic/hardware property set matching the properties of the storage nodes

7. Register and boot the storage nodes
   Register the storage nodes like any other nodes and assign the just created host template (StorNode) to them. Reboot them.

Setting up BeeGFS meta and storage Target on the Storage Nodes

On each storage node perform the following steps (We assume that the above disk config 'StorNode' is initialized on each node):

```
1 $ df -h | grep beegfs
2 SYS/bmeta                              1002T     0 1002T   0%
    ↪ /beegfs/meta
3 SYS/bstor                              1002T     0 1002T   0%
    ↪ /beegfs/stor
```

We also assume that the servers are named as sn-1, sn-2, ... such that $HOSTNAME##*- is the storage node number.

Create Metadata Server Instance

Initialize the meta target

```
1  $ beegfs-setup-meta -p /beegfs/meta -s ${HOSTNAME##*-} -m
     ↪ beosrv-c$
2  Preparing storage directory: /beegfs/meta
3   * Creating format.conf file...
4   * Creating server numeric ID file: /beegfs/meta/nodeNumID
5  Updating config file:
     ↪ /etc/beegfs/meta/available/default.conf
6   * Setting management host: beosrv-c
7   * Setting storage directory in config file...
8   * Disabling usage of uninitialized storage directory in
      ↪ config file...
9   * Setting usage of extended attributes to: true
10 All done.
```

Start the meta server

```
$ /etc/init.d/beegfs-meta start
```

Check meta status

```
1 $ /etc/init.d/beegfs-meta status
2 $ less /var/log/beegfs/beegfs-meta.log
```

Create Storage Server Instance

Initialize storage target

```
$ beegfs-setup-storage -p /beegfs/stor -s ${HOSTNAME###*-}
   ↪ -i ${HOSTNAME###*-}01 -m beosrv-c
Preparing storage target directory: /beegfs/storage
 * Creating format.conf file...
 * Creating chunks directory...
 * Creating mirror directory...
 * Creating target numeric ID file:
    ↪ /beegfs/storage/targetNumID
 * Creating server numeric ID file:
    ↪ /beegfs/storage/nodeNumID
Updating config file:
   ↪ /etc/beegfs/storage/available/default.conf
 * Setting management host: beosrv-c
 * Appending to target directory list in config file...
 * Disabling usage of uninitialized storage targets in
    ↪ config file...
All done.
```

Start storage server

```
$ /etc/init.d/beegfs-storage start
```

Check the storage status

```
$ /etc/init.d/beegfs-storage status
$ less /var/log/beegfs/beegfs-storage.log
```

Register the Config Files on the Head Node for Automatic Start on Boot

Retrieve the config files from the first storage node (they are all identical):

```
$ mkdir -p
   ↪ /etc/qlustar/common/image-files/beegfs/{meta,storage}
$ scp sn-1:/etc/beegfs/meta/available/default.conf \
    /etc/qlustar/common/image-files/beegfs/meta/default.conf
$ scp sn-1:/etc/beegfs/storage/available/default.conf \
    /etc/qlustar/common/image-files/beegfs/storage/default.conf
\\\\
Generate the Qlustar destination files:\\
\begin{lstlisting}[language=Bash]
$ cat <<EOF >
   ↪ /etc/qlustar/common/image-files/destinations.beegfs-server
```

```
10   beegfs/meta/default.conf /etc/beegfs/meta/available
       ↪ root:root 644
11   beegfs/storage/default.conf /etc/beegfs/storage/available
       ↪ root:root 644
12   EOF
13 $ cat <<EOF >
       ↪ /etc/qlustar/common/image-files/destinations.beegfs-client
14   # Link source                    target
15   ../available/default.conf /etc/beegfs/client/enabled
16   EOF
17 $ cat <<EOF > /etc/qlustar/common/image-files/softgroups
18   [DEFAULT]
19
20   beegfs-server: sn-[1-4]
21   beegfs-client: beo-20[1-4] login-c
22   EOF
```

In `/etc/qlustar/common/image-files/softgroups`, the nodes are assigned to their BeegFS role. Change the node names there according to your naming scheme. With these configurations in place, BeeGFS server and clients should start/mount automatically. The default client mount point is `/beegfs/client/default`.

2.7. Modules

Author: Lennart Braun

2.7.1. Motivation

For evaluation of software configurations it is often required to have more than one configuration of a program installed at the same time. A configuration is a set of properties of the installed program. It can include the program version, the compiler used to generate machine instructions, the libraries linked to, and options given at compile time. Additionally, the installed software should be available to all users. The repeated installation of a software package under the same filesystem prefix would result in conflicts.

The chosen solution is to install every configuration under a different prefix. This installation is also called a module. We use the Environment Modules[7] package to load module by setting environment variables. For example the `bin` directory of a module containing executables gets prepended to `PATH`.

[7] `http://modules.sourceforge.net/`

2.7.2. Directory Structure

To obtain a more clearly arranged directory structure, the modules are placed into subdirectories according to their dependencies and versions.

We have four basic components evaluated with implementations by different vendors. The most basic components required by every application are a compiler and an MPI implementation. Furthermore there are different mathematical libraries implementing fast fourier transformation (FFT) algorithms and basic linear algebra subprograms (BLAS). The dependencies among these components are considered when the path to a package is constructed. Other dependencies, for example the GNU Scientific Library, are ignored for the sake of simplicity.

In the software directories (e. g. `gcc/5`) there are two types of subdirectories. The first one is a release directory. It is named after a specific release with the given major version. These directories are used as a prefix for the actual installation of the corresponding version. Consider the GCC directories as an example: `gcc/5/5.1.0/{bin, include, lib, ...}`. Second for each software package that depends on all the major packages noted in the path, there is a directory containing that packages own directory structure: `gcc/5/{mvapich2, openmpi, openblas, python, ...}`.

Figure 2.7 (on page 23) shows an excerpt of the directory tree. It can be noticed that there are two directories named *openblas* in different parts of the tree. The directory `/gcc/5/openblas/0.2` contains the OpenBLAS version 0.2.14 compiled with GCC, which depends only on the compiler. Whereas `/gcc/5/openmpi/1.8/openblas/0.2` contains other software that was built with OpenBLAS, OpenMPI and GCC. These dependencies and the corresponding version numbers are exactly the nodes on the path to the root node.

2.7.3. Installation Script

For each module we wrote an installation script to make the process automatable and easy reproducible. In order to unify the installation a template was provided (listing 2.1 on page 24). If written correctly an installation script can be run from an untarred source code archive, e. g. as `.../lammps-15May15/src$ ./lammps-15May15.sh gcc openmpi fftw none`. This call would install LAMMPS with GCC, OpenMPI and FFTW. A BLAS implementation is not required.

In the installation script the author specifies the name and the major and minor versions. Then the flags for the secondary dependencies are set. Some subroutines are imported from the `template_common.sh` file. These are used to load all required modules based on the given parameters and flags. Furthermore the information is used to build strings containing the paths below which the installation and the modulefile live.

Afterwards the application specific instructions to compile and install the package (e. g. `./configure`, `make`, `make install`) are defined. Finally a modulefile is created, so that the installed module can be loaded with the command `module load`. When this modulefile is loaded, all specified as dependencies are loaded as well. If required it is possible to append something to generated modulefile.

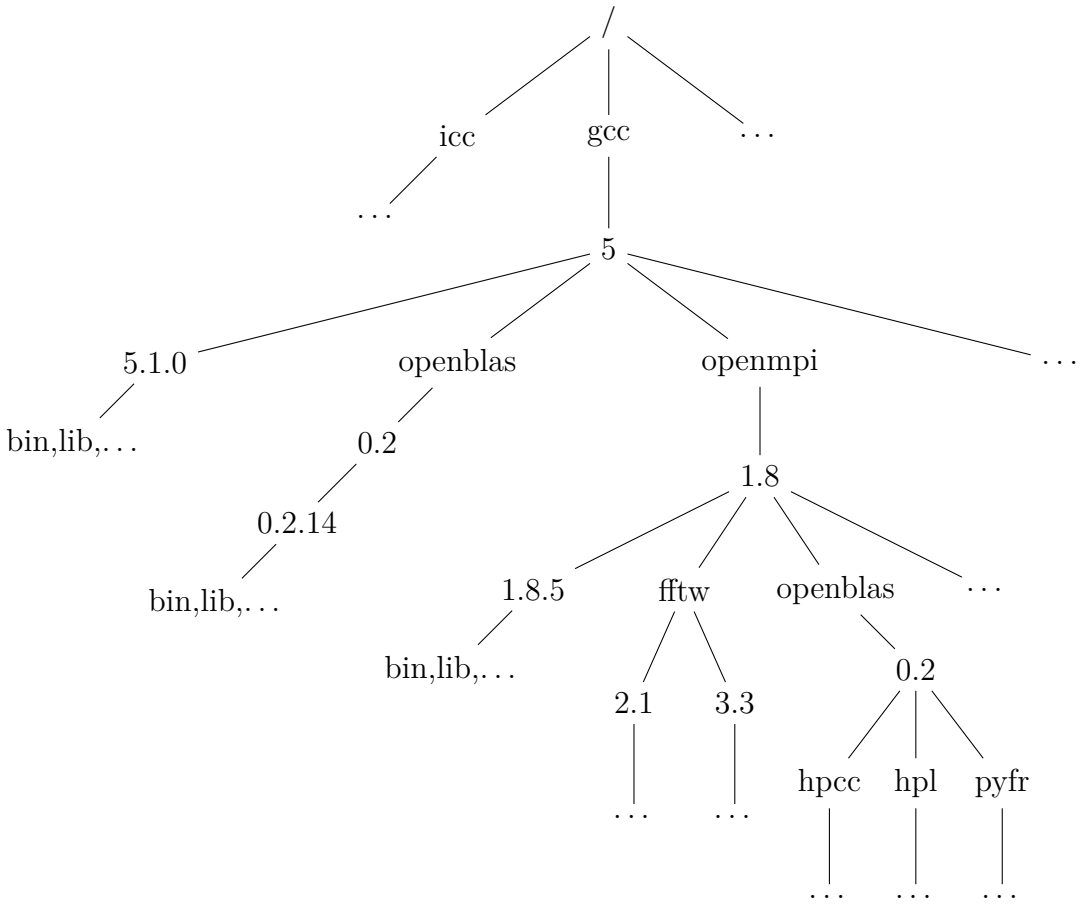


Figure 2.7.: Module directory tree

Listing 2.1: Install template

```
#!/bin/bash
# usage: sh path_to_script.sh compiler mpi fft blas
#   compiler in {gcc, icc, none}
#   mpi      in {openmpi, mvapich2-2.0, mvapich2-2.1,
   ↪ intel-mpi, none}
#   fft      in {fftw, mkl, none}
#   blas     in {openblas, atlas, mkl, none}

name="foo"
version="1.0"
release="1.0.1"

# requirements: aptitude install package0 package1
# other modules: (set to true if required)
orangefs="false"
hdf5="false"
netcdf="false"
netcdf_fortan="false"
gsl="false"
libxc="false"
python="false"  # {false, python2, python3}

set -e
source "$(dirname ${BASH_SOURCE[0]})/install_common.sh"
load_modules $1 $2 $3 $4
prefix="$(build_prefix)"
path="$(build_path)"

## compile software
# ./configure --prefix=$prefix
# make -j $(nproc)

## installation
# mkdir -p $prefix
# make install

## create modulefile
modulefile="$(build_modulefile_name)"
create_modulefile "$modulefile" "$prefix"
# append something to modulefile if required:
# echo "foo bar" >> "$modulefile"
```

3. Libraries

3.1. Math Libraries

Author: Jannek Squar

3.1.1. Overview

Needless to say, our primary objective was to win the Student Cluster Competition. To achieve this, choosing the right mathematics libraries was an important issue. This is due to the fact, that we had little influence on the hardware we were provided by our sponsor Bull. In addition for a long time the applications for benchmarking on the SCC were kept secret, one was even first announced on the competition itself (Graph500). As a result we had focused on exploring possible combination of math libraries in the beginning, to evaluate them as soon as the final hardware was delivered.

The next section (Available Libraries) is an overview of available math libraries from which we picked the best fitting ones. Some applications from chapter 5 and benchmarks from chapter 4 need implementations from a selection of LAPACK, BLAS, and FFT.

LAPACK: **L**inear **A**lgebra **PACK**age is a development from LINPACK and contains algorithms for numerical linear algebra. It is mainly used for „systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems “[1]. LAPACK's basic algebra has been outsourced in BLAS, so every LAPACK implementation also needs a BLAS implementation - for this reason some LAPACK implementations already contain a BLAS implementation.

BLAS: **B**asic **L**inear **A**lgebra **S**ubprograms provides basic vector and matrix operations like addition and multiplication. As LAPACK always relies on BLAS and BLAS contains the most basic operation it is important to choose a high-performance implementation. [ABB+99, p. 154]

FFT: **F**ast **F**ourier **T**ransform contains algorithms for making an Fourier analysis.

For every category there exists a wide variety of implementations which all have their points. After some research we chose a selection of the most promising implementations which we first benchmarked on our given cluster (see section 3.1.3) to get a first impression of their suitability. As soon as the final hardware had arrived we migrated them to it

[1] http://www.netlib.org/lapack/

and expanded our testing to determine the implementations with best performance (see section 3.1.4 and section 3.1.5).

3.1.2. Available Libraries

First of all we did some research for listing suitable implementations of the math libraries. Since there has already been a team from the university of Hamburg in the previous year, which participated in the SCC 2014, we began looking through their field report [FLN+14, Ch. 4]. Next we read through many available benchmarks, which had been executed with different libraries and hardware: [Wit08]. We could just have used a reference implementations from [net], but these are not well tuned. Especially for the BLAS implementation it is advised to choose one with good performance. Nevertheless we include the reference implementations in our list for the purpose of comparison.

LAPACK Implementations

List of implementations to be considered:

- Netlib LAPACK 3.5.0: [2] First released in the year 1992 and relatively up-to-date (last stable release November 2013) this is the reference implementation, which works well for quick starting.
- Intel MKL (Intel Math Kernel Library) 11.2.3: [3] Part of the Intel Parallel Studio XE/System Studio we were provided with a temporary key for it. MKL is easy to use, the automated installation of Intel Parallel Studio XE provides the required library files, no further action is required. Enhanced performance on Intel cores is expected.
- ACML (AMD Core Math Library): [4] We did not spend much time on evaluating this implementation because we assumed, that it would achieve high performance rather on AMD cores than on the Intel cores we use.

In the end we decided to use Netlib LAPACK for the reason that a well tuned BLAS implementation is crucial for high performance. Therefore we chose the common LAPACK implementation from Netlib to focus our efforts on testing different BLAS implementations.

BLAS Implementations

Some LAPACK implementations already contain a BLAS implementation. In this case their description can be found in the previous paragraph.

List of implementations to be considered:

[2] www.netlib.org
[3] https://software.intel.com/en-us/intel-mkl
[4] http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/

- Netlib BLAS 3.5.0 see above
- ATLAS 3.10.2 (Automatically Tuned Linear Algebra Software):[5] The big advantage of this implementation is its ability of automatic tuning. While installing ATLAS on the cluster it automatically tries different install configurations and runs tests whose runtimes are compared. Thus ATLAS chooses the configuration whose test ran fastest. Because of the amount of possible configurations installing ATLAS takes a lot more time compared to the other BLAS implementations (about 10 hours on our test cluster). Because of this and [PyF15] (ATLAS excelled relating to performance if used with PyFR in section 5.1) ATLAS was on our short list.
- GotoBLAS:[6] This implementation is outdated, its last stable version was released in 2010 and is not considered as a high-performance BLAS implementation for our modern cluster hardware.
- OpenBLAS 0.2.14: [7] Other than GotoBLAS, this implementation is still being maintained. Similar to ATLAS this implementation is being optimized depending on the basis of the hardware. Because of this and many benchmarks in which OpenBLAS showed very good performance, this implementation was also on our shortlist.
- Intel MKL see above
- ACML see above

After working through the available sources we chose this BLAS implementations for further benchmarking: ATLAS, OpenBLAS, and Intel MKL.

FFT Implementations

Because it seemed that not much had changed relating to FFT implementations, we relied on the results from [FLN+14, ch.4] and chose FFTW 3.3.4.

3.1.3. Initial Testing

As long as the final hardware was not available we executed benchmarks on our cluster to get an impression which BLAS implementation could give satisfying results. In addition we tested the runtime results with *MVAPICH 2.1* and *OpenMPI 1.8.4*. Because at that time the fine-tuning of HPL was still ongoing, we used PyFR with input *euler_vortex_2d.msh* [8]. Also some changes were made at the configuration file *euler_vortex_2d.ini*:

[5] http://math-atlas.sourceforge.net/
[6] https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2
[7] http://www.openblas.net/
[8] example file is part of the installation, http://www.pyfr.org/user_guide.php

```
[...]
[backend-openmp]
cc = gcc
cblas = <BLAS>
cblas-type = <serial/parallel>
[...]
times = range(0.0, 0.05, 2)
[...]
```

Different run configurations regarding distribution `srun -N x -n y -c z` and taskbinding `srun cpu_bind=...` were also part of this tests[9].

- *MVAPICH 2.1*, `-N 1 -n 48`

ID	BLAS	cblas-type	`cpu_bind`	time [min.]
a	`libopenblas.so`	serial	default	02:41
b			`cores`	02:39
c			`thread`	02:40
d			`socket`	11:34
e		parallel	default	02:40
f			`cores`	02:40
g			`thread`	02:38
h			`socket`	09:59
i	`libmkl_rt.so`	serial	default	02:56
j		parallel		02:57
k	`libsatlas.so`	serial	default	02:51
l		parallel	default	02:50
m	`libtatlas.so`	serial	default	20:54
n		parallel	default	NA

Table 3.1.: Initial benchmarks with MVAPICH

- *OpenMPI 1.8.4*, `-N 1 -n 8 -c 6`

[9]The last run seemed to take more time than the serial run, so it was canceled

ID	BLAS	cblas-type	`cpu_bind`	time [min.]
a	`libopenblas.so`	serial	default	03:04
b			cores	03:05
c			threads	03:07
d			socket	03:07
e		parallel	default	05:34
f			cores	05:35
g			threads	05:34
h			socket	05:34
i	`libmkl_rt.so`	serial	default	03:47
			cores	03:47
			threads	03:47
			socket	03:46
j		parallel	default	04:30
k	`libsatlas.so`	serial	default	03:00
l		parallel	default	07:09
m	`libtatlas.so`	serial	default	$\sim$ 45:00
n		parallel	default	$\sim$ 30:00

Table 3.2.: Initial benchmarks with OpenMPI

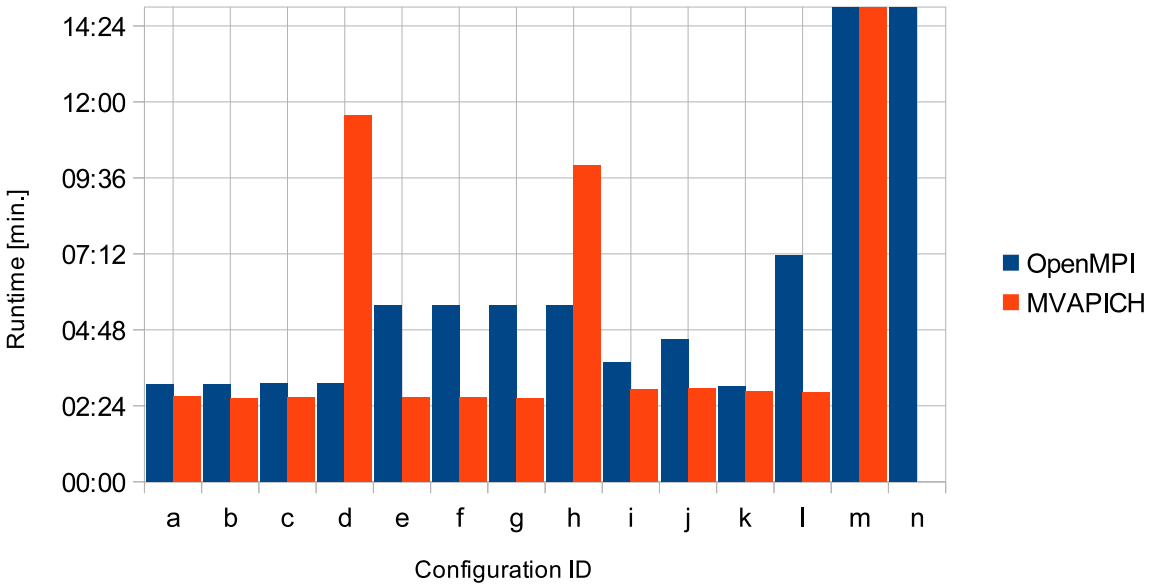


Figure 3.1.: Initial Benchmarks with MVAPICH and OpenMPI (Table 3.1 and 3.2)

3.1.4. Benchmarks on Final Hardware

When the final hardware finally arrived we used the remaining time until the competition to run the benchmarks with PyFR[10] with our preselected math libraries to choose the final configuration.

At the time of this final benchmarks there were still some problems with our implementation of MVAPICH2, therefore we used OpenMPI to select our final BLAS implementation. [11]

Benchmarks for selecting the cblas-type, cpu_bind and BLAS implementation for best performance:[12]

ID	BLAS	cblas-type	mpirun	`cpu_bind`	time [min.]
a	openblas	serial	-N 8 -n 192	cores	05:05
b			-N 8 -n 384	default	04:46
c	mkl		-N 8 -n 192	cores	05:31
d			-N 8 -n 192	default	05:31
e			-N 8 -n 384	default	04:59
f	satlas		-N 8 -n 192	cores	07:33
g			-N 8 -n 384	default	06:51
h	openblas	parallel	-N 8 -n 192	cores	05:51
i			-N 8 -n 384	default	04:47
j	mkl		-N 8 -n 192	cores	36:19
k			-N 8 -n 192	default	$\propto$ 35:00
l			-N 8 -n 384	default	04:57
m	satlas		-N 8 -n 192	cores	07:29
n			-N 8 -n 384	default	06:50

Table 3.3.: Benchmarks of BLAS-libraries

[10] At the moment of this final testing version 1.0 was available. Quick comparisons showed that there was no relevant difference between version 0.8 and version 1.0 regarding the runtime - so there was no need to rerun the initial tests, our selection of libraries persisted. Detailed description of PyFR in section 5.1

[11] When MVAPICH2 was ready for action we reran some small tests to ensure that the subsequent benchmarks with OpenMPI were comparable to it.

[12] with OpenMPI 1.8.4 and additional compiler flag `-hint=multithread`.

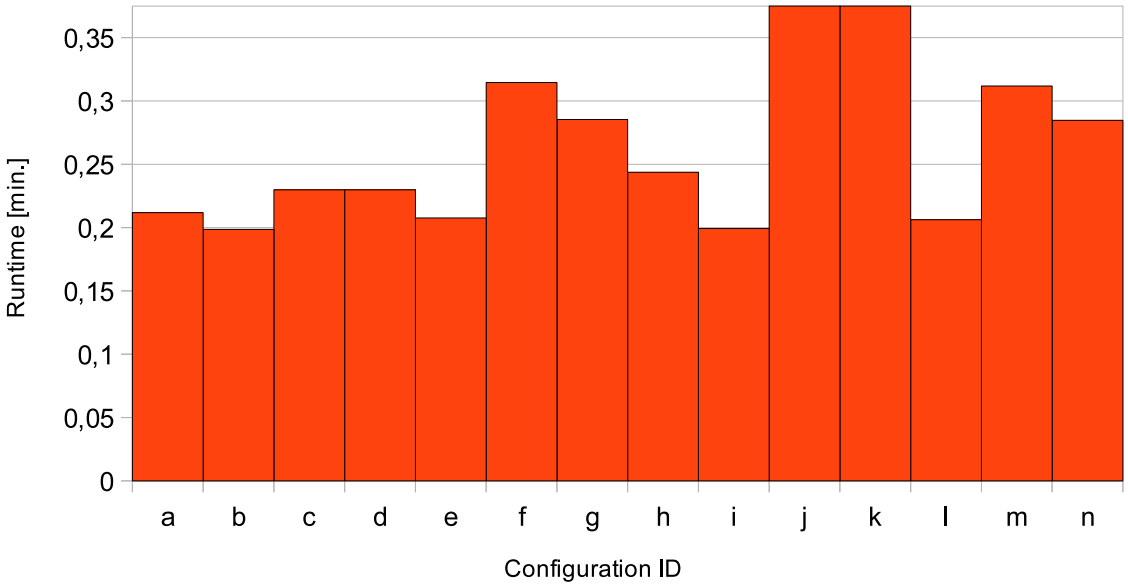


Figure 3.2.: Benchmark of BLAS-libaries (Table 3.3)

In the end we differed the benchmark configuration relating to the number of nodes to analyze how well PyFR was scaling with the chosen BLAS-library.

MPI	BLAS	`cblas-type`	mpirun	`cpu_bind`	time [min.]
OpenMPI	OpenBLAS	serial	-N 8 -n 384	default	04:48
			-N 7 -n 336		07:25
			-N 6 -n 288		06:34
			-N 5 -n 240		07:54
			-N 4 -n 192		09:35
			-N 3 -n 144		12:46
			-N 2 -n 96		18:12
			-N 1 -n 48		34:01

Table 3.4.: Test of scalability of OpenBLAS

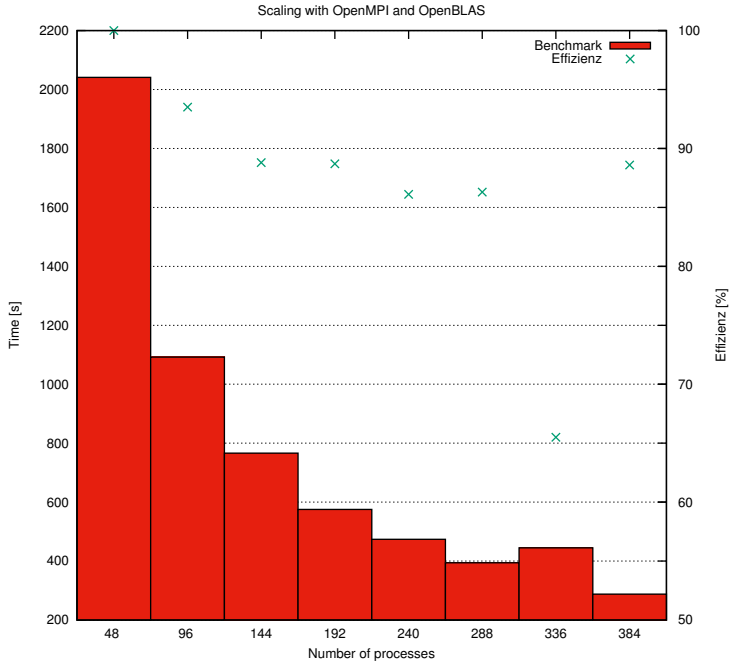


Figure 3.3.: Test of scalability of OpenBLAS (Table 3.4)

As shown in table 3.4 and figure 3.3 the scalability turned well: With increasing number of spawned processes the efficiency of speedup was near to 100% in the beginning but then decreased to about 88%.[13] However, we suppose that this behavior is attributed to the chosen example file; a bigger example file should have resulted in a better trend of efficiency.

3.1.5. Conclusion

Although ATLAS was our favorite right from the start we had to realize over the course of the benchmarks that it was not able to compete with OpenBLAS and MKL. This impression was also confirmed by [GG], where benchmarks showed the advantage in performance of Intel MKL and GotoBLAS (OpenBLAS is a fork of GotoBLAS) over Atlas [14]. In addition to its easy installation the crucial factor for our final choice of implementations were the results from the benchmarks shown in table 3.3: it became apparent that OpenBLAS outperformed the other libraries. Furthermore this benchmark showed best performance relating to PyFR with `cblas-type=serial` and `cpu_bind=default`. By comparing the number of spawned processes we decided to spawn one MPI process for every core.[15]

[13] We do not know why the efficiency clearly worsened for `-N 7 -n 336`, but this behavior turned out to be reproducible, so could not be ascribed to any temporary cause.

[14] But it should be noted that the paper was written by Kazushige Goto, creator of GotoBLAS

[15] For the sake of a clear view we set this test series aside.

3.2. I/O Libraries

Author: Lennart Braun

3.2.1. HDF5

The Hierarchical Data Format 5 is a file format for storing large amounts of data in a structured way. The HDF5 Library[16] can be compiled with MPI support as Parallel HDF5 (PHDF5). It makes use of the MPI-IO API for parallel I/O, which becomes more and more important in HPC.

Installation

The installation follows the classic `./configure`, `make`, `make install` scheme. We set the following flags:

`--enable-parallel` Builds Parallel HDF5.

`--enable-fortran` Builds the Fortran interface.

`--enable-production` Switches compiler optimizations on.

`--enable-shared` Builds shared object.

`--with-zlib` Uses the zlib library.

`./configure --help` gives a complete list about possible options.

Tests

There is the `make check` command which builds and runs tests on the compiled library. The environment variable `HDF5_PARAPREFIX` defines the working directory used in the parallel I/O tests. `RUNPARALLEL` contains the command which should be used to execute MPI programs. While running the tests, we found what appeared to be a bug in OrangeFS. A test writes some data to a contiguous buffer. But different values are read in the verification procedure.

3.2.2. NetCDF

NetCDF[17] is another library for reading and writing scientific data. It is an optional dependency of the Octopus program but was not used during the competition.

There are different NetCDF releases for multiple programming languages. We built the C and the Fortran versions. The latter one depends on the former, so that one has to be installed first. Both depend on HDF5.

[16] `https://www.hdfgroup.org/HDF5/`
[17] `https://www.unidata.ucar.edu/software/netcdf/`

Listing 3.1: Excerpt of the HDF5 install script

```
# compile software
export CC=mpicc
export FC=mpifort
export RUNPARALLEL="srun -n $(nproc)"
# export HDF5_PARAPREFIX=pvfs2:/pvfs
export CFLAGS="-O2"
flags="--enable-parallel --enable-fortran --with-zlib"
flags+=" --enable-production --enable-shared"
./configure --prefix="$prefix" $flags
make -j $(nproc)
# make check

# installation
mkdir -p "$prefix"
make install
# make check-install
```

Listing 3.2: Excerpt of the NetCDF install script

```
...
hdf5="true"
...

# compile software
export CC=mpicc
export CFLAGS="-O2"
export H5DIR="$install_root/$compiler_path/$mpi_path\
/hdf5/1.8/1.8.14"
export CPPFLAGS="$CPPFLAGS -I${H5DIR}/include"
export LDFLAGS="$LDFLAGS -L${H5DIR}/lib"
export LIBS="-ldl"
./configure --prefix="$prefix" --disable-shared
  ↪ --enable-parallel-tests
make -j $(nproc)

# installation
mkdir -p $prefix
make install
```

Listing 3.3: Excerpt of the NetCDF Fortran install script

```
1  ...
2  hdf5="true"
3  netcdf="true"
4  ...
5
6  # compile software
7  export CC=mpicc
8  export FC=mpif90
9  export F77=mpif77
10 export CFLAGS="-O2 -DgFortran"
11 export LIBS="-lnetcdf -lhdf5 -lz -lcurl"
12 ./configure --prefix="$prefix" --disable-shared
     ↪ --enable-parallel-tests
13 make -j $(nproc)
14
15 # installation
16 mkdir -p $prefix
17 make install
```

3.3. MPI

Author: Jonas Gresens

This chapter gives an overview over different MPI implementations as well as advantages and disadvantages in this year's Student Cluster Competition. Further information like tuning results can be found in last year's report.

3.3.1. Overview

MPI - short for Message Passing Interface - is not a library as it is, but a standardized specification for users and developers of message passing. MPI is the de facto standard in High Performance Computing.

There is a number of most used implementations of MPI which differ in performance, cost, network support, documentation, some interface details, etc. These are the most used, best supported, and optimized non-vendor specific MPI implementations:

- OpenMPI[18]
  - free, independent implementation by multiple institutions
- MPICH[19]

[18] http://www.open-mpi.org/
[19] https://www.mpich.org/

  - freely available, portable implementation of MPI
- MVAPICH2[20]
  - free MPICH derivative by the Ohio State University
- Intel® MPI Library[21]
  - commercial MPICH derivative by Intel
- IBM® Platform MPI[22]
  - commercial MPICH derivative by IBM

Table 3.3.1 compares some of the most useful characteristics of these implementations with regard to decision-making whether to use them for a project. The first criterion is the cost. Nearly all MPICH derivatives except MVAPICH2 are commercial, although it is possible to get an academic license for tests and evaluation for Intel® MPI Library and to use the free IBM® Platform MPI Community Edition as long as the cluster has less than 4096 cores. The dates of last releases often show how actively projects are developed. Almost of the listed implementations have been updated in 2015.

This list of criteria is not full and complete as a lot of details have been left out, such as supported operation systems, compilers, or language versions.

Since MVAPICH2, Intel® MPI Library, and IBM® Platform MPI are based on MPICH2 and include more optimizations, MPICH2 has not been looked into, since its derivatives are more focused towards optimization.

Implementation	Freely available	Last stable release
OpenMPI	yes	1.8.5 on 5.05.2015
MVAPICH2	yes	2.1 on 3.04.2015
Intel® MPI Library	no*	5.0u3 on Feb 2015
IBM® Platform MPI	no*	9.1.2 on 20.02.2014

Table 3.5.: Availability of various MPI implementations.

We decided to test each of them against every benchmark and application to find the best combinations in regards of performance and used wattage.

3.3.2. Build

This section gives a short overview over the general approach of building MPI which, however, is not different from compiling any other library.

Before building MPI some additional libraries and library headers should be installed.

[20] `http://mvapich.cse.ohio-state.edu/news/`
[21] `https://software.intel.com/en-us/intel-mpi-library`
[22] `http://www-03.ibm.com/systems/platformcomputing/products/mpi/`

```
apt install {libibmad,libibumad,libpciaccess,libhwloc}-dev
```

- `libibmad-dev` is a developer package for InfiniBand Management Datagram (MAD) library. It provides low layer InfiniBand functions to be used by the InfiniBand diagnostic and management programs.
- `libibumad-dev` is a developer package for InfiniBand Userspace Management Datagram (uMAD) library. It provides uMAD functions which sit on top of the uMAD modules in the kernel. They are used by InfiniBand diagnostic and management tools.
- `libhwloc-dev` - hwloc for MVAPICH2. OpenMPI already includes hwloc. See more about hwloc in Section.
- `libpciaccess-dev` - package library for PCI support, required by hwloc. Instead of `libpciaccess` the `libpci` packages can also be used, which is a question of using GPL[23] or other licenses. PCI is a local computer bus for attaching hardware devices such as InfiniBand cards.

OpenMPI

OpenMPI uses the GNU build system, also known as GNU Autotools, which simplifies the build process to just two commands (`./configure` and `make`).

```
./configure --prefix=$PREFIX --with-pmi
make -j $(nproc)
```

(excerpt from the build script for OpenMPI)

MVAPICH2

MVAPICH2 is built as well with GNU Autotools but has to be configured differently.

```
./configure --prefix=$PREFIX --with-pm=slurm
   ↪ --with-pmi=pmi2 --enable-libpci
make -j $(nproc)
```

(excerpt from the build script for MVAPICH2)

Intel® MPI Library

We got a version of Intel® MPI Library as part of a Intel® Parallel Studio XE Cluster Edition 2015 Update 3 which was already compiled. It is installed using a extensive shell script-based installer.

[23]GNU General Public License

IBM® Platform MPI

IBM® Platform MPI Community Edition is precompiled as well and uses a Java-based installer to extract and install the library from the archive. For some unknown reason it was not possible to run this installer properly on the cluster, since it lead to the following error:

```
$ ./platform_mpi-09.01.02.00u.x64.bin -i console
Preparing to install...
Extracting the JRE from the installer archive...
Unpacking the JRE...
Extracting the installation resources from the installer
  ↪ archive...
Configuring the installer for this system's environment...

Launching installer...

./platform_mpi-09.01.02.00u.x64.bin: 3319: exec:
  ↪ /root/install.dir.29359/Linux/resource/jre/jre/bin/java:
  ↪ not found
```

Since this problem only occurred on the cluster, we tried to get it out of our way by running the installer on a normal Laptop and copying the files to the cluster. Against all odds this worked surprisingly well, although it was necessary to set the `MPI_ROOT` environment variable accordingly to the location of the installed library.

3.3.3. Evaluation

Since the final hardware as well as the license for Intel® MPI Library was not available from the start, we were forced to evaluate the different implementations in multiple stages. The staged availability of hardware (including InfiniBand cards) and libraries prevented us from comprehensively testing everything[24].

First Round (abu)

Before the sponsored hardware by Bull and Mellanox was delivered, we used two *abu* nodes, provided by our university's *Scientific Computing* research group, to compile and test the benchmarks, applications, and libraries. We got the following results:

- OpenMPI
  - good performance, though slightly less than MVAPICH2[25]
  - correct results with every benchmark and application

[24] Every application and benchmark with every MPI implementation, math library, and compiler combination.

[25] missing results

- MVAPICH2
  - highest performance (750.4 GFLOPS)
  - correct results with everything except for Octopus
- Intel® MPI Library
  - not tested, since Intel had not provided us a license yet
- IBM® Platform MPI
  - worse performance than MVAPICH2 and OpenMPI (745.5 GFLOPS)
  - correct results with every benchmark and application

We decided to stay with MVAPICH2 for everything except Octopus, which was the reason to stay with OpenMPI as well. We decided to leave out IBM® Platform MPI because of its inferior performance as well as its problem prone installation.

Final Round (kraken)

Although we received the hardware sponsored by Bull in early June, we had to wait additional three weeks before we got the InfiniBand hardware provided by Mellanox. We tested all three remaining MPI implementations and got the following results:

- OpenMPI
  - good performance (around 14 % slower than MVAPICH2)
  - correct results with every benchmark and application
- MVAPICH2
  - highest performance (5.27 TFLOPS)
  - correct results with every benchmark and application
- Intel® MPI Library
  - good performance (4.56 TFLOPS)
  - correct results with every benchmark and application

Based on this results we decided to use MVAPICH2 for every benchmark and application.

3.3.4. Tuning

In preparation for 2014's Student Cluster Competition serious effort has been put into measuring the performance gain of combining different MPI tuning options. The results were underwhelming as the performance gain was less than one percent.

We decided that we would rather spend our time optimizing the applications than to fine tune the MPI implementations.

3.3.5. MV2_USE_LAZY_MEM_UNREGISTER Flag

The `MV2_USE_LAZY_MEM_UNREGISTER` flag enables MVAPICH2 to use memory registration cache.

Setting `MV2_USE_LAZY_MEM_UNREGISTER=0` fixes the incorrect results for octopus, as the MVAPICH2's cache registration/deregistration code breaks when the burst of generated messages gets too large.[26]

[26] `https://github.com/m-a-d-n-e-s-s/madness/wiki/MADNESS-performance-tuning`

4. Benchmarks

4.1. HPL

Author: Felix

HPL is a software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark.

The algorithm used by HPL can be summarized by the following keywords: Two-dimensional block-cyclic data distribution - Right-looking variant of the LU factorization with row partial pivoting featuring multiple look-ahead depths - Recursive panel factorization with pivot search and column broadcast combined - Various virtual panel broadcast topologies - bandwidth reducing swap-broadcast algorithm - backward substitution with look-ahead of depth 1.

The HPL package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it. The best performance achievable by this software on your system depends on a large variety of factors. Nonetheless, with some restrictive assumptions on the interconnection network, the algorithm described here and its attached implementation are scalable in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage.

The HPL software package requires the availability on your system of an implementation of the Message Passing Interface MPI (1.1 compliant). An implementation of either the Basic Linear Algebra Subprograms BLAS or the Vector Signal Image Processing Library VSIPL is also needed. Machine-specific as well as generic implementations of MPI, the BLAS and VSIPL are available for a large variety of systems.

More information on HPL can be found at the HPL website.[1]

4.1.1. Dependencies

To run HPL you need a BLAS library, an MPI library, and a C compiler, of course. We used Intel MKL, MVAPICH2 2.1, and GCC 5.1.

[1] `http://www.netlib.org/benchmark/hpl/`

4.1.2. How to Build

Unpack the tarball and execute `sh ./setup/make_generic` to create a default Makefile called `Make.UNKNOWN`. Replace `UNKNOWN` with an architecture identifier of your choice. Then edit this file and set the following two options: `LAlib` and `ARCH`. `ARCH` is the architecture identifier you've used in the filename and `LAlib` are the linking options. Here you have to specify the BLAS library, for example `-lblas`.

After the configuration load the corresponding modules for the packages listed under Dependencies and execute `make arch=$arch` where `$arch` is the architecture identifier again. Afterwards the binary `xhpl` and a sample configuration `HPL.dat` will be located under `./bin/$arch/`.

4.1.3. How to Run

First, you have to edit the configuration file. The configuration file shown in listing A.1 should be a good starting point. There are many websites that describe how a good configuration file for HPL should look like and there are even configuration file generators but some options should be tested with different values to produce optimal results. Basically, all options but N, NB, P, Q are fixed because they are optimal for (nearly) every system.

N is the problem size, i.e. the order of the square matrix. For our cluster we used 327680 which roughly equates to 100 GiB memory usage per compute node. The matrix should be as large as possible to achieve good results. But it is advisable to use only about 90 % of the memory so that there is enough room for other data. A good value for N can be calculated with the following formula:

$$N \approx 0.9 \cdot \sqrt{\frac{M}{8}} \text{ where } M \text{ is the total amount of memory in bytes}$$

NB is the block size for computation and communication. A good value for NB should be between 128 and 256. The optimum depends on the computation and communication performance of the cluster, and especially the ratio of them. We chose 256 at the ISC because it was optimal in some sample runs with different NBs.

The last two interesting parameters P and Q describe the layout of the process grid. $P \cdot Q$ should be equal to the total number of CPU cores of the cluster and the grid's shape should be close to a square, i.e. P and Q should be approximately equal with $P \leq Q$.

Now, HPL can be run with `srun -N $NUM_OF_NODES -n $NUM_OF_CORES ./xhpl`. Our HPL run took over an hour and we've reached 4875 GFLOPS.

4.1.4. Problems

Compared to our competitors which used accelerator cards our CPU-only cluster has a long running time of about an hour for one HPL benchmark. Therefore we had not much time for benchmarking different configurations at the SCC. Slightly after the

deadline we finished another run with over 5 TFLOPS. As with Octopus we had to use `MV2_USE_LAZY_MEM_UNREGISTER=0` (see 3.3.5) for correct results with MVAPICH2.

4.1.5. Results

As stated in 4.1.3 we scored 4875 GFLOPS. (1.625 GFLOPS per Watt) With power capping the cluster's R_{peak} is $8 \cdot 2 \cdot 12 \cdot 16 \cdot 2.1 \approx 6451$ GFLOPS, so the HPL benchmark run had an efficiency of about 76%. Another run which did not finish in time had an efficiency of nearly 80%. Both efficiencies are rather good results.

4.2. HPCC

Author: Lennart Braun

4.2.1. Overview

The HPC Challenge Benchmark[2] (HPCC) is a collection of several benchmarks testing the performance of a system in a variety of categories.

We used version 1.4.3 of HPCC. A new alpha release (1.5.0a) became available in May 2015, but it was not stable enough to use it.

- The suite includes the HPL benchmark, which measures the number of executed floating point operations per second (FLOPS), while solving a large linear equation system. The whole system is used to calculate a solution using MPI. See section 4.1 for details.
- The DGEMM benchmark tests the performance of the `dgemm` routine, which is part of the Basic Linear Algebra Subprograms (BLAS) specification. `dgemm` performs a multiplication of general matrices with double precision floating point numbers.
- STREAM is a benchmark measuring the memory bandwidth to main memory. It gets discussed in more detail in section 4.2.3.
- In PTRANS (Parallel TRANSpose) a large matrix which is distributed among the processes gets transposed using MPI. This operation measures the transfer rate of large array between the memories attached to the different processors and nodes.
- RandomAccess executes updates at random positions in an integer array hold in memory. The rate of updates is measured in updates per second (UP/s).
- FFT measures the performance of double precision floating point operations while computing a complex one-dimensional discrete Fourier transform.

[2] `http://icl.cs.utk.edu/hpcc/`

- The last benchmark, Latency/Bandwidth, measures latency and bandwidth of MPI communication. It transmits messages of different size between multiple nodes using different communication patterns.

The reader is referred to [LDK+] for additional information.

The HPCC input file is almost the same as the HPL input file (listing A.1) as all problem sizes are based on the size of the global matrix in HPL. It is possible to specify additional problem sizes for PTRANS.

4.2.2. Installation

HPCC needs to be linked with an MPI implementation and a BLAS library. An external FFT library is optional. The building process runs analog to the one of HPL. A Makefile in the `hpl` subdirectory has to be adjusted to the requirements. Differences to the HPL build are for example the path in the `TOPdir` variable and flags for using an external FFT implementation.

4.2.3. Tuning and Modifications

The rules of HPCC[3] allow the modification of certain parts of the codebase. Alternative algorithms or implementations may be used, provided that no computation is skipped, the same precision is reached and all built in verification tests are passed. For each benchmark, procedures are specified, which are allowed to be substituted with custom code. The SCC rules did not explicitly restrict the optimization of the benchmark, but the build method and all patches had to be submitted with the results[4].

All modifications were developed and tested with OpenMPI. Combined with MVAPICH2 the modifications were not stable enough to use them in the competition.

STREAM

The STREAM benchmark allocates three arrays with sizes depending on the HPL problem size from the input file. The source code provides four function kernels which can be substituted. Each of them performs a simple operation on the elements of one or two arrays and stores the result in the third.

To fully exploit the available resources, we crafted custom assembly code specific for our Intel Haswell processors. We made extensive use of the vector instructions from AVX2. We also did use the FMA3 instruction set extension for fused multiply-add operations.

The tuned Triad kernel is shown as an example in listing 4.2 (page 48). A complete listing of all tuned kernels is available in the appendix (listing A.2 on page 73).

Each kernel consists of three sections.

[3] `http://icl.cs.utk.edu/hpcc/overview/index.html`

[4] `http://www.hpcadvisorycouncil.com/events/2015/isc15-student-cluster-competition/faqs.php`

Listing 4.1: Original STREAM kernels (without OpenMP)

```
1  void tuned_STREAM_Copy()
2  {
3      int j;
4      for (j=0; j<VectorSize; j++)
5          c[j] = a[j];
6  }
7
8  void tuned_STREAM_Scale(double scalar)
9  {
10     int j;
11     for (j=0; j<VectorSize; j++)
12         b[j] = scalar*c[j];
13 }
14
15 void tuned_STREAM_Add()
16 {
17     int j;
18     for (j=0; j<VectorSize; j++)
19         c[j] = a[j]+b[j];
20 }
21
22 void tuned_STREAM_Triad(double scalar)
23 {
24     int j;
25     for (j=0; j<VectorSize; j++)
26         a[j] = b[j]+scalar*c[j];
27 }
```

The first one (lines 4 to 11 in listing 4.2) loads the array addresses into registers and does the setup of the loop counter. The addresses of the last 256 bit words in the input and output arrays are stored in `rsi` and `rdi` respectively (If a third array is used its address is stored in `rcx`). A negative offset `rax` relative to the end of the arrays is used as indexing variable and loop counter.

In the second part (lines 13 to 21) four packed doubled are loaded by `vmovapd` from the used arrays. Then an operation is applied on each of the four doubles in the 256 bit words. In `Scale` the `vmulpd` instruction is used to multiply the source operand in `ymm1` with the scalar value. The result is stored in `ymm2`. In `Add` `vaddpd` adds the content of two vector registers. The last kernel, `Triad`, requires a multiplication followed by an addition. The FMA3 instruction `vfmadd231pd %ymm0, %ymm1, %ymm2` performs the

following calculation (element-wise):

$$\%ymm0 \leftarrow \%ymm1 \cdot \%ymm2 + \%ymm0$$

Finally the result is stored via `vmovntpd` in the destination array. `vmovntpd` is a non-temporal write operation. That is, the cache lines are not updated and the data is written directly to memory. This strategy is more efficient, if the data is not used in the near future. In this case, the written data is not used again at all in the benchmark.

The third part (line 23 and following) performs the same operations as above for the remaining array elements in case the array length is not divisible by four. Contrary to the previous block, only one double at the time is processed.

An overview about the increase in performance is given in table 4.1 and plotted in fig. 4.1 on page 49.

	SingleSTREAM		StarSTREAM (average)	
	default	custom	default	custom
Copy	11.315 GB/s	18.021 GB/s	3.252 GB/s	4.670 GB/s
Scale	11.560 GB/s	18.777 GB/s	3.233 GB/s	4.604 GB/s
Add	12.637 GB/s	19.634 GB/s	3.770 GB/s	4.849 GB/s
Triad	12.628 GB/s	19.441 GB/s	3.975 GB/s	5.098 GB/s

Table 4.1.: Comparison of STREAM Kernel implementations (HPCC on four nodes with 24 tasks each)

FFT

By default HPCC uses a built-in implementation of FFTE. There is an alternative implementation using the FFTW2 API. FFTW2 is deprecated and was last updated in 1999. Since FFTW3 uses a different interface, the two versions are incompatible to each other. Thus linking against FFTW3 results in an error. Information about the differences between version 2 and 3 can be found in the FFTW3 documentation[5]. We developed a patch for the FFT benchmark that allowed us to use the newer FFTW3 library.

Using the patched version in combination with the Intel MKL Library resulted in a segmentation fault during the first MPIFFT benchmark. Because the FFTE library performed almost as good as FFTW3, we chose to use FFTE and MKL in the competition, as the latter one was the favored option for HPL.

An overview about the performances of the different FFT implementations is given in table 4.2 and plotted in fig. 4.2 on page 50.

[5] `http://www.fftw.org/fftw3_doc/Upgrading-from-FFTW-version-2.html`

	FFTW3	FFTE	FFTW2
MPIFFT	30.919	27.918	20.870
StarFFT (average)	3.485	1.678	1.524
SingleFFT	3.194	2.201	1.848

Table 4.2.: Comparison of FFT implementations in GFLOPS (HPCC on two nodes with 24 tasks each)

4.2.4. Results

In this section we give an overview about the performance achieved during the competition.

- HPL: 4934 GFLOPS
- RandomAccess

MPI	0.244 GUP/s
Star (average)	0.015 GUP/s
Single	0.033 GUP/s

- PTRANS

walltime	47.270 GB/s
cputime	61.184 GB/s

- DGEMM

Star (average)	29.716 GFLOPS
Single	41.959 GFLOPS

- STREAM

	Copy	Scale	Add	Triad
Star (average)	4.779 GB/s	4.750 GB/s	5.091 GB/s	5.418 GB/s
Single	17.828 GB/s	19.215 GB/s	19.203 GB/s	19.236 GB/s

- FFT

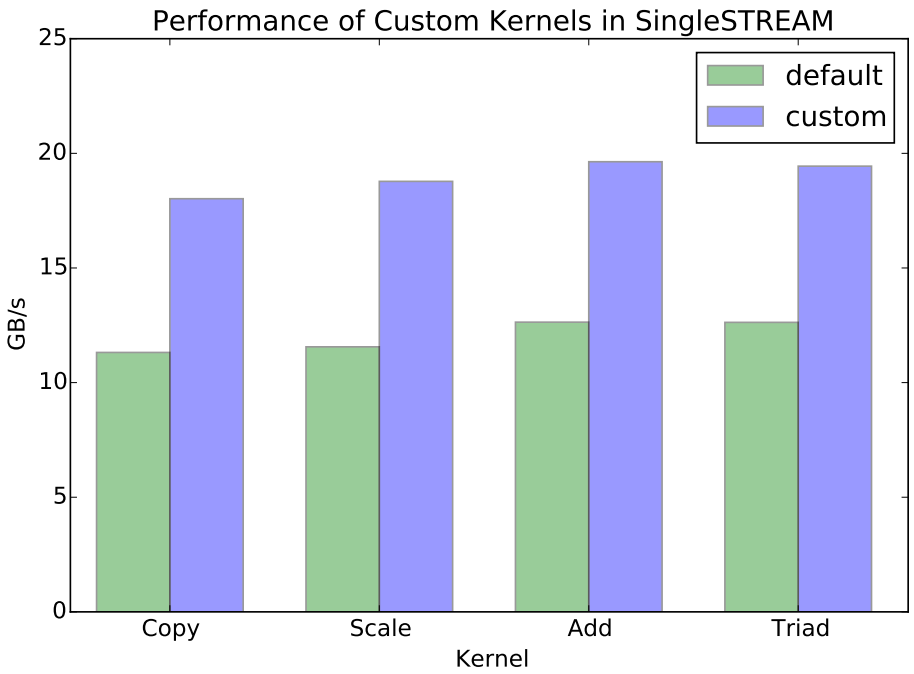
MPI	84.132 GFLOPS
Star (average)	1.689 GFLOPS
Single	2.378 GFLOPS

- LatencyBandwidth

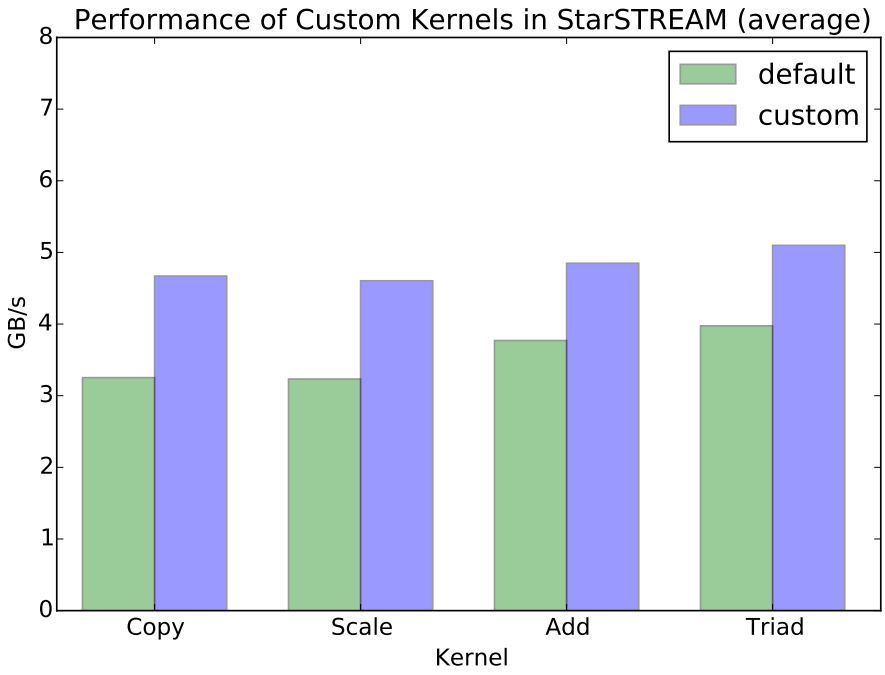
	PingPong	NaturallyOrderedRing	RandomlyOrderedRing
Latency	1.438 µs	1.362 µs	1.918 µs
Bandwidth	8.606 GB/s	0.506 GB/s	0.431 GB/s

Listing 4.2: Tuned STREAM Triad kernel

```
1  void tuned_STREAM_Triad(double scalar)
2  {
3    __asm__ (
4      "mov %[vs], %%rax                    \n" // n
5      "sub $4, %%rax                       \n" // n - 4
6      "lea (%[a], %%rax, 8), %%rdi         \n" // &a[n-4]
7      "lea (%[b], %%rax, 8), %%rcx         \n" // &b[n-4]
8      "lea (%[c], %%rax, 8), %%rsi         \n" // &c[n-4]
9      "neg %%rax                           \n" // -(n - 4)
10     "vbroadcastsd %[scl], %%ymm3         \n" // scalar
11     "jg 2f                               \n" // skip if n < 4
12
13     "1:                                  \n" // begin of loop
14     "vmovapd (%%rsi, %%rax, 8), %%ymm0   \n" // load 4 doubles (c)
15     "vmovapd (%%rcx, %%rax, 8), %%ymm1   \n" // load 4 doubles (b)
16     ".intel_syntax noprefix              \n" //
17     "vfmadd231pd %%ymm1, %%ymm3, %%ymm0 \n" // fma
18     ".att_syntax prefix                  \n" //
19     "vmovntpd %%ymm1, (%%rdi, %%rax, 8) \n" // store 4 doubles
20     "add $4, %%rax                       \n" // next 4 doubles
21     "jl 1b                               \n" // next iteration
22
23     "2:                                  \n"
24     "sub $4, %%rax                       \n" // remainder
25     "jns 4f                              \n" // done
26
27     "3:                                  \n"
28     "vmovsd 32(%%rsi, %%rax, 8), %%xmm0 \n" // load 1 double from a
29     "vmovsd 32(%%rcx, %%rax, 8), %%xmm1 \n" // load 1 double from b
30     ".intel_syntax noprefix              \n" //
31     "vfmadd231sd %%xmm1, %%xmm3, %%xmm0 \n" // fma
32     ".att_syntax prefix                  \n" //
33     "vmovsd %%xmm1, 32(%%rdi, %%rax, 8) \n" // store 1 double to c
34     "add $1, %%rax                       \n" // next 4 doubles
35     "js 3b                               \n" // done
36
37     "4:                                  \n" // end
38     : // no output operands
39     : [a] "r" (a), [b] "r" (b), [c] "r" (c), [vs] "m" (VectorSize),
         ↪ [scl] "m" (scalar) // input operands
40     : "%ymm0", "%ymm1", "%ymm2", "%xmm0", "%xmm1", "%xmm2", "%rax",
         ↪ "%rcx", "%rsi", "%rdi" // clobbered registers
41   );
42 }
```



(a) SingleSTREAM



(b) StarSTREAM (average)

Figure 4.1.: Comparison of STREAM Kernel Implementations (see table 4.1)

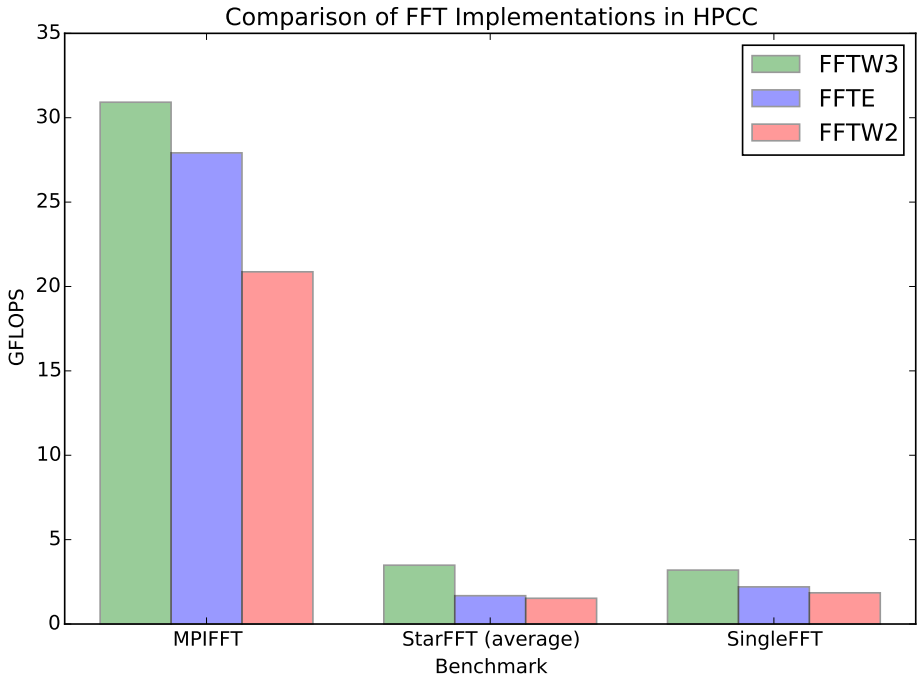
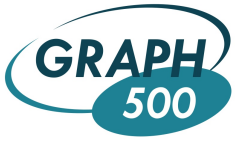


Figure 4.2.: Comparison of FFT implementations (see table 4.2)

4.3. Graph500

Author: Felix Wiedemann



Graph500[6] was the secret application at ISC'15, hence we were not familiar with this benchmark before. Graph500 is a HPC benchmark which constructs a huge graph and does a breadth-first-search on it. It scales from small graphs which are only a couple of megabytes in size to huge graphs with more than one petabyte. There is a reference implementation which features several implementations for specific parallelizations – especially MPI and OpenMP. The output score of Graph500 is the number of traversed edges per second – or short *TEPS*.

4.3.1. Dependencies

We used the reference MPI implementation. The only dependencies are therefore a C compiler and an MPI library.

4.3.2. How to Build

Unpack the source code archive which is available on the homepage of Graph500. Change into the directory `mpi` and execute `make`. Make sure that you load the necessary modules before the compilation.

4.3.3. How to Run

```
srun -N <nodes> -n <total_cores> ./graph500_mpi_<flavor> <scale>
```

4.3.4. Problems

As it was the secret benchmark we hadn't much time for optimizing this benchmark. The Graph500 reference code offers four MPI implementations which greatly vary in speed. Additionally, there are many parameters which control the graph construction and traversal. Also, the runtime is very long for large inputs. In the end we used the defaults for all parameters and a *scale* factor of 19 which is significantly smaller than the scale factor of the smallest predefined problem class called *Toy*. For the same reason we don't have any data of interest to present.

[6] `http://www.graph500.org`

4.3.5. Results

Our result was `2.73 GTEPS` with the *replicated* MPI implementation and a *scale* factor of 19. The result seems to be really bad as a MacBook Air has got `1.2 GTEPS`[7]. However, the Graph500 benchmark does not scale very well. A similar system[8] has got `28 GTEPS` with 128 cores. Upscaled to our cluster with 192 cores (and assuming perfect strong scaling) the score would be `42 GTEPS`, so there is a considerable room for improvement of an order of magnitude.

[7] See place 155 in `http://www.graph500.org/results_nov_2015`

[8] See place 77, 8 nodes with two Intel Xeon E5-2670 2.60GHz per node

5. Applications

5.1. PyFR

Author: Lars Thoms

PyFR is an open-source Python based framework for solving advection-diffusion type problems on streaming architectures using the Flux Reconstruction approach of Huynh. The framework is designed to solve a range of governing systems on mixed unstructured grids containing various element types. It is also designed to target a range of hardware platforms via use of an in-built domain specific language derived from the Mako templating engine.[1]

5.1.1. Dependencies

- GCC 5.1.0
- Python 3.4
- HDF5 1.8.15-patch1
- OpenMPI 1.8.5
- OpenBLAS 0.2.14

5.1.2. How to Build

First of all, we have to install some packages from the Ubuntu repositories: `apt-get install python-setuptools python-virtualenv libmetis-dev libgmp-dev libscotch-5 libhdf5-dev`

After that a virtual-environment for python3 is needed. So load Python3 module, create a virtual-env and activate it: `virtualenv -p python3 PyFR-VirtualEnv`

Afterwards it is necessary to load MPI, compiler (GCC), and HDF5 modules to install their python-wrapper and other libraries: `pip install numpy mpmath mpi4py mako cython appdirs decorator h5py`

[1] Source: http://pyfr.org

To build HDF5-wrapper with MPI-support manual configuration is required.

```
pip install --download-cache="$(pwd)" h5py
tar xf h5py-*.tar.gz
(
    cd h5py-*
    python setup.py configure --mpi
    python setup.py build
    python setup.py install
)
```

Changing to another MPI-implementation is easy: load corresponding module and reinstall *mpi4py*.

Finally install PyFR itself: `python3 ./setup.py install`

5.1.3. How to Run

To solve a problem with PyFR you need two things: a configuration-file (*.ini) and a Gmsh-file. First set right configuration values:

- cc: C compiler
- cblas: path to shared C BLAS library
- cblas-type: serial or parallel

Thereafter convert Gmsh-file to *.pyfrm and start partitioning our problem into n parts, where n = Number of processes.

```
pyfr import cyl-pritet.msh cyl-pritet.pyfrm
mkdir p192
pyfr partition 192 cyl-pritet.pyfrm cyl-1000.00.pyfrs p192/
```

Now it is possible to start simulation. We used Slurm, so start PyFR with srun in the virtual-env.

```
srun -N 8 -n 192 pyfr -p -b openmp restart
   ↪ p192/cyl-pritet.pyfrm cyl-1000.00.pyfrs cylinder.ini
```

After PyFR finished simulation (it takes really some time!) the solution (*.pyfrs) can be exported to a *.vtu-file. To visualize the results we used ParaView. It can handle *.vtu-files and generates nice pictures or videos.

```
pyfr export p192/cyl-pritet.pyfrm cyl-1003.00.pyfrs
   ↪ cyl-1003.00.vtu -d 5
```

5.1.4. Problems

It was very difficult to install ParaView on our cluster. But it was necessary, because generating the two images (figures 5.1a + 5.1b) for the competition used very much memory and processing time.

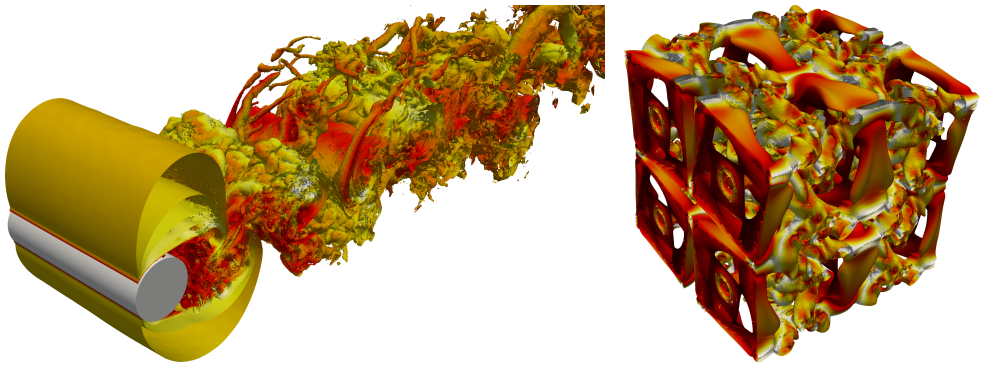
Additionally we have got always a warning, but it did not disappear and did not affect our results:

```
1 WARNING: Error in initializing MVAPICH2 ptmalloc
    ↪ library.Continuing without InfiniBand
```

5.1.5. Results

Here you see the two images from the Student Cluster Competition. To solve the first problem (Cylinder) we used 8 nodes and 192 processes. It took 27 minutes from step 1000.00 to 1003.00. Unfortunately, this was extremely slow compared to the competitors. The second one took 21 minutes from step 84.515 to 91.000.

```
1 srun -N 8 -n 192 -J "PyFR" --hint=multithread pyfr restart
    ↪ -p -b openmp p192/cyl-pritet.pyfrm
    ↪ p192/cyl-1000.00.pyfrs cylinder.ini
2 srun -N 8 -n 192 -J "PyFR" --hint=multithread pyfr restart
    ↪ -p -b openmp p192/taylor_green.pyfrm
    ↪ p192/taylor_green-1-84.515.pyfrs taylor_green.ini
```



(a) Cylinder (b) Taylor green

Benchmarks

Each table represents a standalone test group. Each group has different test sets and runtimes are not comparable to each other!

We tested MPI libraries (MVAPICH2, OpenMPI, and Intel MPI), math libraries (OpenBLAS and MKL, ATLAS was not an alternative), and PyFR parameters.

The result was that MVAPICH2 is the fastest MPI library and MKL the fastest library for math. We started PyFR with one process per core, because spawning threads lead to a longer runtime. Binding processes to cores did not indicate significant effects.

ID	MPI	BLAS (`cblas-type`)	mpirun	`cpu_bind`	time [min.]
1	MVAPICH2	OpenBLAS (serial)	-N 1 -n 24	cores	9:52
2	MVAPICH2	OpenBLAS (parallel)	-N 1 -n 24	cores	10:01
3	MVAPICH2	MKL (serial)	-N 1 -n 24	cores	9:36
4	MVAPICH2	MKL (parallel)	-N 1 -n 24	cores	> 10:00
5	MVAPICH2	MKL (serial)	-N 1 -n 48	cores	> 10:00
6	OpenMPI	MKL (serial)	-N 1 -n 24	cores	9:39
7	MVAPICH2	MKL (serial)	-N 1 -n 24 –hint=multithread	cores	9:16
8	IMPI	MLK (serial)	-N 1 -n 24 –hint=multithread	cores	9:49

Table 5.1.: PyFR-Benchmarks 1

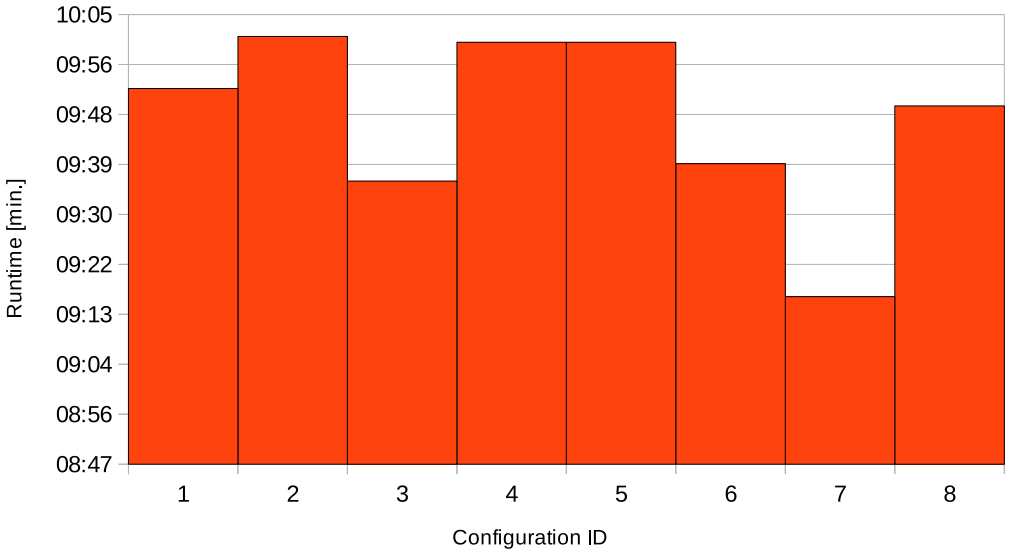


Figure 5.1.: Diagram to benchmark 1

ID	MPI	BLAS (`cblas-type`)	mpirun	`cpu_bind`	time [min.]
1	MVAPICH2	MKL (serial)	-N 8 -n 192 –hint=multithread	cores	5:24
2	MVAPICH2	MKL (serial)	-N 8 -n 384 –hint=multithread	cores	4:38
3	MVAPICH2	MKL (serial)	-N 8 -n 384 –hint=multithread	threads	4:42
4	MVAPICH2	MKL (serial)	-N 8 -n 384 –hint=multithread –cpu-freq=Performance	cores	4:37
5	MVAPICH2	MKL (serial)	-N 8 -n 384 –hint=multithread –cpu-freq=2500000	cores	4:39
6	MVAPICH2	MKL (serial)	-N 8 -n 384 –hint=multithread –cpu-freq=3300000	cores	4:40

Table 5.2.: PyFR-Benchmarks 2

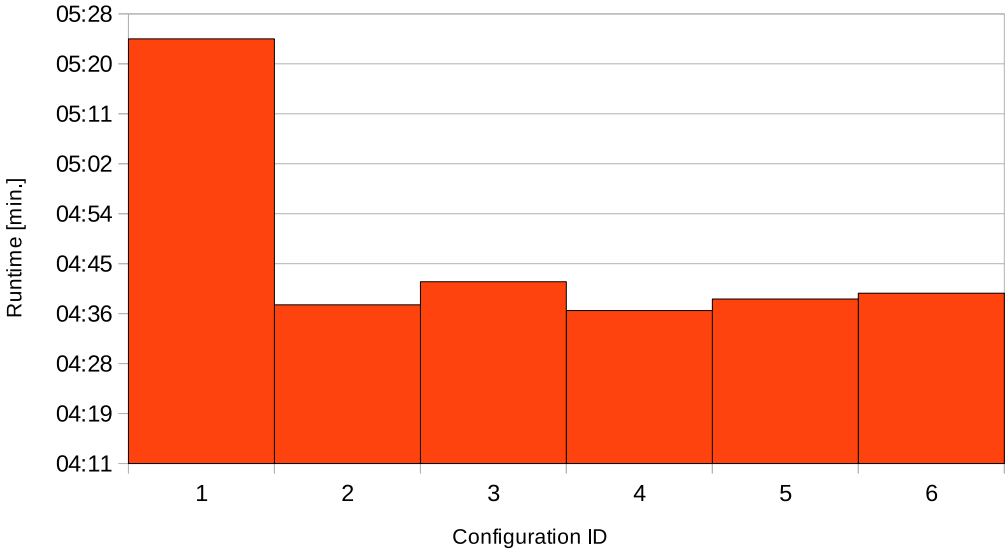


Figure 5.2.: Diagram to benchmark 2

ID	MPI	BLAS (`cblas-type`)	mpirun	`cpu_bind`	time [min.]
1	OpenMPI	OpenBLAS (serial)	-N 8 -n 384 –hint=multithread	default	04:42
2	OpenMPI	OpenBLAS (serial)	-N 7 -n 336 –hint=multithread	default	07:13
3	OpenMPI	OpenBLAS (serial)	-N 6 -n 288 –hint=multithread	default	06:23
4	OpenMPI	OpenBLAS (serial)	-N 5 -n 240 –hint=multithread	default	07:54
5	OpenMPI	OpenBLAS (serial)	-N 4 -n 192 –hint=multithread	default	09:35
6	OpenMPI	OpenBLAS (serial)	-N 3 -n 144 –hint=multithread	default	12:46
7	OpenMPI	OpenBLAS (serial)	-N 2 -n 96 –hint=multithread	default	18:12
8	OpenMPI	OpenBLAS (serial)	-N 1 -n 48 –hint=multithread	default	34:01

Table 5.3.: PyFR-Benchmarks 3

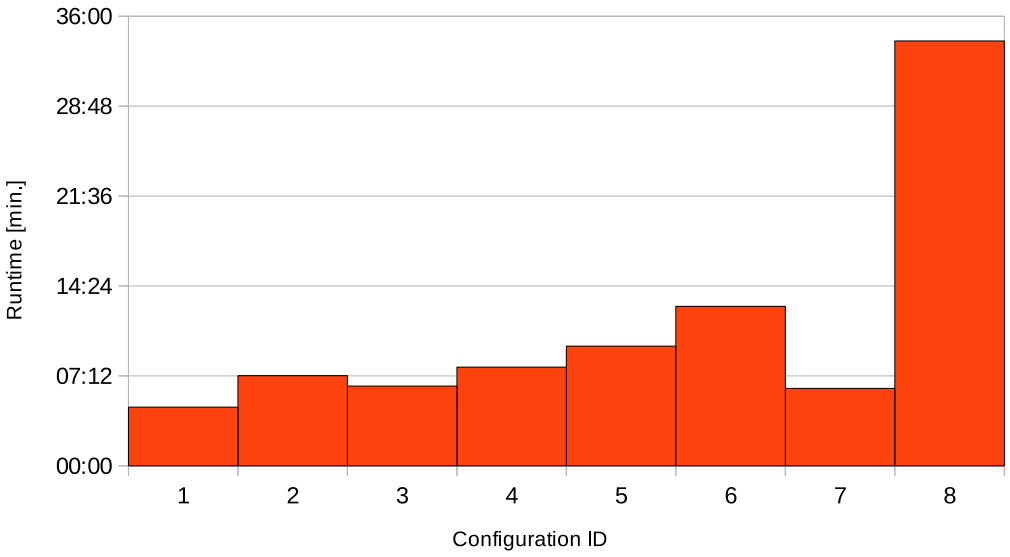


Figure 5.3.: Diagram to benchmark 3

ID	MPI	BLAS (`cblas-type`)	mpirun	`cpu_bind`	time [min.]
1	OpenMPI	OpenBLAS (serial)	-N 8 -n 192 –hint=multithread	cores	05:26
2	OpenMPI	OpenBLAS (serial)	-N 8 -n 384 –hint=multithread	default	05:12
3	OpenMPI	MKL (serial)	-N 8 -n 192 –hint=multithread	cores	05:51
4	OpenMPI	MKL (serial)	-N 8 -n 192 –hint=multithread	default	05:55
5	OpenMPI	MKL (serial)	-N 8 -n 384 –hint=multithread	default	05:19
6	OpenMPI	Atlas (serial)	-N 8 -n 192 –hint=multithread	cores	08:06
7	OpenMPI	OpenBLAS (serial)	-N 8 -n 384 –hint=multithread	default	04:48
8	OpenMPI	OpenBLAS (serial)	-N 8 -n 384	default	04:50
9	OpenMPI	OpenBLAS (serial)	-N 8 -n 192 –hint=multithread	cores	05:05
10	OpenMPI	OpenBLAS (serial)	-N 8 -n 384 –hint=multithread	default	04:46
11	OpenMPI	MKL (serial)	-N 8 -n 192 –hint=multithread	cores	05:31
12	OpenMPI	MKL (serial)	-N 8 -n 192 –hint=multithread	default	05:31
13	OpenMPI	MKL (serial)	-N 8 -n 384 –hint=multithread	default	04:59
14	OpenMPI	Atlas (serial)	-N 8 -n 192 –hint=multithread	cores	07:33
15	OpenMPI	Atlas (serial)	-N 8 -n 384 –hint=multithread	default	06:51
16	OpenMPI	OpenBLAS (parallel)	-N 8 -n 192 –hint=multithread	cores	05:51
17	OpenMPI	OpenBLAS (parallel)	-N 8 -n 384 –hint=multithread	default	04:47
18	OpenMPI	MKL (parallel)	-N 8 -n 192 –hint=multithread	cores	36:19
19	OpenMPI	MKL (parallel)	-N 8 -n 192 –hint=multithread	default	$>$ 35:00
20	OpenMPI	MKL (parallel)	-N 8 -n 384 –hint=multithread	default	04:57
21	OpenMPI	Atlas (parallel)	-N 8 -n 192 –hint=multithread	cores	07:29
22	OpenMPI	Atlas (parallel)	-N 8 -n 384 –hint=multithread	default	06:50

Table 5.4.: PyFR-Benchmarks 4

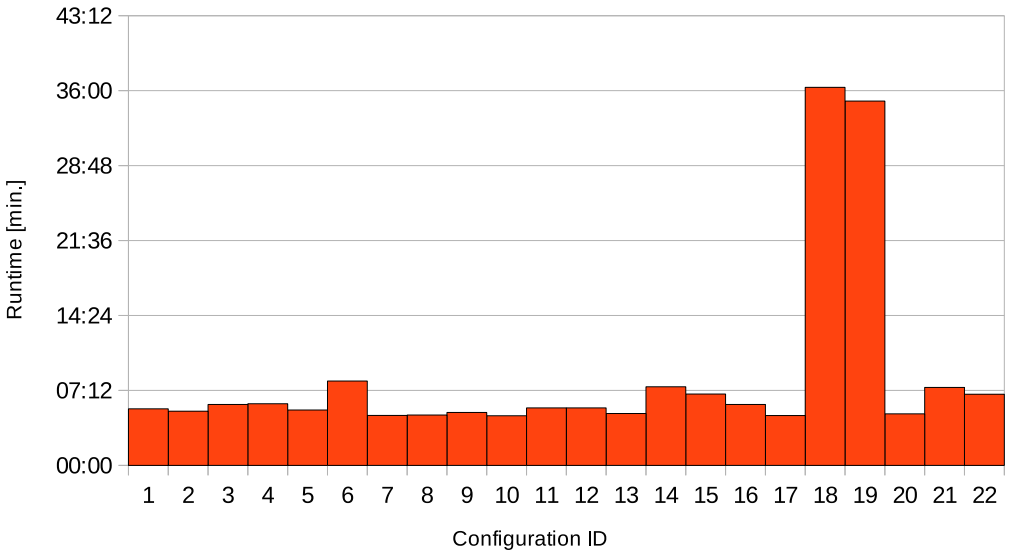


Figure 5.4.: Diagram to benchmark 4

5.2. Octopus

Author: Jonas Gresens

Figure 5.5.: Octopus logo

Octopus is a scientific program aimed at the ab initio virtual experimentation on a hopefully ever-increasing range of system types. Electrons are described quantum-mechanically within density-functional theory (DFT), in its time-dependent form (TDDFT) when doing simulations in time. Nuclei are described classically as point particles. Electron-nucleus interaction is described within the pseudopotential approximation.

For optimal execution performance Octopus is parallelized using MPI and OpenMP and can scale to tens of thousands of processors. It also has support for graphical processing units (GPUs) through OpenCL.[2]

5.2.1. Dependencies

We required the MPI-parallelized version of Octopus, which depends on several libraries:

- libxc is the library of exchange and correlation functionals, developed by the Octopus team. Version 2.0.x or 2.1.x is required.
- GSL: Octopus uses splines, complex numbers, special functions, and more from the GNU Scientific Library. Version 1.9 or later is required.
- LAPACK/BLAS: Octopus depends on an implementation of a LAPACK/BLAS library for its linear-algebra operations. See 3.1.2 for more info.
- FFTW is used to perform Fast Fourier Transforms. Version 3.0.0 or later is required.
- MPI: See 3.3.

The latest list of dependencies can be found at `http://www.tddft.org/programs/octopus/wiki/index.php/Manual:Installation#Requirements`

[2]Source: `http://www.tddft.org/programs/octopus/wiki/index.php/Main_Page`

5.2.2. How to Build

Octopus 4.1.2

We used the following flags to build octopus:

```
# compile software
FLAGS="--prefix=$prefix"
FLAGS+=" --with-blas=$blas_version/lib/libblas.so"
FLAGS+=" --with-libxc-prefix=$libxc_version"
FLAGS+=" --disable-zdotc-test --enable-mpi --enable-utils"
FCCPP="/lib/cpp -ansi" ./configure $FLAGS
make -j $(nproc)
```

SCC'15 Specific Version

The version used in the competition was a stable prerelease of version 5 of Octopus. We were advised to use revision 14111 from its SVN source code repository. This version required a slightly different build process:

We had to unpack and reconfigure the project on the head node ...

```
tar -xf octopus-5.0.x-r14111.tar.gz
cd 5.0.x
autoreconf -i
chmod 755 build-aux/*.pl
```

and build it on one of the compute nodes as the head node was deliberately not designed for such intensive tasks like compilation

```
FCCPP="/lib/cpp -ansi" ./configure --prefix=$PREFIX \
 --with-blas=\
/apps/modules/gcc/5/openblas/0.2/0.2.14/lib/libopenblas.a \
 --with-libxc-prefix=\
/apps/modules/gcc/5/libxc/2.1/2.1.2 \
 --with-netcdf-prefix=\
/apps/modules/gcc/5/mvapich2/2.1/netcdf-fortran/4.4/4.4.2 \
 --disable-zdotc-test \
 --enable-mpi \
 --enable-utils
make -j $(nproc)
```

5.2.3. How to Run

Octopus uses an input file named `inp` which has to reside in the same directory Octopus is called from.

Compiling Octopus with support for MPI generates the `octopus_mpi` executable, which we started using the a job script for SLURM:

```
#!/bin/sh

#SBATCH --time=600
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=24
#SBATCH --error=job.err --output=job.out

srun octopus_mpi
```

As well as the following shell script to do the module management and committing the job script:

```
#!/bin/sh

GCC="gcc/5"
MPI="mvapich2/2.1"
FFTW="fftw/3.3"
BLAS="openblas/0.2"

. /etc/profile.d/modules.sh
module purge
module load $GCC/gsl/1.16
module load $GCC/libxc/2.1/2.1.2
module load $GCC/$BLAS/0.2.14
module load $GCC/$MPI/2.1
module load $GCC/$MPI/$FFTW/3.3.4
module load mvapich-octopus-5.0.x-r14411
module list

MV2_USE_LAZY_MEM_UNREGISTER=0 sbatch octopus.job
```

5.2.4. Problems

C Preprocessor

Building Octopus uses the C preprocessor for C as well for Fortran code, which can cause the compilation to be aborted because of C comments, which are pasted into the Fortran code and interfere with Fortran's syntax. This problem was caused by the missing `-ansi` option in the C preprocessor call.

More detailed info can be found in the Octopus wiki:
`http://www.tddft.org/programs/octopus/wiki/index.php/Preprocessors`

MVAPICH2 with Multiple Nodes

Computing and visualizing the ground state of a benzene molecule (dataset from `http://www.tddft.org/programs/octopus/wiki/index.php/Tutorial:Benzene_molecule`) showed that the computation immediately generated wrong results after the first iteration. The errors were caused by the communication between multiple nodes, since they did not appear on only one node (compare fig. 5.6 and 5.7).

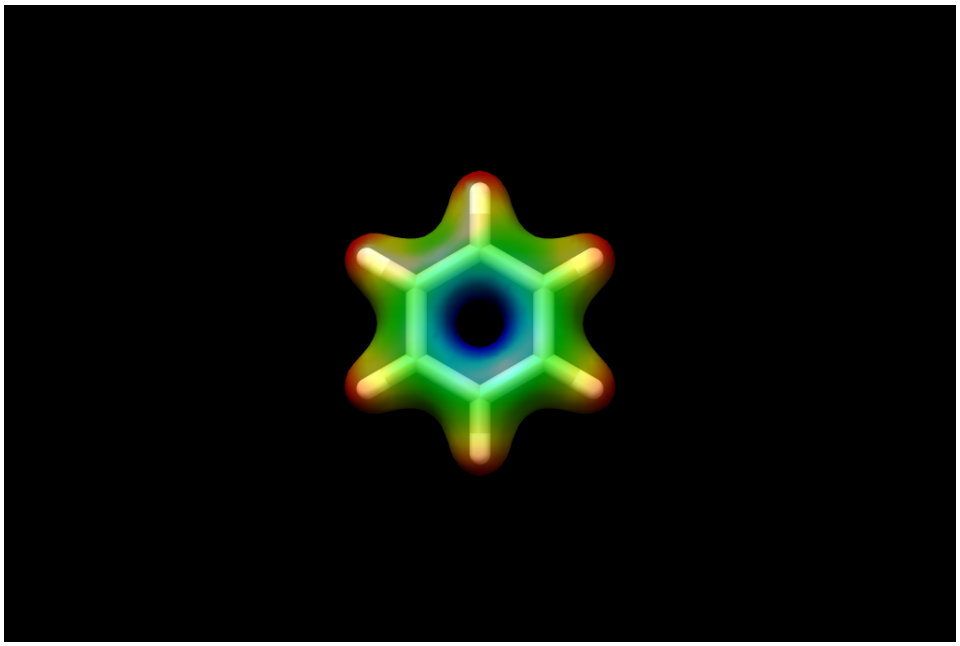


Figure 5.6.: benzene molecule with MVAPICH2 on one node

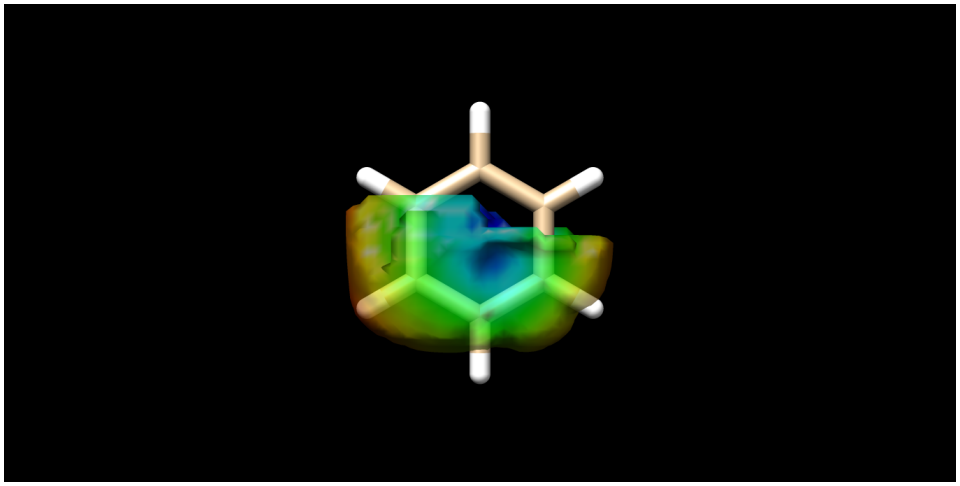


Figure 5.7.: benzene molecule with MVAPICH2 on two nodes

Using `MV2_USE_LAZY_MEM_UNREGISTER=0` fixed this problem (see 3.3.5).
The visualizations were created with UCFS Chimera.

5.2.5. Results

MPI

Our measurements showed that MVAPICH2 is around 15% faster than OpenMPI.

BeeGFS

Octopus usually writes a lot of data to the hard disks, these data may be used for program restarts and subsequent computations. We were able to speed up the write process by parallelizing the writing with BeeGFS. (for a comparison see table 5.5).

Filesystem	def	def_wf
ext4	3m 18.96s	14m 34.40s
BeeGFS	2m 05.65s	10m 11.81s

Table 5.5.: Comparison of BeeGFS and ext4

The runs produce different amounts of data:

- def writes 1 GB
- def_wf writes 10 GB

5.3. LAMMPS

Author: Lennart Braun

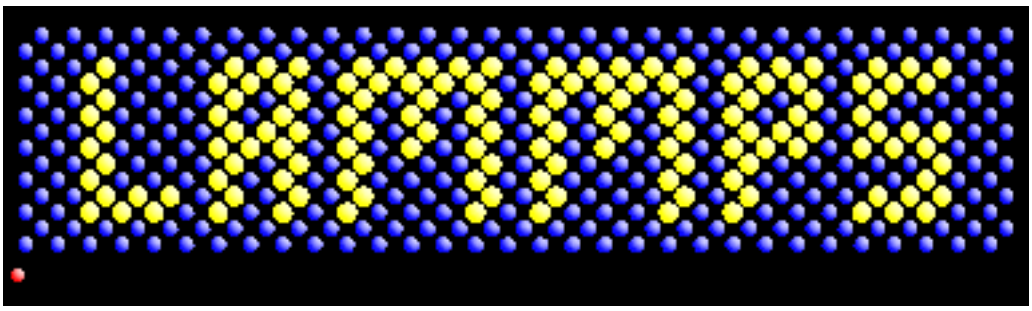


Figure 5.8.: LAMMPS logo

LAMMPS[3] is a simulator for atomic and molecular dynamics. It is written in the C++ programming language and uses MPI for distributed, massively parallel simulations. We used the LAMMPS version from May 15th 2015, which was the most recent at the time of the competition.

[3] `http://lammps.sandia.gov/`

Dependencies

- an MPI implementation
- an FFT library (optional, KISS FFT is included in the LAMMPS source code)
- libpng, libjpeg (optional): dumping images in the PNG and JPEG formats
- FFMPEG (optional): rendering videos

How to Build

There are two ways to build LAMMPS. First a `Makefile` can be written to meet the environment. The second option is to use the newer `Make.py` script. The complete process of building LAMMPS and optional packages can be done in a single line. We did use the following command to compile LAMMPS with the features required in the competition.

```
./Make.py -j 24 -m kokkos_omp -p kokkos reaxc reax \
          -mpi mpi -fft fftw3 -jpg yes -png yes -s ffmpeg \
          -o kraken_kokkos -a lib-all file exe
```

An explanation of the used command line flags:

`-j 24` Uses 24 processes to build (analog to make -j).

`-m kokkos_omp` Uses the `Makefile.kokkos_omp` as starting point.

`-p kokkos reaxc reax` Installs the packages kokkos, reaxc and reax.

`-mpi mpi` Assume that the compiler is a wrapper for MPI.

`-fft fftw3` Selects FFTW3 as FFT library.

`-jpg yes` Enables JPEG output.

`-png yes` Enables PNG output.

`-s ffmpeg` Enables the FFMPEG setting for video output.

`-o kraken_kokkos` The binary is called `lmp_kraken_kokkos`.

`-a lib-all file exe` All auxiliary libraries needed by the selected packages are installed (`lib-all`). A `Makefile.auto` is written (`file`) and the LAMMPS binary is compiled (`exe`).

`Make.py` includes documentation about all available options. It can be viewed using the -h switch.

Packages

A large amount packages are available which provide additional functionality to the LAMMPS core. Some packages enables the modeling of different physical properties, others help to run LAMMPS in different environments such as with OpenMP or with accelerator cards. Packages can be enabled at compile time with `make yes-package` of the `-p` flag of `Make.py` as shown above. Some packages require external libraries to be build prior to the LAMMPS compilation. Many of these are included in the LAMMPS source code archive and can be build with the above command if necessary.

The Kokkos Package

This package enables LAMMPS to make use of the Kokkos library. It contains a C++ abstraction layer to perform efficient calculations on different hardware. It supports OpenMP on multi-core processors and Xeon Phi accelerator cards as well as CUDA on NVIDIA graphic cards. Kokkos has to be enabled and configured at runtime. We used the following command during the competition.

```
1 export MV2_USE_LAZY_MEM_UNREGISTER=0
2 srun -N8 -n192 -o slurm_output ../src/lmp_kokkos_omp \
3      -i in.lj -l log.lammps -sf kk -k on t 1 -v kokkos 3
```

Explanation:

`-i in.lj` Selects `in.lj` (listing A.3) as input file.

`-l log.lammps` Writes log to `log.lammps`.

`-sf kk` Uses commands with the suffix `kk` (for Kokkos) if available.

`-k on t 1` Enables Kokkos with one thread per process.

`-v kokkos 3` Specifies a variable named kokkos with value 3.

The variable introduced with the last option is specific for the input files used in the competition and selects further options regarding Kokkos.

How to Run

A LAMMPS input script defines an experiment. An environment is modeled by a lattice and atoms are placed in it. Also files containing data about the molecular topology can be imported, various force fields and other physical parameters can be set, such as the initial velocity of the particles. The current state of the computation can be dumped as text file or rendered as an image. With the `run` command a number of steps are simulated.

LAMMPS reads commands from the standard input. Use I/O-redirection to let LAMMPS work on an input script: `[srun ...] lmp_krakken < in.file`. Alternatively the input file can be declared with the command line option `-in` (short `-i`). The input is parsed and executed line by line.

With the source code a number of example input files are distributed. These make use of different styles and packages. They are useful to test whether LAMMPS has been built correctly and runs without problems. For performance evaluation different benchmarks are supplied. They measure the efficiency of the parallelization and the results can be compared with published data[4] from other systems.

Competition

At the competition we were given two input files to run. These were slightly modified version of the Lennard-Jones (listing A.3) and the EAM (listing A.4) benchmarks included with LAMMPS.

[4] `http://lammps.sandia.gov/bench.html`

Bibliography

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, et al. *LAPACK Users' Guide: Third Edition*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, 1999.

[FLN+14] A. Fuchs, J. Lüttgau, J. Nissley, M. Tietz, and J. Weging. Student cluster competition 2014. Technical report, Universität Hamburg, 2014.

[GG] KAZUSHIGE GOTO and ROBERT A. VAN DE GEIJN. Anatomy of high-performance matrix multiplication. `https://www.cs.utexas.edu/users/pingali/CS378/2008sp/papers/gotoPaper.pdf`. last visited: 27.10.2015.

[LDK+] Piotr Luszczek, Jack Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. `http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf`. last visited: 18.12.2015.

[net] Netlib.org. `http://www.netlib.org/`. last visited: 25.10.2015.

[PyF15] Pyfr mailing list - example datasets. `https://groups.google.com/forum/#!msg/pyfrmailinglist/osp16U_0UCE/cZSVSjcZaI8J`, March 2015. last visited: 25.10.2015.

[Wit08] Tobias Wittwer. Choosing the optimal blas and lapack library. `http://www.wittwer.nl/wp-content/uploads/2009/08/blas_lapack.pdf`, March 2008. last visited: 25.10.2015.

List of Figures

1.1 Setup day in our bay 4
1.2 Our bay 5

2.1 Official power graph 8
2.2 Power consumption of different applications 9
2.3 QluMan config sets window 11
2.4 QluMan disk config 13
2.5 How OrangeFS works[5] 14
2.6 How BeeGFS works[6] 17
2.7 Module directory tree 23

3.1 Initial Benchmarks with MVAPICH and OpenMPI (Table 3.1 and 3.2) . 29
3.2 Benchmark of BLAS-libaries (Table 3.3) 31
3.3 Test of scalability of OpenBLAS (Table 3.4) 32

4.1 Comparison of STREAM kernel implementations 49
4.2 Comparison of FFT implementations 50

5.1 Diagram to benchmark 1 56
5.2 Diagram to benchmark 2 57
5.3 Diagram to benchmark 3 58
5.4 Diagram to benchmark 4 59
5.5 Octopus logo 60
5.6 benzene molecule with MVAPICH2 on one node 63
5.7 benzene molecule with MVAPICH2 on two nodes 63
5.8 LAMMPS logo 64

List of Tables

3.1 Initial benchmarks with MVAPICH 28
3.2 Initial benchmarks with OpenMPI 29
3.3 Benchmarks of BLAS-libraries 30
3.4 Test of scalability of OpenBLAS 31
3.5 Availability of various MPI implementations. 36

4.1 Comparison of STREAM kernel implementations 46
4.2 Comparison of FFT implementations 47

5.1 PyFR-Benchmarks 1 56
5.2 PyFR-Benchmarks 2 57
5.3 PyFR-Benchmarks 3 58
5.4 PyFR-Benchmarks 4 59
5.5 Comparison of BeeGFS and ext4 64

List of Listings

sources/2.2–Power/powerwatch.sh 8
sources/2.2–Power/powergraph.gnuplot 8
sources/2.2–Power/powergraph_looper.gnuplot 9
sources/2.2–Power/setpowerlimit.sh 9
sources/2.4–Filesystem/disk_config.txt 13
sources/2.4–Filesystem/orangefs–2.9.1.sh 15
sources/2.4–Filesystem/orangefs–init.sh 15
2.1 Install template 24

3.1 Excerpt of the HDF5 install script 34
3.2 Excerpt of the NetCDF install script 34
3.3 Excerpt of the NetCDF Fortran install script 35

4.1 Original STREAM kernels (without OpenMP) 45
4.2 Tuned STREAM Triad kernel 48

sources/5.2–Octopus/octopus.job 62
sources/5.2–Octopus/run–octopus.sh 62

A.1 The HPL.dat file used by us at SCC 15 72
A.2 Tuned STREAM kernels 73
A.3 LAMMPS input file: Lennard-Jones 76
A.4 LAMMPS input file: EAM 76

A. Additional Listings

Listing A.1: The HPL.dat file used by us at SCC 15

```
1  HPLinpack benchmark input file
2  Innovative Computing Laboratory, University of Tennessee
3  HPL.out      output file name (if any)
4  6            device out (6=stdout,7=stderr,file)
5  1            # of problems sizes (N)
6  327680       Ns
7  1            # of NBs
8  256          NBs
9  0            PMAP process mapping (0=Row-,1=Column-major)
10 1            # of process grids (P x Q)
11 12             Ps
12 16             Qs
13 16.0         threshold
14 1            # of panel fact
15 2            PFACTs (0=left, 1=Crout, 2=Right)
16 1            # of recursive stopping criterium
17 4            NBMINs (>= 1)
18 1            # of panels in recursion
19 2            NDIVs
20 1            # of recursive panel fact.
21 1            RFACTs (0=left, 1=Crout, 2=Right)
22 1            # of broadcast
23 1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
24 1            # of lookahead depth
25 1            DEPTHs (>=0)
26 2            SWAP (0=bin-exch,1=long,2=mix)
27 64           swapping threshold
28 0            L1 in (0=transposed,1=no-transposed) form
29 0            U  in (0=transposed,1=no-transposed) form
30 1            Equilibration (0=no,1=yes)
31 8            memory alignment in double (> 0)
32 ##### This line (no. 32) is ignored (it serves as a separator). ######
33 0                               Number of additional problem sizes
   ↪ for PTRANS
34 1200 10000 30000                values of N
35 0                               number of additional blocking sizes
   ↪ for PTRANS
36 40 9 8 13 13 20 16 32 64        values of NB
```

Listing A.2: Tuned STREAM kernels

```
void tuned_STREAM_Copy()
{
  __asm__ (
    "mov %[vs], %%rax                    \n" // n
    "sub $4, %%rax                       \n" // n - 4
    "lea (%[a], %%rax, 8), %%rsi         \n" // &a[n-4]
    "lea (%[c], %%rax, 8), %%rdi         \n" // &c[n-4]
    "neg %%rax                           \n" // -(n - 4)
    "jg 2f                               \n" // skip if n < 4

    "1:                                  \n" // begin of loop
    "vmovapd (%%rsi, %%rax, 8), %%ymm0   \n" // load 4 doubles from a
    "vmovntpd %%ymm0, (%%rdi, %%rax, 8)  \n" // store 4 doubles to c
    "add $4, %%rax                       \n" // next 4 doubles
    "jl 1b                               \n" // next iteration

    "2:                                  \n"
    "sub $4, %%rax                       \n" // remainder
    "jns 4f                              \n" // done

    "3:                                  \n"
    "vmovsd 32(%%rsi, %%rax, 8), %%xmm0  \n" // load 1 double from a
    "vmovsd %%xmm0, 32(%%rdi, %%rax, 8)  \n" // store 1 double to c
    "add $1, %%rax                       \n" // next 4 doubles
    "js 3b                               \n" // done

    "4:                                  \n" // end
    : // no output operands
    : [a] "r" (a), [c] "r" (c), [vs] "m" (VectorSize) // input operands
    : "%ymm0", "%xmm0", "%rax", "%rsi", "%rdi" // clobbered registers
  );
}

void tuned_STREAM_Scale(double scalar)
{
  __asm__ (
    "mov %[vs], %%rax                    \n" // n
    "sub $4, %%rax                       \n" // n - 4
    "lea (%[c], %%rax, 8), %%rsi         \n" // &c[n-4]
    "lea (%[b], %%rax, 8), %%rdi         \n" // &b[n-4]
    "neg %%rax                           \n" // -(n - 4)
    "vbroadcastsd %[scl], %%ymm1         \n" // scalar
    "jg 2f                               \n" // skip if n < 4

    "1:                                  \n" // begin of loop
    "vmovapd (%%rsi, %%rax, 8), %%ymm0   \n" // load 4 doubles
    "vmulpd %%ymm1, %%ymm0, %%ymm2       \n" // scalar multiplication
    "vmovntpd %%ymm2, (%%rdi, %%rax, 8)  \n" // store 4 doubles
    "add $4, %%rax                       \n" // next 4 doubles
    "jl 1b                               \n" // next iteration

    "2:                                  \n"
    "sub $4, %%rax                       \n" // remainder
    "jns 4f                              \n" // done

    "3:                                  \n"
    "vmovsd 32(%%rsi, %%rax, 8), %%xmm0  \n" // load 1 double from a
    "vmulsd %%xmm1, %%xmm0, %%xmm2       \n" // scalar multiplication
    "vmovsd %%xmm2, 32(%%rdi, %%rax, 8)  \n" // store 1 double to c
    "add $1, %%rax                       \n" // next 4 doubles
    "js 3b                               \n" // done

    "4:                                  \n" // end
    : // no output operands
```

```
65      : [b] "r" (b), [c] "r" (c), [vs] "m" (VectorSize), [scl] "m" (scalar) // input operands
66      : "%ymm0", "%ymm1", "%ymm2", "%xmm0", "%xmm1", "%xmm2", "%rax", "%rsi", "%rdi" // clobbered registers
67    );
68  }
69
70  void tuned_STREAM_Add()
71  {
72    __asm__ (
73      "mov %[vs], %%rax                    \n" // n
74      "sub $4, %%rax                       \n" // n - 4
75      "lea (%[a], %%rax, 8), %%rsi         \n" // &a[n-4]
76      "lea (%[b], %%rax, 8), %%rcx         \n" // &b[n-4]
77      "lea (%[c], %%rax, 8), %%rdi         \n" // &c[n-4]
78      "neg %%rax                           \n" // -(n - 4)
79      "jg 2f                               \n" // skip if n < 4
80
81      "1:                                  \n" // begin of loop
82      "vmovapd (%%rsi, %%rax, 8), %%ymm0   \n" // load 4 doubles
83      "vmovapd (%%rcx, %%rax, 8), %%ymm1   \n" // load 4 doubles
84      "vaddpd %%ymm1, %%ymm0, %%ymm2       \n" // addition
85      "vmovntpd %%ymm2, (%%rdi, %%rax, 8)  \n" // store 4 doubles
86      "add $4, %%rax                       \n" // next 4 doubles
87      "jl 1b                               \n" // next iteration
88
89      "2:                                  \n"
90      "sub $4, %%rax                       \n" // remainder
91      "jns 4f                              \n" // done
92
93      "3:                                  \n"
94      "vmovsd 32(%%rsi, %%rax, 8), %%xmm0  \n" // load 1 double from a
95      "vmovsd 32(%%rcx, %%rax, 8), %%xmm1  \n" // load 1 double from b
96      "vaddpd %%xmm1, %%xmm0, %%xmm2       \n" // scalar multiplication
97      "vmovsd %%xmm2, 32(%%rdi, %%rax, 8)  \n" // store 1 double to c
98      "add $1, %%rax                       \n" // next 4 doubles
99      "js 3b                               \n" // done
100
101     "4:                                  \n" // end
102     : // no output operands
103     : [a] "r" (a), [b] "r" (b), [c] "r" (c), [vs] "m" (VectorSize) // input operands
104     : "%ymm0", "%ymm1", "%ymm2", "%xmm0", "%xmm1", "%xmm2", "%rax", "%rcx", "%rsi", "%rdi" // clobbered registers
105   );
106 }
107
108 void tuned_STREAM_Triad(double scalar)
109 {
110   __asm__ (
111     "mov %[vs], %%rax                    \n" // n
112     "sub $4, %%rax                       \n" // n - 4
113     "lea (%[a], %%rax, 8), %%rdi         \n" // &a[n-4]
114     "lea (%[b], %%rax, 8), %%rcx         \n" // &b[n-4]
115     "lea (%[c], %%rax, 8), %%rsi         \n" // &c[n-4]
116     "neg %%rax                           \n" // -(n - 4)
117     "vbroadcastsd %[scl], %%ymm3         \n" // scalar
118     "jg 2f                               \n" // skip if n < 4
119
120     "1:                                  \n" // begin of loop
121     "vmovapd (%%rsi, %%rax, 8), %%ymm0   \n" // load 4 doubles (c)
122     "vmovapd (%%rcx, %%rax, 8), %%ymm1   \n" // load 4 doubles (b)
123     ".intel_syntax noprefix              \n" //
124     "vfmadd231pd %%ymm1, %%ymm3, %%ymm0  \n" // fma
125     ".att_syntax prefix                  \n" //
126     "vmovntpd %%ymm1, (%%rdi, %%rax, 8)  \n" // store 4 doubles
127     "add $4, %%rax                       \n" // next 4 doubles
```

```
128     "jl 1b                                \n" // next iteration
129
130     "2:                                   \n"
131     "sub $4, %%rax                        \n" // remainder
132     "jns 4f                               \n" // done
133
134     "3:                                   \n"
135     "vmovsd 32(%%rsi, %%rax, 8), %%xmm0 \n" // load 1 double from a
136     "vmovsd 32(%%rcx, %%rax, 8), %%xmm1 \n" // load 1 double from b
137     ".intel_syntax noprefix               \n" //
138     "vfmadd231sd %%xmm1, %%xmm3, %%xmm0 \n" // fma
139     ".att_syntax prefix                   \n" //
140     "vmovsd %%xmm1, 32(%%rdi, %%rax, 8) \n" // store 1 double to c
141     "add $1, %%rax                        \n" // next 4 doubles
142     "js 3b                                \n" // done
143
144     "4:                                   \n" // end
145     : // no output operands
146     : [a] "r" (a), [b] "r" (b), [c] "r" (c), [vs] "m" (VectorSize), [scl] "m" (scalar)
          ↪ // input operands
147     : "%ymm0", "%ymm1", "%ymm2", "%xmm0", "%xmm1", "%xmm2", "%rax", "%rcx", "%rsi",
          ↪ "%rdi" // clobbered registers
148   );
149 }
```

Listing A.3: LAMMPS input file: Lennard-Jones

```
# 3d Lennard-Jones melt

variable        kokkos index 0

if "${kokkos} == 1" then &
  "package      kokkos neigh half/thread comm/forward device comm/exchange host" &
  "newton off"
if "${kokkos} == 2" then &
  "package       kokkos neigh full comm/forward device comm/exchange device" &
  "newton off"
if "${kokkos} == 3" then &
  "package       kokkos neigh full comm/forward host comm/exchange host" &
  "newton off"
if "${kokkos} == 4" then &
  "package       kokkos neigh full comm/forward host comm/exchange no" &
  "newton off"
if "${kokkos} == 5" then &
  "package       kokkos neigh half/thread comm/forward no comm/exchange no" &
  "newton off"
if "${kokkos} == 6" then &
  "package       kokkos neigh half/thread comm/forward no comm/exchange no" &
  "newton on"
if "${kokkos} == 7" then &
  "package       kokkos neigh half/thread comm/forward device comm/exchange device" &
  "newton on"

units           lj
atom_style      atomic

lattice         fcc 0.8442
region          box block 0 64 0 64 0 64
create_box      1 box
create_atoms    1 box
mass            1 1.0

velocity        all create 1.44 87287 loop geom

pair_style      lj/cut 2.5
pair_coeff      1 1 1.0 1.0 2.5

neighbor        0.3 bin
neigh_modify    delay 0 every 20 check no

fix             1 all nve
thermo          100
run             20000
```

Listing A.4: LAMMPS input file: EAM

```
# bulk Cu lattice
variable        kokkos index 0
# System size probable want a 7 here
variable        rep index 7
# Number of timesteps want 10000 here to get some MSD data
variable        nsteps index 10000

if "${kokkos} == 1" then &
  "package      kokkos neigh half/thread comm/forward device comm/exchange host" &
  "newton off"
if "${kokkos} == 2" then &
  "package       kokkos neigh full comm/forward device comm/exchange device" &
  "newton off"
if "${kokkos} == 3" then &
  "package       kokkos neigh full comm/forward host comm/exchange host" &
```

```
  "newton off"
if "${kokkos} == 4" then &
  "package       kokkos neigh full comm/forward host comm/exchange no" &
  "newton off"
if "${kokkos} == 5" then &
  "package       kokkos neigh half/thread comm/forward no comm/exchange no" &
  "newton off"
if "${kokkos} == 6" then &
  "package       kokkos neigh half/thread comm/forward no comm/exchange no" &
  "newton on"
if "${kokkos} == 7" then &
  "package       kokkos neigh half/thread comm/forward device comm/exchange device" &
  "newton on"

variable    xx equal 20*${rep}
variable    yy equal 20*${rep}
variable    zz equal 20*${rep}

units       metal
atom_style  atomic

# Sort Frequency and binsize can affect cache locality a lot
atom_modify     map array sort 100 4.0

# MPI rank layout for domain decomposition can have significant influence on
    ↪ communication cost
processors * * *
lattice     fcc 3.615
region      box block 0 ${xx} 0 ${yy} 0 ${zz}
create_box  1 box
create_atoms    1 box

pair_style  eam
pair_coeff  1 1 Cu_u3.eam

velocity    all create 1600.0 376847 loop geom

# Modify skin distance and requency of neighbor list building (also potentially bin
    ↪ size)
# This can have significant performance influence: but if set too agressive
    ↪ invalidates the run (i.e. produces wrong results)
neighbor        1.0 bin
neigh_modify    delay 0 check no every 20

fix     1 all nve

timestep    0.005
thermo      50

run     ${nsteps}
```