Proposal for ASC<year>

Team <team name>

August 2, 2017

1 Introduction

We hereby apply for a spot in the <year> ASC Student Super Computing Challenge in <location>, China. On the following pages we present our team and university along with our prospective hardware and results of the preliminary challenges.

group-picture.jpg

2 Supercomputing activities

The supercomputing activities of the <institute> are centralized in the <local supercomputing center>. It provides several hardware-platforms to work on, e.g. <name> Compute-Cluster with a throughput of 10.4 TFlop/s, the <name> Compute-Cluster (XY.Z TFlop/s) and currently largest and fastes cluster, <name>, with a throughput of XZY TFlop/s, consisting of <many> compute nodes with twenty Intel Xeon E5-XXXXvY cores each.

Furthermore, the <local supercomputing center> provides access to GPU- and Windows-Clusters, an HPC storage system and the HPC systems of <another large supercomputing center>. To ease selection of and switching between different versions of software packages, all

linux-based HPC systems use the modules system[1]. It allows to conveniently load the necessary configurations for different programs or versions of the same program and, if necessary, unload them again later. For Batch processing all of the newer HPC clusters run under the control of Slurm.

For students interested in the field of supercomputing the `<institute>` provides different lectures and modules like "`<name of lecture>`", "`<name of lecture>`", "`<name of seminar>`" and the "`<name of practial>`", in which students learn to plan, launch and maintenance HPC systems and compete at the Student Cluster Competition (SCC) in USA and Germany.

The `<local supercomputing center>` itself conducts active research in various fields of supercomputing. To mention are e.g. performance evaluations of modern processors to determine the ideal computer architecture for every user[2] or the engagement in developing a testbed, especially with tests about interactive HPC (Computational Steering), for future generations of networks[3].

A key achievement in our supercomputing research for sure is the work on optimization for stencil algorithms on multi-core systems. This project develops optimization techniques for stencil-based algorithms like the Jacobi and Gauss-Seidel smoothers and the Lattice-Boltzman-Method (LBM). While much work has been done in the past in this field, the current trend towards multi.core chips with complex cache topologies requires a re-evaluation of existing approaches and the development of new ideas that put the specific features of those processors to use. Thus the researchers of the `<local supercomputing center>` analyzed the potentials of temporal blocking for stencil-based computations. The master thesis of `<name>` is concerned with several different variants of temporal blocking, with a special emphasis on leveraging shared caches in multi-core environments and shows how temporal blocking can be employed on distributed-memory parallel computers. `<Researchers>` also did a approach of pipelined wavefront parallelization for stencil-based computations. By re-using data from cache in the successive wavefronts this multicore-aware parallelization strategy employs temporal blocking in a simple and efficient way, proved on different Intel®x86 quad- and hexa-core processors.

3 Team introduction

Our team represents the `<name of university>`, the second largest University `<somewhere in the world>`. Students from `<this university>` have previously participated in `<various student cluster competitions>`.

After the last competition in November `<last year>`, three of the six members decided to propose as a team for the ASC SCC. To enhance diversity and inter-academic collaboration the team was completed by a girl and a student from `<another university in the region>`. The fitting slogan for our team is "`<motivational quote>`".

Two members of our team, `<name>` and `<name>`, are studying Computer Science (CS), which covers a large range of topics, and provides a solid foundation for programming, cluster computing and system administration.

`<name>` is studying `<no computer science>`, a degree course that combines computer science, numerical and applied mathematics and a wide range of engineering disciplines. High performance computing and programming in general play an important role in `<this degree course>`, especially focused on the real world problems of the engineering domain.

[1] cf. https://modules.sourceforge.net
[2] cf. https://website.html
[3] cf. https://website.shtml

`<name>` is studying `<not computer science either>`. A relatively new degree course, especially focusing on the increasing use of supercomputers in medical applications. Students, already at an very early stage in their studies, learn how to use programs such as Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) and Matlab in order to handle the upcoming challenges in their field.

Finally `<name>` is studying Mathematics, which provides the team not only with a specialist knowledge of mathematical theories, tools and practices, often needed in academic applications in the HPC field, but also with advanced understanding of mathematical and technical language and the ability to analyze and interpret large quantities of data.

4 Technical proposal

4.1 HPC system design

4.1.1 Hardware

Table 1 shows the key specifications of our system.

Table 1: Cluster specifications

Node	16× Inspur NF5280M4 server
CPU	Dual Socket Intel®Xeon®E5-2660 v4 (35 MB Cache, 2.00 GHz)
RAM	16 × 128 GB
Disk Space	16 × 1000 GB HDD
Interconnect	Infiniband FDR optical fiber cable, QSFP Port, 56 Gbit
Switch	Mellanox SX6036 36-port 56Gb/s InfiniBand Switch System
HCA-Card	Mellanox ConnectX®-3 Single-Port QSFP Adapter, FDR-IB
Total RAM	2,048 GB
Disk Space	16,000 GB
Peak FLOP/s	14,336 DP GFLOPS
Peak Bandwidth	1,229 GB/s

The system is equipped with 16 Inspur NF5280M4 servers. A total of 16 servers with a peak power consumption of 300 W each would exceed the power limit of 3000 W.

Considering these numbers one would have only been able to incorporate a total of 9 servers, yet our benchmarking shows that the performance of 16 servers can be exploited for some applications, e.g., the HPCG. For these in order to achieve all nodes can be set to a lower, hence less power consuming frequency. We already applied this technique at the SCC in `<a city in the USA>`, thus, the team is experienced in dynamic clocking of single nodes in a cluster.

4.1.2 Software

Operating System
CentOS7 provides a solid computing platform and is prominently used in the clusters at the university, allowing the team to benefit from the experience and expertise gained by the administrators who manage universities clusters. Because of CentOS is rpm-based and has a big driver package from RedHat it will be the OS with the most functionality the team needs.

Ansible
Ansible in an agent-less configuration management tool, designed to deploy configurations at large scale. Deployments are described in so-called Playbooks, which describe a list of tasks that have to be achieved in order to reach a desired configuration state. A task can be anything one could do manually, e.g. add a user, delete a file, update or install a software package. Tasks are handled by modules, which wrap around common system utilities and are capable of triggering a change on the remote system in case it is necessary.

SLURM
SLURM is a job scheduling system and workload manager, used by 60% of the TOP500 computers. It provides the team with the ability to exclusively allocate the nodes and run them with specific configurations. This is especially useful as we need carefully examined and tested configurations for the benchmarks to stay inside the power limit. SLURM allows us to do this with ease, as we have the opportunity to prepare a jobscript which will ensure this.

Hardware configuration tools
As a powerful tool for monitoring and benchmarking we use `<a really good tool>`, developed by a team from the HPC-Group at the `<local supercomputing center>`. For example it provides CPU-information, or the usage of the CPU-cache, RAM-Usage and many more. For setting the uncore frequency we use the tool model specific register (msr).

4.2 HPL

4.2.1 Software environment

The software used was provided by the Knights Landing (KNL) Cluster system and was based on a CentOs 7.2 which is very common on HPC Systems and used in the clusters from our local data center. Apart from that we used the Intel Composer XE Suites 2017.1.043 with the Intel Math Kernel Library (MKL) for compiling and linking since it can compile highly optimized code for the newest Intel architectures including AVX-512 optimisations. Even though we did not use the Message Passing Interface (MPI) for our best run we needed the MPI Library for linking and so we used the Intel MPI 2017 Update 1 Build 20161016. For scheduling jobs PBS was used. Sadly, the cluster architecture was not the same as described in the KNL Cluster Information provided by the competition which led into some problems through the entire process. Also the limitations made in the PBS configuration was limiting the possibility to compare the single-node-performance. There was no software environment system installed and the paths to the important libraries mentioned above were not set, so we needed to add them manually.

4.2.2 Optimization process

As we competed at the SC`<year>` we already had a general knowledge about working with the HPL benchmark. To gain some general ideas about the difference between the Netlib HPL and the highly optimized Intel binaries we started building and testing the Netlib HPL in our local datacentre on an Intel Xeon 2660v2 “Ivy Bridge” based system. We were able to achieve around 95% of theoretical maximum peak performance by following the Intel compile guide for the Netlib HPL benchmark which was nearly the same value the Intel mp_linpack benchmark achieved. Being aware of this solution, we started working on the KNL platform following the Intel guide. Unfortunately, we were not able to achieve any noteworthy speedup following the

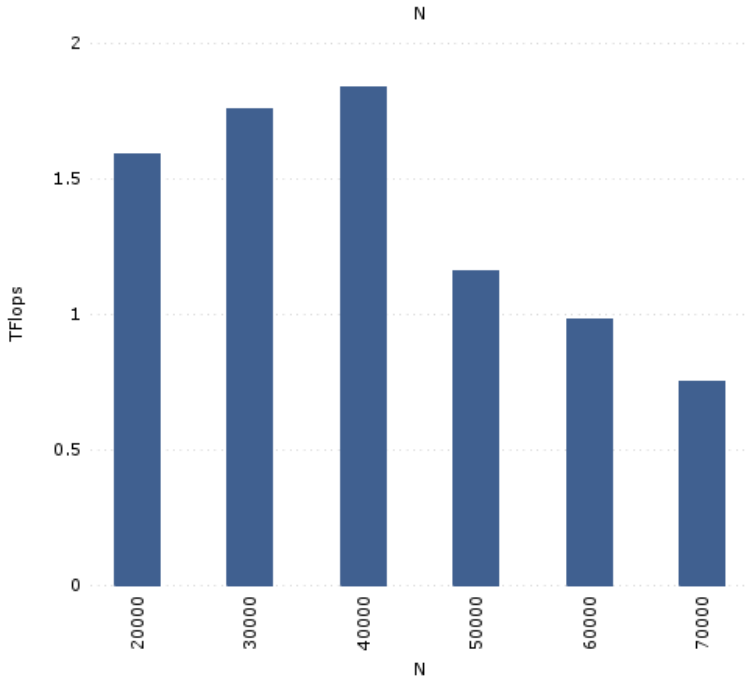

Figure 1: rough N scaling run

Intel instructions. After various attempts of achieving a good performance with the naive Netlib HPL implementation we decided to use the Intel binaries to score good benchmark result.

4.2.3 Performance Estimation

The theoretical peak performance of the processor can easily be calculated by $64[cores] \times 32[tiles/core] \times 1.3\,GHz = 2620\,GFlops$. On a common Intel 64 CPU it is easy to get close to the theoretical peak value using the mp_linpack binary and optimizing the input value but we expect the KNL-System to perform much worse due to overhead and memory limitations.

4.2.4 Performance optimization methods

Compiler:
The most intensive part of the benchmark is a matrix-matrix-multiplication (dgemm) which is part of the cblas interface. There are various optimized implementations for different systems and configurations and MKL has an interface to it as well. Unfortunately, we were not able to improve the performance of the Netlib HPL by using the MKL cblas library, probably caused by a wrong usage of the library or a wrong path configuration. Aside of using the cblas interface the compiler is optimizing the code on many other levels like loop-unrolling or vectorizing with the new AVX-512 intrinsic.

Runtime optimization:
After switching to the Intel binary we were able to start the input-parameter based performance optimization. First of all we deiced not to use hyper threading but thread pinning by setting the

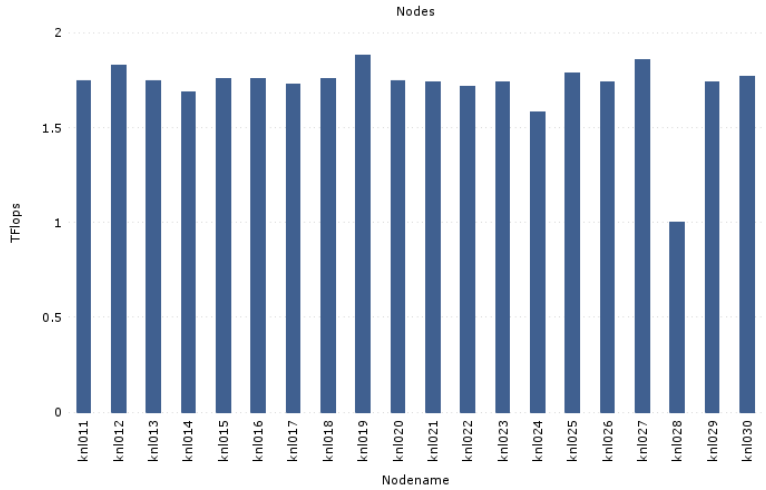

Figure 2: comparing all nodes

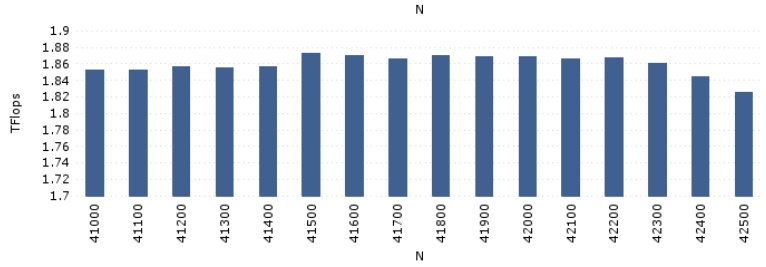

Figure 3: finely graduated N scaling run (baseline is not 0)

correlated environmental variables.

```
export OMP_NUM_THREADS=64
export MKL_NUM_THREADS=64
export KMP_AFFINITY=scatter
```

We started running a basic scaling run to determinate the best N to start with on a random node (1). We discovered that the best value for running a HPL on this KNL system is using an $N \approx 40000$ that just fits into the MCDRAM of the Processor, which is configured as a part of the main memory and not as a cache. After defining a rough N we started to compare the different KNL nodes with some alarming results (2). The best node was nearly two times faster than the slowest node. We were septic about that result, but it was reproducible on various runs. Hence, we decided to use the knl019 node from now on. Having the fastest node on the system we now started a finely graduated scaling run about N to figure out the perfect N for out final Linpack run (3). There is just a small difference between in number but we still decided to use $N = 41500$ for our final benchmark run. After having determined our N, we started scaling about NB and decided to scale about all NBs provided by Intel for there processors (4). We discovered that the $NB = 336$, which is recommended by Intel for this Processor, is not giving the best performance.

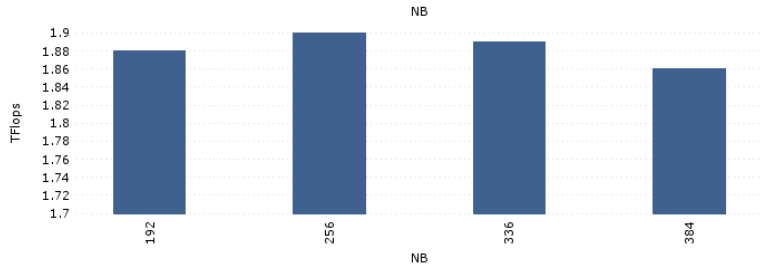

Figure 4: NB scaling

4.2.5 Algorithm

The Algorithm performed in the HPL benchmark is the basic arithmetician operation of solving a system of linear equation $A \times x = b$. It is performed by first computing the LU factorization of A with a row partial pivoting and then substitute this solution backwards. After the substitution the timer is stopped and the solution is checked with a norm-wise backward error. As we are not using MPI on our single-node run, we are not splitting the Matrix over multiple processes, but still dividing it into small blocks by the size of nb to use the cache blocking optimization. Further on we also don't have to worry about broadcasting panels because we are working with shared memory and not with message passing.

4.2.6 Result

After the optimization we achieved a performance of 1.91368 TFlops on one Node. This is about 72% of the theoretical peak performance and an appropriate value. Unfortunately we are not able to find any HPL benchmark values of this Processor to compare to.

4.3 MASNUM_WAVE

4.3.1 Software environment

The Wave Numerical Model MASNUM_WAVE was executed on the Sunway TaihuLight System. The operating system of the TaihuLight is Sunway RaiseOS 2.0.5. For the final runs of the test data we used the MPI-Fortran compiler `mpifort for SW` v. 2.2a with `SWCC Compiler` v. 5.421-sw-485. Further we used the `netcdf` library for reading the boundary conditions and generating the data points. For submitting the runs we allocated 16 nodes using 4 MPE-cores each. The variable `share_size` was set to 6500 MB and the host stack was set to 1024 MB.

For profiling and analyses of our tests we additionally used `swafort swacc` v. 2.0 for OpenACC and `sw5f90.new` with `SWCC Compiler` v. 5.421-sw-485. Further GNU gprof v. 2.20.51.0.2-5.42.e16 was used for analyses.

4.3.2 Optimization process

First of all we identified the methods in MASNUM_wave, which need the relatively most time in execution. For this we used `gprof` on a ten-day-evaluation of the western pacific ocean data set. In Fig. 5 we can see the relative runtime of the ten greediest methods. We decided to concentrate on `PROPAGATE` and `IMPLSCH` for optimization, due to the fact they cumulatively need 25.62 % + 18.69 % = 44.31 % of the total runtime, which was 156 min. For our next step we implemented OpenACC for `propagate.inc`. We set the OpenACC pragma in front of the outer loop:

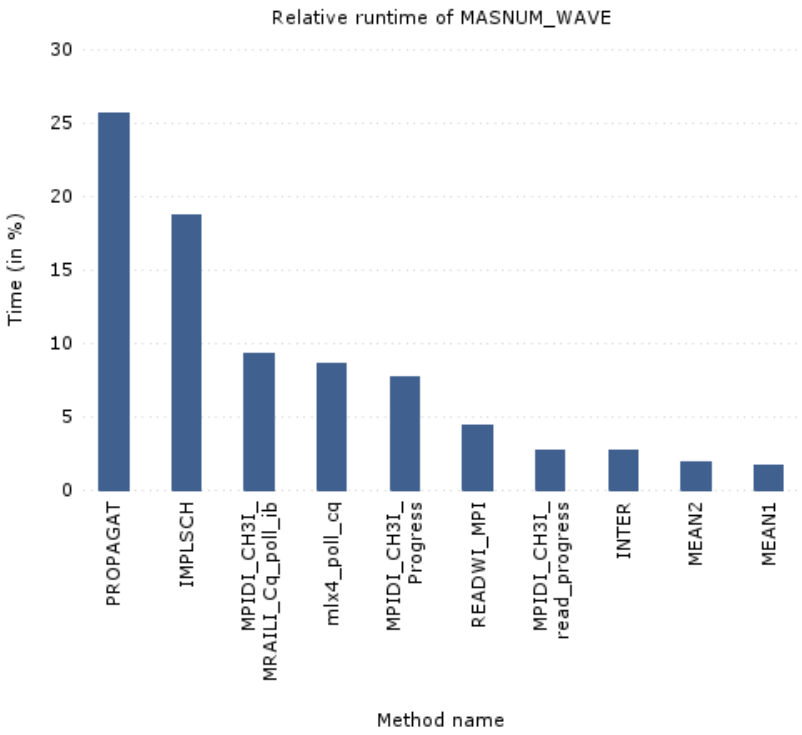

Figure 5: Relative runtime of MASNUM_WAVE of the ten greediest methods.

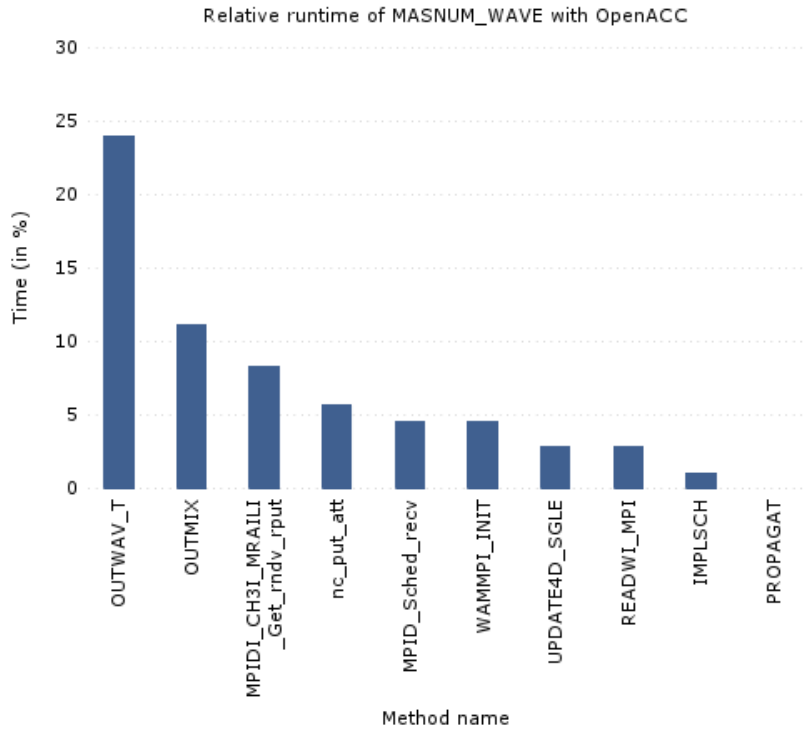

Figure 6: Relative runtime of MASNUM_WAVE with OpenACC of the ten greediest methods.

```
!$ACC PARALLEL LOOP COPYIN(nsp,x,y,d,deltx,delty,dddx,dddy,uxx,uxy,uyx,uyy,thet,wk
    ,wf,ccg,rslat,ee) COPYOUT(e) LOCAL(ia,ic,j,k,i,iyyz,ixx,jyy,ixx1,jyy1,iwk,iwk1
    ,i1,jth,jth1,x0, y0,d0,dddx0,dddy0,duxdx0,duxdy0,duydy0,duydx0,th0,sinth,costh
    ,wk0,ws0,dk0,cg,cgx,cgy,xx,yy,x1,x2,y1,y2,dsidd,ssr1,ssr2,ssrwk,ssrth,wks,fien
    ,fien1,wk1,wk2,dtth0,ths,e11,e12,e13,e14,e1,e21,e22,e23,e24,e2,e32,e33,e34,e3,
    e41,e42,e43,e44,e4,e31,th1,th2,exxyy,x_d,y_d,dx,dy,iii)
do 100 ia=ixs,ixl
do 100 ic=iys,iyl

if(nsp(ia,ic).ne.1) cycle

x0=x(ia)
y0=y(ic)
d0=d(ia,ic)

        !Some code...
        !Some more code...

e(k,j,ia,ic)=max(exxyy, small)

500 continue
200 continue
100 continue
!$ACC END PARALLEL LOOP
```

The gprof run afterwards gave us some interesting information, listed in Fig.6. As expected the relative runtime of `PROPAGATE` is minimal, also the runtime of `IMPLSCH` on the MPE-core

sank. Unfortunately the absolute runtime of MASNUM_WAVE stayed the same. Gprof analyses in the CPE-cores showed that more than a third (35 %)of the total computation time is wasted by waiting for tasks. Furthermore the MPI method `MPIDI_CH3I_MRAILI_Get_rndv_rput` both in the standard and the OpenACC version as the third most time consuming method points out the boundaries of MASNUM in communication and I/O.

Thus, we decided to additionally enhance the MPI communication in the program. For this we used the `mpifort for SW` with `autoSwap` option and compared the results with our previous observations. The algorithm now has a improvement in speed of almost 300 % and only needs 34 % of the standard time for simulating ten days, which is 53 min.

We finally used this configuration for the two examples of the pacific and global ocean surface wave model. Example 1 needed 4:58 hrs, example 2 6:09 hrs. After compiling the validation program with the correct `netcdf` library from the `netcdf_0816` directory and the `gfortran` compiler, the validation succeed for both examples.

4.4 Deep Learning Contest: Traffic Prediction

4.4.1 Summary

Running PaddlePaddle on our local cluster proved rather difficult, so we will put some emphasis on the meassures we had to take in order to get it to run properly. Afterwards, we will give a brief summary of our understanding of the framework and the steps taken to improve our results.

4.4.2 Detailed Process

Setup
The PaddlePaddle webiste poses the options of either installing the .deb package or the docker image[4]. Our local cluster is a CentOS system, ruling out the .deb package, as installing .deb packages via alien doesn't seem to be a very good idea. It would have to install a whole lot of dependencies, leaving us with half a debian install inside of CentOS. As on most clusters, docker images aren't allowed. A developer states that CentOS is not supported at this point[5]. As a result, we decided to compile PaddlePaddle ourselves.

Compilation
Our cluster encourages users to compile their own software, therefore a huge selection of compilers and tools is provided.

We decided to compile with both GPU and CPU support as we were unable to predict whether GPUs or CPUs would be the more viable option at this point. The website[6] provides a general overview how to compile PaddlePaddle, but we soon decided to use ccmake, a ncurses frontend for cmake instead.[7]
It would fetch a lot of libraries and tell us if anything is missing, which we would then download, compile and set up. We had to do all of this in our own python virtualenv[8] in order to be able to set up our own set of python software as required by PaddlePaddle. Finally, we had to download and set up the NVIDIA cuDNN library[9].

This would leave us with Makefiles which would look for the to-be included system libraries like libc in the directory `/var/lib` - which doesn't exist under CentOS, as CentOS stores its

[4] `http://www.paddlepaddle.org/doc/build/`
[5] `https://github.com/PaddlePaddle/Paddle/issues/179`
[6] `http://www.paddlepaddle.org/doc/build/build_from_source.htm`
[7] `https://cmake.org/cmake/help/latest/manual/ccmake.1.html`
[8] `http://docs.python-guide.org/en/latest/dev/virtualenvs/`
[9] `https://developer.nvidia.com/cudnn`

libraries at `/lib` or `/lib64` respectively. It is kind of weird that such a widely-used package would 'hardcode' these paths into its cmake files, but as we did not want to dive that deep into its cmake system we corrected all these paths using the well-known find command in combination with sed:

```
find ./ -type f -exec sed -i 's/string1/string2/g' {} \;
```

Building PaddlePaddle proved successfull at this point. We can now use it after loading the python module, adding the binary to the `$PATH` environment variable and the cuDNN library to the `$LD_LIBRARY_PATH`.

Overview
PaddlePaddle is a deep learning platform, designed by Baidu scientists. The code is available at GitHub[10]. Deep learning – generally known as neural networks – is a branch of machine learning. It can handle various tasks from medicine[11], go[12] to Tetris[13]. It attempts to replace handcrafted features with unsupervised feature learning[14].

The task
Our task was to use the traffic prediction demo[15] and improve the provided net definition to a point where the trained network is able to reliably predict the utilisation at various nodes of a road network at given times.

The demo is split up into multiple files:

- `train.sh` – runs paddle with the network definition given in trainerconfig.py, allowing for setting up the number of passes, gpu usage etc
- `dataprovider.py` – allows to specify the inputs
- `trainerconfig.py` – the network definition. This is where the main part of our work had to be done
- `predict.sh` – predicts on new data after the network has been trained on labeled data
- `/data/graph.csv` – specifies network links
- `/data/speeds.csv` – specifies network utilization at given times. This is the main data we use for training our network
- `/data/train.list` – specifies the data used for training. Contains `speeds.csv` (reason given below)
- `/data/test.list` – same content as train.list

Our approach – avoiding overfitting
The test.list contains the same data as the train.list, so we decided to split the last entries from the data in speeds.csv into another file, solely used for testing. We used the other entries as training data for our network. This allows us to evaluate our network without having access to

[10] `https://github.com/PaddlePaddle/Paddle`
[11] Artificual neural networks in medical diagnosis, Filippo Amato et al, Journal of Applied Biomedicine, 2013
[12] `https://de.wikipedia.org/wiki/AlphaGo`
[13] `http://cs231n.stanford.edu/reports2016/121_Report.pdf`
[14] `https://en.wikipedia.org/wiki/Feature_learning`
[15] `https://github.com/PaddlePaddle/Paddle/tree/develop/demo/traffic_prediction`

the data our network will be evaluated on by the competition while helping to avoid overfitting[16]. We then used Matlab to evaluate the current state (see paragraph Evaluation for details). Consequently, we had to train once again on the whole data set after finding a suitable net definition.

Our approach – analyzing the network

The first thing we did was taking a look at the provided network and trying to understand what was happening.

An excerpt from `trainerconfig.py`:

```
##DATA Configuration##

is_predict = get_config_arg('is_predict', bool, False)
trn = './data/train.list' if not is_predict else None
tst = './data/test.list' if not is_predict else './data/pred.list'
process = 'process' if not is_predict else 'process_predict'
define_py_data_sources2(
    train_list=trn, test_list=tst, module="dataprovider", obj=process)

##Parameter Configuaration##

TERM_NUM = 24
FORECASTING_NUM = 24
emb_size = 16
batch_size = 128 if not is_predict else 1
settings(
    batch_size=batch_size,
    learning_rate=1e-3,
    learning_method=RMSPropOptimizer())

##Algorithm Configuration##

output_label = []

link_encode = data_layer(name='link_encode', size=TERM_NUM)
for i in xrange(FORECASTING_NUM):
    #Each task share same weight.
    link_param = ParamAttr(
        name='_link_vec.w', initial_max=1.0, initial_min=-1.0)
    link_vec = fc_layer(input=link_encode, size=emb_size, param_attr=link_param)
    score = fc_layer(input=link_vec, size=4, act=SoftmaxActivation())
    if is_predict:
        maxid = maxid_layer(score)
        output_label.append(maxid)
    else:
        # Multi-task training.
        label = data_layer(name='label_%dmin' % ((i + 1) * 5), size=4)
        cls = classification_cost(
            input=score, name="cost_%dmin" % ((i + 1) * 5), label=label)
        output_label.append(cls)
outputs(output_label)
```

- Data Configuration: defines the input data by defining where to find train.list and test.list and defines the dataprovider
- Parameter Configuration: the main 'ingredients' of the training process are declared here.
  - embsize: number of neurons per layer. Highly dependant on the input data.

[16] `http://www.ma.utexas.edu/users/mks/statmistakes/ovefitting.html`

  - learningrate: magnitude of change the network can undergo during one step. This is a critical setting as too small values wont converge in a given time and too big values will jump around the correct curve without converging. We decided to leave it at default as the net converged in what seemed to be a reasonable fashion.
  - learningmethod: defines which optimization method to use. PaddlePaddle supports a lot of them, described at[17]. We found the given RMSPropOptimizer to be the most useful, as the others are either optimized for smaller input data sets or did not yield useful results

- Algorithm Configuration: the network is defined here. One can see that the given network is rather simple:
  - linkencode: the input layer (so-called datalayer). It is TERMNUM wide (=24).
  - linkvec: a so called fully connected layer, it takes linkencode as an input and is embsize (=16) wide.
  - score: another fc layer, it takes linkvec as an input and has the size of 4.

A network this small can hardly be seen as useful for such a complicated task. Hence we thought a long time about which of the layers PaddlePaddle supports[18] shall be added to create a bigger network.

Improving the network
The very first thing we tried - as suggested in the task description - was to add the `graph.csv` to the data by adding it to the train.list. Surprisingly this resulted in a slightly higher RMSE - 18.5 compared to 18.2 when trying to reconstruct the test data we removed from the training set. We have no explanation for this, but suspect that this data might disturb the training process to find the correlations on its own. The fact that it is a rather strong assumption to say that a connection is a causation rather than a correlation for similarities at connected nodes finally convinced us to drop the graph.csv completely. The network should - in theory - be able to find these correlations by itself.

Completely rewriting the network appeared to be a rather difficult task as we are required to be completely compatible with the data structure given in the `speeds.csv`. We therefore decided to add our layers in between the given datalayer 'linkencode' and the given fc layer 'score'. This allows us to have our own network definition without having to worry about possible incompatibilities.

We think this would be a fitting use case for recurrent neural networks.[19]

As one can see in the figure, a recurrent neural network allows the network not only to look at the input data from timestep t, but also at the input data from timestep $t+n$ and $t-n$. This is a huge advantage for our data, as it can find correlations in-between timesteps.

PaddlePaddle provides an example for RNNs[20], so we tried to use it as a starting point for our own implementation. We spent a lot of time here, trying to fit the example into the existing data, fixing stuff here and there, but eventually had to give up at this point:

[17] `http://www.paddlepaddle.org/doc/ui/api/trainer_config_helpers/optimizers.html`
[18] `http://www.paddlepaddle.org/doc/ui/api/trainer_config_helpers/layers.html`
[19] `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`
[20] `http://www.paddlepaddle.org/doc/algorithm/rnn/rnn.html`

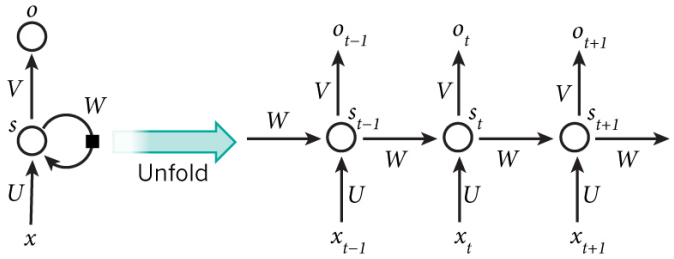

Figure 7: A recurrent neural network and the unfolding in time of the computation involved in its forward computation.

```
./train.sh
I0305 21:00:43.117408 27848 Util.cpp:160] commandline: /home/username/asc/mypaddle
    /bin/../opt/paddle/bin/paddle_trainer --config=trainer_config.py --save_dir=./
    output --trainer_count=2 --log_period=1000 --dot_period=10 --num_passes=10 --
    use_gpu=false --show_parameter_stats_period=3000
[WARNING 2017-03-05 21:00:44,393 networks.py:1444] 'outputs' routine try to
    calculate network's inputs and outputs order. It might not work well. Please
    see follow log carefully.
[INFO 2017-03-05 21:00:44,396 networks.py:1472] The input order is [link_encode,
    label_5min, label_10min, label_15min, label_20min, label_25min, label_30min,
    label_35min, label_40min, label_45min, label_50min, label_55min, label_60min,
    label_65min, label_70min, label_75min, label_80min, label_85min, label_90min,
    label_95min, label_100min, label_105min, label_110min, label_115min,
    label_120min]
[INFO 2017-03-05 21:00:44,396 networks.py:1478] The output order is [cost_5min,
    cost_10min, cost_15min, cost_20min, cost_25min, cost_30min, cost_35min,
    cost_40min, cost_45min, cost_50min, cost_55min, cost_60min, cost_65min,
    cost_70min, cost_75min, cost_80min, cost_85min, cost_90min, cost_95min,
    cost_100min, cost_105min, cost_110min, cost_115min, cost_120min]
I0305 21:00:44.459434 27848 Trainer.cpp:165] trainer mode: Normal
I0305 21:00:45.713069 27848 PyDataProvider2.cpp:243] loading dataprovider
    dataprovider::process
I0305 21:00:45.763485 27848 PyDataProvider2.cpp:243] loading dataprovider
    dataprovider::process
I0305 21:00:45.773516 27848 GradientMachine.cpp:86] Initing parameters..
I0305 21:00:45.784283 27848 GradientMachine.cpp:93] Init parameters done.
F0305 21:01:33.577266 27868 TableProjection.cpp:39] Check failed: in_->ids
*** Check failure stack trace: ***
F0305 21:01:33.577306 27869 TableProjection.cpp:39] Check failed: in_->ids
*** Check failure stack trace: ***
    @          0x930820  google::LogMessage::Fail()
    @          0x930820  google::LogMessage::Fail()
    @          0x93077c  google::LogMessage::SendToLog()
    @          0x93077c  google::LogMessage::SendToLog()
    @          0x930100  google::LogMessage::Flush()
    @          0x930100  google::LogMessage::Flush()
    @          0x9331a7  google::LogMessageFatal::~LogMessageFatal()
    @          0x9331a7  google::LogMessageFatal::~LogMessageFatal()
    @          0x5d94f5  paddle::TableProjection::forward()
    @          0x5d94f5  paddle::TableProjection::forward()
    @          0x61a529  paddle::MixedLayer::forward()
    @          0x61a529  paddle::MixedLayer::forward()
    @          0x6af1b0  paddle::NeuralNetwork::forward()
    @          0x6af1b0  paddle::NeuralNetwork::forward()
    @          0x6b70f7  paddle::TrainerThread::forward()
```

```
    @           0x6b70f7  paddle::TrainerThread::forward()
    @           0x6b940c  paddle::TrainerThread::computeThread()
    @           0x6b940c  paddle::TrainerThread::computeThread()
    @      0x7f02cc683c80  (unknown)
    @      0x7f02cc683c80  (unknown)
    @      0x7f02cd7576ba  start_thread
    @      0x7f02cd7576ba  start_thread
    @      0x7f02cbde982d  clone
    @      0x7f02cbde982d  clone
    @                (nil)  (unknown)
    @                (nil)  (unknown)
```

This is very unfortunate as we - as stated - expected RNNs to do very well, but we had a hard time trying to find information about this both in the source code and the documentation. We append the corresponding folder for reference (called 'RNN'), but did not create the predictions we made using this version.

Our next approach consisted of connecting fully connected layers in between the data- and the score-layer – as can be seen in this excerpt of `trainerconfig.py`:

```
link_encode = data_layer(name='link_encode', size=TERM_NUM)
for i in xrange(FORECASTING_NUM):
    # Each task share same weight.
    link_param = ParamAttr(
        name='_link_vec.w', initial_max=1.0, initial_min=-1.0)

    link_vec = fc_layer(input=link_encode, size=emb_size, param_attr=link_param)

    link_vec2 = fc_layer(input=link_vec, size=emb_size)
    link_vec3 = fc_layer(input=link_vec2, size=emb_size)
    link_vec4 = fc_layer(input=link_vec3, size=emb_size)
    link_vec5 = fc_layer(input=link_vec4, size=emb_size)
    link_vec6 = fc_layer(input=link_vec5, size=emb_size)
    link_vec7 = fc_layer(input=link_vec6, size=emb_size)
    link_vec8 = fc_layer(input=link_vec7, size=emb_size)
    link_vec9 = fc_layer(input=link_vec8, size=emb_size)
    link_vec10 = fc_layer(input=link_vec9, size=emb_size)
    link_vec11 = fc_layer(input=link_vec10, size=emb_size)
    link_vec12 = fc_layer(input=link_vec11, size=emb_size)
    link_vec13 = fc_layer(input=link_vec12, size=emb_size)
    link_vec14 = fc_layer(input=link_vec13, size=emb_size)
    link_vec15 = fc_layer(input=link_vec14, size=emb_size)
    link_vec16 = fc_layer(input=link_vec15, size=emb_size)
    link_vec17 = fc_layer(input=link_vec16, size=emb_size)
    link_vec18 = fc_layer(input=link_vec17, size=emb_size)

        score = fc_layer(input=link_vec18, size=4, act=SoftmaxActivation())
    if is_predict:
        maxid = maxid_layer(score)
        output_label.append(maxid)
    else:
        # Multi-task training.
        label = data_layer(name='label_%dmin' % ((i + 1) * 5), size=4)
        cls = classification_cost(
            input=score, name="cost_%dmin" % ((i + 1) * 5), label=label)
        output_label.append(cls)
outputs(output_label)
```

We connected linkvec2 to the given linkvec, linkvec3 to linkvec2 and so on. Furthermore, a constant layer size proved handy. We actually started with a lot smaller network, but found out during the training that there is only a very small computational cost for adding more layers until we max out the CPU.

This is the model we used for our submission, its configuration can be found in the folder 'submission'.

Training
We used the following parameters to train the network: (excerpt from `train.sh`)

```
set -e

cfg=trainer_config.py
paddle train \
  --config=$cfg \
  --save_dir=./output \
  --trainer_count=12 \
  --log_period=1000 \
  --dot_period=10 \
  --num_passes=50 \
  --use_gpu=false \
  --show_parameter_stats_period=3000 \
  2>&1 | tee 'train.log'
$
```

The batches are being split up into trainer count processes, so '12' seemed reasonable - the system we used features 12 CPU-cores. We set numpasses to 50 as each pass takes about 15 min, resulting in a total runtime of roughly 12 hrs. We decided against using GPUs (reasons stated below).
We actually started with 10 fc layers, but noticed that 10 layers will only create about 550-600 % CPU-load, leaving a lot of ressources unused. Increasing this number to 17 yielded a total CPU-load of just over 1000 %, so an equivalent of 10 cores is used.

Evaluation
We had to calculate the RSME for our testing- and predicted data. We used Matlab to accomplish this task. The code we used is shown below:

```
actualvector=reshape(actualspeeds,329*24,1);
predictedvector=reshape(predictedspeeds,329*24,1);
rms(predictedvector-actualvector)
```

We used Matlabs built-in reshape method to convert the 329×24-sized matrix into a single vector with the length 329×24. The result has been put into the rms method, yielding the RMSE.

GPU or CPU?
We thought a lot about whether to train the network on the CPU or the GPU, but came to a quick conclusion when training the demo network: The CPU of our training system would train the demo network in about 20 min, while two NVIDIA GTX 1080 GPUs would need roughly 2,5 hrs to do the very same task. We could not understand this at first, and so dived a little deeper. The following shows the GPU utilization during training:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.26                 Driver Version: 375.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080     On  | 0000:02:00.0     Off |                  N/A |
| 28%   42C    P2    40W / 180W |    207MiB /  8113MiB |     14%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080     On  | 0000:03:00.0     Off |                  N/A |
```

```
| 30%   45C    P2    40W / 180W |    209MiB /  8113MiB |     14%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     26893      C   ...ddle/bin/../opt/paddle/bin/paddle_trainer   205MiB |
|    1     26893      C   ...ddle/bin/../opt/paddle/bin/paddle_trainer   207MiB |
+-----------------------------------------------------------------------------+
```

As one can see, the GPUs have a very small load (14 %) on them. We suspect that this is related to the problems we came across when combining very few trainers with a very small network - its just not enough to exploit the GPUs computational power, rendering the training memory bound. This should change for very large networks, but even our rather large "final" network proved to be a lot faster on the CPU.

Creating the results.csv

Creating the `results.csv` seemed rather puzzling, as the used training command in the given predict.sh would actually require us to specify the first model[21]. But – as `strace -f` assured us – it would not even access the latest iteration when we specify the first pass. We therefore changed it to specify the latest iteration:

```
1  set -e
2
3  cfg=trainer_config.py
4  # pass choice
5  model="output/pass-00049"
6  paddle train \
7      --config=$cfg \
8      --use_gpu=false \
9      --job=test \
10     --init_model_path=$model \
11     --config_args=is_predict=1 \
12     --predict_output_dir=.
13
14 python gen_result.py > result.csv
15
16 This very result.csv can be found in our .zip.
```

[21] `http://www.paddlepaddle.org/doc/ui/cmd_argument/detail_introduction.html`