HPC @ FHTW

David Meyer, Lars Mehnen

October 9th, 2022

This document briefly introduces the High-Performance Computing Infrastructure at UAS Technikum Wien.

1 Introduction

High-Performance Computing (HPC) means distributing time-consuming computations to several microprocessors so that single steps can be carry out in parallel. These steps can either be independent of each other (e.g., hyperparameter tuning of machine learning algorithms), or interdependent, where the processes running in parallel need to communicate. Applications fitting in the first setting are also called *embarrassingly parallel* since the computation problem can trivially be decomposed in smaller tasks.

A HPC system comprises one or several machines, including one or several processors, each composed of one or several computational cores, possibly hyperthreaded. Several computing machines connected through a network are called a *cluster*. Each PC in the cluster is called *node*. HPC clusters are well-suited for embarrassingly parallel tasks with little communication overhead. Communication-intensive tasks would be slowed down through the network bottleneck.

2 The High Performance Lab Cluster @ FHTW

2.1 Hardware

The **HPL cluster** at FHTW makes use of 19 high-end PCs in a computing room, connected through a 10-GBit-Network. Each PC includes:

- 1 AMD Ryzen 9 3900X 12-Core Processor
- 1 NVIDIA GeForce RTX 2080 Ti GPU
- 32 GB RAM
- 1 80 GB Hard Disk

The PCs are connected to a Network storage extensible up to 32 TB.

In addition to these computing nodes, a master node is available for login and cluster management. This machine is currently a virtual machine with 16 CPUs and 64 GB RAM.

Thus, the cluster comprises 228 CPU cores and 19 GPUs.

A dedicated GPU RIG is in preparation.

2.2 Software

All nodes are running Linux Ubuntu 20. Installed Software includes:

- CUDA libraries for GPU management
- Python 3.8.10 (with Conda)
- R 4.2.1 (stable) as well as an R version linked against optimized BLAS/LINPACK libraries
- SLURM cluster management software

- OpenMPI and mpich libraries for interprocess communication

The management node is available for authorized users via VPN and SSH (`ssh user@ctld-n1.cs.technikum-wien.at`).

Important notes:

- Users should **not** login to the computing nodes directly, in particular not running processes—this would compromise the SLURM system!
- No job should be run on the management node, it is just a simple virtual machine with restricted resources!

3 The SLURM job scheduler

SLURM (Simple Linux Utility for Resource Management) is a wide-spread open-source system for workload management. The main documentation can be found at http://slurm.schedmd.com.

A SLURM system consists of a control daemon running on the management node, as well as slurm daemons running on each of the nodes. The management node provides shell tools for managing (submitting, monitoring, canceling) *jobs* to the cluster. Each job consists of possibly parallel *steps*, and allocates specific resources (e.g., a minimum amount of cores and/or memory, a GPU etc.) The scheduler distributes a specified amount of instances of some program to the allocated resources. If the resources are not available, the job is queued.

Useful commands include:

- `sinfo`: shows the state of the cluster nodes
- `squeue`: shows the current job queue
- `sacct`: lists completed and running jobs
- `sview`: graphical monitoring tool for jobs and nodes
- `salloc`: allocates some resources on the cluster without starting a task
- `srun`: runs a single job interactively
- `sbatch`: runs a job in the background. Also runs job arrays.
- `scancel`: cancels a running job.

The most important commands are `srun` and `sbatch`.

`srun`, if directly executed, runs in interactive mode, that is, the output of the processes started on the nodes are directed to the standard output of the management node. This is typically used for testing.

The following example allocates 20 *cores* on some random nodes and executes 1 instances of the shell command `hostname` on each of these:

```
meyerd@lab-ctld-n1:~$ srun -n 20 hostname
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n7
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
```

```
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
lab-aicl-n6
```

The scheduler chooses nodes 5 and 6, running 10 instances on each.

The following command allocates 4 *nodes* (note `-N` in uppercase) and runs 1 instance on each:

```
meyerd@lab-ctld-n1:~$ srun -N 4 hostname
lab-aicl-n3
lab-aicl-n2
lab-aicl-n1
lab-aicl-n6
```

Both parameters can be combined:

```
meyerd@lab-ctld-n1:~$ srun -n 8 -N 4 hostname
lab-aicl-n3
lab-aicl-n6
lab-aicl-n1
lab-aicl-n2
lab-aicl-n2
lab-aicl-n3
lab-aicl-n6
lab-aicl-n1
```

This distributes 8 *jobs* to 4 *nodes*.

(Note that although the cores are hyperthreaded, the scheduler is configured to run one job per *core*, not per thread, as this is recommended for computing processes. Thus, each job will allocate 2 logical CPUs (2 threads) on a node.)

`srun` alone is of limited use, as the programming instances cannot be parameterized (with, e.g., different input data), and runs in interactive mode. It is mainly used for debugging.

The main command is `sbatch`, capable of running *job arrays*, similar to a `for` loop. This allows to specify a sequence of identifiers for the job. Each job instance will get one of the indices as unique identifer, available through the environment variable `SLURM_ARRAY_TASK_ID`.

3.1 Example using R

The following shell script runs a small R script (for simplicity, specified "inline" as *here document*—alternatively, the R code could be saved in a separate file `foo.R` and the corresponding part of the shell script reduced to `srun Rscript foo.R`):

```
#!/bin/sh
#SBATCH -a 1-100
srun R --slave <<EOF
     ## retrieve job ID from OS
     index = as.integer(Sys.getenv("SLURM_ARRAY_TASK_ID"))

     library(nnet)
     library(e1071)

     ## use index as model hyperparameter
     size = index

     ## do some computation
```

```
    res = tune.nnet(Species ~ ., data = iris, size = size, MaxNWts = 100000)

    ## write results. Use index in file name:
    cat(size, ",", res[["best.performance"]], "\n",
        file = paste0(index, ".res"),
        sep = "")
EOF
```

The R code fits a neural network by setting the number of hidden nodes (`size`) to the index value and performs a 10-fold cross-validation to estimate the training error, eventually saved in a file. This is done in parallel with all parameter values from 1 to 100.

Note the special comment at the beginning of the script, specifying the SLURM parameter:

```
#SBATCH -a 1-100
```

which specifies an index range from 1 to 100. The shell script has to be started using the `sbatch` command:

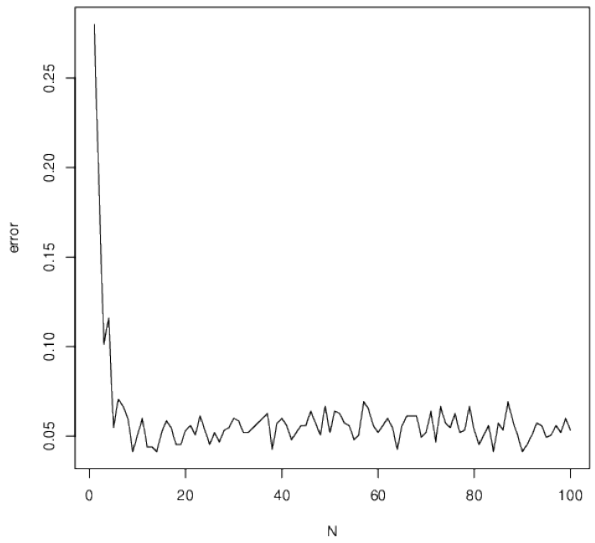
```
meyerd@lab-ctld-n1:~/slurmtest/R$ sbatch slurmtest.sh
Submitted batch job 2162
```

and runs in the background. The job queue, during execution, looks like this:

```
meyerd@lab-ctld-n1:~/slurmtest/R$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            2662_1      base slurmtes   meyerd CG       0:01      1 lab-aicl-n1
...
           2662_18      base slurmtes   meyerd CG       0:02      1 lab-aicl-n2
     2662_[73-100]      base slurmtes   meyerd PD       0:00      1 (Resources)
           2662_52      base slurmtes   meyerd  R       0:01      1 lab-aicl-n6
...
           2662_60      base slurmtes   meyerd  R       0:01      1 lab-aicl-n6
           2662_61      base slurmtes   meyerd  R       0:01      1 lab-aicl-n7
...
           2662_72      base slurmtes   meyerd  R       0:01      1 lab-aicl-n7
           2662_19      base slurmtes   meyerd  R       0:02      1 lab-aicl-n2
...
           2662_24      base slurmtes   meyerd  R       0:02      1 lab-aicl-n2
           2662_25      base slurmtes   meyerd  R       0:02      1 lab-aicl-n3
...
           2662_36      base slurmtes   meyerd  R       0:02      1 lab-aicl-n3
           2662_37      base slurmtes   meyerd  R       0:02      1 lab-aicl-n5
...
           2662_48      base slurmtes   meyerd  R       0:02      1 lab-aicl-n5
           2662_49      base slurmtes   meyerd  R       0:02      1 lab-aicl-n6
           2662_50      base slurmtes   meyerd  R       0:02      1 lab-aicl-n6
           2662_51      base slurmtes   meyerd  R       0:02      1 lab-aicl-n6
```

The job 2662 consists of 100 *tasks* which are distributed to all currently available cluster cores (node 4 was down). Each task results in a separate process, getting its own environment variable set to the corresponding index. It produces one output file with the information written to the standard output (empty here since nothing is printed), and the results file created by the script. These result files can then be collected and aggregated suitably, for example to produce a summary plot:

```
tab = do.call("rbind", lapply(dir(pattern = "*.res"), read.table, sep = ","))
colnames(tab) = c("N", "error")
tab = tab[order(tab$N),]
plot(error ~ N, data = tab, type = "l")
```



If some R packages are using Python behind the scenes (such as Keras/TensorFlow), you will have to activate conda in the script (see below).

3.2 Example using Python

In Python, a similar shell script could look like this:

```
#!/bin/sh
#SBATCH -a 1-100
srun python <<EOF
## retrieve job ID from OS
import os
index = int(os.environ['SLURM_ARRAY_TASK_ID'])

## import libraries
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import numpy as np

## load & prepare data
iris = datasets.load_iris()
X = iris.data
y = iris.target

## do some computation using index as hyperparameter
knn = KNeighborsClassifier(n_neighbors = index)
scores = cross_val_score(estimator = knn, X = X, y = y, groups = y, cv = 10)

## write results using index in file name
f = open(str(index) + ".res", "w")
f.write('k = %d: mean score = %.3f +/- %.3f' % (index, np.mean(scores), np.std(scores)))
f.close()
EOF
```

3.3 Using Conda

Python applications typically require specific versions of packages, and therefore a specific version of Python itself. To escape from the resulting *dependency hell*, virtual environments are commonly used. On the HPL cluster, Conda is available on the cluster nodes (purposely *not* on the controller node!), using a central package cache. Virtual environments are created in the users' home directories (mounted on all nodes), linking to the package cache. Thus, creating environments is `cheap` as files are just links.

Conda environments, exceptionally, must be created and tested on a cluster *node*, since newly installed packages might link to system libraries which might be different on a computing node than on the controller node.

3.3.1 Overview on current environments

The `conda` command is used to manage the conda system. `conda env list` shows the currently available environments:

```
meyer@faramir:~$ ssh meyerd@aicl-n1.cs.technikum-wien.at
meyerd@lab-aicl-n1:~$ conda env list
# conda environments:
#
test                     /home/meyerd/.conda/envs/test
tf                       /home/meyerd/.conda/envs/tf
base                     /opt/conda
```

By default, there is one `base` environment, locally installed (also the corresponding packages). All others, in this example `test` and `tf`, are created in a user's home directory. The base environment comprises Python 3.10.4, pandas 1.4.4, numpy 1.22.3, scipy 1.3.4, scikit-learn 1.1.1. and tensorflow 2.10.0 (making use of GPUs if available).

3.3.2 Activating, using and deactivating an environment

An environment must be activated before it can be used:

```
meyerd@lab-aicl-n1:~$ conda activate base
```

Note that the current environment is shown at the beginning of the prompt. After activation, Python can just be used like any regular installation:

```
(base) meyerd@lab-aicl-n1:~$ python
Python 3.10.4 (main, Mar 31 2022, 08:41:55) [GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

After usage, the current environment should be deactivated:

```
(base) meyerd@lab-aicl-n1:~$ conda deactivate
meyerd@lab-aicl-n1:~$
```

3.3.3 Creating and managing environments

New environments are created using `conda create:`. The following command creates a new environment `myenv`, initialized with Python version 3.7 (older than the one in the base environment):

```
meyerd@lab-aicl-n1:~$ conda create -n myenv python=3.7
```

This will install Python 3.7. along with a few core packages.

After activation, `conda list` shows the currently installed packages:

```
meyerd@lab-aicl-n1:~$ conda activate myenv
(myenv) meyerd@lab-aicl-n1:~$ conda list
# packages in environment at /home/meyerd/.conda/envs/myenv:
#
# Name                    Version                   Build  Channel
_libgcc_mutex             0.1                        main
...
python                    3.7.13               h12debd9_0
...
zlib                      1.2.12               h5eee18b_3
(myenv) meyerd@lab-aicl-n1:~$
```

New packages can be installed using `conda install`, optionally using version qualifiers:

```
meyerd@lab-aicl-n1:~$ conda install numpy pandas=1.0.0
```

(Note that available versions for a package can be retrieved using `conda search pkgname`)

```
(myenv) meyerd@lab-aicl-n1:~$ python
Python 3.7.13 (default, Mar 29 2022, 02:18:16)
[GCC 7.5.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pandas
>>> pandas.__version__
'1.0.0'
```

Packages and environments are removed using `conda remove pkgname` and `conda env remove -n myenv`, respectively.

Note that `pip` can also be used to install packages in the current environment—it might even be necessary at times—but this should be avoided as long as the required packages can be installed using conda. `pip` will not make use of the package cache, and potentially confuse the conda environment!

3.3.4 Using conda in a SLURM job

To make use of a conda environment in a SLURM batch job, conda must be loaded and activated first. The Python example above becomes:

```
#!/bin/sh
#SBATCH -a 1-100

. /opt/conda/etc/profile.d/conda.sh
conda activate base

srun python <<EOF
## retrieve job ID from OS

...

f.close()
EOF
```

(replace `base` with your environment name, if any.). These two lines are also needed for R jobs using Python behind the scenes (Keras/TensorFlow).

4 High-level interfaces in R

4.1 Using a PSOCK cluster

4.1.1 PSOCK clusters

R comes with support for parallel computing in the `parallel` package (part of the base system) which creates a Parallel Socket (PSOCK) Cluster either on a single multi-core machine, or a network of several machines, accessible via remote shell. This essentially consists of R worker processes running on the different cores, listening to the master process via socket communiation, exchanging data through serialization. For remote nodes, one simply specifies a list of host names, accessible, e.g., via SSH—the R function logs into each host, runs an R process, and establishes a socket connection.

The following command starts 12 worker processes on node `lab-aicl-h1`:

```
library(parallel)
cl = makePSOCKcluster(rep("lab-aicl-n1", 12))
```

(Remark: Socket clusters on a single linux machine can more efficiently be created by forking, using `makeForkCluster()`.)

4.1.2 SLURM integration

The `slurmR` package makes this compliant with SLURM, by simply allocating a specified amount of resources using the SLURM `salloc` command, retrieving the names of the corresponding hosts, and passing these to the `makePSOCKcluster` command. In other words, the PSOCK cluster is built within an empty slurm job. When the PSOCK cluster is stopped (or the master R process terminated), the SLURM resources are freed again.

With `slurmR`, the cluster is created using the following command:

```
library(slurmR)
cl = makeSlurmCluster(84)
```

allocating 84 worker process on the cluster (corresponding to the `-n` parameter of `srun`).

Note that due to an internal limitation on the number of sockets, you will not be able to use more than 120 worker processes per process (job).

4.1.3 Parallelizing tasks

As a next step, the workers will need some initializtion, e.g. loading some packages and setting the seed of the random number generator:

```
clusterEvalQ(cl, {
    library(nnet)
    library(e1071)
    set.seed(4711)
})
```

The actual work horse is defined as a function, taking some parameters:

```
f = function(i) tune.nnet(Species ~., data = iris,
                          size = i, NMaxWts = 100000)$best.performance
```

This again trains a neural network using the iris data, the `i` parameter passed to the hidden nodes argument, and returning the result of a 10-fold-cross validation.

To execute the function on the worker processes, several mechanisms are available. The `parallel` package provides parallelized versions of the `*apply` functions in R, used to apply a function to a vector, e.g.:

```
res = parSapply(cl, 1:100, f)
```

applies the `f` function to each element of the index vector created by the `1:100` expression, and returns the error results as a vector of length 100. This corresponds to Map/Reduce approach.

Alternatively, one could use the `foreach` and `doParallel` packages, providing parallelized `for`-like constructs:

```
library(doParallel) ## depends on foreach
registerDoParallel(cl)
foreach(i = 1:100) %dopar% f(i)
```

(Note that the package requires to register the cluster to be used first.)

Finally, the cluster should be shut down to terminate the worker processes and to release the SLURM resources:

```
stopCluster(cl)
```

4.2 Creating an array job with the `slurmR` package

Instead of using a PSOCK cluster on top of SLURM, the `slurmR` package also provides the possibility to create a SLURM job array from within R. The advantage is, that the tasks will run independently from the original R process—the user can log out and retrieve the results later. Also, if some of the tasks fail, they can be repeated and the results collected later.

The first step is to create the worker function—this time, including the initialization:

```
f2 = function(i) {
    library(e1071)
    library(nnet)
    set.seed(4711)
    tune.nnet(Species ~., data = iris,
              size = i, NMaxWts = 100000)$best.performance
}
```

The main step is to create the job array:

```
jobs = Slurm_lapply(1:100, f2, njobs = 84,
                    plan = "wait",
                    job_name = "nnettune",
                    tmp_path = "slurmrjobs")
```

This creates a shell script that is run using the `sbatch` command. It will create a directory (`tmp_path`) to store various files for information exchange. The `plan` parameter controls whether the results are returned immediately (`collect`), or rerieved later on demand (`wait`).

The last step—in this case—is to collect the results. Since we used the lapply-variant, they are returned as a list:

```
res = Slurm_collect(jobs)
unlist(res)
```

4.3 Using the caret package

Finally, some packages also provide direct integration of parallelization. For example, the `caret` package, designed for unified predictive modeling, internally uses the `foreach` package, and thus automatically makes use of parallelization infrastructure if a corresponding cluster is registered first. The following example tunes two parameters of a support vector machine (`cost` and `sigma`) using a search over a specified parameter grid.

First, initialize the SLURM/PSOCK-cluster and load the `caret` package:

```
library(slurmR)
cl = makeSlurmCluster(84)

library(doParallel)
registerDoParallel(cl)

library(caret)
set.seed(4711)
```

Then, create the parameter grid:

```
par = expand.grid(C = 2^c(-10:10), sigma = 2 ^ c(-10:10))
```

Finally, use the generic `train()` function to train and evaluate the models:

```
svmFit <- train(Species ~ ., data= iris,
                method = "svmRadial",
                preProcess = "range",
                trControl = trainControl(seeds = NULL),
                tuneGrid = par)
```

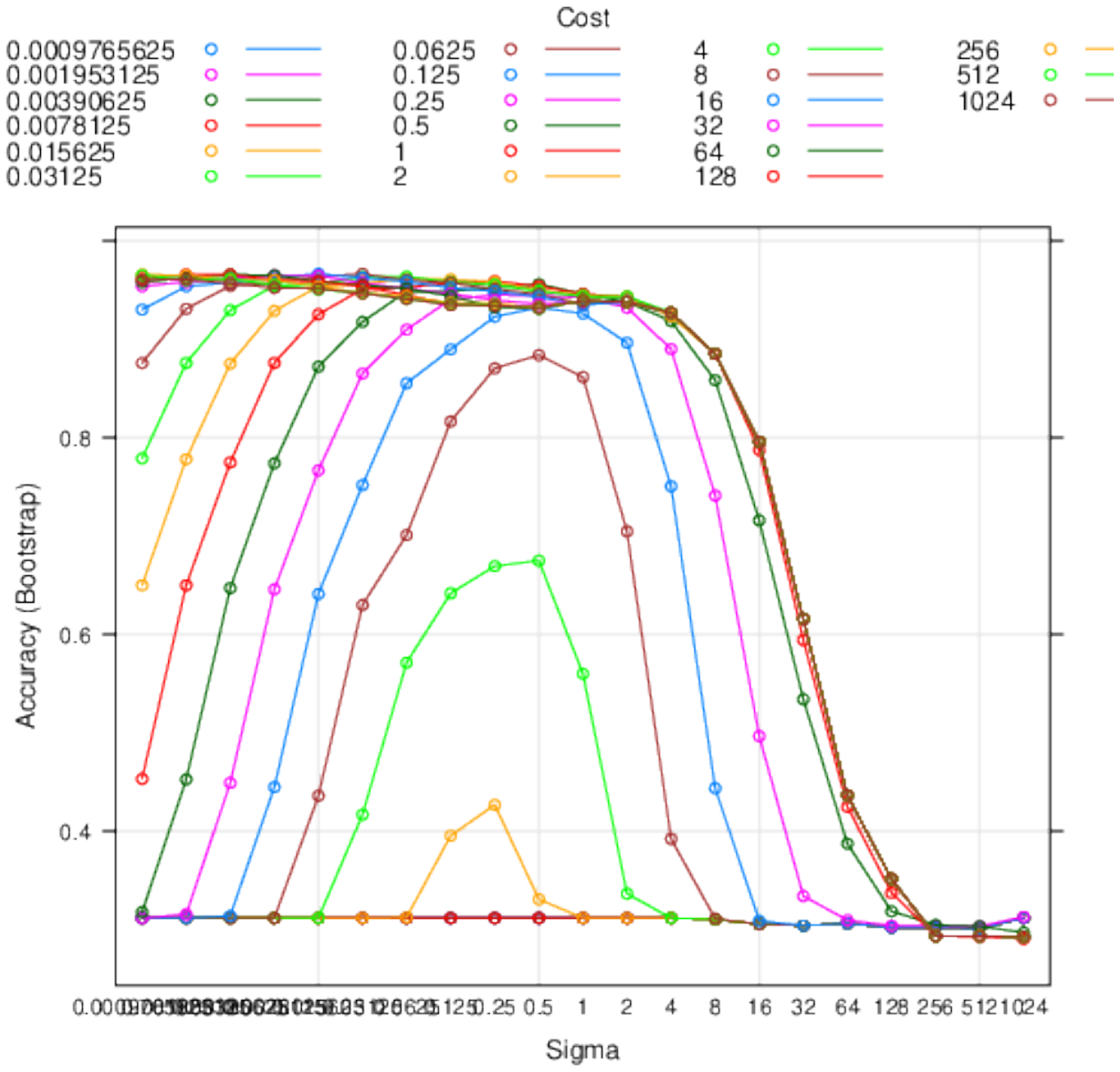
The `train` function accepts a lot of parameters to fine-control the tuning process—here, we just specified a simple preprocessing to scale the data (`preProcess`), and use random seeds for the various worker processes (`seeds = NULL` in the `trainControl` object). By default, the function uses bootstrapping to train and test the different models.

The returned object contains the tuning results and can be inspected:

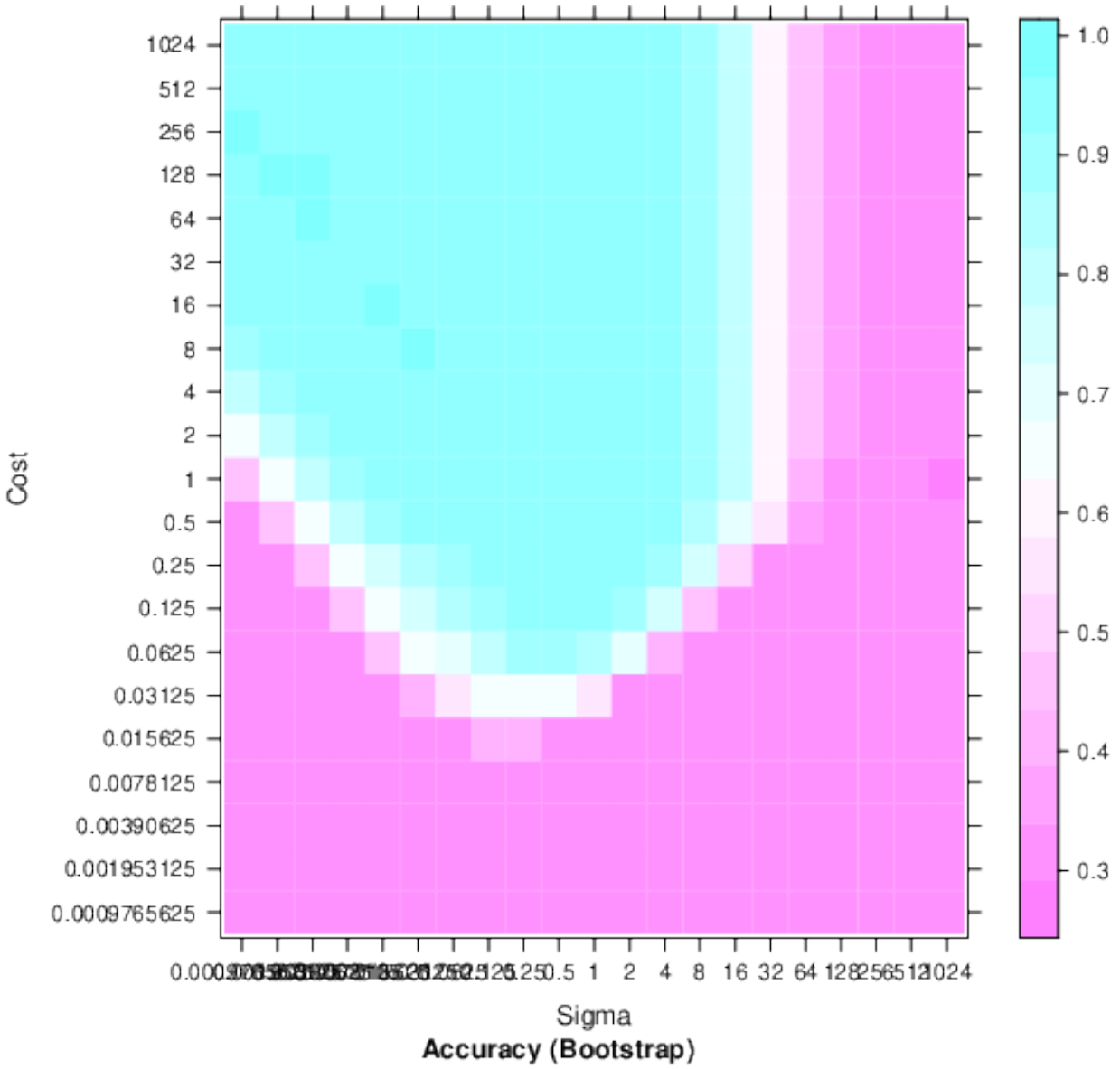
```
> svmFit
Support Vector Machines with Radial Basis Function Kernel

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

Pre-processing: re-scaling to [0, 1] (4)
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
Resampling results across tuning parameters:

  C             sigma         Accuracy   Kappa
  9.765625e-04  9.765625e-04  0.3118531  0.017607143
  9.765625e-04  1.953125e-03  0.3118531  0.017607143
  9.765625e-04  3.906250e-03  0.3118531  0.017607143

...

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.015625 and C = 16.
```

and also graphically analyzed:

```
plot(svmFit, "line")
```




```
plot(svmFit, "level")
```



The best model is retrieved using:

```
svmFit$finalModel
```

Finally, we do not forget to shut down the cluster:

```
stopCluster(cl)
```

5 MPI Hello World

This part, shows a basic MPI hello world application and also discusses how to run an MPI program. The introduction will cover the basics of initializing MPI and running an MPI job across several processes.

5.1 Hello world code examples

Lets dive right into the code from this intro located in mpi_hello_world.c. Below are some excerpts from the code.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

// Initialize the MPI environment
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];

int name_len;

MPI_Get_processor_name(processor_name, &name_len);

// Print off a hello world message
printf("Hello world from processor %s, rank %d out of %d processors\n",
       processor_name, world_rank, world_size);

// Finalize the MPI environment.
MPI_Finalize();
}
```

You will notice that the first step to building an MPI program is including the MPI header files with `#include <mpi.h>`. After this, the MPI environment must be initialized with:

```
MPI_Init(int argc, char*** argv)
```

During `MPI_Init`, all of MPIs global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process. Currently, `MPI_Init` takes two arguments that are not necessary, and the extra parameters are simply left as extra space in case future implementations might need them.

After `MPI_Init`, there are two main functions that are called. These two functions are used in almost every single MPI program that you will write.

```
MPI_Comm_size(MPI_Comm communicator, int* size)
```

`MPI_Comm_size` returns the size of a communicator. In our example, `MPI_COMM_WORLD` (which is constructed for us by MPI) encloses all of the processes in the job, so this call should return the amount of processes that were requested for the job.

```
MPI_Comm_rank(MPI_Comm communicator, int* rank)
```

MPI_Comm_rank returns the rank of a process in a communicator. Each process inside of a communicator is assigned an incremental rank starting from zero. The ranks of the processes are primarily used for identification purposes when sending and receiving messages.

A miscellaneous and less-used function in this program is:

```
MPI_Get_processor_name(char* name, int* name_length)
```

MPI_Get_processor_name obtains the actual name of the processor on which the process is executing. The final call in this program is:

```
MPI_Finalize()
```

MPI_Finalize is used to clean up the MPI environment. No more MPI calls can be made after this one.

5.2 Running the MPI hello world application

Now check out the code and examine the code folder. In it is a makefile.

```
> git clone https://github.com/mpitutorial/mpitutorial
> cd mpitutorial/tutorials/mpi-hello-world/code
> cat makefile

EXECS=mpi_hello_world
MPICC?=mpicc

all: ${EXECS}
mpi_hello_world: mpi_hello_world.c
${MPICC} -o mpi_hello_world mpi_hello_world.c

clean:

rm ${EXECS}
```

My makefile looks for the MPICC environment variable. The mpicc program in your installation is really just a wrapper around gcc, and it makes compiling and linking all of the necessary MPI routines much easier.

```
> export MPICC=/usr/bin/mpicc.mpich
> make
> /usr/bin/mpicc.mpich -o mpi_hello_world mpi_hello_world.c
```

After your program is compiled, it is ready to be executed. Now comes the part where you might have to do some additional configuration. If you are running MPI programs on a cluster of nodes, you will have to set up a host file.

The host file contains names of all of the computers on which your MPI job will execute. For ease of execution, you should be sure that all of these computers have SSH access, and you should also setup an authorized keys file to avoid a password prompt for SSH. My host file looks like this.

```
> cat machinefile

lab-aicl-n1     # Run 1 processes on host1
lab-aicl-n2     # Run 1 processes on host2
lab-aicl-n3     # Run 1 processes on host3
lab-aicl-n4     # Run 1 processes on host4
```

For the run script that is provided in the download, you should set an environment variable called `MPI_HOSTS` and have it point to your hosts file. The provided script will automatically include it in the command line when the MPI job is launched. If you do not need a hosts file, simply do not set the environment variable.

Once this is done, you can use the run.py python script that is included in the main repository. It is stored under the `tutorials` directory and can execute any program in all of the tutorials (it also tries to build the executables before they are executed). Try the following from the root mpitutorial folder.

```
> export MPIRUN=/usr/bin/mpirun.mpich
> export MPI_HOSTS=/home/mehnen/machinefile
> cd tutorials

> ./run.py mpi_hello_world

> /usr/bin/mpirun.mpich -n 4 -f /home/mehnen/machinefile ./mpi-hello-world/code/mpi_hello_world

Hello world from processor lab-aicl-n4, rank 3 out of 4 processors
Hello world from processor lab-aicl-n2, rank 1 out of 4 processors
Hello world from processor lab-aicl-n3, rank 2 out of 4 processors
Hello world from processor lab-aicl-n1, rank 0 out of 4 processors
```

As expected, the MPI program was launched across all of the hosts in my host file. Each process was assigned a unique rank, which was printed off along with the process name. As one can see from my example output, the output of the processes is in an arbitrary order since there is no synchronization involved before printing.

Notice how the script called mpirun. This is program that the MPI implementation uses to launch the job. Processes are spawned across all the hosts in the host file and the MPI program executes across each process. My script automatically supplies the *-n* flag to set the number of MPI processes to four. Try changing the run script and launching more processes! Dont accidentally crash your system though. :-)

Now you might be asking, *My hosts are actually 12-core machines. How can I get MPI to spawn processes across the individual cores first before individual machines?* The solution is pretty simple. Just modify your hosts file and place a colon and the number of cores per processor after the host name. For example, I specified that each of my hosts has two cores.

```
> cat machinefile

lab-aicl-n1:12     # Run 12 processes on host1
lab-aicl-n2:12     # Run 12 processes on host2
lab-aicl-n3:12     # Run 12 processes on host3
lab-aicl-n4:12     # Run 12 processes on host4
```

When I execute the run.py script (I changed 4 processes to 54) again, *voila!*, the MPI job spawns 54 processes on only 4 of my hosts.

```
./run.py mpi_hello_world

/usr/bin/mpirun.mpich -n 54 -f /home/mehnen/machinefile ./mpi-hello-world/code/mpi_hello_world

Hello world from processor lab-aicl-n4, rank 36 out of 54 processors
...
Hello world from processor lab-aicl-n4, rank 38 out of 54 processors

Hello world from processor lab-aicl-n3, rank 27 out of 54 processors
...
Hello world from processor lab-aicl-n3, rank 25 out of 54 processors
```

```
Hello world from processor lab-aicl-n2, rank 15 out of 54 processors
...
Hello world from processor lab-aicl-n2, rank 21 out of 54 processors

Hello world from processor lab-aicl-n1, rank 4 out of 54 processors
...
Hello world from processor lab-aicl-n1, rank 52 out of 54 processors
```

This was just a basic introduction to MPIch. If you want to run jobs on the Cluster/GRID in a manner, that allows task management, so that you don't create conflicts with other colleagues or job-schedules - we have to use the SLURM process manager The only difference is, that you don't use the Hydra process manager but the SLURM process manager.

When doing so - we don't need the machine file and the according export anymore - SLURM will take care of the currently available nodes.

Executing a job with 24 processes (on probably more than one node) looks like this:

```
srun -n 24 mpi_hello_world

Hello world from processor lab-aicl-n1, rank 6 out of 24 processors
...
Hello world from processor lab-aicl-n1, rank 5 out of 24 processors

Hello world from processor lab-aicl-n2, rank 20 out of 24 processors
...
Hello world from processor lab-aicl-n2, rank 19 out of 24 processors
```