Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

R for High Performance Computing @ LRZ

Ch. Bernau (LRZ-Department for HPC), R Meetup Group, February 22nd 2015

Outline

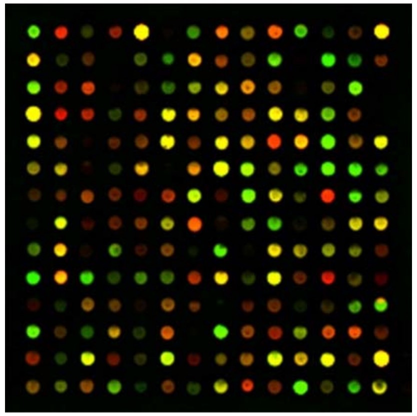
1. Use cases
2. Quick overview: Infrastructure and computing systems at LRZ
3. R-Studio as a first entry point to HPC at LRZ
4. Why parallel programming
5. Common obstacles/nuisances on your way to HPC
6. R-packages for parallel computing
   - Shared memory approaches
   - MPI-based approaches
   - Data base approaches

Optional Tutorial: Opportunity to try some of the approaches above on LRZ’s Rstudio Server

Use cases I: Micro-Array Data

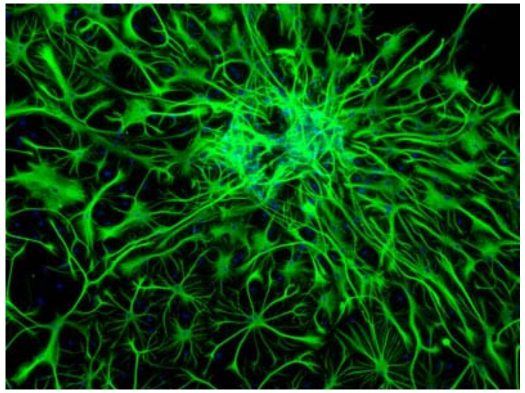
Microarray-Data

- highdimensional noisy data
- number of probes per array: more than 500000
- number of variables after preprocessing (p): between 2000 and 25000
- number of observations (n): between 40 and 300



Interactions of geneexpressions (similar to dependency graphs):

- ca. 20000 gene expression measurements per patient
- 500 preselected variables
- 124750 generalized linear models to be estimated
- tasks are independent

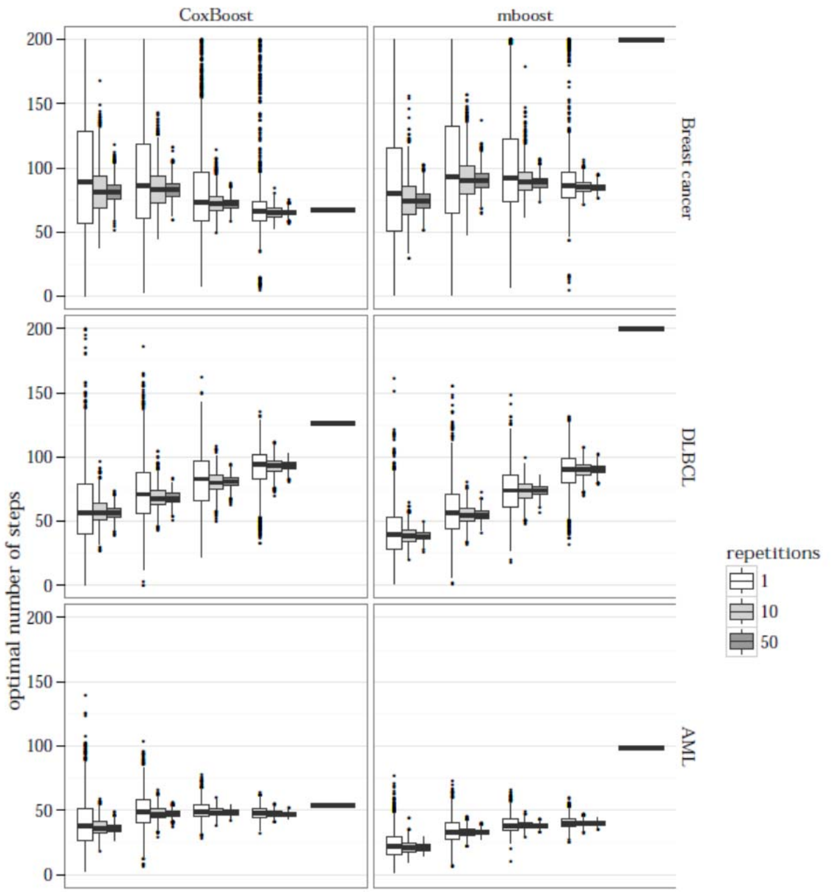


Use cases II: Parameter Studies

Parameter Studies (Seibold et al., 2016):

Assess the effect of the cross-valiation type on the chosen number of boosting steps on the

- R-packages mboost and CoxBoost
- 4 data sets
- Setup for each data set:
    - M times k-fold cross validation for:
        - 4 different numbers of folds (k=3,5,10,LOOCV)
        - 3 different numbers of repetitions (M=1,5,20)
- each setup repeated 2000 times
- 3*4*k*2000 model fits



Use cases III: NGS-Data

BigData and Distributed Memory

De Novo Assembly for Next Generation Sequencing Data

- newest generation of Illumina NGS-Sequencing machine: 500 Billion reads in a single run (between 1-3.5 days)
- translates into roughly 1.2TB of data
- in de no assembly one would optimally like to have all these data in main memory

What is the LRZ ?

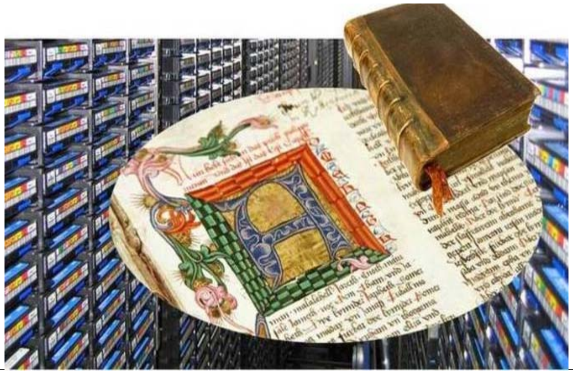
IT Service Provider
for **Munich** Universities

Email, Web,
Multimedia,
IT Security,
HelpDesk,
Virtual Reality,
Trainings, etc.

Regional Computing Centre
for **Bavarian** Universities and
Research institutions

~ 50 PByte Storage/Archive
Digital Archive of the
Bavarian State Library

Munich Scientific Network



German National
Supercomputing Centre

SuperMUC Phase 1+2

3.2+3.2 Pflop/s peak
241,000 compute cores
0.5 PB main memory
15 PB HDD

European Supercomputing Centre

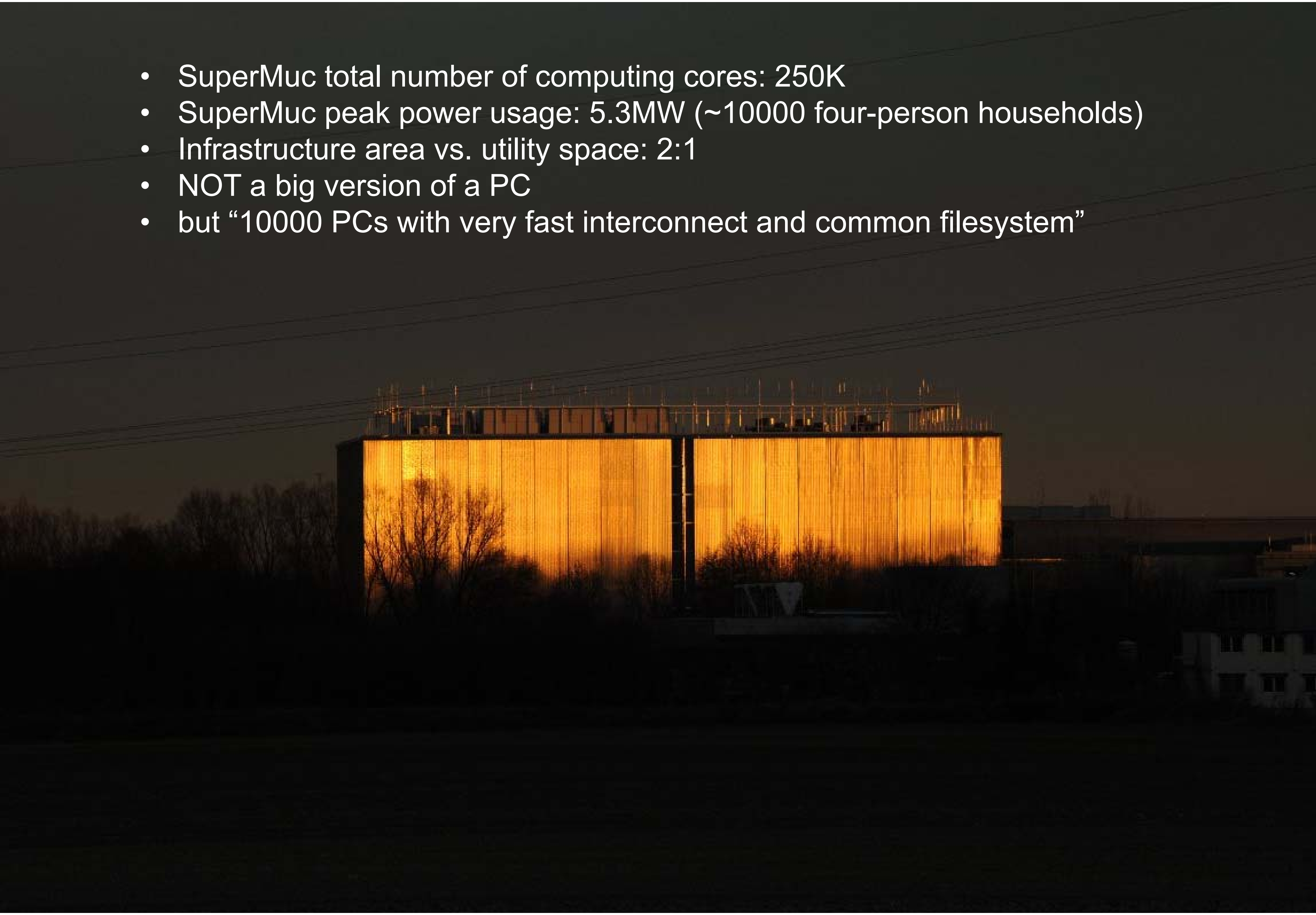
Participating in large European e-Infrastructures

High Performance
Computing

High Speed
Networks

Grid/Cloud
Computing

- SuperMuc total number of computing cores: 250K
- SuperMuc peak power usage: 5.3MW (~10000 four-person households)
- Infrastructure area vs. utility space: 2:1
- NOT a big version of a PC
- but “10000 PCs with very fast interconnect and common filesystem”



Hardware for HPC@LRZ

Linux Cluster

- Massively parallel Cluster (CoolMUC, CoolMUC2 [~10000 cores])
- Big shared memory system (SGI UV [1000 cores, 3TB RAM])
- Serial Cluster
- Remote Visualisation (gvs1-gvs4)

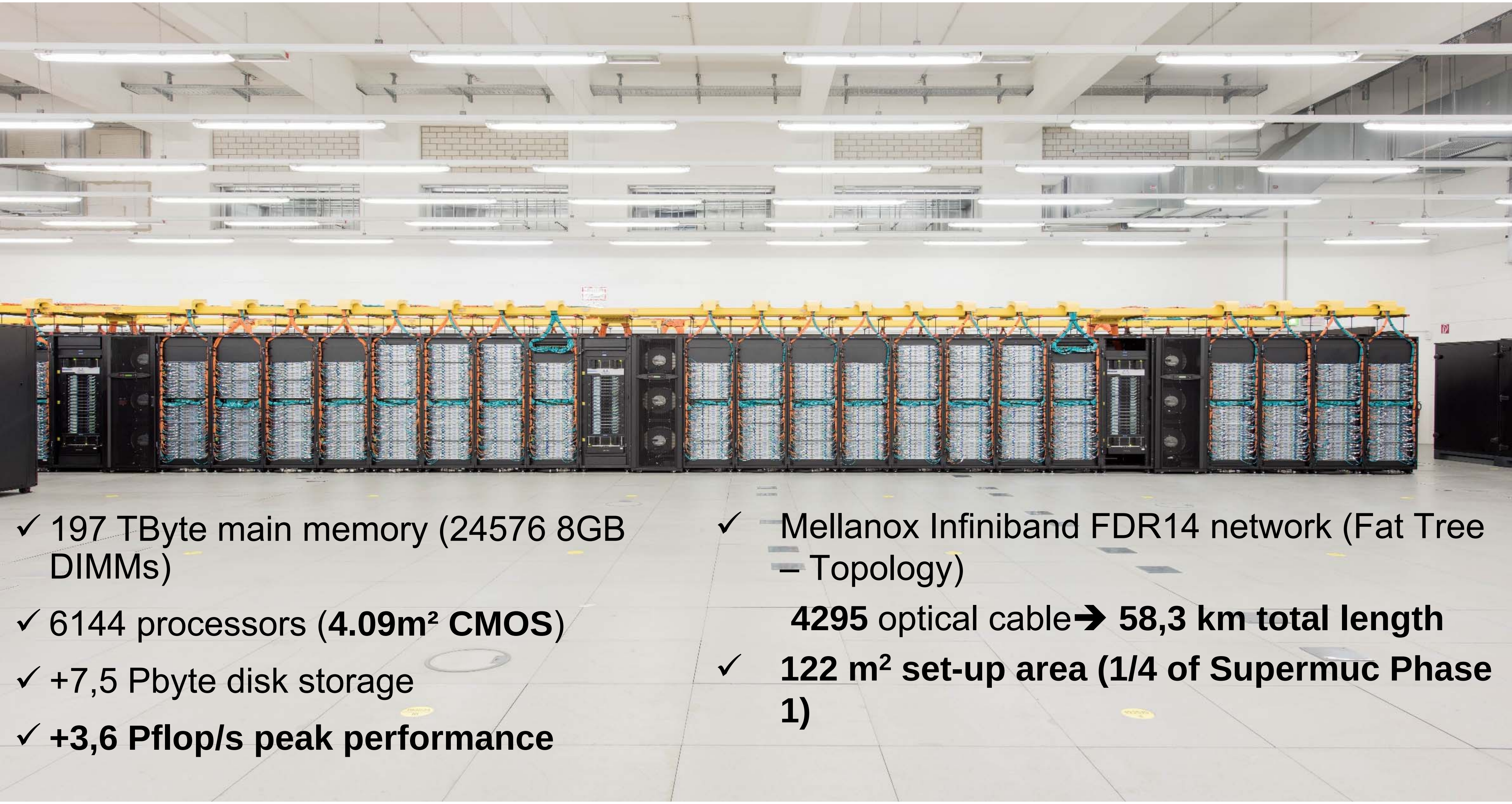
SuperMUC

- Massively parallel Cluster (Phase1, Phase2) [240K cores in total]
- Big shared memory nodes (Fat Nodes, Big Nodes)
- Nodes with accelerators (SuperMIC)
- Remote Visualisation (rvs1-rvs7)

Cloud Systems (Compute and Storage)

- Compute Cloud (openNebula)
- Long running instances (vmware)
- Data Science Storage
- Sync and share

SuperMUC Phase 2



- ✓ 197 TByte main memory (24576 8GB DIMMs)
- ✓ 6144 processors (**4.09m² CMOS**)
- ✓ +7,5 Pbyte disk storage
- ✓ **+3,6 Pflop/s peak performance**

- ✓ Mellanox Infiniband FDR14 network (Fat Tree – Topology)

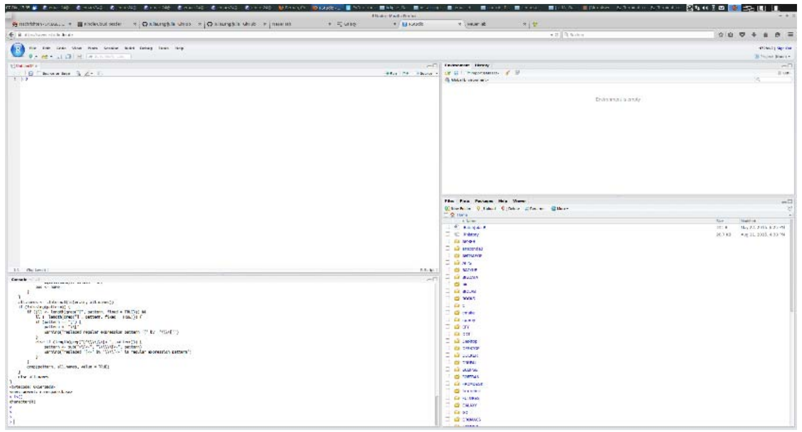
  4295 optical cable➔ **58,3 km total length**
- ✓ **122 m^2 set-up area (1/4 of Supermuc Phase 1)**

Easiest access to R at LRZ: Rstudio (pilot phase)

- Login via www.rstudio.lrz.de
- use LRZ Linuxcluster account or course account

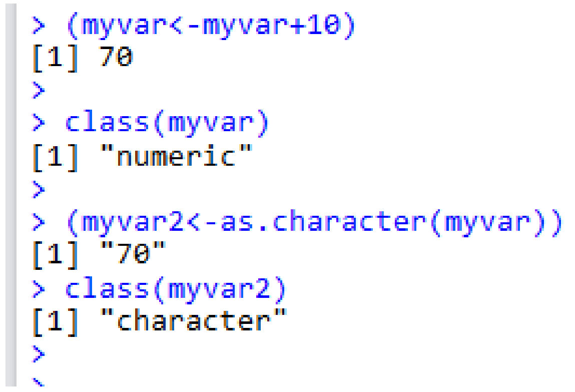
Advantages:

- accessible in a web browser (tablet, smartphone, …)
- nice integrated developer environment
- easy installation of own packages
- easy usage ‘out of the box’
- user sessions are continued after logout
- 20 cores and 256GB RAM
- access to normal LRZ home-folder
- idea: ‚window‘ to LRZ‘s Linuxcluster



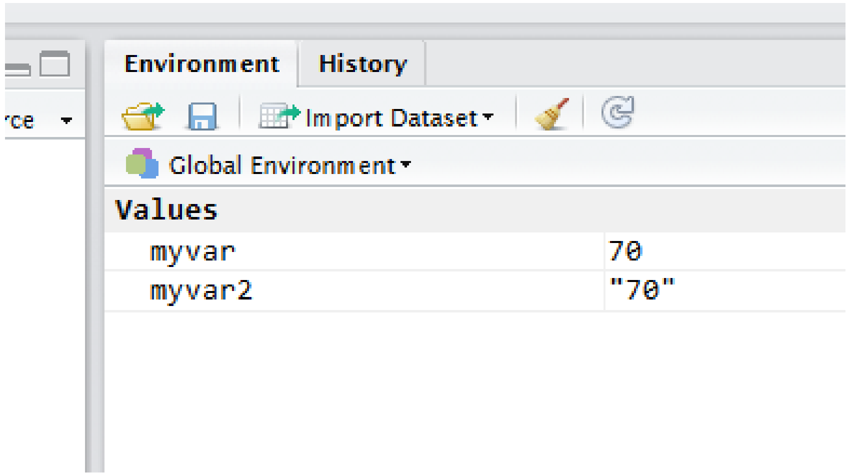
R-Studio

- console in the lower left part



```
> (myvar<-myvar+10)
[1] 70
>
> class(myvar)
[1] "numeric"
>
> (myvar2<-as.character(myvar))
[1] "70"
> class(myvar2)
[1] "character"
>
```

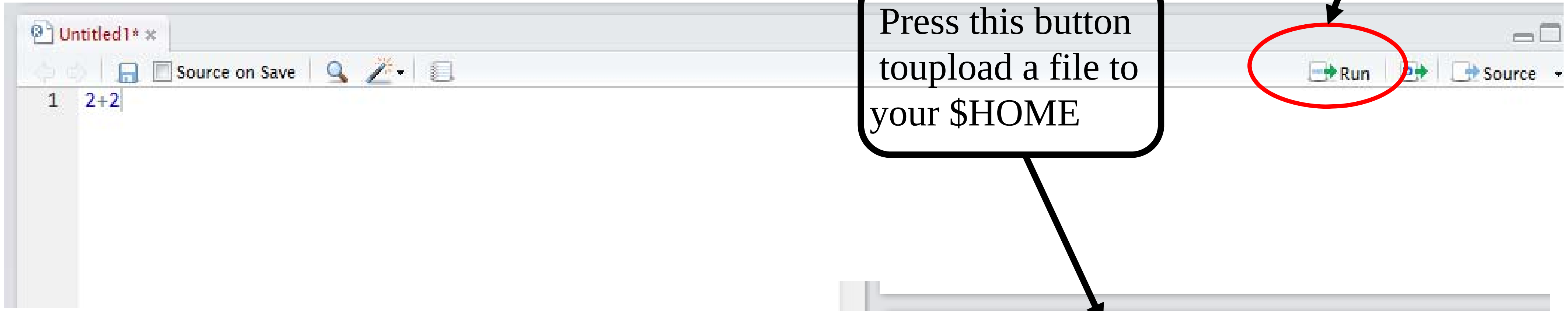
- environment / history in the upper right part
- environment tab shows all defined objects
- history contains all previously run commands (parse through these commands in the console via ‚↑‘)



```
Values
  myvar                          70
  myvar2                         "70"
```

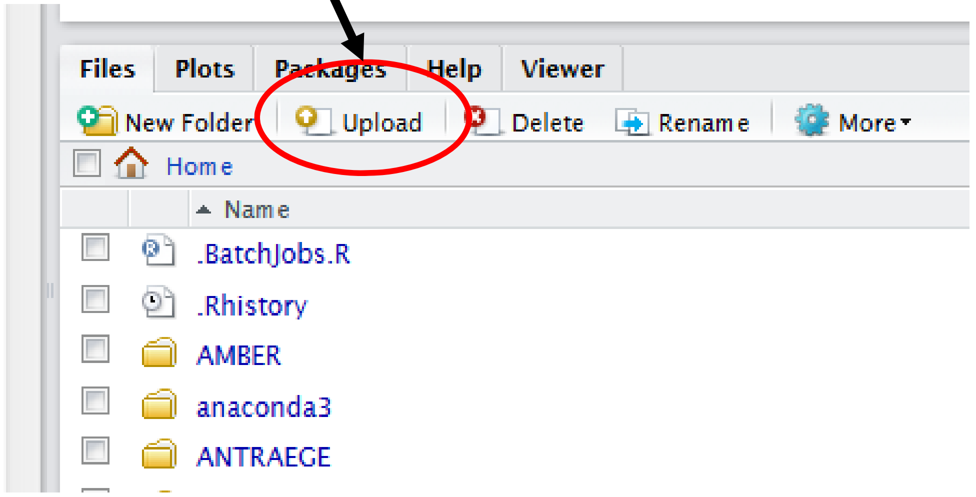
R-Studio

- code editor in the upper left part



```
2+2
```

- homefolder, help and plots in the lower right part



Press this button to run the code in the console

Press this button toupload a file to your $HOME

R on other LRZ-Systems

- R is installed on all LRZ-Systems (Linux-Cluster, Supermuc, hugemem)

- You can run it in the following way:

```
ri72fuv2@login26:~> module load R
ri72fuv2@login26:~> R
```

- You can install your own packages by setting $R_LIBS to one of the directories in your homefolder before you start R:

```
ri72fuv2@login26:~> export R_LIBS=~/Rpacks
```

- You can choose between different modules. Use:

```
ri72fuv2@login26:~> module avail R
```

‘Nuisances’ I: Batch Scheduler

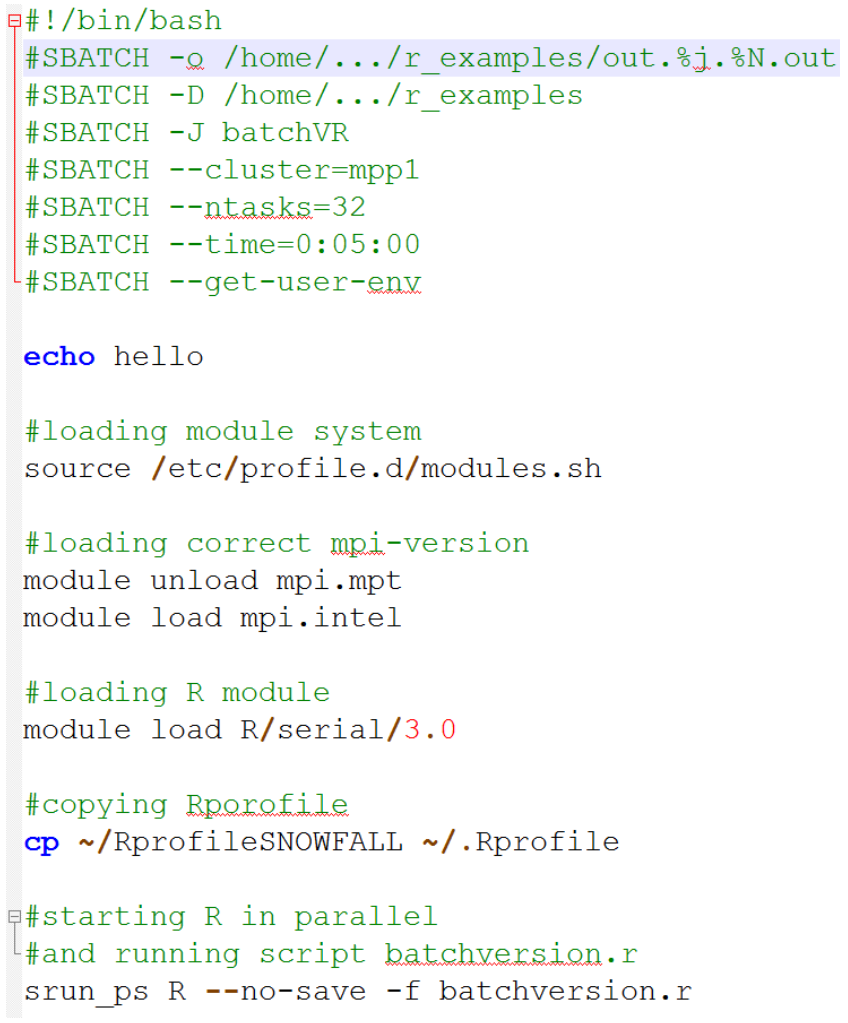
On HPC clusters, users can sually not run interactive R sessions but they have to submit jobs to a batch scheduler (e.g. SLURM or LoadLeveler).

Schedulers are responsible for:

- reserving compute resources:
  - runtime
  - main memory
  - nodes/cores
- distributing and scheduling jobs from all users in an efficient way
- enforcing ‘fair share’ policy
- enforcing job limitations
- cleaning up after jobs

Some important commands (*slurm,LL*):

- job submission: *sbatch, llsubmit*
- job abortion: *scancel, llcancel*
- job/queue status: *squeue, llq*
- current cluster status: *sview, llj*



```
#!/bin/bash
#SBATCH -o /home/.../r_examples/out.%j.%N.out
#SBATCH -D /home/.../r_examples
#SBATCH -J batchVR
#SBATCH --cluster=mpp1
#SBATCH --ntasks=32
#SBATCH --time=0:05:00
#SBATCH --get-user-env

echo hello

#loading module system
source /etc/profile.d/modules.sh

#loading correct mpi-version
module unload mpi.mpt
module load mpi.intel

#loading R module
module load R/serial/3.0

#copying Rporofile
cp ~/RprofileSNOWFALL ~/.Rprofile

#starting R in parallel
#and running script batchversion.r
srun_ps R --no-save -f batchversion.r
```

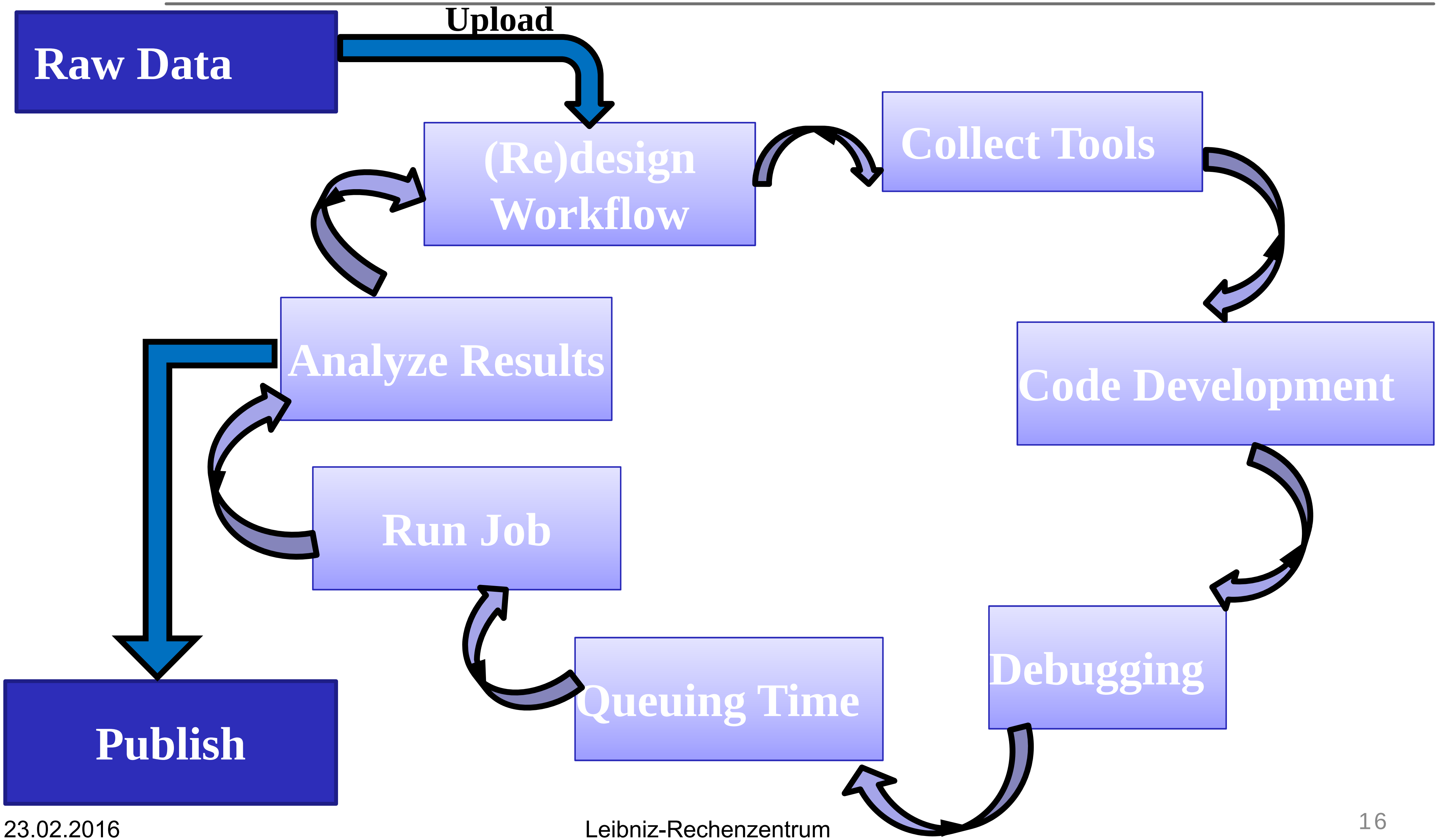
‘Nuisances’ II: Batch Processing

Common user problems with batch processing:

- jobs get aborted if they use more resources than specified
- need to estimate memory and runtime requirements of your computation
  - ➢ estimate memory requirements from a serial run using top or free or from an ‘archived’ job
  - ➢ include some ‘buffer’ for runtime
- queueing times can be high
  - ➢ use *sview* to choose least busy system
  - ➢ smaller, less demanding jobs usually start earlier → benefit from accurate resource estimation

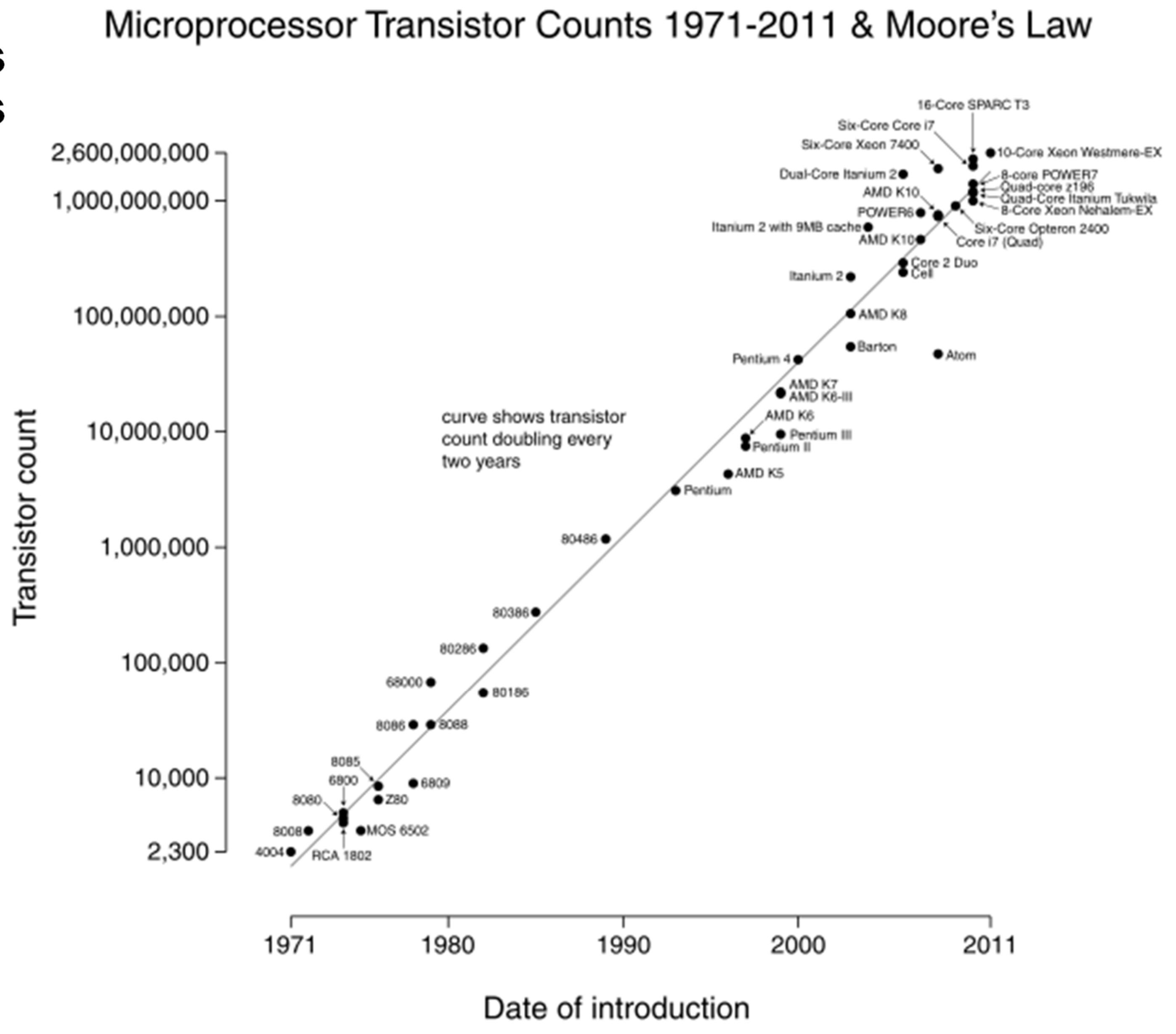
  - ➢ Second talk will provide an easy interface between R and some schedulers which also works at LRZ
- debugging inconvenient
  - time between change in an R-script and feedback much longer than usual
  - compute environment (compute nodes) and test environment (login or interactive nodes) not exactly the same
  - ➢ debug as much as possible in serial, or in small interactive slurm sessions using ‘salloc, poe’
- some connection between scheduler (slurm/load leveler) and startup-command (mpiexec, poe, srun_ps) but need to clearly differentiate between both

Consideration: Data Life Cycle (optional)



Why parallel programming? Moore’s Law

Number of transistors doubles every 2 years



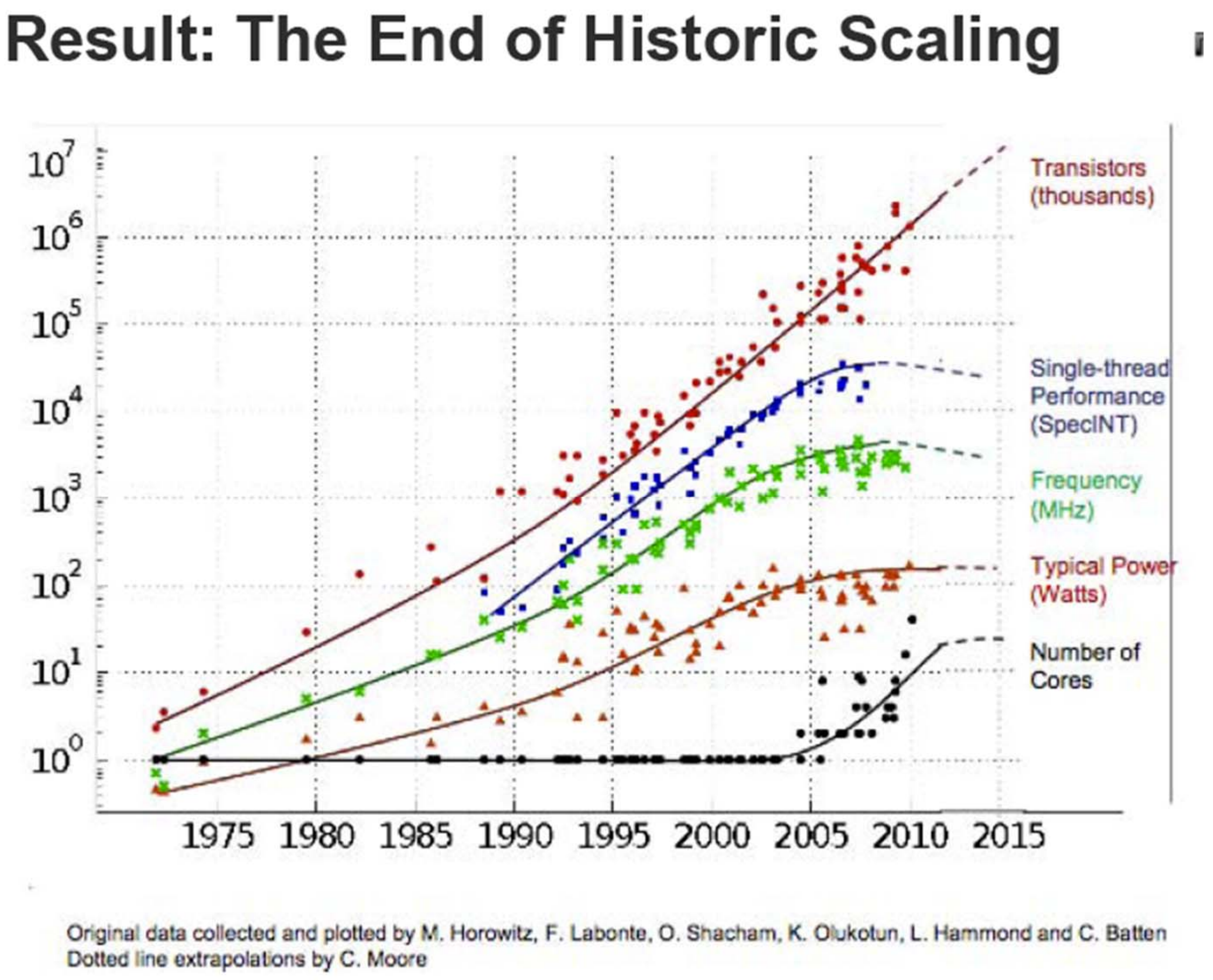
Why parallel programming?

End of the free lunch in 2000 (heat death)

Moore's law means not faster processors, only more of them.

But!
2 x 3 GHz < 6 GHz

(cache consistency, multi-threading, etc)



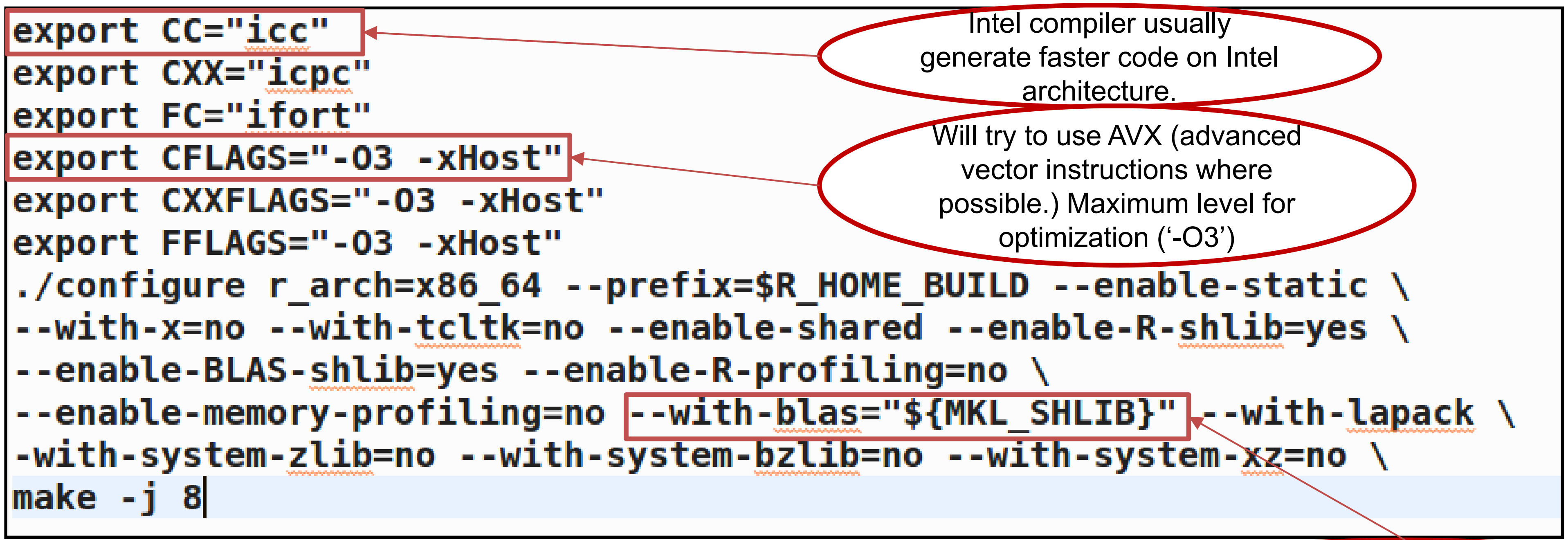
C Moore, *Data Processing in ExaScale-ClassComputer Systems*, Salishan, April 2011

Problem: Moving Data/Latency

Getting data from:		**Getting some food from:**	
CPU register	1ns	fridge	10s
L2 cache	10ns	microwave	100s ~ 2min
memory	80 ns	pizza service	800s ~ 15min
network(IB)	200 ns	city mall	2000s ~ 0.5h
GPU(PCIe)	50.000 ns	mum sends cake	500.000 s~1 week
harddisk	500.000 ns	grown in own garden	5Ms ~ 2months

First steps towards HPC: Compiling R

- Compiling R on your own instead of using precompiled verisons
- Use high performance libraries like MKL (Intel’s Math Kernel Library)



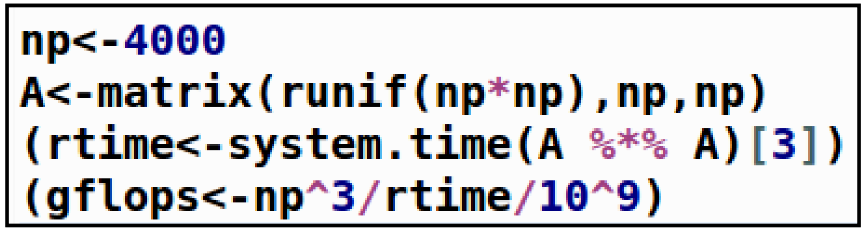
```
export CC="icc"
export CXX="icpc"
export FC="ifort"
export CFLAGS="-O3 -xHost"
export CXXFLAGS="-O3 -xHost"
export FFLAGS="-O3 -xHost"
./configure r_arch=x86_64 --prefix=$R_HOME_BUILD --enable-static \
--with-x=no --with-tcltk=no --enable-shared --enable-R-shlib=yes \
--enable-BLAS-shlib=yes --enable-R-profiling=no \
--enable-memory-profiling=no --with-blas="${MKL_SHLIB}" --with-lapack \
-with-system-zlib=no --with-system-bzlib=no --with-system-xz=no \
make -j 8
```

Intel compiler usually generate faster code on Intel architecture.

Will try to use AVX (advanced vector instructions where possible.) Maximum level for optimization (‘-O3’)

Use MKL for basic linear algebra operations.

- Small benchmark using matrix-matrix multiplication



```
np<-4000
A<-matrix(runif(np*np),np,np)
(rtime<-system.time(A %*% A)[3])
(gflops<-np^3/rtime/10^9)
```

- ➢ Module R/3.2mkl: 8.6sec (7,4GFlops)
- ➢ Module R/3.2: 108.7sec (0.6GFlops)

Shared memory approaches

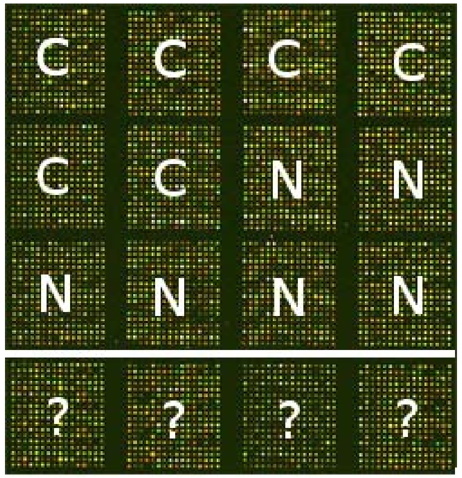
<u>Use all available computing cores of a processor/compute node </u> (by default R will use only one)
Package: *multicore* (now in *parallel*)

- Advantages:
  - shared memory (no need to broadcast R objects to workers)
  - Very easy to implement

- Drawbacks/Pitfalls:
  - No benefit (maybe even performance decrease) for short ‘tasks’
    (might consider task bundling)
  - Load balancing can be problematic
  - R parallel package uses processes not threads, each R process will use at least about 200MB of main memory
  - ‘worker/slave’-mode
- Caution:
  - workers are not removed after parallel section, can create zombies or clutter up your memory

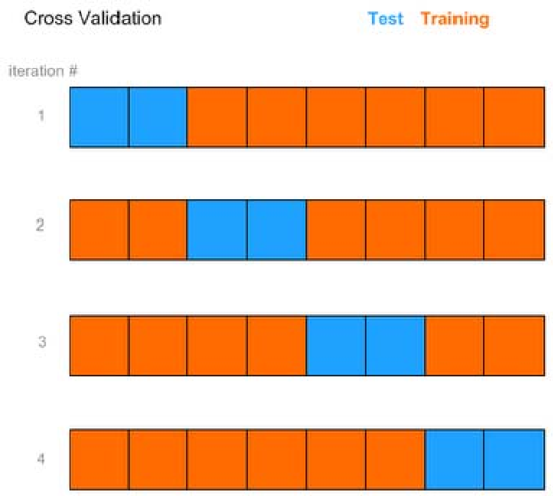
Potential use cases

- Moderate number of independent tasks
- Loops



- Cross Validation
  - Split data set into k folds
    - k-1 folds as training set
    - left out fold as test set
  - Use training set for model fit and calibration of tuning parameters
  - evaluate model on test data

→ each fold is left out once
→ k independent tasks



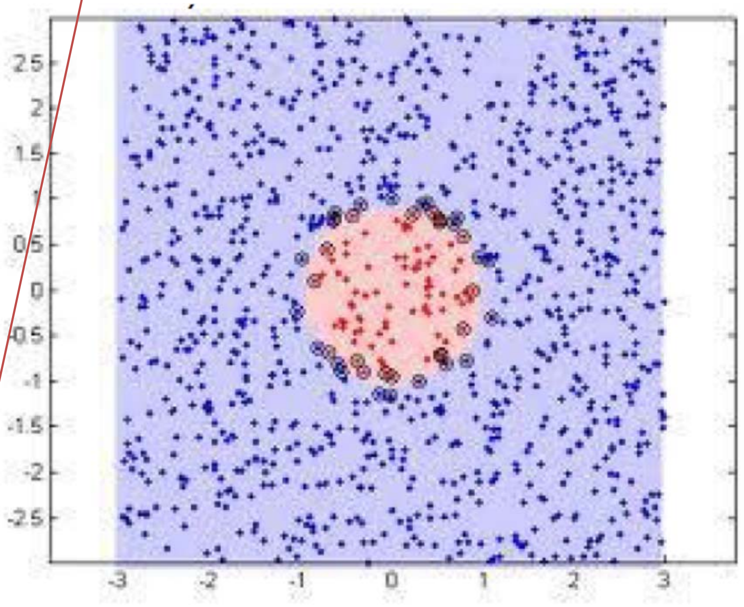
Implementation Example

```
#cross-validation of classification
#on microarray data
library(CMA)
library(parallel)
ncore<-28 #mpp2 Haswell nodes
X<-as.matrix(golub[,-1])
y<-golub[,1]
ls<-GenerateLearningsets(y=y,method='CV',
        fold=10,niter=10000)

#function to be applied on each process
cl2<-function(j){
    ttt<-system.time(cl<-svmCMA(y=y,X=X,
                        learnind=ls@learnmatrix[j,],
                        cost=10))
    list(cl,ttt,Sys.info())
}

mclapply(1:ncore,cl2,mc.cores=ncore)
```

Define a function that shall be run in parallel.



basically like R's normal *lapply, tapply, sapply.*

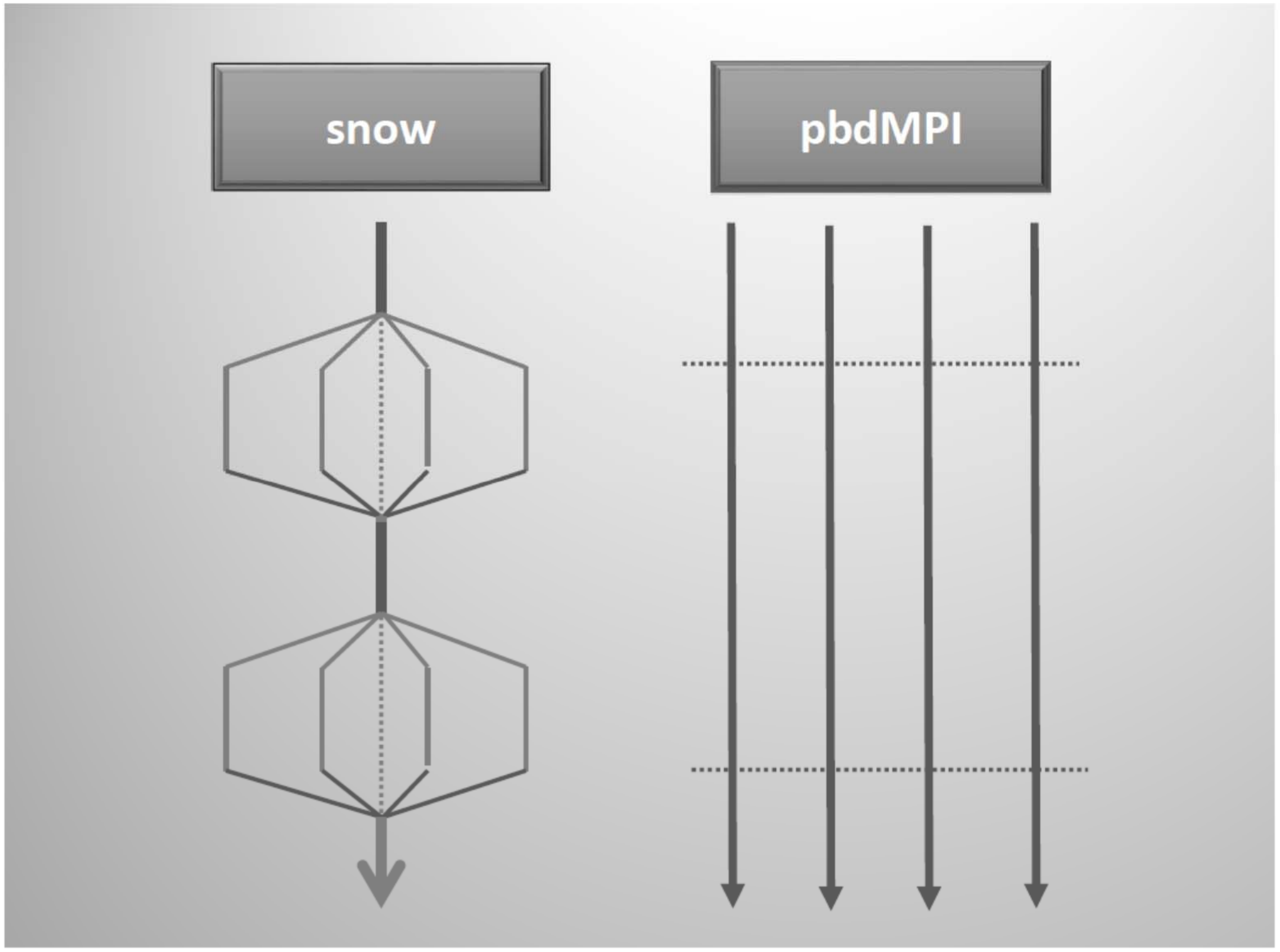
Beyond node boundaries (MPI)

Master/Worker

- RMPI
- Snow
- Snowfall

'Real' MPI

- pbdMPI
- pbdMAT



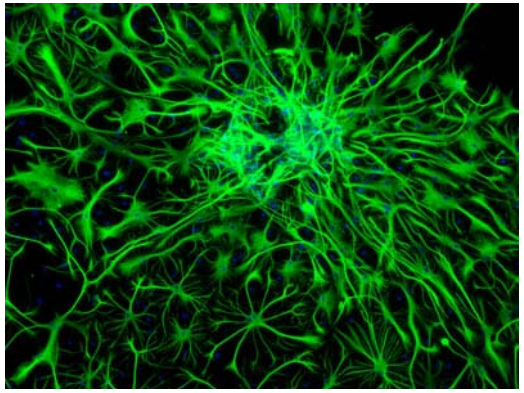
Potential use cases

- vast number of independent tasks
- loops

<u>Example:</u>

Interactions of gene expressions (similar to dependency graphs):

- ca. 20000 gene expression measurements per patient
- 500 preselected variables
- 124750 generalized linear models to be estimated
- tasks are independent



Implementation Example (snowfall)

```r
sfInit(configFile="/home/ubuntu/.sfc/sf_config.cfg")
#preselection of variables (using CMA,X=gene expression matrix, y=response)
gs<-GeneSelection(X=X,y=y,method='t.test')
Ximp<-X[,gs@rankings[[1]][1:nv]]

todo<-c()
for(ii in 2:ncol(Ximp))
    for(cc in 1:(ii-1))
        todo<-rbind(todo,c(ii,cc))

#function to be applied in parallel
interactionpar<-function(k){
    datc<-data.frame(y=y,reg1=Ximp[,todo[k,1]],reg2=Ximp[,todo[k,2]])
    mod<-glm(y~reg1*reg2,family='binomial',data=datc)
    return(coef(summary(mod)))
}

#broadcasting necessary objects
sfExport(list=c('Ximp','y','todo'))
#parallel execution
res<-sfLapply(1:nrow(todo),interactionpar)
sfStop()
```

Omit this on LRZ Linuxcluster: use special .Rprofile.

We use this grid to obtain a function with a single specific argument.

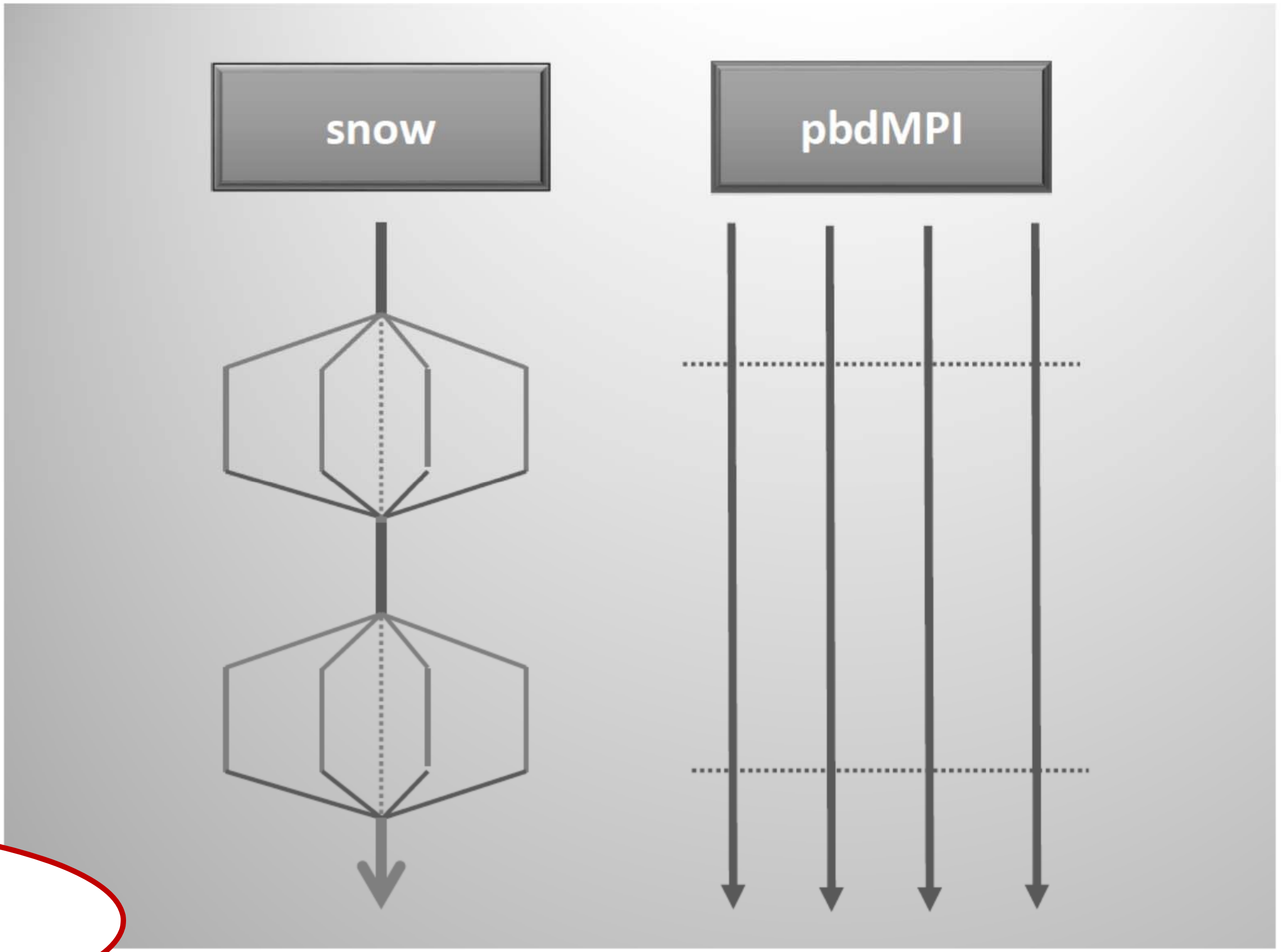
Worker environments are empty. We need to broadcast objects that are not passed directly to the function *interactionpar*.

‘Real MPI’: pbdMPI

Master/Worker
- RMPI
- Snow
- Snowfall

‘Real’ MPI
- pbdMPI
- pbdMAT

Comparably difficult to implement, but it can help in setups with extreme RAM requirements (distributed memory).



Implementation Example (pbdMPI)

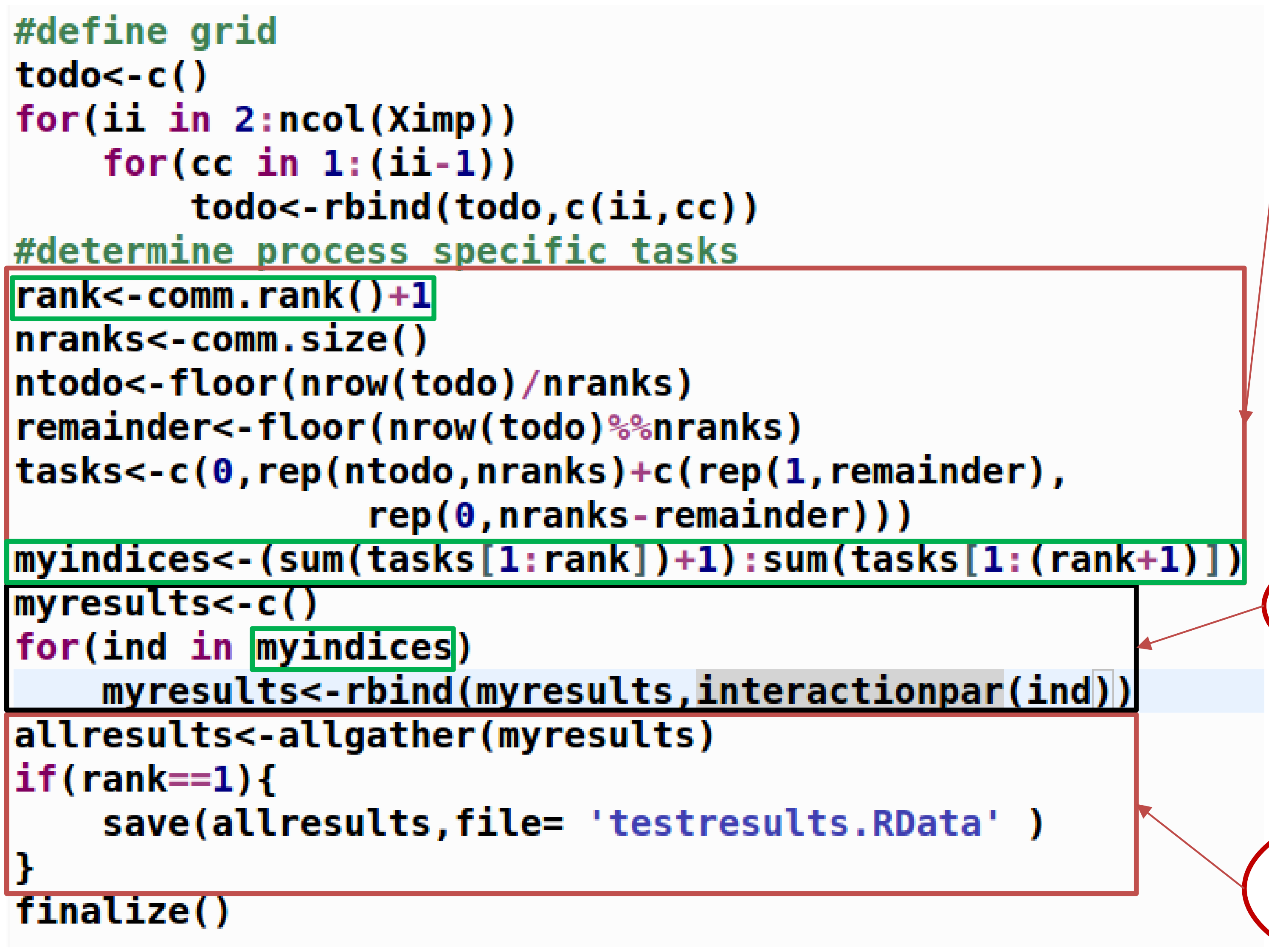
```
library(pbdMPI)
init(set.seed=FALSE)

#prepare gene expressions and response
library(CMA)
data(golub)
X<-as.matrix(golub[,-1])
y<-golub[,1]
#number of genes to be preselected
nv<-40
#preselection of variables
#(using CMA,X=matrix of gene
#expressions, y=response)
gs<-GeneSelection(X=X,y=y,method= 't.test' )
Ximp<-X[,gs@rankings[[1]][1:nv]]
#function to be run in parallel
interactionpar<-function(k){
    datc<-data.frame(y=y,reg1=Ximp[,todo[k,1]],
            reg2=Ximp[,todo[k,2]])
    mod<-glm(y~reg1*reg2,
            family= 'binomial' ,data=datc)
    return(coef(summary(mod)))
}
```

Start with mpiexec –n N Rscript
No need for special .Rprofile.

This code is run on all mpi-processes, thus there is no need to broadcast any objects.

Implementation Example (pbdMPI)



```
#define grid
todo<-c()
for(ii in 2:ncol(Ximp))
     for(cc in 1:(ii-1))
          todo<-rbind(todo,c(ii,cc))
#determine process specific tasks
rank<-comm.rank()+1
nranks<-comm.size()
ntodo<-floor(nrow(todo)/nranks)
remainder<-floor(nrow(todo)%%nranks)
tasks<-c(0,rep(ntodo,nranks)+c(rep(1,remainder),
                    rep(0,nranks-remainder)))
myindices<-(sum(tasks[1:rank])+1):sum(tasks[1:(rank+1)])
myresults<-c()
for(ind in myindices)
     myresults<-rbind(myresults,interactionpar(ind))
allresults<-allgather(myresults)
if(rank==1){
     save(allresults,file= 'testresults.RData' )
}
finalize()
```

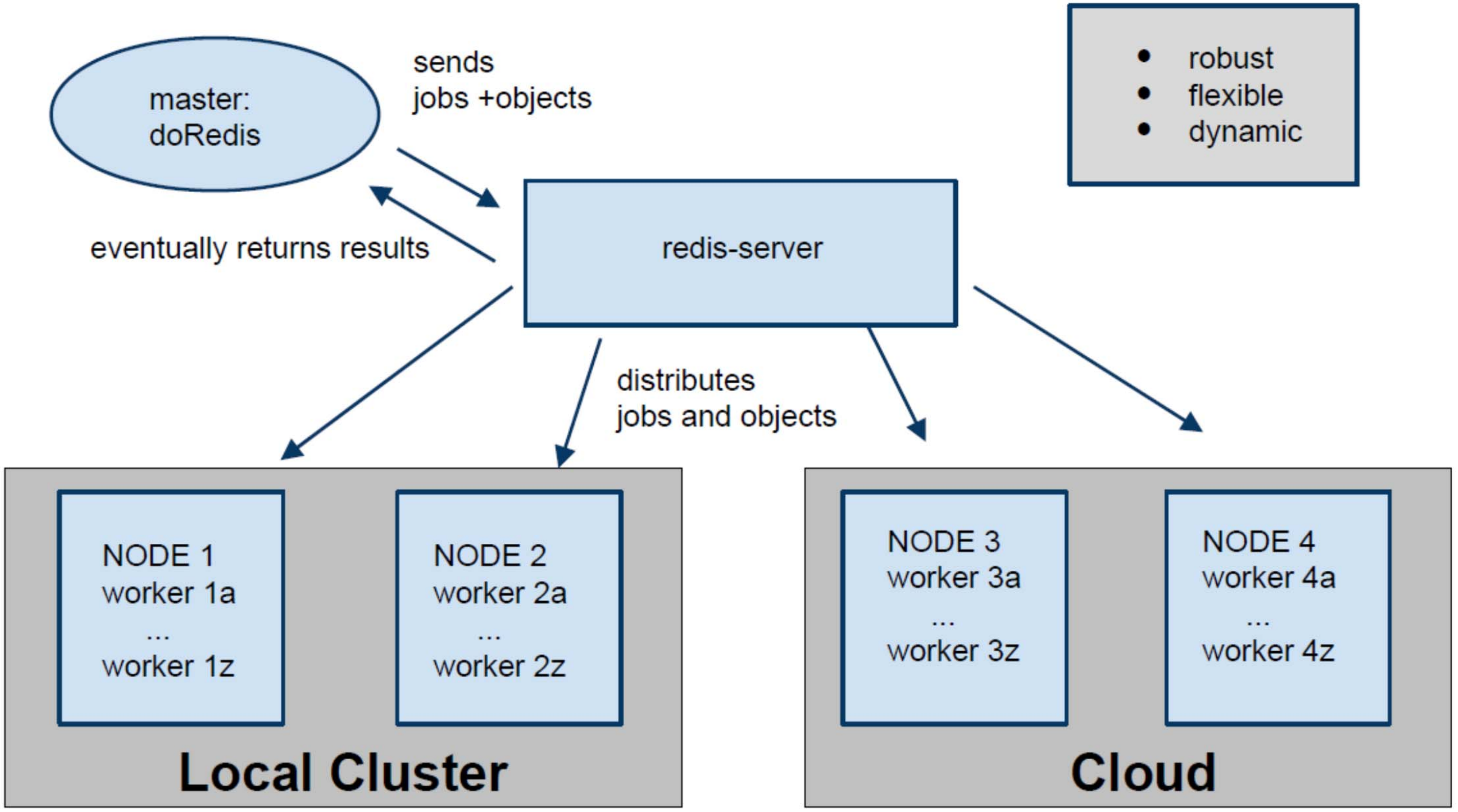
Here, each process determines the tasks it has to do using its MPI RANK and the total number of mpi tasks.

The part that does the actual parallel work looks the same as in a serial implementation.

All processes send their results to all other processes. However, a single process writes the results to disk.

Dynamic Worker Pools / Cloud

Worker processes can run on any R-compatible hardware and can connect at any time



Worker Queues with doRedis

doRedis: essential functions

Overloaded do-operator, c.f. *doSNOW, doMC.*

Master process:

- *registerDoRedis(jobqueue,host)*: connects to the redis-server at 'host' and specifies a jobqueue for the tasks to come
- *foreach(j=1:n)* *%dopar%* *{FUN(j)}*: sends subtasks to redis data base
- *redisFlushAll()*: clears the data base
- *removeQueue()*: removes a queue from the data base

Worker process:

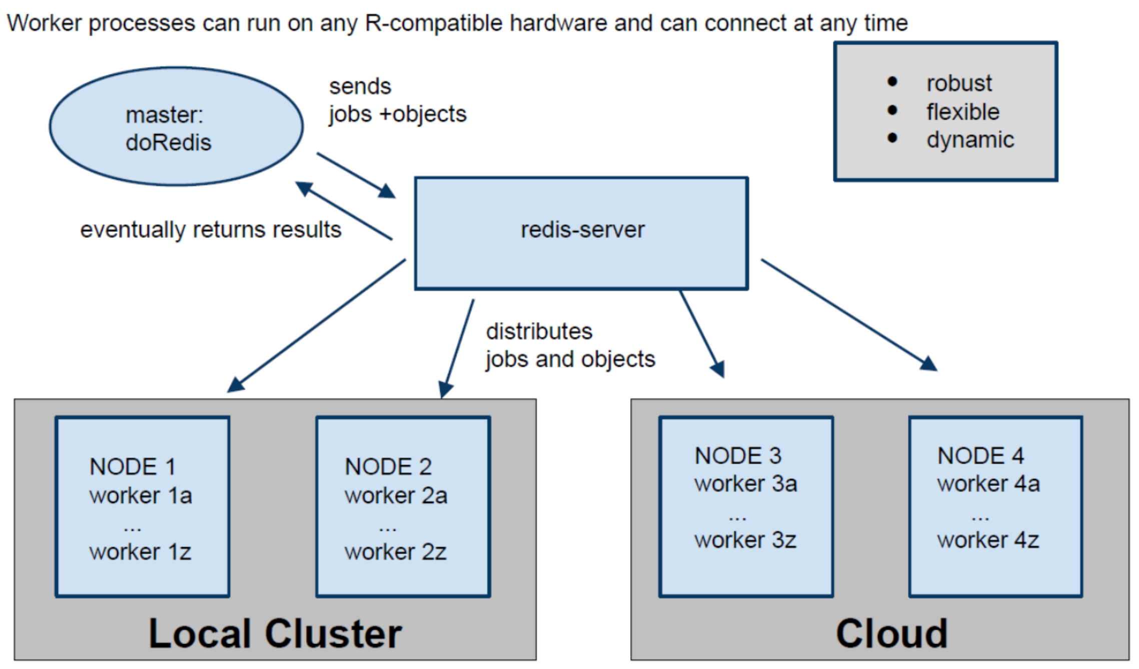
- *registerDoRedis(jobqueue,host)*: registers a jobqueue whose tasks shall be precessed
- *startLocalWorkers(n,jobqueue,host)*: starts n local worker processes the tasks in *jobqueue* (uses multicore)
- *redisWorker(jobqueue,host)*: useful in mpi-environments
- usually users do not request or set the data base values directly

Dynamic Worker Pools / Cloud

Hybridcloud

Combination of cluster and cloud resources:

- some virtual machines as a permanent basis
- adding e.g. employee PCs as additional computing resource if those are currently idle
- cloud bursting (add virtual achines in a cloud e.g. EC2)
- cluster bursting (add machines from LRZ Linux Cluster)


Worker processes can run on any R-compatible hardware and can connect at any time
master: doRedis
sends jobs +objects
eventually returns results
redis-server
robust
flexible
dynamic
distributes jobs and objects
NODE 1 worker 1a ... worker 1z
NODE 2 worker 2a ... worker 2z
NODE 3 worker 3a ... worker 3z
NODE 4 worker 4a ... worker 4z
Local Cluster
Cloud

Implementation Example

Tasks as well as necessary objects are stored and managed by a data base

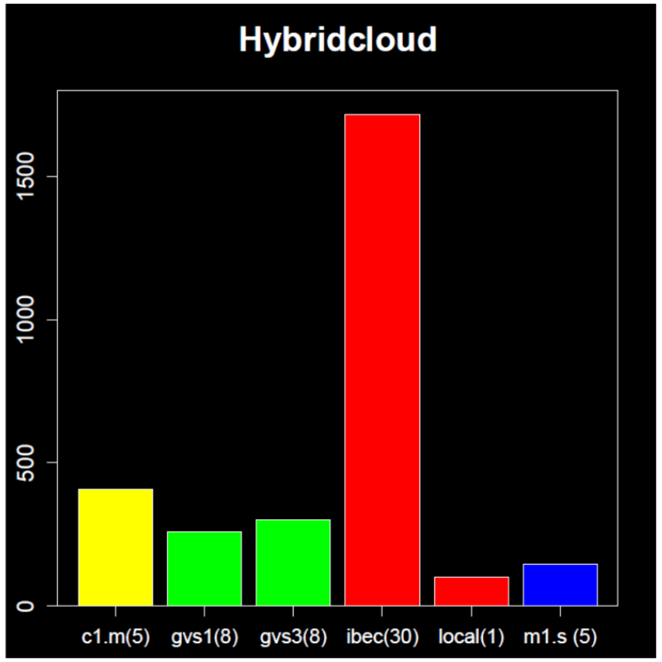
- Data base management by redis-server
- Communication between master and worker nodes via doRedis

Sending Jobs to redis-server:

```
> library(doRedis)
> registerDoRedis('cv_3000')
> results<-foreach(j=1:3000) %dopar% {cviteration(j)}
```

Dynamical adding of additional resource

```
> registerDoRedis('cv_3000',host='localpc@domain.de')
> startLocalWorkers(n=1, queue='cv_3000',iter=5)
```



→Separation of task management and task execution

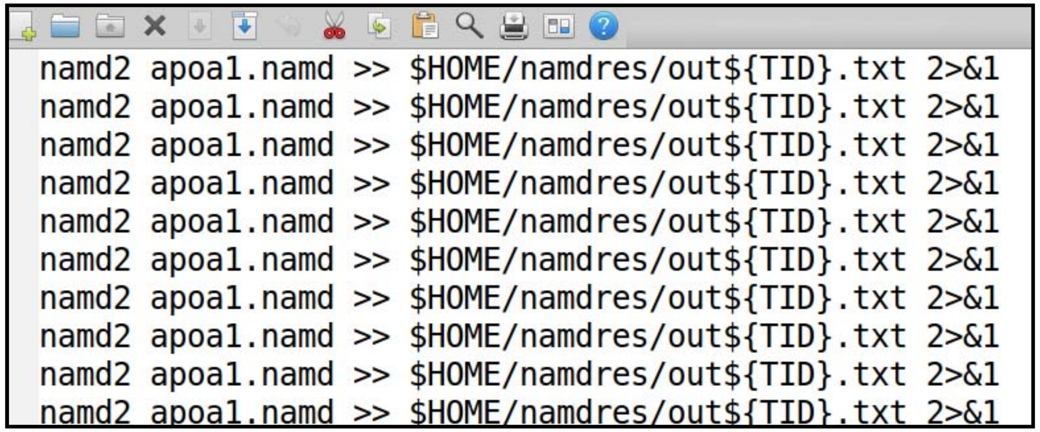
Outlook: R as gluing language / Job scheduler

redisexec

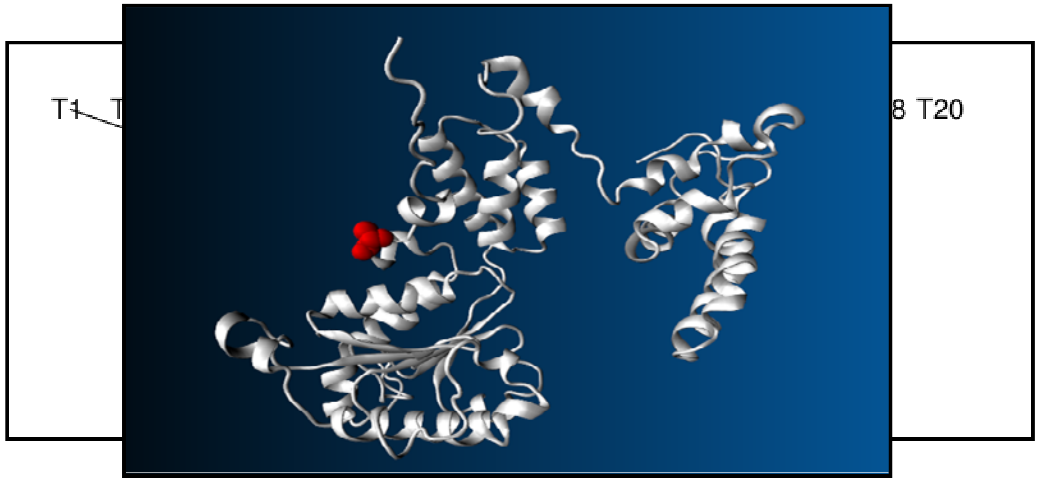
- specify tasks
- describe dependencies

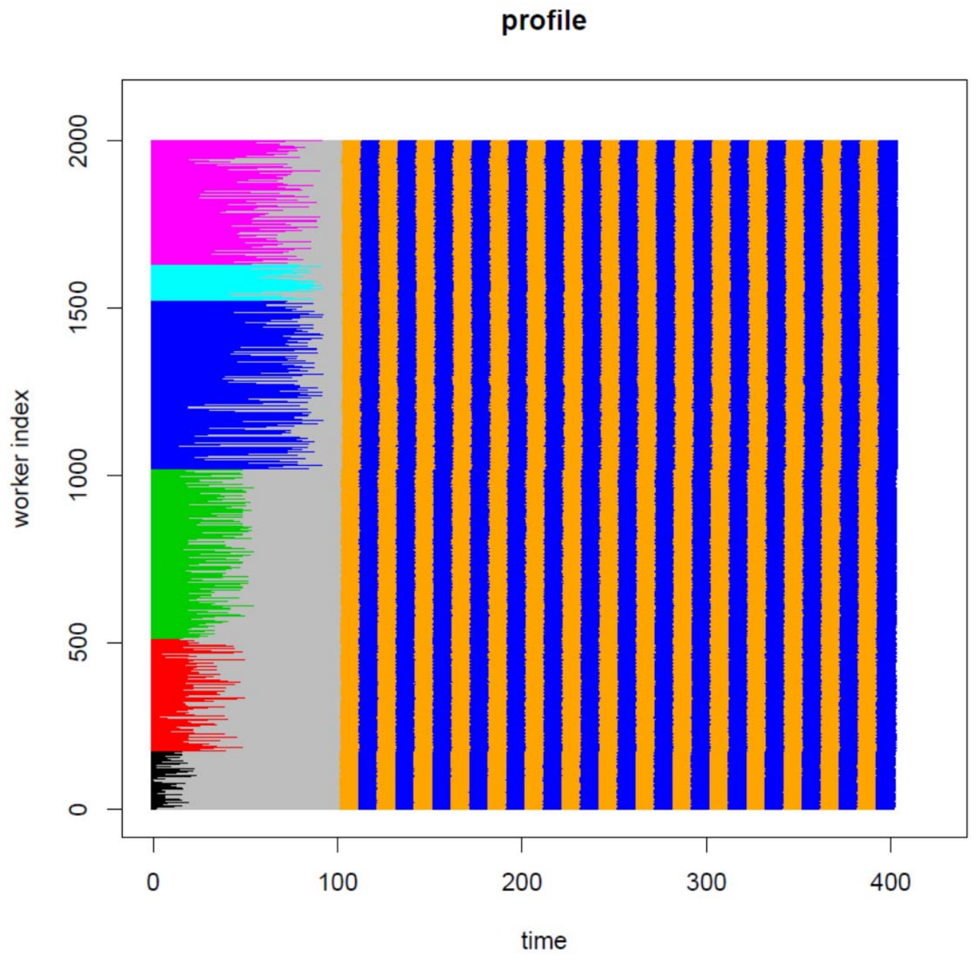
Fault Tolerance

- schedule failed tasks again
- node / connection failures



```
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
namd2 apoa1.namd >> $HOME/namdres/out${TID}.txt 2>&1
```





The End

Thank you for your attention.

SuperMUC System Phase 1 und Phase 2 (back up slide)

Installation phase	Phase 1			Phase 2
Year of Installation	2011	2012	2013	2015
System type	Fat Nodes	Thin Nodes	Many Cores	Haswell Nodes
Processor type	Intel Xeon E7-4870	Intel Xeon E5- 2680	Intel Xeon E5-2680 + Xeon Phi 5110P	Intel Xeon E5-2697v3
Processor Base Frequency [GHz]	2.4	2.7	1.05	2.6
Main Memory per Node [GByte]	256	32	64 + 2x8	64
Cores per Node	40	16	16 + 2x60	28
Number of Nodes	205	9216	32	3072
Number of Cores	8200	147.456	4352	86.016
Theoretical Peak [PFlop/s]	0.078	3.185	0.064 (Phi)	3.6
Main Memory of the System [TByte]	52	288	2.56	197
Typical Power Consumption [MW]	< 2,3			~ 1
Maximum Power Consumption [MW]	3,7			1,5