WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

# High Performance Computing
## with Applications in R

Florian Schwendinger, Gregor Kastner, Stefan Theußl

October 4, 2021

# Outline

Four parts:

- Introduction
- Computer Architecture
- The **parallel** Package
- (Cloud Computing)

# Part I
# Introduction

# About this Course

After this course students will

- be familiar with concepts and parallel programming paradigms in High Performance Computing (HPC),
- have an basic understanding of computer architecture and its implication on parallel computing models,
- be capable of choosing the right tool for time consuming tasks depending on the type of application as well as the available hardware,
- know how to use large clusters of workstations,
- know how to use the **parallel** package,
- be familiar with parallel random number generators in order to run large scale simulations on various nodes (e.g., Monte Carlo simulation),
- understand the cloud; definitions and terminologies.

# Administrative Details

- A good knowledge of R is required.
- You should be familiar with basic Linux commands since we use some in this course.
- The course material is available at `https://atc.r-forge.r-project.org/`.

# What is this course all about?

**High performance computing (HPC)** refers to the use of (parallel) supercomputers and computer clusters. Furthermore, HPC is a branch of computer science that concentrates on developing high performance computers and software to run on these computers.

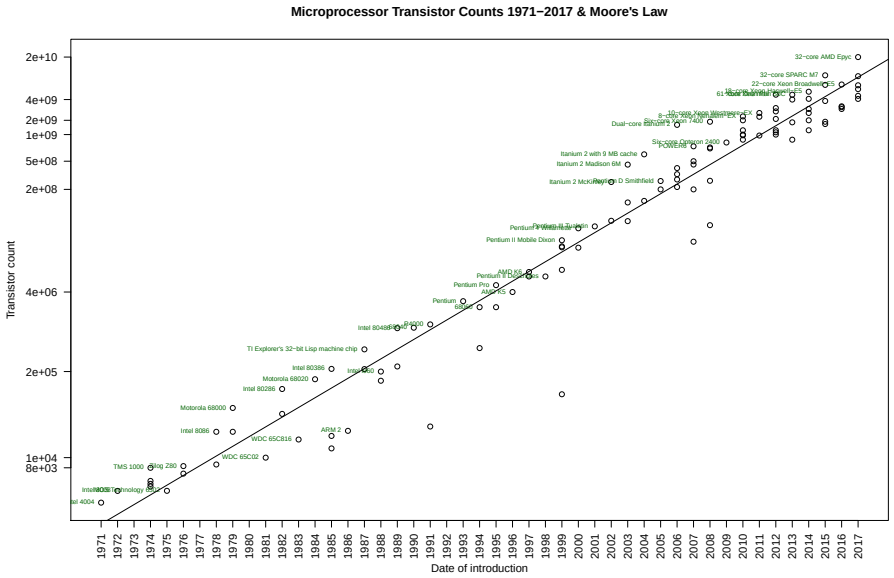
**Parallel computing** is an important area of this discipline. It is referred to as the development of parallel processing algorithms or software.

**Parallelism** is physically simultaneous processing of multiple threads or processes with the objective to increase performance (this implies multiple processing elements).

# Complex Applications in Finance

- In quantitative research we are increasingly facing the following challenges:
  - more accurate and time consuming models (1),
  - computational intensive applications (2),
  - and/or large datasets (3).
- Thus, one could
  - wait (1+2),
  - reduce problem size (3),
- or
  - run similar tasks on independent processors in parallel (1+2),
  - load data onto multiple machines that work together in parallel (3).

# Moore's Law

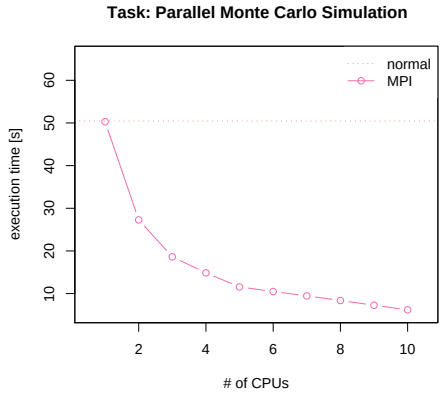

Microprocessor Transistor Counts 1971−2017 & Moore's Law
Transistor count
Date of introduction
2e+10
4e+09
2e+09
1e+09
5e+08
2e+08
4e+06
2e+05
1e+04
8e+03
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
32−core AMD Epoc
32−core SPARC M7
22−core Xeon Broadwell-E5
Dual−core Itanium 2
Itanium 2 with 9 MB cache
Itanium 2 Madison 6M
Pentium II Mobile Dixon
AMD K6
Pentium Pro
AMD K5
Pentium
Intel 8048
Intel 8008
Ti Explorer's 32−bit Lisp machine chip
Intel 80386
Motorola 68020
Intel 80286
Motorola 68000
Intel 8086
ARM 2
WDC 65C816
WDC 65C02
TMS 1000
Zilog Z80

# Some Recent Developments

- Consider Moore's law: number of transistors on a chip doubles every 18 months.
- Until recently this corresponded to a speed increase at about the same rate.
- However, speed per processing unit has remained flat because of:
  - heat
  - power consumption
  - technological reasons
- Graphics cards have been equipped with multiple logical processors on a chip.
  - \+ more than 500 specialized compute cores for a few hundreds of Euros.
  - − special libraries are needed to program these. Still experimental interfaces to languages like R.
- Parallel computing is likely to become essential even for desktop computers.

# Almost Ideal Scenario

Application: pricing a European call option using several CPUs in parallel.




Source: Theußl (2007, p. 104)

Figure: Runtime for a simulation of 5000 payoffs repeated 50 times

# Parallel Programming Tools

- We know that it is hard to write efficient sequential programs.
- Writing correct, efficient parallel programs is even harder.
- However, several tools facilitate parallel programming on different levels.
  - Low level tools: (TCP) sockets [1] for distributed memory computing and threads [2] for shared memory computing.
  - Intermediate level tools: message-passing libraries like MPI [3] for distributed memory computing or OpenMP [4] for shared memory computing.
  - Higher level tools: integrate well with higher level languages like R.
- The latter let us parallelize existing code without too much modification and are the main focus of this course.

[1] http://en.wikipedia.org/wiki/Network_socket
[2] http://en.wikipedia.org/wiki/Thread_(computing)
[3] http://www.mcs.anl.gov/research/projects/mpi/
[4] http://openmp.org/wp/

The best and easiest way to analyze the performance of an application is to measure the execution time. Consequently an application can be compared with an improved version through the execution times.

$$Speedup = \frac{t_s}{t_e} \tag{1}$$

where

- $t_s$ denotes the execution time for a program without enhancements (serial version)
- $t_e$ denotes the execution time for a program using the enhancements (enhanced version)

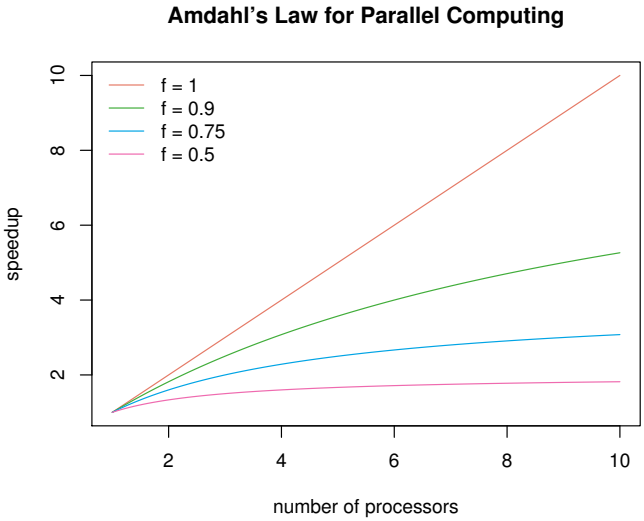
# Parallelizable Computations

- A simple model says intuitively that a computation runs $p$ times faster when split over $p$ processors.
- More realistically, a problem has a fraction $f$ of its computation that can be parallelized thus, the remaining fraction $1 - f$ is inherently sequential.
- Amdahl's law:

$$\text{Maximum Speedup} = \frac{1}{f/p + (1 - f)}$$

- Problems with $f = 1$ are called embarrassingly parallel.
- Some problems are (or seem to be) embarrassingly parallel: computing column means, bootsrapping, etc.

# Amdahl's Law


Amdahl's Law for Parallel Computing
f = 1
f = 0.9
f = 0.75
f = 0.5
speedup
2
4
6
8
10
number of processors

# Literature

- Schmidberger et al. (2009)
- HPC Task View `http://CRAN.R-project.org/view=HighPerformanceComputing` by Dirk Eddelbuettel
- Kontoghiorghes (2006)
- Rossini et al. (2003)
- Notes on WU's cluster and cloud system can be found at `http://statmath.wu.ac.at/cluster/` and `http://cloud.wu.ac.at/manual/`, respectively.

# Applications

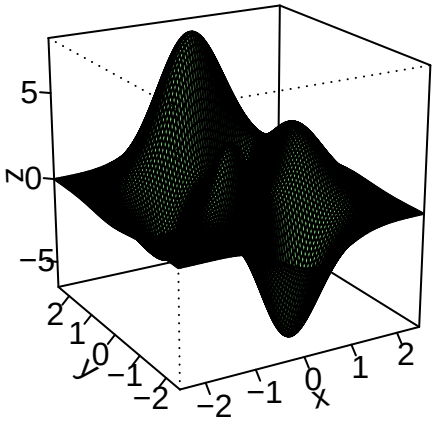
We want to improve the following applications:

- Global optimization using a multistart approach.
- Option pricing using Monte Carlo Simulation.
- Markov Chain Monte Carlo (MCMC) – cf. lecture on Bayesian Computing.

These will be the assignments for this course.

We want to find the global minimum of the following function:

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$



# European Call Options

- Underlying $S_t$ (non-dividend paying stock)
- Expiration date or maturity $T$
- Strike price $X$
- Payoff $C_T$

$$\begin{aligned} C_T &= \begin{cases} 0 & \text{if } S_T \leq X \\ S_T - X & \text{if } S_T > X \end{cases} \\ &= \max\{S_T - X, 0\} \end{aligned} \tag{2}$$

- Can also be priced analytically via Black-Scholes-Merton model

# MC Algorithm (1)

1. Sample a random path for $S$ in a risk neutral world.
2. Calculate the payoff from the derivative.
3. Repeat steps 1 and 2 to get many sample values of the payoff from the derivative in a risk neutral world.
4. Calculate the mean of the sample payoffs to get an estimate of the expected payoff in a risk neutral world.
5. Discount the expected payoff at a risk free rate to get an estimate of the value of the derivative.

# MC Algorithm (2)

**Require:** option characteristics (S, X, T), the risk free yield $r$, the number of simulations $n$

1: **for** $i = 1 : n$ **do**
2: generate $Z^i$
3: $S_T^i = S(0)e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}Z^i}$
4: $C_T^i = e^{-rT}\mathbf{max}(S_T^i - X, 0)$
5: **end for**
6: $\widehat{C}_T^n = \frac{C_T^1+\ldots+C_T^n}{n}$

# Sampling Accuracy

The number of trials carried out depends on the accuracy required. If $n$ independent simulations are run, the standard error of the estimate $\widehat{C}_T^n$ of the payoff is

$$\frac{s}{\sqrt{n}}$$

where $s$ is the (estimated) standard deviation of the discounted payoff given by the simulation. According to the central limit theorem, a 95% confidence interval for the "true" price of the derivative is given asymptotically by

$$\widehat{C}_T^n \pm \frac{1.96s}{\sqrt{n}}.$$

The accuracy of the simulation is therefore inversely proportional to the square root of the number of trials $n$. This means that to double the accuracy the number of trials have to be quadrupled.

# General Strategies

- "Pseudo" parallelism by simply starting the same program with different parameters several times.
- Implicit parallelism, e.g., via parallelizing compilers, or built-in support of packages.
- Explicit parallelism with implicit decomposition.
  - Parallelism easy to achieve using compiler directives (e.g., OpenMP).
  - Incrementally parallelizing sequential code possible.
- Explicit parallelism, e.g., with message passing libraries.
  - Use R packages porting API of such libraries.
  - Development of parallel programs is difficult.
  - Deliver good performance.

# Part II
## Computer Architecture

Shared Memory Systems (SMS) host multiple processors which share one global main memory (RAM), e.g., multi core systems.

Distributed Memory Systems (DMS) consist of several units connected via an interconnection network. Each unit has its own processor with its own memory.

DMS include:

- **Beowulf Clusters** are scalable performance clusters based on commodity hardware, on a private system network, with open source software (Linux) infrastructure (e.g., clusterwu@WU).
- **The Grid** connects participating computers via the Internet (or other wide area networks) to reach a common goal. Grids are more loosely coupled, heterogeneous, and geographically dispersed.

**The Cloud** or cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources.

# Excursion: Process vs. Thread

**Processes** are the execution of lists of statements (a sequential program). Processes have their own state information, use their own address space and interact with other processes only via an interprocess communication mechanism generally managed by the operating system. (A master process may spawn subprocesses which are logically separated from the master process)

**Threads** are typically spawned from processes for a short time to achieve a certain task and then terminate – **fork/join principle**. Within a process threads share the same state and same memory space, and can communicate with each other directly through shared variables.

# Excursion: Overhead and Scalability

**Overhead** is generally considered any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to be utilized or expanded to enable a particular goal.

**Scalability** is referred to as the capability of a system to increase the performance under an increased load when resources (in this case CPUs) are added.

Scaling Efficiency

$$E = \frac{t_s}{t_e p}$$

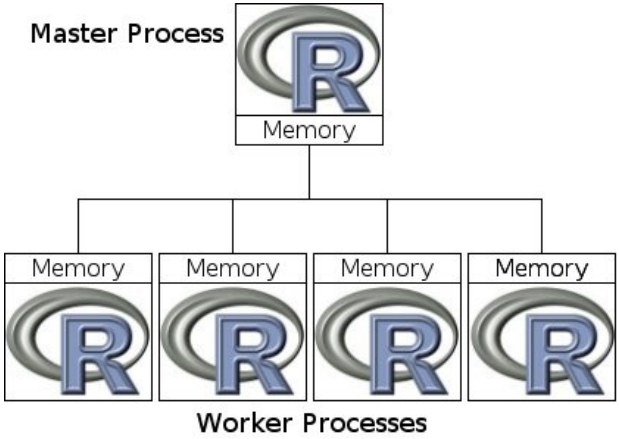
# Shared Memory Platforms

- Multiple processors share one global memory
- Connected to global memory mostly via bus technology
- Communication via shared variables
- SMPs are now commonplace because of multi core CPUs
- Limited number of processors (up to around 64 in one machine)

# Distributed Memory Platforms

- Provide access to cheap computational power
- Can easily scale up to several hundreds or thousands of processors
- Communication between the nodes is achieved through common network technology
- Typically we use message passing libraries like MPI or PVM

# Distributed Memory Platforms



# Shared Memory Computing

Parallel computing involves splitting work among several processors. Shared memory parallel computing typically has

- single process,
- single address space,
- multiple *threads* or light-weight processes,
- all data is shared,
- access to key data needs to be synchronized.

# Typical System

| IBM System p 550 | |
|---|---|
| 4 | 2-core IBM POWER6 @ 3.5 GHz |
| 128 | GB RAM |

This is a total of 8 64-bit computation nodes which have access to 128 gigabytes of shared memory.

# Cluster Computing

Cluster computing or distributed memory computing usually has

- multiple processes, possibly on different computers
- each process has its own address space
- data needs to be exchanged explicitly
- data exchange points are usually points of synchronization

Hybrid models, i.e., combinations of distributed and shared memory computing are possible.

# Cluster@WU (Hardware)

Cluster@WU consists of computation nodes accessible via so-called *queues* (node.q, hadoop.q), a file server, and a login server.

| | |
|---|---|
| **Login server** | |
| 2 | Quad Core Intel Xeon X5550 @ 2.67 GHz |
| 24 | GB RAM |
| **File server** | |
| 2 | Quad Core Intel Xeon X5550 @ 2.67 GHz |
| 24 | GB RAM |
| **node.q – 44 nodes** | |
| 2 | Six Core Intel Xeon X5670 @ 2.93 GHz |
| 24 | GB RAM |

This is a total of 528 64-bit computation cores (544 including login- and file server) and more than 1 terabyte of RAM.

# Connection Technologies

- Sockets: everything is managed by R, thus "easy". Socket connections run over TCP/IP, thus usable almost on any given system. Advantage: no additional software required.
- Message Passing Interface (MPI): basically a definition of a networking protocol. Several different implementations, but openMPI (see `http://www.open-mpi.org/`) is the most common and widely used.
- Parallel Virtual Machine (PVM): nowadays obsolete.
- NetWorkSpaces (NWS): is a framework for coordinating programs written in scripting languages.

# Cluster@WU (Software)

- Debian GNU/Linux
- Compiler Collections
  - GNU 4.4.7 (gcc, g++, gfortran, ...), [g]
- R, some packages from CRAN
  - `R-g` latest R-patched compiled with [g]
  - `R-<g>-<date>` R-devel compiled at `<date>`
- Linear algebra libraries (BLAS, LAPACK, INTEL MKL)
- OpenMPI, PVM and friends
- various editors (emacs, vi, nano, etc.)

# Cluster Login Information

You can use the following account:

- **login host:** `clusterwu.wu.ac.at`
- **user name:** provided in class
- **password:** provided in class

The software packages R 3.4.3, OpenMPI 1.10.0, **Rmpi** 0.6-7 and **snow** 0.4-1 are pre-installed for this account. All relevant data and code is supplied in the directory '`~/HPC_examples`'.

- Remote connection can be established by
  - Secure shell (ssh): type `ssh <username>@clusterwu.wu.ac.at` on the terminal
  - Windows
    - MobaXterm (`http://mobaxterm.mobatek.net/download-home-edition.html`)
    - Combination of PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/`) and WinSCP (`http://winscp.net`)
- First time configuration:
  - Add the following lines to the beginning of your '`~/.bashrc`' (and make sure that '`~/.bashrc`' is sourced at login). Done for you.

```
## OPENMPI
export MPI=/opt/libs/openmpi-1.10.0-GNU-4.9.2-64/
export PATH=${MPI}/bin:/opt/R/bin:/opt/sge/bin/lx-amd64:${PATH}
export LD_LIBRARY_PATH=${MPI}/lib:${MPI}/lib/openmpi:${LD_LIBRARY_PATH}
## Personal R package library
export R_LIBS="~/lib/R/"
```

  - Create your package library using `mkdir -p ~/lib/R`. Done for you.

**Son of Grid Engine** (SGE) is an open source cluster resource management and scheduling software. It is used to run cluster **jobs** which are user requests for resources (i.e., actual computing instances or **nodes**) available in a cluster/grid.
In general the SGE has to match the available resources to the requests of the grid users. SGE is responsible for

- accepting jobs from the outside world
- delay a job until it can be run
- sending job from the holding area to an execution device (node)
- managing running jobs
- logging of jobs

Useful SGE commands:

- `qsub` submits a job.
- `qstat` shows statistics of jobs running on cluster@WU.
- `qdel` deletes a job.
- `sns` shows the status of all nodes in the cluster.

# Submitting SGE Jobs

1. Login,
2. create a plain text file (e.g., 'myJob.qsub') with the job description, containing e.g.:

```
#!/bin/bash
## This is my first cluster job.

#$ -N MyJob

R-g --version
sleep 10
```

3. then type `qsub myJob.qsub` and hit enter.
4. Output files are provided as '<jobname>.o<jobid>' (standard output) and '<jobname>.e<jobid>' (error output), respectively.

# SGE Jobs

An SGE job typically begins with commands to the grid engine. These commands are prefixed with #$.
E.g., the following arguments can be passed to the grid engine:

- `-N` specifying the actual jobname
- `-q` selecting one of the available queues. Defaults to node.q.

`-pe [type] [n]` stets up a parallel environment of type `[type]` reserving `[n]` cores.

`-t <first>-<last>:stepsize` creates a job-array (e.g. `-t 1-20:1`)

`-o [path]` redirects stdout to path

`-e [path]` redirects stderr to path

`-j y[es]` merges stdout and stderr into one file

For an extensive listing of all avialable arguments type **qsub -help** into your terminal.

# A Simple MPI Example

- We want our processes sent us the following message: `"Hello World from processor <ID>"` where `<ID>` is the processor ID (or *rank* in MPI terminology).
- MPI uses the master-worker paradigm, thus a master process is responsible for starting (spawning) worker processes.
- In R we can utilize the MPI library via package **Rmpi**.

Expert note: **Rmpi** can be installed using `install.packages("Rmpi", configure.args="--with-mpi=/opt/libs/openmpi-1.10.4-GNU-4.9.2-64")` in R. Done for you.

# A Simple MPI Example

- Job script ('Rmpi_hello_world.qsub'):

```
#!/bin/sh
## A simple Hello World! cluster job
#$ -cwd                  # use current working dirrectory
#$ -N RMPI_Hello         # set job name
#$ -o RMPI_Hello.log     # write stdout to RMPI_Hello.log
#$ -j y                  # merge stdout and stderr
#$ -pe mpi 4            # use the parallel environment "mpi" with 4 slots

R-g --vanilla < Rmpi_hello_world.R
```

- R code ('Rmpi_hello_world.R'):

```
library("Rmpi")

slots <- as.integer(Sys.getenv("NSLOTS"))

mpi.is.master()
mpi.get.processor.name()
mpi.spawn.Rslaves(nslaves = slots)
mpi.remote.exec(mpi.comm.rank())

hello <- function(){
  sprintf("Hello World from processor %d", mpi.comm.rank())
}

mpi.bcast.Robj2slave(hello)
mpi.remote.exec(hello())
```

- MPI is used via package **Rmpi**.

# Part III
# The **parallel** Package

# The **parallel** Package

- Available since R version 2.14.0.
- Builds on the CRAN packages **multicore** and **snow**.
  - **multicore** for parallel computation on shared memory (unix) plattforms
  - **snow** for parallel computation on distributed memory plattforms
- Allows to use both, shared- (threads, POSIX systems only) and distributed memory systems (sockets).
- Additionally, package **snow** extends **parallel** with other communication technologies for distributed memory computing like MPI.
- Integrates handling of random numbers.

For more details see the **parallel** vignette .

# Functions of the Parallel Package

## Shared Memory

| Function | Description | Example |
|---|---|---|
| detectCores | detect the number of CPU cores | `ncores <- detectCores()` |
| mclapply | parallelized version of lapply | `mclapply(1:5, runif, mc.cores = ncores)` |

## Distributed memory

| Function | Description | Example |
|---|---|---|
| makeCluster | start the cluster [1] | `cl <- makeCluster(10, type="MPI")` |
| clusterSetRNGStream | set seed on cluster | `clusterSetRNGStream(cl, 321)` |
| clusterExport | exports variables to the workers | `clusterExport(cl, list(a=1:10, x=runif(10)))` |
| clusterEvalQ | evaluates expressions on workers | `clusterEvalQ(cl, {x <- 1:3 myFun <- function(x) runif(x)})` |
| clusterCall | calls a function on all workers | `clusterCall(cl, function(y) 3 + y, 2)` |
| parLapply | parallelized version of lapply | `parLapply(cl, 1:100, Sys.sleep)` |
| parLapplyLB | parLapply with load balancing | `parLapplyLB(cl, 1:100, Sys.sleep)` |
| stopCluster | stop the cluster | `stopCluster(cl)` |

[1] (allowed types are PSOCK, FORK, SOCK, MPI and NWS)

# A Simple Multicore Example

- We use the **parallel** package to parallelize certain constructs.
- E.g., instead of `lapply()` we use `mclapply()` which implicitly applies a given function to the supplied parameters in parallel.
- Example: global optimization using a multistart approach.

# A Simple Multicore Example

## Sequential:

```
fun <- function(x){
    3*(1-x[1])^2*exp(-x[1]^2-(x[2]+1)^2) - 10 * (x[1]/5 - x[1]^3 - x[2]^5) * exp(-x[1]^2 - x[2]^2) - 1/3 * exp(-(x[1]+1)^2 - x[2]^2)
}
start <- list( c(0, 0), c(-1, -1), c(0, -1), c(0, 1) )
seqt <- system.time(sol <- lapply(start, function(par) optim(par, fun, method = "Nelder-Mead",
          lower = -Inf, upper = Inf, control = list(maxit = 1000000, beta = 0.01, reltol = 1e-15)))
        )["elapsed"]
seqt
```

## Parallel:

```
require(parallel)
ncores <- detectCores()
part <- system.time(sol <- mclapply(start, function(par)
                     optim(par, fun, method = "Nelder-Mead", lower = -Inf, upper = Inf,
                           control = list(maxit = 1000000, beta = 0.01, reltol = 1e-15)), mc.cores = ncores)
                   )["elapsed"]
```

- You need to be careful when generating pseudo random numbers in parallel, especially if you want the streams be independend and reproducible.
- Identical streams produced on each node are likely but not guaranteed.
- Parallel PRNGs usually have to be set up by the user. E.g., via `clusterSetRNGStream()` in package **parallel**.
- The source file '`snow_pprng.R`' shows how to use such parallel PRNGs.

# Example: Pricing European Options (1)

MC sim using the **parallel** package. Job script ('snow_mc_sim.qsub'):

```
#!/bin/sh
## Parallel MC simulation using parallel/snow

#$ -N SNOW_MC
#$ -pe mpi 4

R-g --vanilla < snow_mc_sim.R
```

Note: to run this example package **snow** has to be installed since no functionality to start MPI clusters is provided with package **parallel**.

# Example: Pricing European Options (2)

R script ('snow_mc_sim.R'):

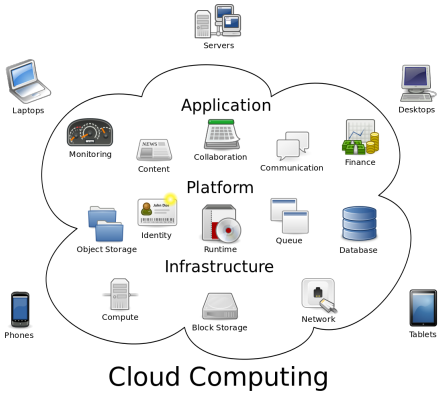
```
require("parallel")

source("HPC_course.R")

## number of paths to simulate
n <- 400000

slots <- as.integer(Sys.getenv("NSLOTS"))

## start MPI cluster and retrieve the nodes we are working on.
cl <- snow::makeMPIcluster(slots)
clusterCall(cl, function() Sys.info()[c("nodename","machine")])

## note that this must be an integer
sim_per_slot <- as.integer(n/slots)

## setup PRNG
clusterSetRNGStream(cl, iseed = 123)
price <- MC_sim_par(cl, sigma = 0.2, S = 120, T = 1, X = 130, r = 0.05, n_per_node = sim_per_slot, nodes = slots)
price

## finally shut down the MPI cluster
stopCluster(cl)
```

# Literature


- R-core (2013)
- Mahdi (2014)
- Jing (2010)
- McCallum and Weston (2011)

# Part IV
# Cloud Computing

What is the Cloud?




Source: Wikipedia, `http://en.wikipedia.org/wiki/File:Cloud_computing.svg`, accessed 2011-12-05

According to the NIST Definition of Cloud Computing, see `http://csrc.nist.gov/groups/SNS/cloud-computing/`, the cloud model

- promotes availability,
- is composed of five essential characteristics (On-demand self-service, Broad network access, Resource pooling, Rapid elasticity, Measured Service),
- three service models:

  IaaS: Infrastructure as a Service
  PaaS: Protocol as a Service
  SaaS: Software as a Service

- and four deployment models (private, community, public, hybrid clouds).

# Essential Characteristics

- *On-demand self-service*. Provision computing capabilities, such as server time and network storage, as needed.
- *Broad network access*. Service available over the network and accessed through API (via mobile, laptop, Internet).
- *Resource pooling*. Different physical and virtual resources (e.g., storage, memory, network bandwidth, virtual machines) are dynamically assigned and reassigned according to consumer demand.
- *Rapid elasticity*. Capabilities can be rapidly and elastically provisioned.
- *Measured Service*. Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

# Service Models

- *Cloud Infrastructure as a Service (IaaS)*. provides (abstract) infrastructure as a service (computing environments available for rent, e.g., Amazon EC2, see `http://aws.amazon.com/ec2/`)
- *Cloud Platform as a Service (PaaS)*. corresponds to programming environments, or, platform as a service (development environment for web applications, e.g., Google App Engine).
- *Cloud Software as a Service (SaaS)*. refers to the provision of software as a service (web applications like office environments, e.g., Google Docs).

# Deployment Models

- *Private cloud*. The cloud infrastructure is operated solely for an organization (e.g., **wu.cloud**).
- *Community cloud*. The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns.
- *Public cloud*. The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services (e.g., Amazon EC2).
- *Hybrid cloud*. The cloud infrastructure is a composition of two or more clouds (private, community, or public) bound together by standardized or proprietary technology.

# Terminology

**Image** (also called "appliance"): A stack of an operating system and applications bundled together. **wu.cloud** users can select from a variety of appliances (both GNU/Linux and Windows 7 based) with standard scientific tools (R, Matlab, Mathematica, STATA, etc.).

**Instance:** Cloud images are started in their own separate virtual machine environments (with up to 196 GB RAM and 8 CPU cores). This process is called "instancing", running appliances are called "instances".

**EBS Volume:** is "off-instance storage that persists independently from the life of an instance." (Amazon, 2011). EBS Volumes can be attached to running instances to provide virtual disk space for large datasets, calculation results and custom software configurations.

# Private Clouds

**wu.cloud** is a private cloud service and thus the following characteristics hold.

- Emulate public cloud on (existing) private resources,
- thus, provides benefits of clouds (elasticity, dynamic provisioning, multi-OS/arch operation, etc.),
- while maintaining control of resources.
- Moreover, there is always the option to scale out to the public cloud (going hybrid).

# wu.cloud

**wu.cloud** is

- solely operated for WU members and projects,
- thus, network access only via Intranet/VPN (`https://vpn.wu.ac.at`),
- on-demand self-service,
- resource pooling via virtualization,
- extensible/elastic,
- Infrastructure as a Service (IaaS),
- Platform as a Service (PaaS).

- **wu.cloud** is a private cloud system based on the open source software package *Eucalyptus* (see `http://open.eucalyptus.com/`).
- Accessible via `http://cloud.wu.ac.at/`.
- Consists of a *frontend* (website, management software) and a *backend* (providing resources) system.

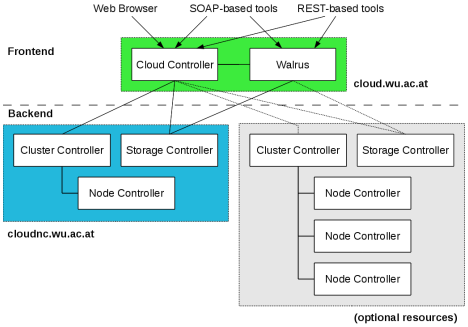



Figure: wu.cloud setup

# **wu.cloud** Hardware

Backend system:

(c) 2010 IBM Corporation, from Datasheet XSD03054-USEN-05

- 2x IBM X3850 X5
- 8x8 (64) core Intel Xeon CPUs 2.26 GHz
- 1 TB RAM
- EMC$^2$ Storage Area Network: 7 TB fast + 4 TB slow disks
- Suse Linux Enterprise Server 11 SP1
- Xen 4.0.1
- Eucalyptus backend components (cluster, storage, node controller)

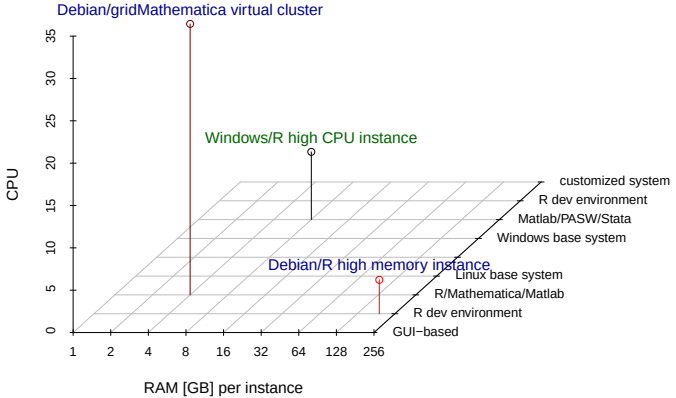
Frontend System:

- Virtual (Xen) instance
- Apache Webserver
- Eucalyptus frontend components (cloud controller, walrus)

# **wu.cloud** Characteristics

**wu.cloud** aims at scaling in three different dimensions:

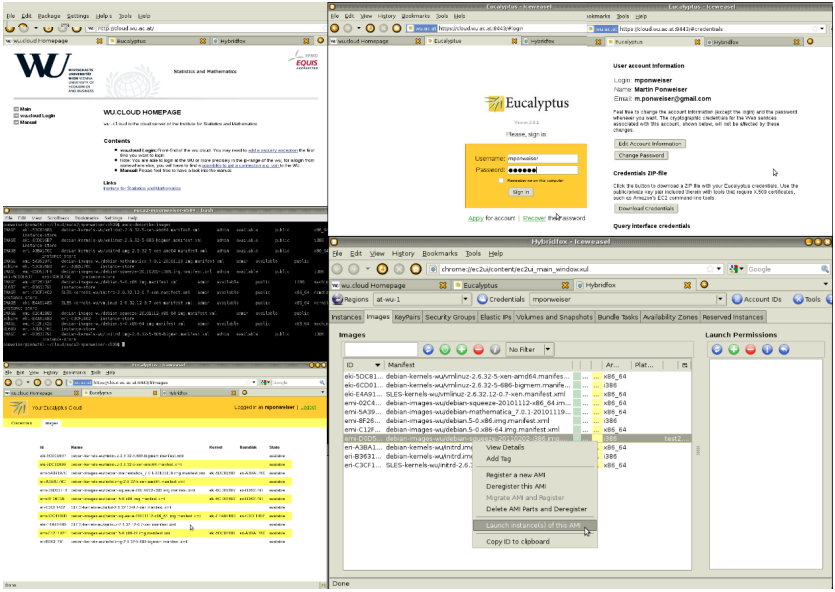
- Compute-nodes: number of cloud instances and cores employed
- Memory: amount of memory per instance requested
- Software: Windows vs. Linux and software packages installed



# **wu.cloud** User Interface

- Amazon EC2 API
  - allows for using tools like ec2/euca2ools, Hybridfox, etc., primarily designed for EC2
  - transparent use of **wu.cloud** and EC2/S3 side by side
- Remote connection to cloud instances can be established by
  - Secure shell (ssh), PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/`)
  - VNC (Linux)
  - Remote Desktop (Windows)

# **wu.cloud** User Interface



# Contact

Florian Schwendinger
Institute for Statistics and Mathematics
email: florian.schwendinger@wu.ac.at
URL: http://www.wu.ac.at/statmath/faculty_staff/faculty/fschwendinger

WU Vienna
Welthandelsplatz 1/D4/level 4
1020 Wien
Austria

# References


L. Jing. *Parallel Computing with R and How to Use it on High Performance Computing Cluster*, 2010. URL `http://datamining.dongguk.ac.kr/R/paraCompR.pdf`.

E. Kontoghiorghes, editor. *Handbook of Parallel Computing and Statistics*. Chapman & Hall, 2006.

E. Mahdi. A survey of r software for parallel computing. *American Journal of Applied Mathematics and Statistics*, 2(4): 224–230, 2014. ISSN 2333-4576. doi: 10.12691/ajams-2-4-9. URL `http://pubs.sciepub.com/ajams/2/4/9`.

Q. E. McCallum and S. Weston. *Parallel R*. O'Reilly Media, Inc., 2011. ISBN 1449309925, 9781449309923.

R-core. *Package 'parallel'*, 2013. URL `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`. https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.R.

A. Rossini, L. Tierney, and N. Li. Simple Parallel Statistical Computing in R. *UW Biostatistics Working Paper Series*, (Working Paper 193), 2003. URL `http://www.bepress.com/uwbiostat/paper193`.

M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art in parallel computing with r. *Journal of Statistical Software*, 31(1):1–27, 8 2009. ISSN 1548-7660. URL `http://www.jstatsoft.org/v31/i01`.

S. Theußl. Applied high performance computing using R. Master's thesis, WU Wirtschaftsuniversität Wien, 2007. URL `http://statmath.wu-wien.ac.at/~theussl/publications/thesis/Applied_HPC_Using_R-Theussl_2007.pdf`.