Karlsruher Institut für Technologie

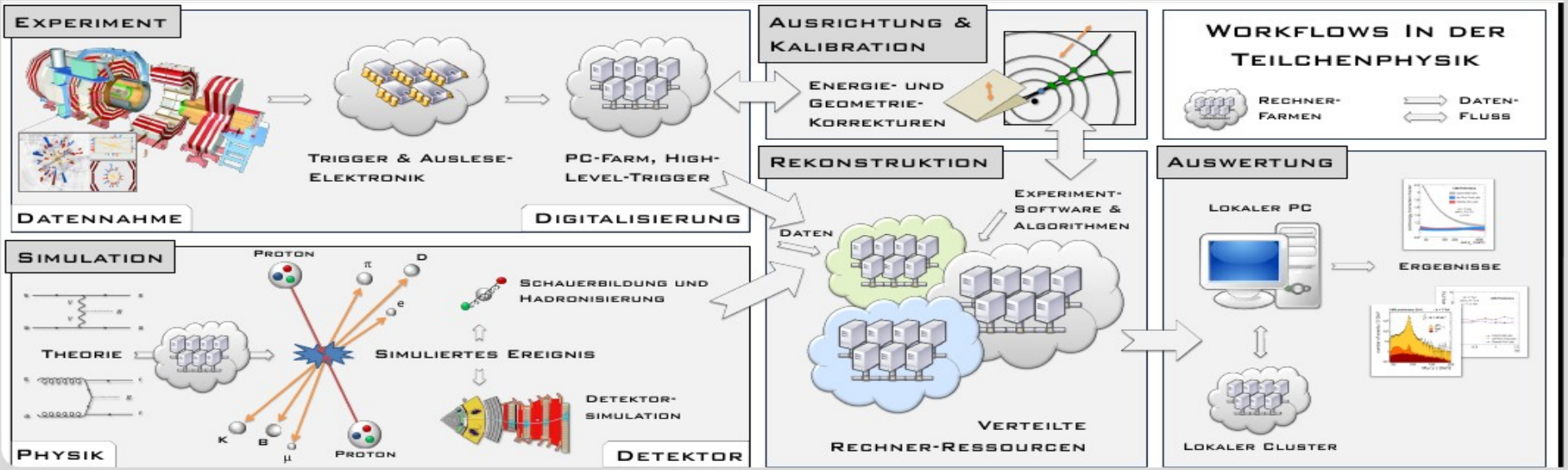
Vorlesung: Rechnernutzung in der Physik

Softwareentwicklung & Computing in der Wissenschaft

Günter Quast

Fakultät für Physik
Institut für Experimentelle Kernphysik

WS 2023/24



Hinweis: Sommerstudentenprogramme

Große Forschungseinrichtungen bieten **„Sommerstudentenprogramme“** an:

- akademisches Programm mit Vorlesungen und Führungen
- Mitarbeit in Forschungsprojekten
- auch als Bachelor- oder Teil einer Masterarbeit möglich

(benötigt Koordination und Abstimmung mit Universität)

CERN summer student program:

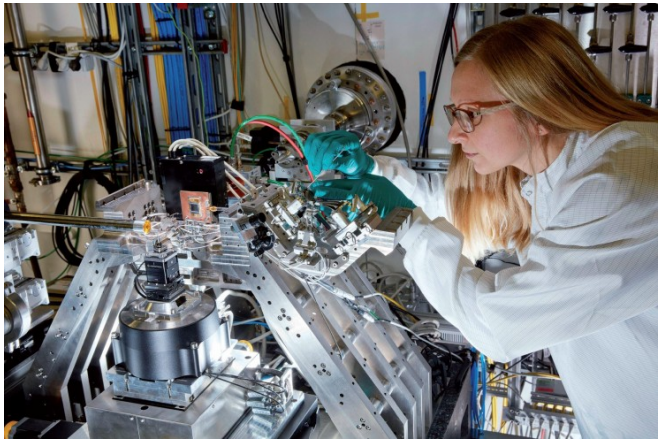
https://careers.cern/summer

- work on an advanced project in an experimental or engineering team
- attendance of specially prepared lectures, discussion sessions, workshops and visits theoretical and experimental particle physics, engineering and computing.

DESY summer student program:

https://summerstudents.desy.de/

- Photon science
- Elementary particle physics & Accelerators
- Astroparticle physics



Block 2: Rechner & Software

- 1. Rechnerarchitekturen, Parallelisierung und Vektorisierung
- 2. Kollaborative Softwareentwicklung

Interaktion mit dem Computer

Steuerung von Arbeitsabläufen

Arbeitsabläufe sind meist mehrschrittig und komplex

Beispiel: **Vergleich von Messdaten mit Modell**

- Berechnung der Modellvorhersage(n)
- Datenaufnahme und Korrekturen
- Statistische Datenanalyse und Cross-Checks, Systematische Unsicherheiten
- Dokumentation und Sicherstellung der Reproduzierbarkeit der Ergebnisse

interaktiv / Maus	versus	**Script**

++ sehr intuitiv

\+ leicht erlernbar

\- kaum Reproduzierbarkeit bei komplexen „Workflows“

\- - spezielle Menü-Optionen nicht im Ergebnis dokumentiert

\- - keine permanente Dokumentation der Vorgehensweise

\- - in der Regel auf eine Anwendung und deren Tools festgelegt

0 erfordert Einarbeitung

\+ relativ leicht erlernbar

++ Skript enthält alle relevanten Schritte: vollständige Dokumentation des Arbeitsablaufs

++ vollständig reproduzierbar

++ einfache Nutzung verschiedener Programm-Pakete und Tools

C/C++ vs. Python

Python ist eine interpretierte Sprache wie Shell-Scripts oder Pearl, Basic
– im Gegensatz zu Compiler-Sprachen wie C / C++ / Fortran

Python:
* interpretiert
  + kurze Entwicklungszyklen
  – evlt. langsamer bei Ausführung

* moderater, aber ständig wachsender Sprachumfang

* Vielzahl von Hilfspaketen

* als Skritp-Sprache ideal geeignet zur Verknüpfung von Arbeitsschritten

* Harware-nahe Programmierung erschwert bis unmöglich

C/C++
* Compilieren/Linken notwendig
  – langsamer Entwicklungszyklus
  + schneller Code bei Ausführung

* komplexer Sprachumfang

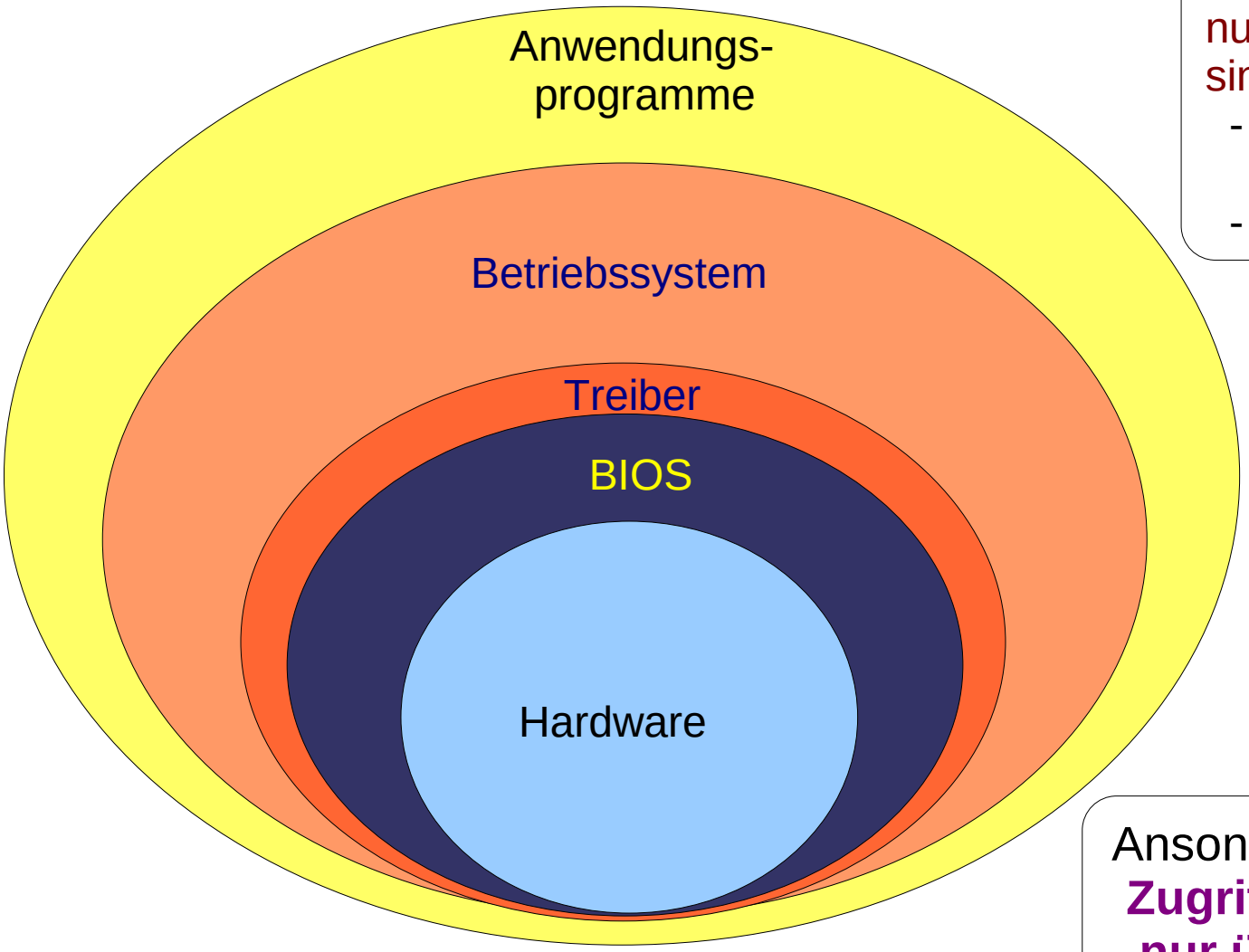
* Vielzahl von „Bibliotheken“ mit Hilfsfunktionen/Anwendungen

* ideal für hochperformante numerische Berechnungen oder harware-nahe Programmierung

→ **für „einfache“ Aufgaben ist Python ideal;**
in der Praxis empfiehlt sich ein „heterogener Ansatz“:
Zeit- bzw. Speicher-kritische Probleme in C/C++, verknüpft mit Python

Betriebssystem

Betriebssystem



Direkter Zugriff von Anwenderprogrammen **auf Hardware**
nur in Ausnahmefällen sinnvoll / möglich:
- ein-Prozess-Betrieb (z.B. Spiele!)
- Echtzeit-Anwendungen

Ansonsten:
Zugriff auf Hardware nur über Betriebssystem

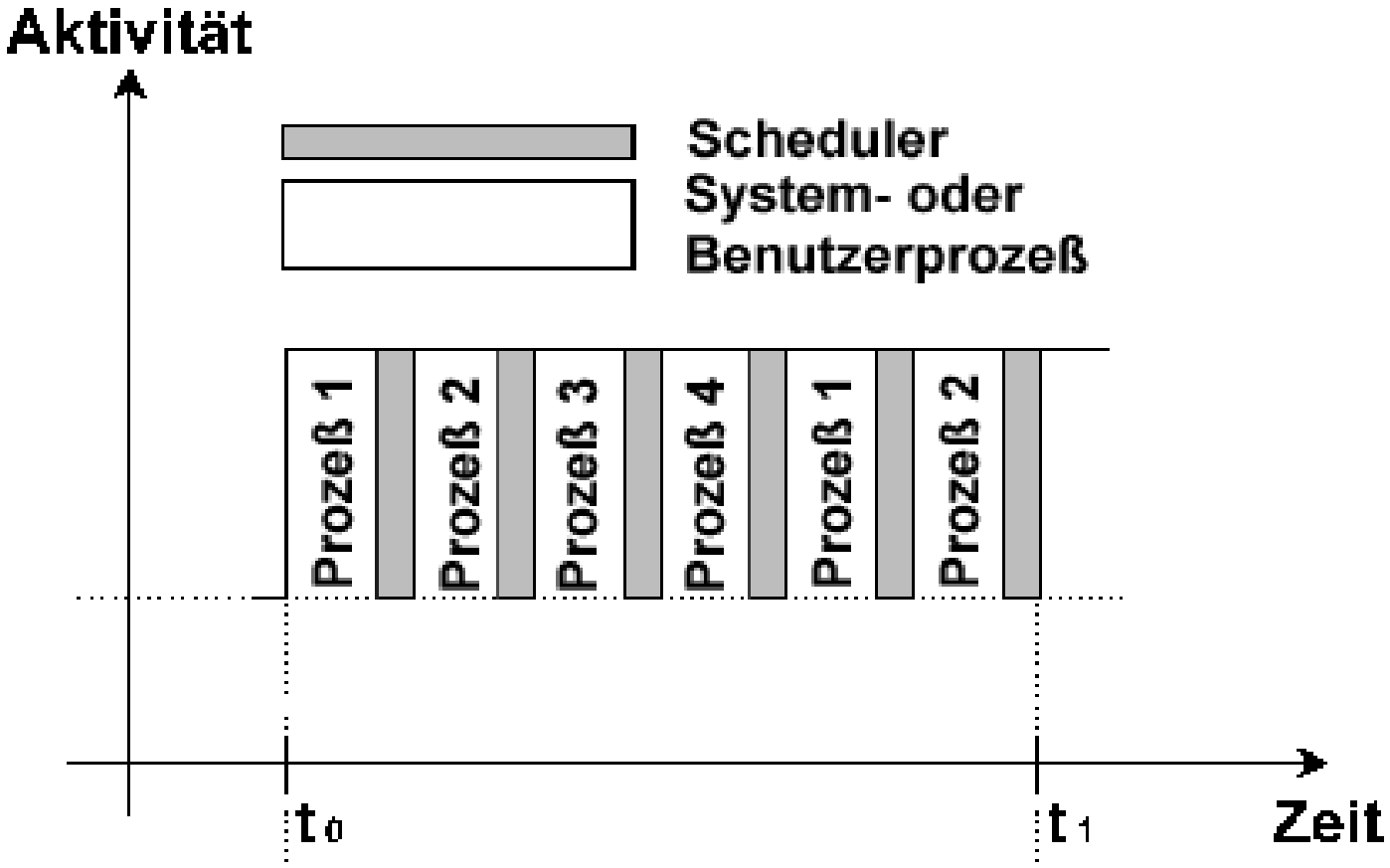
Aufgaben des Betriebssystems

- Verbergen der Komplexität der Maschine vor dem Anwender („Abstraktion“)
- Bereitstellung einer normierten (Software-)Schnittstelle für Programme
- ggf. Compiler, Linker, Editor, Entwicklungsumgebung, …
- Bereitstellung einer Nutzerschnittstelle (Shell, Command-Interpreter, graphische Oberfläche, …)
- Steuerung und Überwachung von Nutzerprogrammen, evtl. Multi-Prozess und Multi-User-Betrieb, Ressourcen-Zuteilung
- Verwalten und Schützen der Ressourcen der Maschine (CPU, Speicher, I/O)

Unterscheidung zwischen Betriebssystem-Komponenten und Hilfsprogrammen nicht immer eindeutig

Multi-Tasking

Alle modernen Betriebssysteme führen mehrere Prozesse („tasks“) quasi-parallel aus, indem abwechselnd jedem Prozess CPU-Zeit zugeteilt wird

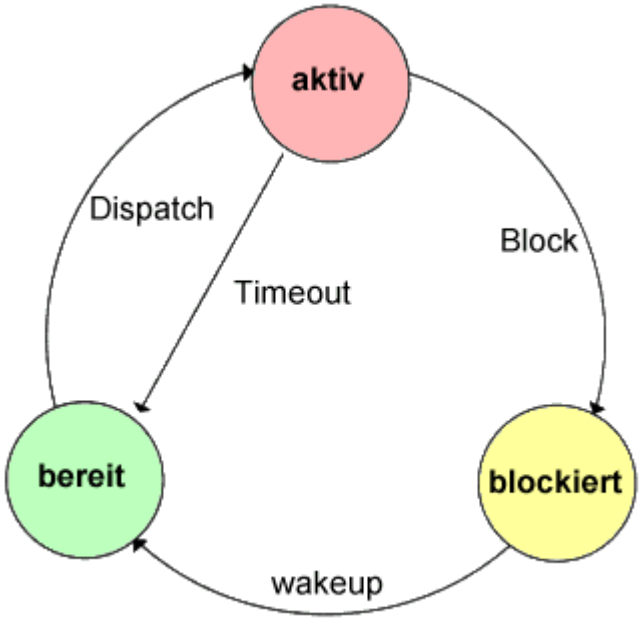


Vorteil: Wartezeiten eines Prozesses auf I/O können von anderen Prozessen sinnvoll genutzt werden

Prozessverwaltung

Scheduler sorgt für **gerechte und effiziente Zuteilung von CPU-Zeit**

- Kooperatives Multitasking:
  Prozess gibt CPU frei
- Verdrängendes Multitasking:
  Prozess wird durch Timer-Interrupt unterbrochen



Prozess (task, thread) wird beschrieben durch:
- Zugeordnete Speicherbereiche: Programm-, Daten-, Stack-Segment
- Statusinformationen: Registerinhalte, Zustand
- Zugeordnete Ressourcen (Dateien, I/O)
- ID, Benutzer, Priorität

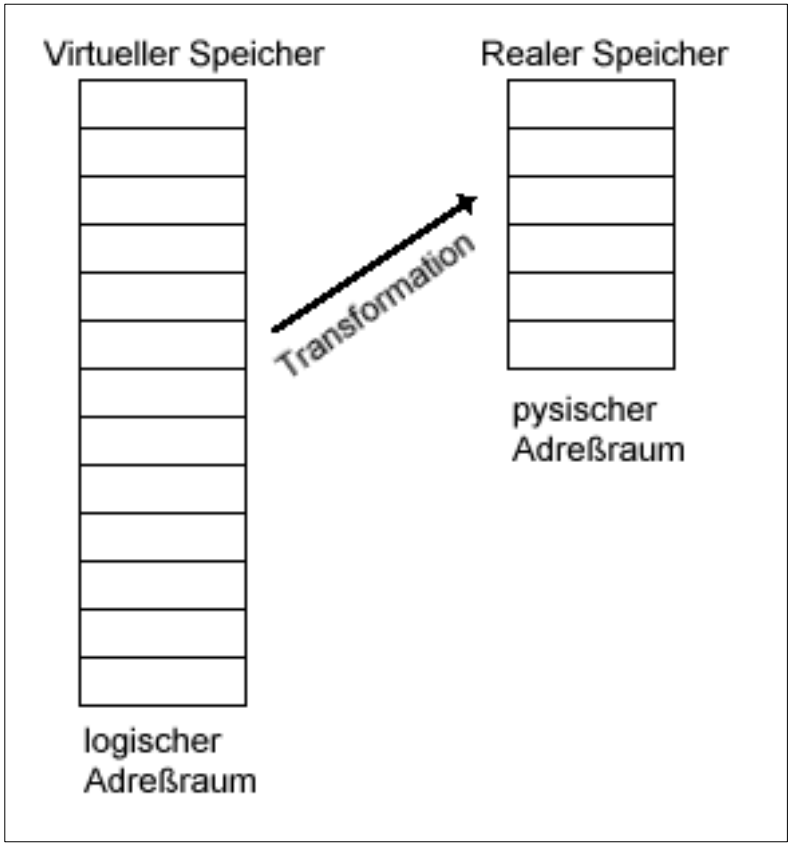
Moderne Prozessoren stellen mehrere CPU-Kerne bereit, über die die Tasks verteilt werden

Ein Problem bei Multitasking: Synchronisation von Prozessen

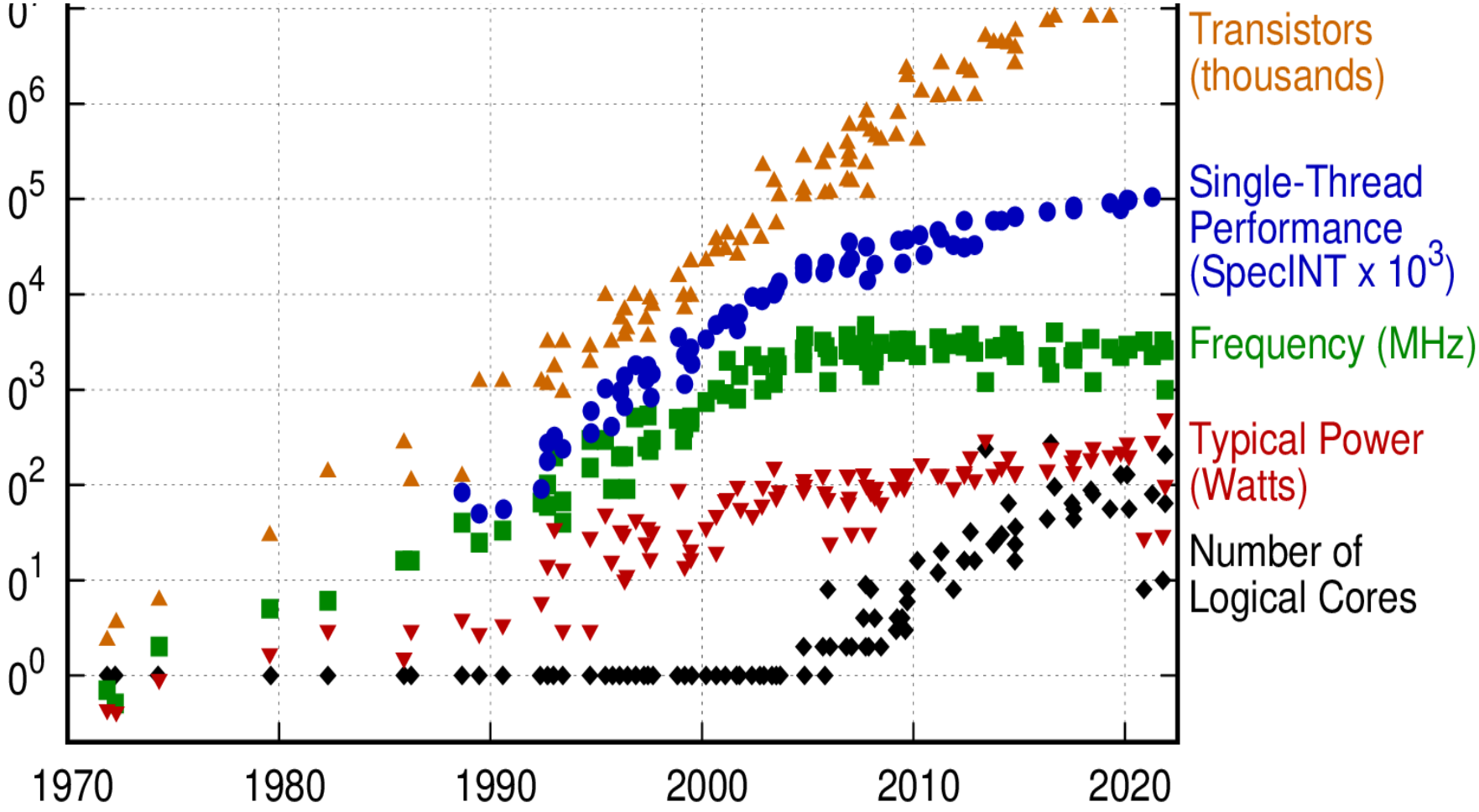
Virtueller Speicher

Virtueller Speicher:

- Großer logischer Adressraum wird auf begrenzten physikalischen Speicher abgebildet
- Verwendung von logischen Adressen im Code
  -> Code ist unabhängig von der Position des Programms im Speicher
- Vergrößerung des Speichers durch Hintergrundspeicher (Festplatte)



noch einmal: das Moore'sche Gesetz



https://github.com/karlrupp/microprocessor-trend-data

Trotz immer noch exponentiell anwachsender Zahl der Transistoren stagnieren Taktrate („Power Wall") und Einzelprozess-Leistung

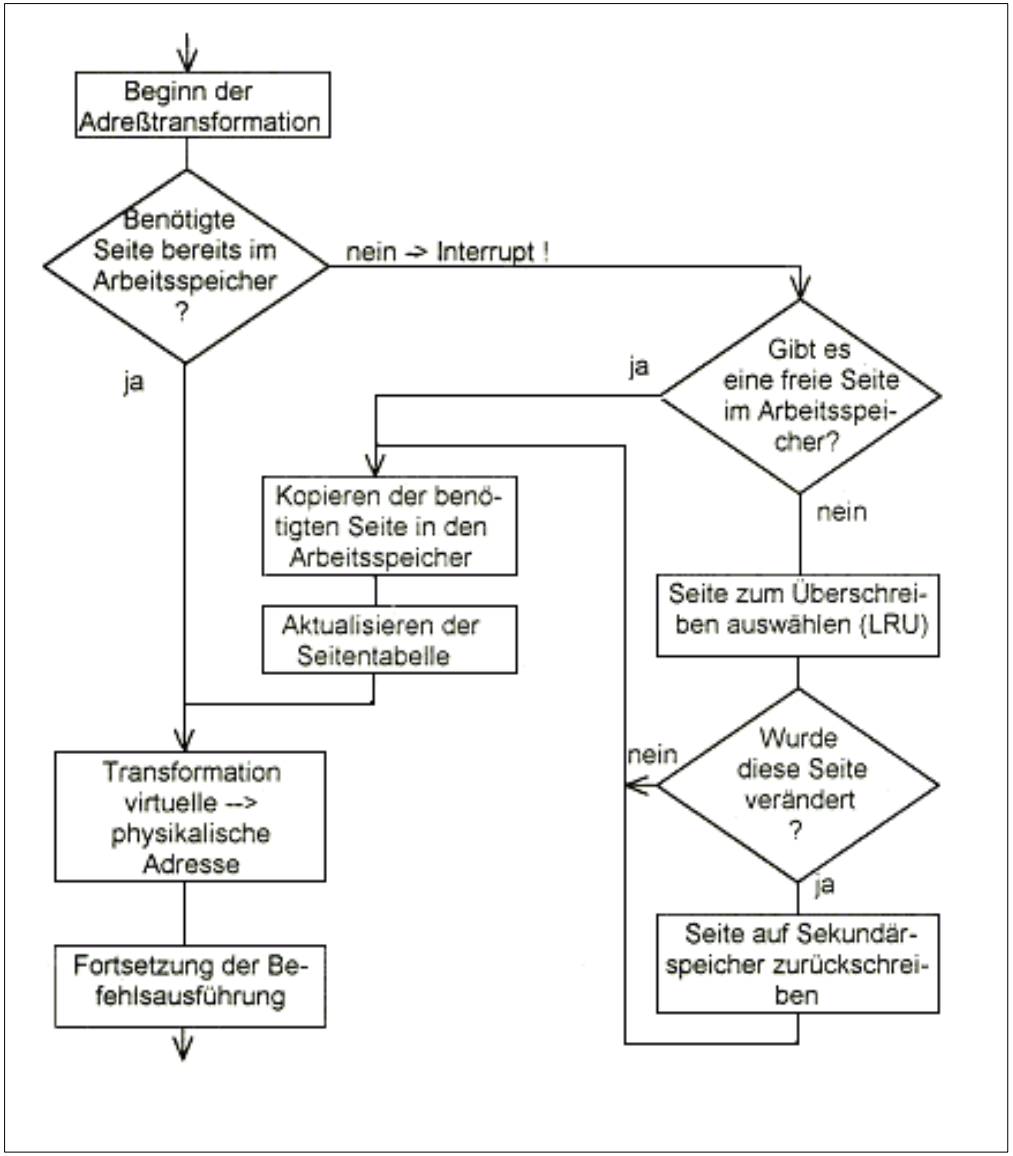
⇒ **Multi-Core und Multi-Prozess-Anwendungen werden immer wichtiger !**

Speicherverwaltung - „Swapping“

CPU (memory management unit, MMU) nimmt Übersetzung von logischer in physikalische Adresse vor:

- Falls Seite nicht im Speicher, wird „Page Fault“-Interrupt ausgelöst und das OS liest die Seite vom Hintergrundspeicher
- OS entscheidet, welche Seite ersetzt wird (least recently used, dirty-bit)

- Schutz vor unbefugten Speicherzugriffen („Protection“-Interrupt)



Datenträgerverwaltung

Dateisystem abstrahiert von Hardwaredetails:
Speichereinheiten auf dem Datenträger (Spur/Sektor) -> Dateien

Meist realisiert als Baumstruktur von Verzeichnissen

Bereitstellung von Datei-Informationen:
- Name
- Typ
- Größe
- Eigentümer
- Zugriffsrechte
- Datum von Erstellung / letzter Änderung / letzter Lesezugriff

Unix/Linux

Linux ist eine Re-Implementierung (Open Source) von Unix
(begonnen Anfang der 90er von Linus Torvalds)

LINUX ist (wie UNIX) ein portables (in C programmiertes), recht einfach aufgebautes Betriebssystem, mit folgenden wesentlichen Eigenschaften:

-Multitasking (Prozess-Scheduling, Prozess-Kommunikation)

- Multiuser (Zugangskontrolle und Abrechnung)

- dialogorientiert

- ein Werkzeugkasten mit viele hundert Dienstprogrammen

- (mehrere) Shell(s) mit mächtiger Script-Sprache;
  Scripte laufen als Unterprozesse des startenden Prozesses

- Dateisystem mit Baum-artiger Struktur, Rechtekontrolle für Datei-Zugriff

- graphische Oberfläche X-Windows (X11)

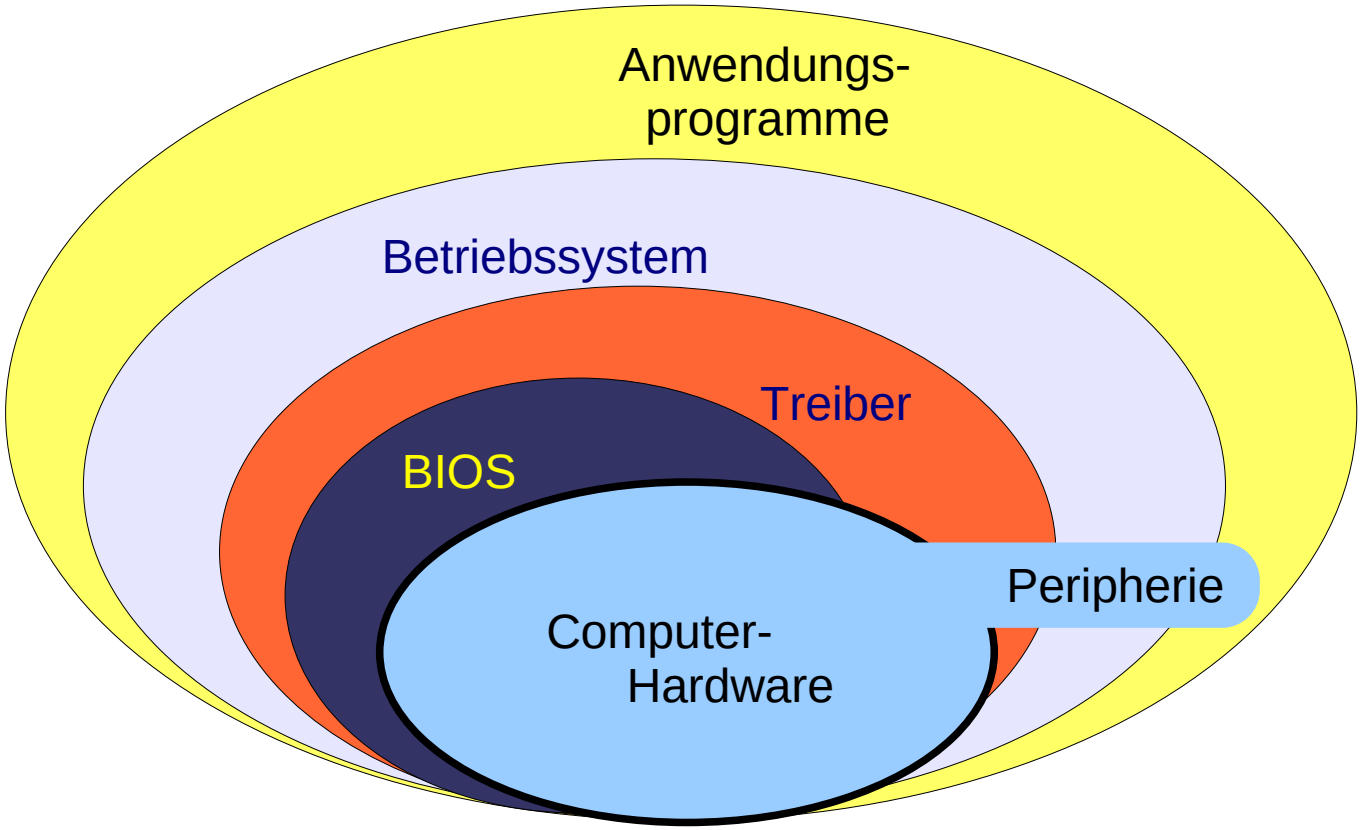
- volle Netzwerkunterstützung, incl. Netzwerk-Laufwerken, remote shells etc.

Linux ist der quasi-Standard im wissenschaftlichen Rechnen

Softwareentwicklung und Werkzeuge

Interaktion mit dem Rechner/Betriebssystem

betrifft alle Schichten: von Direktzugriff auf Hardware bis zu Anwenderprogrammen



Aufgaben von PhysikerInnen bei Datenaufnahme und - Analyse

- Hardwarenahe Programmierung zum Zugriff auf angeschlossene Hardware
- Software zur Simulation und Datenverarbeitung

Softwareentwicklung ?

Warum Software-Entwicklung ?

Wer will/braucht das ?

Antwort: **Jede/r PhysikerIn, also Sie !**

Physikalische Forschung geschieht an der vordersten Front der Erkenntnis:

– d.h. keine „fertigen Programme“ oder „Apps“

– Entwicklung von Experimenten/ Messgeräten/Detektoren erfordert sorgfältige **(numerische) Simulation**

– komplexe Modellbildung in der Theorie benötigt effiziente **numerische Verfahren**

– Moderne Experimente liefern große Datenmengen
→ **Datenauswertung** nur am Computer möglich

– Statistische Verfahren in der Datenanalyse oft aufwändig simulierte **Pseudo-Experimente** im Rechner

Werkzeuge („Tools“)

Wichtige Hilfsmittel für die Softwareentwicklung:

Editoren, z.B:

- vi(m): textbasiert, fast auf jedem Linux system installiert
- pico: textbasiert
- (x)emacs: graphisch, sehr mächtig

kate: graphisch, sehr intuitiv bedienbar

Compiler+Linker

- c++, (gfortran)

bzw. Interpreter:

python, java, (Julia, Rust), ...

Make, Debugger, Profiler:

- make, gdb, gprof, ...

Codeverwaltung:

- git als aktueller Standard

Integrierte Entwicklungsumgebungen (IDE):

KDevelop, PyCharm, Eclipse, kommerzielle Produkte, ...

GNU Compiler g++ (gcc)

- GNU compiler collection
- Compiler und Linker für C++ (und andere Sprachen)
- Verarbeitungsschritte:
  Quellcode
  → Assemblercode
  → Maschinencode
  → ausführbares Programm,
  statische oder dynamische Bibliothek

- Wichtige Optionen:
  -c: kompilieren
  -o: Ausgabedatei
  -I: Pfad für eingebundene Dateien (include)
  -L, -l: Pfad und Name von Bibliotheken
  -shared, -g, -O, -Wall, -fPIC: dynamische Bibliothek, Debugsymbole, Optimierung, Warnungen, positionsunabhängiger Code

s. Vorlesung „Programmieren“ im 4. Semster

make

Zweck:

Organisation der Erzeugung eines Produkts aus Quelldateien

Includes – Quellcode – Bibliotheken

- Ausführen von Befehlen in Abhängigkeit von Änderungen an Dateien:
  Schritt wird ausgeführt, wenn Datei neuer ist als eine davon abhängende
- Insbesondere verwendet für Compiler/Linker
- Dateiname: GNUmakefile, makefile, Makefile (oder angegeben mit -f)
- Aufbau aus Regeln:

```
target: dependency
 <tab>   command
```

- Implizite Abhängigkeiten: z.B. `.cc .o`
- Definition von Variablen:

```
VAR = value
SOURCES = $(wildcard *.cc)
DIR = $(shell /bin/pwd)
```

- Automatische Ermittlung von Abhängigkeiten: `makedepend`

Beispiel - `Makefile`

```
CC=gcc
CFLAGS=-I.
mCcode: myCcode.o func.o
        gcc -o mCcode mCcode.o func.o -I.
```

Beispiel: make

make ist nicht beschänkt auf C-Code,
hier ein Beispiel zur Erzeugung von Dokumentation mit pandoc

Datei Makefile

```
# Makefile for pandoc: output pdf or html

NAME = Datenauswertung
IN = $(NAME).md

htmlOUT = $(NAME).html
pdfOUT = $(NAME).pdf

EXE=pandoc
OPT = -V geometry:margin=2.5cm -V lang=de
htmlOPT = -s --mathjax
pdfOPT = --toc

all: html pdf
html: $(htmlOUT)
pdf: $(pdfOUT)
clean:
	rm $(pdfOUT)
	rm $(htmlOUT)

$(htmlOUT): $(IN)
	$(EXE) $(OPT) $(htmlOPT) -o $(htmlOUT) $(IN)

$(pdfOUT): $(IN)
	$(EXE) $(OPT) $(pdfOPT) -o $(pdfOUT) $(IN)
```

Aufruf:

> make

oder

> make html

oder

> make pdf

Debugger

Ein **Debugger** ermöglicht

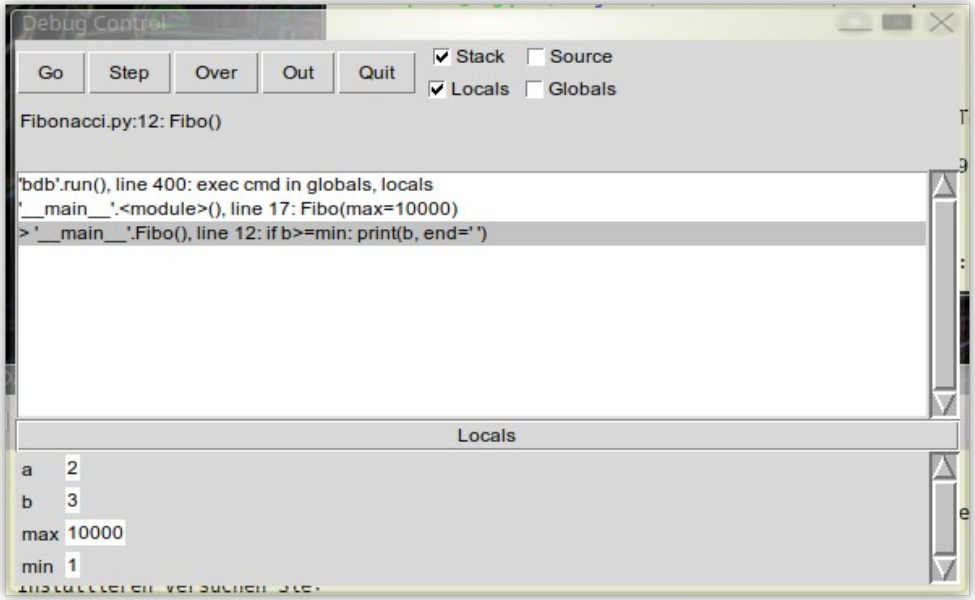
- schrittweises Ausführen von Programmen
- Anzeige der Werte relevanter Variablen

```
def Fibo(min=1, max=1000):
  a, b = 0, 1
  while b < max:
    if b >= min: print(b, end=' ')
    a, b = b, a+b

print('*==* Demo Script: Fibonacci numbers')
print('calling Fibo(max=10000)')
Fibo(max=10000)

print('\n calling Fibo(10000,1000000)')
Fibo(10000,1000000)
print()
```

Beispiele IDLE



Empfehlenswert für Python: **PyCharm**

sehr mächtig, erfordert Einarbeitung !

für compilierten Code: Debugger gdb

https://www.gnu.org/software/gdb/

- Debugger zum Auffinden von Laufzeitfehlern
- Ausgabe der Position in Programm: **backtrace**, **bt**
- Ausgabe von Variablenwerten: **print**, **p**
- Setzen von Haltepunkten: **break**
- Schrittweises Ausführen: **next**, **n**, **step**, **s**
- Programm muss mit Option -g kompiliert sein, um Debug-Symbole zu enthalten
- kann coredump-Dateien lesen

```
#include <iostream>

using namespace std;
int divint(int, int);

int main() {
  int x = 5, y = 2;
  cout << divint(x, y);
  x =3; y = 0;
  cout << divint(x, y);
  return 0;
}
int divint(int a, int b)
{
  return a / b;
 }
```

crash.cc

Fehlerhaftes Program `crash.cc` übersetzen mit `-g` option, dann in gdb ausführen

```
> g++ -g crash.cc -o crash
> gdb crash
```

Profiler

Profiler dienen der Effizienzsteigerung von Programmen:
Auffinden von Programmteilen mit viel CPU-Zeit

- Ausführen bzw. Compilieren und Linken mit Profiler-Unterstützung
- Programmausführung ergibt Datei mit Profil-Information:
  - Zahl der Aufrufe pro Funktion
  - CPU-Zeit pro Aufruf
  - gesamte von einer Funktion verbrauchte CPU-Zeit

einfachste Variante:

> time <befehl> zeigt Laufzeit eines Linux-Befehls an

etwas ausführlicher: Paket perf

> perf record <befehl>

> perf report zeigt Zeit in einzelnen Programm-Teilen an

Python-Paket timeit

```
from timeit import timeit
def fib(n):
    return n if n < 2 else fib(n - 2) + fib(n - 1)
total_time = timeit("(30)", number=100, globals=globals())
```

Python-Paket cProfile

Anzahl Aufrufe pro Funktion,
verbrachte CPU-Zeit

```
import cProfile
from package import <function>
cProfile.run('<function>( <args> )')
```

Profiler

Paket gprof (Teil der GNU binutils) für compilierter Programme

`gprof` ist Teil der GNU binutils:
`https://www.gnu.org/software/binutils/`

`http://valgrind.org`

zum Debuggen und optimieren von Programmen

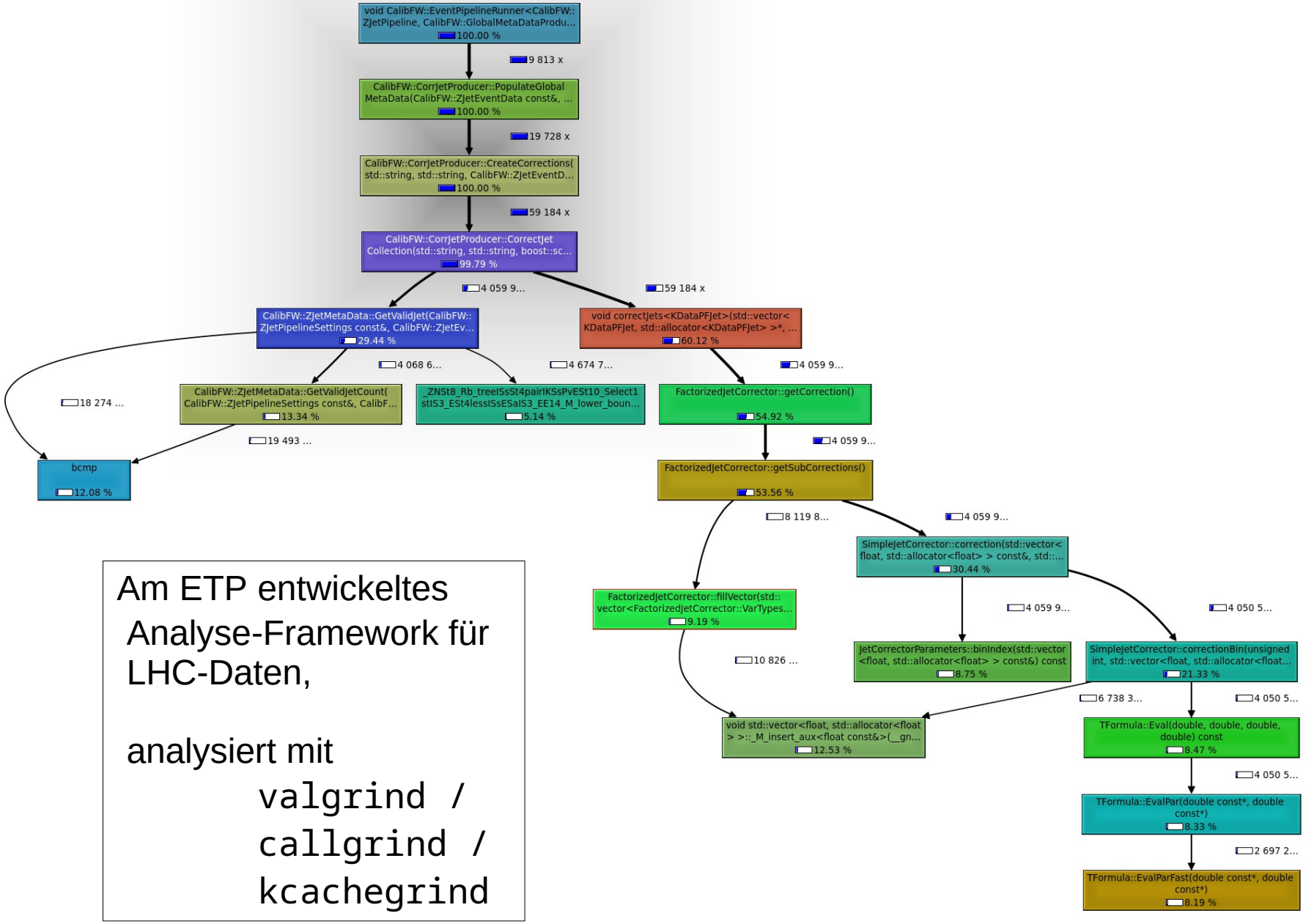
Anwendung:

```
> valgrind  --tool = callgrind myProg
```

grafische Darstellung mit

```
> kcachegrind callgrind.out
```

Beispiel Valgrind



Am ETP entwickeltes Analyse-Framework für LHC-Daten,

analysiert mit

`valgrind / callgrind / kcachegrind`

Versionskontrolle mit GIT

`git-scm.com`

generelle **Vorgehensweise**:

Dateien in zentralem Repository werden
- lokal herunter geladen
- lokal bearbeitet
- nur Änderungen zurückgespeichert

Quellcode Management absolut notwendig, um Software in Teams zu entwickeln

Anm.: `git` *auch nützlich bei der gemeinsamen Erstellung von großen Texten*

Dokumentieren sichert Wiederverwendbarkeit

gute Dokumentation von Software ist das **A und O** für gemeinsam genutzte, wartbare und wiederverwendbare Software

Prinzip: Möglichkeiten für Kommentarzeilen
`#` in Python und Script-Sprachen, `//` bzw. `/* ... */` in C/C++
wird genutzt, um Dokumentar-Text und Strukturinformationen über den Code zu extrahieren (meist in html-Format).

doxygen `http://www.doxygen.org/`

sphinx `http://sphinx-doc.org/` für Python (siehe z.B. „kafe“)

Doxygen: spezielle Kommentarblocks übernehmen
kommentar in die Dokumentation: z.B. C/C++

```
/**  kommentar

 */
```

Beispiel zu SPHINX

Python Documentation Generator

Beispiel einer Funktion aus PhyPraKit

```
def wmean(x, sx, V=None, pr=True):
    """weighted mean of np-array x with uncertainties sx
    or covariance matrix V; if both are given, sx**2 is
added
    to the diagonal elements of the covariance matrix

    Args:
      * x: np-array of values
      * sx: np-array uncertainties
      * V: optional, covariance matrix of x
      * pr: if True, print result

    Returns:
      * float: mean, sigma
    """

# begin of python code

    ...
```

mit SPHINX erzeugte Dokumentation

PhyPraKit.**wmean**(*x*, *sx*, *V=None*, *pr=True*)

weighted mean of np-array x with uncertainties sx or covariance matrix V; if both are given, sx**2 is added to the diagonal elements of the covariance matrix

Args:

- x: np-array of values
- sx: np-array uncertainties
- V: optional, covariance matrix of x
- pr: if True, print result

Returns:

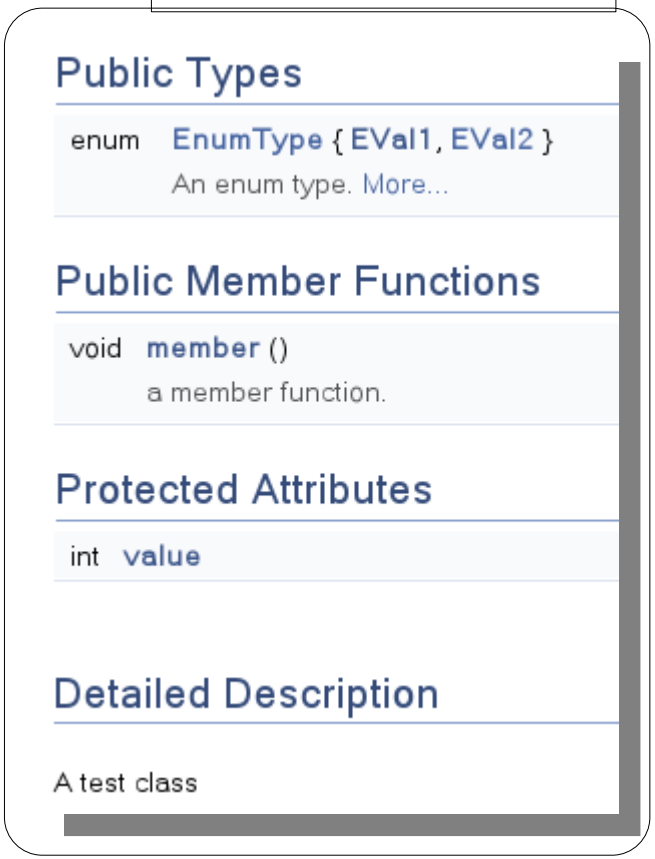
- float: mean, sigma

Beispiel zu Doxygen

Beispielcode

```
/*! A test class */
 class Test
{
   public:
       /** An enum type.
        * The documentation block cannot be put after the enum!
        */
      enum EnumType
       {
          int EVal1, /**< enum value 1 */
          int EVal2 /**< enum value 2 */
      };
      void member(); //!< a member function.
   protected:
      int value; /*!< an integer value */
};
```

von Doxygen erzeugte html-Dokumentation



Public Types

enum EnumType { EVal1, EVal2 }

An enum type. More...

Public Member Functions

void member ()

a member function.

Protected Attributes

int value

Detailed Description

A test class

Entwicklungsumgebungen (IDE)

= **I**ntegratded **D**evelopment **E**nvironment

Nützliches **Werkzeug für Softwareprojekte**

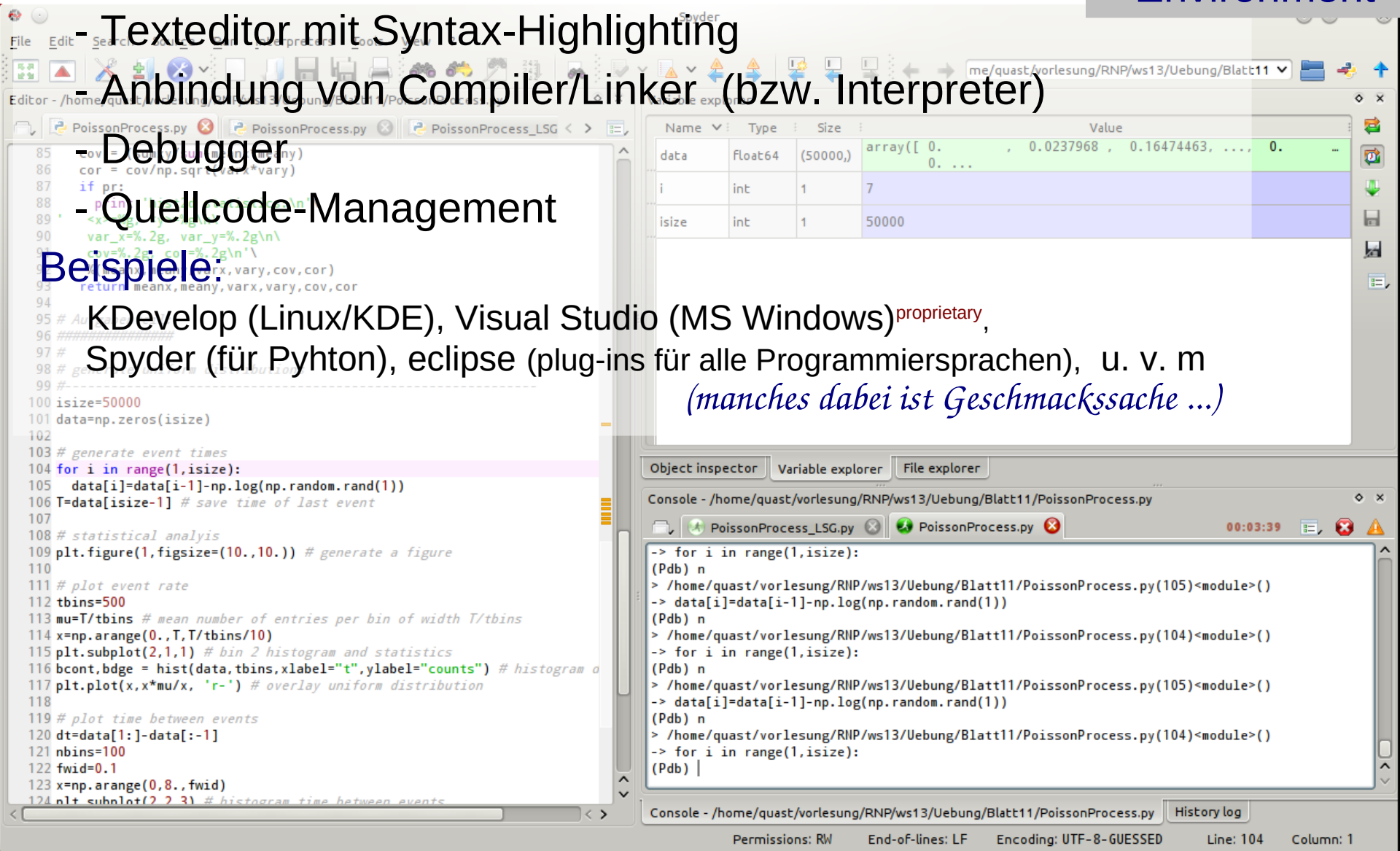
typischerweise enthalten:

- Texteditor mit Syntax-Highlighting
- Anbindung von Compiler/Linker (bzw. Interpreter)
- Debugger
- Quellcode-Management

Beispiele:

KDevelop (Linux/KDE), Visual Studio (MS Windows)[proprietary],
Spyder (für Pyhton), eclipse (plug-ins für alle Programmiersprachen), u. v. m

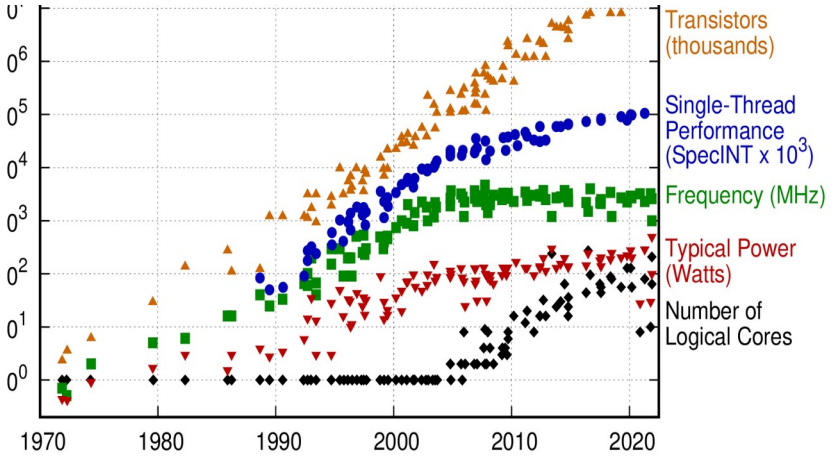
(manches dabei ist Geschmackssache ...)



Aktuelle Herausforderungen

Aktuelle Herausforderungen

Technologische Entwicklung:



Trotz noch exponentiell anwachsender Zahl der Transistoren stagnieren Taktrate („Power Wall“) und Einzelprozess-Leistung

Kleinere Strukturen

– führten früher zu Steigerung der Geschwindigkeit

seit ca. 2005: Grenze der energetisch vertretbaren Taktraten erreicht

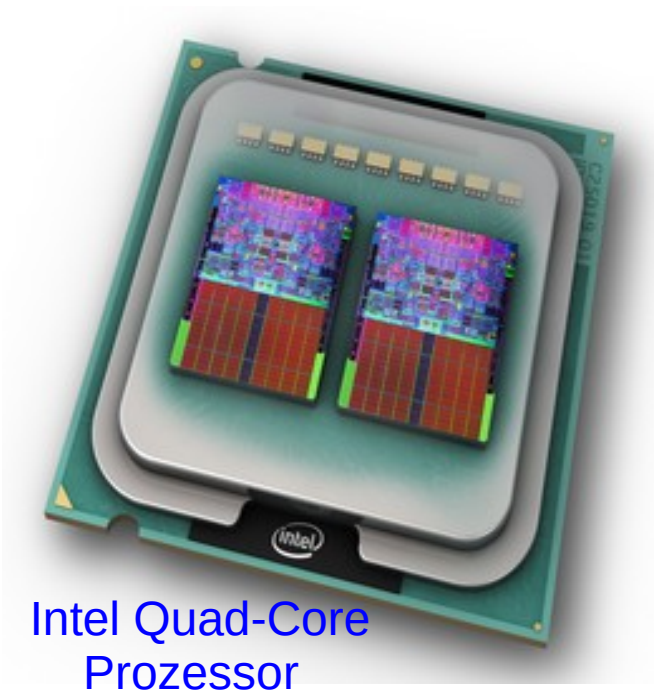
→ Steigerung der Komplexität der CPUs:

- Parallelisierung
- Grafikeinheiten

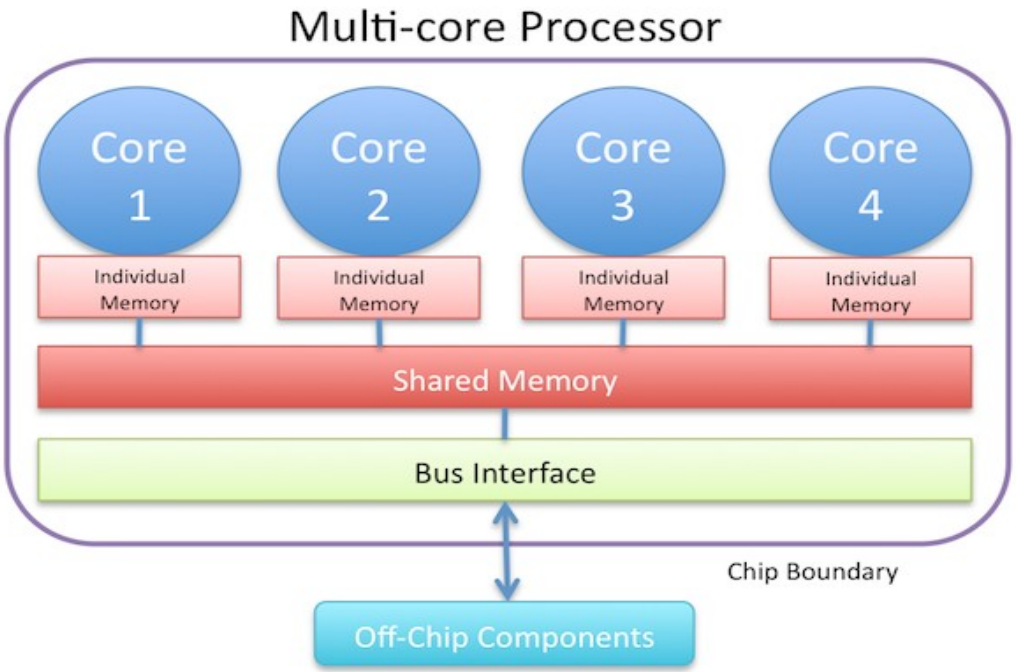
→ Multi-Core-Architekturen

Herausforderung für die Software-Entwicklung

Multi-Core-Prozessoren



Intel Quad-Core Prozessor



- Zahl der Cores pro Chip steigt weiter an (z Zt. 12-cores in Server-CPUs)
- Vektoreinheiten im Prozessor: SIMD (Single Instruction Multiple Data)
- Chips mit integriertem Grafik-Prozessor
- aus Gründen der Energieeffizienz kaum Steigerung der Taktraten
- Datentransfer zum Speicher wird immer mehr zum Flaschenhals

Entwicklungen der letzten Zeit

mehr Transitoren/Chip →

– seit 2005 keine signifikante Steigerung der Taktraten

„free Lunch is over“

– komplexere Prozessor-Architekturen mit

- mehr/größeren Registern
  16bit (8086,), 32bit (80386), 64bit (AMD Athlon 64, x86-64),
  128bit (SSE), 256bit (AVX), 512bit (AVX-512)
- größere Cache-Speicher
- Vektorisierung („Single Instruction Multiple Data“, **SIMD**)
  z..B. MMX, SSE 1//2/3/4/5, AVX
- Multi-Core-Architekturen mit 2/4/6/8 … Kernen pro CPU
  auch „Hypterthreading“ bei Intel-Architektur
- Piplining (d.h. parallele Ausführung) von Instruktionen
- Sprung-Vorhersage
- integrierte Grafik-Einheiten (GPU)

Zunehmende Parallelisierung und Heterogenität der Architekturen
Herausforderung für die Entwicklung effizienter Programmpakete

Herausforderungen auf Software-Seite

- Code auf Vektorisierungs-Möglichkeiten hin optimieren

  Ausnutzung von „Autovektorisierung“ im C++ -Compiler

  Beispiel:

  Arrays of Pointers ↔ Pointers to Arrays

  *„**Data-oriented**“-Programming statt **Object-oriented um jeden Preis***

- Multi-Threading:

  parallele Ausführung von (Teilen von) Algorithmen,

  die auf den gleichen Datenstrukturen arbeiten, aber „thread-save“ sind

  C++ unterstützt dies seit Vers. 11 (ISO-Standard seit Aug. 2011)

- GPU-Nutzung

  aufwändige, problemabhängige Optimierung von Algorithmen

  Implementierung unterstützt durch Frameworks wie

  - CUDA (NVIDIA) oder
  - OpenCL = Schnittstelle für uneinheitliche Parallelrechner
    Anpassung an Hardware erfolgt beim Übersetzen

Parallelisierung

Aufgabe aus der Schulmathematik:

Ein Maurer baut eine Mauer in fünf Stunden.

Wie lange brauchen fünf Mauer?

Richtige Antwort lt. Schulbuch:

Fünf Maurer bauen die Mauer in einer Stunde.

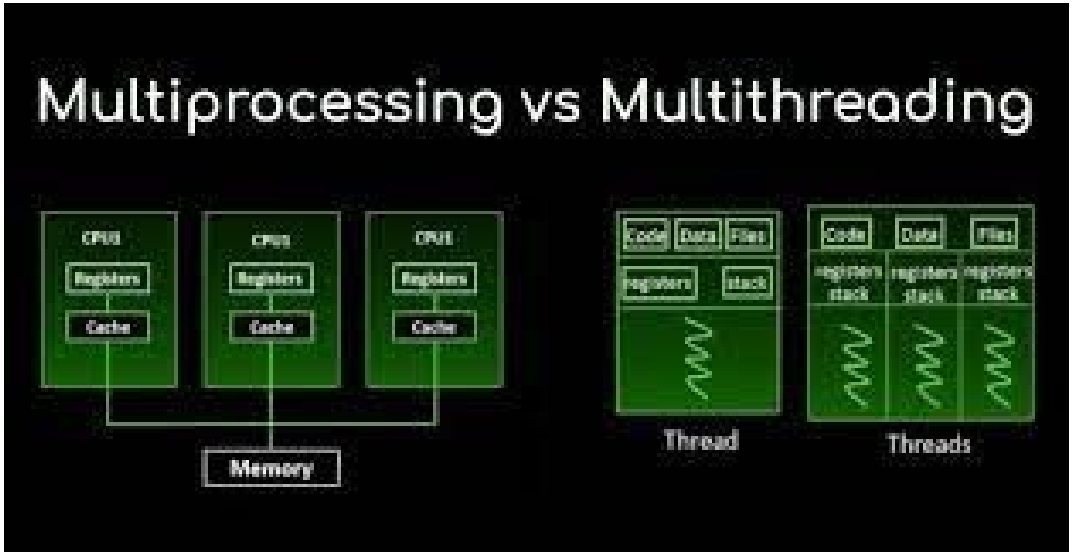
Stimmt das ?

Parallelisierung

Ähnliches **Problem beim Computing**:

Ein Prozess erledigt eine Aufgabe in 4.4 s. Wie langen brauchen *n* Prozesse ?

Antwort: n Prozesse erledigen die Aufgabe in 4.4s / *n* *???*



Parallelisierung

Ähnliches **Problem beim Computing**:

Ein Prozess erledigt eine Aufgabe in 4.4 s. Wie langen brauchen *n* Prozesse ?

Antwort: n Prozesse erledigen die Aufgabe in 4.4s / *n* *???*

Probieren wir's aus:

```
> source set_num_threads.sh 1
> time test.py
  ... <output von test.py>
real   0m 4.403s
user   0m 3.011s
sys    0m 0.152s
>
> source set_num_threads.sh 12
> time test.py
  ... <output von test
real   0m 4.48s
user   0m 5.03s
sys    0m 8.804
```

gleiche Zeitdauer, aber
4.4-facher Aufwand an CPU !

Parallelisierung

Ähnliches **Problem beim Computing**:

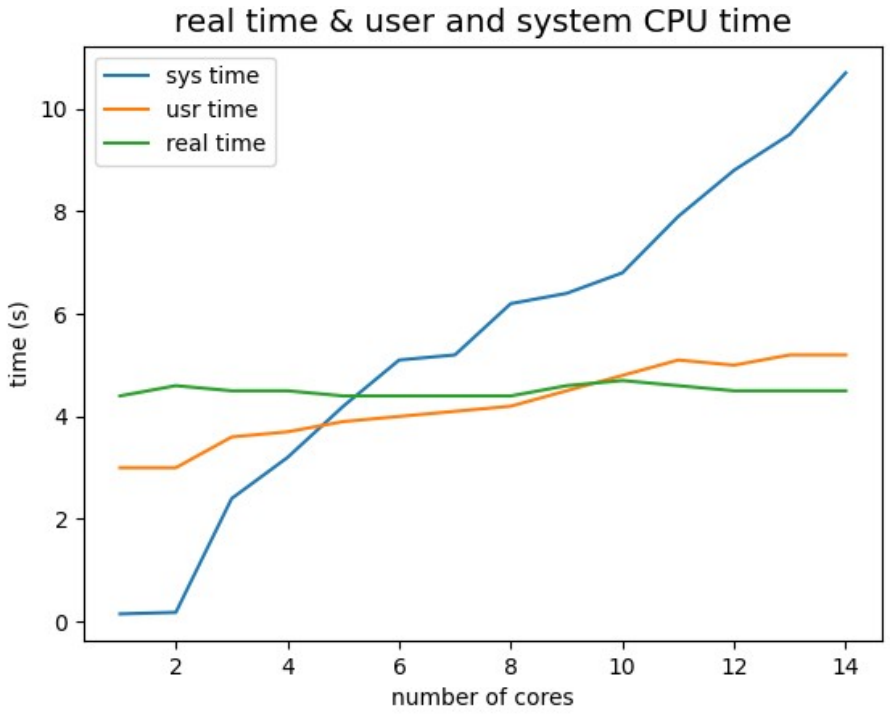
Ein Prozess erledigt eine Aufgabe in 4.4 s. Wie langen brauchen *n* Prozesse ?

Antwort: n Prozesse erledigen die Aufgabe in 4.4s / *n* ???

Probieren wir's aus:

```
> source set_num_threads.sh 1
> time test.py
  ... <output von test.py>
real   0m 4.403s
user   0m 3.011s
sys    0m 0.152s
>
> source set_num_threads.sh 12
> time test.py
  ... <output von test
real   0m 4.48s
user   0m 5.03s
sys    0m 8.804
```

gleiche Zeitdauer, aber
4.4-facher Aufwand an CPU !



Parallelisierung

Ähnliches **Problem beim Computing**:

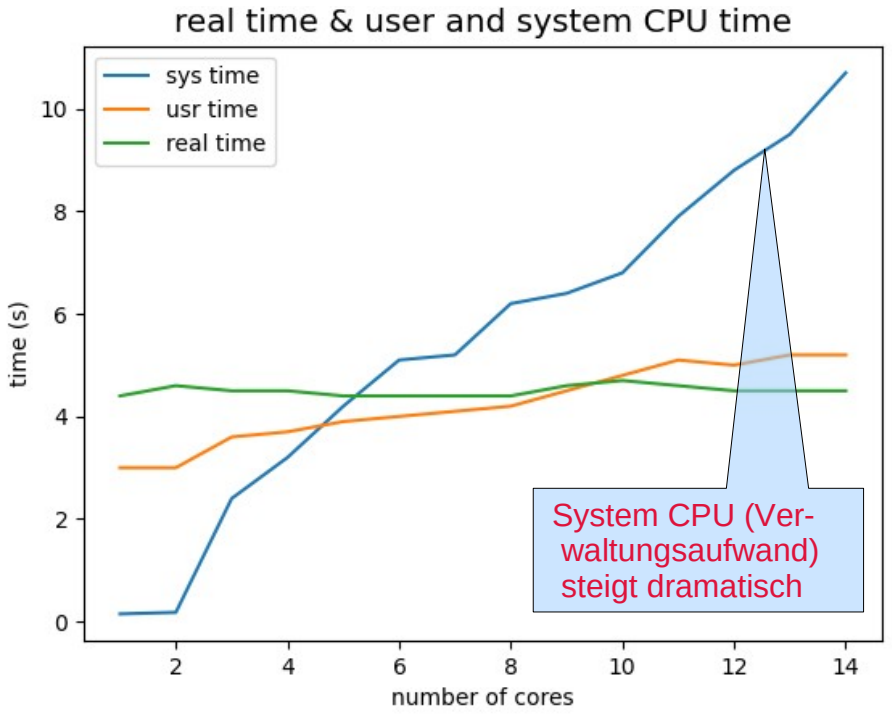
Ein Prozess erledigt eine Aufgabe in 4.4 s. Wie langen brauchen *n* Prozesse ?

Antwort: n Prozesse erledigen die Aufgabe in 4.4s / *n* ???

Probieren wir's aus:

```
> source set_num_threads.sh 1
> time test.py
  ... <output von test.py>
real   0m 4.403s
user   0m 3.011s
sys    0m 0.152s
>
> source set_num_threads.sh 12
> time test.py
  ... <output von test
real   0m 4.48s
user   0m 5.03s
sys    0m 8.804
```

gleiche Zeitdauer, aber
4.4-facher Aufwand an CPU !



Nicht untypisch:

Schlecht gemachte Parallelisierung verbessert nichts,
aber verschwendet Ressourcen !

Auch schlecht im Sinne der Nachhaltigkeit !

Parallelisierung

Ähnliches **Problem beim Computing**:

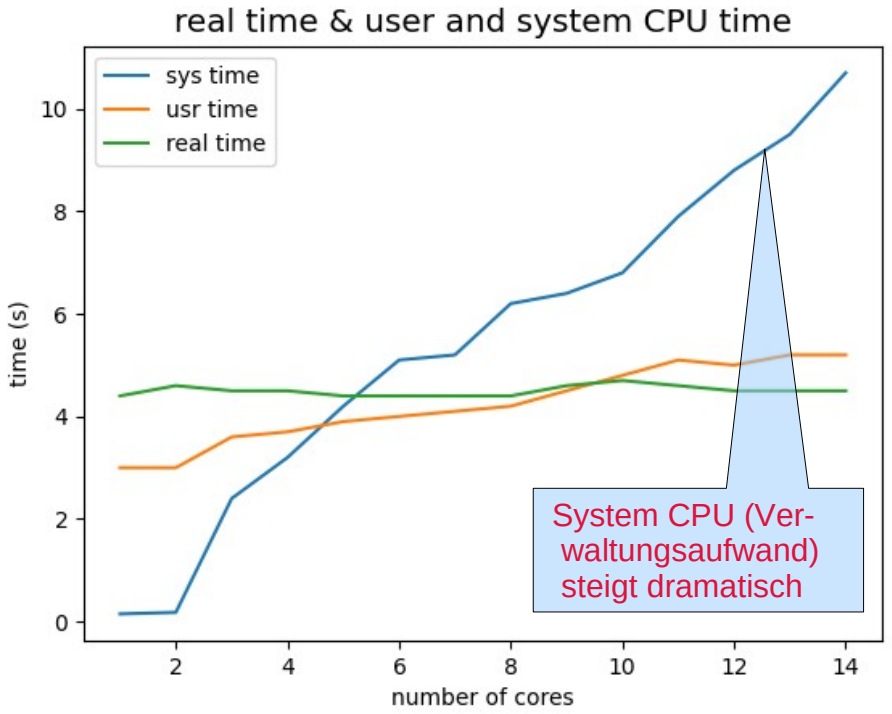
Ein Prozess erledigt eine Aufgabe in 4.4 s. Wie langen brauchen *n* Prozesse ?

Antwort: n Prozesse erledigen die Aufgabe in 4.4s / *n* ???

Probieren wir's aus:

```
> source set_num_threads.sh 1
> time test.py
  ... <output von test.py>
real   0m 4.403s
user   0m 3.011s
sys    0m 0.152s
>
> source set_num_threads.sh 12
> time test.py
  ... <output von test
real   0m 4.48s
user   0m 5.03s
sys    0m 8.804
```

gleiche Zeitdauer, aber
4.4-facher Aufwand an CPU !



Nicht untypisch:

Schlecht gemachte Parallelisierung verbessert nichts,
aber verschwendet Ressourcen ! Auch schlecht im Sinne der Nachhaltigkeit !

Achtung: Wenn man nichts unternimmt (z.B. source script set_num_threads.sh 3)
greift sich numpy alle vorhandenen Kerne – meistens ist das ganz schlecht !!!!

Parallelprozessierung in Python

betrachten zwei (extreme) Funktionen:

1. durch Ein-/Ausgabe limitiert
2. durch Rechenaufwand limitiert

```
def io_bound(sec):
    ...
def cpu_bound(count):
    ...
```

s. Notebook
parallel_python.ipynb

multithreading

abwechselnde Ausführung der
Threads auf einem Prozessor-Kern

```
from threading import Thread
start_time = time.time()
t1 = Thread(target = io_bound, args =(SLEEP, ))
t2 = Thread(target = io_bound, args =(SLEEP, ))
t1.start()
t2.start()
t1.join()
t2.join()
print("time taken in s", time.time() - start_time)
```

multiprocessing

gleichzeitige Ausführung der Prozesse
auf je einem Prozessor-Kern
(falls vorhanden)

```
from multiprocessing import Process
start_time = time.time()
p1 = Process(target = cpu_bound, args =(COUNT, ))
p2 = Process(target = cpu_bound, args =(COUNT, ))
p1.start()
p2.start()
p1.join()
p2.join()
print("time taken in s", time.time() - start_time)
```

Parallelprozessierung in Python (2)

Multithreading ist gut, wenn genügend CPU verfügbar ist

- Wechsel zwischen den verschiedenen Threads vom Python-Interpreter übernommen, der als einzelner Prozess auf einem CPU-Kern läuft
- Threads können CPU konstant nutzen, während andere Threads warten
- Variablen im NameSpace des aufrufenden Prozesses in allen Threads verfügbar

Mutiprocessing ermöglicht CPU-Nutzung auf allen verfügbaren Kernen

- Task-Switching vom Betriebssystem durch Erzeugung von Sub-Prozessen
- Prozesse nutzen jeweils einen Kern; wenn alle Kerne genutzt werden, erfolgt die CPU-Zuweisung durch den Task-Scheduler des Betriebssystems
- Python-Umgebung und Namespace werden beim Start eines Prozesses kopiert, können aber nicht dynamisch aktualisiert werden
- Messaging - Methoden (Queue, Pipe) müssen verwendet werden, um Daten zwischen Prozessen zu übertragen
- Gemeinsame Speicherbereiche (shared memory) ebenfalls vorhanden, um allen Prozessen Zugang zu gemeinsamem Speicher zu geben
- Initialisierung eines Prozesses ist ressourcen-intensiver als die eines

Manche Ressourcen (wie Hardware oder gemeinsamer Speicher) erfordern den exklusiven Zugriff durch nur einen Thread oder Prozess. Dies wird durch eine Lock-Methode erreicht, die sowohl von den Multiprocessing- als auch Multithreading-Paketen bereitgestellt wird.

Parallelprozessierung in Python (3)

Beispiel mit **Lock** für thread-safe Drucken:

Um vermischte Ausgaben zu vermeiden, muss muss ein Prozess eine Sperre (Lock) anfordern, bevor er auf die Ressource zugreift.

Diese Sperre wird nur gewährt, wenn kein anderer Thread oder Prozess die Sperre hält.

Nach Erledigung der so gegen Verwendung durch andere geschützten Aktivität Sperre wieder freigeben

```
from multiprocessing import
Lock
lock = Lock

def threadsafe_print(text):
    lock.acquire()
    print(text)
    lock.release()
```

Achtung: Nicht versuchen, einen Sperrmechanismus selbst zu implementieren!

Das Sperren beruht auf einer speziellen Anweisung auf Maschinensprachebene ("test_and_set"), die den alten Wert einer Speicherstelle zurückgibt und ihn gleichzeitig auf True setzt.
Diese sogenannte "atomare" Anweisung kann durch den Scheduler des Betriebssystems unterbrochen werden und ist daher "thread-safe".

Thread-sichere Methoden für den Zugriff auf die oben erwähnten Queues, Pipes und Shared-Memory-Bereiche existieren und müssen in Anwendungen verwendet werden, um die Thread-Sicherheit Ihrer Programme zu gewährleisten.

Im Zweifelsfall sollten Sie die oben beschriebene Lock-Methode verwenden; dabei aber beachten, dass Locks zu Lasten der Effizienz gehen können

Beispiel: multiprocessing in python

Skript `test_mp.py`

Grundprinzip zur Verteilung von Aufgaben auf mehrere Prozessor-Kerne in python:

Paket `multiprocessing`

– Start der jeweils gleichen python-Funktion als einzelne Sub-Prozesse

– Datenaustausch über eine FIFO-Queue

Anm.: es gibt auch das Paket `threading`:

Unterprozesse (=threads) im gleichen python-Interpreter

globale Variablen in allen `threads` sichtbar, aber auf einen Kern beschränkt

Anm. 2: für volle Flexibilität empfiehlt sich C++; sehr sehr gute Unterstützung f. „concurrent programming“

```
import multiprocessing as mp

def worker(id, input, Q):
  '''a simple worker process
     Args:
       * int id            identification number
       * float input       input
       * mp.Queue          multiprocessing Queue
  '''
  print ('worker %i started'%id, time.strftime('%y/%m/%d-%H:%M:%S') )
  # code to be executed on worker here -->
  result = calcF(input)
  # send result with worker id to main process
  Q.put( (id, result) )
  print ('worker %i finished'%id, time.strftime('%y/%m/%d-%H:%M:%S') )

def calcF(input):
  # …
  return results

#   this line protects rest of the script from execution in sub-processes
if __name__ == "__main__": # - - - - - - - - - - - - - - - - - - - - - - -
# set up a Queue for data transfer (results from workers)
  nworkers = 4
  Q =  mp.Queue(nworkers)
# define and start worker processes
  input = np.random.rand(nworkers)
  procs=[]
  for i in range(nworkers):
    name='worker %i'%(i)
    procs.append(mp.Process(name=name,target=worker,args=(i, input[i], Q)))

# start processes and wait for them to finish
  for prc in procs:
    prc.start()
    prc.join()
```

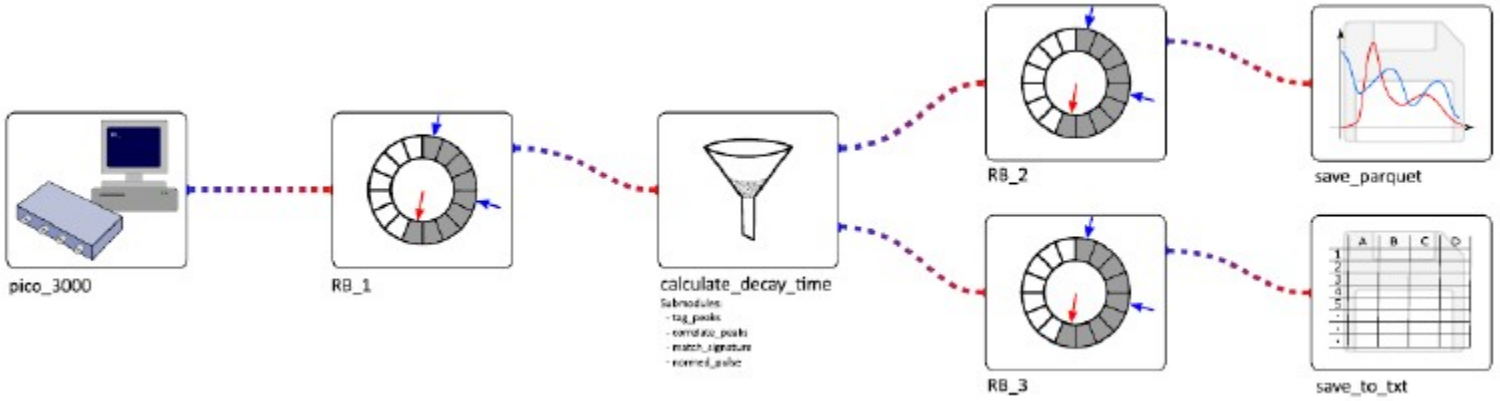
Echtzeit-Datennahme

Problem: Poisson-Ereignisse treffen zufällig ein und
müssen mit konstanter Rechenleistung verarbeitet werden

Methode:

- Schnelle Speicherung in einem Ring-Puffer durch parallelen Prozess;
- Verarbeitung, d.h.
  - Prozessierung,
  - Filterung,
  - graphische Darstellung,
  - Abspeicherung

  durch weitere parallele Prozesse.

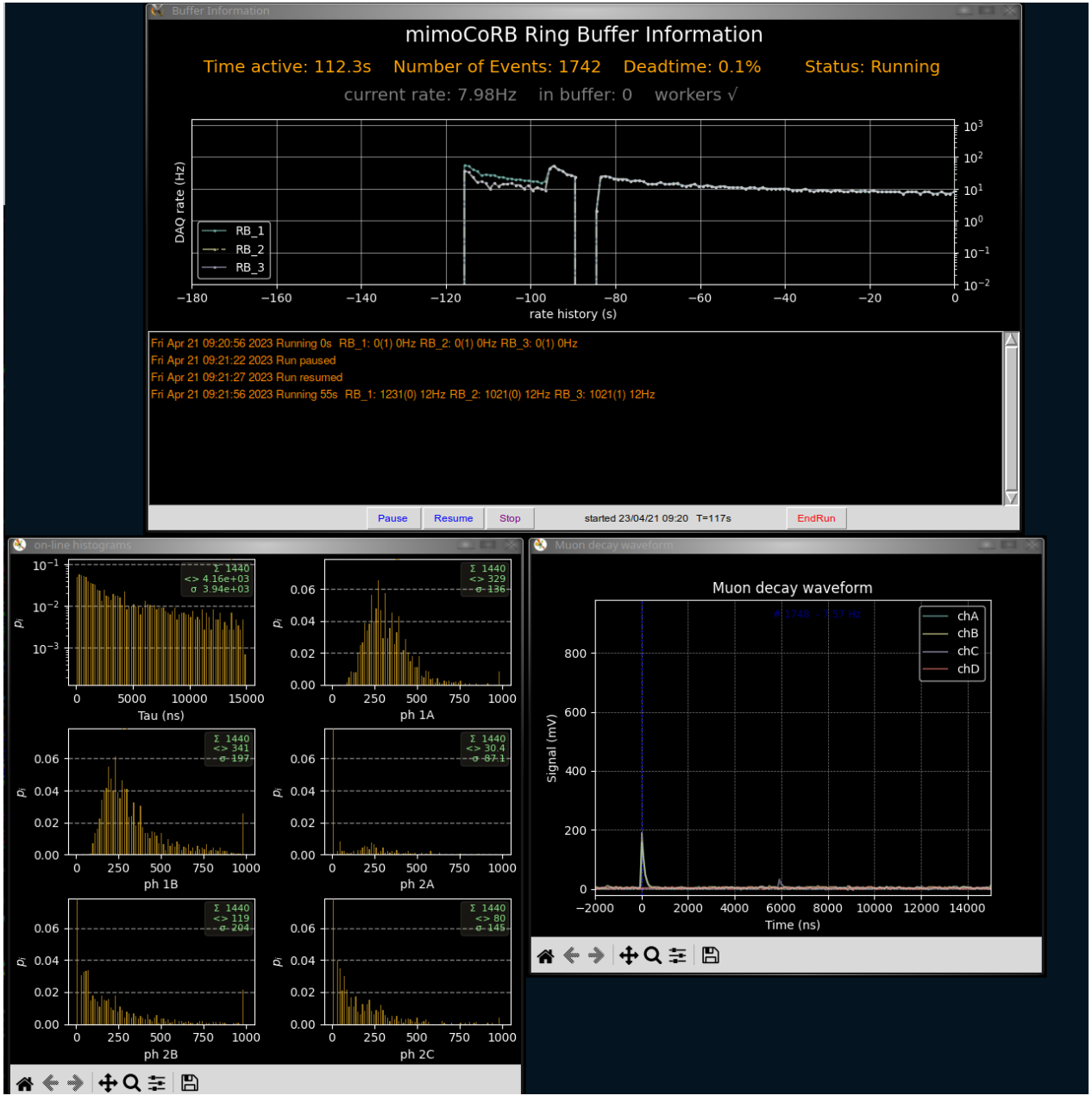


Paket mimoCoRB (**m**ultiple-**i**n **m**ultiple-**o**ut **Co**nfiguragle **R**ing **B**uffer) im F-Praktikum,

https://github.com/GuenterQuast/mimoCoRB

screenshot: mimoCoRB

~/git/mimoCoRB/examples$ ../run_daq.py simulsource_setup.yaml



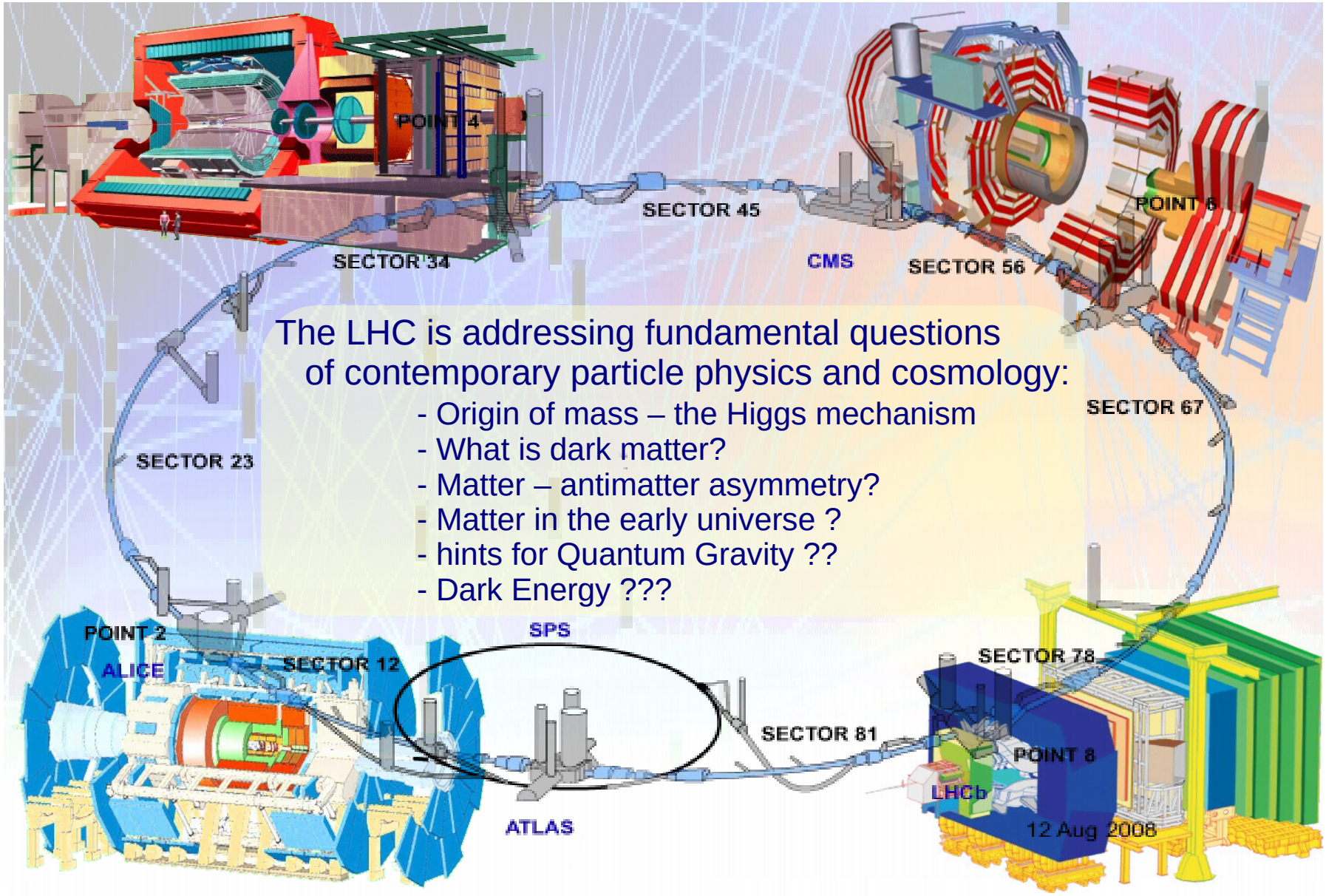
Entwicklung wissenschaftlicher Software

Programmentwicklung im Wissenschaftlichen Bereich ist viel mehr als „Hobby-Programmierung“

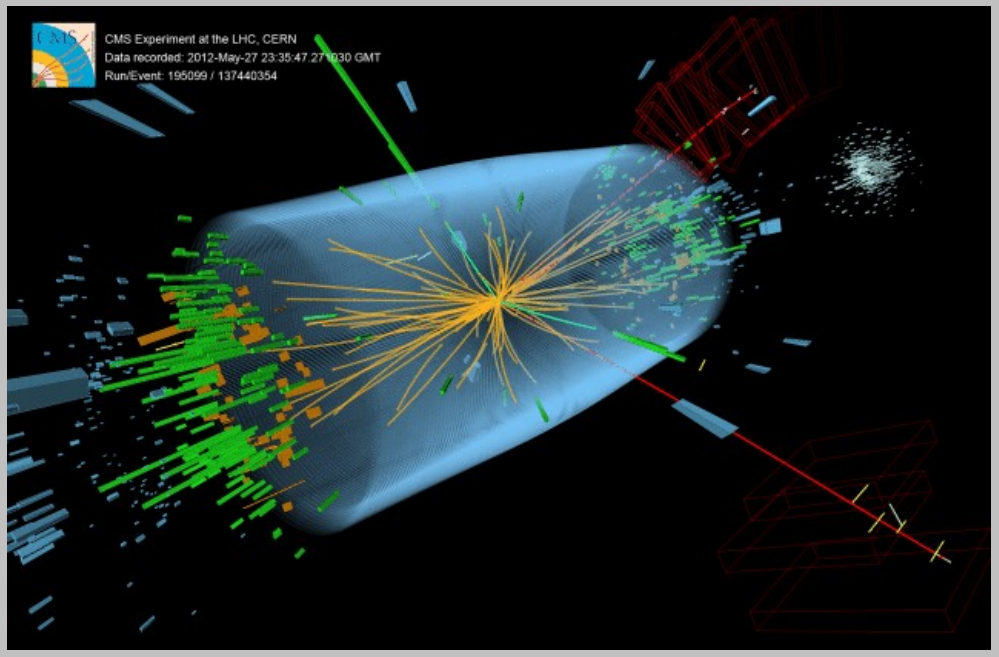
- Erfordert Spezialisierung wie in anderen Bereichen
  (Hardware-Entwicklung, Datenanalyse, Theoretische Physik, …)
- Bei Interesse unbedingt weiterführende Angebote nutzen:
  - Collaborative Software Design (HOC Kurs v. Mitarbeitern ETP)
  - Master-Kurs „Moderne Methoden der Datenanalyse“
  - Nebenfach aus dem Bereich Informatik im Master
  - „privates Engagement“ in kleinen Software-Projekten
  - Hiwi-Jobs

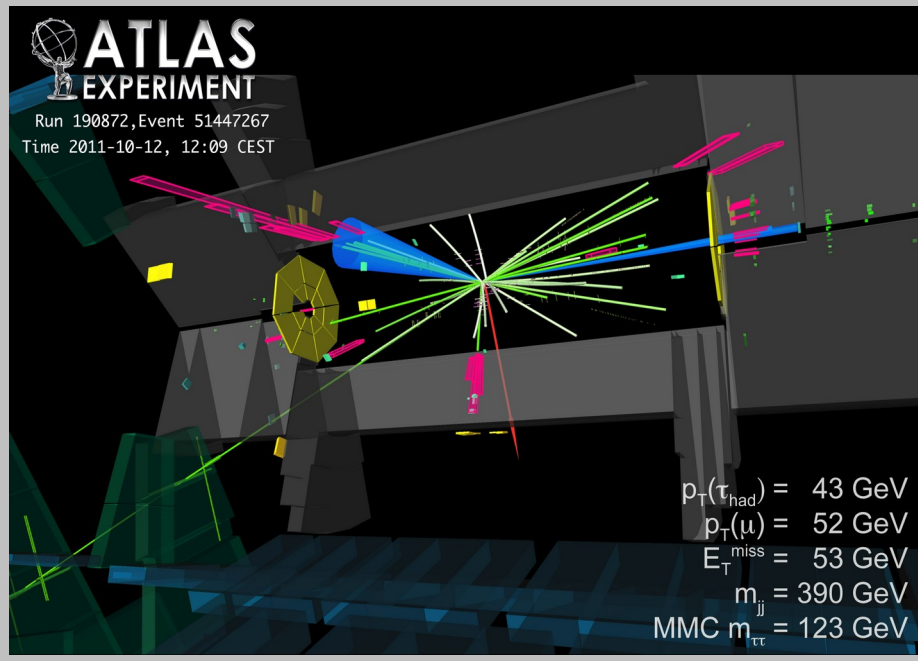
Software und Computing
in
großen Experimenten

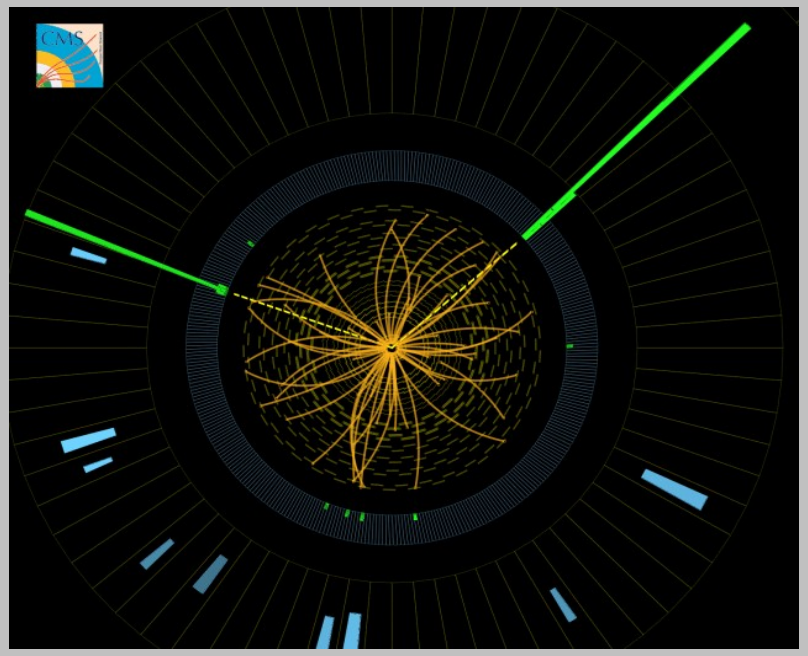
Datenquelle Large Hadron Collider LHC am CERN

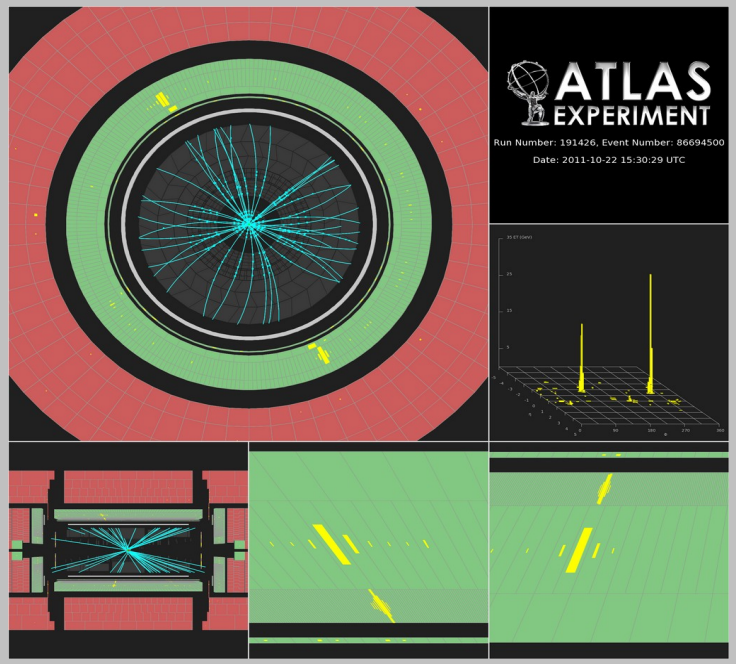


Das Ergebnis: Milliarden Ereignisse/Jahr





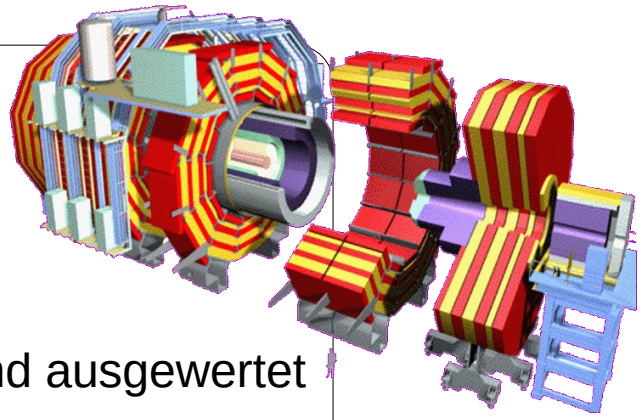




Große Experimente der Physik

Moderne Experimente der Physik

- sind internationale Großprojekte
- sind enorme „Datenquellen“
- werden von einer große Zahl beteiligter Physiker geplant, gebaut, betrieben und ausgewertet
- müssen gleichberechtigten Zugang aller Beteiligten zu benötigten Daten gewähren



Beispiel:
der CMS-Detektor am Large Hadron Collider

Herausforderungen an Computing und Software

– Detektoren sind einzigartige „Prototypen“
→ Erstellung von Datenauslese- und Rekonstruktionssoftware

– Management der Software-Beiträge von Hunderten (oft nicht „professionellen“) Entwicklern (=Physikern) *(einige Millionen Code-Zeilen)*

– Daten- und Workflow-Management auf großen, verteilten Systemen
(z. Zt. ~150 Rechenzentren mit ca. 300'000 CPU-Kernen und 200'000 TB Plattenspeicher)

– Nutzerzugang für Tausende von Physikern sicher stellen

– Anpassungen an Änderungen der Computer-Hardware in den Rechenzentren

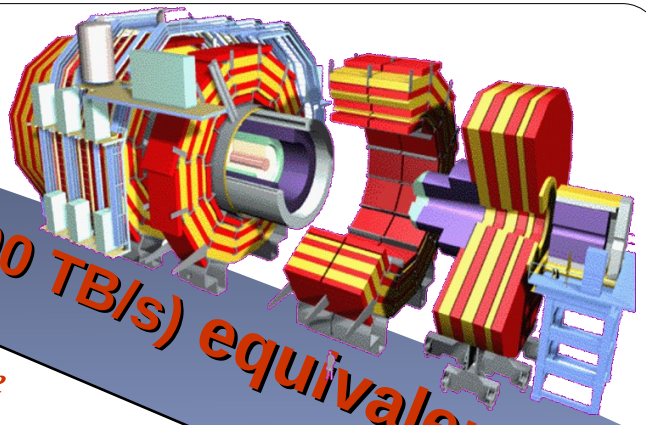
On-line Datenreduktion

- ~ 100 Millionen Detektorzellen
- LHC Kollisionsrate: 40 MHz
- 10-12 bits/Zelle

→ **~1000 Tbyte/s Rodaten**

Nullunterdrückung & **Trigger** reduzieren dies auf
„nur" einige 100 Mbyte/s

i.e. 1[illegible]/sec



40 MHz (~1000 TB/s) equivalent

Level 1 - Hardware

100 Khz (~100 GB/s digitized)

Level 2 – Online Farm

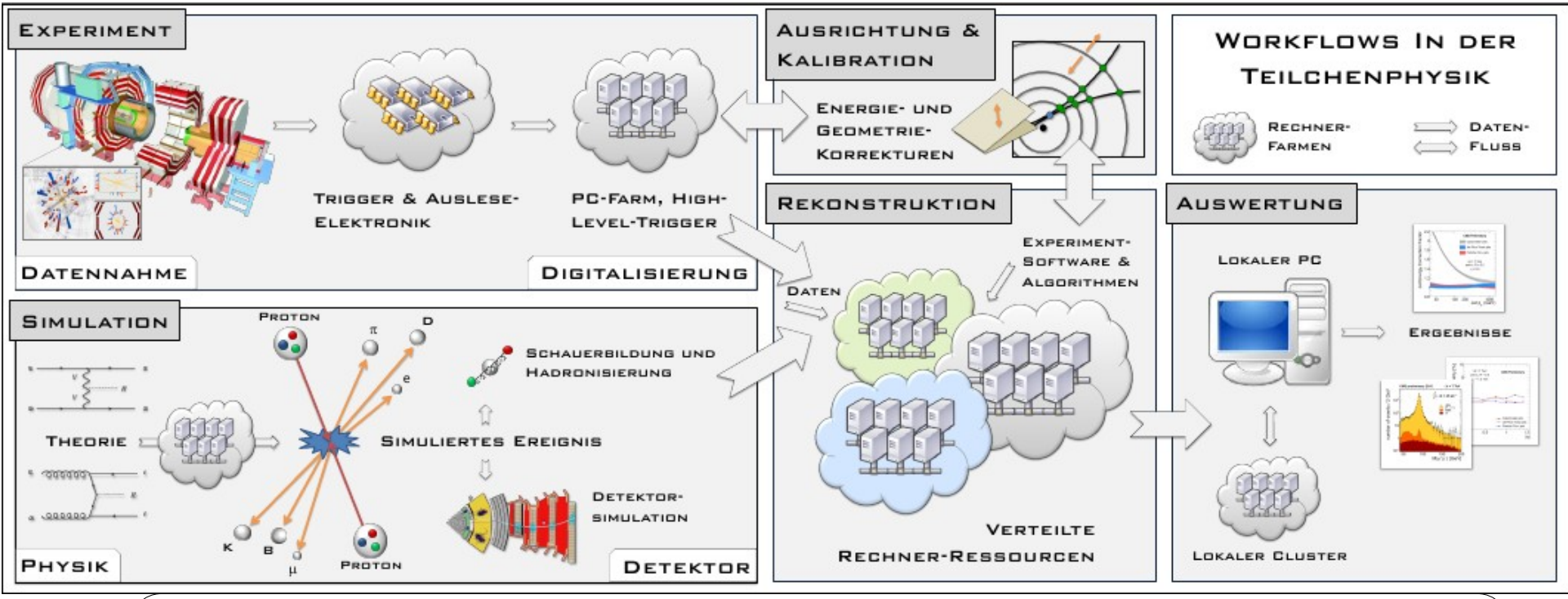
5 Khz (~5 GB/s)

Level 3 – Online Farm

1000 Hz (~1 GB/s)

Die meisten Daten können gar nicht gespeichert werden !

Computereinsatz in großen Experimenten

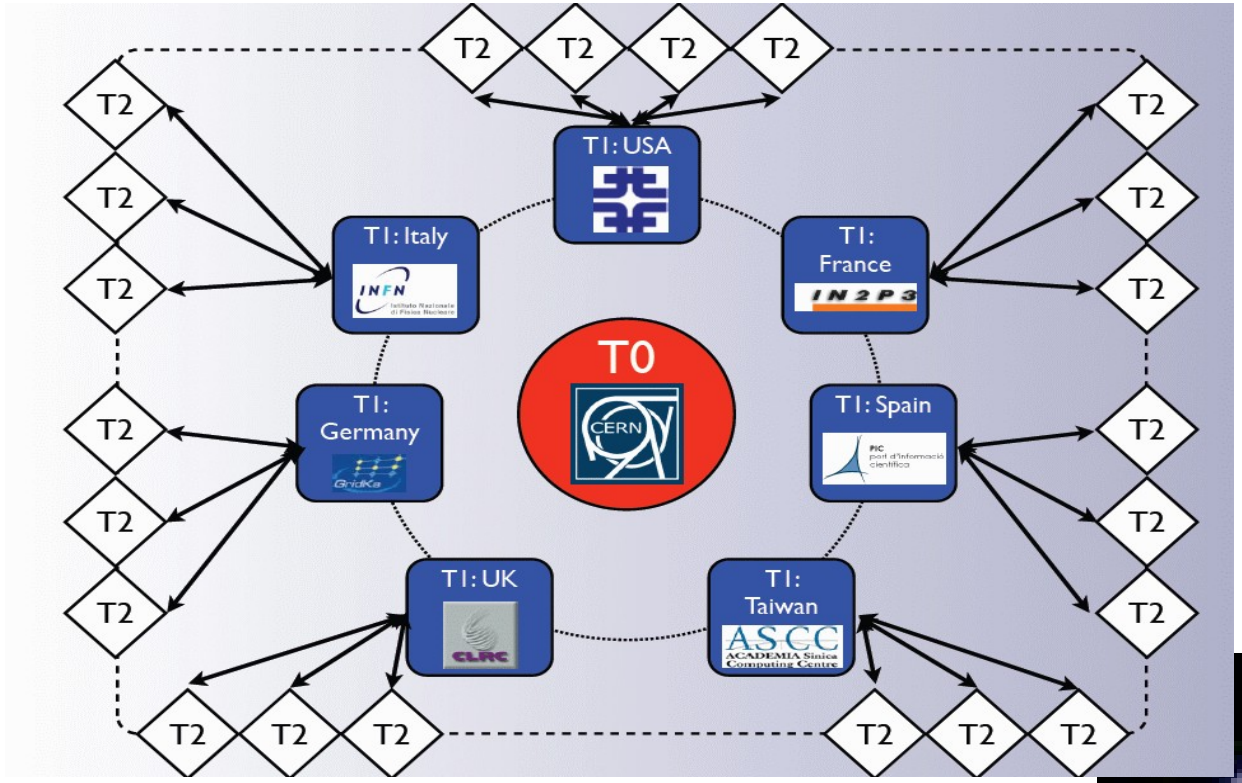


Datenspeicherung und -analyse auf weltweit verteiltem „Grid“ aus z.Zt. 150 Rechenzentren mit insgesamt ca. 1‘000'000 CPU-Kernen, 850'000 TB Plattenplatz und 1‘500‘000 TB Bandspeicher

Breite Palette an Anwendungen mit ganz unterschiedlichem I/O-Bedarf:

- zentral organisierte Simulation und Rekonstruktion großer Datensätze benötigen viel Rechenleistung
- Datenauswahl und Verteilung benötigen hohe I/O- und Netzwerk-Bandbreite
- Analyse durch einzelne Physiker mit wahlfreiem Zugriff auf Daten und Ressourcen mit oft unausgereiftem Code → **die wirkliche Herausforderung !**

Verteilte Datenanalyse im „Grid“



Computing-Model des CMS-Experiments:

CERN, 7 Ebene-1 und ~50 Ebene-2 Zentren

Das

Worldwide LHC Computing Grid

ein weltumspannendes Netz von Datenleitungen, Platten- und Band-Speichern sowie CPU-Ressourcen

Es gibt viel zu tun ...

we need you !

Nächste Vorlesungen

Ab nächste Woche:

Block 3: Computeralgebra

M. Steinhauser

5 VL/Übg.

- Warum Computeralgebra
- Einführung Mathematica
- Mathematica in der Physik
- Numerische Integration
- FORM und andere Computeralgebrasysteme