Humboldt University

Computer Science Department
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Operating Systems Principles

C11

Lab 0 - Auswertung

Make

Das Unix-Tool **make** bietet die Möglichkeit mehrere, voneinander unabhängige Jobs (z.B. das Übersetzen von C-Datei) parallel auszuführen. Dadurch kann besonders auf Multiprozessor/Multicore-Maschinen die Zeit z.B. zum Bauen von großen Software-Projekten deutlich reduziert werden.

Mit der Make-Option "-j" gibt man die Anzahl der parallel ausgefuehrten Jobs angeben.

Aufgabe

Untersuchen Sie den Zusammenhang zwischen der Anzahl parallel ausgeführter Jobs (make -j x) und der Anzahl der Kerne/Prozessoren. Bauen Sie dazu 2 ausgewaehlte Softwarepakete mehrfach mit unterschiedlich vielen parallelen Jobs und bestimmen Sie jeweils die Zeit bis zur Beendigung.
Stellen Sie ihre Ergebnisse entsprechend dar! Erhöhen Sie die Anzahl der Jobs, bis die Dauer des Make-Aufrufs wieder steigt!

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 0 - Auswertung

Fragen

1) Welchen Zusammenhang stellen Sie zwischen Anzahl der Jobs, Anzahl der Kerne und der Zeit fest! Stellen Sie Ihre Ergebnisse mit geeigneten Grafiken dar? (5 Punkte)
2) Erklären Sie diesen Zusammenhang! (2 Punkte)
3) Wie kommen Sie zu konfidenten Ergebnissen? (2 Punkte)
4) Welchen Unterschiede gab es bei den verschiedenen Projekten und warum? (1 Punkt)

Vorgabe

Nutzen Sie für Ihre Evaluierung folgende Software (2 aus 4):

Boostlibary: http://sourceforge.net/projects/boost/files/boost/1.54.0/boost_1_54_0.tar.bz2/download

Linux-Kernel: https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.11.6.tar.xz

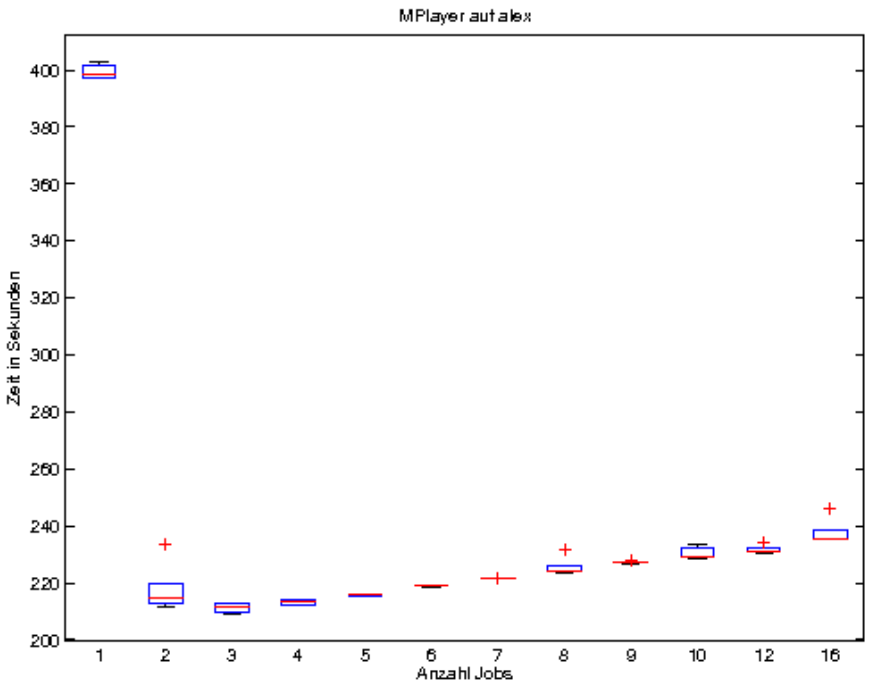
ClickModularRouter: http://www.read.cs.ucla.edu/click/click-2.0.1.tar.gz

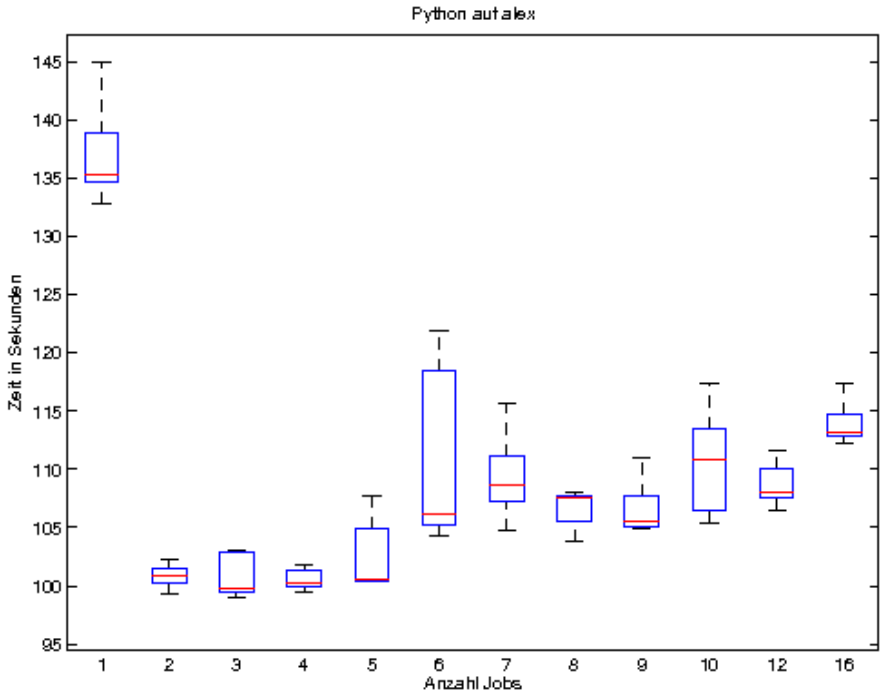
MPlayer: http://www.mplayerhq.hu/MPlayer/releases/MPlayer-1.1.1.tar.xz

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 0 - Beispiel

Rechner: alex

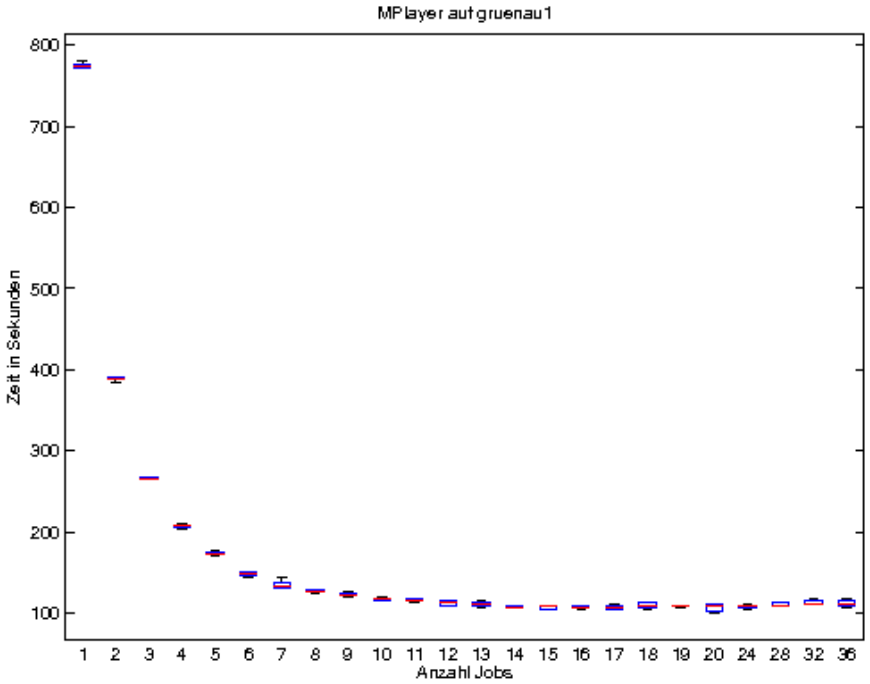


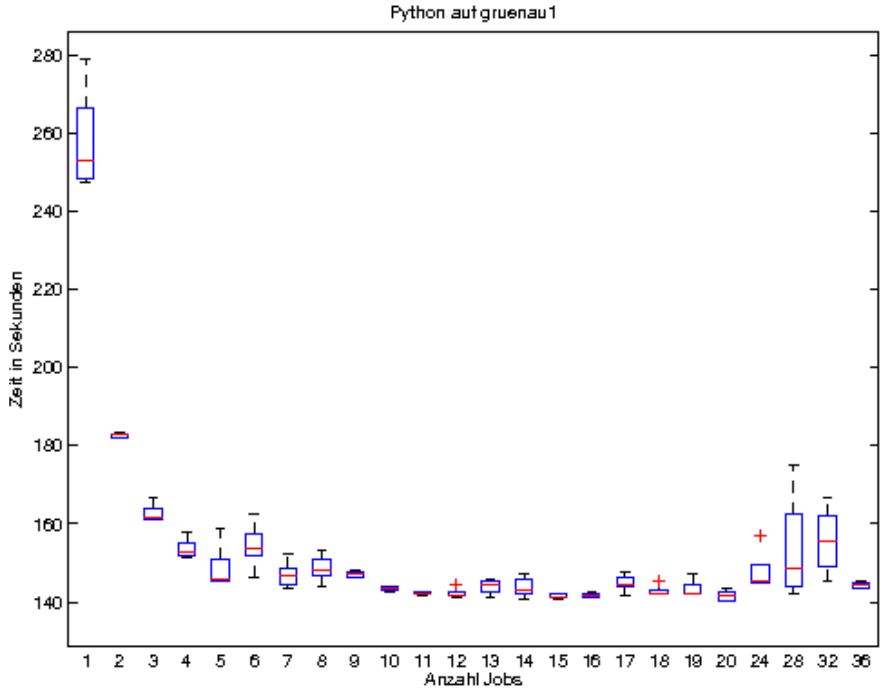


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 0 - Beispiel

Rechner: gruenau1

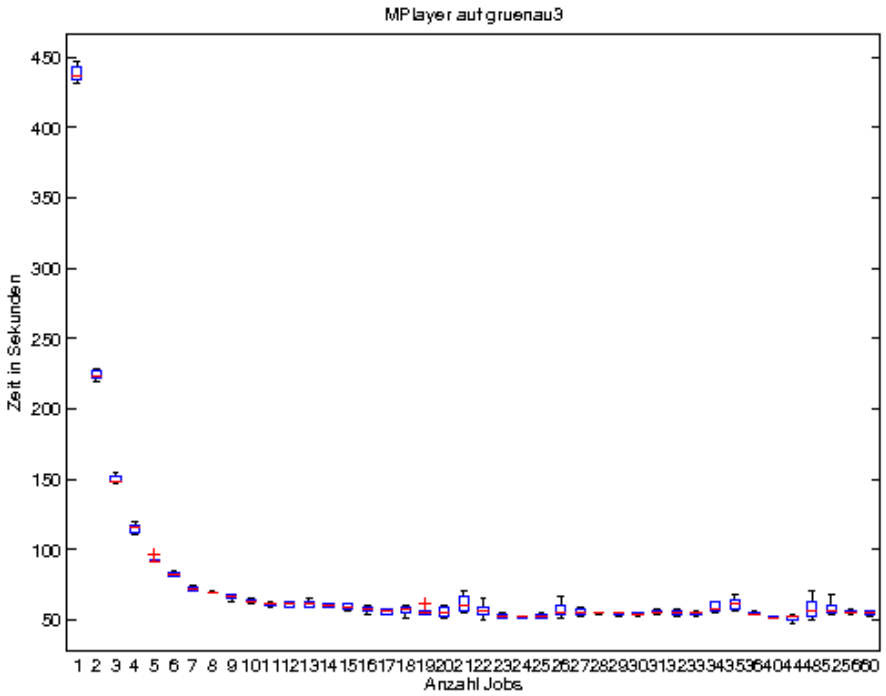


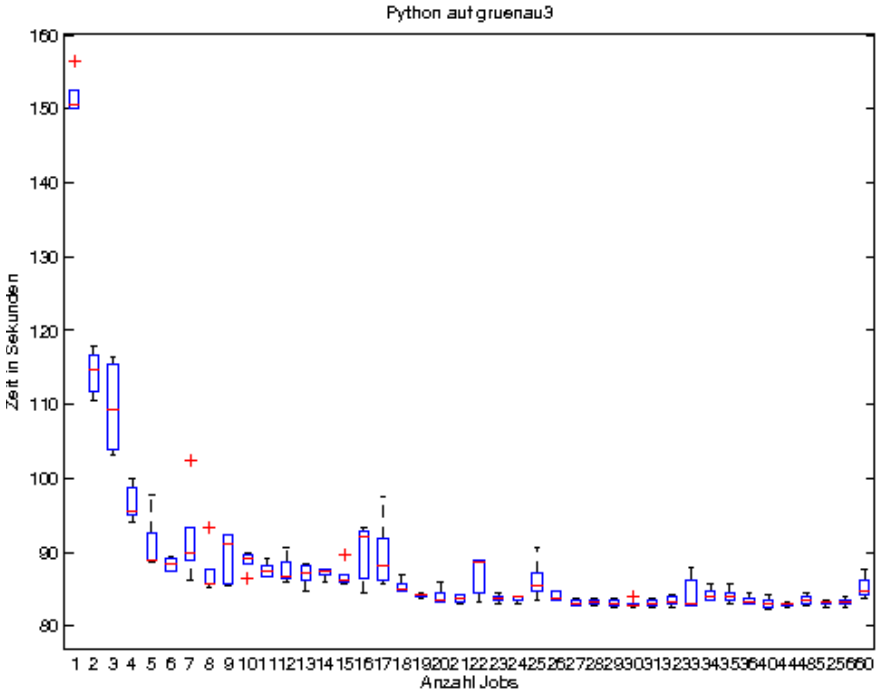


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Lab 0 - Beispiel

Rechner: gruenau3





Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Threads

std::thread

The class thread represents a single thread of execution. Threads allow multiple pieces of code to run asynchronously and simultaneously.

```
#include  <thread>  //  for  std::thread
#include  <iostream>

void  write_message(std::string  const&  message)  {
  std::cout<<message;
}

int  main()  {
  std::thread  t(write_message,
                 "hello  world  from  std::thread\n");
  write_message("hello  world  from  main\n");
  t.join();
}
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Locks

std::mutex

The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

```
#include <iostream>
#include <mutex>
#include <thread>
#include <unistd.h>

std::mutex lock;

void foo() {
  std::cout << "  thread 1: start" << std::endl; sleep(5);
  std::cout << "  thread 1: done" << std::endl;
  lock.unlock();
}

int main() {
  lock.lock();
  std::cout << "main: starting thread 1" << std::endl;
  std::thread t1(foo);

  std::cout << "main: lock again" << std::endl;
  lock.lock();
  std::cout << "main: done" << std::endl;
  t1.join();
  return 0;
}
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Mutex

Mutexes have 3 basic operations, which form the Lockable concept:

- m.lock()
- m.try_lock()
- m.unlock()

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Atomic

std::atomic

Each instantiation & full specialization of the std::atomic template defines an atomic type. Objects of atomic types are the only C++ objects that are free from data races. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

std::atomic<bool> ready (false);

void thread_func(int id) {
  while (!ready) {}                 // wait for the ready signal
};

int main ()
{
  std::thread t(thread_func,1);
  ready = true;
  t.join();

  return 0;
}
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Atomic

Assembler

```
main:
-> _ZNSt11atomic_boolaSEb
-> _ZNSt13__atomic_baseIbEaSEb
-> _ZNSt13__atomic_baseIbE5storeEbSt12memory_order
-> xchgb
```

The exchange operation (xchg) on the x86 swaps the value in a register with the value stored in a memory location. This is an atomic instruction and so can be used to safely get a value and set the value at the same time without interference from any other thread or process.

xchg memoryLocation, %eax

If register %eax had the value 1 and memoryLocation had the value 0, after this instruction the values would be 0 and 1 respectively.

Future / Promise

std::future

The class template std::future provides a mechanism to access the result of asynchronous operations

std::promise

The class template std::promise provides a facility to store a value that is later acquired asynchronously via a std::future object, that the std::promise can supply.

Manually setting futures

- std::promise allows you to explicitly set the value

Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Future / Promise

```
#include <iostream>       // std::cout
#include <functional>     // std::ref
#include <thread>         // std::thread
#include <future>         // std::promise, std::future
#include <unistd.h>

void print_int (std::future<int>& fut) {
  int x = fut.get();
  std::cout << "   after get: " << fut.valid() << '\n';
}

int main ()
{
  std::promise<int> prom;
  std::future<int> fut = prom.get_future();

  std::thread th1 (print_int, std::ref(fut));

  std::cout << "before promise set: " << fut.valid() << std::endl;
  sleep(2);
  prom.set_value (10);

  std::cout << "after promise set: " << fut.valid() << std::endl;

  th1.join();

  sleep(2);
  std::cout << "after join: " << fut.valid() << std::endl;
  return 0;
}
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de

Async - Spawning asynchronous tasks

std::async

The template function async runs the function f asynchronously (potentially in a separate thread) and returns a std::future that will eventually hold the result of that function call.

template< class Function, class... Args>
std::future<typename *std::result_of<Function(Args...)* >::type>
async(Function&& f, Args&&... args);

template< class Function, class... Args >
std::future <typename *std::result_of<Function(Args...)* >::type>
async(*std::launch policy*, Function&& f, Args&&... args);

std::launch policy - bitmask value, where individual bits control the allowed methods of execution

Bit	Explanation
std::launch::async	enable asynchronous evaluation
std::launch::deferred	enable lazy evaluation

Async - Spawning asynchronous tasks

std::launch

std::launch::async => “as if” in a new thread.

std::launch::deferred => executed on demand.

std::launch::async | std::launch::deferred => implementation chooses (default).

Async - Spawning asynchronous tasks


```
#include <iostream>     // std::cout
#include <future>       // std::async, std::future, std::launch
#include <chrono>       // std::chrono::milliseconds
#include <thread>       // std::this_thread::sleep_for

void print_ten (char c, int ms) {
  for (int i=0; i<10; ++i) {
    std::this_thread::sleep_for (std::chrono::milliseconds(ms));
    std::cout << c;
  }
}

int main () {
  std::cout << "with launch::async:\n";
  std::future<void> foo = std::async (std::launch::async,print_ten,'*',100);
  std::future<void> bar = std::async (std::launch::async,print_ten,'@',200);
  // async "get" (wait for foo and bar to be ready):
  foo.get();
  bar.get();
  std::cout << "\n\n";

  std::cout << "with launch::deferred:\n";
  foo = std::async (std::launch::deferred,print_ten,'*',100);
  bar = std::async (std::launch::deferred,print_ten,'@',200);
  // deferred "get" (perform the actual calls):
  foo.get();
  bar.get();
  std::cout << '\n';

  return 0;
}
```

Transactional Memory

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

std::atomic<bool> ready (false);

int a,b,c;


void thread_func(int id) {
  while (!ready) {}                 // wait for the ready signal

  __transaction_atomic { c = a - b; }

};

int main ()
{
  a = b = 5;
  c = 0;
  std::thread t(thread_func,1);
  ready = true;

  __transaction_atomic { if (a > b) b++; }

  t.join();

  return 0;
}
```


Humboldt Universität Informatik
Systems Architecture Group
http://sar.informatik.hu-berlin.de