



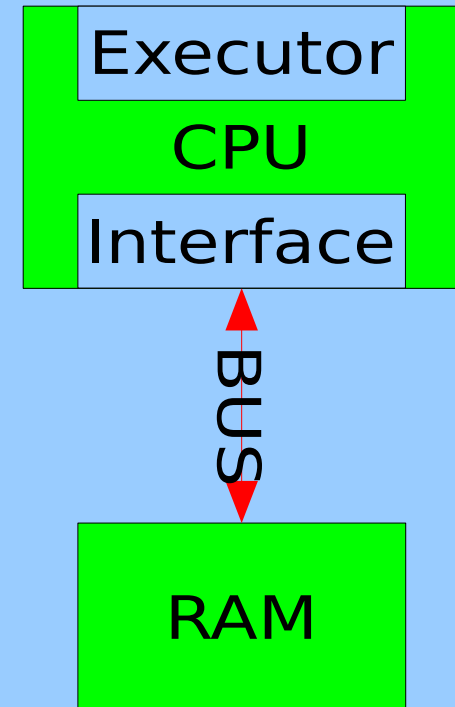
# Multiprozessoren und Multithreading

- Vortragsreihe „Chaos-Seminar“
  - Veranstalter: Chaos Computer Club Ulm
  - Infos: <http://ulm.ccc.de/ChaosSeminar>
  - Kontakt: [mail@ulm.ccc.de](mailto:mail@ulm.ccc.de)
  - Montagstreff: <http://ulm.ccc.de/MontagsTreff>
- Referenten:
  - [Alexander.Bernauer@ulm.ccc.de](mailto:Alexander.Bernauer@ulm.ccc.de)
  - [Markus.Schaber@ulm.ccc.de](mailto:Markus.Schaber@ulm.ccc.de)



# Grundlagen - Alles ganz einfach!

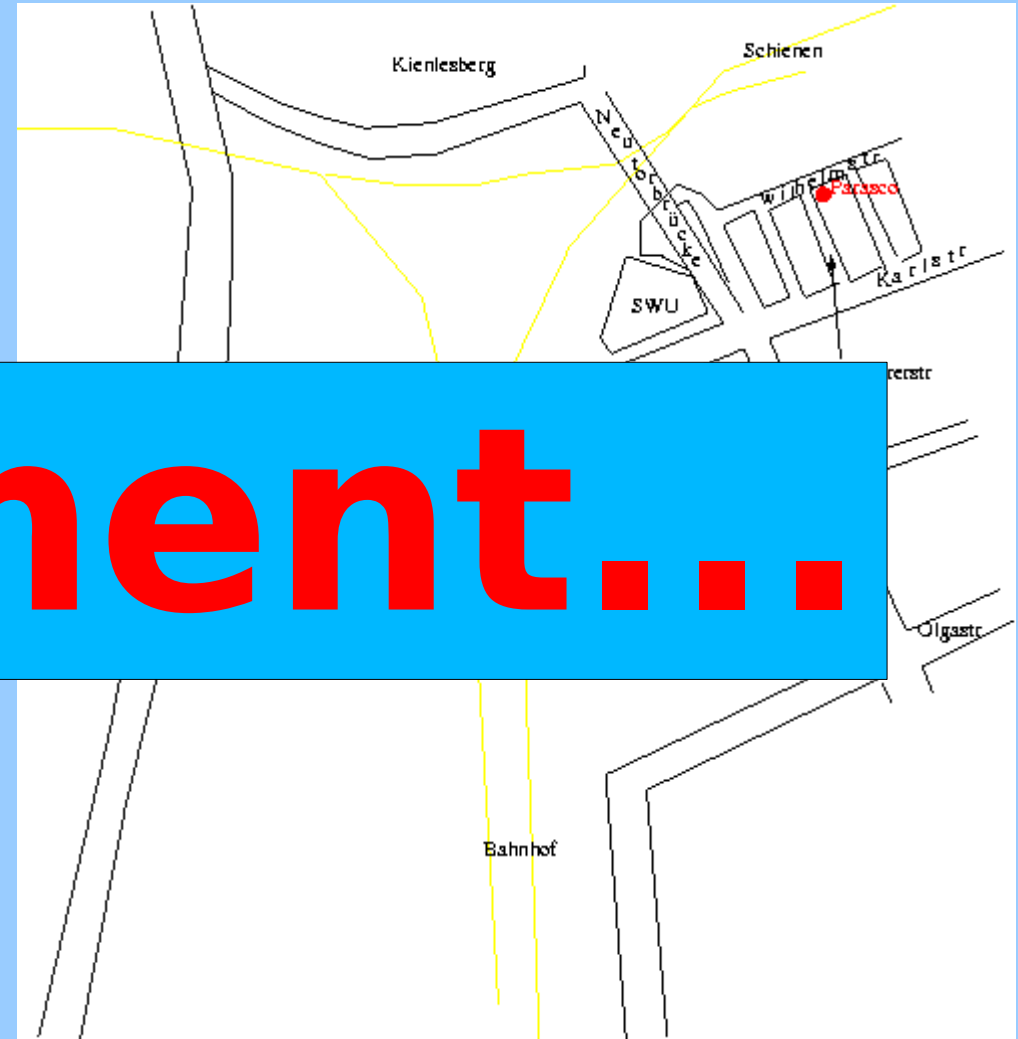
- Von-Neumann Architektur
- Sequenzieller Ablauf
  - Befehl laden
  - Befehl dekodieren
  - Daten laden
  - Daten verarbeiten
  - Daten zurückschreiben
- übliches Modell
  - fast alle verbreiteten Programmiersprachen
  - in den Köpfen der Entwickler





# Das wars...

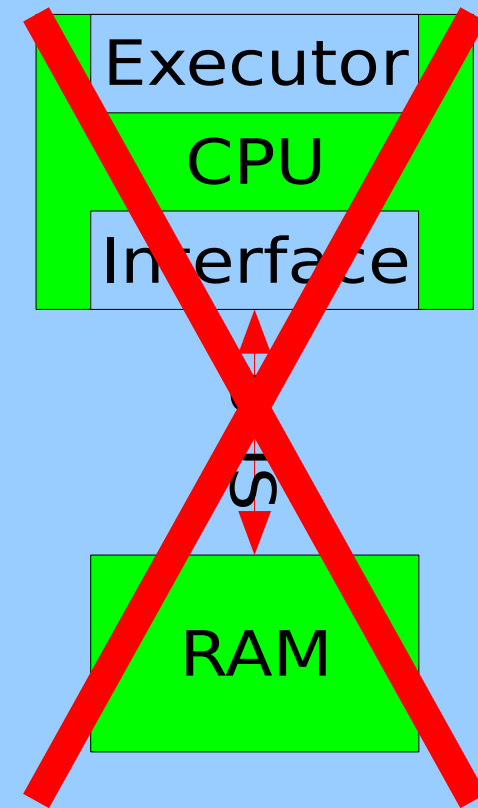
- Noch Fragen?
- Auf geht's zu Parasco.





# Grundlagen - alles ganz einfach?

- präemptive Betriebssysteme
  - Threads, Prozesse, Interrupts, Signale
- ahnungslose Compiler
- Hardware
  - Pipelines
  - Caches
  - Multiprozessoren
  - DMA
  - Busoptimierung



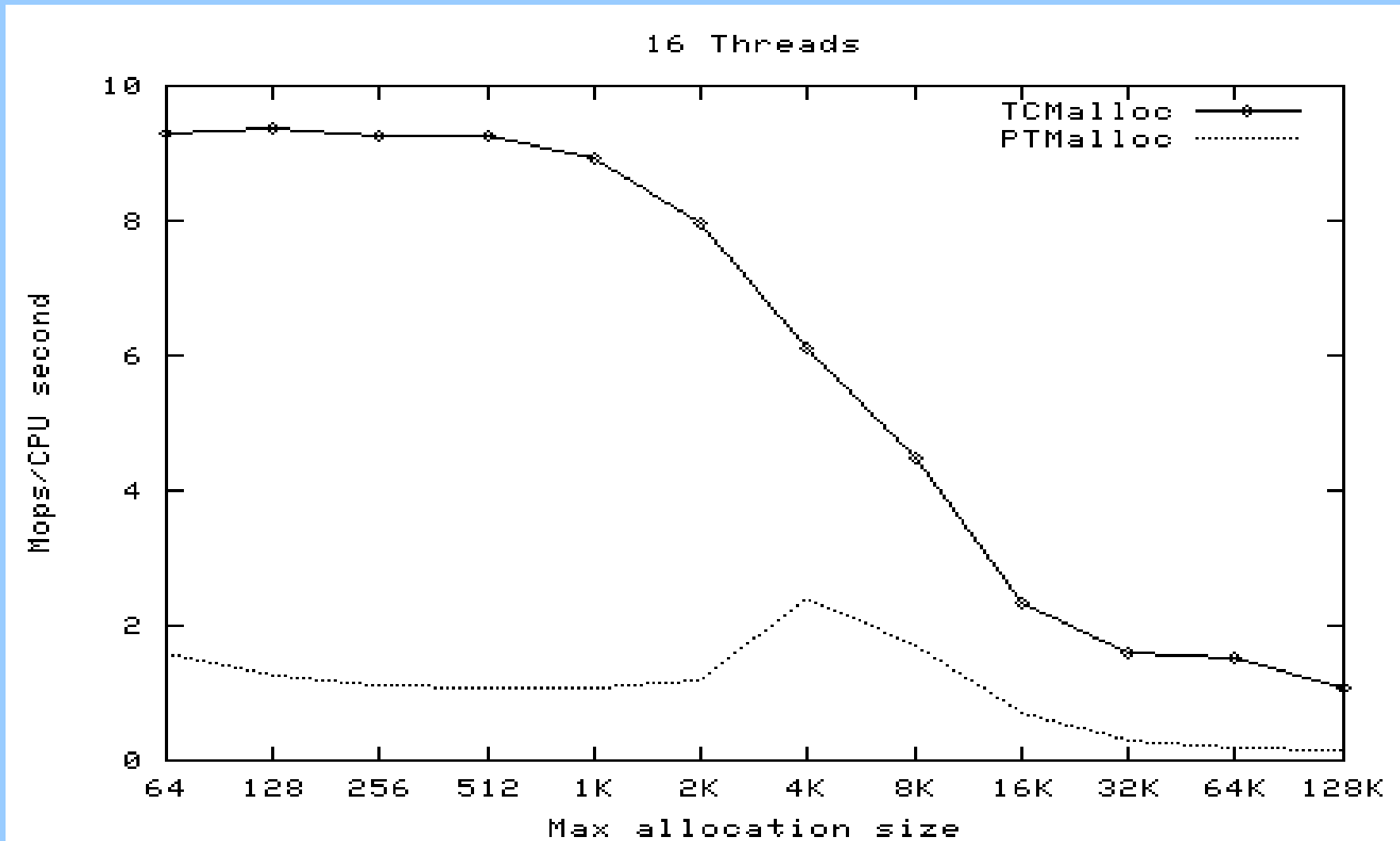


# Motivation

- Hardware wird massiv parallel
- Software hinkt hinterher
- Beispiel Freispeicherverwaltung:
  - Objekt kann von Thread 1 angelegt und von Thread 2 freigegeben werden
  - naiver Ansatz erfordert viel Synchronisation

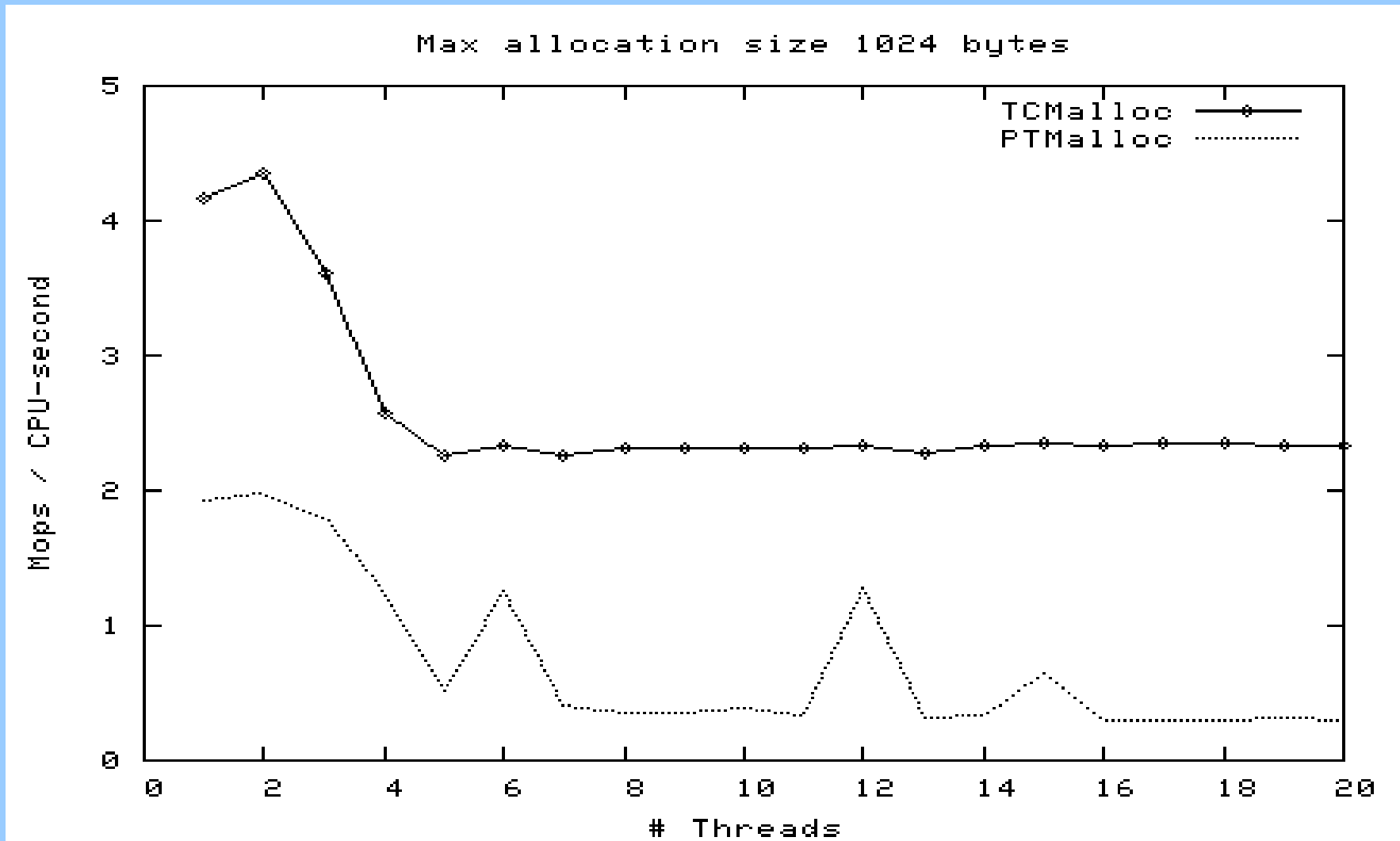


# Thread Caching Malloc





# Thread Caching Malloc





# Nebenläufigkeit der Software

- Abhängigkeiten durch Kommunikation
- Betriebssystem simuliert Parallelität
  - Prozesse
  - Threads
- OS-Kern schaltet „beliebig“ um
  - Zeitscheiben abgelaufen
  - Warten auf I/O
  - asynchrone I/O
    - Benutzereingabe, Festplatte, ...



# Nebenläufigkeit der Hardware

- Prozessoren arbeiten parallel
  - CPUs
  - I/O Bausteine
  - Grafik
  - etc.
- Timing nicht exakt vorhersehbar
  - Hardware, Netzwerk, Nutzer, Berechnungsdauer



# Beispiel Nebenläufigkeit

```
int a = 0;  
int b = 0;
```

```
// CPU 1  
a += 1;  
b += 1;
```

```
// CPU 2  
locala = a;  
localb = b;  
if (localb > locala) {  
    printf("error");  
}
```



# Beispiel Nebenläufigkeit (korrekt)

```
int a = 0;  
int b = 0;
```

```
// CPU 1  
a += 1;  
b += 1;
```

```
// CPU 2  
localb = b;  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# statisches Reordering

- Compiler optimiert
  - Optimale Auslastung der Hardware
    - Pipelines
    - Execution Units
    - Caches
    - Branch Prediction Units
  - Code size
  - Alignment
- EPIC



# Beispiel statisches Reordering

```
int a = 0;  
int b = 0;
```

```
// CPU 1  
a += 1;  
b += 1;
```

```
// CPU 2  
localb = b;  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# Beispiel statisches Reordering (korrekt)

```
volatile int a = 0;  
volatile int b = 0;
```

```
// CPU 1  
a += 1;  
b += 1;
```

```
// CPU 2  
localb = b;  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# dynamisches Reordering

- Schneller Befehl überholt langsamen
  - Pipelines
  - mehrere Execution Units
- Busszugriffsoptimierung
  - Write combining
  - Delayed write
  - Cut-Through-Switching
- Bricht Treiber
  - Vertauschen von I/O Steuerungszugriffen



# Beispiel dynamisches Reordering

```
volatile int a = 0;  
volatile int b = 0;
```

```
// CPU 1  
a += 1;  
b += 1;
```

```
// CPU 2  
localb = b;  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# Beispiel dynamisches Reordering (korrekt)

```
volatile int a = 0;  
volatile int b = 0;
```

```
// CPU 1  
a += 1;  
writeBarrier();  
b += 1;
```

```
// CPU 2  
localb = b;  
readBarrier();  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



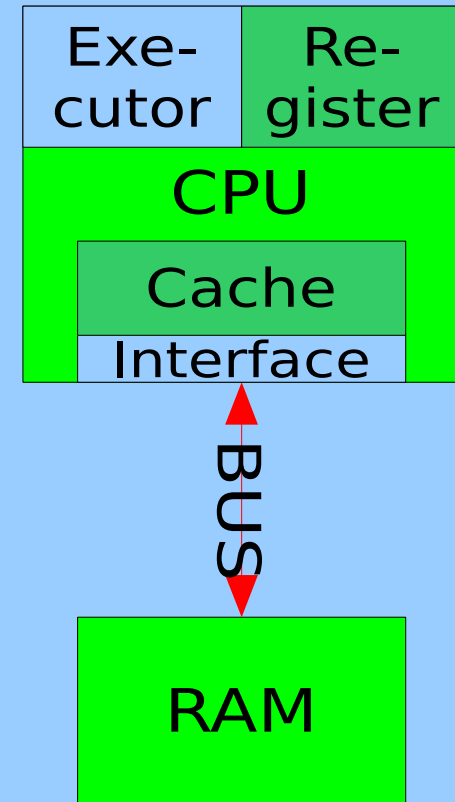
# dynamisches Reordering: Lösungen

- Lösungen
  - Spezielle Maschinenbefehle
  - Spezielle Buszyklen
  - Adressbereichsabhängiges Reordering
    - I/O Adressraum, MTRR & co
  - Oft inline-Assembler



# Caches

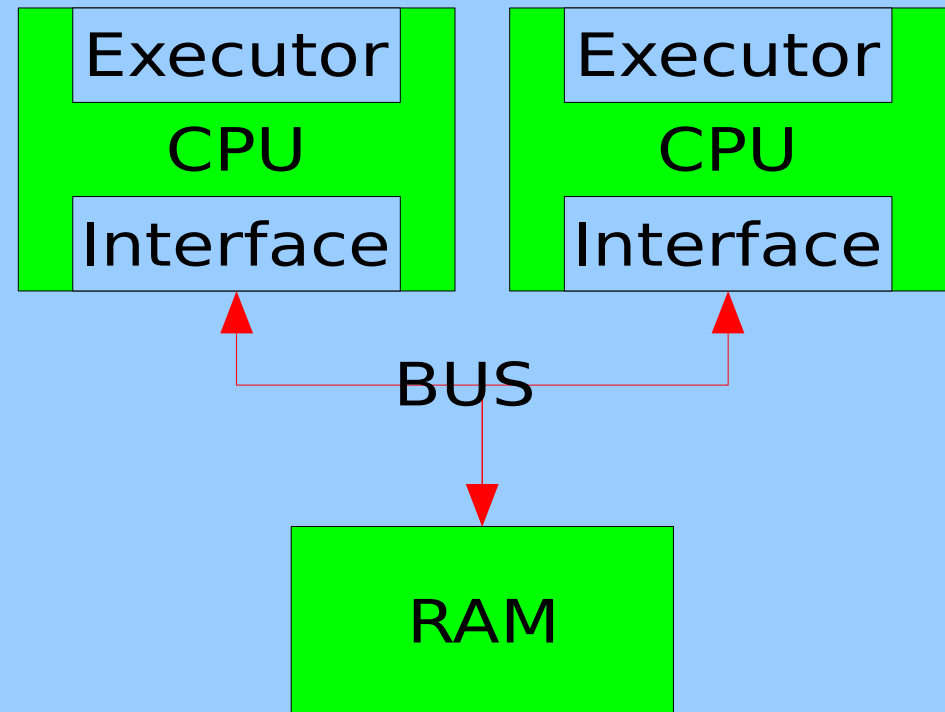
- Speicher ist zu langsam
- BUS ist ein Flaschenhals
- Schnelle Zwischenspeicher
  - Register
  - Puffer an Busschnittstellen
  - Code-Cache
  - Daten-Cache
  - 2<sup>nd</sup> bis n<sup>th</sup> Level Cache
  - Software-Caches





# SMP

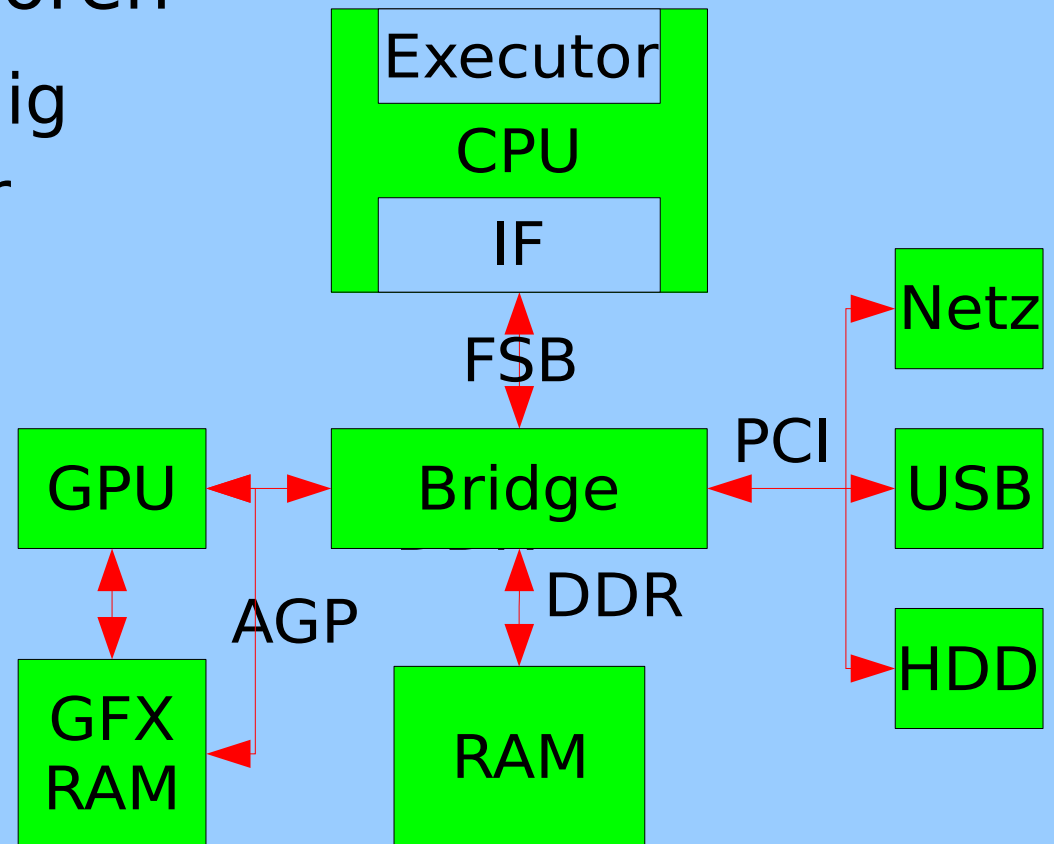
- ein Prozessor ist zu langsam
- wir nehmen mehrere
- Jede hat Caches
- Variationen
  - HyperThreading
  - Multi-Core Chip
  - Multi-Chip Package
  - Niagara





# Caches: AMP / IO / Coprozessoren

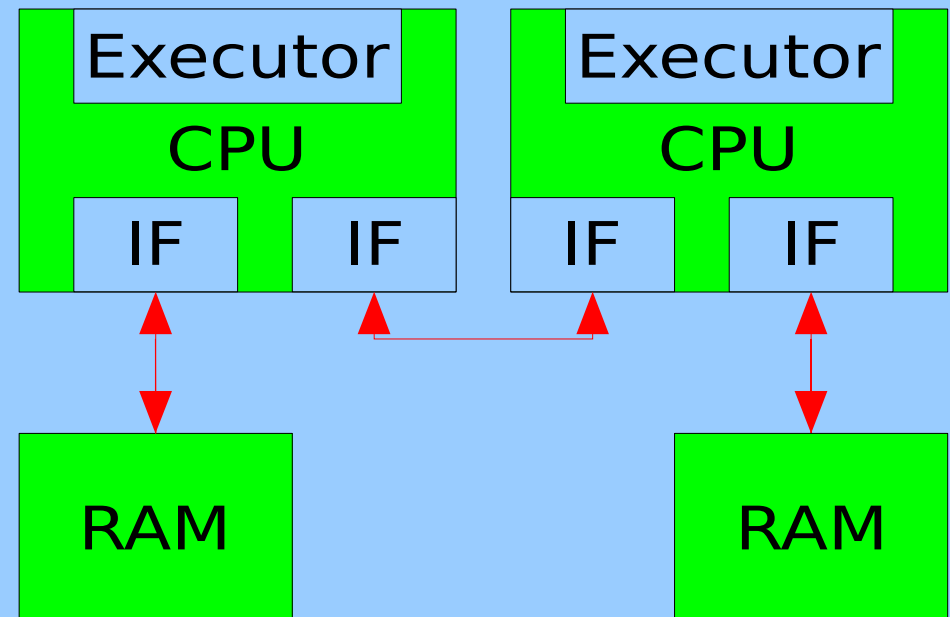
- Viele Spezialprozessoren
  - agieren eigenständig
  - eigenes RAM/Puffer
  - eigene Caches
- Spezialfälle
  - Externe FPU
  - AMP
  - Cell SPU





# Caches: NUMA

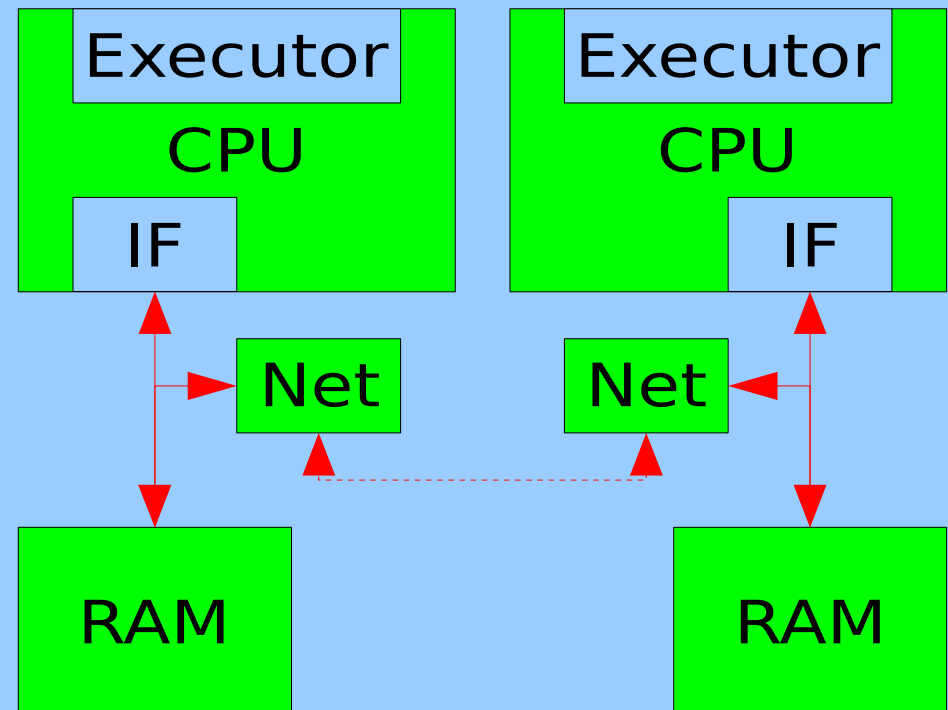
- BUS wieder Flaschenhals
- jede CPU kriegt eigenen Speicher
- Verteilte Caches





# Caches: Clustering

- Eigenständige Rechner
- Kopplung per Software
  - Ethernet
  - Myrinet
- Seitencaching per Software





# Cache Kohärenz

- Nicht alle Akteure wissen von allen Caches
- Ständiges aktualisieren teuer
- Probleme:
  - Cache enthält veraltete Daten
  - Aktuelle Daten nur im Cache, RAM veraltet
  - Widersprüchliche Daten in verschiedenen Caches
  - DMA-Hardware ignoriert Caches
  - Compiler hat aktuellen Stand noch im Register



# Cache Kohärenz: Lösungen

- Hardware-Kohärenzprotokolle
  - z. B. AMD Hypertransport: ccNUMA
  - Aufwändig, skaliert nicht hoch
- Befehle zur Cache-Kontrolle
  - Flush, Invalidate
- Sprache und Bibliothek
  - Kapselung der Hardwarebefehle
  - Häufig Synchronisation und Caches gekoppelt
    - acquire / release semantics



# Beispiel Caches:

```
volatile int a = 0;  
volatile int b = 0;
```

```
// CPU 1  
a += 1;  
writeBarrier();  
b += 1;
```

```
// CPU 2  
localb = b;  
readBarrier();  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# Beispiel Caches (korrekt):

```
volatile int a = 0;  
volatile int b = 0;
```

```
// CPU 1  
a += 1;  
writeBarrier();  
b += 1;  
cacheFlush();
```

```
// CPU 2  
cacheInvalidate();  
localb = b;  
readBarrier();  
locala = a;  
if (localb > locala) {  
    printf("error");  
}
```



# Lokalität der Daten

- Viele verschiedene Speicher
  - NUMA-Nodes
  - Grafik-RAM
  - Puffer für I/O
- Zugriffe unterschiedlich teuer
- Lösungen:
  - Lokale Allokation von Ressourcen
  - Binden von Programmen an NUMA-Nodes
  - Replikation von Daten
  - Migrationsverfahren / Load Balancing



# Lokalität: False Sharing

- Objekte auf gemeinsamer Cache-Line / Seite
  - Thrashing / Flapping
- Spezialfall: Memory Location Collision
  - Getrennte Variablen nicht getrennt adressierbar
  - z. B. BitFields
  - undefiniertes Verhalten
- Lösung
  - Padding / Alignment



# Probleme: Skalierungsprobleme

- Gute Skalierbarkeit:
  - Problem leicht zerlegbar
  - Kommunikationsaufwand gering
  - z. B. Video Ray Tracing, DES knacken, ...
- Schlechte Skalierbarkeit:
  - Rechenschritte bauen aufeinander auf
  - Hoher Kommunikationsaufwand
    - manchmal durch Spezialhardware lösbar



# Heisenbugs

*„Ein Fehler, der verschwindet, oder sein Verhalten ändert, wenn man versucht, ihn im Programmcode aufzuspüren.“*

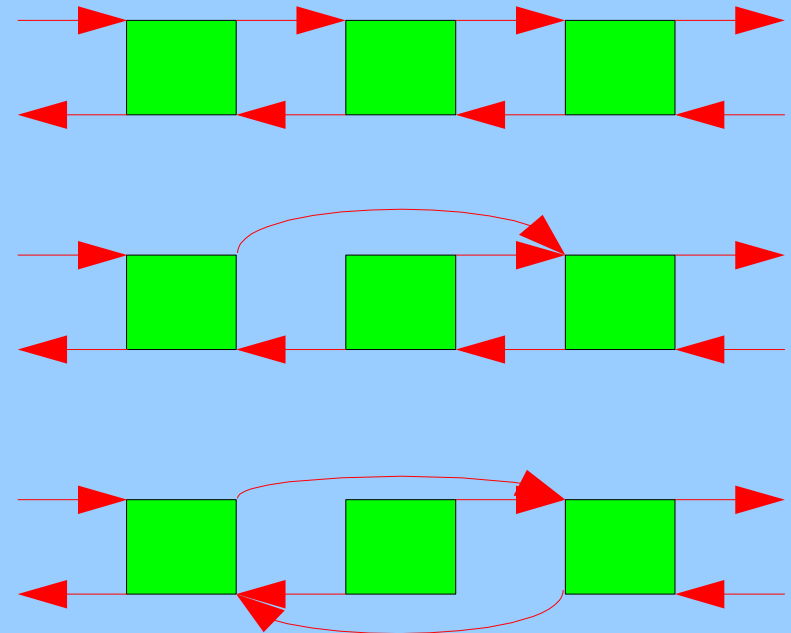
- Hier besonders häufig
  - Debuggen verändert Timing
  - Debuggen erzeugt Signale
  - Debugger beeinflusst Caches
  - Compileroptimierung abgeschaltet



# Nebenläufigkeit – komplexere Daten

- Atomare Befehle
  - Bus Locking
  - compare/exchange
- Kritische Abschnitte
  - Semaphores
  - Rendezvous
  - Locks
  - Synchronized
- Lock Free Synchronization
  - Nur in Spezialfällen möglich

Beispiel: doppelt  
verkettete Liste





# C/C++

- Spezifikation einer abstrakten Maschine
- „observable behavior“ muss korrekt sein
  - Zugriff auf volatile Objekte
  - Aufruf von „library I/O functions“
- C/C++ kennt keine Threads



# pthread

- keine formale Definition des Memory Models
- manuelle Synchronisation
- Granularität ist „memory location“



# Extra Writes

- Zugriff

```
struct {  
    int a : 17;  
    int b : 15;  
} x;  
x.a = 42;
```

- Code

```
uint32_t tmp = x;  
tmp &= ~0x1ffff;  
tmp |= 42;  
x = tmp;
```

- Schreibzugriff auf a schreibt auch b!
- Variable != „memory location“



# pthread portabel?

- Zugriff

```
struct {  
    char a; char b;  
    char c; char d;  
} x;
```

```
x.b='b';
```

```
x.c='c';
```

```
x.d='d';
```

- Code

```
x = 'dcb\0' | x.a;
```



# pthread portabel??

- „Mit genügend Platz dazwischen ist's 'ne andere memory location, oder?“
- Linker ordnet globale Variablen beliebig an
- pthread ist (theoretisch) nicht portabel
- in den meisten Fällen passt es dennoch



# optimieren!1!

- nur locken, falls anderer Thread läuft

```
for (...) {  
    if (cond) pthread_mutex_lock(...);  
    X = ... X ...  
    if (cond) pthread_mutex_unlock(...);  
}
```



# Compiler optimiert auch :-o

- x wird in einem Register gehalten

```
r = x;
```

```
for (...) {
```

```
    if (cond) { x=r; pthread_mutex_lock(...); r=x; }
```

```
    r = ... r ...
```

```
    if (cond) { x=r; pthread_mutex_unlock(...); r=x; }
```

```
}
```

```
x = r;
```



# Praxis: LibNUMA

- Abfragen und Setzen
  - Speicher
    - Prozess, Region, Shared Memory
    - default, bind, preferred, interleaved
  - CPU
    - CPU oder Node
  - vererbt bei fork / exec & co
- Kommandozeile: `numactl`, `numastat`
- Zukunft: Hierarchien, OS Caches



# Praxis: Java Memory Model

- Typsicherheit auch bei Data Races
  - Sandboxing
- Threads in der Sprache unterstützt
- In Java 1.4/1.5 überarbeitet
  - Intuitiveres Modell
  - Bessere Optimierungsmöglichkeiten
  - Korrekter alter Code bleibt korrekt



# Praxis: Java Memory Model Sprachunterstützung

- synchronized
  - Exklusivität
  - Sichtbarkeit
  - Optimizer Constraints
- volatile
  - Sichtbarkeit
  - Optimizer Constraints
- Memory Location klar definiert
  - Membervariable
  - Array-Element



# Praxis: Java Memory Model Initialisierungssicherheit

- Alte Semantik von final:

```
String s1 = "/usr/tmp";
```

```
String s2 = s1.substring(4); // contains "/tmp"
```

- Optimierter Code:

- Allokieren (nullt aus)
- Adresse in s2 schreiben
- Konstruktorcode bearbeiten

- String hat final attribute length und offset

- s2 kurzzeitig „/usr“ statt „/tmp“

- Überarbeitet in 1.5

- Verstoß innerhalb des Konstruktors noch möglich



# Praxis: Java Memory Model Double Checked Locking

- Lazy Initialization
  - Teure Konstruktion erst bei Bedarf
- Probleme
  - Race Condition
  - Kein Cache berücksichtigt

```
class SomeClass {  
    private Resource resource = null;  
    public Resource getResource() {  
        if (resource == null) {  
            resource = new Resource();  
        }  
        return resource;  
    }  
}
```

**KAPUTT!**



# Praxis: Java Memory Model Double Checked Locking

- Liest aus veraltetem Cache

```
class SomeClass {
    private Resource resource = null;
    public Resource getResource() {
        if (resource == null) {
            synchronized {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

**KAPUTT!**



# Praxis: Java Memory Model Double Checked Locking

- Altes Memory Modell:
  - kaputt
- Neues Memory Modell:
  - klappt, aber ineffizient

```
class SomeClass {  
    private volatile Resource  
        resource = null;  
    public Resource getResource() {  
        if (resource == null) {  
            synchronized {  
                if (resource == null)  
                    resource = new Resource();  
            }  
        }  
        return resource;  
    }  
}
```



# Praxis: Java Memory Model

## Double Checked Locking

- Alternative:
  - Klasseninitialisierung garantiert einmalig
  - Initialisierung bei erster Nutzung
  - Klassenvariablen garantiert sichtbar

```
class Holder {
    private static class innerHolder {
        public static Something something = new Something();
    }
    public static Something getInstance() {
        return innerHolder.something;
    }
}
```



# OpenMP

- Portable Alternative zu Message Passing
- Kommunikation durch Shared Memory
- Inkrementelle Parallelisierung
- Compilerdirektiven und Laufzeitfunktionen



# fork & join model

```
int parameter, ergebnis;  
#pragma omp parallel for \  
    shared(parameter) \  
    reduction(+:ergebnis)  
for (i=0; i<anzahl_teile; i++) {  
    int teilergebnis = teilberechnung(i, parameter);  
    ergebnis += teilergebnis;  
}
```



# OpenMP

- thread creation (`parallel`)
- work sharing (`do`, `section`, `single`)
- data scoping (`shared`, `private`)
- synchronisation (`critical`, `barrier`, `nowait`)
- scheduling (`schedule static`, `dynamic`)
- runtime (`omp_get_thread_num`)
- environment (`OMP_NUM_THREADS`)
- ...



# Effizienz vs. Komfort

- Parallelität
  - automatische Parallelisierung
  - automatische Synchronisation
  - Compiler und Laufzeit optimiert für jeweilige Umgebung



# Über dem Tellerrand...

- verteilte Systeme (pessimistisch)
  - Teilnehmer schicken sich Nachrichten zu
  - Behandlung erzeugt ggf. neue Nachrichten
  - Nachrichtenbasierte Abarbeitung garantiert Einhaltung der Kausalitäten.



# ...findet sich allerhand

- Programm = Objekte und Nachrichten.
- Optimierungsaufgabe für den Compiler
  - Zeitliche Verteilung der Objekte auf die Threads
  - Minimierung der inter-Thread Kommunikation
- nicht entscheidbare Probleme in die Laufzeit verschieben.
- ggf. Hints für andere Semantikebenen nötig



# Effizienz vs. Komfort II

- Laufzeit-API
  - gefühlt synchron
  - in Wirklichkeit asynchron
- Systemobjekte mit blockierenden Methoden
- transparente Entkopplung
- Ein Thread pro Core!



# Beispiel

```
foreach(packet = socket.read()) {  
    file = packet.process()  
    content = file.read()  
    answer = content.process()  
    socket.write(answer)  
}
```

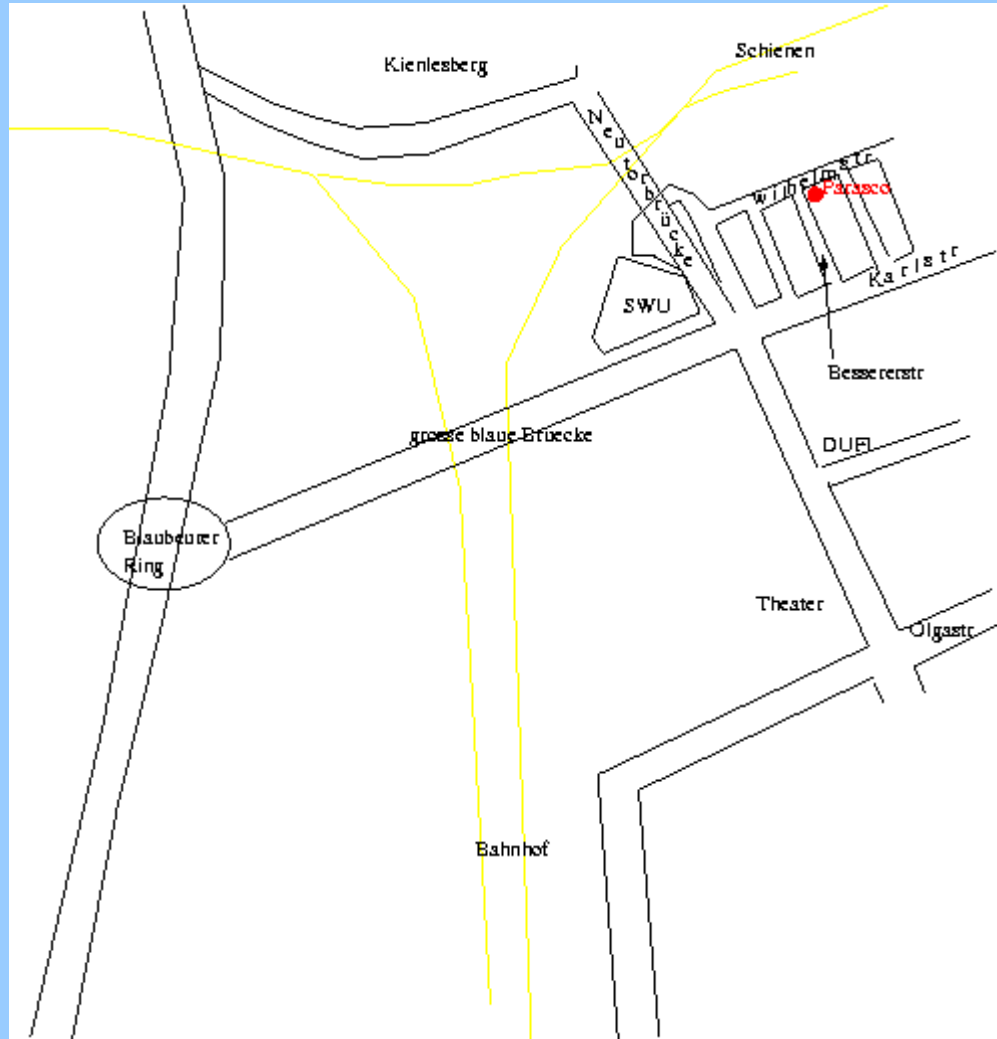


# Das wars...

- Nur die wichtigsten Dinge gestreift, einiges ausgelassen
- Alle Warenzeichen sind natürlich Eigentum der jeweiligen Inhaber.
- Noch Fragen?
- Verwendete Software:
  - OpenOffice
  - Debian GNU/Linux
  - u. A...



# Auf geht's zu Parasco



- Weitere Diskussion
- Griechisches Essen
- Kühle Getränke

<http://parasco.home.pages.de>



# Quellen

- C99: ISO-9899-1999
- ISO C++: ISO-14882-1998
- Single UNIX Specification: IEEE 1003.1-2001
- lock opcode: [http://www.gnu.org/software/binutils/manual/gas-2.9.1/html\\_node/as\\_199.html](http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_node/as_199.html)
- Java Memory Modell – Motivation: <http://www.ibm.com/developerworks/java/library/j-jtp02244.html>
- Java memory Modell – Linkliste: <http://www.cs.umd.edu/~pugh/java/memoryModel/>
- Lock Free Synchronization: <http://www.google.com/search?q=Lock+Free+Synchronisation>
- Threads and Memory Modell for C++: [http://www.hpl.hp.com/personal/Hans\\_Boehm/c++mm/](http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/)
- LibNUMA – NUMA API for Linux: <http://www.novell.com/collateral/4621437/4621437.pdf>
- Threads Cannot be Implemented as a Library: <http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html>
- Memory Barriers: <http://ridiculousfish.com/blog/archives/2007/02/17/barrier/>
- OpenMP: <http://www.sc.rwth-aachen.de/Teaching/Lectures/HPC05/openmp-mpi-pi.pdf>
- Thread Caching Malloc: <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>