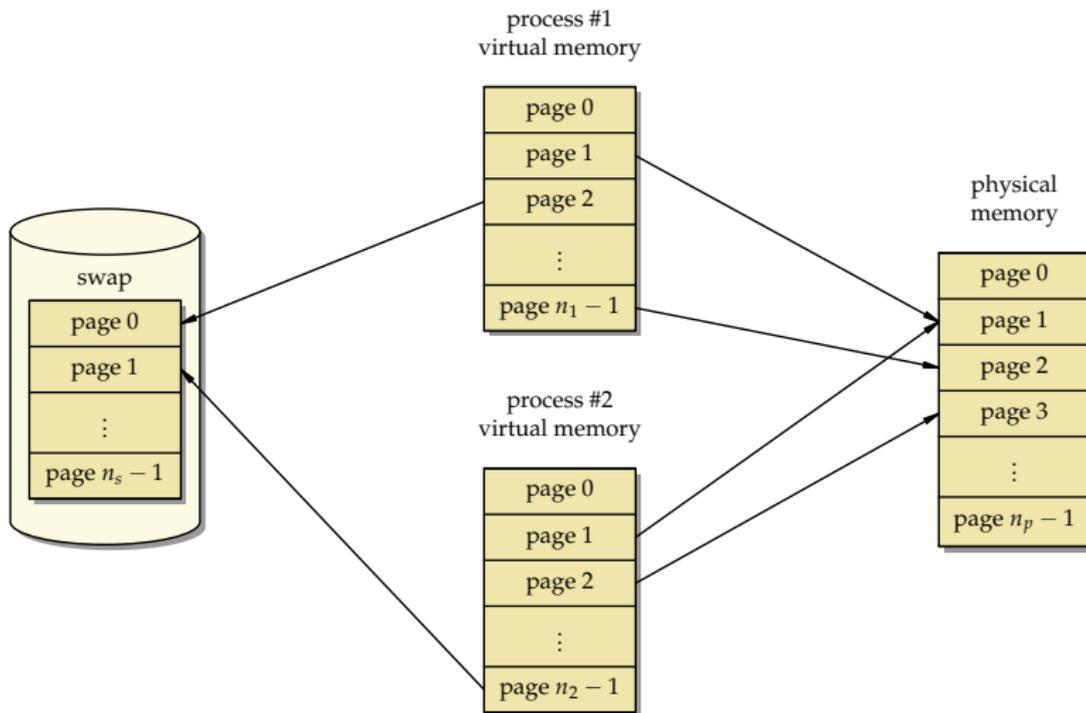


- Sofern gemeinsamer Speicher zur Verfügung steht, so bietet dies die effizienteste Kommunikationsbasis parallelisierter Programme.
- Die Synchronisation muss jedoch auf anderem Wege erfolgen. Dies kann entweder mittels der entsprechenden Operationen für Threads (etwa mit *join*), über lokale Netzwerkkommunikation oder anderen Synchronisierungsoperationen des Betriebssystems erfolgen.
- Am häufigsten anzutreffen ist die UMA-Variante (*uniform memory access*, siehe Abbildung). Diese Architektur finden wir auf der Theseus und den einzelnen Knoten der Pacioli vor.

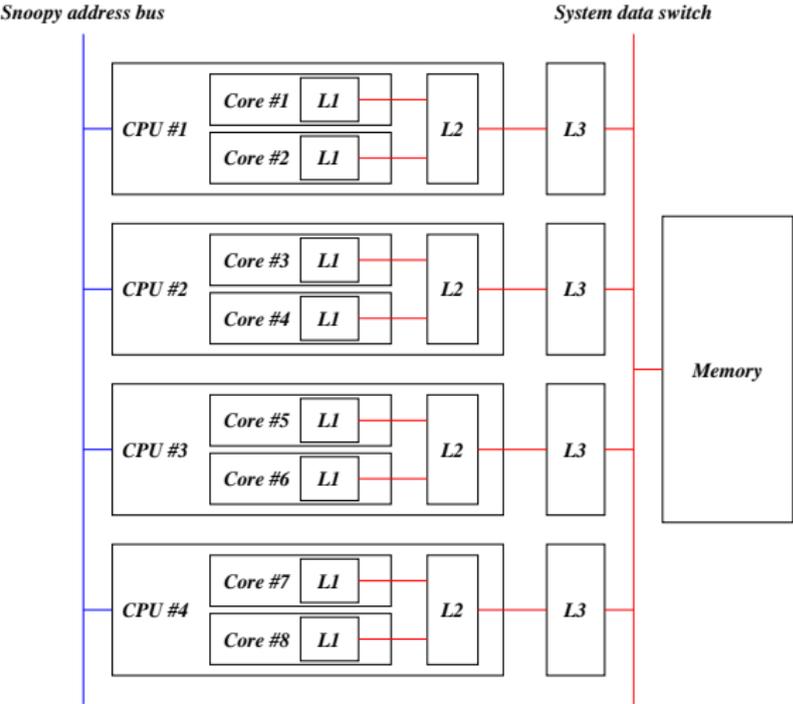


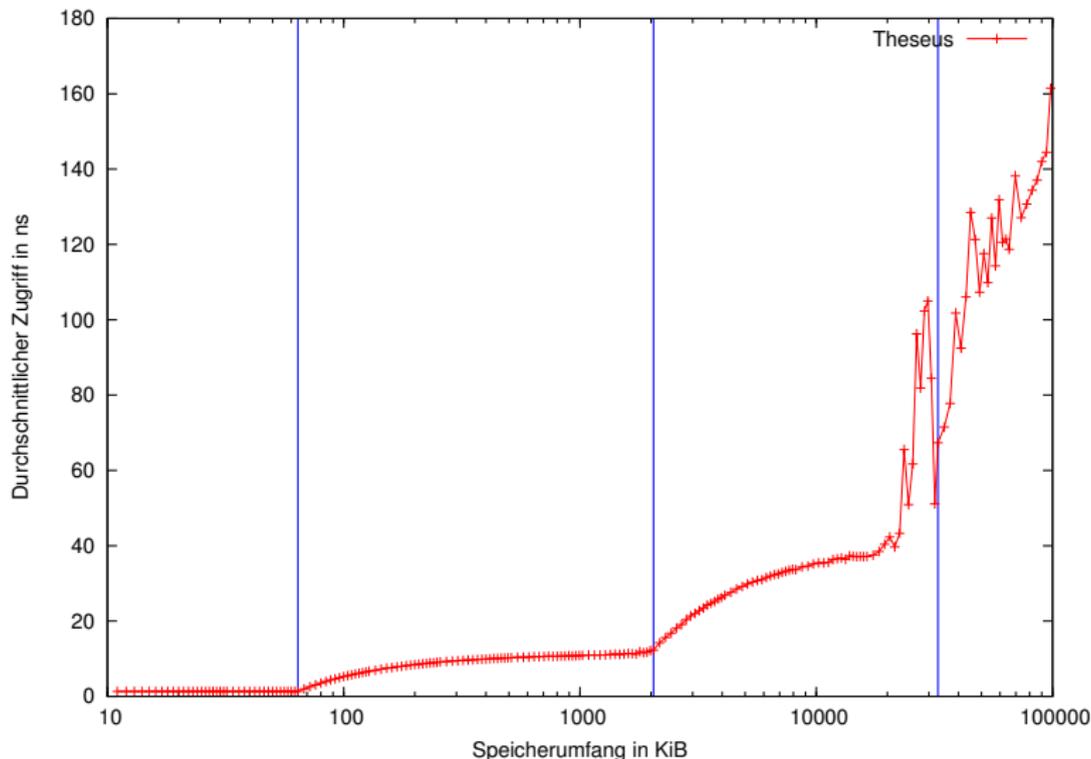
- Der Adressraum eines Prozesses ist eine virtuelle Speicherumgebung, die von dem Betriebssystem mit Unterstützung der jeweiligen Prozessorarchitektur (MMU = *memory management unit*) umgesetzt wird.
- Die virtuellen Adressen eines Prozesses werden dabei in physische Adressen des Hauptspeichers konvertiert.
- Für diese Abbildung wird der Speicher in sogenannte Kacheln (*pages*) eingeteilt.
- Die Größe einer Kachel ist systemabhängig. Auf der Theseus sind es 8 KiB, auf Pacioli und Hochwanner 4 KiB (abzurufen über den Systemaufruf *getpagesize()*).
- Wenn nicht genügend physischer Hauptspeicher zur Verfügung steht, können auch einzelne Kacheln auf Platte ausgelagert werden (*swap space*), was zu erheblichen Zeitverzögerungen bei einem nachfolgendem Zugriff führt.

- Jeder Prozess hat unter UNIX einen eigenen Adressraum.
- Mehrere Prozesse können gemeinsame Speicherbereiche haben (nicht notwendigerweise an den gleichen Adressen). Die Einrichtung solcher gemeinsamer Bereiche ist möglich mit den Systemaufrufen *mmap* (*map memory*) oder *shm\_open* (*open shared memory object*).
- Jeder Prozess hat zu Beginn einen Thread und kann danach (mit dem Initialisieren von *std::thread*-Objekten) weitere Threads erzeugen.
- Alle Threads eines Prozesses haben einen gemeinsamen virtuellen Adressraum. Gelegentlich wird bei Prozessen von Rechtegemeinschaften gesprochen, da alle Threads die gleichen Zugriffsmöglichkeiten und -rechte haben.

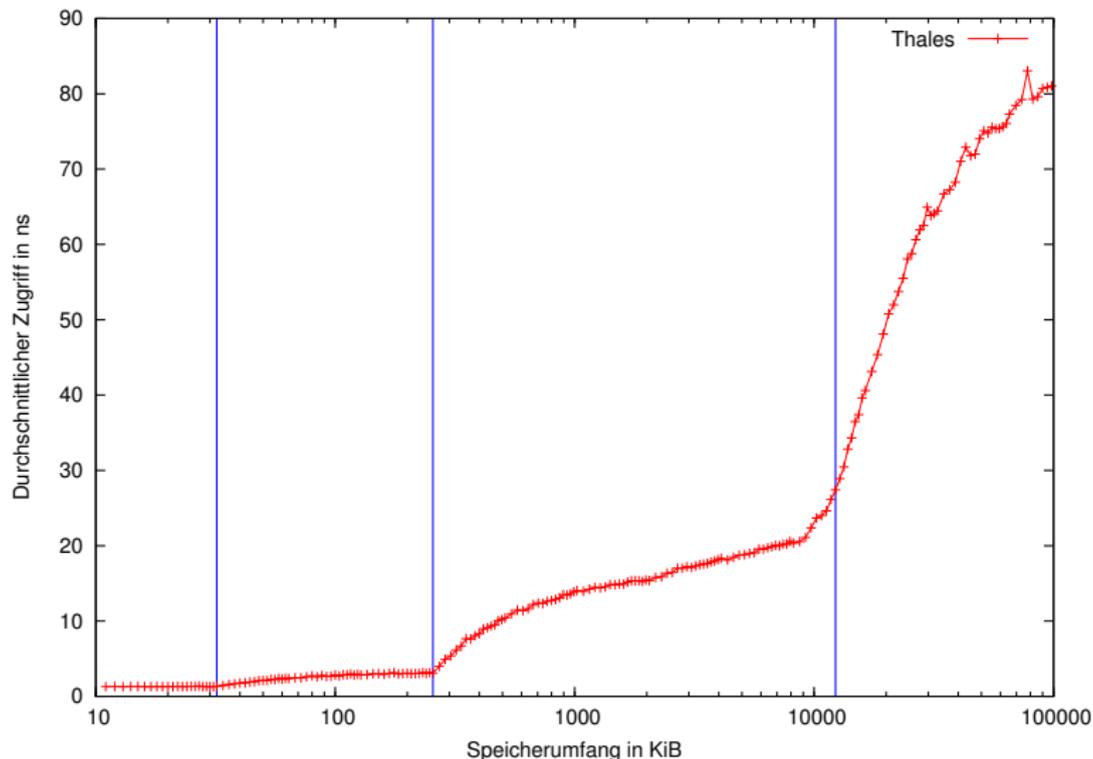
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theseus, die mit Prozessoren des Typs UltraSPARC IV+ ausgestattet ist:

Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	64 KiB	2-3
L2-Cache	2 MiB	um 10
L3-Cache	32 MiB	um 60
Hauptspeicher	32 GiB	um 250

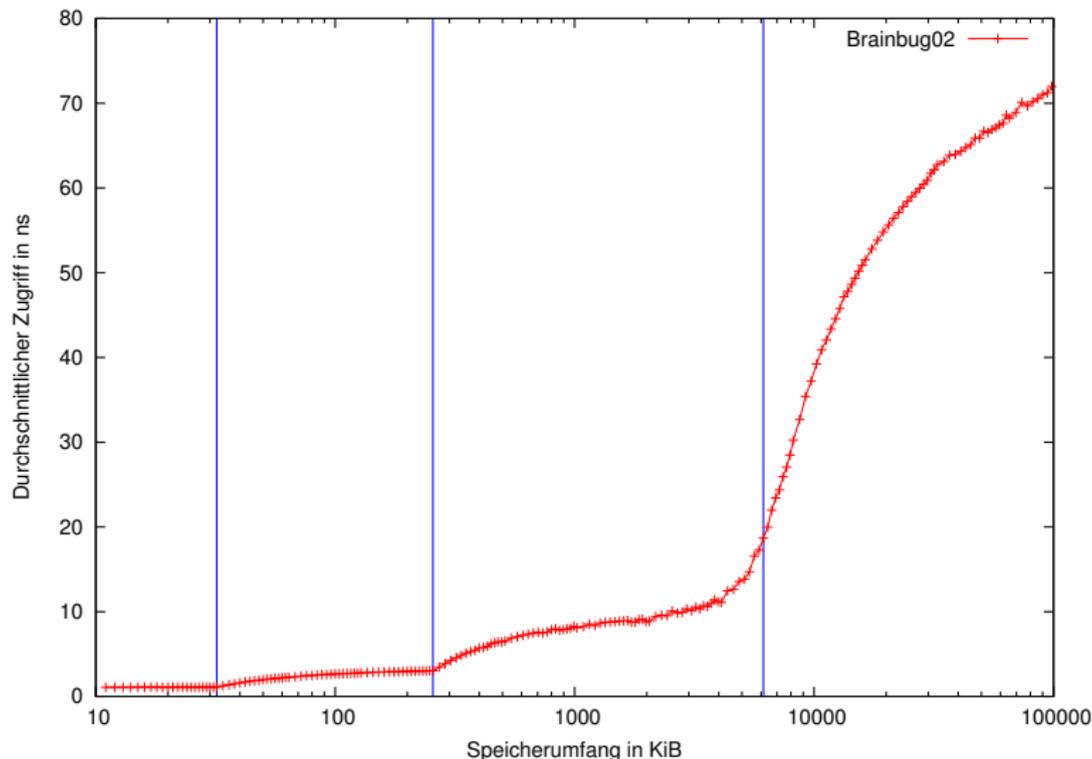




Theseus: 4 UltraSPARC IV+ Prozessoren mit je zwei Kernen  
 Caches: L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)

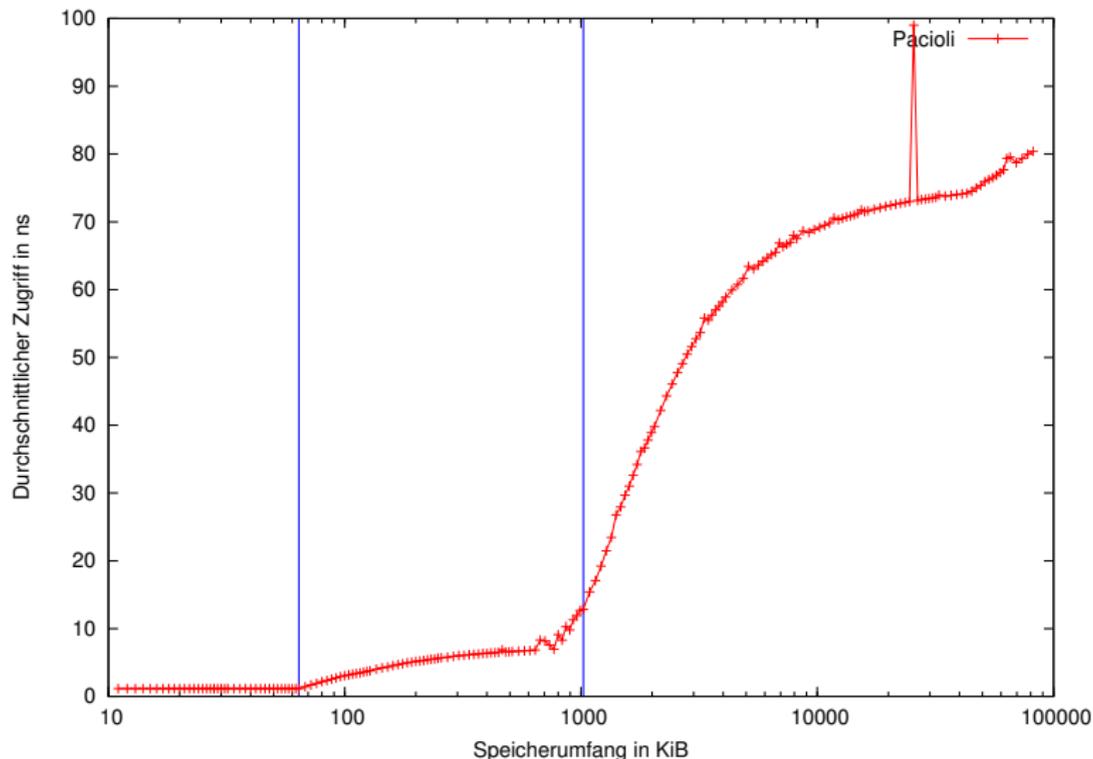


Theseus: 2 Intel X5650-Prozessoren mit je 6 Kernen mit je 2 Threads  
 Caches: L1 (32 KiB), L2 (256 KiB), L3 (12 MiB)



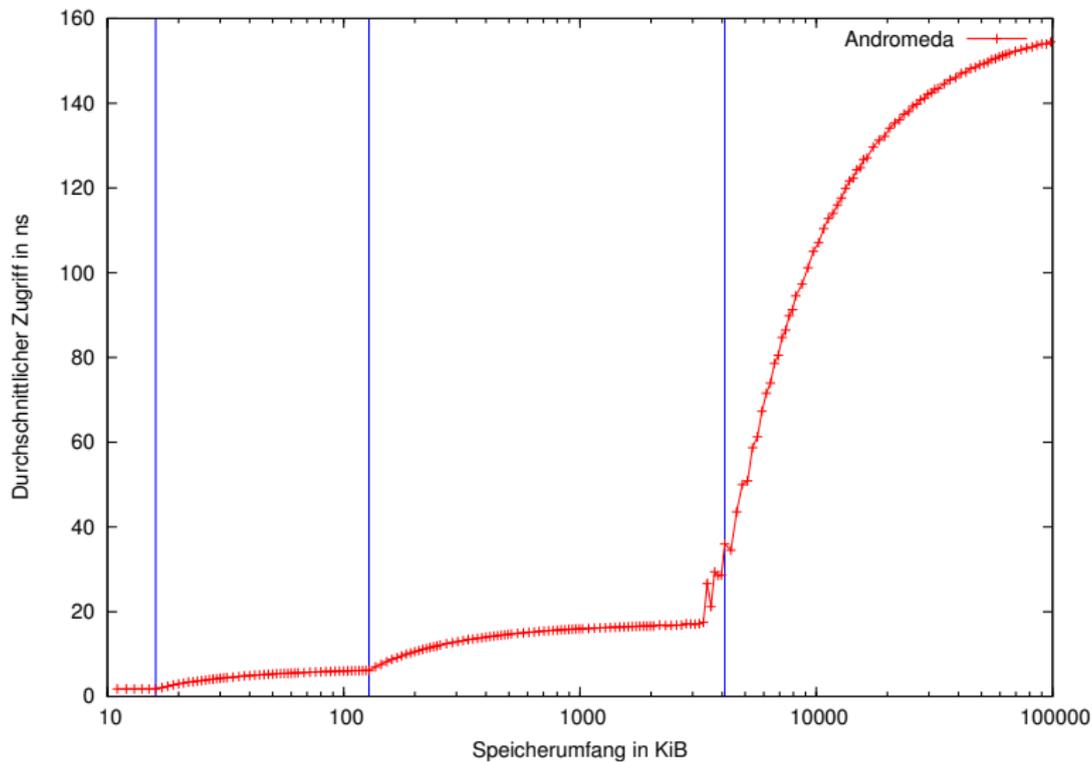
Brainbug02: Intel Quadcore i5-2500S CPU

Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)

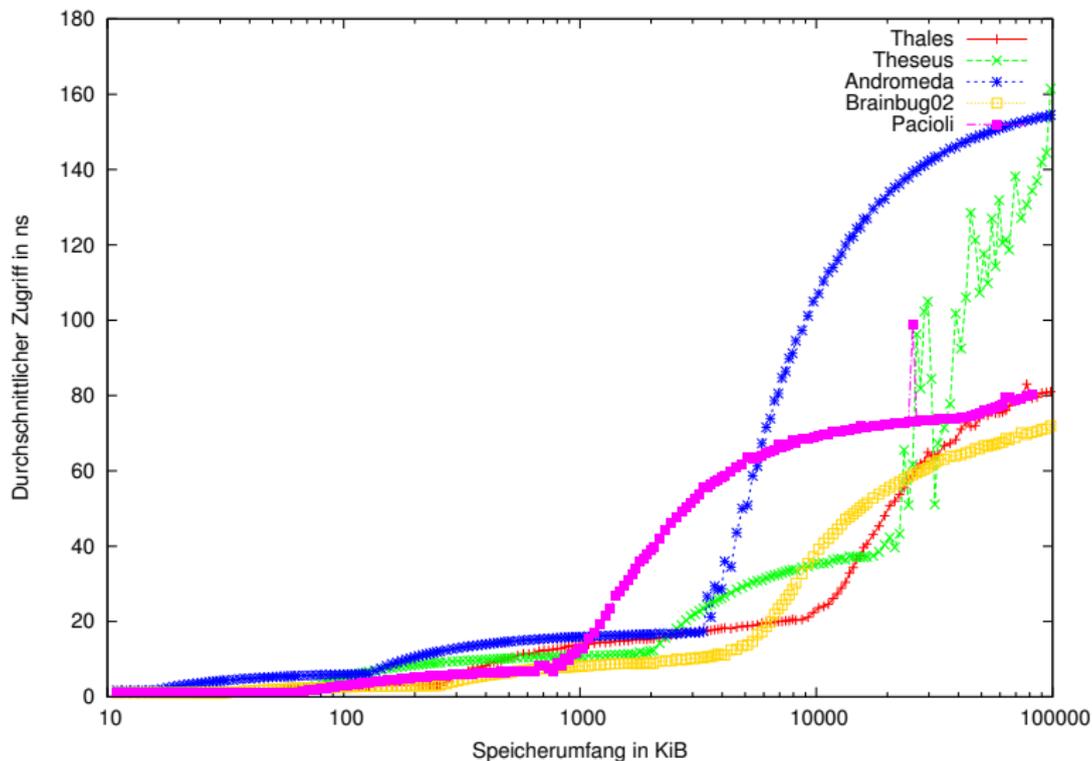


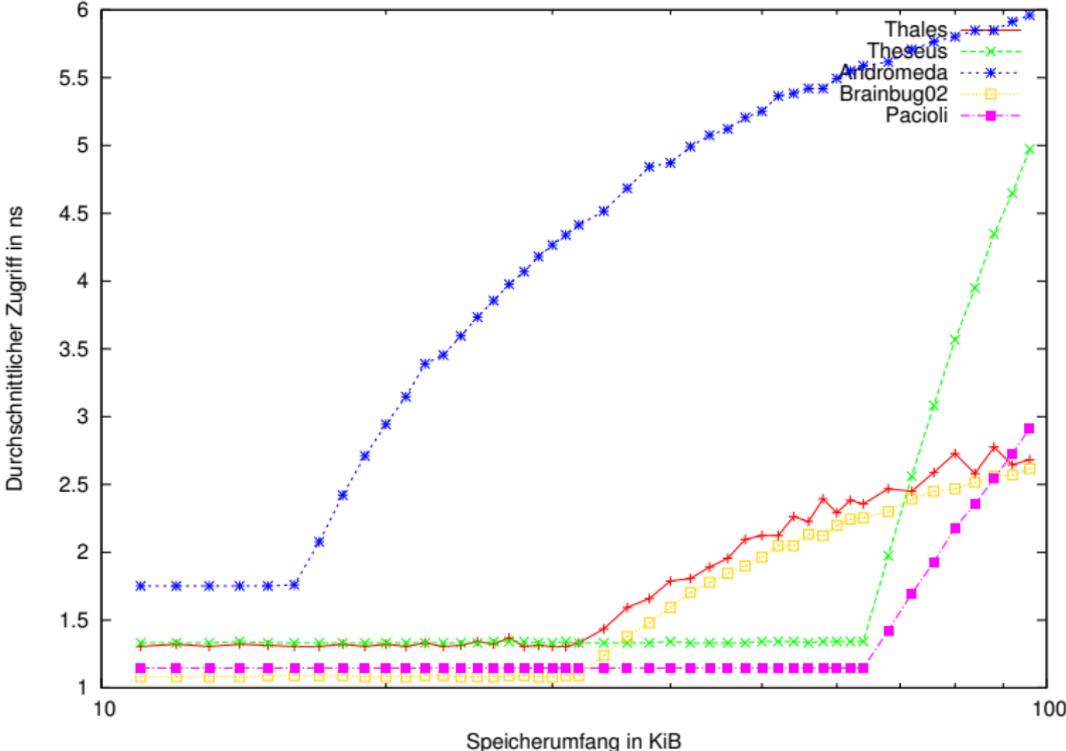
Pacioli: AMD Opteron 252

Caches: L1 (64 KiB), L2 (1 MiB)



Andromeda: 2 SPARC-T4-Prozessoren mit je 8 Kernen mit je 8 Threads  
Caches: L1 (16 KiB), L2 (128 KiB), L3 (4 MiB)





- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf der Theseus sind es beispielsweise 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf  $a[i]$  mit recht hoher Wahrscheinlichkeit auch  $a[i+1]$  zur Verfügung steht.
- Entweder sind Caches vollassoziativ (d.h. jede *cache line* kann einen beliebigen Hauptspeicherabschnitt aufnehmen) oder für jeden Hauptspeicherabschnitt gibt es nur eine *cache line*, die in Frage kommt (*fully mapped*), oder jeder Hauptspeicherabschnitt kann in einen von  $n$  *cache lines* untergebracht werden (*n-way set associative*).

- Diese Optimierungstechnik des Übersetzers bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel:  $x = a[i] + 5; y = b[i] + 3;$   
Hier ist es für den Übersetzer sinnvoll, zuerst die Ladebefehle für  $a[i]$  und  $b[i]$  zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
    for (int i = 0; i < n; ++i) {
        y[i] += alpha * x[i];
    }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch  $n$  gegeben,  $x$  und  $y$  zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
$n$	%o0
$alpha$	%o1 und %o2
$x$	%o3
$y$	%o4

```

        add    %sp, -120, %sp
        cmp    %o0, 0
        st     %o1, [%sp+96]
        st     %o2, [%sp+100]
        ble    .LL5
        ldd    [%sp+96], %f12
        mov    0, %g2
        mov    0, %g1
.LL4:
        ldd    [%g1+%o3], %f10
        ldd    [%g1+%o4], %f8
        add    %g2, 1, %g2
        fmuld  %f12, %f10, %f10
        cmp    %o0, %g2
        fadd   %f8, %f10, %f8
        std    %f8, [%g1+%o4]
        bne    .LL4
        add    %g1, 8, %g1
.LL5:
        jmp    %o7+8
        sub    %sp, -120, %sp
    
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur:  $8 * \text{sizeof}(\text{double}) == 64$ .
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei  $x$  und  $y$  zu laden.
- Der Code ist deswegen so umfangreich, weil
  - ▶ die Randfälle berücksichtigt werden müssen, wenn  $n$  nicht durch 8 teilbar ist und
  - ▶ die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- Bei einem Mehrprozessorsystem hat jede CPU ihre eigenen Caches, die voneinander unabhängig gefüllt werden.
- Problem: Was passiert, wenn mehrere CPUs die gleiche *cache line* vom Hauptspeicher holen und sie dann jeweils verändern? Kann es passieren, dass konkurrierende CPUs von unterschiedlichen Werten im Hauptspeicher ausgehen?

- Die Eigenschaft der Cache-Kohärenz stellt sicher, dass es nicht durch die Caches zu Inkonsistenzen in der Sichtweise mehrerer CPUs auf den Hauptspeicher kommt.
- Die Cache-Kohärenz wird durch ein Protokoll sichergestellt, an dem die Caches aller CPUs angeschlossen sind. Typischerweise erfolgt dies durch Broadcasts über einen sogenannten Snooping-Bus, über den jeder Cache beispielsweise den anderen mitteilt, wenn Änderungen durchgeführt werden.
- Das hat zur Folge, dass bei konkurrierenden Zugriffen auf die gleiche *cache line* einer der CPUs sich diese wieder vom Hauptspeicher holen muss, was zu einer erheblichen Verzögerung führen kann.
- Deswegen sollten konkurrierende Threads nach Möglichkeit Schreibzugriffe auf die gleichen *cache lines* vermeiden.

vectors.cpp

```
void axpy(int n, double alpha,
          const double* x, int incX, double* y, int incY) {
    for (int i = 0; i < n; ++i, x += incX, y += incY) {
        *y += alpha * *x;
    }
}
```

- Die Funktion *axpy* berechnet

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \leftarrow \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} + \alpha \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

- Sowohl  $\vec{x}$  als auch  $\vec{y}$  liegen dabei nicht notwendigerweise als zusammenhängende Arrays vor. Stattdessen gilt  $x_i = x[i*incX]$  (für  $\vec{y}$  analog), so dass auch etwa die Spalte einer Matrix übergeben werden kann, ohne dass deswegen eine zusätzliche (teure) Umkopieraktion notwendig wird.

vectors.cpp

```
class AxyThread {
public:
    AxyThread(int _n, double _alpha, double* _x, int _incX,
              double* _y, int _incY) :
        n(_n), alpha(_alpha), x(_x), incX(_incX),
        y(_y), incY(_incY) {
    }
    void operator()() {
        axpy(n, alpha, x, incX, y, incY);
    }
private:
    int n;
    double alpha;
    double* x; int incX;
    double* y; int incY;
};
```

- Wie gehab werden die Parameter wieder in eine Klasse verpackt.
- Doch wie werden die zu addierenden Vektoren auf die einzelnen Threads aufgeteilt?

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
    assert(n > 0 && nofthreads > 0);
    thread axpy_thread[nofthreads];

    // spawn threads and pass parameters
    int chunk = n / nofthreads;
    int remainder = n % nofthreads;
    int nextX = 0; int nextY = 0;
    for (int i = 0; i < nofthreads; ++i) {
        int len = chunk;
        if (i < remainder) ++len;
        axpy_thread[i] = thread(AxpyThread(len, alpha,
            x + nextX * incX, incX, y + nextY * incY, incY));
        nextX += len; nextY += len;
    }

    // wait for all threads to finish
    for (int i = 0; i < nofthreads; ++i) {
        axpy_thread[i].join();
    }
}
```

```
void mt_axpy(int n, double alpha, double* x, int incX, double* y,
             int incY, int nofthreads) {
    assert(n > 0 && nofthreads > 0);
    thread axpy_thread[nofthreads];

    // spawn threads and pass parameters
    int chunk = n / nofthreads;
    int remainder = n % nofthreads;
    int nextX = 0; int nextY = 0;
    for (int i = 0; i < nofthreads; ++i) {
        int len = chunk;
        if (i < remainder) ++len;
        axpy_thread[i] = thread(AxpyThread(len, alpha,
            x + i * incX, incX * nofthreads,
            y + i * incY, incY * nofthreads));
        nextX += len; nextY += len;
    }

    // wait for all threads to finish
    for (int i = 0; i < nofthreads; ++i) {
        axpy_thread[i].join();
    }
}
```

```
theseus$ time vectors 10000000 1

real    0m0.78s
user    0m0.57s
sys     0m0.16s
theseus$ time bad-vectors 10000000 8

real    0m0.70s
user    0m2.33s
sys     0m0.17s
theseus$ time vectors 10000000 8

real    0m0.44s
user    0m0.66s
sys     0m0.17s
theseus$
```

- Der erste Parameter spezifiziert die Länge der Vektoren  $n = 10^7$ , der zweite die Zahl der Threads. (Dieses Beispiel könnte noch besser skalieren, wenn auch die Initialisierung der Vektoren parallelisiert worden wäre.)
- Die ungünstige alternierende Aufteilung führt dazu, dass Threads um die gleichen *cache lines* konkurrieren, so dass erhebliche Wartezeiten entstehen, die insgesamt den Vorteil der vielen Threads fast vollständig aufhoben.

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \  
> vectors 10000000 8  
time lwp      event      pic0      pic1  
0.302  2 lwp_create      0          0  
0.321  3 lwp_create      0          0  
0.382  4 lwp_create      0          0  
0.382  2 lwp_exit        224        1192  
0.382  3 lwp_exit        143        556  
0.442  5 lwp_create      0          0  
0.442  4 lwp_exit        134        278  
0.504  6 lwp_create      0          0  
0.504  5 lwp_exit        46         642  
0.562  7 lwp_create      0          0  
0.562  6 lwp_exit        128        373  
0.623  8 lwp_create      0          0  
0.623  7 lwp_exit        116        909  
0.664  9 lwp_create      0          0  
0.683  8 lwp_exit        126        400  
0.719  9 lwp_exit        172        486  
0.720  1      exit          4659       5437  
theseus$
```

- Moderne CPUs erlauben die Auslesung diverser Cache-Statistiken. Unter Solaris geht dies mit *cputrack*. Hier liegt die Zahl der L2- und L3-Cache-Invalidierungen wegen Parallelzugriffen bei 5.437, während bei der ungünstigen Aufteilung diese auf 815.276 ansteigt...

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \  
> bad-vectors 10000000 8  
time lwp      event      pic0      pic1  
0.294  2 lwp_create      0         0  
0.301  3 lwp_create      0         0  
0.432  4 lwp_create      0         0  
0.518  5 lwp_create      0         0  
0.518  2 lwp_exit      1076     195702  
0.518  3 lwp_exit      186      1222  
0.692  6 lwp_create      0         0  
0.693  4 lwp_exit      4508     304384  
0.820  7 lwp_create      0         0  
0.821  5 lwp_exit      109      2176  
0.932  8 lwp_create      0         0  
0.932  6 lwp_exit      196      856  
0.947  9 lwp_create      0         0  
0.948  7 lwp_exit      131      959  
1.007  1 tick           3230     646  
1.067  8 tick           276     308152  
1.077  9 tick           25      702  
1.079  8 lwp_exit      369     308404  
1.110  9 lwp_exit      123      893  
1.112  1 exit          10263    815276  
theseus$
```

- Es sind hier nicht alle Threads gleichermaßen betroffen, da jeweils zwei Threads sich einen Prozessor (mit zwei Kernen) teilen, die den L2- und L3-Cache gemeinsam nutzen. (Den L1-Cache gibt es jedoch für jeden Kern extra.)

- OpenMP ist ein seit 1997 bestehender Standard mit Pragma-basierten Spracherweiterungen zu Fortran, C und C++, die eine Parallelisierung auf MP-Systemen unterstützt.
- Pragmas sind Hinweise an den Compiler, die von diesem bei fehlender Unterstützung ignoriert werden können.
- Somit sind alle OpenMP-Programme grundsätzlich auch mit traditionellen Compilern übersetzbar. In diesem Falle findet dann keine Parallelisierung statt.
- OpenMP-fähige Compiler instrumentieren OpenMP-Programme mit Aufrufen zu einer zugehörigen Laufzeitbibliothek, die dann zur Laufzeit in Abhängigkeit von der aktuellen Hardware eine geeignete Parallelisierung umsetzt.
- Die Webseiten des zugehörigen Standardisierungsgremiums mit dem aktuellen Standard finden sich unter <http://www.openmp.org/>

openmp-vectors.cpp

```
void axpy(int n, double alpha, const double* x, int incX,  
         double* y, int incY) {  
#pragma omp parallel for  
    for (int i = 0; i < n; ++i) {  
        y[i*incY] += alpha * x[i*incX];  
    }  
}
```

- Im Unterschied zur vorherigen Fassung der *axpy*-Funktion wurde die **for**-Schleife vereinfacht (nur eine Schleifenvariable) und es wurde darauf verzichtet, die Zeiger *x* und *y* zu verändern.
- Alle für OpenMP bestimmten Pragmas beginnen mit **#pragma omp**, wonach die eigentliche OpenMP-Anweisung folgt. Hier bittet **parallel for** um die Parallelisierung der nachfolgenden **for**-Schleife.
- Die Schleifenvariable ist für jeden implizit erzeugten Thread privat und alle anderen Variablen werden in der Voreinstellung gemeinsam verwendet.

```
Sources := $(wildcard *.cpp)
Objects := $(patsubst %.cpp,%.o,$(Sources))
Targets := $(patsubst %.cpp,%, $(Sources))
CXX := g++
CXXFLAGS := -std=gnu++11 -Ofast -fopenmp
CC := g++
LDFLAGS := -fopenmp
.PHONY: all clean
all: $(Targets)
clean: ; rm -f $(Objects) $(Targets)
```

- Die GNU Compiler Collection (GCC) unterstützt OpenMP für Fortran, C und C++ ab der Version 4.2, wenn die Option „-fopenmp“ spezifiziert wird.
- Der C++-Compiler von Sun berücksichtigt OpenMP-Pragmas, wenn die Option „-xopenmp“ angegeben wird.
- Diese Optionen sind auch jeweils beim Binden anzugeben, damit die zugehörigen Laufzeitbibliotheken mit eingebunden werden.

```
theseus$ time openmp-vectors 100000000

real    0m7.81s
user    0m5.79s
sys     0m1.52s
theseus$ OMP_NUM_THREADS=8 time openmp-vectors 100000000

real    4.1
user    6.8
sys     1.5
theseus$ cd ../pthreads-vectors/
theseus$ time vectors 100000000 8

real    0m4.26s
user    0m6.76s
sys     0m1.54s
theseus$
```

- Mit der Umgebungsvariablen *OMP\_NUM\_THREADS* lässt sich festlegen, wieviele Threads insgesamt durch OpenMP erzeugt werden dürfen.

- Zu parallelisierende Schleifen müssen bei OpenMP grundsätzlich einer der folgenden Formen entsprechen:

$$\text{for } (index = start; index \left\{ \begin{array}{l} < \\ \leq \\ \geq \\ > \end{array} \right\} end; \left\{ \begin{array}{l} index++ \\ ++index \\ index-- \\ --index \\ index += inc \\ index -= inc \\ index = index + inc \\ index = inc + index \\ index = index - inc \end{array} \right\} )$$

- Die Schleifenvariable darf dabei auch innerhalb der **for**-Schleife deklariert werden.

- Per Voreinstellung ist nur die Schleifenvariable privat für jeden Thread.
- Alle anderen Variablen werden von allen Threads gemeinsam verwendet, ohne dass dabei eine Synchronisierung implizit erfolgt. Deswegen sollten gemeinsame Variable nur lesenderweise verwendet werden (wie etwa bei *alpha*) oder die Schreibzugriffe sollten sich nicht ins Gehege kommen (wie etwa bei *y*).
- Abhängigkeiten von vorherigen Schleifendurchläufen müssen entfernt werden. Dies betrifft insbesondere weitere Schleifenvariablen oder Zeiger, die fortlaufend verschoben werden.
- Somit muss jeder Schleifendurchlauf unabhängig berechnet werden.

simpson.cpp

```
double simpson(double (*f)(double), double a, double b, int n) {
    assert(n > 0 && a <= b);
    double value = f(a)/2 + f(b)/2;
    double xleft;
    double x = a;
    for (int i = 1; i < n; ++i) {
        xleft = x; x = a + i * (b - a) / n;
        value += f(x) + 2 * f((xleft + x)/2);
    }
    value += 2 * f((x + b)/2); value *= (b - a) / n / 3;
    return value;
}
```

- *xleft* und *x* sollten für jeden Thread privat sein.
- Die Variable *xleft* wird in Abhängigkeit des vorherigen Schleifendurchlaufs festgelegt.
- Die Variable *value* wird unsynchronisiert inkrementiert.

omp-simpson.cpp

```
double simpson(double (*f)(double), double a, double b, int n) {
    assert(n > 0 && a <= b);
    double value = f(a)/2 + f(b)/2;
    double xleft;
    double x = a;
    double sum = 0;
#pragma omp parallel for \
    private(xleft) \
    lastprivate(x) \
    reduction(+:sum)
    for (int i = 1; i < n; ++i) {
        xleft = a + (i-1) * (b - a) / n;
        x = a + i * (b - a) / n;
        sum += f(x) + 2 * f((xleft + x)/2);
    }
    value += sum;
    value += 2 * f((x + b)/2);
    value *= (b - a) / n / 3;
    return value;
}
```

- Einem OpenMP-Parallelisierungs-Pragma können diverse Klauseln folgen, die insbesondere die Behandlung der Variablen regeln.
- Mit *private(xleft)* wird die Variable *xleft* privat für jeden Thread gehalten. Die private Variable ist zu Beginn undefiniert. Das gilt auch dann, wenn sie zuvor initialisiert war.
- *lastprivate(x)* ist ebenfalls ähnlich zu *private(x)*, aber der Haupt-Thread übernimmt nach der Parallelisierung den Wert, der beim letzten Schleifendurchlauf bestimmt wurde.
- Mit *reduction(+:sum)* wird *sum* zu einer auf 0 initialisierten privaten Variable, wobei am Ende der Parallelisierung alle von den einzelnen Threads berechneten *sum*-Werte aufsummiert und in die entsprechende Variable des Haupt-Threads übernommen werden.
- Ferner gibt es noch *firstprivate*, das ähnlich ist zu *private*, abgesehen davon, dass zu Beginn der Wert des Haupt-Threads übernommen wird.

omp-simpson-explicit.cpp

```
double mt_simpson(double (*f)(double), double a, double b, int n) {
    assert(n > 0 && a <= b);
    double sum = 0;
#pragma omp parallel reduction(+:sum)
    {
        int nofthreads = omp_get_num_threads();
        int nofintervals = n / nofthreads;
        int remainder = n % nofthreads;
        int i = omp_get_thread_num();
        int interval = nofintervals * i;
        int intervals = nofintervals;
        if (i < remainder) {
            ++intervals;
            interval += i;
        } else {
            interval += remainder;
        }
        double xleft = a + interval * (b - a) / n;
        double x = a + (interval + intervals) * (b - a) / n;
        sum += simpson(f, xleft, x, intervals);
    }
    return sum;
}
```

- Grundsätzlich ist es auch möglich, die Parallelisierung explizit zu kontrollieren.
- In diesem Beispiel entspricht die Funktion *simpson* wieder der nicht-parallelisierten Variante.
- Mit **#pragma omp parallel** wird die folgende Anweisung entsprechend dem Fork-And-Join-Pattern parallelisiert.
- Als Anweisung wird sinnvollerweise ein eigenständiger Block verwendet. Alle darin lokal deklarierten Variablen sind damit auch automatisch lokal zu den einzelnen Threads.
- Die Funktion *omp\_get\_num\_threads* liefert die Zahl der aktiven Threads zurück und *omp\_get\_thread\_num* die Nummer des aktuellen Threads (wird von 0 an gezählt). Aufgrund dieser beiden Werte kann wie gehabt die Aufteilung erfolgen.