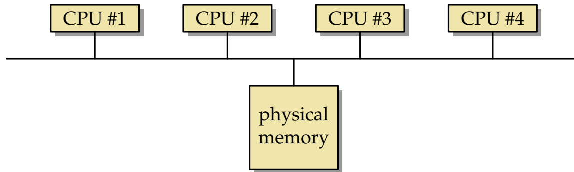
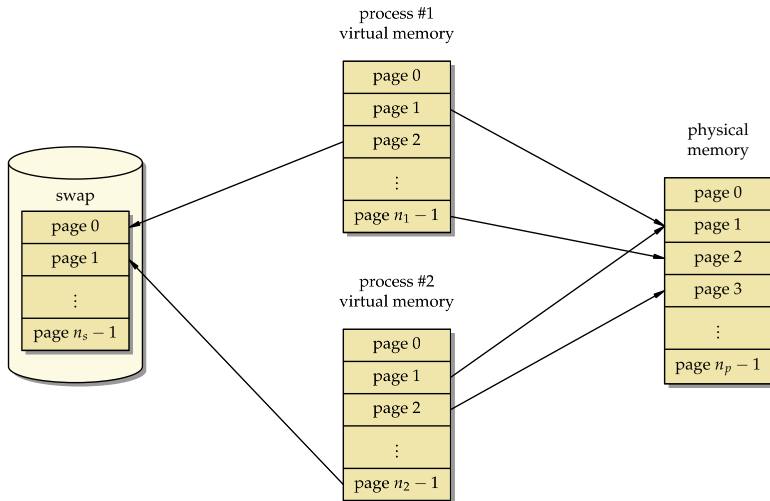


- Sofern gemeinsamer Speicher zur Verfügung steht, so bietet dies die effizienteste Kommunikationsbasis parallelisierter Programme.
- Am häufigsten anzutreffen ist die UMA-Variante (*uniform memory access*, siehe Abbildung). Diese Architektur finden wir auf unseren Rechnern vor.
- Die Synchronisation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

Aus der Sicht unserer Programme ist Speicher eine virtuelle Ressource, auf die über eine Cache-Hierarchie zugegriffen wird:

- Die Hardware-Caches L1, L2 und typischerweise L3: Die CPU greift auf L1 zu, L1 ggf. auf L2 usw.
- Der physisch zur Verfügung stehende Speicher wird als Cache für den virtuellen Speicher betrachtet. Es ist also möglich, dass Inhalte des virtuellen Speichers zeitweilig nur auf einer Platte (*swap space*) liegen.
- Die Zugriffsgeschwindigkeiten auf den einzelnen Ebenen sind extrem unterschiedlich.

- 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 Theon und Livingstone 4 KiB (abzurufen über den Systemaufruf *getpagesize*()).
- Wenn nicht genügend physischer Hauptspeicher zur Verfügung steht, können auch einzelne Kacheln auf eine Platte ausgelagert werden (*swap space*), was zu erheblichen Zeitverzögerungen bei einem nachfolgendem Zugriff führt.


process #1
virtual memory
page 0
page 1
page 2
⋮
page $n_1 - 1$
physical
memory
page 0
page 1
page 2
page 3
⋮
page $n_p - 1$
swap
page 0
page 1
⋮
page $n_s - 1$
process #2
virtual memory
page 0
page 1
page 2
⋮
page $n_2 - 1$

- 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 *pthread_create* bzw. *std*::*thread*) 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.

Zwei Aspekte in Bezug auf Korrektheit und Performance sind besonders relevant:

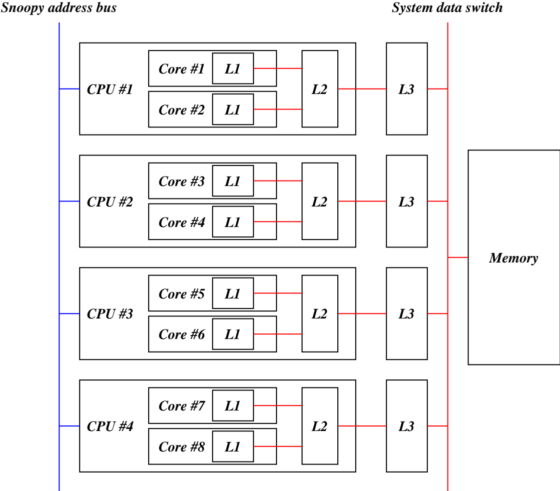
- Wie ist das Verhalten des gemeinsamen Speichers, wenn mehrere Threads konkurrierend und unsychronisiert (d.h. insbesondere ohne Mutex-Variablen) auf die gleichen Datenbereiche zugreifen?
- Welche Auswirkungen hat das Zugriffsverhalten der Threads auf die Performance?

Die Relevanz ergibt sich daraus, dass

- gemeinsamer Speicher sich nicht mehr „intuitiv" verhält bzw. ein erzwungenes „intuitives" Verhalten so restriktiv wäre, dass ein erhebliches Optimierungspotential verloren gehen würde, und dass
- ohne sorgfältige Planung und Koordinierung von Speicherzugriffen in Abhängigkeit der gegebenen Architektur die Laufzeit ein Vielfaches im Vergleich mit einer optimierten Konfiguration betragen kann.

- 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


Snoopy address bus
System data switch
CPU #1
Core #1
L1
Core #2
L1
L2
L3
CPU #2
Core #3
L1
Core #4
L1
L2
L3
Memory
CPU #3
Core #5
L1
Core #6
L1
L2
L3
CPU #4
Core #7
L1
Core #8
L1
L2
L3

Messungen arbeiten mit Speicherbereichen, die als Array von Zeigern organisiert sind, für die folgendes gilt:

- Alle Zeiger verweisen in das gleiche Array,
- mit einem beliebigen Zeiger in dem Array lassen sich alle anderen Zeiger erreichen und
- das gesamte Array wird abgedeckt.

Wenn ein solcher Speicherbereich konfiguriert ist, lassen sich die Speicherzugriffszeiten messen, indem die Zeigerkette n Mal verfolgt wird:

```
void** p = (void**) memory[0];
while (n-- > 0) {
    p = (void**) *p;
}
```

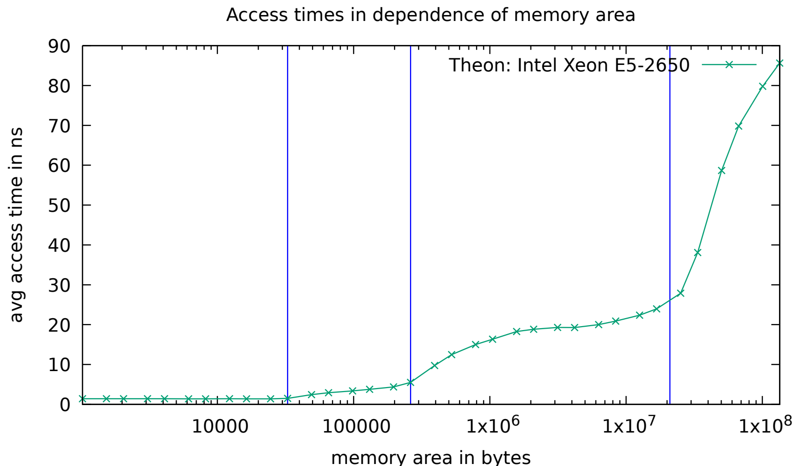
Das Konstrukt `p = (void**)*p` erzwingt die Serialisierung der Speicherzugriffe.

Möglichkeiten, die Zeigerkette in dem Speicherbereich zu organisieren:

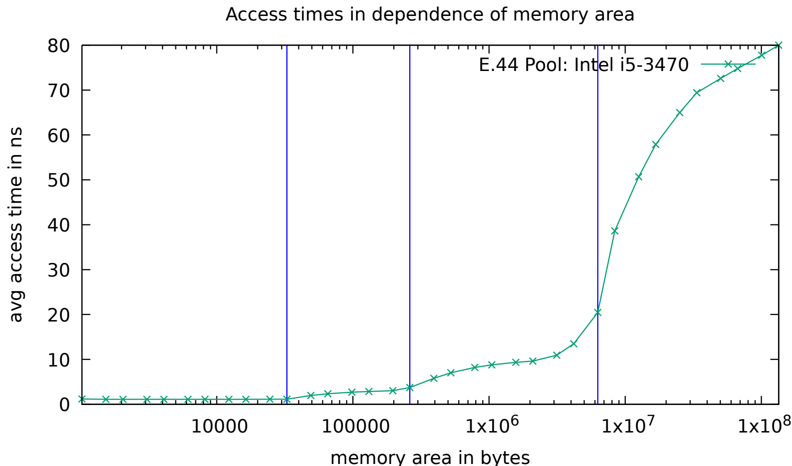
- Zufällig: Bei einer zufälligen Anordnung ist die jeweils nächste Lokation im Speicher für die Maschine nicht vorhersehbar. Entsprechend nähern wir uns damit dem *worst case*-Fall, bei dem der nächste Zeiger nicht in einem der Caches liegt und deswegen aus dem Hauptspeicher geholt werden muss.
- Linear: Die Zeigerkette verwendet weitgehend konstante Abstände. Moderne Prozessoren erkennen solche Zugriffsmuster und können dann bereits auf Verdacht aus dem Speicher in den Cache laden, bevor die entsprechende Lade-Instruktion kommt.
- Fused: Hier arbeiten wir mit n miteinander verwobenen linearen Zugriffsketten. Damit lässt sich testen, wieviele linearen Zugriffsmuster von dem Prozessor parallel verfolgt werden können.

Voraussetzung ist natürlich in jedem Fall, dass der Speicherbereich deutlich größer als die Kapazität der Caches ist.
Werkzeuge hierzu:
`https://github.com/afborchert/pointer-chasing`

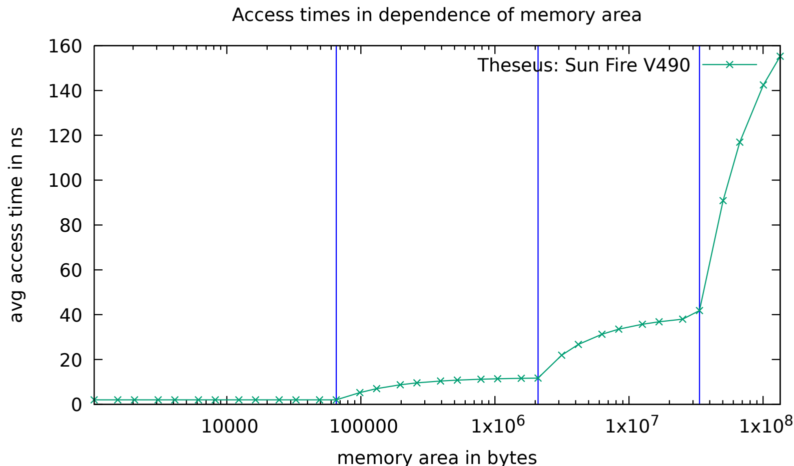
Gemessene Speicherzugriffszeiten: Theon



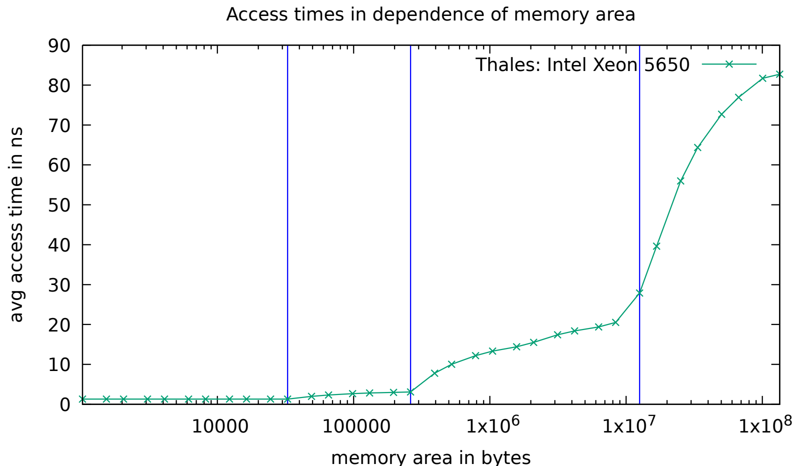
Theon: 1 Intel E5-2650-Prozessor mit 12 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (20 MiB)



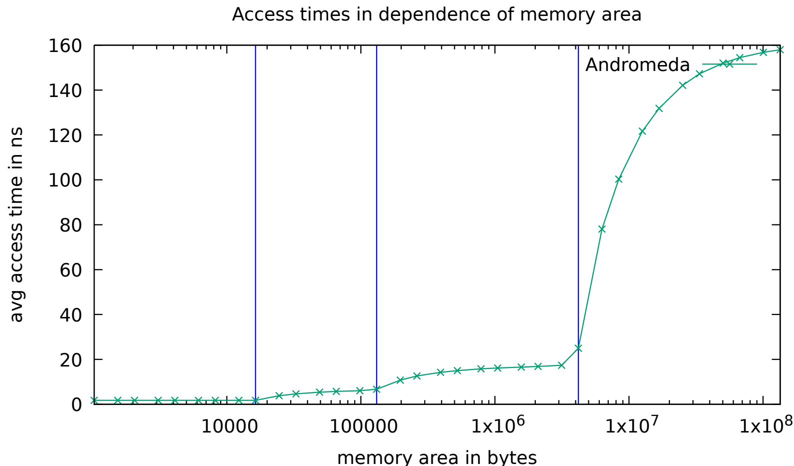
E.44: 1 Intel i5-3470-Prozessor mit 4 Kernen
Caches: L1 (32 KiB), L2 (256 KiB), L3 (6 MiB)



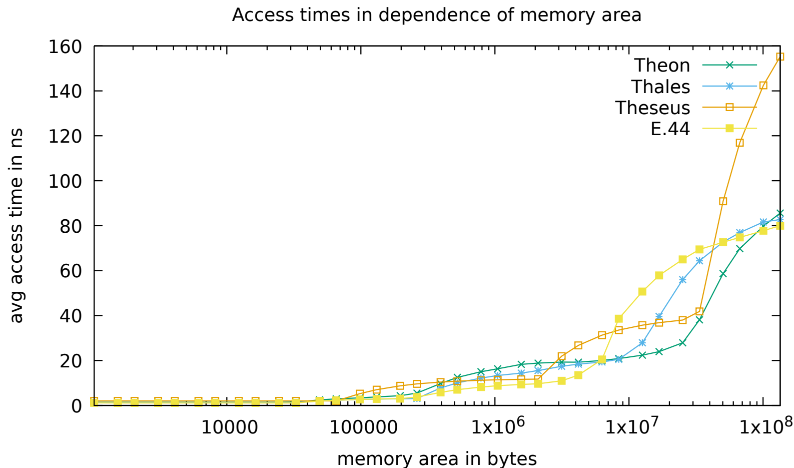
Theseus: 4 UltraSPARC IV+ Prozessoren mit je zwei Kernen
Caches: L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)



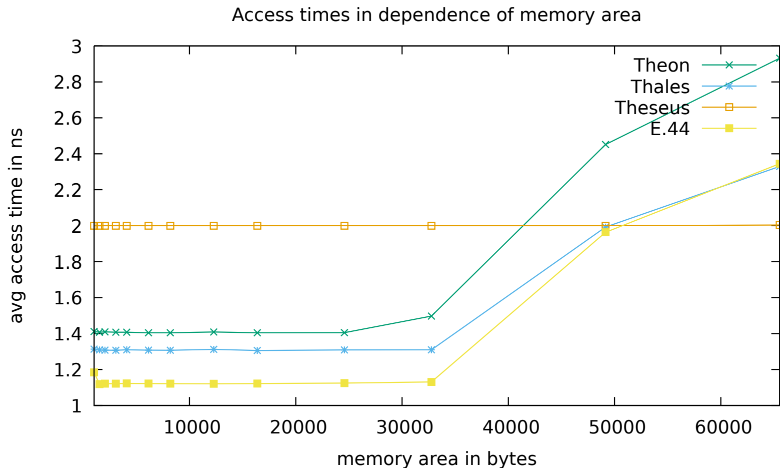
Thales: 2 Intel X5650-Prozessoren mit je 6 Kernen mit je 2 Threads
Caches: L1 (32 KiB), L2 (256 KiB), L3 (12 MiB)



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


Access times in dependence of memory area
avg access time in ns
memory area in bytes
Theon
Thales
Theseus
E.44
0
20
40
60
80
100
120
140
160
10000
100000
1x10^6
1x10^7
1x10^8

Gemessene Speicherzugriffszeiten im L1-Bereich



- 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 unseren Maschinen sind fast durchweg 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*).

- Es gibt keine portable Möglichkeit, die Cache-Konfiguration zu ermitteln.
- Unter Linux gibt es in der Hierarchie unter */sys/devices/system/cpu/cpu0/cache* für jeden der Caches der *cpu0* ein Verzeichnis *index0*, *index1* usw.
- In jedem Cache-Verzeichnis finden sich folgende Dateien:

level	Level des Cache, typischerweise 1, 2 oder 3
type	„Data“, „Instruction“ oder „Unified“
coherency_line_size	Größe einer *cache line*
ways_of_associativity	wieviele *cache lines* kommen in Frage, um einen Abschnitt unterzubringen? Alle *cache lines*, die auf diese Weise zusammengehören, werden einem Set zugeordnet.
number_of_sets	Zahl der Sets (s.o.)
size	Produkt aus *coherency_line_size*, *ways_of_associativity* und *number_of_sets*.

Ziel ist es, die zur Verfügung stehenden CPUs auszulasten, d.h. es sollte möglichst wenig Zeit damit verbracht werden, auf Speicherzugriffe zu warten. Dazu gibt es folgende Ansätze:

- Cache-optimierte Datenstrukturen mit möglichst wenig Indirektionen durch Zeiger. Eine Matrix sollte beispielsweise zusammenhängend im Speicher abgelegt werden und nicht mit Hilfe einer Zeigerliste (wie in Java) realisiert werden.
- Einsatz fortgeschrittener Optimierungstechniken wie *instruction schedulung*, ggf. in Verbindung mit *loop unrolling*.

- 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 den anderen Caches über Änderungen benachrichtigen kann.
- 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

```
template<typename T>
void axpy(std::size_t n, T alpha, const T* x,
      std::ptrdiff_t incX, T* y, std::ptrdiff_t incY) {
   for (std::size_t 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

```
template<typename T>
void mt_axpy(std::size_t n, T alpha, T* x, std::ptrdiff_t incX, T* y,
             std::ptrdiff_t incY, unsigned int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   std::vector<std::thread> axpy_thread(nofthreads);

   // spawn threads and pass parameters
   std::size_t chunk = n / nofthreads;
   unsigned int remainder = n % nofthreads;
   std::ptrdiff_t nextX = 0; std::ptrdiff_t nextY = 0;
   for (unsigned int i = 0; i < nofthreads; ++i) {
      std::size_t len = chunk;
      if (i < remainder) ++len;
      if (len == 0) break; // # of threads > n
      axpy_thread[i] = std::thread([=]() -> void {
         axpy(len, alpha, x + nextX * incX, incX, y + nextY * incY, incY);
      });
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (auto& t: axpy_thread) {
      if (t.joinable()) t.join();
   }
}
```

bad-vectors.cpp

```
template<typename T>
void mt_axpy(std::size_t n, T alpha, T* x, std::ptrdiff_t incX, T* y,
             std::ptrdiff_t incY, unsigned int nofthreads) {
   assert(n > 0 && nofthreads > 0);
   std::vector<std::thread> axpy_thread(nofthreads);

   // spawn threads and pass parameters
   std::size_t chunk = n / nofthreads;
   unsigned int remainder = n % nofthreads;
   std::ptrdiff_t nextX = 0; std::ptrdiff_t nextY = 0;
   for (unsigned int i = 0; i < nofthreads; ++i) {
      std::size_t len = chunk;
      if (i < remainder) ++len;
      if (len == 0) break; // # of threads > n
      axpy_thread[i] = std::thread([=]() -> void {
         axpy(len, alpha,
            x + i * incX, incX * nofthreads,
            y + i * incY, incY * nofthreads);
      });
      nextX += len; nextY += len;
   }

   // wait for all threads to finish
   for (auto& t: axpy_thread) {
      if (t.joinable()) t.join();
   }
}
```

```
theon$ time vectors 10000000 1

real 0m0.280s
user 0m0.084s
sys 0m0.144s
theon$ time vectors 10000000

real 0m0.106s
user 0m0.233s
sys 0m0.634s
theon$ time bad-vectors 10000000

real 0m0.376s
user 0m0.950s
sys 0m2.635s
theon$
```

- Der erste Parameter spezifiziert die Länge der Vektoren $n = 10^7$, der zweite die Zahl der Threads.
- Die ungünstige alternierende Aufteilung führt dazu, dass Threads um die gleichen *cache lines* konkurrieren, so dass erhebliche Wartezeiten entstehen, die zu einer längeren Ausführungszeit führen als bei der Single-Thread-Variante.

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> vectors 10000000 4
   time lwp      event       pic0       pic1
  0.021   2 lwp_create          0          0
  0.061   3 lwp_create          0          0
  0.093   4 lwp_create          0          0
  0.141   5 lwp_create          0          0
  0.142   2   lwp_exit        291       6167
  0.208   3   lwp_exit        170       7292
  0.230   4   lwp_exit        175       5811
  0.230   5   lwp_exit        144       7906
  0.239   6 lwp_create          0          0
  0.281   7 lwp_create          0          0
  0.321   8 lwp_create          0          0
  0.362   9 lwp_create          0          0
  0.364   6   lwp_exit        141       4405
  0.431   7   lwp_exit        153       6961
  0.449   8   lwp_exit        169       2313
  0.449   9   lwp_exit         50       5382
  0.456  10 lwp_create          0          0
  0.491  11 lwp_create          0          0
  0.531  12 lwp_create          0          0
  0.571  13 lwp_create          0          0
  0.696  11   lwp_exit     625166        581
  0.727  10   lwp_exit     625160     159516
  0.727  12   lwp_exit     625164        257
  0.736  13   lwp_exit     625172        404
  0.737   1       exit    2508726     207166
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 207.166, während

```
theseus$ cputrack -c pic0=L2_rd_miss,pic1=L2L3_snoop_inv_sh \
> bad-vectors 10000000 4
   time lwp      event      pic0      pic1
  0.020   2 lwp_create         0         0
  0.061   3 lwp_create         0         0
  0.114   4 lwp_create         0         0
  0.201   5 lwp_create         0         0
  0.427   3   lwp_exit       294    817869
  0.491   2   lwp_exit       177    784073
  0.491   4   lwp_exit       153   1242056
  0.510   5   lwp_exit       172    773822
  0.519   6 lwp_create         0         0
  0.561   7 lwp_create         0         0
  0.601   8 lwp_create         0         0
  0.641   9 lwp_create         0         0
  0.935   7   lwp_exit       156   1115834
  0.935   8   lwp_exit        48   1120283
  0.939   6   lwp_exit       174    928491
  0.939   9   lwp_exit       160       168
  0.945  10 lwp_create         0         0
  0.961  11 lwp_create         0         0
  0.991  12 lwp_create         0         0
  1.011  13 lwp_create         0         0
  1.599  11   lwp_exit   1575955   1218185
  1.599  13   lwp_exit   1200570   1195271
  1.620  10   lwp_exit   2500159   1218183
  1.620  12   lwp_exit   2500152     88961
  1.621   1       exit   7784993  10503359
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

```
theon$ cputrack -c \
> pic0=l2_rqsts.demand_data_rd_miss,pic1=l2_lines_in.i,pic2=l2_lines_out.demand_dirty \
> vectors 10000000 4
   time lwp      event      pic0      pic1      pic2
  0.014   2 lwp_create         0         0         0
  0.030   3 lwp_create         0         0         0
  0.055   4 lwp_create         0         0         0
  0.071   5 lwp_create         0         0         0
  0.071   2   lwp_exit      6947         0    173843
  0.088   3   lwp_exit     14938         0    241217
  0.088   4   lwp_exit     13801         0    227001
  0.106   5   lwp_exit      4573         0    138879
  0.109   6 lwp_create         0         0         0
  0.130   7 lwp_create         0         0         0
  0.150   8 lwp_create         0         0         0
  0.150   6   lwp_exit      5416         0    126920
  0.153   9 lwp_create         0         0         0
  0.171   7   lwp_exit     16675         0    159869
  0.188   9   lwp_exit     11065         0    125569
  0.189   8   lwp_exit     12306         0    127164
  0.194  10 lwp_create         0         0         0
  0.199  11 lwp_create         0         0         0
  0.209  12 lwp_create         0         0         0
  0.209  11   lwp_exit     17736         0     60528
  0.214  13 lwp_create         0         0         0
  0.221  13   lwp_exit     23740         0     72545
  0.222  10   lwp_exit     45862         0     68643
  0.222  12   lwp_exit     93323         0     64846
  0.224   1       exit    282471         0   1593124
theon$
```

```
theon$ cputrack -c \
> pic0=l2_rqsts.demand_data_rd_miss,pic1=l2_lines_in.i,pic2=l2_lines_out.demand_dirty bad-
vectors \
> 10000000 4
   time lwp      event       pic0       pic1       pic2
  0.012   2 lwp_create          0          0          0
  0.051   3 lwp_create          0          0          0
  0.061   4 lwp_create          0          0          0
  0.071   5 lwp_create          0          0          0
  0.188   2   lwp_exit      23118         32     415689
  0.188   3   lwp_exit      23534         30     393471
  0.189   4   lwp_exit      22914         61     398109
  0.189   5   lwp_exit      22495         33     398287
  0.192   6 lwp_create          0          0          0
  0.231   7 lwp_create          0          0          0
  0.272   8 lwp_create          0          0          0
  0.272   6   lwp_exit       2607          0     945222
  0.287   9 lwp_create          0          0          0
  0.287   7   lwp_exit       1261          0    1009765
  0.296   9   lwp_exit       1240          0     967632
  0.309   8   lwp_exit       1190          0     864913
  0.312  10 lwp_create          0          0          0
  0.330  11 lwp_create          0          0          0
  0.330  10   lwp_exit    1362846          0    1040161
  0.332  12 lwp_create          0          0          0
  0.350  13 lwp_create          0          0          0
  0.350  12   lwp_exit    1371179          0     964744
  0.351  11   lwp_exit    1537411          0     988887
  0.373  13   lwp_exit    1034884          0     909211
  0.374   1       exit    5419798        156    9302393
theon$
```