- Bei einer Datenbank verändern Transaktionen den Zustand in einer Weise, die
  - atomar (Ausführung ganz oder überhaupt nicht),
  - konsistent (wenn die Datenbank zuvor konsistent war, bleibt sie es),
  - isoliert (konkurrierende Transaktionen beeinflussen sich nicht gegenseitig) und
  - dauerhaft

  ist (ACID-Prinzip).
- Dieses Konzept wird auch gerne auf den Hauptspeicher übernommen, wobei hier die Dauerhaftigkeit wegfällt.
- Die prinzipielle Idee ist, dass vom optimistischen Fall ausgegangen wird, dass kein Konflikt vorliegt. Wird dennoch ein solcher festgestellt, dann wird eine der betroffenen Transaktionen zurückgerollt und neu gestartet, sobald die andere abgeschlossen ist.
- Von der Nutzerseite ist ein besonderer Vorteil darin zu sehen, dass die Verwendung expliziter Locks wegfällt.

- Prinzipiell sind jetzt bereits einige Architekturen wie die modernen x86-Prozessoren in der Lage, im Rahmen des Pipelinings im Umgang mit bedingten Sprüngen mehrere Varianten parallel spekulativ zu verfolgen und dann mit Verzögerung sich für eine der Varianten zu entscheiden.
- Das lässt sich auch ausdehnen auf Speicherzugriffe, wenn die Cache-Hierarchie einen spekulativen Zustand unterstützt, der im Erfolgsfalle atomar in den Hauptspeicher abgesichert werden kann oder der im Falle eines Abbruchs der Transaktion wieder zurückgerollt werden kann.
- Der Cache kann dann ähnlich wie die Register zumindest partiell als zum lokalen Zustand eines Cores gehörig betrachtet werden.

Sun Microsystems plante dies für den 2005 bis 2009 entwickelten Rock-Prozessor, der die SPARC-Architektur weiterführen sollte:

- Jeder Core unterstützte 2 reguläre Threads und je nach Konfiguration ein oder zwei sogenannte Scout-Threads, die die spekulative Ausführung einer Transaktion übernahmen.
- Im Rahmen einer Transaktion markierte ein Scout-Thread im eigenen L1-Cache veränderte Cache-Lines und führte in einem Commit-Buffer eine Liste der veränderten Cache-Lines.
- Im Erfolgsfalle wurde der Commit-Buffer genutzt, um die veränderten Cache-Lines aus dem L1 atomar im Hauptspeicher zu sichern.
- Wenn ein eine veränderte Cache-Line durch einen anderen Thread invalidiert wurde, kam es zum Abbruch der Transaktion.
- Die Kapazitäten waren wegen der Beschränkung auf den L1 mit 32 KiB begrenzt.

In seiner Supercomputer-Serie Blue Gene führte IBM 2011 bei den Blue-Gene/Q-Systemen ebenfalls Transactional Memory ein:

- Statt dem L1 wird hier der L2 mit 32 MiB verwendet.
- Es werden keine Scout-Threads benötigt. Stattdessen unterstützt der L2-Cache mehrere Versionen für eine Cache-Line und entsprechende atomare Operationen.
- Seit 2015 wurde von IBM die weitere Entwicklung der Blue-Gene-Familie eingestellt.

TSX sind ein Erweiterung der x86-Architektur, die zunächst auf der Haswell- und Broadwell-Architektur eingeführt wurde, dort aber fehlerhaft implementiert war. Benutzbar ist TSX somit ab der 2015 eingeführten Skylake-Architektur.

- Zum Einsatz kommt hier der L1-Cache mit 32 KiB (auf Haswell).
- Wie bei Rock führen Cache-Konflikte zum Abbruch der Transaktion.
- Wenn eine spekulativ veränderte Cache-Line ersetzt werden muss (*cache line eviction*), dann wird die Transaktion ebenfalls abgebrochen.
- Umgesetzt wird dies über x86-Instruktionspräfixe **XACQUIRE** und **XRELEASE** (sind NOPs bei älteren Architekturen, in Kombination mit dem **LOCK**-Präfix, Ausführung zunächst ohne **LOCK**, im Erfolgsfalle Wiederholung mit **LOCK**) bzw. **XBEGIN** und **XEND**, bei denen explizit spezifiziert wird, wie im Falle einer abgebrochenen Transaktion zu reagieren ist, und die mit der **XABORT**-Instruktion einen Abbruch ermöglichen.

- Mit N4514 gibt es einen Vorschlag für eine entsprechende C++-Erweiterung.
- Für C++17 wird sie wohl nicht berücksichtigt, da sie nicht in den aktuellen Draft aufgenommen wurde.
- Eine Berücksichtigung in möglicherweise veränderter Form ist in einer späteren Version wahrscheinlich. Momentan fehlt es noch an genügend Erfahrung.
- g++ bietet ab 6.1 eine entsprechende Unterstützung mit der Option „-fgnu-tm“. Nach meiner Erfahrung führt diese aber noch häufig zu internen Übersetzerfehlern und ist daher noch nicht ausgereift.
- Angeboten werden Synchronisationsblöcke (keine Transaktionen, aber gegenseitiger Ausschluss ohne explizite Locks) und atomare Blöcke, die auch den Abbruch einer Transaktion vorsehen. Letztere haben das Potential mit entsprechenden Hardware-Erweiterungen umgesetzt zu werden, kommen daher aber auch mit einer Reihe von Restriktionen.

- 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/`. Aktuell ist 4.5 – der GCC unterstützt bislang den Standard in der Version 4.0.

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.

Makefile

```
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:

$$\textbf{for}\ (index = start;\ index \left\{ \begin{array}{c} < \\ <= \\ >= \\ > \end{array} \right\} end; \left\{ \begin{array}{l} index{+}{+} \\ {+}{+}index \\ index{-}{-} \\ {-}{-}index \\ index \mathrel{+}= inc \\ index \mathrel{-}= inc \\ index = index + inc \\ index = inc + index \\ index = index - inc \end{array} \right\})$$

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

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];
   }
}
```

- 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.