- 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 5.0 – der GCC unterstützt bislang den Standard in der Version 4.0. Der GCC 9 unterstützt inzwischen teilweise OpenMP 5.0.

openmp-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) {
#pragma omp parallel for
   for (std::size_t 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.

```
theon$ time env OMP_NUM_THREADS=1 openmp-vectors 100000000
time in ms: 145.496

real 0m2.223s
user 0m0.707s
sys 0m1.091s
theon$ time env OMP_NUM_THREADS=4 openmp-vectors 100000000
time in ms: 58.2448

real 0m1.127s
user 0m0.947s
sys 0m1.593s
theon$ cd ../threads-vectors
theon$ time vectors 100000000 4

real 0m1.519s
user 0m0.875s
sys 0m1.641s
theon$
```

- 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

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

simpson4.cpp

```
template<typename T, typename F>
T simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > 0 && a <= b);
   T value = f(a)/2 + f(b)/2;
   T xleft;
   T x = a;
   for (std::size_t 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

```
template<typename F, typename T>
T simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > T() && a <= b);
   T xleft;
   T x = a;
   T sum{};
#pragma omp parallel for \
      private(xleft) \
      lastprivate(x) \
      reduction(+:sum)
   for (std::size_t 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);
   }
   T value = f(a)/2 + f(b)/2 + sum + 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

```
template<typename T, typename F>
T mt_simpson(F&& f, T a, T b, std::size_t n) {
   assert(n > 0 && a <= b);
   T sum{};
#pragma omp parallel reduction(+:sum)
   {
      auto nofthreads = omp_get_num_threads();
      auto nofintervals = n / nofthreads;
      auto remainder = n % nofthreads;
      auto i = omp_get_thread_num();
      auto interval = nofintervals * i;
      auto intervals = nofintervals;
      if (i < remainder) {
         ++intervals;
         interval += i;
      } else {
         interval += remainder;
      }
      auto xleft = a + interval * (b - a) / n;
      auto 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.

- Die bekanntesten numerischen Verfahren zum Auffinden von Nullstellen wie etwa die Bisektion und die Regula Falsi arbeiten nur lokal, d.h. für eine stetige Funktion f wird ein Intervall $[a, b]$ benötigt, für das gilt $f(a) * f(b) < 0$.
- Für das globale Auffinden aller einfachen Nullstellen in einem Intervall (a, b) erweist sich der von Plagianakos et al vorgestellte Ansatz als recht nützlich: V. P. Plagianakos et al: *Locating and computing in parallel all the simple roots of special functions using PVM*, Journal of Computational and Applied Mathematics 133 (2001) 545–554
- Dieses Verfahren lässt sich parallelisieren. Prinzipiell kann das Gesamtintervall auf die einzelnen Threads aufgeteilt werden. Da sich jedoch die Nullstellen nicht notwendigerweise gleichmäßig verteilen, lohnt sich ein dynamischer Ansatz, bei dem Aufträge erzeugt und bearbeitet werden.

- Gegeben sei die zweimal stetig differenzierbare Funktion $f : [a, b] \in \mathbb{R} \to \mathbb{R}$.
- Dann lässt sich die Zahl der einfachen Nullstellen $N_{f,a,b}$ der Funktion f auf dem Intervall (a, b) folgendermaßen bestimmen:

$$N_{f,a,b} = -\frac{1}{\pi}\left[\int_a^b \frac{f(x)f''(x) - f'^2(x)}{f^2(x) + f'^2(x)}\,dx - \arctan\left(\frac{f'(b)}{f(b)}\right) + \arctan\left(\frac{f'(a)}{f(a)}\right)\right]$$

- Da das Resultat eine ganze Zahl ist, lässt sich das Integral numerisch recht leicht berechnen, weil nur wenige Schritte notwendig sind, um die notwendige Genauigkeit zu erreichen.

- Um alle Nullstellen zu finden, wird die Zahl der Nullstellen auf dem Intervall (a, b) ermittelt.
- Wenn sie 0 ist, kann die weitere Suche abgebrochen werden.
- Wenn sie genau 1 ist, dann kann eines der traditionellen Verfahren eingesetzt werden.
- Bei größeren Werten kann das Intervall per Bisektion aufgeteilt werden. Auf jedem der Teilintervalle wird dann rekursiv die gleiche Prozedur angewandt nach dem Teile- und Herrsche-Prinzip.

rootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   if (numOfRoots == 0) return;
   if (numOfRoots == 1) {
      roots->push_back(bisection(a, b, eps)); return;
   }
   Real midpoint = (a + b) / 2;
   unsigned int numOfLeftRoots = get_count(a, midpoint);
   unsigned int numOfRightRoots = get_count(midpoint, b);
   if (numOfLeftRoots + numOfRightRoots < numOfRoots) {
      roots->push_back(midpoint);
   }
   get_roots(a, midpoint, eps, numOfLeftRoots, roots);
   get_roots(midpoint, b, eps, numOfRightRoots, roots);
}
```

- Wenn keine Parallelisierung zur Verfügung steht, wird ein Teile- und Herrsche-Problem typischerweise rekursiv gelöst.

prootfinder.hpp

```
struct Task {
   Task(Real a, Real b, unsigned int numOfRoots) :
      a(a), b(b), numOfRoots(numOfRoots) {
   }
   Task() : a(0), b(0), numOfRoots(0) {
   }
   Real a, b;
   unsigned int numOfRoots;
};
```

- Wenn bei einer Parallelisierung zu Beginn keine sinnvolle Aufteilung durchgeführt werden kann, ist es sinnvoll, Aufträge in Datenstrukturen zu verpacken und diese in einer Warteschlange zu verwalten.
- Dann können Aufträge auch während der Abarbeitung eines Auftrags neu erzeugt und an die Wartschlange angehängt werden.

prootfinder.hpp

```
   // now we are working on task
   if (task.numOfRoots == 0) continue;
   if (task.numOfRoots == 1) {
      Real root = bisection(task.a, task.b, eps);
#pragma omp critical
      roots->push_back(root); continue;
   }
   Real midpoint = (task.a + task.b) / 2;
   unsigned int numOfLeftRoots = get_count(task.a, midpoint);
   unsigned int numOfRightRoots = get_count(midpoint, task.b);
   if (numOfLeftRoots + numOfRightRoots < task.numOfRoots) {
#pragma omp critical
      roots->push_back(midpoint);
   }
#pragma omp critical
   {
      tasks.push_back(Task(task.a, midpoint, numOfLeftRoots));
      tasks.push_back(Task(midpoint, task.b, numOfRightRoots));
   }
```

prootfinder.hpp

```
#pragma omp critical
   {
      tasks.push_back(Task(task.a, midpoint, numOfLeftRoots));
      tasks.push_back(Task(midpoint, task.b, numOfRightRoots));
   }
```

- OpenMP unterstützt kritische Regionen.
- Zu einem gegebenen Zeitpunkt kann sich nur ein Thread in einer kritischen Region befinden.
- Bei der Pragma-Instruktion **#pragma** omp critical zählt die folgende Anweisung als kritische Region.
- Optional kann bei der Pragma-Instruktion in Klammern die kritische Region benannt werden. Dann kann sich maximal nur ein Thread in einer kritischen Region dieses Namens befinden.

prootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   std::list<Task> tasks; // shared

   if (numOfRoots == 0) return;
   tasks.push_back(Task(a, b, numOfRoots));
#pragma omp parallel
   for(;;) {
#pragma omp critical
      if (tasks.size() > 0) {
         task = tasks.front(); tasks.pop_front();
      } else {
         break;
      }
      // process task and possibly generate new tasks
   }
}
```

- Wenn als Abbruchkriterium eine leere Auftragsschlange genommen wird, besteht das Risiko, dass sich einzelne Threads verabschieden, obwohl andere Threads noch neue Aufträge erzeugen könnten.

prootfinder.hpp

```
void get_roots(Real a, Real b, Real eps, unsigned int numOfRoots,
          RootVector* roots) const {
   std::list<Task> tasks; // shared
   if (numOfRoots == 0) return;
   tasks.push_back(Task(a, b, numOfRoots));
#pragma omp parallel
   while (roots->size() < numOfRoots) {
      // fetch next task, if there is any
      // ...

      // now we are working on task
      // ...
   }
}
```

- Alternativ bietet es sich an, die einzelnen Threads erst dann zu beenden, wenn das Gesamtproblem gelöst ist.
- Aber was können die einzelnen Threads dann tun, wenn das Gesamtproblem noch ungelöst ist und es zur Zeit keinen Auftrag gibt?

Lösung mit einer *busy loop*

prootfinder.hpp

```
#pragma omp parallel
   while (roots->size() < numOfRoots) {
      // fetch next task, if there is any
      Task task;
      bool busyloop = false;
#pragma omp critical
      if (tasks.size() > 0) {
         task = tasks.front(); tasks.pop_front();
      } else {
         busyloop = true;
      }
      if (busyloop) {
         continue;
      }

      // now we are working on task
      // ...
   }
```

- Diese Lösung geht in eine rechenintensive Warteschleife, bis ein Auftrag erscheint.

Nachteil einer *busy loop*

- Eine rechenintensive Warteschleife (*busy loop*) nimmt eine Recheneinheit sinnlos ein, ohne dabei etwas Nützliches zu tun.
- Bei Maschinen mit anderen Nutzern oder mehr Threads als zur Verfügung stehenden Recheneinheiten, ist dies sehr unerfreulich.
- Eine Lösung wären Bedingungsvariablen. Aber diese werden von OpenMP nicht unterstützt.
- Eine Lösung zu diesem Problem kam erst mit der Einführung von OpenMP 4.0.

- Beginnend mit OpenMP 4.0 sind einige Erweiterungen hinzugekommen, die auch die Unterstützung des Master/Worker-Patterns vorsehen.
- Um das Master/Worker-Pattern umzusetzen, wird normalerweise **#pragma** omp parallel unmittelbar mit **#pragma** omp single kombiniert, d.h. die nachfolgende Anweisung oder der nachfolgende Block wird nur von einem Thread ausgeführt (dem Master), während die anderen Threads (die Worker) auf Aufträge warten.
- Mit Hilfe von **#pragma** omp task können dann einzelne Anweisungen oder Blöcke an einen Worker delegiert werden. Dies kann in beliebiger dynamischer und rekursiver Form erfolgen. Insbesondere dürfen auch die Worker selbst diese Direktive verwenden und damit neue Aufträge erzeugen.
- Am Ende des **#pragma** omp parallel-Blocks findet implizit eine Synchronisierung statt. Lokale Synchronisierungsblöcke, auch innerhalb eines Worker-Prozesses, sind mit **#pragma** omp parallel möglich.

prootfinder.hpp

```
template<typename OutputIterator>
void get_roots(OutputIterator outit,
      Real a, Real b, Real eps) const {
   unsigned int numOfRoots = get_count(a, b);
   if (numOfRoots > 0) {
      #pragma omp parallel
      #pragma omp single
      get_roots(outit, a, b, eps, numOfRoots);
   }
}
```

- Zu Beginn der Rekursion wird mit **#pragma** omp parallel die Parallelisierung eröffnet, wobei wegen **#pragma** omp single zunächst nur der Master-Thread beginnt und die anderen noch warten – entweder auf das Ende des Blocks oder die Vergebung von Aufträgen.

prootfinder.hpp

```
template<typename OutputIterator>
void get_roots(OutputIterator& outit,
          Real a, Real b, Real eps, unsigned int numOfRoots) const {
   unsigned int numOfRootsFound = 0; // shared
   if (numOfRoots == 0) return;
   if (numOfRoots == 1) {
      Real root = bisection(a, b, eps);
      #pragma omp critical
      { *outit++ = root; ++numOfRootsFound; }
   } else {
      // more than one root in the remaining interval
      // ...
   }
}
```

- In dieser Implementierung ist die private *get_roots*-Methode rekursiv. Sie wird zu Beginn vom Master-Thread aufgerufen, danach aber auch von den Workern.
- Eine explizite Verwaltung der Aufträge findet nicht mehr statt. Kritische Regionen werden daher nur noch für den Output-Iterator benötigt.

prootfinder.hpp

```
// more than one root in the remaining interval
Real midpoint = (a + b) / 2;
unsigned int numOfLeftRoots = 0; // shared
unsigned int numOfRightRoots = 0; // shared
#pragma omp taskgroup
{
   #pragma omp task shared(numOfLeftRoots)
   numOfLeftRoots = get_count(a, midpoint);
   #pragma omp task shared(numOfRightRoots)
   numOfRightRoots = get_count(midpoint, b);
}
// divide and conquer
// ...
```

- Zu Beginn muss die Zahl der Nullstellen im linken und im rechten Teilintervall ermittelt werden.
- Die entsprechenden Aufträge werden hier separat vergeben und durch **#pragma** omp taskgroup synchronisieren wir uns mit der Fertigstellung der beiden Threads gemäß dem Fork-and-Join-Pattern.

prootfinder.hpp

```
if (numOfLeftRoots + numOfRightRoots < numOfRoots) {
   #pragma omp critical
   {
      *outit++ = midpoint; ++numOfRootsFound;
   }
}
#pragma omp task shared(outit)
get_roots(outit, a, midpoint, eps, numOfLeftRoots);
#pragma omp task shared(outit)
get_roots(outit, midpoint, b, eps, numOfRightRoots);
```

- Gemäß dem Teile- und Herrsche-Prinzip wird hier die Aufgabe rekursiv aufgeteilt in die beiden Teilintervalle.
- Alle Variablen, auf die ein **#pragma** omp task erzeugter Auftrag zugreift, sind lokale Kopien – es sei denn, die Variablen werden mit *shared* deklariert. Bei allen Referenzen (wie hier *outit*) ist dies zwingend erforderlich.