- 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.