Parallele Programmierung mit OpenMP

Wolfgang Dautermann

FH Joanneum

Chemnitzer Linuxtage 2008

1 Motivation

2 OpenMP Übersicht

3 Hello World - der erste Code

4 OpenMP-Compilerdirektiven
- Threaderzeugung
- Lastverteilung auf mehrere Threads
- Gültigkeitsbereiche von Daten
- Synchronisation

5 OpenMP Funktionen

6 Umgebungsvariablen

7 Compiler, die OpenMP unterstützen

Parallelisieren – wozu?

- Eine CPU immer schneller machen, gelingt heute nicht mehr, stattdessen werden Dual- und Quadcore CPUs in die Rechner (inzwischen auch in handelsübliche PCs) eingebaut.
- Normalerweise nutzt Software aber nur eine CPU - v.a. bei rechenintensiven Aufgaben eine Verschwendung.
- Software wird standardmässig Step-by-Step abgearbeitet – muss daher parallelisiert werden.

Parallelisierungsmethoden

- Distributed Memory – mehrere einzelne Rechner in einem Cluster.
  Message Passing Interface (MPI)
- Shared Memory – mehrere CPUs in einem Rechner, die auf einen gemeinsamen Speicher zugreifen können.
  OpenMP

- Implizit – automatisch durch den Compiler, keine speziellen Anweisungen, Funktionen, etc. notwendig.
- Explizit – macht der Programmierer durch spezielle Anweisungen, Funktionen, etc.

OpenMP - Übersicht

- Explizite shared Memory Parallelisierung
- Erweiterung für existierende Programmiersprachen (C/C++/Fortran)
  - Hauptsächlich über Compiler Direktiven
  - Einige wenige Library-Funktionen
- Inkrementelle Parallelisierung (Parallelisierung eines existierenden seriellen Programms).
- Schwerpunkt: Parallelisierung von Loops.

OpenMP – Fork-Join Ausführungsmodell

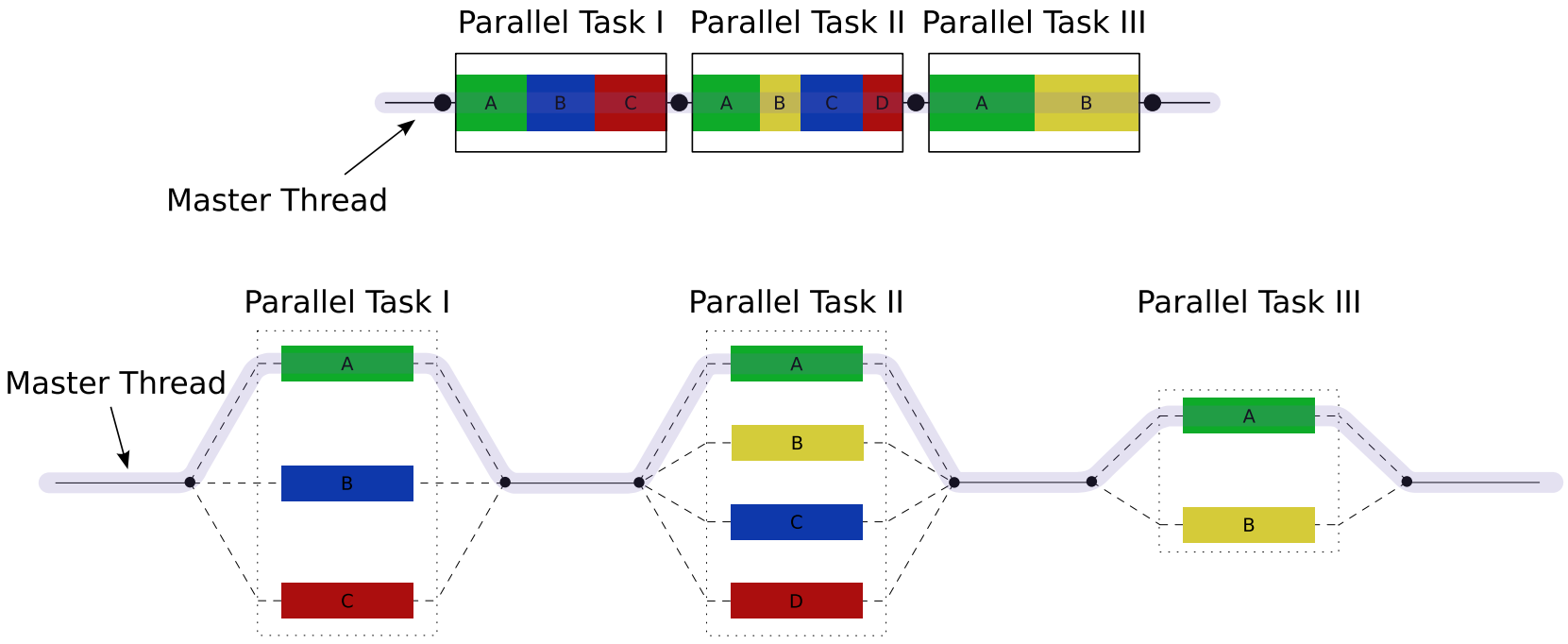


Abbildung: Fork-join Modell von OpenMP

OpenMP Architecture Review Board

(... von der OpenMP Homepage)

> *The OpenMP Architecture Review Board or the OpenMP ARB or just the ARB, is the non-profit corporation that owns the OpenMP brand, oversees the OpenMP specification and produces and approves new versions of the specification.*

Etliche Firmen sind darin vertreten $\Rightarrow$ herstellerunabhängiger Standard zur Parallelprogrammierung

- Permanent Members of the ARB:
  AMD, Cray, Fujitsu, HP, IBM, Intel,Microsoft, NEC, The Portland Group, SGI, Sun
- Auxiliary Members of the ARB:
  ASC/LLNL, cOMPunity, EPCC, NASA, RWTH Aachen

Ein Hello World Programm mit OpenMP (C)

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(void)
{
  int i;
#pragma omp parallel for
  for (i = 0; i < 4; ++i)
  {
    int id = omp_get_thread_num();
    printf("Hello␣World␣from␣thread␣%d\n", id);
    if (id==0)
      printf("There␣are␣%d␣threads\n", omp_get_num_threads());
  }
  return 0;
}
```

Ein Hello World Programm mit OpenMP (Fortran)

```
      PROGRAM HELLO
      USE omp_lib
      INTEGER ID, NTHRDS, I
C$OMP PARALLEL DO PRIVATE(ID,I,NTHRDS)
      DO I=1,4
        ID = OMP_GET_THREAD_NUM()
        PRINT *, 'HELLO WORLD FROM THREAD', ID
        IF ( ID .EQ. 0 ) THEN
          NTHRDS = OMP_GET_NUM_THREADS()
          PRINT *, 'THERE ARE', NTHRDS, 'THREADS'
        END IF
      END DO
C$OMP END PARALLEL DO
      END
```

Compilerdirektiven

OpenMP wird hauptsächlich über Compilerdirektiven definiert.

Direktivenformat in C/C++

```
#pragma omp direktivenname [klauseln ...]
```

Direktivenformat in Fortran

```
C234567
!$OMP direktivenname [klauseln ...] [! Kommentar]
```

(in Fortran sind auch andere *Sentinels* erlaubt[1] (`C$OMP`, `*$OMP`).

[1] nur in fixed source form

Compilerdirektiven II

- für nicht OpenMP-fähige Fortran Compiler erscheinen diese Direktiven als Kommentar.
- nicht OpenMP-fähige C-Compiler ignorieren unbekannte Direktiven:

```
$ gcc -Wall test.c  # (alte gcc Version (< 4.2))
test.c: In function 'main':
test.c:12: warning: ignoring #pragma omp parallel
$
```

⇒ Werden OpenMP Compilerdirektiven in ein Programm eingefügt werden, kann das Programm weiterhin von jedem (auch nicht OpenMP-fähigen) Compiler compiliert werden.

Bedingte Compilierung

- C/C++: Das Makro _OPENMP wird definiert.

```
# ifdef _OPENMP
/* Openmp spezifischer Code, z.B. */
nummer = omp_get_thread_num() ;
#endif
```

- Fortran: OpenMP erlaubt die bedingte Compilierung in Fortran.

```
C234567
!$     NUMMER = OMP_GET_THREAD_NUM()
```

Erscheint als Kommentar, wenn OpenMP nicht aktiviert ist (bzw. bei einem nicht OpenMP-fähigen Compiler), wird normal compiliert, wenn OpenMP aktiviert ist.

omp parallel

Erzeugen zusätzlicher Threads, die Arbeit wird von allen Threads ausgeführt. Der originale Thread (master thread) bekommt die Thread ID 0.

- C/C++:

```
#pragma omp parallel [klauseln]
   /* strukturierter Block (keine gotos...) */
```

- Fortran:

```
!$OMP PARALLEL [klauseln]
   block
!$OMP END PARALLEL
```

Arbeitsaufteilung zwischen den Threads I

Loops: omp for (C/C++)

Die Arbeit wird unter den Threads aufgeteilt – (Beispiel: bei zwei Threads bearbeitet einer die Schleife von $1 \cdots (\frac{N}{2} - 1)$, der zweite von $\frac{N}{2} \cdots N$)[2]

```
#pragma omp parallel [klauseln ...]
#pragma omp for [klauseln ...]
    for (i=0;i<N;i++)
        a[i]= i*i;
```

Dies kann auch zusammengefasst werden (`omp parallel for`):

```
#pragma omp parallel for [klauseln ...]
    for (i=0;i<N;i++)
        a[i]= i*i;
```

[2] Das Verhalten kann durch Verwendung der `schedule()`-Klausel (`static`, `dynamic`, `guided`, `runtime`) beeinflusst werden...

Arbeitsaufteilung zwischen den Threads I

Loops: omp do (Fortran)

Die Arbeit von DO-Loops wird von allen Threads gemeinsam bearbeitet.

```
!$OMP PARALLEL [klauseln...]
!$OMP DO [klauseln...]
      DO I=1,N
         A(I) = I*I
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Auch hier wieder möglich:

```
!$OMP PARALLEL DO [klauseln...]
      [...]
!$OMP END PARALLEL DO
```

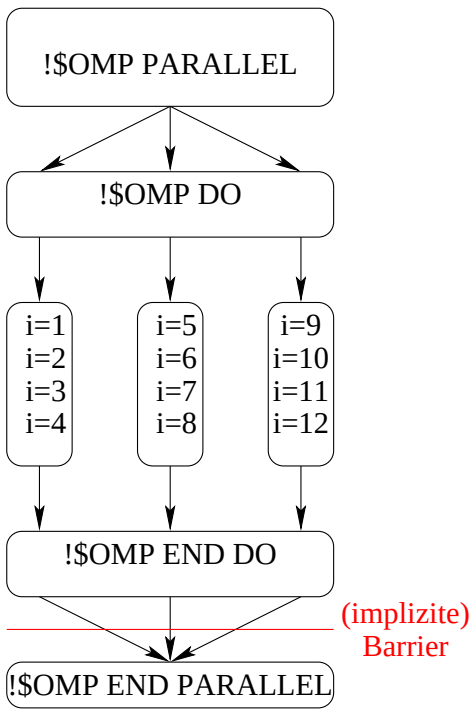


Abbildung: (par.) DO Schleife

Arbeitsaufteilung zwischen den Threads II

Parallele Sections (C/C++)

Die Arbeit wird unter den Threads aufgeteilt. Jeder bearbeitet eine Section.

```
#pragma omp parallel [klauseln ...]
#pragma omp sections [klauseln ...]
{
#pragma omp section
    [...Programmteil A läuft parallel zu B...]
#pragma omp section
    [...Programmteil B läuft parallel zu A...]
}
```

Auch hier kann man das kombinieren:

```
#pragma omp parallel sections [klauseln ...]
```

Arbeitsaufteilung zwischen den Threads II

Parallele Sections (Fortran)

Die Arbeit wird unter den Threads aufgeteilt. Jeder bearbeitet eine Section.

```
!$OMP PARALLEL [klauseln ...]
!$OMP SECTIONS [klauseln ...]
!$OMP SECTION
    [...Programmteil A läuft parallel zu B...]
!$OMP SECTION
    [...Programmteil B läuft parallel zu A...]
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Auch hier kann man das kombinieren:

```
!$OMP PARALLEL SECTIONS [klauseln ...]
```

Arbeitsaufteilung zwischen den Threads III

Parallel Workshare (nur Fortran)

Für Operationen mit Arrays, z.B.

```
      REAL, DIMENSION (100,100) :: A, B, C
      [...]
!$OMP PARALLEL WORKSHARE
      A = B + C
!$OMP END PARALLEL WORKSHARE
```

Gültigkeitsbereiche von Daten - shared(), private(), ...

Achtung: Häufige Fehlerquelle!

Klauseln, die bei OpenMP-Direktiven angegeben werden, regeln wie Variablen zwischen den Threads ausgetauscht werden...

- `shared()`: Die Daten werden unter allen parallelen Regionen geshared, d.h. sie sind in allen Threads les- und schreibbar. Wenn ein Thread eine Variable ändert, ist sie auch in allen anderen Threads modifiziert. Default: Alle Variablen sind `shared()` – ausser Schleifenvariablen bei `OMP DO`/`omp for`.
- `private()`: Jeder Thread bekommt eine private Kopie der Variablen, diese wird nicht initialisiert. Default: Nur Schleifenvariablen sind privat.
- `default(shared|private|none)`: legt den Defaultwert fest. `none`: Jede Variable muss `shared()` oder `private()` deklariert werden.

Gültigkeitsbereiche von Daten II

- `firstprivate()`: wie `private()`, aber alle Kopien werden mit dem Wert der Variablen vor der parallelen Schleife/Region initialisiert.
- `lastprivate()`: Die Variable wird nach der parallelen Schleife/Region mit dem Wert aus dem Thread initialisiert, der (sequentiell) am spätesten drangekommen wäre.

Gültigkeitsbereiche von Daten II

reduction() – Aufsummieren, etc.

```
a = 0 ; b = 0 ;
#pragma omp parallel for private(i) shared(x, y, n) \
                         reduction(+: a, b)
for (i=0; i<n; i++) {
      a = a + x[i] ;
      b = b + y[i] ;
}
```

Eine lokale Kopie jeder Variable in reduction() wird erzeugt und abhängig vom Operator initialisiert (z.B. 0 für ”+”). Wenn dann (z.B.) 3 Threads je ein Drittel der Schleife bearbeiten, werden am Ende die Teilsummen aller Threads noch aufsummiert.

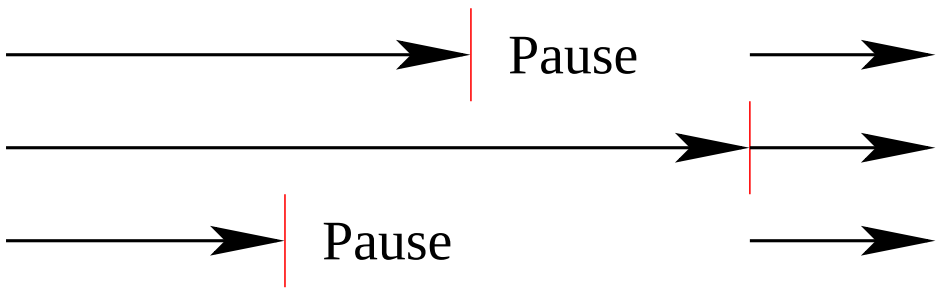
Synchronisation

Barrier – Alle Threads warten, bis alle diesen Punkt erreicht haben

```
#pragma omp barrier   (C/C++)          !$OMP BARRIER   (Fortran)
```

z.B. bei zwei parallelisierten for-Schleifen

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i < N; i++)  a[i] = b[i] + c[i];
#pragma omp barrier
#pragma omp for
  for (i=0; i < N; i++)  d[i] = a[i] + b[i];
}
```



Synchronisation II

Critical/Atomic – alle Threads führen den Code aus, aber nur einer zu einem Zeitpunkt.

```
#pragma omp critical [name]
{ code-block }
#pragma omp atomic [name]
    statement

!$OMP CRITICAL [name]
      code-block
!$OMP END CRITICAL
!$OMP ATOMIC [name]
      statement
```

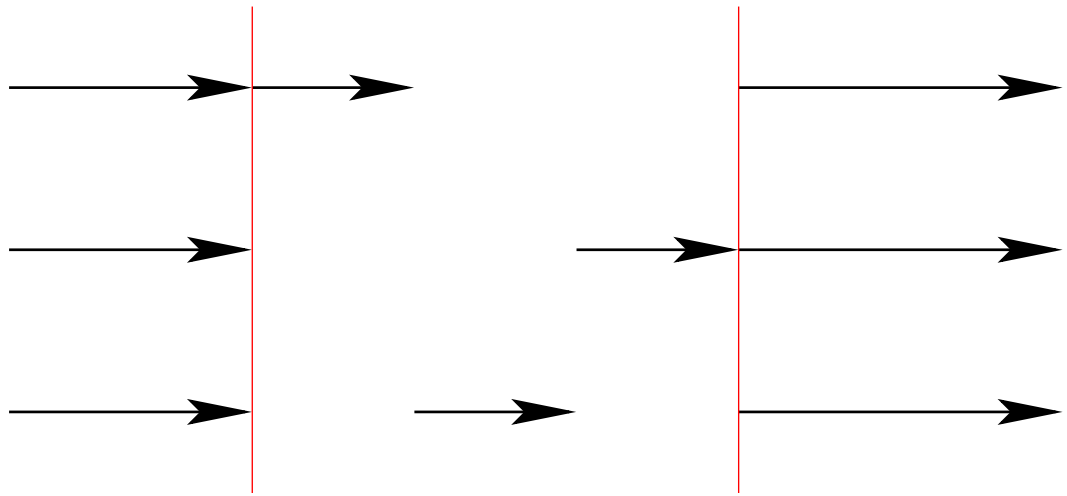


Abbildung: CRITICAL Region

Alle Threads warten am Beginn der `critical` Region, bis kein anderer Thread eine `critical` Region mit demselben Namen ausführt. `critical` Regionen ohne Namen werden alle demselben (nicht spezifizierten) Namen zugeordnet.

Synchronisation III

Critical – Codebeispiel

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main() {
    double a[1000000];
    int i;
    #pragma omp parallel for
    for (i=0; i<1000000; i++) a[i]=i;
    double summe = 0;
    #pragma omp parallel for shared (summe) private (i)
    for ( i=0; i < 1000000; i++) {

        summe = summe + a[i];
    }
    printf("summe=%lf\n",summe);
}
```

Summiert (ähnlich wie reduction()) die Elemente des Arrays auf.
Was ist das Problem?

Synchronisation III

Critical – Codebeispiel (korrekt)

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main() {
    double a[1000000];
    int i;
    #pragma omp parallel for
    for (i=0; i<1000000; i++) a[i]=i;
    double summe = 0;
    #pragma omp parallel for shared (summe) private (i)
    for ( i=0; i < 1000000; i++) {
        #pragma omp critical (summierung)
        summe = summe + a[i];
    }
    printf("summe=%lf\n",summe);
}
```

Bei der Summierung kann eine Race-Condition auftreten. Diese muss mit `#pragma omp critical` vermieden werden.

Synchronisation IV

Nur ein Thread soll eine Region ausführen – z.B. für I/O

```
#pragma omp master {
  [Code der nur einmal
  (nur vom Master Thread) ausgeführt wird]
}

#pragma omp single {
  [Code der nur einmal
  (nicht unbedingt vom Master Thread) ausgeführt wird]
}
```

Synchronisation V

Flush, Ordered

`FLUSH`: sorgt für eine konsistente Sicht auf den Speicher.

```
#pragma omp flush(variablen)
```

```
!$OMP FLUSH(variablen)
```

`ORDERED`: Legt fest, dass die Reihenfolge der Ausführung der Iterationen des betreffenden Blocks die gleiche wie bei serieller Programmausführung sein muss – zulässig nur innerhalb einer for-Schleife

```
#pragma omp for ordered
    { ... } ;
```

OpenMP-Funktionen

OpenMP stellt div. Library Funktionen zur Verfügung.

- Settings mit diesen Funktionen haben Vorrang vor den entsprechenden Umgebungsvariablen.
- Empfehlung: Verwendung nur mit `#ifdef _OPENMP` bzw. bedingter Compilierung (Fortran), damit der Code auch ohne OpenMP lauffähig bleibt.
- C/C++: `#include <omp.h>`
- Fortran: `USE omp_lib`

OpenMP-Funktionen II

- Threading
  - `int omp_get_num_threads()`
  - `int omp_get_thread_num()`
  - `int omp_in_parallel()`
  - `void omp_set_num_threads(int)`
  - `...`
- Locking
  - `void omp_init_lock(omp_lock_t*)`
  - `void omp_destroy_lock(omp_lock_t*)`
  - `void omp_set_lock(omp_lock_t*)`
  - `void omp_unset_lock(omp_lock_t*)`
  - `int omp_test_lock(omp_lock_t*)`
  - `...`
- `...`

Umgebungsvariablen

- `OMP_NUM_THREADS=integer`
  Maximale Anzahl der OpenMP-Threads
- `OMP_SCHEDULE=''kind[,chunk_size]''`
  (static, dynamic, guided) – betrifft das Aufteilen der Arbeit in Loops:
  - `static`: jeder Thread bekommt statisch gleich viele Iterationen zugeteilt
  - `dynamic`: jeder Thread holt sich neue Arbeit ab, sobald er arbeitslos ist
  - `guided`: wie `dynamic`, nur werden die zugewiesenen Pakete laufend kleiner
- `OMP_DYNAMIC=''true|false''`
  (kann die Anzahl der Threads dynamisch modifiziert werden?)
- `OMP_NESTED=''true|false''`

Compiler - gcc ab Version 4.2

. . . ist möglicherweise (noch) nicht bei der eigenen Distribution dabei, aber selbstcompilieren ist kein grosses Kunststück:
Download der akutellen Release von `ftp://ftp.gnu.org/gnu/gcc/` (oder einem Mirror, z.B. `ftp://gd.tuwien.ac.at/gnu/gcc/releases/`

Selbstcompilieren von gcc

```
tar xvjf gcc-4.2.3.tar.bz2
cd gcc-4.2.3
./configure --prefix=/opt/gcc-4.2.3 \
            --enable-languages=fortran,c,c++
make # das dauert, Zeit für eine Tasse Kaffee...
make install # als root
```

gcc - Verwenden des selbstcompilierten GCC

Verwendung des selbstcompilierten GCC

```
$ /opt/gcc-4.2.3/bin/gcc -Wall -fopenmp helloworld.c
$ export LD_LIBRARY_PATH=/opt/gcc-4.2.3/lib64
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World from thread 3
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
There are 4 threads
$
```

gcc - Verwenden des selbstcompilierten GCC (mit `-rpath`)

Die Linker-option `-rpath` codiert den Suchpfad für Libraries in das Binary.

- Vorteil: läuft (am eigenen System) ohne `$LD_LIBRARY_PATH` Hacks...
- Nachteil: läuft (möglicherweise) nur am eigenen System...

Verwendung des selbstcompilierten GCC

```
$ /opt/gcc-4.2.3/bin/gcc -Wall -fopenmp \
  -Wl,-rpath=/opt/gcc-4.2.3/lib64 helloworld.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World from thread 3
Hello World from thread 0
There are 4 threads
Hello World from thread 1
Hello World from thread 2
$
```

Compiler - Sun Studio 12

Download von `http://developers.sun.com/sunstudio/` (Registrierung nötig)

Verwendung des Sun CC

```
$ cd /opt
$ tar xvjf SunStudio12ml-linux-x86-200709-ii.tar.bz2
$ cd $HOME
$ /opt/sunstudio12/bin/cc -xopenmp helloworld.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World from thread 0
There are 4 threads
Hello World from thread 2
Hello World from thread 3
Hello World from thread 1
$
```

Compiler - Intel Compiler

Linux-Version für nichtkommerziellen Gebrauch ist gratis. Download von `http://www.intel.com/cd/software/products/asmo-na/eng/download/download/` (Registrierung nötig)

Verwendung des Intel CC

```
$ icc -openmp helloworld.c
helloworld.c(8): (col. 1) remark:
OpenMP DEFINED LOOP WAS PARALLELIZED.
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World from thread 0
There are 4 threads
Hello World from thread 1
Hello World from thread 3
Hello World from thread 2
$
```

Links

- OpenMP Homepage: `http://www.openmp.org`
- OpenMP auf Wikipedia: `http://en.wikipedia.org/wiki/OpenMP`
- OpenMP Spezifikationen: `http://www.compunity.org/speci/`

Fragen?

Ausblicke - was ich nicht behandelt habe...

- weitere OpenMP Direktiven und Funktionen (es gibt noch mehr, ...)[3]
- Vergleich mit anderen Modellen zur Parallelprogrammierung, z.B. MPI

Vielen Dank für Ihre Aufmerksamkeit

Wolfgang Dautermann
wolfgang.dautermann@fh-joanneum.at

[3] das war auch eine OpenMP-Einführung, kein Expertenvortrag ☺