Parallele Algorithmen

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- mwilhelm@hs-harz.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt

1. Einführung, Literatur, Motivation
2. Architektur paralleler Rechner
3. Software
4. **Open MP**
5. MIMD
6. Algorithmen
7. Computernumerik
8. PS/3

Open MP: Open Spezifikation für Multiprocessing

- OpenMP (Open Multi-Processing) ist ein Programmierinterface (API)
- Standardisiert, einfache Schnittstelle, portabel, skalierbar
- Unterstützt Multiplattform mit “shared memory”
- Programmiersprachen C/C++ und Fortran, neu C#, OpenMP für Java
- Vielfältige Architekturen (Unix, Linux und Windows)
- Es basiert auf Compiler Direktiven, verwendet werden Libraries (DLL, LIB)
- Wenig Änderung am Quellcode, mit einem Mausklick Übersetzung als Single-Thread-Programm (_OPENMP)
- Gesteuert von Environment Variablen (Anzahl Threads, Nested loop)
- Explizite Parallelität, vom Programmierer
- Grundidee sind die parallelen Schleifen, Threads und Abschnitte
- getPackages pro Thread, Option : Schedules
- Kein paralleles I/O, Sicherstellen vom Programmierer, MPI2 hat paralleles I/O
- Läuft vom PC bis zum Supercomputer.
- Für PC´s ab Visual Studio 2005

Open MP-Literatur

- **Chapman, Barbara M. (Hrsg.)**

Shared Memory Parallel Programming with Open MP
5th International Workshop on Open MP Application and Tools, WOMPAT 2004, Houston, TX, USA, May 17-18, 2004
Reihe: Lecture Notes in Computer Science, Band 3349
2005, X, 149 p. ISBN: 3-540-24560-X

- **Quinn Michael J,**

Parallel Programming in C with MPI and OpenMP McGraw-Hill Inc. 2004. ISBN 0-07-058201-7

R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald,

Parallel Programming in OpenMP. Morgan Kaufmann, 2000. ISBN 1558606718

Open MP-Literatur

- **S. Hoffmann, R. Lienhart**

OpenMP, Informatik im Fokus

Springer-Verlag, ISBN 978-3-540-73122-1

- **R. Eigenmann (Editor), M. Voss (Editor),**

OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, IN, USA, July 30-31, 2001. (Lecture Notes in Computer Science). Springer 2001. ISBN 354042346X

Links für Open MP-Literatur

Referenzseite:

http://www.openmp.org

Tutorial:

http://www.llnl.gov/computing/tutorials/openMP/

http://gcc.gnu.org/projects/gomp/

Vorlesung:

High Performance Computing mit Unterlagen, u. a. Einführung zu OpenMP:

http://www.sc.rwth-aachen.de/Teaching/Lectures/HPC05/

Eigenschaften

- OpenMP hat einen hohen Abstraktionsgrad. Die Threads werden durch den Compiler und die Laufzeitumgebeung verwaltet.
- Die Aufteilung der Felder geschieht durch die OpenMP-Bibliothek
- Die ursprüngliche sequenzielle Codestruktur bleibt erhalten. Ein nicht "OpenMP-Compiler" ignoriert die Anweisungen
- Man kann schrittweise das Programm parallelisieren (Last-Minute-Feature)
- Lokal begrenzt
- Einfach zu verstehen (für einige Studenten) !

Compiler-Direktiven

- #pragma omp <Directive> [Klausel , Klausel …]

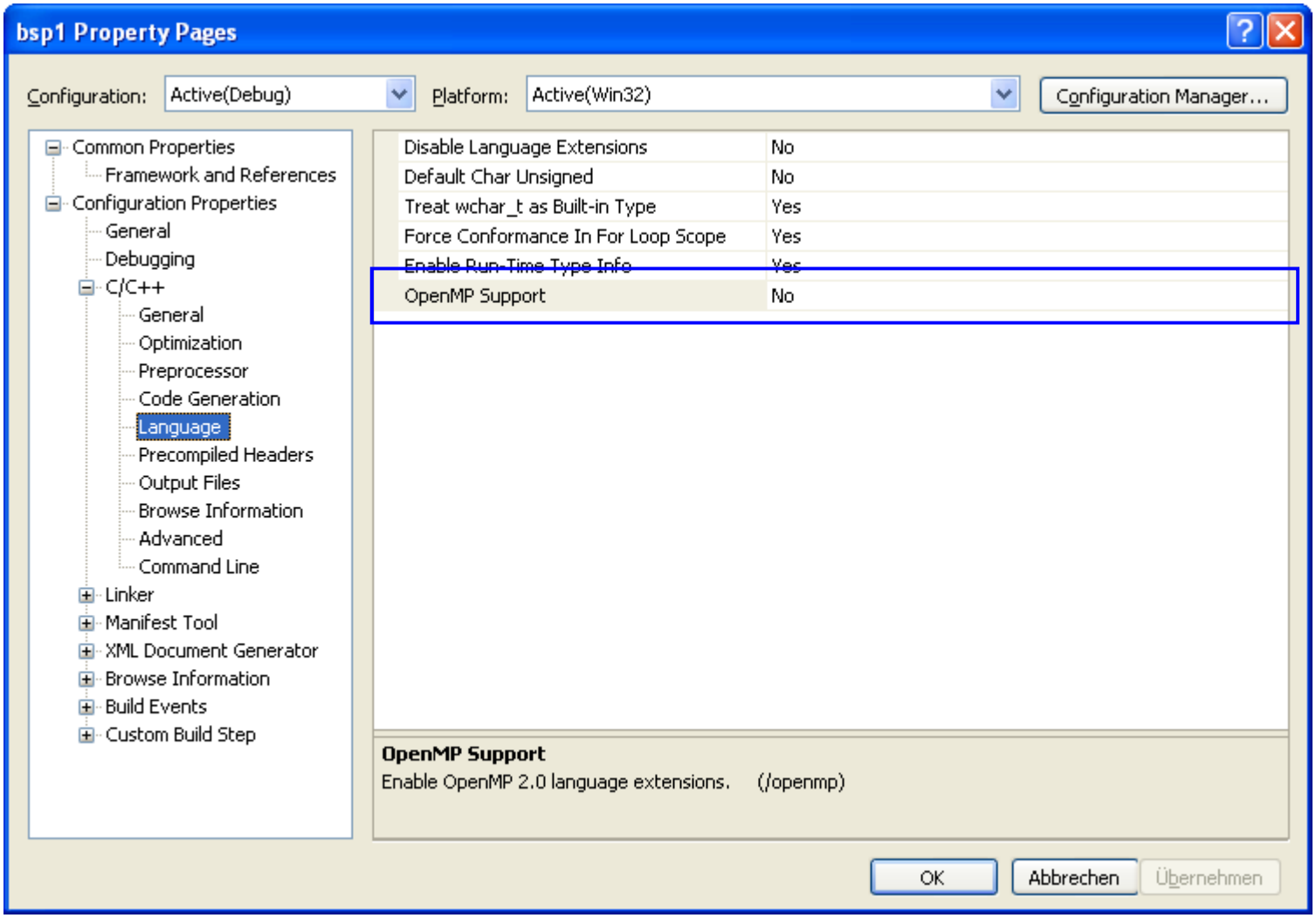
- #pragma omp parallel {
  - // Fehler, da Klammer falsch ist gesetzt wurde
- }

- #pragma omp parallel
- {
  - // Korrekt
- }

Omp-Headerdatei

- Für Zusatzfunktion notwendig:
- include <omp.h>
- bzw.
- #ifdef _OPENMP
- #include <omp.h>
- #endif

Omp-Headerdatei einstellen: /openmp



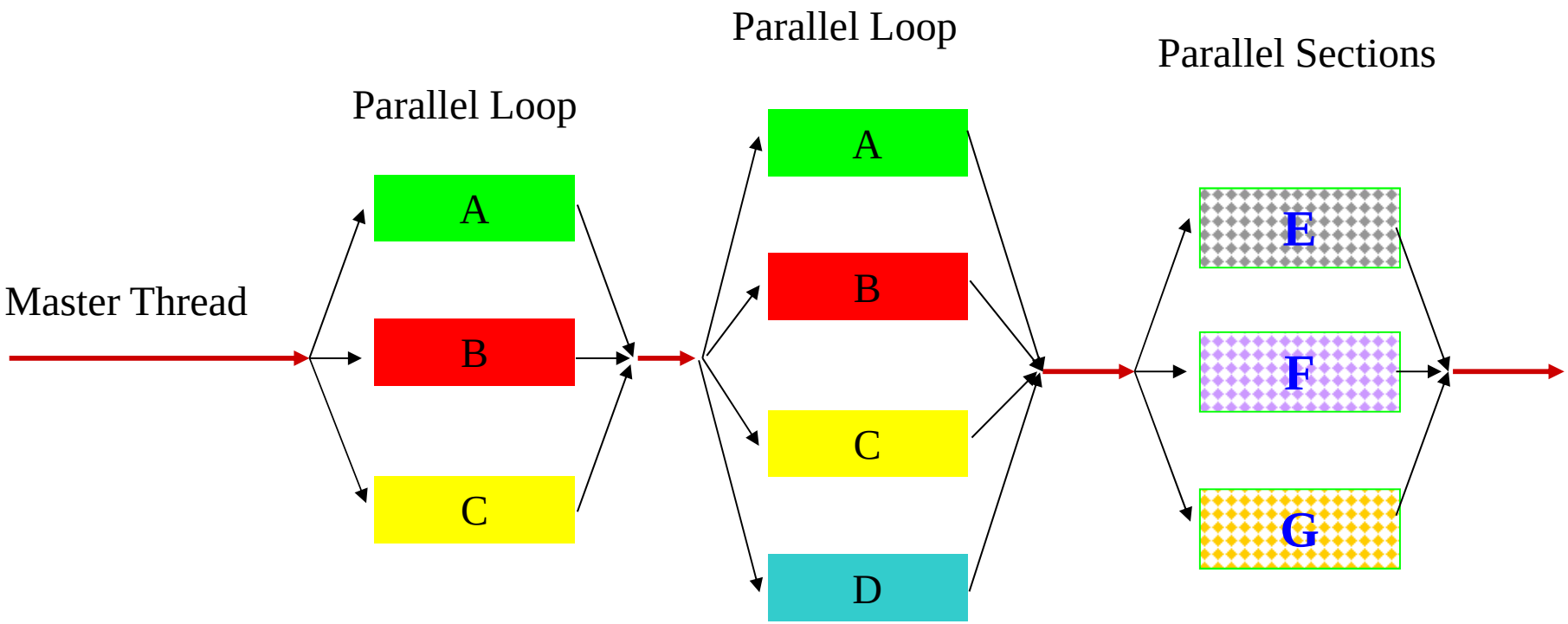
Beispielcode: Wertetabelle für eine lineare Funktion

```
const int MAX=100;

main ()  {
        int i;
        double a, b;
        double x[MAX], y[MAX];
        input(&a, &b, x, MAX);

        for (i=0; i<100; i++) {
                y[i] = a + b*x[i];
        }
}
```

Beispielcode: Wertetabelle für eine lineare Funktion

```
const int MAX=100;

main ()  {
        int i;
        double a, b;
        double x[MAX], y[MAX];
        input(&a, &b, x, MAX);  // Eingabe der Werte
    .
        #pragma omp parallel for
        for (i=0; i<100; i++) {
                y[i] = a + b*x[i];
        }
}
```

Ablauf-Prinzip: Fork vs. Join



Loop: gleicher Quellcode

Section: unterschiedlicher Quellcode

Beispielcode

```
main ()  {
        int var1, var2, var3;
        // Serieller Code
    .
        // Start des parallelen Abschnitts. Autom. create Theads
        // Definieren des Verhaltens der Variablen var1 bis var3
        #pragma omp parallel for private(var1, var2) shared(var3)
        {
                //  Paralleler Abschnitt aller Threads
                for (i=0; i<MAX; i++) {
                    …
                }
      .         // Alle Threads müssen warten
        }
         // Serieller Code       .
}
```

Allgemeine OpenMP Optionen

#pragma omp parallel [clause ...]

if (scalar_expression)	*if (bParallelLoop)*
private (list)	*untersch. Variablen pro Thread*
shared (list)	*Shared nur eine Variable*
default (shared | none)	
firstprivate (list)	*Init mit globalem Wert*
reduction (operator: list)	*(+:summe)*
copyin (list)	*Copy from Master*
num_threads (integer)	*Anzahl Threads*

structured_block *Quellcode { … }*

Wichtige OpenMP Optionen

Option	Beschreibung
parallel	Definiert einen parallelen Abschnitt. Mehrere Thread haben den gleichen Quellcode.
sections	Definiert einen parallelen Abschnitt. Mehrere Thread haben unterschiedliche Quellcodes.
single	Dieser Abschnitt in einem parallelen Abschnitt wird nur von einem Thread durchlaufen (der erste bekommt es, z. B. I/O). Alle warten !
copyprivate	Nur mit "single" gültig. Private Variable des single-Threads. NACH der Schleife erhalten alle Threads die Werte der Variablen.
atomic	Zugriff, Anweisung auf Variable ist geschützt (sum+=a). Schneller als critical. **Nur += oder -= etc.**
critical	Dieser Abschnitt wird nur von einem Thread pro Zeit durchlaufen (critical section)

Option	Beschreibung
flush	Jeder Thread kann ein "Cache" der globalen Variablen haben. Mit "flush" werden diese Variablen synchronisiert
parallel for	Angabe einer parallelen For-Schleife innerhalb eines parallelen Bereiches
ordered	Zeigt an, dass die nächsten Anweisungen "aufsteigend"´wie im sequenziellen Programm laufen sollen (z. B. Ausgabe)
if (bedingung)	Die for-Schleife wird solange ausgeführt, bis die bestimmte Bedingung zutrifft: if (n>100)
barrier	Alle Threads müssen hier warten (Join).
Master	Dieser Abschnitt wird nur vom Master durchlaufen (siehe single). Alle anderen laufen WEITER !

Option	Beschreibung
default	Angabe, ob private Variable in parallelen Breichen geteilt werden sollen
Shared	Gemeinsame Benutzung der variablen durch alle Threads. Meistens Readonly
num_threads()	Setzen der Anzahl der Threads für die Schleife

Option	Beschreibung
private	Angabe von privaten Variablen (N-Kopien) Keine Initialisierung
firstprivate	Private Variable, aber die Initialisierung erfolgt aus dem "globalen" Wert vor der Schleife.
lastprivate	Private Variable, aber der absolut letzte Wert entscheidet über den Wert nach der Schleife
copyin	Fast identisch mit firstprivate, private Variablen werden NICHT initialisiert
threadprivate	Es existiert eine globale Variable, die in der Schleife als lokale Variable deklariert wird. Der Wert wird aus der globalen Variablen genommen. Eine Zuweisung innerhalb der Schleife ändert den globalen Wert **NICHT**
copyprivate	

Wichtige OpenMP Optionen

Option	Beschreibung
nowait	Thread wartet nicht auf eine Barrriere
reduction	Gruppierung Operation auf diese Variable NACH der Schleife (Addition, Subtraktion, Multiplikation, bitweise und logische And und Oder, Xor) Keine Division Initialisierung: +: 0 -: 0 *: 1 ^: 0 &:~0 |: 0 ||:0 &&:1

Bibliotheksfunktionen (mit include <omp.h>)

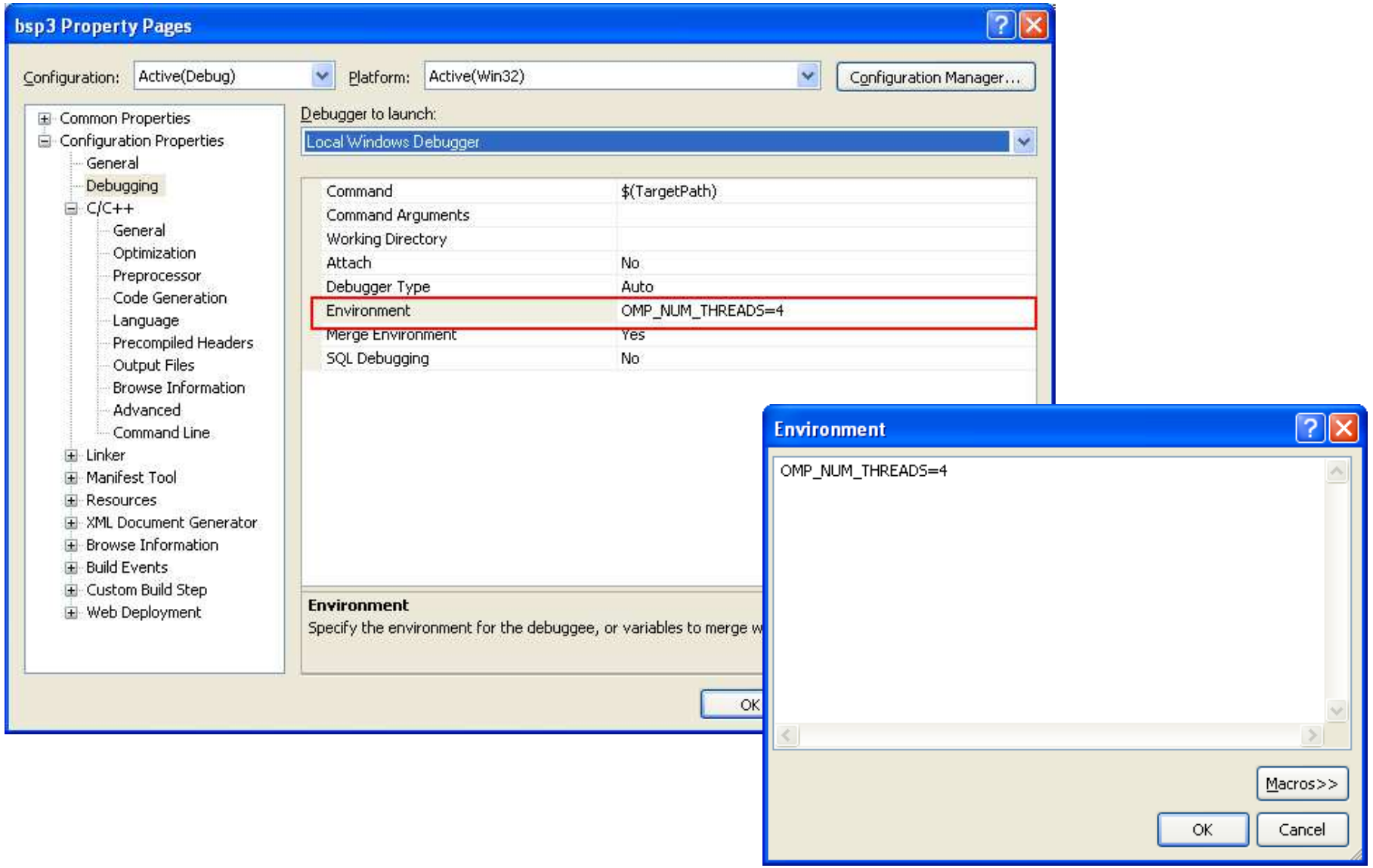
- omp_get_thread_num *Anzahl der Threads in diesem Abschnitt*
- omp_get_num_procs *Anzahl der CPU´s*
- omp_set_nested *Setzt verschachtelten Parallelismus*
- omp_get_nested
- omp_init_lock(omp_lock_t * lock)
- omp_destroy_lock
- omp_set_lock *Semaphor down*
- omp_unset_lock
- omp_test_lock
- omp_get_wtime *Zeitmessung im double-Format*
- omp_get_wtick *Zeit zwischen Prozessor ticks*

Anzahl der Threads (holen und setzen)

- Environment-Variable OMP_NUM_THREADS
- Angabe in #pragma num_threads(integer), statisch
- Aufruf der API-Funktion, dynamisch
  - omp_set_num_threads(integer)
  - omp_get_num_threads()
  - omp_get_max_threads()

- Dynamische Threads
  - bool omp_get_dynamic(); // Ausgabe enabled disabled
  - omp_set_dynamic(true/false); // setzt Status

Environment-Variable setzen: jeweils pro Modus (debug/release)



Schleifen in OpenMP (1)

- single-Loop, normale Schleife
  - for (int i=0; i<n; i++) {
  - }
- parallele Schleife
  - #pragma omp parallel for
  - for (int i=0; i<n; i++) {
  - }
  - Schleifenindex immer private, copy
- Verschachtelte parallele Schleife
- Beispiel
  - #pragma omp parallel for
  - for (int i=0; i<n; i++) {
    a[i] = b[i] + a[i-1]
  - }

Fehlerhafte Schleife

Schleifen in OpenMP (2)

- Allgemeine parallele Schleife

```
#ifdef _OPENMP
 #include <omp.h>
#endif
int _tmain(int argc, _TCHAR* argv[]) {
        int id,   numThreads;
        #pragma omp parallel private (id, numThreads)
        {
                numThreads = omp_get_num_threads();  // Anzahl Threads
                id=omp_get_thread_num();  // Threadnummer
                printf("%d  %d\n",id, numThreads);   // Ausgabe  0/2 und 1/2
        }
        return 0;
}
```

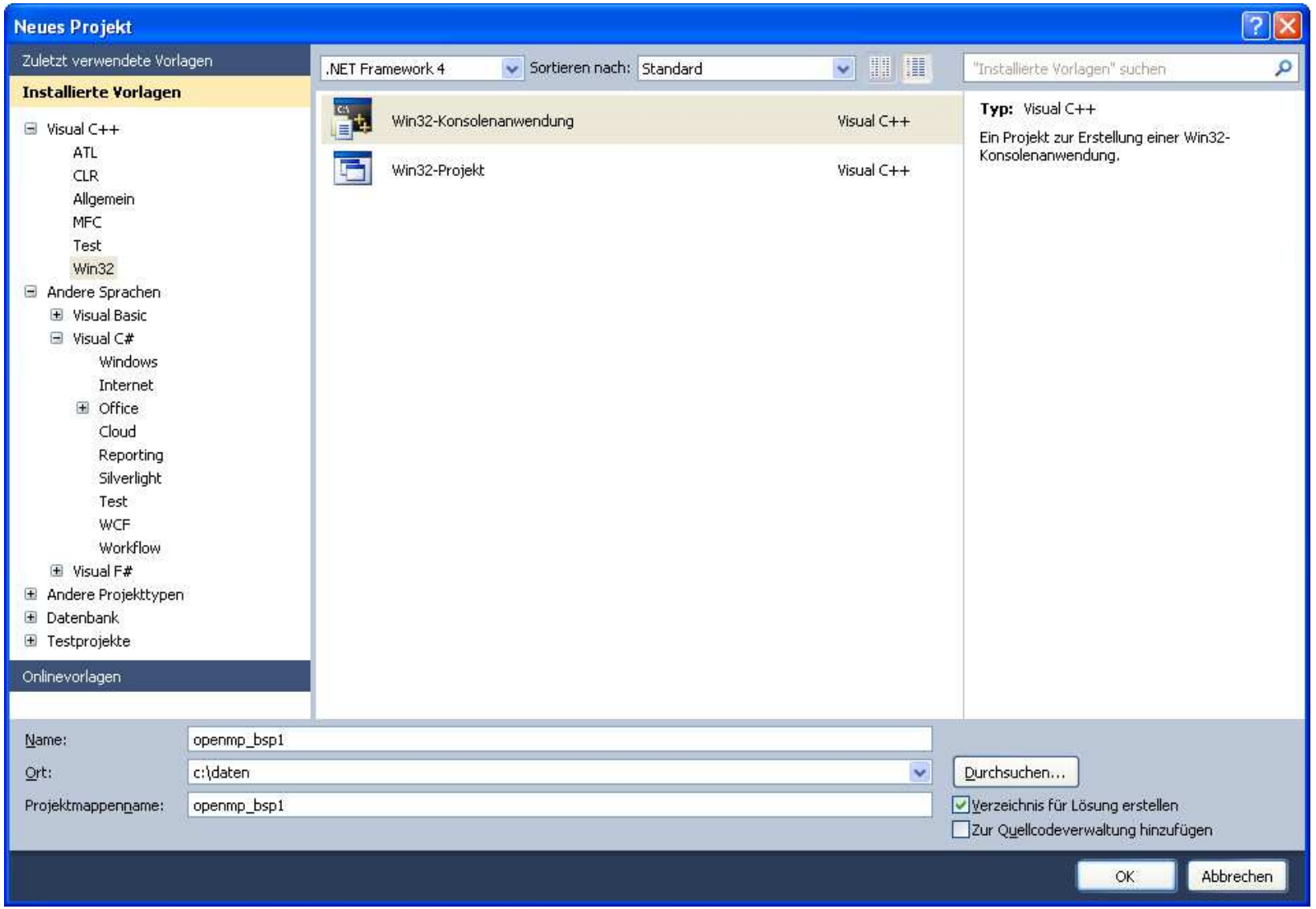
Restriktionen bei Schleifen in OpenMP

- Der Schleifenindex muss ein Integer sein
- Die Schrittweite muss vorher bekannt sein, keine While-Schleife
- Die Schleife darf nicht mit break verlassen werden
- continue ist erlaubt
- Eine Exception muss innerhalb der Schleife abgefangen werden

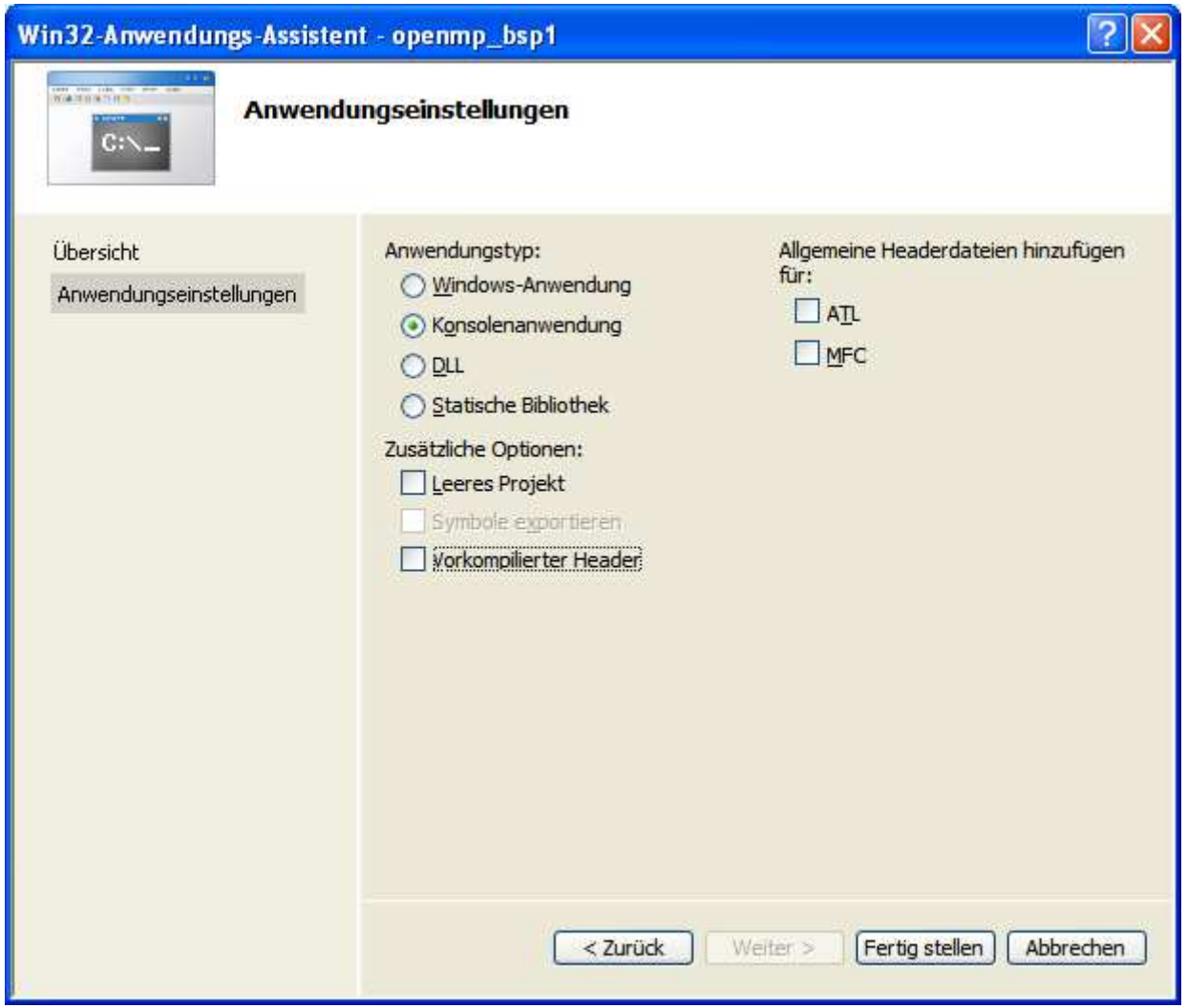
2. Beispiel in OpenMP: Initialisieren eines Feldes

- Aufruf Visual Studio 2005/2008
- Neues Projekt
  - Win32-Projekt oder CLR-Projekt
  - Console Application
  - Name: bsp2
  - Verzeichnis: c:\daten
- pragma-Anweisung
- Debug-Option setzen, Alt+F7, C/C++, Language
- Definieren der Felder
- Init-Schleife über die alle Elemente

Neues Project



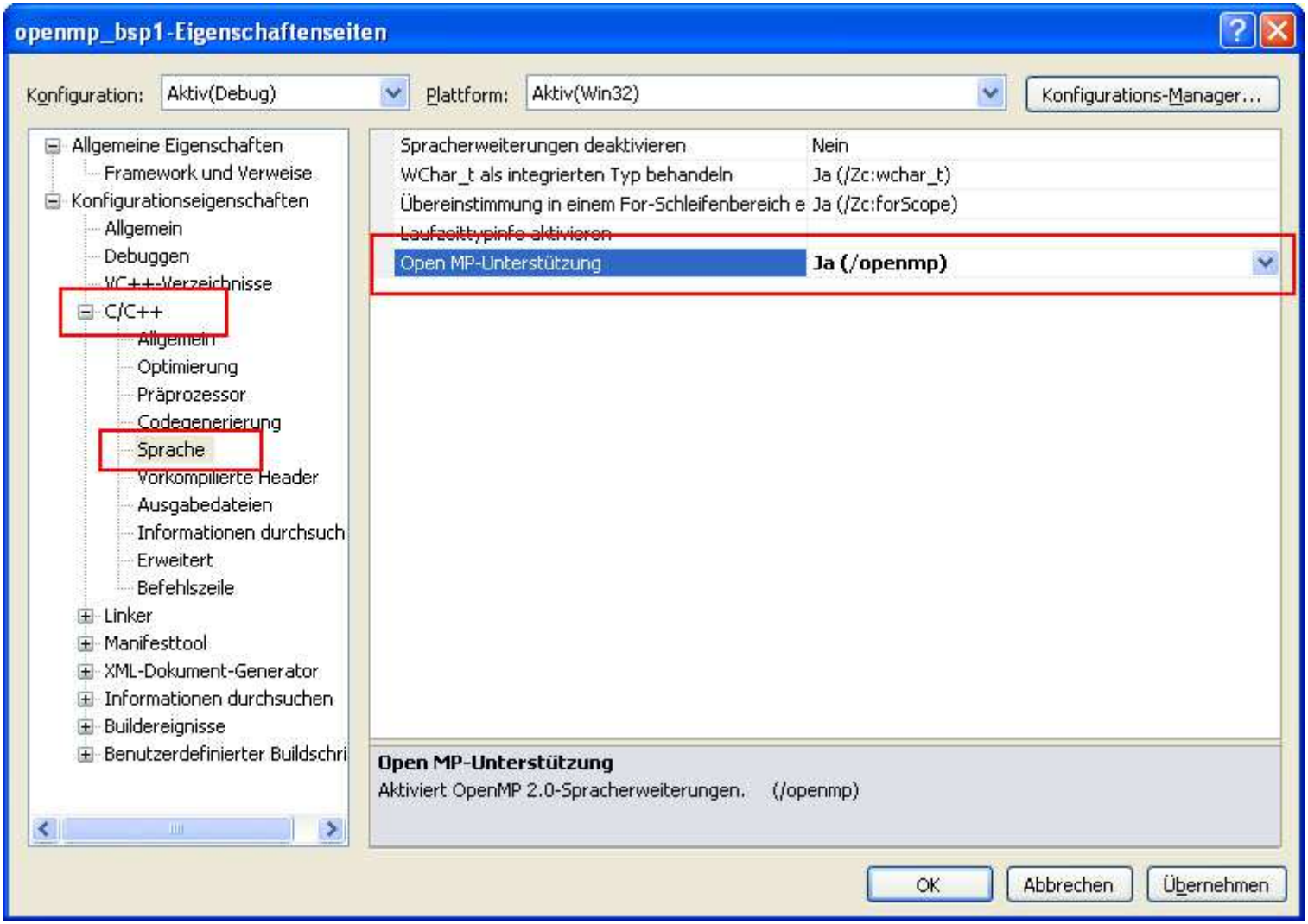
Neues Project



Beispielquellcode

```
#include "stdafx.h"

#ifdef _OPENMP
  #include <omp.h>
#endif

int _tmain(int argc, _TCHAR* argv[])
{
        puts("Hello World");
        return 0;
}
```

OpenMP anschalten: Projekt Eigenschaften jeweils pro Modus



Beispielquellcode: Win32-Projekt

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    const int MAX = 1000;
    int a[MAX];
    #pragma omp parallel for
        for (int i = 0; i < MAX; ++i)  {
            a[i]=i+1;            // wird aufgeteilt auf die Threads
        }

    for (int i = 0; i < MAX; ++i) {
        printf("A[ %d ] = %d\n",      i, a[i]    );
    }
    return 0;
}
```

3. Beispiel in OpenMP:

- Erweiterung / Änderung des Projekts bsp2
  - Angabe der Anzahl der Threads
  - Ausgabe der Thread-ID

```
Max Threads: 2  Anz CPU 2
Max Threads: 4  Anz CPU 2
Thread 3  lokales i: 8
Thread 3  lokales i: 9
Thread 2  lokales i: 6
Thread 2  lokales i: 7
Thread 1  lokales i: 3
Thread 1  lokales i: 4
Thread 1  lokales i: 5
Thread 0  lokales i: 0
Thread 0  lokales i: 1
Thread 0  lokales i: 2
```

```
#include <omp.h>  // Vorraussetzung

printf("Max Threads: %d  Anz CPU %d",
                    omp_get_max_threads(),    omp_get_num_procs() );
omp_set_num_threads(4);

printf("Max Threads: %d  Anz CPU %d",
          omp_get_max_threads(), omp_get_num_procs() );

#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
          int id = omp_get_thread_num();
          printf("Thread %d  lokales i: %d\n", id, i );
}
```

4. Beispiel in OpenMP:

- Erweiterung / Änderung des Projekt bsp3a
  - Bilden der Summe

$$s = \sum_{i=1}^{100} i = \frac{100 \cdot 101}{2} = 5050$$

```
// bsp4.cpp : Summe von 1 bis 100
#include "stdafx.h"
#include <omp.h>

int main(array<int argc, _TCHAR* argv[])  {
    omp_set_num_threads(4);
    int sum=0;
    int j;
    #pragma omp parallel for private(j) shared(sum)
        for (int i = 0; i < 100; ++i) {
            j=sum;
            sum=j  +  i+1;
        }
    printf("Summe: %d  ", sum );
    return 0;
}
```

Ergebnis ist nicht immer korrekt

```
// bsp5.cpp : Aufteilen in 4 Zwischensumme
#include "stdafx.h"
#include <omp.h>

int main(array<System::String ^> ^args)  {
        int j;
        int summe, sum[4];
        for (int i=0; i<4; ++i) sum[i]=0;
         omp_set_num_threads(4);
        #pragma omp parallel for private(j) shared(sum)
                for (int i = 0; i < 100; ++i) {
                        int id = omp_get_thread_num();  // Aufwand, interner Thread
                        j=sum[id];
                        sum[id]=j + i+1;
                }
        summe=0;
        for (int i=0; i<4; ++i) summe+=sum[i];
        printf("Summe: %d\n  ", summe );
        return 0;
}
```

bsp5b

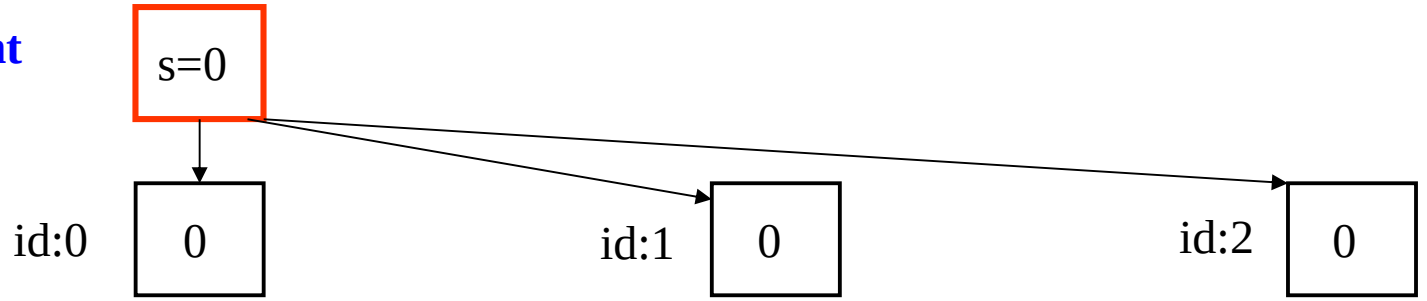
Reduktion bei Schleifen in OpenMP

- Variable einmal definiert
- Automatische Aufteilung pro Thread
- Automatische Reduzierung, je nach Operation
- **Beispiel:**

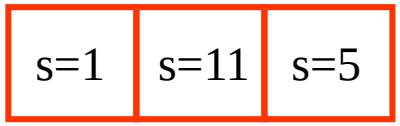
```
double test1() {
    double sum=0.0;
    int i;
    double a[100];
    a[i]=??;
    #pragma omp parallel for private(i)  reduction(+:sum)
        for (i=0; i<100; i++) {
            sum += a[i];
        }
    return sum;
}
```

1) Initialisierung s=0

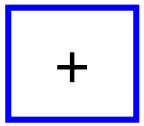
2) Reduce-Sent

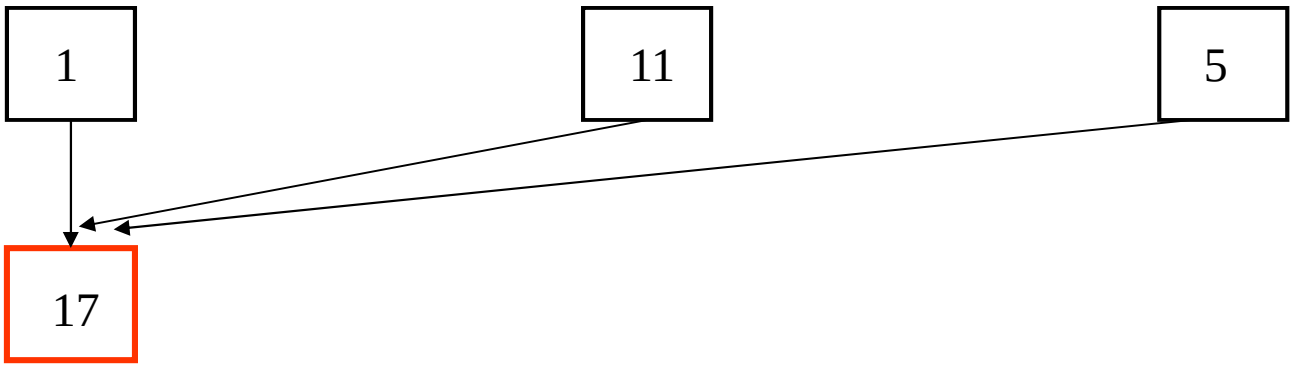


3) Berechnung



4) Reduce-Op





Reduktion bei Schleifen in OpenMP

- Erlaubte Operatoren für die Reduktion

Operation	Zeichen	Initialisierung
plus	**+**	**0**
Subtraktion	**-**	**0**
Multiplikation	*****	**1**
Bitweises UND	**&**	**~0 !0**
Bitweises ODER	**|**	**0**
Bitweises XOR	**^**	**0**
Logisches UND	**&&**	**1**
Logisches ODER	**||**	**0**

```
// bsp5.cpp : main project file
#include "stdafx.h"
#include <omp.h>

int main(array<System::String ^> ^args)  {
        int j;
        omp_set_num_threads(4);
        int summe=0;       // Diesen Wert erhält jeder Thread, à la copyprivate
        #pragma omp parallel for private(j) reduction(+:summe)
                for (int i = 0; i < 100; ++i) {
                        j=summe;
                        summe=j + i+1;
                }
        printf("Summe: %d\n  ", summe );
        return 0;
}
```

6. Beispiel in OpenMP:

- Einfache „Schleife“ mit n-Threads
- Ohne For-Schleife, bsp6
  - Angabe der Anzahl der Threads
  - Ausgabe der Thread-ID
- Ziel: getPackageProgramming à SETI

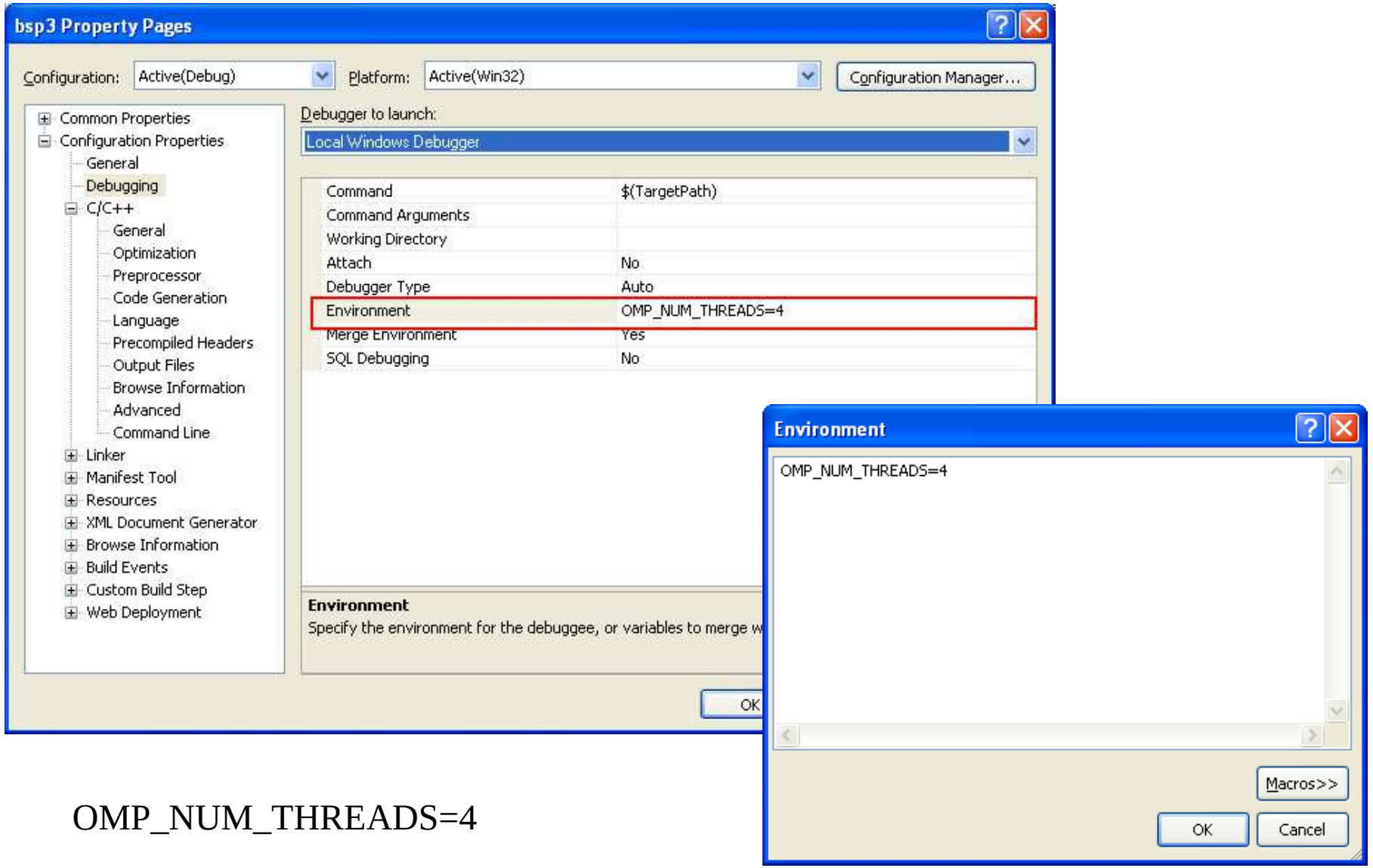
- Ausgabe: vier Threads

```
Thread id:0  x: 31125
Thread id:1  x: 93625
Thread id:3  x: 218625
Thread id:2  x: 156125
```

```
// bsp6.cpp : Beispiel für allgemeine Schleifen
#include "stdafx.h"
#include <omp.h>

int main(array<System::String ^> ^args)  {
    int id, numThreads;
    // hier werden je nach Definition der Env-Var die Threads erzeugt und gestartet
    #pragma omp parallel private (id, numThreads)
   {
        numThreads = omp_get_num_threads();
        id=omp_get_thread_num();
        printf("Thread id: %d\n", id);
    }
    return 0;
}
```

Enviromnent: gesetzt : OMP_NUM_THREADS=4

Environment-Variable setzen: jeweils pro Modus (debug/release)



OMP_NUM_THREADS=4

```
// bsp6.cpp : manuelle Aufteilung mit Threads, feste Angabe
#include "stdafx.h"
#include <omp.h>

int main(array<System::String ^> ^args)  {
    const int MAX=1000;
    int id, numThreads, start, points;
    double x;
    #pragma omp parallel private (id, numThreads)  num_threads(4)
     {
        numThreads = omp_get_num_threads();
        id=omp_get_thread_num();
        points = MAX / numThreads;
        start = id * points;
        x = calc(id, start, points);
        printf("Thread id: %d    x: %d\n", id,x);
    }
    return 0;
}
```

Ausgabe von bsp6a

```
double calc(int id, int start, int points)
{
        return start*points;
}
```

```
Thread id:2  numThreads: 4    start: 500,    points: 250,    x: 125000
Thread id:3  numThreads: 4    start: 750,    points: 250,    x: 187500
Thread id:1  numThreads: 4    start: 250,    points: 250,    x: 62500
Thread id:0  numThreads: 4    start: 0,      points: 250,    x: 0

Ende
```

7. Beispiel in OpenMP:

Dynamische Aufteilung der Pakete

- Die Schleifenbereiche werden erst in der Schleife aufgeteilt. Wenn die Schleifenarbeit fertig ist, wird ein neues i zugeteilt. Möglich sind auf mehrere Pakete (default chunk size = 1).
- Ziel: getPackageMuster, SETI
- Verteilung abhängig von den echten CPU`s
- Threadregistersatz, Pro Thread ein Satz Register

N=1000, AnzThread: 10

ID: 0 Summe: 308
ID: 1 Summe: 292
ID: 2 Summe: 213
ID: 3 Summe: 183
ID: 4 Summe: 0
ID: 5 Summe: 0
ID: 6 Summe: 0
ID: 7 Summe: 0
ID: 8 Summe: 0
ID: 9 Summe: 0

Option SCHEDULE

Beschreibt, wie die Thread in einer Schleife aufgeteilt werden:

STATIC

Die Schleifenbereiche werden vor der Schleife aufgeteilt. Wenn die Grenzen nicht bekannt sind, werden die Indizes kontinuierlich aufgeteilt.

DYNAMIC

Die Schleifenbereiche werden erst in der Schleife aufgeteilt. Wenn die Schleifenarbeit fertig ist, wird ein neues i zugeteilt. Möglich sind auf mehrere Pakete (default chunk size = 1).

GUIDED

Die Schleifenbereiche werden vor der Schleife aufgeteilt. Wenn die Grenzen nicht bekannt sind, werden die i´s kontinuierlich aufgeteilt. Hier besteht die Möglichkeit, dass man mehrere “Pakete” von Indizes vergibt.

RUNTIME

Mit schedule(runtime) kann man die Aufteilung der Indizes durch die Environment Variable OMP_SCHEDULE definieren.

```
// bsp7.cpp : SETI-Projekt, Teil1
const int MAX=10;
int sum[MAX];
int i, tid, N;
 omp_set_num_threads(MAX);
 for (int i=0; i<MAX; i++) sum[i]=0;
 N = 1000;
  #pragma omp parallel for schedule(dynamic,1) private(i)
     for (i = 0; i < N; ++i)    {
            tid = omp_get_thread_num();
            int n = (int) (rand()/100000);
            for (int j=0; j<(n); j++) double d = sin(33.44);
            sum[tid]++;
     }
 for (int i=0; i<MAX; i++)
      printf("ID: %d   Summe: %d\n", i, sum[i]);
 }
```

```
// bsp7.cpp : Seti-Projekt, Teil2: Zwei Parallele Schleife
const int MAX=10;
int sum[MAX];
int i, tid, N;
omp_set_num_threads(MAX);
for (int i=0; i<MAX; i++) sum[i]=0;
N = 1000;
  #pragma omp parallel private(i)
     #pragma omp for schedule(dynamic,1) nowait       // chunk=1
          for (i = 0; i < N; ++i)  {
            tid = omp_get_thread_num();
            sum[tid]++;
          }
for (int i=0; i<MAX; i++)
      printf("ID: %d   Summe: %d\n", i, sum[i]);
```

8. Beispiel in OpenMP:

Sectionsaufteilung

- Definieren mehrerer Abschnitte / Sections
- Unterschiedlicher Quellcode pro Thread

Ergebnis:

Section 1 id: 0 AnzThread: 4
Section 5 id: 0 AnzThread: 4
Section 3 id: 3 AnzThread: 4
Section 2 id: 2 AnzThread: 4
Section 4 id: 1 AnzThread: 4

```
// bsp8.cpp : Sectionsaufteilung jede Section wird einmal gestartet
omp_set_num_threads(MAX);
 #pragma omp parallel sections private(tid)
 {
    #pragma omp section
    {
            tid = omp_get_thread_num();
            Console::WriteLine("Section 1  id: {0}", tid);
    }
    #pragma omp section
    {
            tid = omp_get_thread_num();
            printf("Section 2  id: %d\n", tid);
    }

 }
```

Alternative Lösung mit drei Schleifen

```
// numThreads ist zwei
chunkSize = middleLoopCount / numThreads;
for(int j = 0; j < outerLoopCount; j++) {
        #pragma omp parallel sections
        {
                #pragma omp section
                  for (int i = 0; i < chunkSize; i++) {
                        Work(innerLoopCount);
                  }
                #pragma omp section
                  for (int i = chunkSize; i < middleLoopCount; i++) {
                        Work(innerLoopCount);
                  }
        }
}
```

Nested Loop in OpenMP

- Mehrere Schleifen können unterschiedlich parallelisiert werden
  - Nur die äußeren Schleifen
  - Nur die inneren Schleifen
  - Zwei oder alle Schleifen
- Das Ergebnis ist immer gleich
- Die Performance nicht!, innere Schleifen manchmal mit einem Thread
- Man sollte sich an den vorhandenen Prozessoren halten
- Verschachtelte Schleifen erlauben mit
  - omp_get_nested()
  - omp_set_nested()
  - OMP_NESTED Environment Variable wird TRUE gesetzt

9. Beispiel in OpenMP:

Threads in verschachtelten Schleifen

- Außen-Parallelisieren
- InnenParallelisieren
- Innen und Außen-Parallelisieren
- Genaue Zeitmessung (Release-Modus, CLR vs. Win32)
  - System::Diagnostics
  - Stopwatch^ sw = gcnew Stopwatch
  - Start, Stop, Reset, Elapsed

Quellcode der beiden verschachtelten Schleifen:

```
#pragma omp parallel for
    for (int i1=0; i1<N; i1++) {
        a[0][i1] = 0.0;
        #pragma omp parallel for
        for (int i2=0; i2<N; i2++) {
            a[0][i2] += a[i2][i1];
        }
    }
```

1. Variante:
Äußere Schleife parallesiert

```
// bsp9.cpp : Zwei Schleifen verschachtelt
#include "stdafx.h"
#include <omp.h>
int main(array<System::String ^> ^args) {
    const int N = 200;
    double a[N][N];
    LARGE_INTEGER t1,t2;
    long z;
    t1 = GetTickCount2();
    #pragma omp parallel for                    // zwei Thread erzeugt
        for (int i1=0; i1<N; i1++) {
            a[0][i1] = 0.0;
            for (int i2=0; i2<N; i2++) {
                a[0][i2] += a[i2][i1];
            }
        }
    t2 = GetTickCount2();
    z=  (t2.HighPart-t1.HighPart)  +  (t2.LowPart-t1.LowPart);
    printf("Außen parallelisiert  time: %u\n ", z );
```

2. Variante: Innen Schleife

```
t1 = GetTickCount2();
for (int i1=0; i1<N; i1++) {
    a[0][i1] = 0.0;
        // Hier werden jeweils 2 Threads erzeugt, insgesamt N*2
    #pragma omp parallel for
        for (int i2=0; i2<N; i2++) {
            a[0][i2] += a[i2][i1];
        }
}
t2 = GetTickCount2();
z=  (t2.HighPart-t1.HighPart)  +  (t2.LowPart-t1.LowPart);
printf("Innen parallelisiert  time: %u\n ", z );
```

3. Variante: Innen und Außen

```
    t1 = GetTickCount2();
    #pragma omp parallel for
    for (int i1=0; i1<N; i1++) {
        a[0][i1] = 0.0;
        #pragma omp parallel for
            for (int i2=0; i2<N; i2++) {
                a[0][i2] += a[i2][i1];
            }
    }
   t2 = GetTickCount2();
    z=  (t2.HighPart-t1.HighPart)  +  (t2.LowPart-t1.LowPart);
    printf("Innen und Außen parallelisiert  time: %u\n ", z );
    return 0;
}
```

Ergebnisse:

C#:, N=200	time	
Außen parallelisiert:	0.0474915	2 Threads
Innen parallelisiert:	0.0063868	2*n Threads
Innen und Außen parallelisiert:	0.0029696	2*n+2 Threads

Win32, N=200	Ticks	
Ein Thread	2780	0,000777 s
Außen parallelisiert	4570	0,001277 s
Innen parallelisiert	28380	0,079292 s
Innen und Außen parallelisiert	3383	0,000945 s

Frequenz: 3579545

Weitere OpenMP-Direktive zur Synchronisation von Threads:

#pragma omp barrier

Barrieren-Synchronisation (nur in einem Anweisungsblock):

- **#pragma omp critical**

Kritischer Bereich, z.B. zur Steuerung des schreibenden Zugriffs mehrerer Threads auf eine Datenstruktur; nur ein Thread darf sich zu einem Zeitpunkt in einem kritischen Bereich befinden:

#pragma omp master

Ausführung von Anweisungen nur durch den Master-Thread:

#pragma omp single

Ausführung von Anweisungen nur durch einen Thread ausgeführt

- **#pragma omp flush(var1, var2)**

Erzwingen einer konsistenten Darstellung der Inhalte von Variablen in allen Threads (Vorsicht bei Zeigern !); wird keine Variable angegeben, so erhalten alle globalen Variablen einen konsistenten Wert. Bemerkung: erfolgt implizit bei barrier, critical sowie beim Verlassen einer for-, section- und single-Direktive (falls kein nowait)

10. Beispiel in OpenMP: Win32

Threads mit critical Section

- Berechnung greift auf ein oder mehrere Variablen zu
- Synchronisation mittels
  - pragma omp critical
  - pragma atomic
  - Semaphor

// bsp10.cpp : Threads ohne critical Section

```
int _tmain(int argc, _TCHAR* argv[]) {
    const int MAX = 100;
    double M[MAX][MAX], x[MAX], y[MAX], summe, total;
    int i, j, tid;
    // Init vom M und x
    total = 0;                    y[i]=0;
    #pragma omp parallel for shared(M,x,y,total) private(tid,i,j)  num_threads(10)
        for (i=0; i < MAX; i++) {
            summe=total;
            for (j=0; j < MAX; j++)
                y[i] += (M[i][j] * x[i]);
            total = total + y[i];
        }
    printf("Gesamtsumme y = %d\n",total);
    return 0;
}
```

```
// bsp10.cpp : Teil2, mit critical Section
int main(array<System::String ^> ^args){
    const int MAX = 100;
    double M[MAX][MAX], x[MAX], y[MAX], summe, total;
    int i, j, tid;
    // Init vom M und x
    total = 0;              y[i]=0;
    #pragma omp parallel for shared(M,x,y,total) private(tid,i,j)  num_threads(10)
        for (i=0; i < MAX; i++) {
            summe=total;
            for (j=0; j < MAX; j++)
                y[i] += (M[i][j] * x[i]);
            #pragma omp critical
                total = total + y[i];
        }
    printf("Gesamtsumme y = %d\n",total);
    return 0;
}
```

```
// bsp10.cpp : Teil2, mit atomic
int main(array<System::String ^> ^args){
    const int MAX = 100;
    double M[MAX][MAX], x[MAX], y[MAX], summe, total;
    int i, j, tid;
    // Init vom M und x
    total = 0;                                  y[i]=0;
    #pragma omp parallel for shared(M,x,y,total) private(tid,i,j)  num_threads(10)
        for (i=0; i < MAX; i++) {
            summe=total;
            for (j=0; j < MAX; j++)
                y[i] += (M[i][j] * x[i]);
            #pragma omp atomic
                total += y[i];
        }
    printf("Gesamtsumme y = %d\n",total);
    return 0;
}
```

Ergebnisse bsp10

Frequency: 0 3579545

Verfahren	Summe	Ticks	Zeit
Ohne Sync	408958.3	19931	
Mit critical Section	792690	2095	
Mit atomic	792690	2235	
Mit atomic, 1 Loop	792690	1387	

Beispiel „private und firstprivate“

```
i=1;
j=2;
k=3;
// private:       Wert in der Schleife unbestimmt, null gesetzt
// firstprivate: Wert in der Schleife aus der globalen Variablen
#pragma omp parallel private(i)  firstprivate(j)
{
     int id = omp_get_thread_num();
     printf("Thread %d  lokales i: %d  lokales j:  %d\n", id, i, j );
     i+=id;
     j+=id;
}
printf("Nach Loop i: %d  j: %d\n ", i, j);     //Nach Loop i: 1  j: 2
```

Beispiel „lastprivate“

Der letzte Wert in der Schleife wird der globalen Variablen zugewiesen

```
i=-1;
j=-2;
int n=10;
#pragma omp parallel for shared(j) lastprivate(i)
    for (i=0; i<n-1; i++) {
        j=i;
    }
printf("Nach Loop i: %d  j: %d\n ", i, j);   // Nach Loop i: 9  j: 8
printf("Nur shared(j) shared(j)  i: %d  j: %d\n ", i, j);   // Nach Loop i: -1  j: 6
```

copyprivate: der zugewiesene Wert wird allen zugestellt BCast

```
float a, b, x, y;
a=-1;        b=-2;    x=-3;                          y=-4;
#pragma omp parallel private(a,b,x,y)
{
    #pragma omp single copyprivate(a,b)
    {
        // lesen aus Datei etc.
        a=2;
        b=3;
        int id = omp_get_thread_num();
        printf("Thread-Init %d\n", id);
    }
    int id = omp_get_thread_num();
    x=(float)id;
    y=a+ b*x;
    printf("Thread {0}  a: %d  b: %d x: %d  y: %d\n", id, a, b , x, y  );
}
```

Ergebnis „copyprivate“

Init durch Master-Thread: 0

Thread 2	a: 2	b: 3	x: 2	y: 8
Thread 1	a: 2	b: 3	x: 1	y: 5
Thread 0	a: 2	b: 3	x: 0	y: 2
Thread 3	a: 2	b: 3	x: 3	y: 11

Performance-Untersuchung

- Verwendete Techniken
  - Single-Loop
  - Loop mit Section (Aufteilung ev. problematisch, 3 auf 2 Sections)
  - Parallel For (statisch, dynamische Zuweisung)
  - Win32 Threads
- Testquellcode:

```
for (int i=0; i < C_OUTERLOOP; ++i) {
    for (int j=0; j < C_MIDDLELOOP; ++j) {
        work(C_INNERLOOP);
    }
}
```

```
int work (int loopCount) {
    unsigned long result=0;
    for (int i=0; i<loopCount; ++i)
        result+=i;
    return result;
}
```

Projekt: testthread

```
void testSectionCode(int chunksize) {
    omp_set_num_threads(2);
    for (int i=0; i < C_OUTERLOOP; ++i) {
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                for (int j=0; j < chunksize; ++j) {
                    work(C_INNERLOOP);
                }
            }
            #pragma omp section
            {
                for (int j=chunksize; j < C_MIDDLELOOP; ++j) {
                    work(C_INNERLOOP);
                }
            }
        }
    }
}
```

Parallel-Code

```
void testParallelForCode(int numberThreads){
    omp_set_num_threads(numberThreads);
    for (int i=0; i < C_OUTERLOOP; ++i) {
        #pragma omp parallel for
            for (int j=0; j < C_MIDDLELOOP; ++j) {
                work(C_INNERLOOP);
            }
    }
}
```

Konstante vs. Parameter

Win32-Code

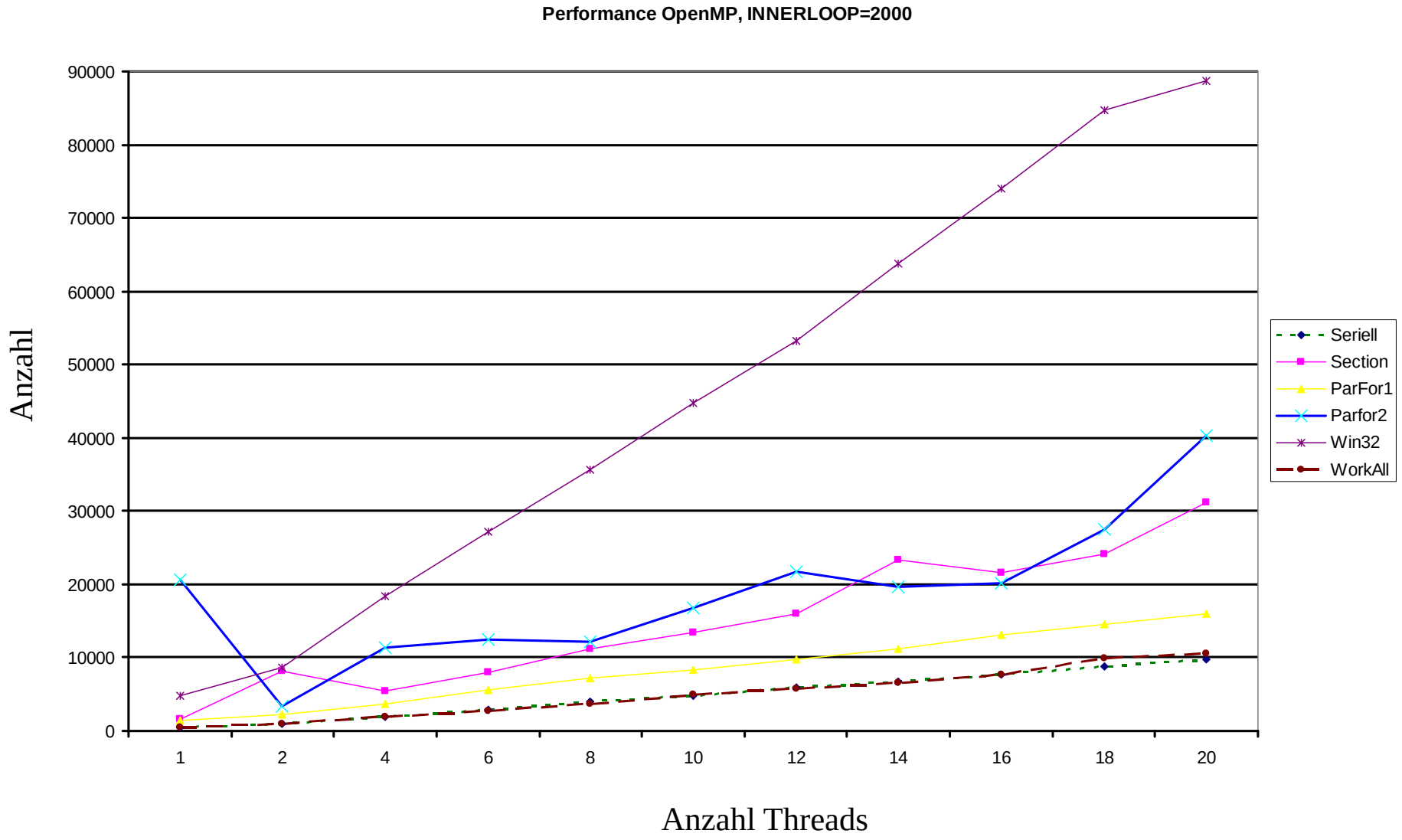
```
struct arguments {
    int i1, i2;
    int innerLoopCount;
    int middleLoopCount;
};

DWORD WINAPI LoopThreadFunction(LPVOID arguments) {
    struct arguments *args;
    args = (struct arguments *) arguments;
    int innerLoopCount=args–>innerLoopCount;
    int middleLoopCount=args–>middleLoopCount;
    for (int i=args–>i1; i < args–>i2; ++i) {
            work(C_INNERLOOP);
    }
    return 0;    // Summe
}
```

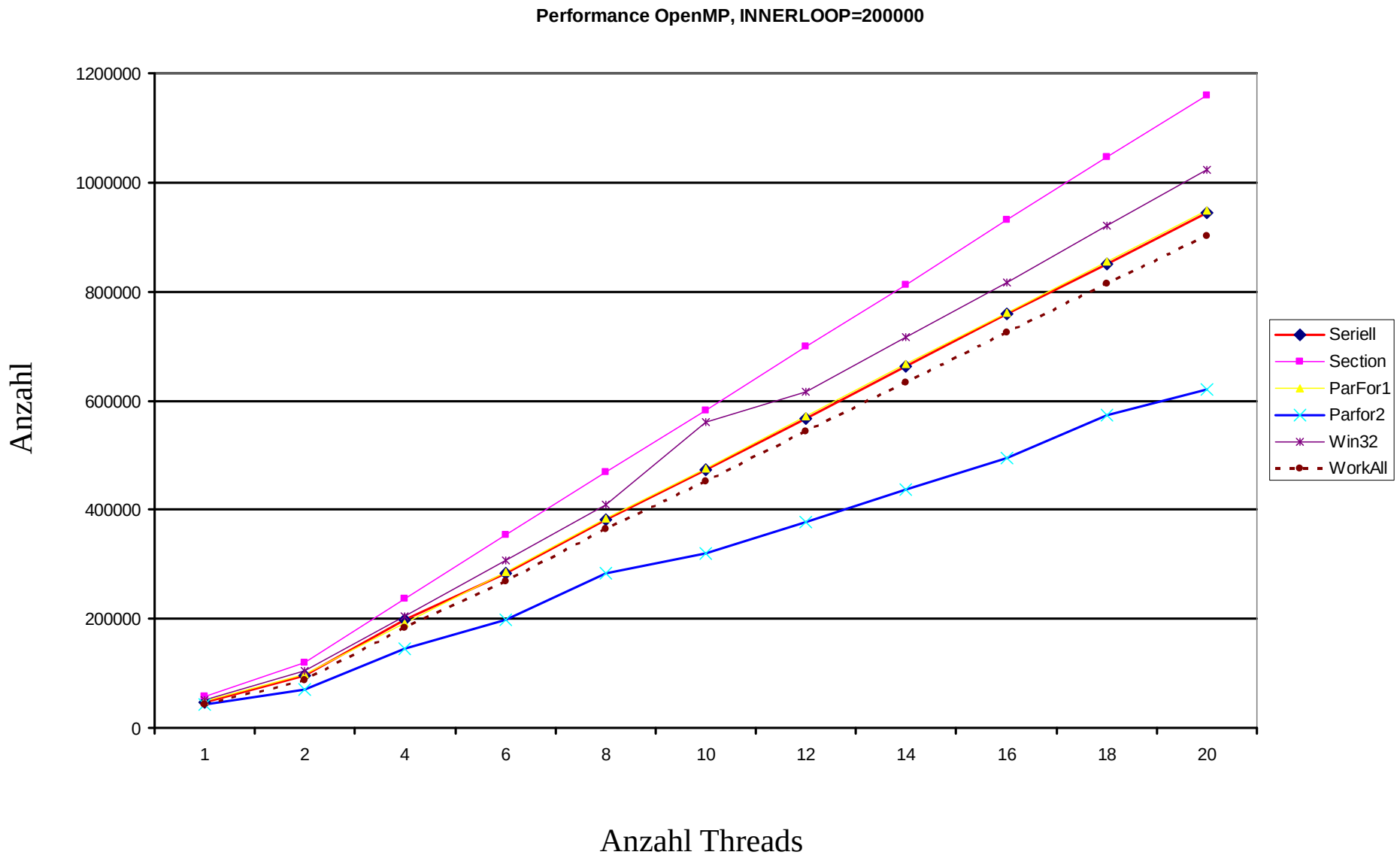
Testdaten mit GetTickCount2, Projekttyp Win32

#define C_OUTERLOOP 5
#define C_MIDDLELOOP 20
#define C_INNERLOOP 2000
Frequency: 3579545

Typ	**ticks**
Schleife Seriell	2332
Schleife 2+Sectionen	4381
Schleife Parallel-For1:	3134
Schleife Parallel-For2:	3168
Schleife Parallel-For:1 time:	2838
Schleife Parallel-For:2 time:	3235
Schleife Win32Threads time:	18465

Testergebnisse ohne Optimierung


Performance OpenMP, INNERLOOP=2000
Anzahl
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
1
2
4
6
8
10
12
14
16
18
20
Anzahl Threads
Seriell
Section
ParFor1
Parfor2
Win32
WorkAll


Performance OpenMP, INNERLOOP=200000
Anzahl
1200000
1000000
800000
600000
400000
200000
0
1
2
4
6
8
10
12
14
16
18
20
Anzahl Threads
Seriell
Section
ParFor1
Parfor2
Win32
WorkAll

If there are multiple independent loops within a parallel region, you can use the **nowait** clause (Section 2.4.1 on page 11) to avoid the implied barrier at the end of the **for** directive, as follows:

```
#pragma omp parallel
{
#pragma omp for nowait
for (i=1; i<n; i++)
b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
for (i=0; i<m; i++)
y[i] = sqrt(z[i]);
```

Pragma Critical

```
#pragma omp parallel
{
    unsigned short* buffer;
    int y;
    int summe=0;
    int nid = omp_get_num_threads();
    buffer = (unsigned short*)::AlignedAlloc(bufSize, 32);
    #pragma omp for schedule(dynamic, 4)   reduction(+: summe)
        for (y = 0; y < ymax; y++){
            summe += ... // hier die Berechnung
             #pragma omp critical
                m_dwDone++; // oder InterlockedIncrement benutzen
        } // for
    ::AlignedFree(buf);
} // omp parallel
```

Möglichkeiten der Arbeitsverteilung auf Threads:

- Master/Slave: ein Master-Thread steuert Programm, erzeugt mehrere
- Slave-Threads und verteilt Arbeit gleichmäßig auf Threads
- Pipelining: Thread i produziert Daten für Thread i +1
- Pool: mehrere Threads holen sich nach Abarbeitung einer Aufgabe eine neue Aufgabe aus einem Aufgaben-Pool
- Divide-and-Conquer: jeder Thread erzeugt rekursiv einen weiteren Thread, bis Aufgabe ausreichend fein zerlegt ist
- Wettbewerb: jeder Thread führt eine andere Strategie aus

Zusammenfassung: Vorteile

- Paralleler Quellcode ist einfach aufzubauen
- Paralleler Quellcode ist gut lesbar
  - #pragma Anweisung ist direkt davor
- Overhead ist geringer als mit normalen Thread-Klassen
- Mathematische Algorithmen lassen sich gut parallelisieren
- Ausschalten der Parallelität möglich
- Läuft auch auf Single-CPU ohne Änderung der Exe-Datei
- Nur für Multiprocessing mit gemeinsamen Speicher geeignet
- Portierbar
- OpenMP ist nicht geeignet für GUI-Programmierung
- Serieller Code ist immer der Flaschenhals (Amdahl-Gesetz)

Zusammenfassung: Nachteile

- Open MP läuft nur mit Prozessoren mit gemeinsamen Speicher
- Erfordert einen Compiler, der OpenMP unterstützt
- Die Skalierbarkeit hängt von der Speicherarchitektur ab
- Zuverlässige Fehlerbahndlung ist durch die internen Thread nicht möglich
- Die genaue Zuordnung der Thread zur CPU wird nicht unterstützt
- Synchronization zwischen Teilmengen von Thread ist nicht erlaubt

```
for (i = n-1; i >= 0; i--) {
        delete [] b[i]; // Deallocate rows
}

delete [] b; // Deallocate row pointers
```