TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Technische Informatik, Professur Rechnerarchitektur

Konzepte der parallelen Programmierung

OpenMP: Sprachbeschreibung

Zellescher Weg 12

Willersbau A108

Tel. +49 351 - 463 - 34787

Bernd Trenkler (bernd.trenkler@tu-dresden.de)

ZIH
Zentrum für Informationsdienste und Hochleistungsrechnen

OpenMP: Überblick

OpenMP ist ein API (Application Programmer Interface) zum Schreiben von Multithreading-Anwendungen.

Die meisten Konstrukte in Open MP sind Compilerdirektive oder Pragmas.

Programmiersprache C und C++

#pragma omp Anweisung [Parameter [Parameter]...]

Programmiersprache Fortran

C$OMP Anweisung [Parameter [Parameter]...]
!$OMP Anweisung [Parameter [Parameter]...]
***$OMP Anweisung [Parameter [Parameter]...]**

Darüber hinaus stellt OpenMP noch einen Satz von Library-Routinen zur Verfügung.

OpenMP: Überblick

Anweisungen von Open MP

Parallele Regionen

Lastverteilung

Daten-Umgebung

Synchronisation

Laufzeit-Funktionen und Umgebungsvariable

OpenMP: Parallele Regionen

Basisfunktionen

```
omp_set_num_threads(4)
#pragma omp parallel
    {
        structured block (strukturierter Block, z.B. i=8)
    }
```

Besonderheiten

natürliche Barriere an der Stelle, wo parallele Region in den Master-Thread übergeht

→ Dynamischer Mode (default mode)
→ Statischer Mode

OpenMP: Lastverteilung

Basisfunktionen

```
#pragma  omp  parallel
#pragma  omp  for
for (i=lower_bound; i op upper_bound; incr_expr)
        {
        Schleifenrumpf
        }
```

- op {<, <=, >, >=}

Möglichkeiten der Lastverteilung

- schedule(static, block_size)
- schedule(dynamic, block_size)
- schedule(guided, block_size)
- schedule(runtime)

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

1.) „static“ mit Angabe von „chunks“

```
#define STATIC_CHUNK 5
…
#pragma omp for schedule(static, STATIC_CHUNK)
```

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)
Iterationen pro Block = c = 5 (Letzter Block kann weniger als c sein)
Blöcke werden in Reihenfolge der Threadnummern verteilt (round-robin)

00000 11111 22222 00000 11111 22222

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

2.) „static“ ohne Angabe von „chunks“

#pragma omp for schedule(static)

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)
chunks = Iterationen pro Block = n/p = 30/3 = 10
Blöcke werden in Reihenfolge der Threadnummern verteilt (round-robin)

0000000000 1111111111 2222222222

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

3.) „dynamic“ mit Angabe von „chunks“

```
#define DYNAMIC_CHUNK 5
...
#pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
```

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)

chunks = Iterationen pro Block = c = 5

- Der Thread bekommt den neuen Block, der fertig ist
  (Für die erste Runde in Reihenfolge der Threadnummern verteilt)
- Letzter Block kann weniger als c sein

z.B.:

00000 11111 22222 11111 22222 11111
00000 11111 22222 22222 11111 00000
...

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

4.) „dynamic“ ohne Angabe von „chunks“

#pragma omp for schedule(dynamic)

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)

Iterationen pro Block = **c = 1**

- Der Thread bekommt den neuen Block, der fertig ist
  (Für die erste Runde in Reihenfolge der Threadnummern verteilt)

  012 02100112 usw.

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

5.) „guided“ mit Angabe von „chunks“

```
#define GUIDED_CHUNK 5
...
#pragma omp for schedule(guided, GUIDED_CHUNK)
```

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)

Iterationen pro Block = **c = 5**

Dynamisches Scheduling mit abnehmender Blockgröße

- Der Thread bekommt den neuen Block, der fertig ist
  (Für die erste Runde in Reihenfolge der Threadnummern verteilt)
- Berechnungsschema für die Blöcke:
  - anfangs n/p: 30/3=10 (Blockgröße >= c? → ja)
    1. Block: 10 Iterationen

OpenMP: Lastverteilung bei der Schleifenparallelisierung

- dann abnehmend
  - noch nicht bearbeitete Iterationen / Anzahl der Threads

    chunk entspricht der minimalen Bockgröße mit einer Ausnahme (letzten Block)

20/3 =7(obere Gauss-Klammer)
(Blockgröße >= c? → ja) 2. Block: 7 Iterationen
13/3=5
(Blockgröße >= c? → ja) 3. Block: 5 Iterationen
8/3 =3
(Blockgröße >= c? → nein)
(letzter Block? → nein) 4. Block: 5 Iterationen
3/3 =1
(Blockgröße >= c? → nein)
(letzter Block? → ja) 5. Block: 3 Iterationen

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

6.) „guided“ ohne Angabe von „chunks“

#pragma omp for schedule(guided)

Lösung: (0 = Thread Nr. 0, 1 = Thread Nr. 1 , 2 = Thread Nr. 2)

Iterationen pro Stück = c = 1

- Der Thread bekommt das neue Stück, der fertig ist
  (Für die erste Runde in Reihenfolge der Threadnummern verteilt)
- Berechnungsschema für die Blöcke:
  - anfangs n/p: 30/3=10 1.Block: 10 Iterationen
  - dann abnehmend
    - noch nicht bearbeitete Iterationen / Anzahl der Threads
    - 20/3 =7(obere Gauss-Klammer) 2. Block: 7 Iterationen
    - 13/3=5 3. Block: 5 Iterationen
    - 8/3 =3 4. Block: 3 Iterationen
    - 5/3 =2 5. Block: 2 Iterationen
    - 3/3 =1 6. Block: 1 Iterationen
    - 2/3 =1 7. Block: 1 Iterationen
    - 1/3 =1 8. Block: 1 Iterationen

OpenMP: Lastverteilung bei der Schleifenparallelisierung

n=30 Iterationen werden auf p=3 Threads verteilt

7.) „runtime“

#pragma omp for schedule(guided)

Darf **nicht** zusammen mit einem Chunk-Wert angegeben werden!

Umgebungsvariable OMP_SCHEDULE wird zur Laufzeit ausgewertet und di
e hier festgelegte Strategie realisiert.

z.B. vor Programmstart auf Konsole:

setenv OMP_SCHEDULE „dynamic, 4“ oder

setenv OMP_SCHEDULE „guided“

Ist Umgebungsvariable OMP_SCHEDULE nicht gesetzt:

→ Scheduling hängt von Implementierung der verwendeten OpenMP-Bibliothek ab.

OpenMP: Lastverteilung

kombinierte paralleles Lastverteilungs-Konstrukt

```
#pragma  omp parallel for
   for (i=lower_bound; i op upper_bound; incr_expr)
     {
     Schleifenrumpf
     }
```

Barriere nach Beendigung von #pragma omp for

Aufhebung durch: #pragma omp for nowait

#pragma omp single {structured block} innerhalb von #pragma omp for

→ **der entsprechende strukturierte Block wird nur durch einen Thread ausgeführt**

OpenMP: Lastverteilung (#pragma omp section)

```
#pragma omp parallel
{
#pragma omp sections
  {{      a=…;
          …   ;     }
#pragma omp section
  {       b=…;
          …   ;     }
#pragma omp section
  {       c=…;
          …   ;     }
#pragma omp section
  {       d=…;
          …         ;         }
  }/*omp end sections*/
}/*omp end parallel*/
```

OpenMP: Lastverteilung (workshare-Directive ab OpenMP 2.5)

- nur für Fortran

```
!$OMP WORKSHARE

    structured block

!$OMP END WORKSHARE [ NOWAIT ]
```

- dividiert die Ausführung eines strukturierten Blockes in einzelne “units of work”

- gut verwendbar bei Array-Anweisungen
  → “units of work” ist Anweisung für jedes Einzelelement

OpenMP: Lastverteilung (task-Directive ab OpenMP 3.0)

#pragma omp task [Parameter [Parameter] …]
Codeblock

Task

- ein Stück Arbeit ohne Datenaustausch
→ Anweisungen im Codeblock werden sequentiell abgearbeitet

- Verteilung der Tasks zwecks Abarbeitung erfolgt zur Laufzeit

- welcher Thread aus dem Team welche Task ausführt, spielt keine Rolle

OpenMP: Daten-Umgebung

shared

private

firstprivate

lastprivate

default

reduction

threadprivate

copyin

copyprivate (ab OpenMP 2.0)

OpenMP: Synchronisation

master

critical

barrier

atomic

flush

ordered

taskwait (ab OpenMP 3.0)

OpenMP: Laufzeit-Funktionen und Umgebungsvariable

omp_set_num_threads
omp_get_num_threads

omp_get_max_threads

omp_get_thread_num

omp_get_num_procs

omp_in_parallel

omp_set_dynamic
omp_get_dynamic

omp_set_nested
omp_get_nested

omp_get_wtime (ab OpenMP 2.0)
omp_get_wtick (ab OpenMP 2.0)