Pemrograman OpenMP (2)

Kuliah#8 TSK617 Pengolahan Paralel - TA 2011/2012

Eko Didik Widianto

Teknik Sistem Komputer - Universitas Diponegoro

Tentang Kuliah #8 Pemrograman OpenMP

- Akan dibahas tentang **pemrograman paralel dengan OpenMP** menggunakan kompiler directive
  - Arsitektur memori: shared (SMP, symmetric multi-processor)
  - Model programming: thread
- Pokok Bahasan: (kuliah #8 akan membahas item yang ditebalkan)
  1. Pengantar OpenMP
  2. Membuat Thread
  3. Sinkronisasi dengan *critical, atomic*
  4. **Loop dan Worksharing**
  5. **Sinkronisasi dengan *barrier, single, master, ordered***
  6. **Environment Data**
  7. Menjadwalkan *for* dan *section*
  8. Model memori
  9. OpenMP 3.0

Kompetensi Dasar

- Setelah mempelajari bab ini, mahasiswa akan mampu:
  1. [C2] Mahasiswa memahami konsep pemrograman paralel menggunakan OpenMP
  2. [C3] Mahasiswa akan mampu membuat program paralel dari program serial menggunakan compiler-directive dan pustaka-pustaka OpenMP
  3. [C5] Mahasiswa akan mampu memprogram suatu aplikasi komputasi matrik menggunakan OpenMP serta menghitung faktor speedupnya
- Link
  - Website: http://didik.blog.undip.ac.id/2012/02/25/kuliah-tsk-617-pengolahan-paralel-2011/
  - Email: didik@undip.ac.id

Acknowledment

- Materi dan gambar didapat dari:
  - Tim Mattson, Larry Meadows (2008): “A Hands-on Introduction to OpenMP“
  - Website: http://openmp.org/wp/resources/

Bahasan

Bahasan

SPMD vs Worksharing

- A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program
  - each thread **redundantly executes the same code**
- How do you split up pathways through the code between threads within a team?
  - This is called **worksharing**
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct

Bahasan

Loop Construct

- The loop workharing construct **splits up loop iterations** among the threads in a team
- Loop construct name:
  - C/C++: **for**
  - Fortran: **do**

```
#pragma omp parallel
{

    #pragma omp for
    for (i=0;i<N;i++){
       NEAT_STUFF(i);
    }

}
```

- The variable i is made “**private**” to each thread **by default**. You could do this **explicitly** with a “private(i)” clause

```
#pragma omp parallel private(i)
{

    #pragma omp for
    for (i=0;i<N;i++){
       NEAT_STUFF(i);
    }

}
```

Construct Worksharing Loop: Contoh

Sequential code

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel
{

    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1)*N/Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;i<iend;i++) {
        a[i] = a[i] + b[i];
    }

}
```

OpenMP parallel region and a worksharing for construct

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

Kombinasi Construct Paralel/Worksharing

- OpenMP shortcut: Put the “parallel” and the worksharing directive **on the same line**

```
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for(i=0;i<MAX;i++) {
      res[i] = huge();
    }
}
```

```
double res[MAX]; int i;
#pragma omp parallel for
    for(i=0;i<MAX;i++) {
      res[i] = huge();
    }
```

- Both codes are equivalent

Bahasan

Working with Loop

- **Basic approach:**
  1. Find **compute intensive loops**
  2. Make **the loop iterations independent** ..
     - So they can safely execute in any order without loop-carried dependencies
  3. **Place** the appropriate OpenMP directive and test

```
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
   j +=2;
   A[i] = big(j);
}
```

```
int i, A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
   int j = 7 + 2*i;
   A[i] = big(j);
}
```

- Remove loop dependency, i.e from variable j

Bahasan

Reduction

- How do we handle this case?

```
double ave=0.0, A[MAX]; int i;
for (i=0;i< MAX; i++) {
   ave + = A[i];
}
ave = ave/MAX;
```

- It is combining values into a single accumulation variable (**ave**)
  - There is a true dependence between loop iterations that **can't be trivially removed**
  - This is a very common situation
    - It is called a "**reduction**".
- Support for reduction operations is included in **most** parallel programming environments

Reduction

- OpenMP reduction clause:

```
reduction (op : list)
```

Op is an operator

- Inside a parallel or a work-sharing construct:
  - **A local copy** of each list variable is made and **initialized** depending on the “op” (e.g. 0 for “+”).
  - Compiler finds standard reduction expressions containing “**op**” and uses them to **update** the local copy.
  - Local copies are reduced into **a single** value and **combined** with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
   ave + = A[i];
}
ave = ave/MAX;
```

Reduction operands/initial-values

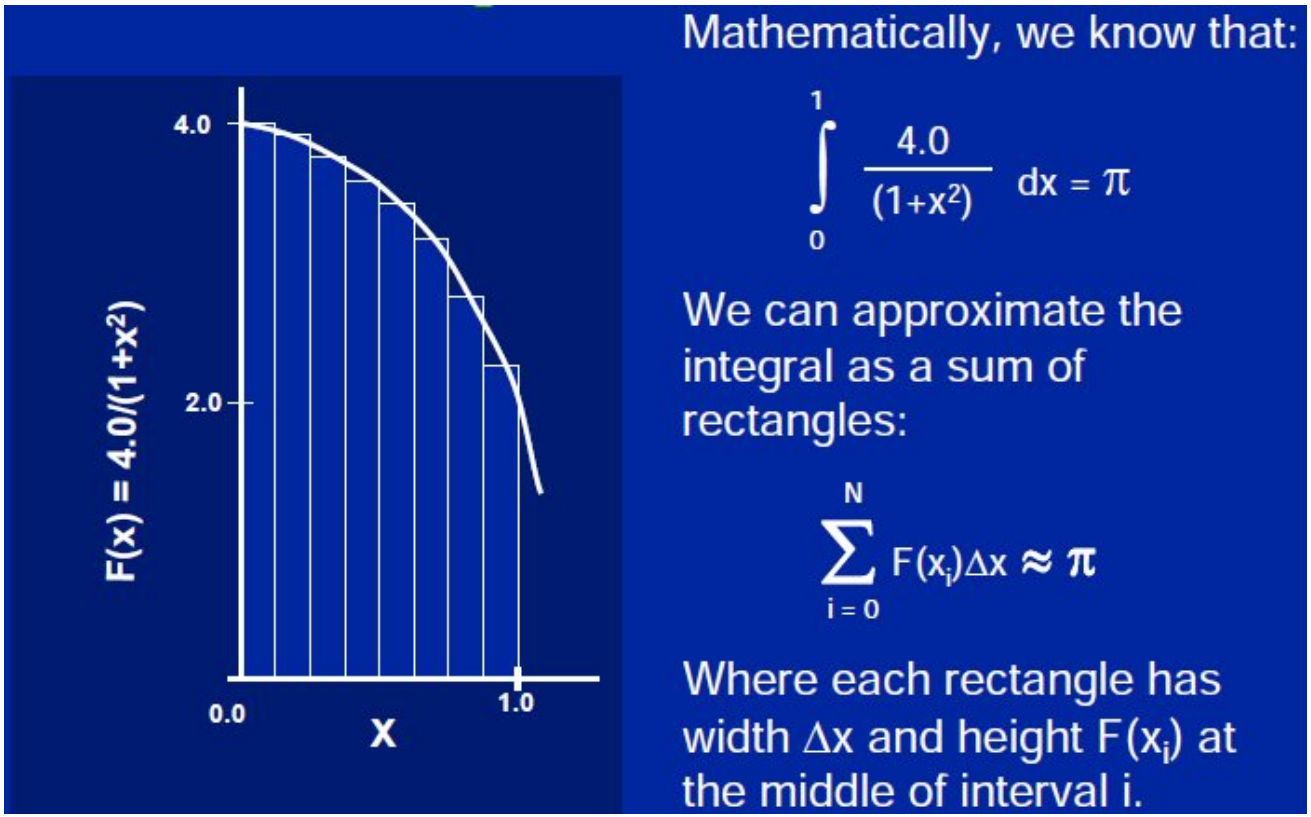
- Many different associative operands can be used with reduction
  - Initial values are the ones that make sense mathematically.

Operator	Initial Value
+	0
*	1
-	0

C/C++ Only	
Operator	Initial Value
&	~0
|	0
^	0
&&	1
||	0

Exercise

- Parallelize serial pi program with a loop construct
  - The goal is to **minimize the number changes** made to the serial program.



Parallelize Pi Program

```
/*Serial pi program*/
#include <stdlib.h>
#include <omp.h>
define num_steps 1000000;
double step;
int main (int argc, char *argv[])
{
  int i, j;
  double x, pi, sum = 0.0;
  step = 1.0/num_steps;
 for (i=0;i< num_steps; i++){
  x = (i+0.5)*step;
  sum += 4.0/(1.0+x*x);
  }
  pi = step * sum;
  return EXIT_SUCCESS;
}
```

```
/*Serial pi program*/
#include <stdlib.h>
#include <omp.h>
#define num_steps 1000000;
double step;
int main (int argc, char *argv[])
{
  int i, j;
  double x, pi, sum = 0.0;
  step = 1.0/num_steps;
    #pragma omp parallel for re-
duction(+:sum)
  for (i=0;i< num_steps; i++){
  x = (i+0.5)*step;
  sum += 4.0/(1.0+x*x);
  }
  pi = step * sum;
  return EXIT_SUCCESS;
}
```

Bahasan

Synchronization: Barrier

- **Barrier**: Each thread **waits** until all threads arrive

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id=omp_get_thread_num();
  A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
  for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
      // implicit barrier at the end of worksharing construct
#pragma omp for nowait
  for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
      // no implicit barrier due to nowait
  A[id] = big_calc4(id);
} // implicit barrier at the end of a parallel region
```

Bahasan

Master Construct

- The **master construct** denotes a structured block that is only executed by the **master** thread.
  - The other threads just skip it (**no synchronization** is implied).

```
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    { exchange_boundaries(); }
#pragma omp barrier
    do_many_other_things();
}
```

Bahasan

Single Worksharing Construct

- The single construct denotes a block of code that is executed by **only one thread** (not necessarily the master thread)
  - A barrier is implied at the end of the single block (can remove the barrier with a nowait clause)

```
#pragma omp parallel
{
    do_many_things();
#pragma omp single
    { exchange_boundaries(); }
    do_many_other_things();
}
```

Bahasan

Synchronization: Ordered

- The ordered region executes in the **sequential order**

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (i=0;i<N;i++){
  tmp = NEAT_STUFF(i);
#pragma ordered
  res += consum(tmp);
}
```

Synchronization: Lock Routines

- **Simple Lock routines**
  - A simple lock is available if it is unset
    omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- **Nested Locks**
  - A nested lock is available if **it is unset or if it is set** but owned by the thread executing the nested lock function
    omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()
- **Note**:
  - A lock implies a memory fence (a “flush”) of all thread visible variables
  - A thread always accesses the most recent copy of the lock, so you **don’t need** to use a flush on the lock variable

Sinkronisasi: Simple Lock

- Protect resources with locks

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  // wait here for your turn
  omp_set_lock(&lck);
  printf("%d %d", id, tmp);
  // release the lock so the next thread gets a turn
  omp_unset_lock(&lck);
}
// Free-up storage when done
omp_destroy_lock(&lck);
```

Bahasan

Rutin-rutin Pustaka (Runtime)

- Runtime environment routines
  - Modify/Check the number of threads
    omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    omp_in_parallel()
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    omp_set_dynamic(), omp_get_dynamic();
  - How many processors in the system?
    omp_num_procs()

Runtime Routine Usage

- To use a known, fixed number of threads in a program
  1. tell the system that you don't want dynamic adjustment of the number of threads
  2. set the number of threads
  3. save the number you got

```
#include <omp.h>
void main()
{
  int num_threads;
  // Disable dynamic adjustment of the num-
ber of threads
  omp_set_dynamic( 0 );
  // Request as many threads as you have processors
  omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
  {
    int id=omp_get_thread_num();
#pragma omp single
    num_threads = omp_get_num_threads();
    do_lots_of_stuff(id);
  }
}
```

Bahasan

Variabel Environment

- Set the default number of threads to use.

  `OMP_NUM_THREADS int_literal`

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled

  `OMP_SCHEDULE “schedule[, chunk_size]”`

Bahasan

Data Environment: Atribut Storage Default

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are **SHARED** among threads
  - **File scope** variables, **static**
  - dynamically allocated memory (**ALLOCATE**, **malloc**, **new**)
- But not everything is shared...
  - Stack variables in functions called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE

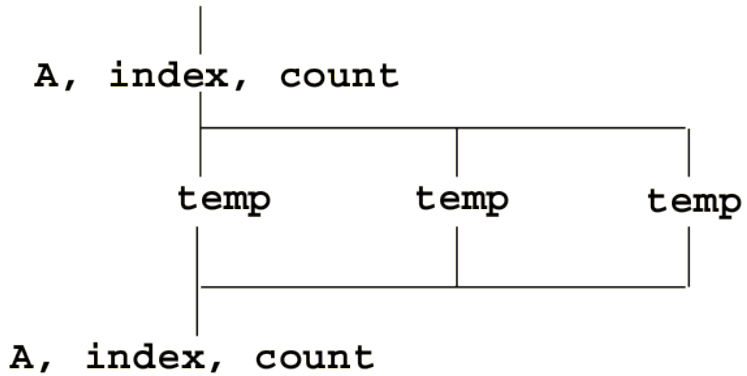
Data Sharing: Contoh

```
double A[10];
int main() {
    int index[10];
#pragma omp parallel
    work(index);
    printf("%d\n", index[0]);
}
```

- Function work() is implemented on different file from function main()

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

- A, index and count are shared by all threads
- temp is local to each thread



Bahasan

Data Sharing: Mengubah Atribut Storage

- One can selectively change storage attributes for constructs using the following clauses:
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)
    - DEFAULT(PRIVATE) is Fortran only
- All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.

Data Sharing: Clause Private

- **private(var)** creates a new local copy of var for each thread
  - The value is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

```
void wrong() {
  int tmp = 0;
#pragma omp for private(tmp)
  for (int j = 0; j < 1000; ++j)
  tmp += j; // tmp was not initialized
  printf("%d\n", tmp);
   // tmp: 0 in 3.0, unspecified in 2.5
}
```

Data Sharing: Clause Private

When is the original variable valid?

- The original variable's value is unspecified in OpenMP 2.5.
- In OpenMP 3.0, if it is referenced outside of the construct
- Implementations may reference the original variable or a copy
  - A dangerous programming practice!

```
int tmp;
void danger() {
  tmp = 0;
#pragma omp parallel pri-
vate(tmp)
  work();
  printf("%d\n", tmp); // tmp has un-
specified value
}
```

```
extern int tmp;
void work() {
  tmp = 5; // unspeci-
fied which copy of tmp
}
```

Data Sharing: Clause Firstprivate

- **Firstprivate** is a special case of private
  - **Initializes each private copy** with the corresponding value from the master thread

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    // Each thread gets its own tmp with an initial value of 0
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
     // tmp: 0 in 3.0, unspecified in 2.5
}
```

Data Sharing: Clause Lastprivate

- Lastprivate passes the value of a private from the last iteration to a global variable

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
    lastprivate(tmp)
    for (int j = 0; j < 1000; ++j)
        // Each thread gets its own tmp with an initial value of 0
        tmp += j;
    printf("%d\n", tmp);
      // tmp is defined as its value at the "last sequential" itera-
tion (i.e., for j=999)
}
```

Data Sharing: Testing Data Environment

- Consider this example of PRIVATE and FIRSTPRIVATE Variables A,B, and C = 1

```
#pragma omp parallel private(B) firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?
- Inside this parallel region
  - A is shared by all threads; equals 1
  - B and C are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1
- Outside this parallel region
  - The values of B and C are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct

Data Sharing: Clause Default

- The default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - Exception: ***#pragma omp task***
    - Task default storage is **firstprivate**
- To change default: **use** DEFAULT(PRIVATE), Only on Fortran
  - each variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **DEFAULT(NONE)**: no default for variables in static extent
  - Must list storage attribute for each variable in static extent
    - Good programming practice!

Data Sharing: Task (OpenMP 3.0)

- The default for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
     int C;
     compute(A, B, C);
   }
}
```

- A is **shared**, B is **firstprivate**, and C is **private**
- Task in detail will be presented in next lecture

Data Sharing: Threadprivate

- Makes global data private to a thread
  - In C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities)

Data Sharing: Contoh Threadprivate

- Use threadprivate to create a counter for each thread

```
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
    counter++;
    return (counter);
}
```

Penyalinan Data: Copyin

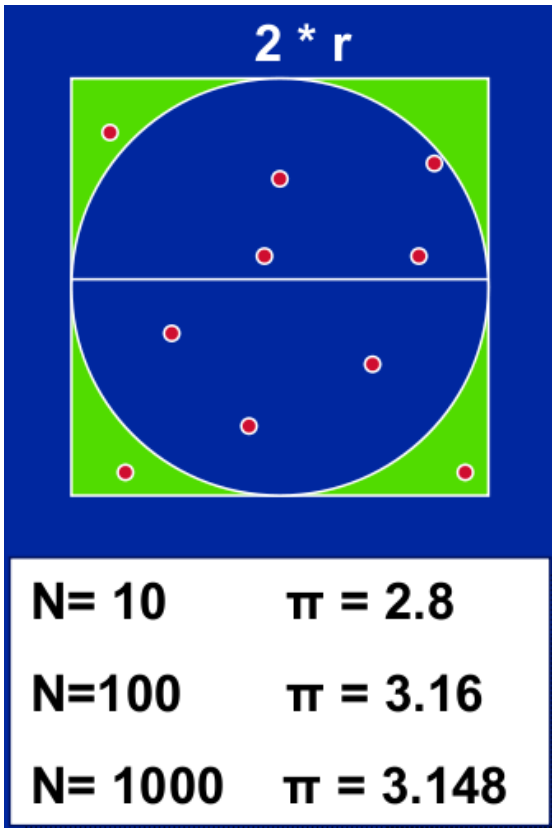
- Initialize threadprivate data using a copyin clause
- Used with a single region to broadcast values of privates from one member of a team to the rest of the team

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);
void main()
{
   int Nsize, choice;
#pragma omp parallel private (Nsize, choice)
  {
#pragma omp single copyprivate (Nsize, choice)
      input_parameters (Nsize, choice);
  }
}
```

Penghitungan Monte Carlo

Menggunakan bilangan random untuk memecahkan problem

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board



- Throw darts at the circle/square
- Chance of falling in circle is proportional to ratio of areas:
  - $A_c = r^2 * \pi$
  - $A_s = (2 * r) * (2 * r) = 4 * r^2$
  - $P = A_c / A_s = \pi / 4$
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute π

Latihan: Penghitungan Monte Carlo

- File untuk program serial:
  - pi_mc.c: program pi dengan metode monte carlo
  - random.c: generator random sederhana
  - random.h: file header untuk generator random
- Buat program paralel menggunakan OpenMP!

Lisensi

Creative Common Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

- Anda bebas:
  - untuk **Membagikan** — untuk menyalin, mendistribusikan, dan menyebarkan karya, dan
  - untuk **Remix** — untuk mengadaptasikan karya
- Di bawah persyaratan berikut:
  - **Atribusi** — Anda harus memberikan atribusi karya sesuai dengan cara-cara yang diminta oleh pembuat karya tersebut atau pihak yang mengeluarkan lisensi.
  - **Pembagian Serupa** — Jika Anda mengubah, menambah, atau membuat karya lain menggunakan karya ini, Anda hanya boleh menyebarkan karya tersebut hanya dengan lisensi yang sama, serupa, atau kompatibel.
- Lihat: Creative Commons Attribution-ShareAlike 3.0 Unported License