A brief look at OpenMP

Mats Brorsson

KTH School of Information and Communication Technology

1

Before we begin...

- **Who is mainly programming in:**
  - **C**
  - **C++**
  - **Java**
  - **C#**
  - **Other**
- **Who has experience with parallelization:**
  - **With pthreads/java threads?**
  - **OpenMP?**
  - **Other models?**

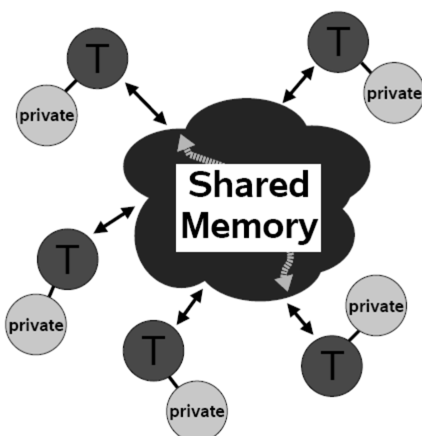
2

OpenMP

- A portable shared memory multiprocessing API based on compiler directives
  - Fortran 77/90, C, C++
  - Multi-vendor support for both Unix and NT
- Standardises fine grain (loop, task) parallelism
- Also support coarse grained algorithms
- http://www.openmp.org
- http://www.compunity.org
- It is NOT automatic parallelization

3

The OpenMP model



- All threads have the same access to the same globally shared memory
- Data can be shared or private
- Private data is accessible only by threads who owns it
- Data transfer is transparant to programmers
- Synchronization takes place but can be implicit

4

A first example

- **For-loop with independent iterations**

```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

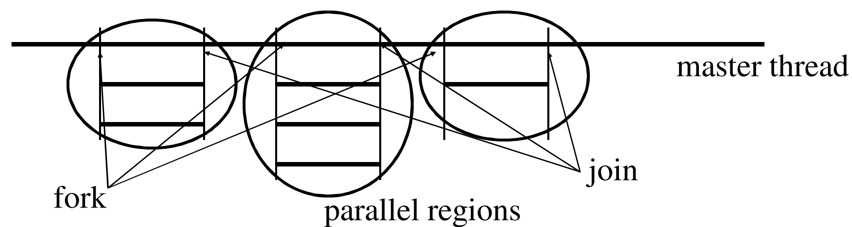
```
Thr 1       Thr 2
c[0]=...    c[4]=...
c[1]=...    c[5]=...
c[2]=...    c[6]=...
c[3]=...    c[7]=...
```

- **Parallelized version**

```
#pragma omp parallel for
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

5

OpenMP execution model



- **OpenMP has a fork-join execution model**
- **The program starts as any sequential program**
- **Threads are forked off at parallel regions**
- **Threads join after a parallel region**

6

OpenMP components

Directives
- Parallel regions
- Work sharing
- Tasks
- Synchronization
- Data-sharing attr.
  - Private
  - Shared
  - reduction
- Orphaning

Environment variables
- Number of threads
- Scheduling iterations
- Dynamic thread adjustment
- Nested parallelism

Runtime
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locks

7

A more elaborate example

```
#pragma omp parallel if (n>limit) default(none) \
  shared(n, a, b, c, x, y, z) private (f, i ,
  scale)
{
  f = 1.0;
#pragma omp for nowait
  for (i=0; i<n; i++)
      z[i] = x[i] + y[i];
#pragma omp for nowait
  for (i=0; i<n; i++)
      a[i] = b[i] + c[i];
#pragma omp barrier
  ...
  scale = sum(a,0,n) + sum(z,0,n) + f;
}
```


8

OpenMP – parallel regions

• **Example of a parallel region:**

```
main(){
  A;     /* Master thread executes */
#pragma omp parallel
  {             /* fork parallel threads */
    B;          /* Executes in parallel */
  }             /* join parallel threads */

  C;     /* Master thread executes */
}
```

• **How many threads are forked off?**

9

Degree of parallelism

- **The master threads has number 0**
- **The number of threads created is determined by:**
  - Environment variables
    - `setenv OMP_NUM_THREADS 8`
  - Library calls
    - `omp_set_num_threads(8);`
  - Clause in parallel construct
    - `num_threads(8)`

10

Work sharing constructs

- **`for`** loop parallelism
- **`sections`** functional parallelism
- **`single`** temporary sequential execution

11

for — loop parallelism

```
#pragma omp for
  for (i = 1; i < n; i++)
     b[i] = (a[i] + a[i-1]) / 2.0;
```

- **Must be in a parallel region**
- **The iterations are distributed among the parallel threads**
- **How are iterations distributed?**

12

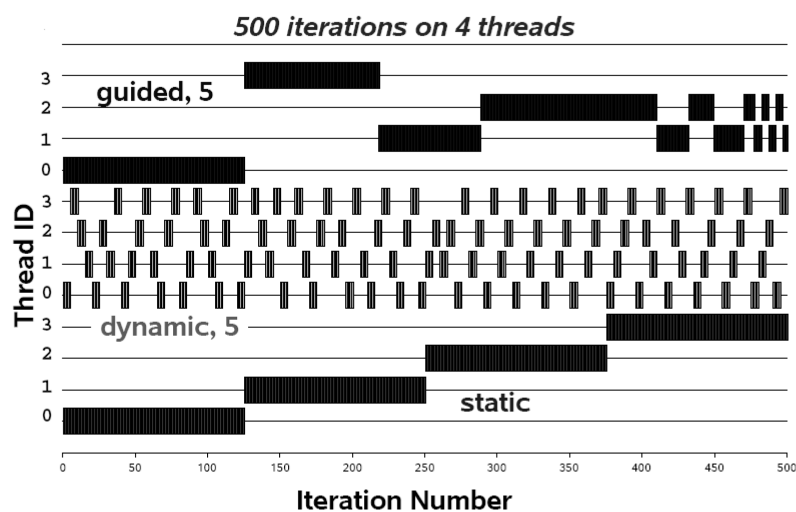
for — loop parallelism

```
#pragma omp for schedule(<schedule>, <chunk>)
  for (i = 1; i < n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
```

- **`<schedule>` Three different schedules can be specified**
  - static
  - dynamic
  - guided
- **Default is implementation dependent**
- **`<chunk>` Chunk size: sequence of iterations handled by one thread**

13

Scheduling effects



14

Sections — functional parallelism

```
#pragma omp sections
  {
  #pragma omp section
    {
     A;
    }
  #pragma omp section
    {
     B;
    }
  }
```

- Must be in a parallel region
- A and B are executed in parallel

15

Sequential execution in parallel code

```
#pragma omp single
{
  A;
}

#pragma omp master
{
  B;
}
```

- Must be in a parallel region
- Only one thread executes A
- Only master thread executes B

16

Communication between threads — Shared memory

- **Threads read and write shared variables**
  No need for explicit message-passing
- **Use *synchronisation* to protect against race conditions**
- **Change *storage attributes* to minimise synchronisation**

17

Synchronisation

- **Mutual exclusion**
- **Atomic**
- **Barrier synchronisation**

18

Mutual exclusion

```
#pragma omp critical
  {
    sum = sum + local_sum;
    …
  }
```

- Must be in a parallel region
- The block after the critical directive is executed by only one thread at a time

19

Atomic

```
#pragma omp atomic
  sum += f(x);
```

- Almost the same as `critical`
- Ensures that the updating of one memory location (sum) is atomic
- The execution of f(x) is not atomic
- Only one statement is allowed

20

Barrier synchronisation

```
#pragma omp barrier
```

- **Must be in a parallel region**
- **Each thread waits until all threads have reached this point**
- **`for`, `sections` and `single` directives end implicitly with a barrier**

21

Why barrier?

- **Suppose we run each of these two loops in parallel over i:**

```
for (i=0; i < N; i++)
       a[i] = b[i] + c[i];
for (i=0; i < M; i++)       // M < N
       d[i] = a[i] + b[i];
```

- **This might give us wrong answer (one day)**

Why?

22

Why barriers

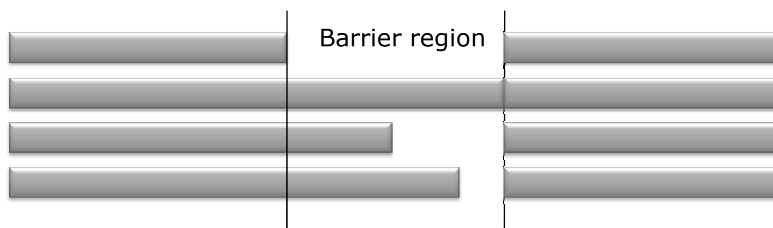
- **We need to have updated all of a[ ] first, before using a[ ]**

```
for (i=0; i < N; i++)
      a[i] = b[i] + c[i];
_______________________________________   Barrier
for (i=0; i < M; i++)      // M < N
      d[i] = a[i] + b[i];
```

- **All threads wait at the barrier point and only continue when all threads have reached the barrier point.**

23

Load imbalance



- **All threads have to wait for the slowest thread**
- **Barriers are easy to understand but may lead to performance bugs**
- **Use them with care**

24

Storage

- **There are three types of storage in a sequential program**
  - ***Stack data* — Local variables declared in a function or a subroutine**
  - ***Static data* — Global variables declared at the outermost level of the program**
  - ***Dynamic data* — data allocated with *new* or *malloc*. Referenced through pointers.**

25

Storage attributes

- **Data can be:**
  - ***shared* among threads, or**
  - ***private* to a thread**
- **Data is shared by default**
- **Stack data in functions called from a parallel region are private**
- **Data can also be declared private by means of a storage attribute**

26

Storage attributes — private

```
int B;
B = 10;
#pragma omp parallel private(B)
   B = … ;
```

- **A private *uninitialised* copy of B is created before the parallel region begins**
- **B is *not* the same within the parallel region as outside**

27

Storage attributes — firstprivate

```
int B;
B = 10;
#pragma omp parallel firstprivate(B)
   B = B + … ;
```

- **A private *initialised* copy of B is created before the parallel region begins**
- **The copy of each thread gets the same value**

28

Reduction operators

```
#pragma omp parallel
{
   sum = 0;
   #pragma omp for reduction(+: sum)
   for (i = 0; i < N; i++) {
      …
      sum = sum + i*f(i);
   }
}
```

- **A reduction operation keeps *local copies* of a variable and *combines* them into one scalar at the end of a for-operation**

29

OpenMP example: Computing π

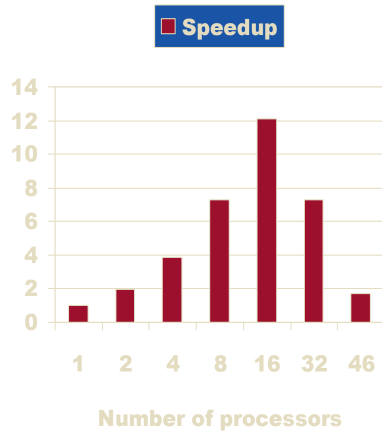
```
#define N 100000000

main(){
   double t, pi=0.0, w;
   long i;
   w = 1.0 / N;
#pragma omp parallel private(t)
#pragma omp for reduction(+: pi)
   for (i=0; i < N; i++) {
      t = (i + 0.5) * w;
      pi = pi + 4.0 / (1.0 + t*t);
   }
   printf("pi is %f\n", pi*w);
}
```

Storage attribute

Reduction operator

30

π — Speedup



- **SGI Origin 2000**
- **46 MIPS R10000 250 MHz**
- **Good speedup with up to 16 processors on a *loaded machine***

31

OpenMP example: Computing π again

Equivalent to a reduction operation

```
#define 100000000
main(){
    double t, pi=0.0, w;
    long i;
    w = 1.0 / N;
#pragma omp parallel private(t)
    {
        int local_pi = 0.0;
    #pragma omp for
        for (i=0; i < N; i++) {
            t = (i + 0.5) * w;
            local_pi = local_pi + 4.0 / (1.0 + t*t);
        }
    #pragma omp atomic
        pi += local_pi;
    }
    printf("pi is %f\n", pi*w);
}
```

32

OpenMP memory model

- **Relaxed-consistency, shared memory**
- **Each thread is allowed to have its own *temporary view* of memory**
  - **Important for performance reasons and compiler optimizations**
- **Single variable accesses are not guaranteed to be atomic**
- **The temporary view of memory needs to be synchronized with flush operations (explicit or implicit)**

33

Flush

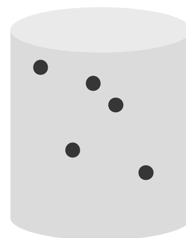
- **Explicit: `#pragma omp flush [(list)]`**
- **Implicit:**
  - **Barrier**
  - **Entry and exit from parallel region**
  - **Entry and exit from critical region**
  - **Exit from for/sections/single**

34

Task parallelism

- **Most important change in OpenMP 3.0**
- **Conceptual task pool associated with a parallel region**

Encountering threads add task to pool



Threads execute task in pool

35

Task construct

- **Syntax:**

```
#pragma omp task [clause [, clause]]
   structured block
```

- **Adds a task to the pool of tasks**
- **Each thread in a team of threads (created at a parallel region) executes an *implicit* task**

36

Task synchronization

- Syntax:

```
#pragma omp taskwait
```

- Current task is suspended until all children tasks, generated within the current task up to this point, are complete

Example task program

- List traversal

```
p = listhead ;
while (p) {
      process (p) ;
      p=next (p) ;
}
```

- Difficult to do in parallel with OpenMP 1.0-2.5

Example task program

- **List traversal**

```
#pragma omp parallel
{
  #pragma omp single private(p)
  {
      p = listhead ;
      while (p) {
             #pragma omp task
                   process (p) ;
             p=next (p) ;
      }
  }
}
```

39

LU-decomposition: another tasking example

```
1 int sparseLU ( ) {
2 int ii, jj, kk;
3 #pragma omp parallel
4 #pragma omp single nowait
5  for (kk=0; kk<NB; kk++) {
6     lu0(A[kk][kk]);
7
8 /* fwd phase */
9     for (jj=kk+1; jj < NB; jj++)
10        if(A[kk][jj] != NULL)
11 #pragma omp task
12        fwd(A[kk][kk], A[kk][jj]);
13 /* bdiv phase */
14        for (ii=kk+1; ii<NB; ii++)
15           if(A[ii][kk] != NULL)
16 #pragma omp task
17         bdiv(A[kk][kk],A[ii][kk]);
18 #pragma omp taskwait
19
20 /* bmod phase */
21   for (ii=kk+1; ii<NB; ii++)
22 ´   if (A[ii][kk] != NULL)
23       for (jj=kk+1; jj<NB; jj++)
24          if (A[kk][jj]!= NULL)
25 #pragma omp task
26         {
27           if (A[ii][jj]==NULL)
28             A[ii][jj]=
               allocatecleanblock();
29           bmod(A[ii][kk],
              A[kk][jj], A[ii][jj]);
30         }
31 #pragma omp taskwait
32    }
33 }
```

40

Summary

- **OpenMP is a small, yet powerful, programming model for multicore processors**
- **It is now a de-facto standard that all compiler vendors support (including gcc)**
- **With OpenMP 3.0 the general usability is greatly enhanced**
  - **Explicit threads are evil**
  - **Tasks are good**

41