OpenMP: a shared-memory parallel programming model

Eduard Ayguadé
Computer Sciences Department Associate Director (BSC)
Professor of the Computer Architecture Department (UPC)

First definition in 1996
- Today, industry standard, main vendors support it

Advantages
- Easy to program, debug, modify and maintain
- Incremental parallelization from the beginning
  - Improve programming productivity
- Neither communication nor data distribution needed

Language extensions to Fortran77/90 and C/C++
- Directives or pragmas that can be ignored when compiled in sequential
- Intrinsic function in OpenMP library
- Environment variables
Three components of OpenMP

- **OMP directives/pragmas**
  - These form the major elements of OpenMP programming, they
    - Create threads
    - Share the work amongst threads
    - Synchronize threads

- **Library routines**
  - These routines can be used to control and query the parallel execution environment such as the number of processors that are available for use

- **Environment variables**
  - The execution environment such as the number of threads to be made available to an OMP program can also be set at the operating system level before the program execution is started (an alternative to calling library routines)

PARALLEL region construct

- **Specification of parallel region**
  - C$OMP [END] PARALLEL [clause[,... clause]]
  - #pragma omp parallel [clause[clause]]

- **Execution model:**
  - When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The number of threads in a team remains constant for the duration of the parallel region
  - Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program
Some useful intrinsic functions

- To identify individual threads by number
  - Fortran:
    
    ```
    INTEGER FUNCTION OMP_GET_THREAD_NUM()
    ```
  - C/C++:
    
    ```
    int omp_get_thread_num(void)
    ```
  - Returns value between 0 ... `OMP_GET_NUM_THREADS()` - 1

- To find out how many threads are being used
  - Fortran:
    
    ```
    INTEGER FUNCTION OMP_GET_NUM_THREADS()
    ```
  - C/C++:
    
    ```
    int omp_get_num_threads(void);
    ```
  - Returns value 1 if outside the parallel region else the number of threads available

PARALLEL region construct

- Each thread executes the same code redundantly
  ```
  double A[1000];
  omp_set_num_threads(4); #pragma omp parallel
  |
  int ID = omp_get_thread_num();
  pooh(ID, A);
  |
  printf("all done\n");
  ```

A single copy of A is shared between all threads

Threads wait here for all threads to finish before proceeding (i.e. a barrier)
**PARALLEL region construct**

- **Clauses:**
  - `NUM_THREADS(integer_exp), IF(logical_exp), PRIVATE(list), SHARED(list), FIRSTPRIVATE(list), REDUCTION({operator|intrinsic}:list), COPYIN(list)`

- **Number of threads at each level:**
  - Environment variable OMP_NUM_THREADS
  - Intrinsic function `omp_set_num_threads` (in serial part)
  - `NUM_THREADS` clause

**Number of threads examples:**
- Environment variable: `setenv OMP_NUM_THREADS=3`
- Intrinsic function: `omp_set_num_threads(3)`
- Serial region: `omp_set_num_threads(3)`

**First example: computation of PI**

Mathematically, we know that:

\[
\int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi
\]

We can approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi
\]

Where each rectangle has width \(\Delta x\) and height \(F(x_i)\) at the middle of interval \(i\).
static long num_steps = 100000;
double step;
void main ()
{
    int i;
double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=1; i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}

#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i, id;
double x, pi, sum;
    step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel private(x, i, id) reduction(+:sum)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
Work distribution

- Work sharing constructs
  - Split up loop iterations among the threads in the team
  - Give a different structured block to each thread in the team
  - Give a structured block to just one thread in the team

Work distribution: DO loops

- Syntax:
  - `#pragma for [clause[clause]...]
  - `C$OMP [END] DO [clause[[], clause]...]

- Clauses:
  - Data scope: `PRIVATE(list), `LASTPRIVATE(list), `FIRSTPRIVATE(list), `REDUCTION(list)
  - Iteration scheduling: `SCHEDULE(type[,chunk])
  - Synchronization: `NOWAIT, `ORDERED
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1; i<=num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
INTEGER i, iter, time

factor=1/1.0000001
DO iter=1,5
C$OMP PARALLEL DO SCHEDULE(STATIC)
C$OMP& \text{SHARED(dummy) PRIVATE(i, time)}
DO i=0,N
\quad dummy(i)= dummy(i)*factor
\quad time = i/100
\quad call delay(time)
ENDDO
ENDDO
END

---

PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
INTEGER i, iter, time

factor=1/1.0000001
DO iter=1,5
C$OMP PARALLEL DO SCHEDULE(DYNAMIC)
C$OMP& \text{SHARED(dummy) PRIVATE(i, time)}
DO i=0,N
\quad dummy(i)= dummy(i)*factor
\quad time = i/100
\quad call delay(time)
ENDDO
ENDDO
END
Synthetic example: work unbalance

PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
INTEGER i, iter, time
factor=1/1.0000001
DO iter=1,5
C$OMP PARALLEL DO SCHEDULE(DYNAMIC)
C$OMP&     SHARED(dummy) PRIVATE(i, time)
DO i=0,N
    dummy(i)= dummy(i)*factor
    time = i/100
    call delay(time)
ENDDO
ENDDO
END

Low unbalance
High overhead

Synthetic example: work unbalance

PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
INTEGER i, iter, time
factor=1/1.0000001
DO iter=1,5
C$OMP PARALLEL DO SCHEDULE(DYNAMIC, 50)
C$OMP&     SHARED(dummy) PRIVATE(i, time)
DO i=0,N
    dummy(i)= dummy(i)*factor
    time = i/100
    call delay(time)
ENDDO
ENDDO
END

Less overhead
Some imbalance:
    Heavy chunks towards the end

Less overhead
Some imbalance:
    Heavy chunks towards the end
Synthetic example: work unbalance

- **Less overhead**
- **Good load balance:**
  - Heavy chunks towards the beginning
- **Dynamic:**
  - Non repetitive pattern

```fortran
PROGRAM test
PARAMETER (N=1024)
REAL dummy(N), factor
INTEGER i, iter, time
factor=1/1.0000001
DO iter=1,5
  C$OMP PARALLEL DO SCHEDULE(GUIDED)
  C$OMP&         SHARED(dummy) PRIVATE(i, time)
  DO i=0,N
    dummy(i)= dummy(i)*factor
    time = i/100
    call delay(time)
  ENDDO
ENDO
ENDDO
END
```
**SECTIONS: worksharing construct that gives a different structured block to each thread in a team**

**Fortran:**

```
$OMP SECTIONS [clause [,] clause ...]
  :
$OMP SECTION
  :
$OMP END SECTIONS [clause]
```

**C/C++:**

```
#pragma omp sections [clause [clause] ...]
{  
#pragma omp section
  [structured-block]
#pragma omp section
  [structured-block]
  :
}
```

---

**Example**

```c
 !$OMP PARALLEL
 !$OMP SECTIONS
 !$OMP SECTION
call init(x)
call processA(x)
 !$OMP SECTION
call init(y)
processB(y)
 !$OMP SECTION
call init(z)
processC(z)
 !$OMP END SECTIONS
 !$OMP END PARALLEL
```
Work distribution: SINGLE

- SINGLE: worksharing that gives a structured block to a single thread in a team

Example

```c
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    { input(y);
    }
    work(x,y);
}
```

How do threads interact?

- OpenMP is a shared memory model
  - Threads communicate by sharing variables

- Unintended sharing of data causes race conditions
  - Race condition: when the program’s outcome changes as the threads are scheduled differently

- To control race conditions
  - Use synchronization to protect data conflicts

- Synchronization is expensive so
  - Change how data is accessed to minimize the need for synchronization
OpenMP synchronization constructs

- **Mutual exclusion:**
  
  `C$OMP [END] CRITICAL [(name)]`  
  `#pragma critical [(name)]`

- **Atomic execution:**
  
  `C$OMP ATOMIC`  
  `#pragma atomic`

- **Barrier synchronization:**
  
  `C$OMP BARRIER`  
  `#pragma barrier`

- **Ordered execution for loops:**
  
  `C$OMP [END] ORDERED`  
  `#pragma ordered`

---

Synchronization: CRITICAL sections

- **Only one thread at a time can enter a critical section**

  ```c
  float res;
  #pragma omp parallel
  {
    float B;  int i;
    #pragma omp for
    for(i=0; i<niters; i++){
      B = big_job(i);
      #pragma omp critical
      consum (B, RES);
    }
  }
  ```

  Threads wait their turn – only one at a time calls consum
Synchronization: ATOMIC access

- Atomic is a special case of a critical section that can be used for certain simple statements.
- It applies only to the access to a memory location (the read and update of X in the following example).

```
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
C$OMP ATOMIC
    X = X + B
C$OMP END PARALLEL
```

- Makes use of special instructions in the processor.

Synchronization: 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);
    #pragma omp for nowait
    for(i=0;i<N;i++) B[i]=big_calc2(C,i);
    A[id] = big_calc3(id);
}
```

- Implicit barrier at the end of a for worksharing.
- No implicit barrier due to nowait.
- Implicit barrier at the end of parallel region.
The ordered construct enforces the sequential order for a block (in the example, following the lexicographical iteration ordering)

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

The flush construct denotes a sequence point where a thread tries to create a consistent view of memory
- All memory operations (both reads and writes) defined prior to the sequence point must complete
- All memory operations (both reads and writes) defined after the sequence point must follow the flush
- Variables in registers or write buffers must be updated in memory

Arguments to flush specify which variables are flushed
- No arguments specifies that all thread visible variables are flushed.
This example shows how FLUSH is used to implement pair-wise synchronization

```fortran
integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
   IAM = OMP_GET_THREAD_NUM()
   ISYNC(IAM) = 0
C$OMP BARRIER
   CALL WORK()
   ISYNC(IAM) = 1 ! I’m all done; signal this to others
C$OMP FLUSH(ISYNC)
   DO WHILE (ISYNC(NEIGH) .EQ. 0)
      C$OMP FLUSH(ISYNC)
   END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write

Make sure the read picks up the latest copy from memory

OpenMP has survived with no support for this kind of parallelization strategies until now (version 2.5)
Dynamic work generation schemes

Decouples work generation and execution
- One thread generates all work
- Amount of work unknown

Intel extension to 2.0

Another example: handling recursivity

```c
... C$OMP SINGLE
    CALL traverse(1, list, next)
C$OMP END SINGLE
...

SUBROUTINE traverse(i, list, next)
    INTEGER i, list(100), next(100)
    INTEGER res
C$OMP TASK
    CALL compute(list, list(i), res)
C$OMP CRITICAL
    total = total + res
C$OMP END CRITICAL
C$OMP END TASK
IF (next(i) .NE. 0) THEN
    CALL traverse(next(i), list, next)
END IF
END
```

OpenMP extension in 3.0