A brief introduction to OpenMP

Alejandro Duran

Barcelona Supercomputing Center
Outline

1. Introduction
2. Writing OpenMP programs
3. Data-sharing attributes
4. Synchronization
5. Worksharings
6. Task parallelism
It’s an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines

- Current version is 3.1 (June 2010)
- Supported by most compiler vendors
  - Intel, IBM, PGI, Oracle, Cray, Fujitsu, HP, GCC, ...
- Natural fit for multicores as it was designed for SMPs

Maintained by the Architecture Review Board (ARB), a consortium of industry and academia

http://www.openmp.org
A bit of history

- OpenMP Fortran 1.0: 1997
- OpenMP C/C++ 1.0: 1998
- OpenMP Fortran 1.1: 1999
- OpenMP C/C++ 2.0: 2000
- OpenMP Fortran 2.0: 2002
- OpenMP C/C++ 2.0: 2002
- OpenMP 2.5: 2005
- OpenMP 3.0: 2008
- OpenMP 3.1: 2010
Target machines

Shared Multiprocessors

- Chip
- Chip
- Chip

Memory interconnect

Memory
Memory is shared across different processors
Communication and synchronization happen *implicitly* through shared memory
Multicores/SMTs

- Core
- L1 Caches
- L2 Cache
- Off-chip Cache
- Memory

Chip
More commonly

- Access to memory addresses is not uniform
- Memory migration and locality are very important
Why OpenMP?

- Mature standard and implementations
  - Standardizes practice of the last 20 years
- Good performance and scalability
- Portable across architectures
- Incremental parallelization
- Maintains sequential version
- (mostly) High level language
  - Some people may say a medium level language :-)
- Supports both task and data parallelism
- Communication is implicit
Why not OpenMP?

- Communication is implicit
  - beware false sharing
- Flat memory model
  - can lead to poor performance in NUMA machines
- Incremental parallelization creates false sense of glory/failure
- No support for accelerators
- No error recovery capabilities
- Difficult to compose
- Pipelines are difficult
Outline

1. Introduction
2. Writing OpenMP programs
3. Data-sharing attributes
4. Synchronization
5. Worksharings
6. Task parallelism
OpenMP at a glance

OpenMP components

- Constructs
- Compiler
- OpenMP Exec
- OpenMP API
- Environment Variables
- OpenMP Runtime Library
- ICVs
- OS Threading Libraries
- SMP
- CPU (6)

Alex Duran (BSC)  
A brief introduction to OpenMP  
10th October 2011  
13 / 47
OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- `!$OMP` or `C$OMP` or `*$OMP` in fixed format
- `!$OMP` in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not recognize OpenMP
Hello world!

Example

```c
int id;
char *message = "Hello, world!";

#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    printf("Thread_%d_says:_s\n", id, message);
}
```
Hello world!

Example

```c
int id;
char *message = "Hello world!";

#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    printf("Thread_%d_says:_%s\n", id, message);
}
```

- Creates a parallel region of `OMP_NUM_THREADS`
- All threads execute the same code
- id is private to each thread
- message is shared among all threads
Hello world!

Example

```c
int id;
char *message = "Hello world!"; #pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    printf("Thread_%d_says:%s\n", id, message);
}
```

- `id` is private to each thread
- Each thread gets its id in the team
- `message` is shared among all threads
Writing OpenMP programs

Hello world!

Example

```c
int id;
char *message = "Hello world!";

#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    printf("Thread_%d_says:_%s\n", id, message);
}
```

message is shared among all threads
Execution model

Fork-join model

- OpenMP uses a **fork-join** model
  - The **master** thread spawns a **team** of threads that joins at the end of the parallel region
  - Threads in the same team can **collaborate** to do work
Memory model

- OpenMP defines a weak relaxed memory model
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
    - synchronization constructs, parallelism creation points, ...
  - Luckily, the default points are usually enough
- Variables can have shared or private visibility for each thread
Outline

1. Introduction
2. Writing OpenMP programs
3. Data-sharing attributes
4. Synchronization
5. Worksharings
6. Task parallelism
When creating a new parallel region (and in other cases) a new data environment needs to be constructed for the threads. This is defined by means of clauses in the construct:

- `shared`
- `private`
- `firstprivate`
- `default`
- `threadprivate`  

Not a clause!
Data-sharing attributes

**Shared**

When a variable is marked as `shared` all threads see the same variable

- Not necessarily the same value
- Usually need some kind of synchronization to update them correctly

**Private**

When a variable is marked as `private`, the variable inside the construct is a new variable of the same type with an undefined value.

- Can be accessed without any kind of synchronization
Data-sharing attributes

Firstprivate

When a variable is marked as `firstprivate`, the variable inside the construct is a new variable of the same type but it is initialized to the original variable value.

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization
Data-sharing attributes

Example

```c
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \ 
num_threads(2)
{
    x++; y++; z++;
    printf("%d\n",x);
    printf("%d\n",y);
    printf("%d\n",z);
}
```

The parallel region will have only two threads
Prints 2 or 3. Unsafe update!
Prints any number
Prints 2
Data-sharing attributes

Example

```c
int x=1, y=1, z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z)
num_threads(2) {
    x++; y++; z++;
    printf("%d\n",x);
    printf("%d\n",y);
    printf("%d\n",z);
}
```

The parallel region will have only two threads
Data-sharing attributes

Example

```c
int x=1, y=1, z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \ 
num_threads(2)
{
    x++; y++; z++;
    printf("%d\n", x);
    printf("%d\n", y);
    printf("%d\n", z);
}
```

Prints 2 or 3. Unsafe update!
Data-sharing attributes

Example

```c
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \ 
num_threads(2)
{
    x ++; y ++; z ++;
    printf("%d\n",x);
    printf("%d\n",y);
    printf("%d\n",z);
}
```

Prints any number
Data-sharing attributes

Example

```c
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \
num_threads(2)
{
    x++; y++; z++;
    printf("%d\n",x);
    printf("%d\n",y);
    printf("%d\n",z);
}
```

Prints 2 or 3. Unsafe update!
Threadprivate storage

The threadprivate construct

- How to parallelize:
  - Global variables
  - Static variables
  - Class-static members

- Use threadprivate storage
  - Allows to create a per-thread copy of “global” variables.
Threaprivate storage

Example

```c
char* foo ()
{
    static char buffer[BUF_SIZE];
    #pragma omp threadprivate(buffer)
    ...
    return buffer;
}
```

Creates one *static* copy of *buffer* per thread.
Data-sharing attributes

Threaprivate storage

Example

```c
char* foo ()
{
    static char buffer[BUF_SIZE];
    #pragma omp threadprivate(buffer)
    ...
    return buffer;
}
```

Now `foo` can be called by multiple threads at the same time
Threaprivate storage

Example

```c
char* foo ()
{
    static char buffer[BUF_SIZE];
    #pragma omp threadprivate(buffer)

    ...

    return buffer;
}
```

Simpler than redefining the interface. More costly.
Why synchronization?

Mechanisms

Threads need to synchronize to impose some ordering in the sequence of actions of the threads. OpenMP provides different synchronization mechanisms:

- barrier
- critical
- atomic
- taskwait
- low-level locks
Synchronization

Barrier

Example

```c
#pragma omp parallel
{
  foo();
  #pragma omp barrier
  bar();
}
```

Synchronizes all threads of the team

Forces all `foo` occurrences to happen before all `bar` occurrences.
Synchronization

Barrier

Example

```c
#pragma omp parallel
{
  foo();
  #pragma omp barrier
  bar();
}
```

Forces all **foo** occurrences too happen before all **bar** occurrences
Critical construct

Example

```c
int x=1;
#pragma omp parallel num_threads(2)
{
  #pragma omp critical
  x++;  
}
printf("%d\n",x);
```
Synchronization

Critical construct

Example

```c
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;
    x++;
}
printf("%d\n",x);
```

Only one thread at a time here
Synchronization

Critical construct

Example

```c
int x=1;
#pragma omp parallel num_threads(2)
{
    #pragma omp critical
    x ++;
    x ++;
}
printf("%d\n",x);
```

Only one thread at a time here

Prints 3!
Atomic construct

Example

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
  x++;
}
printf("%d\n", x);
```
Synchronization

Atomic construct

Example

```c
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
  x ++;
}
printf("%d\n",x);
```

Only one thread at a time updates x here
Atomic construct

Example

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
    #pragma omp atomic
    x ++;
}
printf("%d\n", x);
```

Specially supported by hardware primitives
Atomic construct

Example

```c
int x = 1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf("%d\n", x);  // Prints 3!
```
OpenMP provides **lock** primitives for low-level synchronization

- `omp_init_lock`: Initialize the lock
- `omp_set_lock`: Acquires the lock
- `omp_unset_lock`: Releases the lock
- `omp_test_lock`: Tries to acquire the lock (won’t block)
- `omp_destroy_lock`: Frees lock resources
Worksharings

Worksharing constructs divide the execution of a code region among the threads of a team

- Threads *cooperate* to do some work
- Better way to split work than using thread-ids

In OpenMP, there are four worksharing constructs:
- loop worksharing
- single
- section
- workshare

Restriction: worksharings cannot be nested
The for construct

Example

```c
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel
    #pragma omp for private(j)
    for ( i = 0; i < N; i++ )
        for ( j = 0; j < M; j++ )
            m[i][j] = 0;
}
```
The for construct

Example

```c
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel
    #pragma omp for private(j)
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            m[i][j] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop.
The for construct

Example

```c
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel
    #pragma omp for private(j)
    for ( i = 0; i < N; i++ )
        for ( j = 0; j < M; j++ )
            m[i][j] = 0;
}
```

Loop iterations must be independent
The for construct

Example

```c
void foo (int *m, int N, int M)
{
    int i;
    #pragma omp parallel
    #pragma omp for private (j)
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < M; j++)
            m[i][j] = 0;
    }
}
```

The $i$ variable is automatically privatized.
The for construct

Example

```c
void foo ( int *m, int N, int M)
{
    int i;
    #pragma omp parallel
    #pragma omp for private(j)
    for ( i = 0; i < N; i ++ )
        for ( j = 0; j < M; j ++ )
            m[i][j] = 0;
}
```

Loop iterations must be independent
The `i` variable is automatically privatized
Must be explicitly privatized
The reduction clause

Example

```c
int vector_sum (int n, int v[n])
{
    int i, sum = 0;
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        sum += v[i];
    return sum;
}
```

Common pattern. All threads accumulate to a shared variable
The reduction clause

Example

```c
int vector_sum (int n, int v[n])
{
    int i, sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++)
        sum += v[i];
    return sum;
}
```

Efficiently solved with the `reduction` clause
The reduction clause

Example

```c
int vector_sum (int n, int v[n])
{
    int i, sum = 0;
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        sum += v[i];
    return sum;
}
```

- Private copy initialized here to the identity value
- Shared variable updated here with the partial values of each thread
The *schedule* clause determines which iterations are executed by each thread.

- Important to choose for performance reasons only

There are several possible options as schedule:

- **STATIC**
  - Good locality, low overhead, load imbalance
- **STATIC, chunk**
- **DYNAMIC [, chunk]**
- **GUIDED [, chunk]**
- **AUTO**
- **RUNTIME**

Bad locality, higher overhead, load balance
The single construct

Example

```c
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Hello world!\n");
        }
    }
}
```

This program outputs just one "Hello world"
The single construct

Example

```c
int main ( int argc, char **argv )
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Hello world!\n");
        }
    }
}
```

This program outputs just one “Hello world”
Outline

1. Introduction
2. Writing OpenMP programs
3. Data-sharing attributes
4. Synchronization
5. Worksharings
6. Task parallelism
Parallelism is extracted from “several” pieces of code
Allows to parallelize very unstructured parallelism
  Unbounded loops, recursive functions, ...
What is a task in OpenMP?

Tasks are work units whose execution *may* be deferred
- they can also be executed immediately

Tasks are composed of:
- **code** to execute
- a **data** environment
  - Initialized at creation time
- internal control variables (**ICVs**)

Threads of the team **cooperate** to execute them
When are task created?

- **Parallel** regions create tasks
  - One *implicit* task is created and assigned to each thread
    - So all task-concepts have sense inside the parallel region
  - Each thread that encounters a *task* construct
    - Packages the code and data
    - Creates a new *explicit* task
List traversal

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first ; e ; e = e->next )
    {
        #pragma omp task
        process(e);
        e is firstprivate
    }
}
```
Task parallelism

Taskwait

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first ; e ; e = e->next )
        #pragma omp task
        process(e);

    #pragma omp taskwait
}
```

Suspends current task until all children are completed
Task parallelism

Taskwait

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l->first; e; e = e->next )
        #pragma omp task
            process(e);

    #pragma omp taskwait
}
```

All tasks guaranteed to be completed here
Task parallelism

Taskwait

Example

```c
void traverse_list ( List l )
{
    Element e;
    for ( e = l→first; e; e = e→next )
        #pragma omp task
        process(e);
    #pragma omp taskwait
}
```

Now we need some threads to execute the tasks
List traversal
Completing the picture

Example

List l

#pragma omp parallel
traverse_list(l);
List traversal
Completing the picture

Example

List l

```c
#pragma omp parallel
traverse_list(l);
```

This will generate multiple traversals
List traversal
Completing the picture

Example

List l

```
#pragma omp parallel
traverse_list(l);
```

We need a way to have a single thread execute **traverse_list**
List traversal
Completing the picture

Example

List 1

```c
#pragma omp parallel
#pragma omp single
traverse_list(l);
```
List traversal
Completing the picture

Example

```c
#pragma omp parallel
#pragma omp single
traverse_list(l);
```

One thread creates the tasks of the traversal
List traversal

Completing the picture

Example

```c
List l

#pragma omp parallel
#pragma omp single
traverse_list(l);
```

All threads cooperate to execute them
Task parallelism

Another example

Search problem

Example

```c
void search (int n, int j, bool *state)
{
    int i, res;

    if (n == j) {
        /* good solution, count it */
        mysolutions ++;
        return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i ++)
#pragma omp task
    {
        bool *new_state = alloc(sizeof(bool)*n);
        memcpy(new_state, state, sizeof(bool)*n);
        new_state[j] = i;
        if (ok(j+1, new_state)) {
            search(n, j+1, new_state);
        }
    }

#pragma omp taskwait
}
```
Task parallelism

Summary

OpenMP...

- allows to incrementally parallelize applications for SMP
- has good support for data and task parallelism
- requires you to pay attention to locality
- has many other features beyond this short presentation
  - http://www.openmp.org
The End

Thanks for your attention!