A brief introduction to OpenMP

Alejandro Duran

Barcelona Supercomputing Center

Outline

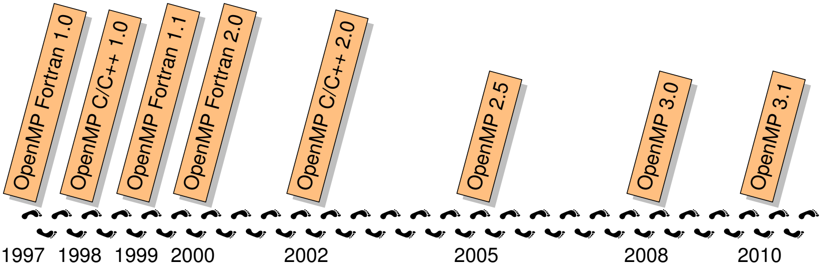
Outline

What is OpenMP?

- 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

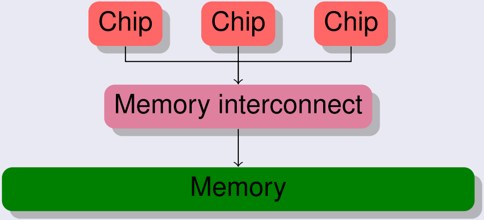
http://www.openmp.org

A bit of history

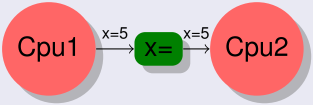


Target machines

Shared Multiprocessors



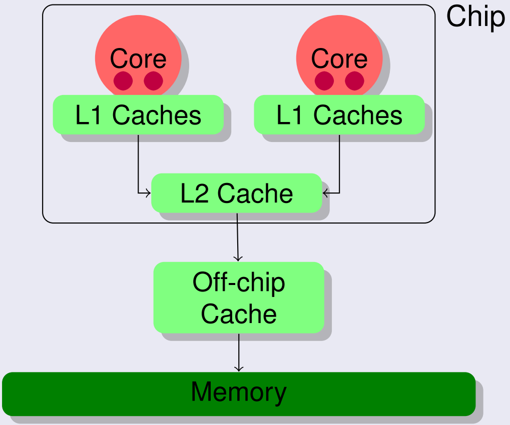
Shared memory



- Memory is shared across different processors
- Communication and synchronization happen implicitely through shared memory

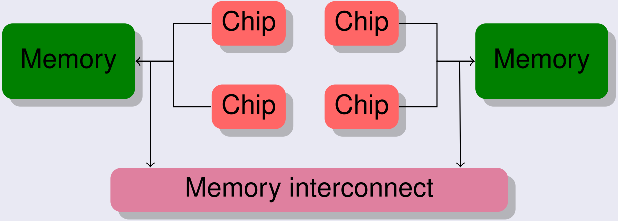
Including...

Multicores/SMTs



More commonly

NUMA



- 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

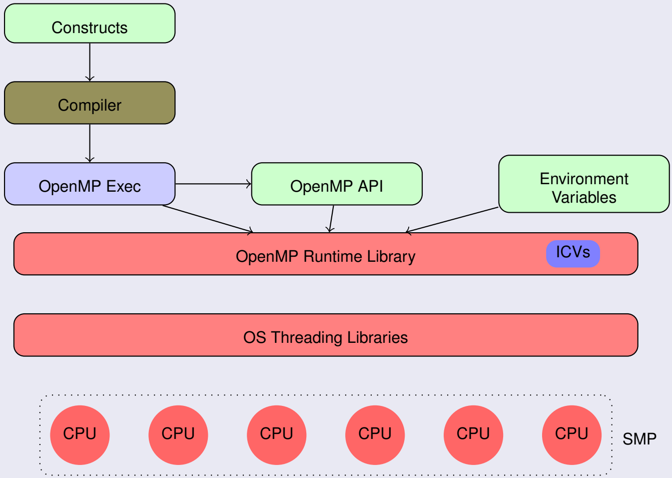
Why not OpenMP?

- 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

OpenMP at a glance

OpenMP components



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

```
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

```
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

Hello world!

Example

```
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

Hello world!

Example

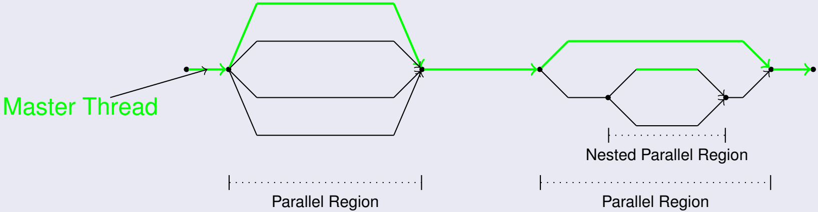
```
int id;
char *message = "Hello world!";

#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
    - syncronization constructs, parallelism creation points, ...
  - Luckily, the default points are usually enough
- Variables can have shared or private visibility for each thread

Outline

Data environment

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

```
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);
}
```

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) fi
                     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

```
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

```
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 ← (printf("%d\n",y);)

Data-sharing attributes

Example

```
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
}
```

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

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

  ...

  return buffer;
}
```

Creates one *static* copy of *buffer* per thread

Threaprivate storage

Example

```
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

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

  ...

  return buffer;
}
```

Simpler than redefining the interface. More costly

Outline

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

Barrier

Example

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

Syncronizes all threads of the team

Barrier

Example

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

Forces all foo occurrences too happen before all bar occurrences

Critical construct

Example

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

Critical construct

Example

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

Only one thread at a time here

Critical construct

Example

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

Only one thread at a time here

Prints 3!

Atomic construct

Example

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

Atomic construct

Example

```
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

```
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

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

Prints 3!

Locks

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

Outline

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

```
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

```
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

```
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

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

The *i* variable is automatically privatized

The for construct

Example

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

Must be explicitly privatized

The reduction clause

Example

```
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

```
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

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma
      for (
         sum
   return su
}
```

Private copy initialized here to the identity value

Shared variable updated here with the partial values of each thread

The schedule clause

The **schedule** clause determines which iterations are executed by each thread.

- Importart to choose for performance reasons only

There are several possible options as schedule:

- **STATIC** ← Good locality, low overhead, load imbalance
- **STATIC,chunk** ← (Good locality, low overhead, load imbalance)
- **DYNAMIC[,chunk]** ← Bad locality, higher overhead, load balance
- **GUIDED[,chunk]** ← (Bad locality, higher overhead, load balance)
- **AUTO**
- **RUNTIME**

The single construct

Example

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

The single construct

Example

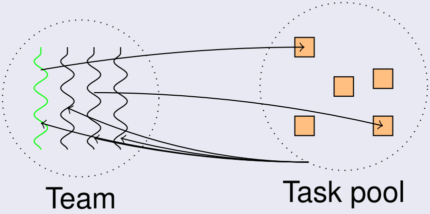
```
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
       #pragma omp single
       {
           printf("Hello world!\n");
       }
    }
}
```

This program outputs just one "Hello world"

Outline

Task parallelism in OpenMP

Task parallelism model



- 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

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

e is **firstprivate**

Taskwait

Example

```
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

Taskwait

Example

```
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

Taskwait

Example

```
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

#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 l

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

List traversal

Completing the picture

Example

```
List l

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

← One thread creates the tasks of the traversal

List traversal

Completing the picture

Example

```
List l

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

All threads cooperate to execute them

Another example

Search problem

Example

```
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 = alloca(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
}
```

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!