9th Advanced School on PARALLEL COMPUTING

# Evolution of OpenMP

**Marco Rorro**– m.rorro@cineca.it
SuperComputing Applications and Innovation Department

CINECA

February 11 - 15, 2013

## Outline

- Task parallelism in OpenMP
  - task construct
  - data scoping
  - synchronization
  - task switching
  - optimizations
- The future of OpenMP
  - atomic
  - SIMD
  - affinity
  - user defined reductions
  - support for accelerators

## History

- Born to satisfy the need of unification of proprietary solutions
- The ancient past
  - October 1997 – Fortran version 1.0
  - October 1998 – C/C++ version 1.0
  - November 1999 – Fortran version 1.1 (interpretations)
  - November 2000 – Fortran version 2.0
  - March 2002 – C/C++ version 2.0
  - May 2005 – combined C/C++ and Fortran version 2.5
- The recent past (currently supported by most of compilers)
  - May 2008 – version 3.0 (task!)
  - July 2011 – version 3.1
- The present and next future
  - November 2012 – TR1 on Directives for Attached Accelerators
  - November 2012 – version 4.0 public RC1

## Pointer chasing in OpenMP 2.5

```
#pragma omp parallel private(p)
   p=head;
   while ( p ) {
      #pragma omp single nowait
         process(p->item);
      p = p->next;
   }
```

- Transformation to a “canonical” loop can be very labour-intensive/inefficient
- The main drawback of the **`single nowait`** solution is that it is not composable
- Recall that worksharing construct can not be nested

## Tree traversal in OpenMP 2.5

```
void preorder(node *p) {
    process(p->data);
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
            if (p->left)
                preorder(p->left);
        #pragma omp section
            if (p->right)
                preorder(p->right);
    }
}
```

- You need to set OMP_NESTED to true
- But stressing nested parallelism so much is not a good idea… extra overheads, extra synchronizations …

## Task parallelism

- Better solution for those problems
- Main addition to OpenMP 3.0 enhanced in 3.1 and 4.0
- Allow to parallelize irregular problems
    - Unbounded loop
    - Recursive algorithms
    - Producer/consumer schemes
    - Multiblock grids, Adaptive Mesh Refinement
    - …

## First and foremost tasking construct

```
#pragma omp parallel [clauses]
{
    // structured block
}
```

- Creates both threads and tasks
- These tasks are “implicit”
- Each one is immediately executed by one thread
- Each of them is tied to the assigned thread

## New tasking construct

```
#pragma omp task [clauses]
{
    // structured block
}
```

where *clauses* is:

```
private, firstprivate, shared, default, if, untied
final, mergeable
```

- Immediately creates a new task but no a new thread
- This task is “explicit”
- It will be executed by a thread in the current team
- It can be deferred until a thread is available to execute
- The data environment is built at creation time

## Pointer chasing using task

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

p is firstprivate inside this task

- One thread creates task
  - packages code and data environment
- When it finishes, it reaches the implicit barrier and starts to execute the task
- The other threads reach straight the implicit barrier and start to execute task

## Data scoping in explicit tasks

- **`private`** and **`firstprivate`**: business as usual

- **`shared`**: same business, from a new perspective
  - shared among all tasks (“horizontal”)
  - shared among a task and a descendant (“vertical”)
  - different synchronizations are required in the two cases

- Most implicitly determined variables: firstprivate!
  - safety by default, programmers have full control
  - spares programmers a lot of keystrokes
  - can be altered with a **`default`** clause

## Task data scoping example

```
#pragma omp parallel shared(a) private(b)
{
    …
    #pragma omp task
        int c;
        process(a,b,c);
    }
}
```

a is shared
b is firstprivate
c is private

- 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 inner most enclosing parallel construct are shared, because the barrier guarantees task completion

## Load balancing on lists with tasks

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for (i=0; i<num_lists; i++) {
      p = heads[i];
      while(p) {
      #pragma omp task
         process(p);
      p=p->next;
      }
   }
}
```

- Assign one list per thread could be unbalanced
- Multiple threads create tasks
- All the team cooperates executing them

## Tree traversal with task

```
void preorder(node *p) {
   process(p->data);
   if (p->left)
   #pragma omp task
      preorder(p->left);
   if (p->right)
   #pragma omp task
      preorder(p->right);
}
```

- Tasks are composable
- It isn’t a worksharing construct
- But what about postorder traversal?

## Postorder tree traversal

```
void postorder(node *p) {
  if (p->left)
    #pragma omp task
      postorder(p->left);
  if (p->right)
    #pragma omp task
      postorder(p->right);
  #pragma omp taskwait

  process(p->data);
}
```

suspend point

- Parent task suspended until children tasks complete

## “Vertical” sharing

```
int fibonacci(int n) {
   int i, j;
   if (n<2) return n;

   #pragma omp task shared(i)
      i = fibonacci(n-1);
   #pragma omp task shared(j)
      j = fibonacci(n-2);
   #pragma omp taskwait

   return i+j;
}
```

synchronization

- Allow results to be returned to parent

## When/where explicit tasks complete?

- `#pragma omp taskwait`
  - applies only to tasks generated in the current task, not to descendants
- `#pragma omp taskgroup`
  ```
  {
   create_a_group_of_task(could_create_nested_task)
  }
  ```
  - at the end of the region current task is suspended until all child tasks generated in the region and their descendants complete execution (version 4.0RC1)
- `#pragma omp barrier`
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
  - obviously applies to implicit barriers too

## Enter *task switching*

- What: the ability of a thread to suspend a task and execute another one before resuming
- Where:
  - at *task scheduling points*: **`task`**, **`taskwait`**, **`barrier`** directives, and implicit barriers
  - at a **`taskyield`** construct
- When:
  - whenever is needed or useful
  - up to the implementation
- Why:
  - to lift pressure on runtime data structures
  - because it can’t be dispensed with completely
- Consequence: locks owned by tasks!

## Lifting pressure on runtime

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - dive into the encountered task (could be very cache-friendly)

## Enter *thread switching*

```
#pragma omp single
{
  #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…
- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too unsafe to be the default, the programmer is responsible!

## The `if` clause

- When the **`if`** clause argument is false
  - the encountered task is executed immediately by the encountering thread, and the enclosing task is suspended up to its end
  - the data environment is still local to the new task
  - and it's still a different task wrt. synchronization

- It's a user directed optimization
  - when the cost of the task is comparable to the runtime overhead
  - to control cache and memory affinity

## The `final` clause

- When the **`final`** clause argument is true
  - the generated task will be a final task
  - all task encountered during execution of a final task will generate included task
    - an included task is a task for which execution is sequentially included in the generating task region; that is, it is undeferred and executed immediately by the encountering threads
- It’s another user directed optimization

- **`omp_in_final()`** returns true if the enclosing task region is final. Otherwise, it returns false

## Example: if and final

```
#pragma omp task if(0) // This is undeferred
{
  #pragma omp task // This is a regular task
     for( i = 0; i < 3; i++ ) {
       #pragma omp task // This is a regular task
         bar();
     }
}
#pragma omp task final(1) // This is a regular task
{
  #pragma omp task // This is included
    for(i=0;i<3;i++){
      #pragma omp task // This is also included
        bar();
    }
}
```

## The `mergeable` clause

- When the **`mergeable`** clause is present on a task construct, and the generated task is undeferred or included, the implementation may generate a merged task instead
  - a merged task is a task whose data environments, inclusive of ICVs, is the same as that of its generating task region

- It is still another optimization …

  - **`pragma omp task final(d > LIMIT) mergeable`**

## Conclusions on task

- Tasks allow to express a lot of irregular parallelism
- The tasking concept opens up opportunities to parallelize a wider range of applications
- Some issues can be improved in the language
  - reductions, data capturing, dependencies, …
- Some extension are already planned for RC2
  - point to point dependencies
  - …

## `atomic` enhancements

- The **`atomic`** construct was extended (in 3.1) to include **`read`**, **`write`**, and **`capture`** forms, and an **`update`** clause was added to apply the already existing form of the **`atomic`** construct.
- The allowed instruction differ between Fortran and C/C++
  - refer to the standard for their list
- Added **`seq_cst`** clause for atomics in 4.0; removes need for flush ...
  - **`seq_cst`** stand for sequentially consistent
- Any **`atomic`** construct with a **`seq_cst`** clause forces the atomically performed operation to include an implicit flush operation without a list.

## atomic examples

```
#pragma omp atomic update
   x += n*mass; // x is updated atomically
#pragma omp atomic read
   v = x; // x is read atomically
#pragma omp atomic write
   x = n*mass; // x is written atomically
#pragma omp atomic capture
   v = x++; //atomically update x,
            //but capture original value in v
#pragma omp atomic capture
   v = ++x; //atomically update x,
            //then capture that value
```

## Transactional Memory in 4.0?

- Transactional memory attempts to simplify parallel programming by grouping read and write operations and running them like a single operation.
- Hardware techniques track temporary results and then commit them if no conflict has occurred
- The existing hardware cache coherency mechanism is used to track read and write sets, usually tagged with a special transactional bit.
- This mechanism also effectively limits the size of transactions to a cache size

- The BG/Q compute chip includes a versioning L2 cache that can associate version numbers with cache tags. Thus, the cache can contain multiple instances of the same address. This mechanism allows the BG/Q compiler and runtime to implement TM support.

## Transactional Memory on BG/Q

- Transactional memory is enabled with the **"-qtm"** compiler option, and requires thread safe compilation mode (mpixlc**_r**)
- Transactions are implemented through regions of code that you can designate to be single operations for the system. The regions of code that implement the transactions are called transactional atomic regions:

```
#pragma tm_atomic [safe_mode] { structured block }
```

- The **safe_mode** clause asserts to the compiler that the TM region does not contain irrevocable actions such as I/O or writing to device memory space. Using the **safe_mode** clause reduces overhead and increases performance.
- However, if **safe_mode** is specified, irrevocable actions are not checked at runtime. The run result is undefined if an irrevocable action occurs during the execution.

## TM on BG/Q: execution mode

- **Speculation mode** (default)
  - Under speculation mode, Kernel address space, devices I/Os, and most memory-mapped I/Os are protected from the irrevocable actions except when the **safe_mode** clause is specified. The transaction goes into irrevocable mode if such an action occurs to guarantee the correct result.
- **Irrevocable mode**
  - System calls, irrevocable operations such as I/O operations, and OpenMP constructs trigger transactions to go into irrevocable mode, which serializes transactions. Transactions are also running in irrevocable mode when the maximum number of transaction rollbacks has been reached.

  - Under irrevocable mode, each memory update of a thread is committed instantaneously instead of at the end of the transaction. Therefore, memory updates are immediately visible to other threads. If the transaction becomes irrevocable, the threads run nonspeculatively.

9th Advanced School on PARALLEL COMPUTING
February 11 - 15, 2013 - BOLOGNA

## TM on BG/Q: example

```
#pragma omp parallel for private(block,fbeg,fend,face,ii,jj)
schedule(static)
   for (block = 0; block < numblocks; block++) {
      if (block < leftover) {
         fbeg = block*(blocksize + 1);
         fend = fbeg + blocksize + 1;
      }
      else {
         fbeg = leftover + block*blocksize;
         fend = fbeg + blocksize;
      }
   #pragma tm_atomic
   {
         for (face=fbeg; face<fend; face++) {
            ii = ii_list[face];
            jj = jj_list[face];
            y[ii] += A[face] * x[jj];
            y[jj] += A[face] * x[ii];
         }
   }
   }
```

## SIMD constructs in OpenMP 4.0

```
#pragma omp simd [clauses]
    for loops
```

where *clauses* is:

```
safelen, linear, aligned, private, lastprivate,
reduction, collapse
```

- Executes iterations of the associated loops in SIMD chunks
  - SIMD chunk is set of iterations executed concurrently by a SIMD lanes
- Clauses control data environment, how loop is partitioned
  - **`safelen(`**`length`**`)`** limits the number of iterations in a SIMD chunk
  - **`linear`** lists variables with a linear relationship to the iteration space
  - **`aligned`** specifies byte alignments of a list of variables
  - **`private`**, **`lastprivate`**, **`reduction`** and **`collapse`** have usual meanings

## SIMD functions in OpenMP 4.0

```
#pragma omp declare simd [clauses]
   function definition or declaration
```

where *clauses* is:

```
safelen, linear, aligned, uniform, private,
reduction, inbranch, notinbranch
```

- Enable the creation of a function that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.
- Clauses control data environment, how function is used
  - **simdlen(**length**)** specifies the number of concurrent arguments
  - **uniform** lists invariant arguments across concurrent SIMD invocations
  - **inbranch** and **notinbranch** imply always/never invoked in a conditional statement
  - **linear**, **aligned**, **reduction** are similar to **simd** clauses

## loop and parallel loop SIMD

```
#pragma omp for simd [clauses]
    for loops
```

- Loop is first divided into SIMD chunks
- SIMD chunks are divided across implicit tasks
- clause can be any of the clauses accepted by the **`for`** or **`simd`** directives with identical meanings
- **`parallel`** and **`for simd`** construct can be combined:

```
#pragma omp parallel for simd [clauses]
    for loops
```

## Affinity support

- Request binding of threads to places (in OpenMP 3.1)
  ```
  export OMP_PROC_BIND=true
  ```
- New extensions (4.0) specify thread locations
  - Increased choices for **`OMP_PROC_BIND`**
  - Can still specify **`true`** or **`false`**
  - Can now provide a list (possible item values: **`master`**, **`close`** or **`spread`**) to specify how to bind implicit tasks of parallel regions
  - Assign threads to same place as **`master`**
  - Assign threads **`close`** in place list to parent thread
  - Assign threads to maximize **`spread`** across places
- Added **`OMP_PLACES`** environment variable
  - Can specify abstract names including **`threads`**, **`cores`** and **`sockets`**
  - Can specify an explicit ordered list of places
- Added a new clause to the parallel construct
  ```
  proc_bind(master | close | spread)
  ```

## User Defined Reductions (UDRs)

```
#pragma omp declare reduction (reduction-identifier :
typename-list : combiner ) [identity(identy-expr)]
```

- `reduction-identifier` gives a name to the operator
- `typename-list` is a list of types to which it applies
- `combiner` expression specifies how to combine values
- `identity` can specify the identity value operator

## UDR example

```
using namespace std;

#pragma omp declare reduction (merge : vector<int> :
omp_out.insert(omp_out.end(), omp_in.begin(),
omp_in.end()))

void schedule (vector<int> &v, vector<int> &w)
{
 #pragma omp parallel for reduction (merge : w)
 for (vector<int>::iterator it = v.begin();
      it < v.end(); it++)
    if ( filter(*it) ) w.push_back(*it);
}
```

- `omp_out` refers to private copy that holds combined value
- `omp_in` refers to the value to be combined

## OpenMP in a heterogeneous world

- Hardware
  - GPUs (NVIDIA and AMD)
  - Intel Xeon PHI
  - FPGA
  - DSP
- Application Programming Interfaces
  - CUDA, CUDA Fortran (PGI)
  - OpenCL
  - OpenACC
  - HMPP
- The OpenACC API is based on directives and its experience will be the base approach for integration in OpenMP

## OpenMP on accelerators

- New directives
  - target
  - target data
  - target update – target mirror – target linkable
- New runtime functions
  - omp_get_device_num
  - omp_set_device_num
- New environment variable
  - OMP_DEVICE_NUM

- It’s only a Technical Report

## targed

```
#pragma omp target [clauses]
 parallel-loop-construct|parallel-sections-construct
```

where *clauses* is:

```
device, map, mapto, mapfrom, scratch, num_threads,
if
```

- Create a device data environment and execute the construct on the same device.
- Example:

```
sum = 0;
#pragma omp target device(acc0) map(B,C)
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
    sum += B[i] + C[i];
```

## targed update

```
#pragma omp target update [clauses]
```

where *clauses* is:

```
device, mapto, mapfrom, if
```

- Update(to) a variable from the data environment of the current task to the enclosing device data environment, or update(from) a variable from the enclosing device data environment to the data environment of the current task.
- Example:

```
#pragma omp target update mapfrom(A,B)
#pragma omp target update mapto(C,D)
```

## declare targed

```
#pragma omp declare target
function-definition or declaration
```

- The **declare target** construct can be applied to a function to enable the creation of a device specific version that can be called from a target region.

```
#pragma omp declare target mirror (list)
```

- Map a global variable to a device for the duration of the program

```
#pragma omp declare target linkable(list)
```

- Assert that the user has mapped a global variable to a device

## Conclusions and credits

- OpenMP is evolving fast
  - support for accelerators
  - error handling
  - thread affinity
  - tasking extension
  - support for Fortran 2003
- Stay tuned: www.openmp.org
- Credits
  - Federico Massaioli
  - Ruud van  der Pas
  - Alejandro Duran
  - Bronis R. de Supinski
  - Tim Mattson and Larry Meadows
  - James Beyer and Eric Stotzer