OpenMP 3.0: What's new?

Alejandro Duran

Barcelona Supercomputing Center

Outline

Outline

1. Introduction
2. 3.0: not only tasks
   - Improvements to loop parallelism
   - Improvements to nested parallelism
   - Odds and ends
3. 3.0 is tasks
   - Task parallelism
   - Task pitfalls
4. Conclusions

OpenMP 3.0

- Work of over two years by lots people
  - From industry
    - Intel, Sun, IBM, PGI, Cray, AMD, Fujitsu, SGI, HP
  - And academia
    - LLNL, CASPUR, EPCC, Aachen Univ., Housto Univ., UPC-BSC
- Appeared on May 2008
  - Already fully supported by Intel, IBM, SUN, PGI and GCC

3.0 in a nutshell

- New task parallelism
- Improvements to loop parallelism
  - Loop collapsing
  - New AUTO schedule
- Improvements to nested parallelism
  - Better control of resources
  - API to gather information
  - Better definition of Internal Control Variables
- Different language fixes and clarifications

Outline

Outline

1. Introduction
2. 3.0: not only tasks
   - Improvements to loop parallelism
   - Improvements to nested parallelism
   - Odds and ends
3. 3.0 is tasks
   - Task parallelism
   - Task pitfalls
4. Conclusions

STATIC schedule guarantees

Example

```
#pragma omp for nowait
   for ( i = 0; i < N; i++ )
      a[i] = ...

#pragma omp for
   for ( i = 0; i < N; i++ )
      c[i] = a[i] + ...
```

STATIC schedule guarantees

Example

```
#pragma omp for nowait
   for ( i = 0; i < N; i++ )
      a[i] = ...

#pragma omp for
   for ( i = 0; i < N; i++ )
      c[i] = a[i] + ...
```

Wrong in 2.5 & 3.0

Loops are not guaranteed to have the same schedule

STATIC schedule guarantees

Example

```
#pragma omp for nowait schedule(STATIC)
   for ( i = 0; i < N; i++ )
      a[i] = ...

#pragma omp for schedule(STATIC)
   for ( i = 0; i < N; i++ )
      c[i] = a[i] + ...
```

STATIC schedule guarantees

Example

```
#pragma omp for nowait schedule(STATIC)
   for ( i = 0; i < N; i++ )
      a[i] = ...

#pragma omp for schedule(STATIC)
   for ( i = 0; i < N; i++ )
      c[i] = a[i] + ...
```

Wrong in 2.5

STATIC is not guaranteed to be the same in both loops

STATIC schedule guarantees

Example

```
#pragma omp for nowait schedule(STATIC)
   for ( i = 0; i < N; i++ )
      a[i] = ...

#pragma omp for schedule(STATIC)
   for ( i = 0; i < N; i++ )
      c[i] = a[i] + ...
```

Correct in 3.0

If (and only if):

- number of iterations is the same
- chunk is the same (or no chunk)

Loop collapsing

Motivation

Example

```
for ( i = 0; i < N; i++ )
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

- Both i and j loops are fully parallel
  - Ideally we would just parallelize one of them

Loop collapsing

Motivation

Example

```
for ( i = 0; i < N; i++ )
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

- Both i and j loops are fully parallel
  - Ideally we would just parallelize one of them
- But if N and M are small compared to the number of processors

Loop collapsing

Motivation

Example

```
for ( i = 0; i < N; i++ )
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

- Both i and j loops are fully parallel
  - Ideally we would just parallelize one of them
- But if N and M are small compared to the number of processors
  - We need to get work from both loops!

Loop collapsing
Motivation

Example

```
#pragma omp parallel for
for ( i = 0; i < N; i++ )
#pragma omp parallel for
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

In OpenMP 2.5

- Use nested paralellism
  - Unneeded synchronizations
  - Overhead

Loop collapsing

Motivation

Example

```
#pragma omp parallel for
for ( i = 0; i < N; i++ )
#pragma omp parallel for
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

In OpenMP 2.5

- Use nested paralellism
  - Unneeded synchronizations
  - Overhead
- Manually transform the loops

Loop collapsing

Motivation

Example

```
#pragma omp parallel for collapse(2)
for ( i = 0; i < N; i++ )
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

In OpenMP 3.0

Loop collapse

- Iteration space from both loops will be collapsed into a single one

Loop collapsing
Rules

```
#pragma omp for/do ... collapse(N) ...
```

- The N loops are collapsed
  - Loops must be perfectly nested
  - Iteration spaces must be rectangular
- Iterations are distributed in serial order

Loop collapsing

Rules

```
#pragma omp for/do ... collapse(N) ...
```

- The N loops are collapsed
  - Loops must be perfectly nested
  - Iteration spaces must be rectangular
- Iterations are distributed in serial order

Wrong

```
#pragma for collapse(2)
for ( i = 0; i < N; i ++ ) {
   foo(i);
   for ( j = 0; j < M; j++)
      bar(i,j);
}
```

Loop collapsing

Rules

```
#pragma omp for/do ... collapse(N) ...
```

- The N loops are collapsed
  - Loops must be perfectly nested
  - Iteration spaces must be rectangular
- Iterations are distributed in serial order

Wrong

```
#pragma for collapse(2)
for ( i = 0; i < N; i ++ ) {
   for ( j = 0; j < i; j++)
      bar(i,j);
}
```

New induction variable types

Example

```
#pragma omp for
for ( unsigned int i = 0; i < N ; i++ )
   foo(i);

Vector<int> v;
#pragma omp for
for ( Vector<int>::iterator it = v.begin();
      it < v.end();
      it++)
   foo(it);
```

In OpenMP 2.5

Illegal types

New induction variable types

Example

```
#pragma omp for
for ( unsigned int i = 0; i < N ; i++ )
   foo(i);

Vector<int> v;
#pragma omp for
for ( Vector<int>::iterator it = v.begin();
      it < v.end();
      it++)
   foo(it);
```

In OpenMP 3.0

New types:

- Unsigned int
- Random access iterators

New induction variable types

Example

```
char a[N];
#pragma omp for
for ( char *p = a; p < (a+N); p++ )
   foo(p);
```

Bonus

C/C++ pointers are random access iterators :)

New induction variable types

Example

```
#pragma omp for
for ( it = v.begin(); it != v.end(); it++)
   foo(it);

#pragma omp for
for ( char *p = s; p != s2; p++ )
   foo(it);
```

Careful!

!= is not a relational operator

New features

- AUTO schedule
  - Assignment of iterations to threads decided by the implementation
    - at compile time and/or execution time
    - from STATIC to advanced feedback guided schedules
- schedule API
  - new Internal Control Variable
  - omp_set_schedule
  - omp_get_schedule

Outline

Nested control variables

- Control maximum number of active parallel regions
  - OMP_MAX_NESTED_LEVEL environment variable
  - omp_set_max_nested_levels()
  - omp_get_max_nested_levels()
- Control maximum number of OpenMP threads created
  - OMP_THREAD_LIMIT enviroment variable
  - omp_get_thread_limit()

Nested API

To obtain information about nested parallelism

- How many nested parallel regions at this point?
  - omp_get_level()
- How many active (with 2 or more threads) regions?
  - omp_get_active_level()
- Which thread-id was my ancestor?
  - omp_get_ancestor_thread_num(level)
- How many threads there are at a previous regions?
  - omp_get_team_size(level)

Multiple ICVs

Controlling parallel regions size

Example

How many threads are created here?

```
#pragma omp parallel num_threads(3)
{
  omp_set_num_threads(omp_get_thread_num()+2);
  #pragma omp parallel
      foo();
}
```

Multiple ICVs

Controlling parallel regions size

Example

How many threads are created here?

```
#pragma omp parallel num_threads(3)
{
  omp_set_num_threads(omp_get_thread_num()+2);
  #pragma omp parallel
      foo();
}
```

In OpenMP 2.5

Unknown behavior

Multiple ICVs

- The standard defines multiple copies of ICVs
  - One for each parallel region
    - Actually, for each task (coming in shortly)
- Each region can have its own behavior
  - Values inherited from parent region
  - Changes affect new child regions
    - Not the current one
- Some ICVs still have a single global value
  - Check the documentation

Multiple ICVs

Controlling parallel regions size

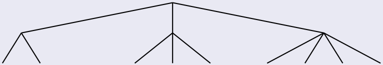
Example

How many threads are created here?

```
#pragma omp parallel num_threads(3)
{
  omp_set_num_threads(omp_get_thread_num()+2);
  #pragma omp parallel
      foo();
}
```

In OpenMP 3.0

Well defined:



Multiple ICVs

Others as well

Example (run-sched-var ICV)

```
omp_sched_t schedules[] =
  { omp_sched_static, omp_sched_dynamic, omp_sched_auto };
#pragma omp parallel num_threads(3)
{
   omp_set_schedule(schedules[omp_get_thread_num()],0);

   #pragma omp parallel for
      for ( i = 0 ; i < N; i++ ) foo(i);
}
```

Outline

Odds and ends

New control variables

- Control of children thread's stack size
  - OMP_SET_STACKSIZE environment variable
- Control of threads idle behavior
  - OMP_WAIT_POLICY environment variable (hint)
    - active: use CPU while waiting
      - good for dedicated systems
    - passive: avoid use of CPU while waiting
      - good for shared systems

Odds and ends

Language fixes

- Clearer rules for how private objects are constructed/destructed
- C++ static data members can be threadprivate
- Fortran allocatables can appear in private-related clauses (private, firstprivate, ...)

Outline

Outline

Why task parallelism?

Example

```
void traverse_list ( List l )
{
Element e;

#pragma omp parallel private(e)
    for ( e = l->first; e ; e = e->next )
        #pragma omp single nowait
            process(e);
}
```

OpenMP 2.5 style

- Ackward
- Very poor performance
- Not composable

Why task parallelism?

Example

```
void traverse (Tree *tree)
{
#pragma omp parallel sections
{
#pragma omp section
     if ( tree->left )
       traverse(tree->left);
#pragma omp section
     if ( tree->right )
       traverse(tree->right);
}

process(tree);
}
```

OpenMP 2.5 style

- Too many parallel regions
  - Extra overheads
  - Extra synchronizations
  - Not always well supported

Task parallelism

- Better solution for those problems
- Main addition to OpenMP 3.0
  - Design decision: tightly integration[a]
- Allows to parallelize irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer schemes
  - ...

[a] Ayguadé et al., The Design of OpenMP Tasks, IEEE TPDS March 2009

What is an OpenMP task?

- Tasks are work units which execution may be deferred
  - they can also be executed immediately
- Tasks are composed of:
  - code to execute
  - data environment
    - Initialized at creation time
  - internal control variables (ICVs)

Task directive

```
#pragma omp task [clauses]
    structured block
```

- Each encountering thread creates a task
  - Packages code and data environment
- Can be nested
  - inside other tasks
  - inisde worksharings

Why not a worksharing directive?

Alternative (as omp sections)

```
#pragma omp tasks
+
#pragma omp task
```

Disadvantages

- Tasks are bound to a region
- Cannot be nested

List traversal

Example

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

List traversal

Example

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

What is the scope of e?

Task data scoping

Data scoping clauses

- shared(list)
- private(list)
- firstprivate(list)
  - data is captured at creation
- default(shared|none)

Task data scoping

Data scoping clauses

- shared(list)
- private(list)
- firstprivate(list)
  - data is captured at creation
- default(shared|none)

If no clause

- Implicit rules apply
  - e.g., global variables are shared
- and then? What is the default?

Task data scoping

Design issues

Options

shared

- Consisted with the rest
- Experienced users may expect it
- Most of the time not what you want
- Increases the out-of-scope problem

firstprivate

- Not consistent with the rest
- Most of the time is what you want

Task data scoping

Design issues

Options

shared
- Consisted with the rest
- Experienced users may expect it
- Most of the time not what you want
- Increases the out-of-scope problem

firstprivate
- Not consistent with the rest
- Most of the time is what you want

At the end...

Mix of both:

- firstprivate by default
- But, shared attributed is lexically inherited

Task data scoping

In practice...

Example

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

Task data scoping

In practice...

Example

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

Task data scoping

In practice...

Example

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

Task data scoping

In practice...

Example

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

Task data scoping

In practice...

Example

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

Tip

Use default(none) if you do not see it clear

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

List traversal

Example

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

}
```

how we can guarantee here that the traversal is finished?

Task synchronization

- Barriers (implicit or explicit)
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit
- Task barrier

```
#pragma omp taskwait
```

  - Encountering task suspends until child tasks complete
    - Only direct childs not descendants!

List traversal

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

Task execution model

- Task are executed by a thread of the team that generated it
  - Can be executed immediately by the same thread that creates it
- Parallel regions in 3.0 create tasks!
  - One implicit task is created for each thread
  - This is important so all task-concepts have sense inside the parallel region
- Threads can suspend the execution of a task and start/resume another

List traversal

Example

```
List l

#pragma omp parallel
traverse_list(l)
```

List traversal

Example

```
List l

#pragma omp parallel
traverse_list(l)
```

Careful!

Multiple traversals of the same list

List traversal

Single traversal

Example

```
List l

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

Single traversal

- One thread enters single and creates all tasks
- All the team cooperates executing them

List traversal

Multiple traversals

Example

```
List l[N]

#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++)
   traverse_list(l[i])
```

Multiple traversals

- Multiple threads create tasks
- All the team cooperates executing them

Task scheduling

Design issues

Goal

Give flexibility to implementations for scheduling

- to solve imbalance issues
- increase locality
- ...

Task scheduling
Design issues

Goal

Give flexibility to implementations for scheduling

- to solve imbalance issues
- increase locality
- ...

The problem

Scheduling indeterminism does not mix well with thread-based applications

- threadprivate variables
- use of *thread-id*
- locks and criticals

Task scheduling
Design issues

How it works?

- Tasks are tied by default
  - Tied tasks are executed always by the same thread
  - Tied tasks have scheduling restrictions
    - Determinist scheduling points (creation, synchronization, ...)
    - Another constraint to avoid deadlock problems
  - Tied tasks may run into performance problems
- Programmer can use untied clause to lift all restrictions
  - This gives the implementation scheduler much more freedom to optimize
  - Note: Do not expect much from untied at this points
  - Note: Mix very carefully with threadprivate, critical and thread-ids

The IF clause

- If the the expression of a if clause evaluates to false
  - The encountering task is susended
  - The new task is executed immediately
    - own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - allows the implementation to optimize task creation

IF example

Example

```
void branch ( int level , int m ) {
  int i ;
  if ( solution () ) return ;
  for ( i = 0; i < m; i++ )
    if ( !prune () )
      #pragma omp task untied if (level < LIMIT_LEVEL)
          branch (level+1 ,m) ;
}
```

Prune generates imbalance. Untied can help.

level and m are firstprivate

Limit task creation after a certain level

Outline

1. Introduction
2. 3.0: not only tasks
   - Improvements to loop parallelism
   - Improvements to nested parallelism
   - Odds and ends
3. 3.0 is tasks
   - Task parallelism
   - Task pitfalls
4. Conclusions

Task pitfalls

Out-of-scope data

The problem

1. Shared variables may reside in the parent scope (stack)
2. The parent task can finish (or exit the scope) before the child executes
3. The child gets garbage or even seg faults
4. Avoided with extra synchronizations are needed or moving data to heap

Task pitfalls

Out-of-scope data

Example

```
void foo ()
{
   int a[LARGE_N];
   #pragma omp task shared(a)
   {
      bar(a); // a is not modified
   }

}
```

Task pitfalls

Out-of-scope data

Design decision

Th choices were:

- The user needs to take care of the problem
  - Error prone
- Compiler introduces ensures correctness
  - Safer
  - Easier for the user
  - Potentially reduces parallelism

Task pitfalls

Out-of-scope data

Design decision

Th choices were:

- The user needs to take care of the problem
  - Error prone
- Compiler introduces ensures correctness
  - Safer
  - Easier for the user
  - Potentially reduces parallelism

At the end...

- Because base language is not always possible to detect
  - Choice two could give a false sense of security

Task pitfalls

Out-of-scope data

Design decision

Th choices were:

- The user needs to take care of the problem
  - Error prone
- Compiler introduces ensures correctness
  - Safer
  - Easier for the user
  - Potentially reduces parallelism

At the end...

- Because base language is not always possible to detect
  - Choice two could give a false sense of security
- So, the user needs to take care
  - Bottomline: be careful when using shared data in tasks

Task pitfalls

Out-of-scope data

Example

```
void foo ()
{
    int a[LARGE_N];
    #pragma omp task shared(a)
    {
        bar(a); // a is not modified
    }
    #pragma omp taskwait
}
```

Synchronization to ensure a is alive while task is executing

Moving a to the heap is also possible

Task pitfalls

Pointers

Example

```
void foo (int n, char *state)
{
  int i;
  modify_state(state);
  for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
      foo(n,state);
}
```

Each task needs its own state to progress

Task pitfalls

Pointers

Example

```
void foo (int n, char *state)
{
  int i;
  modify_state(state);
  for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
      foo(n,state);
}
```

Wrong!

Each task copies only the pointer!
All tasks modify the same state

Task pitfalls

Pointers

Example

```
void foo (int n, char *state)
{
  int i;
  modify_state(state);
  for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
      foo(n,state);
}
```

Problem

firstprivate does not allow to capture through pointers

Task pitfalls

Pointers

Example (Possible solution)

```
void foo (int n, char *state)
{
  int i;
  modify_state(state);
  for ( i = 0; i < n; i++ )
    #pragma omp task firstprivate(state)
    {
      char new_state[n];
      memcpy(new_state,state);
      foo(n,new_state);
    }
  #pragma omp taskwait

}
```

- Allocate manually a new state for the task
- Copy it from parent

Extra sync to ensure data is alive

Task pitfalls

Threadprivate

Example

```
int dummy;
#pragma omp threadprivate(dummy)

void bar() { dummy = ...; }
void foo () { ... = dummy; }

#pragma omp task untied
{
foo();
bar();
}
```

The task could switch to another thread here!

Task pitfalls

Tied generators

Example

```
#pragma omp parallel
#pragma omp single
for (kk=0; kk<NB; kk++) {
    lu0(A[kk][kk]);
    for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
#pragma omp task untied
            fwd(A[kk][kk], A[kk][jj]);

    ...
}
```

Problem

Certain task schedulers may have problems:

1. Decides to switch to one created task
2. The single portion is tied
   - No more tasks are generated

Task pitfalls

Tied generators

Example

```
#pragma omp parallel
#pragma omp single
#pragma omp task untied
for (kk=0; kk<NB; kk++) {
    lu0(A[kk][kk]);
    for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
#pragma omp task untied
            fwd(A[kk][kk], A[kk][jj]);
    ...
}
```

Solution

Introduce an *artificial* untied task

Outline

OpenMP 3.0

- Moves torwards task parallelism
  - Opens a new domain of applications
- Small (but useful) improvements in other areas

If you use it, we like to know what you think.
http://www.openmp.org

Beyond 3.0

Challenges for OpenMP

1. Modularity
   - Error model
   - Better composability
   - Interoperatibility
     - with other models (MPI, pthreads, ...)
     - across vendors
2. Break the flat world model
3. Task model improvements
   - New synchronizations types
   - Performance optimizations
4. ...

Beyond 3.0
Challenges for OpenMP

1. Modularity
   - Error model
   - Better composability
   - Interoperatibility
     - with other models (MPI, pthreads, ...)
     - across vendors
2. Break the flat world model
3. Task model improvements
   - New synchronizations types
   - Performance optimizations
4. ...

So, stay tuned! :-)

The End

Thanks for your attention!