Parallel Programming with OpenMP

Alejandro Duran

Barcelona Supercomputing Center

Agenda

-	Thursday
10:00 - 11:15	OpenMP Basics
11:00 - 11:30	Break
11:30 - 13:00	Hands-on (I)
13:00 - 14:30	Lunch
14:30 - 15:15	Task parallelism in OpenMP
15:15 - 17:00	Hands-on (II)
-	Friday
10:00 - 11:00	Data parallelism in OpenMP
11:00 - 11:30	Break
11:30 - 13:00	Hands-on (III)
13:00 - 14:30	Lunch
14:30 - 15:00	Other OpenMP topics
15:00 - 16:00	Hands-on (IV)
16:00 - 16:30	OpenMP in the future

Part I

Data Parallelism in OpenMP

Outline

- The worksharing concept
- Loop worksharing

Outline

- The worksharing concept
- Loop worksharing

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
- Lower overhead than using **`tasks`**
  - But, less flexible

In OpenMP, there are four worksharing constructs:

- single
- loop worksharing
- section
- workshare

We'll see them later

Restriction: worksharings cannot be nested

Outline

- The worksharing concept
- Loop worksharing

Loop parallelism

The for construct

```
#pragma omp for [clauses]
   for( init-expr ; test-expr ; inc-expr )
```

where clauses can be:

- private
- firstprivate
- **lastprivate(*variable-list*)**
- **reduction(*operator:variable-list*)**
- **schedule(*schedule-kind*)**
- **nowait**
- **collapse(*n*)**
- ordered ← We'll see it later

The for construct

How it works?

The iterations of the loop(s) associated to the construct are divided among the threads of the team.

- Loop iterations must be independent
- Loops must follow a form that allows to compute the number of iterations
- Valid data types for inductions variables are: integer types, pointers and random access iterators (in C++)
  - The induction variable(s) are automatically privatized
- The default data-sharing attribute is **`shared`**

It can be merged with the **`parallel`** construct:

```
#pragma omp parallel for
```

The for construct

Example

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

Must be explicitly privatized

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::iterator it = v.begin() ;
        it < v.end() ;
        it ++ )
    *it = 0;
}
```

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::iterator it = v.
        it < v.end() ;
        it ++ )
    *it = 0;
}
```

random access iterators (and pointers) are valid types

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::ite
        it < v.end() ;
        it ++ )
    *it = 0;
}
```

!= cannot be used in the test expression

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    v[i] = x;
    x += dx;
}
```

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    v[i] = x;
    x += dx;
}
```

Each iteration *x* depends on the previous one. Can't be parallelized

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    x = i * dx;
    v[i] = x;
}
```

But *x* can be rewritten in terms of *i*.
Now it can be parallelized

Removing dependences

Example

```
x = 0;
#pragma omp parallel for private(x)
for ( i = 0; i < n; i++ )
{
    x = i * dx;
    v[i] = x;
}
```

The lastprivate clause

When a variable is declared `lastprivate`, a private copy is generated for each thread. Then the value of the variable in the last iteration of the loop is copied back to the original variable.

- A variable can be both `firstprivate` and `lastprivate`

The lastprivate clause

Example

```
int i
#pragma omp for lastprivate(i)
for ( i = 0; i < 100; i++ )
   v[i] = 0;

printf("i=%d\n",i);
```

The lastprivate clause

Example

```
int i
#pragma omp for lastprivate(i)
for ( i = 0; i < 100; i++ )
   v[i] = 0;

printf("i=%d\n",i);
```

$\leftarrow$ prints 100

The reduction clause

A very common pattern is where all threads accumulate some values into a shared variable

- E.g., n += v[i], our pi program, ...
- Using **critical** or **atomic** is not good enough
  - Besides being error prone and cumbersome

Instead we can use the **reduction** clause for basic types.

- Valid operators for C/C++: +,-,*,|,||,&,&&,^
- Valid operators for Fortran: +,-,*,.and.,.or.,.eqv.,.neqv.,max,min
  - also supports reductions of arrays
- The compiler creates a **private** copy that is properly initialized
- At the end of the region, the compiler ensures that the **shared** variable is properly (and safely) updated.

We can also specify **reduction** variables in the **parallel** construct.

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;
}
```

The reduction clause

Example

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

Private copy initialized here to the identity value

Shared variable updated here with the partial values of each thread

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

reduction available in parallel as well

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

Prints the number of threads

The schedule clause

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

- If no **`schedule`** clause is present then is implementation defined

There are several possible options as schedule:

- **`STATIC`**
- **`STATIC,chunk`**
- **`DYNAMIC[,chunk]`**
- **`GUIDED[,chunk]`**
- **`AUTO`**
- **`RUNTIME`**

The schedule clause

Static schedule

The iteration space is broken in chunks of approximately size $N/num-threads$. Then these chunks are assigned to the threads in a Round-Robin fashion.

Static,N schedule (Interleaved)

The iteration space is broken in chunks of size N. Then these chunks are assigned to the threads in a Round-Robin fashion.

Characteristics of static schedules

- Low overhead
- Good locality (usually)
- Can have load imbalance problems

The schedule clause

Dynamic,N schedule

Threads dynamically grab chunks of N iterations until all iterations have been executed. If no chunk is specified, $N = 1$.

Guided,N schedule

Variant of **`dynamic`**. The size of the chunks deceases as the threads grab iterations, but it is at least of size N. If no chunk is specified, $N = 1$.

Characteristics of dynamic schedules

- Higher overhead
- Not very good locality (usually)
- Can solve imbalance problems

The schedule clause

Auto schedule

In this case, the implementation is allowed to do whatever it wishes.

- Do not expect much of it as of now

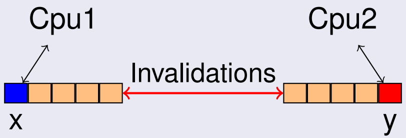
Runtime schedule

The decision is delayed until the program is run through the **`sched-nvar`** ICV. It can be set with:

- The **`OMP_SCHEDULE`** environment variable
- The **`omp_set_schedule`**() API call

False sharing

- When a thread writes to a cache location, and another thread reads the same location the coherence protocol will copy the data from one cache to the other. This is called true sharing
- But it can happen that this communication happens even if two threads are not working on the same memory address. This is false sharing



Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

i loop quite unbalaced

Scheduling

Example

```
int v[N];

#pragma omp for   ← dynamic schedule?
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
    for ( int j = 0; j < i
        v[i] += j;
```

lots of false sharing!

The nowait clause

When a worksharing has a **`nowait`** clause then the implicit **`barrier`** at the end of the loop is removed.

- This allows to overlap the execution of non-dependent loops/tasks/worksharings

The nowait clause

Example

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

First and second loop are independent so we can overlap them

The nowait clause

Example

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

On a side note, you would be better by fusing the loops in this case

The nowait clause

Example

```
#pragma omp for nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for
for ( i = 0; i < n ; i++ )
   a[i] = v[i]*v[i];
```

First and second loop are dependent!. No guarantees that the previous iteration is finished

The nowait clause

Exception: static schedules

If the two (or more) loops have the same **static** schedule and all have the same number of iterations.

Example

```
#pragma omp for schedule(static,2) nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for schedule(static,2)
for ( i = 0; i < n ; i++ )
   a[i] = v[i]*v[i];
```

The collapse clause

Allows to distribute work from a set of *n* nested loops.

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space

The collapse clause

Allows to distribute work from a set of *n* nested loops.

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space

Example

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

i and *j* loops are folded and iterations distributed among all threads. Both *i* and *j* are privatized

Coffee time! :-)

Part II

Hands-on (III)

Outline

Before you start

Copy the exercises to your directory:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_2/worksharing
.
```

Enter the worksharing directory to do the following exercises.

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Matrix Multiply

Parallel loops

The file matmul implements a sequential matrix multiply.

1. Use OpenMP worksharings to parallelize the application.
   - check the init_mat and matmul functions
2. Run it up to 8 threads to check the scalability

Remember: To submit it use make run-matmul.omp-$threads

Matrix Multiply

Memory matters!

To optimize accesses to the cache in these kind of algorithms, it is a common practice to “logically” split the matrix in blocks of size *BxB*, and do computation block-a-block instead of going through all the matrix at once.

1. Implement such a blocking scheme for our matrix multiply
2. Experiment with different sizes of *B*
3. Run it up to 8 threads and compare the results with the previous version

Tip: You need two additional inner loops

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Computing Pi

Using data parallelism

1. Complete the implementation of our pi algorithm using data parallelism
2. Execute with 1 and 2 threads.
    - Does it scale?
    - How does it compare to our previous implementation with tasks?
    - What is the problem?

Computing Pi

Problem

The number of synchronizations is still very high for this program to scale.

Using `reduction`

1. Change the program to make use of the `reduction` clause
2. Run it up to 8 threads
3. How it compares to the previous version?

Outline

Mandelbrot

More data parallelism

We will now parallelize an algorithm that generates sections of the Mandelbrot function.

1. Edit file mandel.c and complete the parallelization in function mandel
    - Note that there is a dependence on the variable x

Mandelbrot

Uncover load imbalance

We can see that each point in the final output is computed through the mandel_point function. If we check the code of that function we can see that the number of iterations it takes will be different from one point to another.

We want to know how many iterations (this also happens to be the result of mandel_point) each thread does.

1. Add a private counter to each thread
2. Add to this counter the result of each mandel_point call by that thread
3. Output the count for each thread at the end of the parallel region
4. What do you observe?

Mandelbrot

Playing with schedules

To overcome the observed load imbalance we can use a different loop schedule.

- Use the clause `schedule(runtime)` so the schedule is not fixed at compile time
- Now run different experiments with different schedules and number of threads
  - Try at least `static`, `dynamic` and `guided`
- Which one obtains the best result?

Tip: Change `OMP_SCHEDULE` before doing make run-...

Part III

Other OpenMP Topics

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Only the master thread

The master construct

```
#pragma omp master
    structured block
```

- The structured block is only executed by the master thread
  - Useful when we want always the same thread to execute something
- No implicit barrier at the end

Master construct

Example

```
void foo ()
{
   #pragma omp parallel
   {
      #pragma omp single
         printf("I␣am␣%d\n", omp_get_thread_num());

      #pragma omp master
         printf("I␣am␣%d\n", omp_get_thread_num());
   }
}
```

Master construct

Example

```
void foo ()
{
   #pragma omp parallel
   {
      #pragma omp single
         printf("I␣am␣%d\n", omp_get_thread_num());   ← Can be any thread

      #pragma omp master
         printf("I␣am␣%d\n", omp_get_thread_num());   ← It's always thread 0
   }
}
```

Outline

Ordering

The ordered construct

```
#pragma omp ordered
    structured block
```

- Must appear in the dynamic extend of a loop worksharing
    - The worksharing must also have the `ordered` clause
- The structured block is executed in the iteration's sequential order

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

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

OpenMP also provides nested locks where the thread owning the lock can reacquire the lock without blocking.

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
       omp_set_lock(&lock);
       // mutual exclusion region
       omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
       omp_set_lock(&lock);
       // mutual exclusion region
       omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Lock must be initialized before being used

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
       omp_set_lock(&lock);
       // mutual exclusion region
       omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Only one thread at a time here

Locks

Example

```
#include <omp.h>

omp_lock_t lock;

void foo ()
{
   omp_set_lock(&lock);
}

void bar ()
{
   omp_unset_lock(&lock);
}
```

Locks

Example

```
#include <omp.h>

omp_lock_t lock;

void foo ()
{
    omp_set_lock(&lock);
}

void bar ()
{
    omp_unset_lock(&lock);
}
```

Locks are unstructured

Outline

Nested parallelism

- OpenMP **`parallel`** constructs can dynamically be nested. This creates a hierarchy of teams that is called nested parallelism.
- Useful when not enough parallelism is available with a single level of parallelism
  - More difficult to understand and manage
  - Implementations are not required to support it

Controlling nested parallelism

Related Internal Control Variables

- The ICV `nest-var` controls whether nested parallelism is enabled or not.
  - Set with the `OMP_NESTED` environment variable
  - Set with the `omp_set_nested` API call
  - The current value can be retrieved with `omp_get_nested`.
- The ICV `max-active-levels-var` controls the maximum number of nested regions
  - Set with the `OMP_MAX_ACTIVE_LEVELS` environment variable
  - Set with the `omp_set_max_active_levels` API call
  - The current value can be retrieved with `omp_get_max_active_levels`.

Nested parallelism info 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 region?
  - `omp_get_team_size`(level)

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Static tasks

The sections construct

```
#pragma omp sections [clauses]
#pragma omp section
    structure block
    ...
```

- The different `section` are distributed among the threads
- There is an implicit barrier at the end
- Clauses can be:
    - `private`
    - `lastprivate`
    - `firstprivate`
    - `reduction`
    - `nowait`

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Sections

Example

```
#pragma omp parallel sections num_thr
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Combined construct

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Sections distributed among threads

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Nested parallel region

Supporting array syntax

The workshare construct

```
$!OMP WORKSHARE
    array syntax
!$OMP END WORKSHARE [NOWAIT]
```

- Only for Fortran
- The array operation is distributed among threads

Example

```
$!OMP WORKSHARE
   A(1:M) = A(1:M) * B(1:M)
!$OMP END WORKSHARE NOWAIT
```

Outline

Other Environment variables

`OMP_STACKSIZE`	Controls the stack size of created threads
`OMP_WAIT_POLICY`	Controls the behaviour of idle threads
`OMP_THREAD_LIMIT`	Limit of threads that can be created
`OMP_DYNAMIC`	Turns on/off thread dynamic adjusting

Other API calls

`omp_in_parallel`	Returns true if inside a parallel region
`omp_get_wtick`	Returns the precision of the **`wtime`** clock
`omp_get_thread_limit`	Returns the limit of threads
`omp_set_dynamic`	Returns whether thread dynamic adjusting is on or off
`omp_get_dynamic`	Returns the current value of dynamic adjusting
`omp_get_schedule`	Returns the current loop schedule

Part IV

Hands-on (IV)

Outline

Before you start

Copy the exercises to your directory:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_2/other .
```

Enter the other directory to do the following exercises.

Nested parallelism

First take

1. Edit the file nested.c and try to understand what it does
2. Run make
3. Execute the programe nested with differents numbers of threads
   - How many messages are printed? Does it match your expectations?
4. Run the program again the defining the OMP_NESTED variable. E.g.:

```
$ OMP_NUM_THREADS=2 OMP_NESTED=true
./nested
```

5. What is the difference? Why?

Nested parallelism

Shaping the tree

1. Now, change the code so the nested level only creates as many threads as the parent id+1
   - Thread 0 creates a nested parallel region of 1
   - Thread 1 creates a nested parallel region of 2
   - ...

Tip: Use either `omp_set_num_threads` or `num_threads`

Locks

Exclusive access

1. Edit the file lock.c and take a look at the code
2. Parallelize the first two loops of the application
3. Now run it several times with different numbers of threads
4. We see that result differs because of improper synchronization
5. Use critical to fix it
    - What problem do we have?

Locks

Locks to the help

1. Use locks to implement a fine grain locking scheme
2. Assign a lock to each position of the array *a*
3. Then use it to lock only that position in the main loop
   - Does it work better?
4. Now compare it to an implementation using atomic

Part V

OpenMP in the future

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

The OpenMP Language Committee

Body that prepares new standard versions for the ARB.

- Composed by representatives of all ARB members
  - Lead by Bronis de Supinski from LLNL
- Integrates the information about the different subcommittees
- Currently working on OpenMP 3.1

The OpenMP Subcommittees

When a topic is deemed important or too complex usually a separate group is formed (with a subset of the same people usually).
Currently, the following subcommittees exist:

1. Error model subcommittee
   - In charge of defining an error model for OpenMP
2. Tasking subcommittee
   - In charge of defining new extensions to the tasking model
3. Affinity subcommittee
   - In charge of breaking the flat memory model
4. Accelerators subcommittee
   - In charge of integrating accelerator computing into OpenMP
5. Interoperability and Composability subcommittee

What can we expect in the future?

Disclaimer

- This are my subjective appreciations.
- All these dates and topics are my guessings.
- They might or might not happen.

Tentative Timeline

November 2010	3.1 Public comment version
May 2011	3.1 Final version
June 2012	4.0 Public comment version
November 2012	4.0 Final version

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

Clarifications

Several clarifications to different parts of the specification

- Nothing exciting but needs to be done

Atomic extensions

Extensions to the **atomic** construct to allow:

- to do atomic writes

```
#pragma omp atomic
    x = value;
```

- to capture the value before/after the atomic update

```
#pragma omp atomic
    v = x, x--;
```

User-defined reductions

Allow the users to extend reductions to cope with non-basic types and non-standard operators.

- In 3.1
  - Including pointer reductions in C
  - Including class members and operators in C++
- In 4.0
  - Array for C
  - Template reductions for C++

User-defined reductions

Example

```
#pragma omp declare reduction(+:std::string:omp_out += omp_in)

void foo ()
{
  std::string s;

  #pragma omp parallel reduction(+:s)
  {
     s += "I'm␣a␣thread"
  }

  std::cout << s << std::endl;

}
```

Affinity extensions

New environment variables

- **`OMP_PROCBIND`**=true, false
  - Portable mechanism to bind threads
- Extend **`OMP_NUM_THREADS`** to support multiple levels of parallelism
- **`OMP_AFFINITY`**=scatter,compact
  - Specifies how threads should be distributed in the machine
- **`OMP_MEMORY_PLACEMENT`**=first_touch|round_robin|random
  - Portable mechanisms to specify memory placement policies

Tasking extensions

New constructs/clause

- the **`taskyield`** construct to allow user-defined scheduling points
- the **`final`** clause to allow the optimization of leaf tasks

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

Error model

- Allow the programmer to catch and react to runtime errors
- Integrate C++ exceptions into this model
- Allow the programmer to cancel nicely the parallel computation

It looks like we are leaning towards a model based on callbacks

Error model

Example

```
void error_handler ( omp_err_info_t *info , int *nths )
{
   if ( omp_get_error_type(info) == OMP_ERR_NOT_ENOUGH_THREADS )
      *nths = *nths > 1 ? *nths -1 : 1;
   return OMP_RETRY;
}

nths = 4;
#pragma omp parallel onerror(error_handler,&nths) num_threads(nths)
{
  ....
}
```

Other tasking improvements

- Tasking reductions
  - Add a **`reduction`** clause to the **`task`** construct
- Tasking dependences
  - Allow finer tasking synchronizations by means of expressing data dependences among tasks
- Scheduling hints for the runtime
  - Allow the programmer to express some kind of task priority

Task dependences

Example

```
for ( ; ; ) {
   char *buffer;
   #pragma omp task output(buffer)
   {
       buffer = malloc(...);
       stage1(buffer);
   }
   #pragma omp task inout(buffer)
   {
       stage2(buffer)
   }
   #pragma omp task input(buffer)
   {
       stage3(buffer)
   }
}
```

Accelerators support

- Discussion is in the very early stages.
    - Several proposals on the table
- Cover both data and task parallelism
- Will probably take care of the backend compilation

A glimpse into BSC proposal

Example

```
int main( void ){
for (int i = 0; i < NB; i++)
   for (int j = 0; j < NB; j++)
      for (int k = 0; k < NB; k++)
      #pragma omp target device(smp,cell) \
                  copy_in([BS][BS] A, [BS][BS] B, [BS][BS] C) \
                  copy_out([BS][BS] C)
      #pragma omp task inout([BS][BS] C)
         matmul ( A[i][k], B[k][j], C[i][j] );
}
```

Outline

OpenMP is Open

Compunity

Compunity represents the OpenMP User's Group.

- It is an special ARB member
  - Representative: Barbara Chapman from Univ of Houston
- Anyone can join and participate
  - and also give feedback

OpenMP Forum

- Forum oversighted by ARB members
  - OpenMP usage forum
  - Spec clarifications forum
- Several 3.1 clarifications have its origin in comments from users

Where to go now?

- http://www.openmp.org
- http://www.compunity.org
- http://nanos.ac.upc.edu