OpenMP Tasking Explained

Ruud van der Pas
Senior Principal Software Engineer
SPARC Microelectronics
ORACLE
Santa Clara, CA, USA

SC’13 Talk at OpenMP Booth
Wednesday, November 20, 2013
What Was Missing?
Before OpenMP 3.0

- Constructs worked well for many cases
- But OpenMP’s Big Brother had to see everything
  - Loops with a known length at run time
  - Finite number of parallel sections
  - ....
- This didn’t work well with certain common problems
  - Linked lists and recursive algorithms being the cases in point
- Often, a solution was feasible, but ugly at best
Today’s All New Episode

TASKING
Tasking was introduced in OpenMP 3.0

Until then it was impossible to efficiently and easily implement certain types of parallelism

The initial functionality was very simple by design

The idea was (and still) is to augment tasking as we collectively gain more insight and experience

Note that tasks can be nested

But not for the faint of heart
The Tasking Concept In OpenMP
Who Does What And When?

**Developer**

Use a pragma to specify where the tasks are

(The assumption is that all tasks can be executed independently)

**OpenMP runtime system**

- When a thread encounters a task construct, a new task is generated
- The moment of execution of the task is up to the runtime system
- Execution can either be immediate or delayed
- Completion of a task can be enforced through `task synchronization`
The Tasking Construct

<table>
<thead>
<tr>
<th>#pragma omp task</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$omp task</td>
</tr>
</tbody>
</table>

Defines a task
Task Synchronization

There are two task synchronization constructs:

```c
#pragma omp barrier
```

```c
#pragma omp taskwait
```

```c
!$omp barrier
```

```c
!$omp taskwait
```

```c
#pragma omp taskwait
```
Task Completion

Explicitly wait on the completion of child tasks:

- #pragma omp taskwait
- !$omp flush taskwait
Tasking Explained By Ways Of One Example
A Simple Plan

Your Task for Today:

Write a program that prints either “A race car” or “A car race” and maximize the parallelism
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("A ");
    printf("race ");
    printf("car ");

    printf("\n");
    return(0);
}
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    } // End of parallel region
    printf("\n");
    return(0);
}
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car

Note that this program could (for example) also print

“A A race race car car” or
“A race A car race car”, or
“A race A race car car”, or

.....

But I have not observed this (yet)
What will this program print using 2 threads?
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car

But of course now only 1 thread executes ......
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {
                printf("race ");
            }
            #pragma omp task
            {
                printf("car ");
            }
        }
        // End of parallel region
    }
    printf("\n");
    return(0);
}

What will this program print using 2 threads?
Tasks can be executed in arbitrary order
Another Simple Plan

You did well and quickly, so here is a final task to do

Have the sentence end with “is fun to watch”
(hint: use a print statement)
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {
                printf("race ");
            }
            #pragma omp task
            {
                printf("car ");
            }
            printf("is fun to watch ");
        }
    } // End of parallel region
    
    printf("\n");
    return(0);
}
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {
                printf("car ");
            }
            #pragma omp task
            {
                printf("race ");
            }
            #pragma omp taskwait
            printf("is fun to watch ");
        }
    } // End of parallel region
    printf("\n");return(0);
}
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch

Tasks are executed first now
Thank You And ..... Stay Tuned!

ruud.vanderpas@oracle.com
More About Tasking

- As the computation progresses, the work performed per task may shrink
  - Recursive algorithms are an example
  - This is where the `final` clause may come handy
- The data environment can also grow too much
  - This is why the `mergeable` clause has been added
Example – Linked List/1

........
while(my_pointer) {

(void) do_independent_work (my_pointer);
my_pointer = my_pointer->next ;
} // End of while loop
........

Hard to do before tasking:
First count number of iterations, then convert while loop to for loop
Walking through the linked list is a serial process

- Scan each entry until the NULL pointer has been hit

How do we create the tasks then?

The idea is actually quite simple:

- Use the **single** construct: one thread generates the tasks
- All other threads execute the tasks as they become available
Example – Linked List/3

```c
my_pointer = listhead;

#pragma omp parallel
{
    #pragma omp single
    {
        while(my_pointer)
            #pragma omp task firstprivate(my Pointer)
            {
                (void) do_independent_work (my_pointer);
            }
        my_pointer = my_pointer->next;
    }
} // End of single
} // End of parallel region
```

OpenMP Task is specified here (executed in parallel)
Example – Linked List/4

```c
my_pointer = listhead;

#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

Can eliminate a barrier
Main Tasking Extensions in 4.0

- The `depend` clause to support task dependences
  - Forces additional constraints on task scheduling
  - Expressed through: list item(s) + dependence type
  - Dependence types are: in, out and inout

- The `taskgroup` construct
  - Specifies to wait on completion of child tasks and their descendant tasks
  - Note: `taskwait` only joins direct child tasks