How To Get OpenMP Tasks To Do Your Work
Leverage Tasking to Make Your Life Easier

Ruud van der Pas
Webinar, September 22, 2021
My background is in mathematics and physics

Previously, I worked at the University of Utrecht, Convex Computer, SGI, and Sun Microsystems

Currently I work in the Oracle Linux Engineering organization

I have been involved with OpenMP since the introduction

I am passionate about performance and OpenMP in particular
How to Get OpenMP Tasks to Do Your Work

Copyright © Ruud van der Pas 2021, All Rights Reserved

https://www.openmp.org

A printed copy of the 5.1 OpenMP specs is also available on Amazon (700+ pages, 1.7 kg)
Food for the Eyes and Brains

OpenMP 2.5 and intro Parallel Computing
Covers the OpenMP Basics to get started
Focus on the Advanced Features
What goes on under the hood

NEW!
Why Tasks are Needed
The OpenMP Compiler Had To Know Everything

And in advance, (right) before execution

For example, the loop length, number of parallel sections, etc

Gets hard with more dynamic problems like processing linked lists, divide and conquer, recursion, etc

A solution was ugly. At best
Tasking came to the rescue!

*In this talk we will show it all works*

*But no formal terminology, definitions, etc.*

*These are covered in the specs*
Talking about the Specifications - 3.0

OpenMP Application Program Interface

Version 3.0 May 2008

About 4+ pages on the tasking feature (aside from definitions, scoping, etc)

2.6.3 parallel workshare Construct ........................................ 58
2.7 task Construct ............................................................... 59
2.7.1 Task Scheduling .......................................................... 62
2.8 Master and Synchronization Constructs .............................. 63
2.8.1 master Construct ......................................................... 63
2.9.1 making Construct ....................................................... 65
Talking about the Specifications - 5.1

About 22+ pages on the tasking feature (aside from definitions, scoping, etc)

2.12 Tasking Constructs ................................................................. 161
  2.12.1 task Construct ............................................................... 161
  2.12.2 taskloop Construct .......................................................... 166
  2.12.3 taskloop simd Construct .................................................. 171
  2.12.4 taskyield Construct .......................................................... 173
  2.12.5 Initial Task ................................................................. 174

2.12.6 Task Scheduling ............................................................... 175

2.13 Memory Management Directives .............................................. 177

  2.19.5 taskwait Construct .......................................................... 261
  2.19.6 taskgroup Construct ......................................................... 264
The Tasking Concept
**What Is a Task?**

**A task is a chunk of independent work**

You guarantee that different tasks can be executed simultaneously

```c
#pragma omp task
{“this is my task”}
```
When are Tasks Executed?

The OpenMP run time system decides on the scheduling of tasks

At certain points (implicit and explicit) tasks are guaranteed to be completed
This is what you need to do:

- Identify independent portions of work, the tasks
- Use the `#pragma omp task` construct to define the tasks

The OpenMP runtime system handles everything else:

- When a thread encounters a task construct, a new task is created
- It is up to the runtime system when this task is executed and by whom
- Execution of a task is either immediate, or may be delayed/deferred
- Task synchronization may be used to enforce task completion
Task Creation and Execution in OpenMP

- `#pragma omp task` defines a task.
- `barrier (implied or explicit)`
- `#pragma omp taskwait`
- `#pragma omp taskgroup`

Task Synchronization Points
A Common Execution Model with Tasks

Thread

Generate tasks

Execute tasks

Thread

Thread

Thread

Thread
A Day at the Races
Write a program that prints either “A race car” or “A car race” and maximize the parallelism
The Starting Point

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("A ");
    printf("race ");
    printf("car ");
    printf("\n");
    return(0);
}
```

```
$ gcc hello.c
$ ./a.out
A race car
$

What will this program print?
The Parallel Version

```c
#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);
}
```

What will this program print using 2 threads?
$ gcc -fopenmp 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
.....
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
Let's Try It

$ gcc -fopenmp hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car

**But of course only 1 thread executes now ...**
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);
}
Tasks may be executed in arbitrary order
More to Come

That went 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);
}
Oops!

```bash
$ gcc -fopenmp 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
$

Tasks are executed at a Task Synchronization Point
(and not necessarily where you see them ...)
```
Where is the Task Synchronization Point?

```c
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 ");
        }  // Implied barrier
    } // End of parallel region
    printf("\n"); return(0);
}
```

Task synchronization point
Use the Taskwait Feature

```c
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);
}
```

What will this program print using 2 threads?

Task synchronization point

“The taskwait construct specifies a wait on the completion of child tasks of the current task”
Mission Accomplished!

$ gcc -fopenmp 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
$

The tasks are executed first now
Algorithm Examples
### When to Use Tasks?

<table>
<thead>
<tr>
<th>If you need them</th>
<th>Tasks do not replace everything</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>For example, if the loop construct works well, don’t touch it</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Consider tasks when things are much more dynamic</strong></td>
<td></td>
</tr>
<tr>
<td><strong>If you can’t predict how much work will be done, for example</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Or if the workload is unbalanced, or if ...</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Three Scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>An Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Irregular problems</em></td>
<td><em>Scan a linked list</em></td>
</tr>
<tr>
<td><em>Dependencies</em></td>
<td><em>Overlap I/O and computation</em></td>
</tr>
<tr>
<td><em>Recursion</em></td>
<td><em>A sorting algorithm</em></td>
</tr>
</tbody>
</table>
Scan a Linked List
Scan Through a Linked List

```c
1  my_pointer = head_of_list;
2  while (my_pointer != NULL)
3    {
4      (void) do_independent_work(my_pointer->value);
5      my_pointer = my_pointer->next;
6    } // End of while loop
```

This is cumbersome to do without tasking:

- First count the number of passes through the while-loop
- Convert the while-loop to a for-loop
The Problem and the Idea

The Problem

• Scanning through the linked list is a serial process
• Check each record until the NULL pointer is encountered

The Idea

• Make the execution of function do_independent_work a task
• Use the single construct to have a single thread generate the tasks
• The other threads start executing the tasks as they become available
The Relevant OpenMP Code Fragment

```c
1 my_pointer = head_of_list;

2 #pragma omp parallel firstprivate(my_pointer)
3  {
4    #pragma omp single nowait
5      {
6        while (my_pointer != NULL)
7          {
8            #pragma omp task firstprivate(my_pointer)
9              {
10               (void) do_independent_work(my_pointer->value);
11              } // End of task
12            my_pointer = my_pointer->next;
13          } // End of while loop
14        } // End of single region
15 } // End of parallel region
```

Removes the implied barrier here
Overlap I/O and Computation
A Common Problem

The processing part waits for data to arrive from disk

In case the data is read in chunks, tasking may help

The main idea is to overlap I/O and computations

This keeps the processor (more) busy

An elegant solution makes use of task dependencies
The Main Idea - Set Up a Pipeline

- In reality, the two phases may not take the same time
- Regardless of that, this approach reduces the total time
- In the implementation there will be two tasks
- Through a dependence, we ensure processing does not start too early
- Although not shown here, a third stage may handle the output processing
It is possible to set up dependencies between tasks

This is implemented through the `depend` clause

Various dependence types are supported

In this case, we need an `out` and `in` dependence pair

This is used to express that the computation depends on the I/O

Let’s look at the code to see how this works
The OpenMP Code Structure

```c
for (int64_t i=0; i<n_io_chunks; i++) {
    (void) read_input(&status_read[i],...);  // Read block “i”

    (void) compute_results(...);            // Process block “i”

    // End of for-loop
}
```
A Sorting Algorithm
The Quicksort Algorithm

A commonly used sorting algorithm

A divide-and-conquer strategy is used

The sequential algorithm:

1. Select a pivot
2. Re-order the elements in the array to be sorted such that:
   • All elements to the left of the pivot are smaller
3. Repeat for the parts to the left and right of the pivot
An Example of the Quicksort Algorithm in Action

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

select pivot
### The Sequential Code

#### in the main program:

```c
(int64_t *a, int64_t lo, int64_t hi)
{
    if ( lo < hi ) {
        p = partition(a, lo, hi);
        (void) quicksort(a, lo, p - 1); // Left
        (void) quicksort(a, p + 1, hi); // Right
    }
    return(p);
}
```
The OpenMP Implementation

```c
#pragma omp parallel ...
{
    #pragma omp single nowait
    { (void) ompQuicksort(a, 0, n-1);}
} // End of parallel region

void ompQuicksort(int64_t *a, int64_t lo, int64_t hi)
{
    if ( lo < hi )
    {
        int64_t p = partitionArray(a, lo, hi);
        #pragma omp task ...
        {(void) ompQuicksort(a, lo, p - 1);}
        #pragma omp task ...
        {(void) ompQuicksort(a, p + 1, hi);}
        #pragma omp taskwait
    }
}
```

In the main program

```
Left task

Right task
```
This algorithm requires some tuning

Tasks are not for free and carry some overhead

If an array section gets too small, it may be more efficient to switch to the sequential version

This is not so elegant, but it is efficient :-)

---

This page is extracted from the book "How to Get OpenMP Tasks to Do Your Work" by Ruud van der Pas. All rights reserved.
Tuning Algorithms that Use Tasks

The tasking system is very dynamic

In general, with tasking, there are some things to keep in mind

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many tasks should be created?</td>
</tr>
<tr>
<td>Should small tasks be merged at some point?</td>
</tr>
<tr>
<td>Should tasks be forced to execute as soon as possible?</td>
</tr>
<tr>
<td>How about relative priorities?</td>
</tr>
<tr>
<td>Is it necessary to keep a task tied to the same thread?</td>
</tr>
</tbody>
</table>
## Tuning Controls for Tasking

<table>
<thead>
<tr>
<th>Clause</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>final(expr)</code></td>
<td>If <code>expr</code> is true, stop generating more tasks</td>
</tr>
<tr>
<td><code>mergeable</code></td>
<td>Merge the data environment</td>
</tr>
<tr>
<td><code>untied</code></td>
<td>The task is not tied to a specific thread</td>
</tr>
<tr>
<td><code>if (expr)</code></td>
<td>If <code>expr</code> is false, generate an undeferred* task</td>
</tr>
<tr>
<td><code>priority(hint)</code></td>
<td>Give a priority hint to the runtime system</td>
</tr>
</tbody>
</table>

*) In simple terms, an undeferred task is executed immediately*
The Parallel Quicksort in Practice
Preliminary Findings

We have run some tests with the parallel quicksort algorithm

Please keep in mind that this is very early work and incomplete

The quicksort algorithm is a good target for additional tuning

- if-clause
- final-clause
- switch point for the sequential version
- how to handle idle threads
- NUMA controls
- ...
We used an 8 core VM with 16 hardware threads

Intel Xeon Platinum 8167M CPU @ 2.00GHz (“Skylake”)

We sorted 40M 64 bit integers (320 MB of data)

The algorithmic parameters were:

• final-clause: cutoff at 4000 elements (0.01% of the length)
• switch point for the sequential version: 400 (0.001% of the length)
The Performance Using 8 Cores

A speed up of 5.5 on 8 cores is not bad at all!
The Dynamic Behaviour Using 16 Threads

Preliminary Results!
Use Color Coding to Distinguish the Phases

Preliminary Results!

= sequential version

= taskwait
Many Stones Unturned
What Was not Talked About

The focus of this talk has been on the functionality

Many important topics have not even been touched upon

For example, the taskloop, task_reduction, and taskyield features

Data scoping has been skipped as well

These are useful/relevant topics though and there is more, so please check for more information about tasking
At SC21 there is a full day tutorial on tasking on November 14 “Mastering Tasking with OpenMP”

Additional OpenMP tutorials at SC21:

- The OpenMP Common Core - A Hands-on Introduction
- Advanced OpenMP - Host Performance and 5.1 Features
- Programming your GPU with OpenMP - A Hands-on Introduction
- See also:

  https://sc21.supercomputing.org/program/tutorials/#schedule
You can download a set of example codes
They have been updated for 5.1
The book “Using OpenMP - The Next Step” covers many of the 4.5 features, including a full chapter on tasking.

Although this covers 4.5, it provides a solid introduction into the more advanced topics and prepares you for 5.1.
Thank You And ... Stay Tuned!

Bad OpenMP
Does Not Scale

Ruud van der Pas
Webinar, September 22, 2021