Tasking in OpenMP

IWOMP Tutorial: 30th September 2015

Christian Terboven
Michael Klemm

Members of the OpenMP Language Committee
Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Agenda

- Intro by Example: Sudoku
- Data Scoping
- Example: Fibonacci
- Scheduling and Dependencies
- Taskloops
- More Tasking Stuff
Intro by Example: Sudoku
Let's solve Sudoku puzzles with brute multi-core force.

(1) Find an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:
   Delete number,
   Insert next number

(4 b) If valid:
   Go to next field
The OpenMP Task Construct

Each encountering thread/task creates a new task
- Code and data is being packaged up
- Tasks can be nested
  - Into another task directive
  - Into a Worksharing construct

Data scoping clauses:
- `shared(list)`
- `private(list)`    `firstprivate(list)`
- `default(shared | none)`
Barrier and Taskwait Constructs

- OpenMP `barrier` (implicit or explicit)
  - All tasks created by any thread of the current `Team` are guaranteed to be completed at barrier exit

```
C/C++
#pragma omp barrier
```

- Task barrier: `taskwait`
  - Encountering task is suspended until child tasks are complete
  - Applies only to direct childs, not descendants!

```
C/C++
#pragma omp taskwait
```
# Parallel Brute-force Sudoku

This parallel algorithm finds all valid solutions:

1. Search an empty field
2. Insert a number
3. Check Sudoku
4. If invalid: Delete number, insert next number
4. If valid: Go to next field
5. Wait for completion

```c
// First call contained in a
#pragma omp parallel
#pragma omp single
such that one task starts the
execution of the algorithm

#pragma omp task
needs to work on a new copy
of the Sudoku board

#pragma omp taskwait
wait for all child tasks
```

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>9</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>15</td>
<td>11</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>4</td>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>9</td>
<td>12</td>
<td>13</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>16</td>
<td></td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>10</td>
<td>11</td>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>10</td>
<td></td>
<td></td>
<td>7</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

---

*Advanced OpenMP Tutorial*
Christian Terboven, Michael Klemm
OpenMP parallel region creates a team of threads

```c
#pragma omp parallel
{
  #pragma omp single
  solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

→ Single construct: One thread enters the execution of `solve_parallel`

→ the other threads wait at the end of the `single` ...

→ ... and are ready to pick up threads „from the work queue“

Syntactic sugar (either you like it or you don‘t)

```c
#pragma omp parallel sections
{
  solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```
The actual implementation

```c
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
    if (!sudoku->check(x, y, i)) {
        #pragma omp task firstprivate(i,x,y,sudoku)
        {
            // create from copy constructor
            CSudokuBoard new_sudoku(*sudoku);
            new_sudoku.set(y, x, i);
            if (solve_parallel(x+1, y, &new_sudoku)) {
                new_sudoku.printBoard();
            }
        }
    }
} // end omp task
```

#pragma omp task

need to work on a new copy of the Sudoku board

```c
#pragma omp taskwait
```

wait for all child tasks
Performance Evaluation

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding
- Speedup: Intel C++ 13.1, scatter binding

Is this the best we can can do?
Performance Analysis

Event-based profiling gives a good overview:

Every thread is executing ~1.3m tasks...

... in ~5.7 seconds.

=> average duration of a task is ~4.4 μs

Tracing gives more details:

Tasks get much smaller down the call-stack.
**Performance Analysis**

Event-based profiling gives a good overview:

Every thread is executing ~1.3m tasks...
- if-clause
- final-clause, mergeable-clause
- natively in your program code

Example: stop recursion

... in ~5.7 seconds.

=> average duration of a task is ~4.4 μs

Tracing gives more details:

Tasks get much smaller down the call-stack.
Performance Evaluation

Sudoku on 2x Intel Xeon E5-2650 @2.0 GHz

- Intel C++ 13.1, scatter binding
- Intel C++ 13.1, scatter binding, cutoff
- speedup: Intel C++ 13.1, scatter binding
- speedup: Intel C++ 13.1, scatter binding, cutoff
Data Scoping
Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply:
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private

- If `shared` scoping is not inherited:
  - Orphaned Task variables are `firstprivate` by default!
  - Non-Orphaned Task variables inherit the `shared` attribute!
  - Variables are `firstprivate` unless `shared` in the enclosing context
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a:
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
Data Scoping Example (3/7)

```c
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
```
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d:
            // Scope of e:
        } } }
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e:
        }
    }
}
Data Scoping Example (6/7)

```c
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}
```
int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            // Scope of a: shared, value of a: 1
            // Scope of b: firstprivate, value of b: 0 / undefined
            // Scope of c: shared, value of c: 3
            // Scope of d: firstprivate, value of d: 4
            // Scope of e: private, value of e: 5
        }
    }
}
Use default(none)!

int a = 1;
void foo()
{
    int b = 2, c = 3;
    #pragma omp parallel shared(b) default(none)
    #pragma omp parallel private(b) default(none)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}

Hint: Use default(none) to be forced to think about every variable if you do not see clear.
Example: Fibonacci
Recursive approach to compute Fibonacci

```c
int main(int argc,
        char* argv[]) {
    [...] fib(input);
    [...]}
```

```c
int fib(int n) {   
    if (n < 2) return n;
    int x = fib(n - 1);
    int y = fib(n - 2);
    return x+y;
}
```

- On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.
Only one Task / Thread enters `fib()` from `main()`, it is responsible for creating the two initial work tasks

Taskwait is required, as otherwise `x` and `y` would be lost
Scalability measurements (1/3)

- Overhead of task creation prevents better scalability!

![Graph showing speedup of Fibonacci with Tasks](image-url)
**Improved parallelization with Tasking (omp-v2)**

- **Improvement:** Don‘t create yet another task once a certain (small enough) \( n \) is reached

```c
int main(int argc, char* argv[]) {
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) \
    if(n > 30) {
        x = fib(n - 1);
    }
    #pragma omp task shared(y) \
    if(n > 30) {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```
Scalability measurements (2/3)

- Speedup is ok, but we still have some overhead when running with 4 or 8 threads

![Graph showing speedup of Fibonacci with tasks]

Legend:
- optimal
- omp-v1
- omp-v2
**Improved parallelization with Tasking (omp-v3)**

- **Improvement**: Skip the OpenMP overhead once a certain $n$ is reached (no issue w/ production compilers)

```c
int main(int argc, char* argv[]) {
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}

int fib(int n) {
    if (n < 2) return n;
    if (n <= 30)
        return serfib(n);
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```
Scalability measurements (3/3)

Everything ok now 😊
Scheduling and Dependencies
Tasks in OpenMP: Scheduling

- Default: Tasks are *tied* to the thread that first executes them → not necessarily the creator. Scheduling constraints:
  - Only the thread a task is tied to can execute it
  - A task can only be suspended at task scheduling points
    - Task creation, task finish, taskwait, barrier, taskyield
  - If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all tasks tied to the thread

- Tasks created with the *untied* clause are never tied
  - Resume at task scheduling points possibly by different thread
  - No scheduling restrictions, e.g. can be suspended at any point
  - But: More freedom to the implementation, e.g. load balancing
Unsafe use of untied Tasks

- Problem: Because untied tasks may migrate between threads at any point, thread-centric constructs can yield unexpected results.

- Remember when using untied tasks:
  - Avoid threadprivate variable
  - Avoid any use of thread-ids (i.e. `omp_get_thread_num()`)
  - Be careful with critical region and locks

- Simple Solution:
  - Create a tied task region with
    ```
    #pragma omp task if(0)
    ```
If the expression of an `if` clause on a task evaluates to `false`

→ The encountering task is suspended

→ The new task is executed immediately

→ The parent task resumes when new tasks finishes

→ Used for optimization, e.g. avoid creation of small tasks
The taskyield Directive

The `taskyield` directive specifies that the current task can be suspended in favor of execution of a different task.

→ Hint to the runtime for optimization and/or deadlock prevention

```
C/C++
#pragma omp taskyield

Fortran
!$omp taskyield
```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
    {
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
    }
}
taskyield Example (2/2)

```c
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
    {
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
    }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations.
The depend Clause

C/C++

```c
#pragma omp task depend(dependency-type: list)
... structured block ...
```

- The *task dependence* is fulfilled when the predecessor task has completed
  - *in* dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - *out* and *inout* dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
  - The list items in a `depend` clause may include array sections.
Concurrent Execution w/ Dep.

Degree of parallelism exploitable in this concrete example:
T2 and T3 (2 tasks), T1 of next iteration has to wait for them

void process_in_parallel() {
    #pragma omp parallel
    #pragma omp single
    {
        int x = 1;
        ...
        for (int i = 0; i < T; ++i) {
            #pragma omp task shared(x, ...) depend(out: x) // T1
            preprocess_some_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T2
            do_something_with_data(...);
            #pragma omp task shared(x, ...) depend(in: x) // T3
            do_something_independent_with_data(...);
        }
    } // end omp single, omp parallel
}
Concurrent Execution w/ Dep.

- The following allows for more parallelism, as there is one $i$ per thread. Hence, two tasks may be active per thread.

```c
void process_in_parallel() {
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < T; ++i) {
            #pragma omp task depend(out: i)
            preprocess_some_data(...);
            #pragma omp task depend(in: i)
            do_something_with_data(...);
            #pragma omp task depend(in: i)
            do_something_independent_with_data(...);
        }
    } // end omp parallel
}
```
Concurrent Execution w/ Dep.

- The following allows for even more parallelism, as there now can be two tasks active per thread per i-th iteration.

```c
void process_in_parallel() {
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < T; ++i) {
            #pragma omp task firstprivate(i)
            {
                #pragma omp task depend(out: i)
                preprocess_some_data(...);
                #pragma omp task depend(in: i)
                do_something_with_data(...);
                #pragma omp task depend(in: i)
                do_something_independent_with_data(...);
            } // endomp task
        } // endomp single, endomp parallel
    }
```
void blocked_cholesky( int NB, float A[NB][NB] ) {
    int i, j, k;
    for (k=0; k<NB; k++) {
        #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
        for (i=k+1; i<NT; i++) {
            #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm (A[k][k], A[k][i]);
            // update trailing submatrix
            for (i=k+1; i<NT; i++) {
                for (j=k+1; j<i; j++)
                    #pragma omp task depend(in:A[k][i],A[k][j])
                    depend(inout:A[j][i])
                    sgemm( A[k][i], A[k][j], A[j][i] );
                    #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
                    ssyrk (A[k][i], A[i][i]);
            }
        }
    }
}
The `taskloop` Construct
The taskloop Construct

- Parallelize a loop using OpenMP tasks
  - Cut loop into chunks
  - Create a task for each loop chunk

- Syntax (C/C++)
  #pragma omp taskloop [simd] [clause[[], clause],...]
  for-loops

- Syntax (Fortran)
  !$omp taskloop[simd] [clause[[], clause],...]
  do-loops
  !$omp end taskloop [simd]
Clauses for `taskloop` Construct

- Taskloop constructs inherit clauses both from worksharing constructs and the `task` construct
  - `shared`, `private`
  - `firstprivate`, `lastprivate`
  - `default`
  - `collapse`
  - `final`, `untied`, `mergeable`

- `grainsize` (`grain-size`)
  Chunks have at least `grain-size` and max $2 \times \text{grain-size}$ loop iterations

- `num_tasks` (`num-tasks`)
  Create `num-tasks` tasks for iterations of the loop
Example: Sparse CG

```c
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}
```

```c
void matvec(Matrix *A, double *x, double *y) {
    // ...
    #pragma omp parallel for \
    private(i,j,is,ie,j0,y0) \ 
    schedule(static)
    for (i = 0; i < A->n; i++) {
        y0 = 0;
        is = A->ptr[i];
        ie = A->ptr[i + 1];
        for (j = is; j < ie; j++) {
            j0 = index[j];
            y0 += value[j] * x[j0];
        }
        y[i] = y0;
    }  
    // ...
}
```
Example: Sparse CG

#pragma omp parallel
#pragma omp single
for (iter = 0; iter < sc->maxIter; iter++) {
    precon(A, r, z);
    vectorDot(r, z, n, &rho);
    beta = rho / rho_old;
    xpay(z, beta, n, p);
    matvec(A, p, q);
    vectorDot(p, q, n, &dot_pq);
    alpha = rho / dot_pq;
    axpy(alpha, p, n, x);
    axpy(-alpha, q, n, r);
    sc->residual = sqrt(rho) * b;
    if (sc->residual <= sc->tolerance) break;
    rho_old = rho;
}

void matvec(Matrix *A, double *x, double *y) {
    // ...
}

#pragma omp taskloop private(j,is,ie,j0,y0) \ 
grain_size(500)
for (i = 0; i < A->n; i++) {
    y0 = 0;
    is = A->ptr[i];
    ie = A->ptr[i + 1];
    for (j = is; j < ie; j++) {
        j0 = index[j];
        y0 += value[j] * x[j0];
    }
    y[i] = y0;
} // ...

Awesome !!!
More Tasking Stuff
final Clause

- For recursive problems that perform task decomposition, stop task creation at a certain depth exposes enough parallelism while reducing the overhead.

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp task final(expr)</td>
<td>!$omp task final(expr)</td>
</tr>
</tbody>
</table>

- Merging the data environment may have side-effects

```c
void foo(bool arg)
{
    int i = 3;
    #pragma omp task final(arg) firstprivate(i)
        i++;
    printf("%d\n", i);  // will print 3 or 4 depending on expr
}
```
mergeable Clause

- If the mergeable clause is present, the implementation might merge the task‘s data environment
  - if the generated task is undeferred or included
    - undeferred: if clause present and evaluates to false
    - included: final clause present and evaluates to true

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp task mergeable</td>
<td>!$omp task mergeable</td>
</tr>
</tbody>
</table>

- Personal Note: As of today, no compiler or runtime exploits final and/or mergeable so that real world application would profit from using them 😞.
The taskgroup Construct

C/C++
#pragma omp taskgroup
... structured block ...

Fortran
!$omp taskgroup
... structured block ...
!$omp end task

- Specifies a wait on completion of child tasks and their descendent tasks

- „deeper“ synchronization than taskwait, but

- with the option to restrict to a subset of all tasks (as opposed to a barrier)

- Main use case for now in OpenMP 4.0: Cancellation...
The last slide...

```c
#pragma omp parallel for num_threads(2)
for (int i = 0; i < iwomp.num_attendees(); i++) {
    #pragma omp task
    {
        attendee.get_lunch();
    }
}
```