Advanced OpenMP Features

Christian Terboven, Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
{terboven,schmidl}@itc.rwth-aachen.de
Sudoku
Sudoku for Lazy Computer Scientists

- **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

![Sudoku puzzle grid](image-url)
Parallel Brute-force Sudoku

This parallel algorithm finds all valid solutions

(1) Search 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
Wait for completion
Parallel Brute-force Sudoku (2/3)

- 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 taskwait

#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 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 and Scalability Tuning Idea: If you have created sufficiently many tasks to make your cores busy, stop creating more tasks!

Duration: 0.16 sec

Duration: 0.047 sec

Duration: 0.001 sec

Duration: 2.2 μs
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
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)
    ```
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

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#pragma omp taskyield</code></td>
<td><code>!$omp taskyield</code></td>
</tr>
</tbody>
</table>
#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);
        }
}
```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; may also avoid deadlock situations.
The taskgroup Construct

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp taskgroup</td>
<td>!$omp taskgroup</td>
</tr>
<tr>
<td>... structured block ...</td>
<td>... structured block ...</td>
</tr>
<tr>
<td>!$omp end task</td>
<td></td>
</tr>
</tbody>
</table>

- Specifies a wait on completion of child tasks and their descendant tasks
  - „deeper“ synchronization than taskwait, but
  - with the option to restrict to a subset of all tasks (as opposed to a barrier)
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 to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

```c
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
}
```
The following code allows for more parallelism, as there is one i per thread. Thus, 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(...);
            } // end omp task
        } // end omp single, end omp 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
                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]);
        }
    }
}
void blocked_cholesky( int NB, float A[NB][NB] ) {
    int i, j, 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])
    ssysr (A[k][i], A[i][i]);
}

[...]
The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. Until now, libraries like PLASMA had to rely on custom built task schedulers; [...] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory. **We view OpenMP as the natural path forward for the PLASMA library and expect that others will see the same advantages to choosing this alternative.**

taskloop Construct
Traditional Worksharing

- Worksharing constructs do not compose well
- Pathological example: parallel dgemm in MKL

```c
void example() {
    #pragma omp parallel
    {
        compute_in_parallel(A);
        compute_in_parallel_too(B);
        // dgemm is either parallel or sequential,
        // but has no orphaned worksharing
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    m, n, k, alpha, A, k, B, n, beta, C, n);
    }
}
```

- Writing such code either
  - oversubscribes the system,
  - yields bad performance due to OpenMP overheads, or
  - needs a lot of glue code to use sequential dgemm only for sub-matrixes
Ragged Fork/Join

- Traditional worksharing can lead to ragged fork/join patterns

```c
void example() {
    compute_in_parallel(A);
    compute_in_parallel_too(B);
    cblas_dgemm(..., A, B, ...);
}
```
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;
    }  
    // ... 
}
```
The taskloop Construct

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

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

- Syntax (Fortran)
  ```fortran
  !$omp taskloop[simd] [clause[[], clause],…]
  do-loops
  !$omp end taskloop [simd]]
  ```
Claususes 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 \((\text{grain-size})\)
  Chunks have at least \(\text{grain-size}\) and max \(2^{*}\text{grain-size}\) loop iterations

- num\_tasks \((\text{num-tasks})\)
  Create \(\text{num-tasks}\) tasks for iterations of the loop
OpenMP Tasking In-Depth
C. Terboven | IT Center der RWTH Aachen University

Example: Sparse CG

```c
#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) * bnrm2;
    if (sc->residual <= sc->tolerance)
        break;
    rho_old = rho;
}
```

```c
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;
    }
    // ...
}
```
More Tasking Stuff
priority Clause

C/C++

```c
#pragma omp task priority(priority-value)
... structured block ...
```

- **The *priority* is a hint to the runtime system for task execution order**
- **Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones**
  - priority is non-negative numerical scalar (default: 0)
  - priority <= max-task-priority ICV
    - environment variable OMP_MAX_TASK_PRIORITY

- **It is not allowed to rely on task execution order being determined by this clause!**
For recursive problems that perform task decomposition, stopping task creation at a certain depth exposes enough parallelism but reduces overhead.

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

### C/C++

```c
#pragma omp task mergeable
```

### Fortran

```fortran
!$omp task mergeable
```

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