OpenMP Tasking

The slides are jointly developed by:

Christian Terboven
Michael Klemm
Ruud van der Pas
Eric Stotzer
Bronis R. de Supinski

Members of the OpenMP Language Committee
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.
Agenda

- Intro by Example: Sudoku
- Data Scoping
- Scheduling and Dependencies
- Taskloops
- More Tasking Stuff
Intro by Example: Sudoku
Sudoku for Lazy Computer Scientists

- Lets solve Sudoku puzzles with brute multi-core force

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

(1) Search an empty field

(2) Try all numbers:

(2 a) Check Sudoku

(2 b) If invalid: skip

(2 c) If valid: Go to next field

Wait for completion
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 complete
  
  ➞ Applies only to children, not descendants!

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

- This parallel algorithm finds all valid solutions.

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

- **(1)** Search an empty field.
- **(2)** Try all numbers:
  - **(2 a)** Check Sudoku
  - **(2 b)** If invalid: skip
  - **(2 c)** If valid: Go to next field

- **OpenMP Taskwait**: 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
```
Parallel Brute-force Sudoku (3/3)

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

need to work on a new copy of the Sudoku board

wait for all child tasks
Performance Evaluation

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

Runtime [sec] for 16x16

#threads

Is this the best we can do?
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
Data Scoping Example (1/7)

```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:
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        } } }
```
Data Scoping Example (2/7)

```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:
            // 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:
            // 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:
        } } }
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)!

```c
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 clearly.
Scheduling and Dependencies
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

```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
}
```

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

Variables in the depend clause do not necessarily have to indicate the data flow.
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
    }
} // 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[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]) ;
            }
    }
}
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
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++)
  #pragma omp taskloop [simd] [clause[[], clause],...]
  for-loops

- Syntax (Fortran)
  !$omp taskloop[simd] [clause[[], clause],...]
  do-loops
  !$omp end taskloop [simd]
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*grain-size loop iterations

num_tasks (num-tasks)
Create num-tasks tasks for iterations of the loop
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;
} // ...
```
Summary

- Traditional threading should be considered outdated
- Tasking is more flexible and relieves programmers from thinking about mapping execution to threads
- OpenMP has a powerful tasking model (since almost a decade, BTW)