Using OpenMP for Intranode Parallelism

Useful Information

Bronis R. de Supinski

Paul Petersen

Thanks to: Tim Mattson (Intel), Ruud van der Pas (Oracle), Christian Terboven (RWTH Aachen University), Michael Klemm (Intel)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.
Outline

- Scheduling loop iterations
- Nested Computation
- Arbitrary Tasks
- NUMA Optimizations
- Memory Model
Scheduling loop iterations

- OpenMP provides different algorithms for assigning loop iterations to threads
- This is specified via the schedule() clause of the worksharing construct

```c
!$omp do schedule(static)
do i=1,n
    a(i) = ....
end do
```

```c
#pragma omp for schedule(static)
for (i = 0; i < N; ++i)
    a[i] = ....
```
Loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - `schedule(static[,chunk])`
    - Deal-out blocks of iterations of size “chunk” to each thread
    - Pre-determined and predictable by the programmer
    - When chunk=1 you get round-robin (or cyclic) scheduling
  - `schedule(dynamic[,chunk])`
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled
  - `schedule(guided[,chunk])`
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds
  - `schedule(runtime)`
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library)
  - `schedule(auto)`
    - Schedule is left up to the runtime to choose (does not have to be any of the above)
Loops (cont.)

• **Use `schedule(runtime)` for more flexibility**
  - allow implementations to implement their own schedule kinds
  - can get/set it with library routines
    ```c
    omp_set_schedule()
    omp_get_schedule()
    ```

• Schedule kind `auto` gives full freedom to the runtime to determine the scheduling of iterations to threads.

• **NOTE:** C++ random access iterators are allowed as loop control variables in parallel loops
Choosing the “right” schedule clause

• The goal of loop scheduling is to balance the work assigned to each thread in the team
• Many factors interact, so sometime experimentation is necessary
• Triangular loop nests usually are better with (static,N) or (dynamic,N) rather than (static)
• It may help to arrange your loop so the iterations with the largest execution time are assigned first
Barrier: Necessary across adjacent loops?

- OpenMP guarantees that this works ... i.e. that the same schedule is used in the two loops
- You must ensure that all data accesses to the same location are aligned to the same iteration

```
!$omp do schedule(static)  
do i=1,n
   a(i) = ....
end do

!$omp end do nowait

!$omp do schedule(static)  
do i=1,n
   .... = a(i)
end do

#pragma omp for \  
schedule(static) nowait
for (i = 0; i < N; ++i)
   a[i] = ....

#pragma omp for \  
schedule(static)
for (i = 0; i < N; ++i)
   .... = a[i]
```
Outline

• Scheduling loop iterations
• Nested Computation
• Arbitrary Tasks
• NUMA Optimizations
• Memory Model
Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```c
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        ....
    }
}
```

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop may complicate balancing the load.
Nested parallelism

- Allows parallel regions to be contained in each other
- Often done dynamically by having parallel regions in different functions
- Total number of threads created is the *product* of the number of threads in the teams at each level
- Requires: OMP_NESTED=true or omp_set_nested(1) otherwise the inner parallel region will be executed by a team of one thread (may happen anyway)
- Use omp_set_num_thread(n) or the num_threads() clause
- Multiple levels of nesting team sizes can be defined via the OMP_NUM_THREADS environment variable
  - setenv OMP_NUM_THREADS 4,2
Nested parallelism
(illustrated)

- The OpenMP runtime organizes threads in a pool.

New features in 4.0 support mapping threads to cores
Outline

• Scheduling loop iterations
• Nested Computation
• Arbitrary Tasks
• NUMA Optimizations
• Memory Model
Arbitrary tasks

- Counted loops are often a natural means of organizing the computation in a program.
- But sometimes you need the ability to partition arbitrary computation between the threads.
- Or you may need the ability to parallelize more than “counted loops”, such as “while loops” or computations expressed as “recursive function calls”.

Basic OpenMP: Sections worksharing construct

• The *Sections* worksharing construct gives a different structured block to each thread.

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        X_calculation();
        #pragma omp section
        y_calculation();
        #pragma omp section
        z_calculation();
    }
}
```

By default, there is an implicit barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
Combining nesting and sections

• Creating nested activity is quite common
  – Modular programming creates abstraction boundaries
• Sections allow arbitrary work units but are not composable
• Nested parallel regions often cause unexpected results

Tasking in OpenMP combines the best of these two ideas
The OpenMP task construct

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

- 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)`
void walk_list( node head ) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            node p = head;
            while (p) {
                #pragma omp task
                {
                    process( p );
                }
                p = p->next;
            }
        }
    }
}
Sudoko for lazy computer scientists

Let's solve Sudoku puzzles with brute multi-core search

(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
This parallel algorithm finds all valid solutions:

1. Search an empty field
2. Insert a number
3. Check Sudoku
4. If invalid:
   a. Delete number, insert next number
   b. Go to next field

Parallel brute-force sudoku (1/3)

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

#pragma omp task

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
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“
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
    }
} // end omp task
```

#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
Task Synchronization
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 suspends until child tasks are complete
    - Only child tasks, not their descendants!
      
      ```
      C/C++
      #pragma omp taskwait
      ```
Tasking in Detail
General OpenMP scoping rules

- Managing the data environment is required in OpenMP

- **Scoping** in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on parallel region
  - *private*-list and *shared*-list on worksharing constructs
  - General default is *shared*, *firstprivate* for tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to parallel regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master's value / value captured at task creation
    - *lastprivate*: Value of last loop iteration is written back to master
  - Static variables are *shared*
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;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a:
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a: shared
            // Scope of b:
            // Scope of c:
            // Scope of d:
            // Scope of e:
        }
    }
}
int a;
void foo() {

    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c:
            // Scope of d:
            // Scope of e:
        } } } }
int a;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d:
            // Scope of e:
        }  
    }  
}  
}  
}
int a;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e:
        }
    }
}
int a;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}
int a;
void foo()
{
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;

            // 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.
Task 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 to which a task is tied can execute the task
  - A task can only be suspended at a task scheduling point
    - Task creation, task finish, taskwait, barrier
  - 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
  - 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
If clause

- 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
**final clause**

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

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

- **Warning:** 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
}
```
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++

```c
#pragma omp taskyield
```

Fortran

```fortran
!$omp taskyield
```


Taskyield example (1/2)

```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);
        }
    }
}
```
#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);
        }
    }
}
Outline

• Scheduling loop iterations
• Nested Computation
• Arbitrary Tasks
• NUMA Optimizations
• Memory Model
OpenMP and performance

- The transparency and ease of use of OpenMP are a mixed blessing
  - Makes things pretty easy
  - May mask performance bottlenecks

- In an ideal world, an OpenMP application “just runs well”. Unfortunately, this is not always the case…

- Two of the more obscure things that can negatively impact performance are cc-NUMA effects and false sharing

- Neither of these are caused by OpenMP
  - But they most show up because you used OpenMP
  - In any case they are important enough to cover here
Memory hierarchy

- In modern computer design memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory

- Access follows the scheme:
  - Registers whenever possible
  - Then the cache
  - At last the main memory

- "DRAM Gap"
Cache coherence (cc)

- If there are multiple caches not shared by all cores in the system, the system takes care of the cache coherence.

**Example:**

```c
int a[some_number]; //shared by all threads
thread 1: a[0] = 23;    thread 2: a[1] = 42;
--- thread + memory synchronization (barrier) ---
thread 1: x = a[1];    thread 2: y = a[0];
```

- Elements of array `a` are stored in continuous memory range
- Data is loaded into cache in 64 byte blocks (cache line)
- Both `a[0]` and `a[1]` are stored in caches of thread 1 and 2
- After synchronization point all threads need to have the same view of (shared) main memory

- The system is not able to distinguish between changes within one individual cache line.
False sharing

- False sharing: Storing data into a shared cache line invalidates the other copies of that line!

- Caches are organized in lines of typically 64 bytes: integer array $a[0-4]$ fits into one cache line.

- Whenever one element of a cache line is updated, the whole cache line is invalidated.

- Local copies of a cache line have to be re-loaded from main memory and the computation may have to be repeated.
False sharing indicators

- Be alert, if all of these three conditions are met
  - Shared data is modified by multiple processors
  - Multiple threads operate on the same cache line(s)
  - Update occurs simultaneously and very frequently

- Use local data where possible

- Shared read-only data does not lead to false sharing
Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread

double* A;
A = (double*) malloc(N * sizeof(double));

for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
First touch memory placement

- First touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core that executes the thread that initializes the respective partition

```c
double* A;
A = (double*) malloc(N * sizeof(double));
omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
```
Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 ("Westmere") using Intel® Composer XE 2013 compiler with different thread binding options:

- Serial vs. Parallel initialization
Peak Performance is only achievable if everything is done right (NUMA, Vectorization, FLOPS, …)!
Outline

• Scheduling loop iterations
• Nested Computation
• Arbitrary Tasks
• NUMA Optimizations
• Memory Model
The OpenMP memory model (1)

- All threads have access to the same, globally shared memory.
- Data in private memory is only accessible by the thread that owns this memory.
- No other thread sees the change(s) in private memory.
- Data transfer is through shared memory and is 100% transparent to the application.
OpenMP and relaxed consistency

• OpenMP supports a relaxed-consistency shared memory model.
  – Threads can maintain a temporary view of shared memory that is not consistent with that of other threads.
  – These temporary views are made consistent only at certain points in the program.
  – The operation that enforces consistency is called the flush operation
The OpenMP memory model (2)

- Need to get this right
  → Part of the learning curve
- Private data is undefined on entry and exit
  → Can use firstprivate and lastprivate to address this
- Each thread has its own temporary view on the data
  → Applicable to shared data only
  → Means different threads may temporarily not see the same value for the same variable ...

- Let me illustrate the problem we have here...
If shared variable X is kept within a register, the modification may not be made visible to the other thread(s).

```
Thread A

x = 0
.
.
.
.
x = 1
.
.
.
.

Thread B

while (x == 0)
{
   "wait"
}
```
The *flush* directive (2)

- Example of the flush directive, source taken from “Using OpenMP” pipeline code example

```c
void wait_read(int i)
{

    #pragma omp flush

    while ( execution_state[i] != READ_FINISHED )
    {

        system("sleep 1");

        #pragma omp flush
    }

} /*-- End of wait_read --*/
```
Flush operation

• Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  – All previous read/writes by this thread have completed and are visible to other threads
  – No subsequent read/writes by this thread have occurred
  – A flush operation is analogous to a **fence** in other shared memory API’s
Flush and synchronization

• A flush operation is implied by OpenMP synchronizations, e.g.
  – at entry/exit of parallel regions
  – at implicit and explicit barriers
  – at entry/exit of critical regions
  – whenever a lock is set or unset
  
  ....

  (but not at entry to worksharing regions or entry/exit of master regions)
What is the big deal with flush?

• Compilers routinely reorder instructions implementing a program
  – This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
• Compiler generally cannot move instructions:
  – past a barrier
  – past a flush on all variables
• But it can move them past a flush with a list of variables so long as those variables are not accessed
• Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.
The `flush` directive (3)

- Strongly recommended: do **not** use this directive with a list
  - Could give very subtle interactions with compilers
  - If you insist on still doing so, be prepared to face the OpenMP language lawyers
  - Necessary much less often with the addition of sequentially consistent atomics in OpenMP 4.0

- Implied on many constructs
  - A good thing
  - This is your safety net

- Really, try to avoid at all, if possible!
Conclusion

• OpenMP is powerful and flexible APIs that gives you the control you need to create high-performance applications

• We covered a wide variety of advanced topic exploring the effective use of OpenMP
  – Scheduling loop iterations
  – Nested Computation
  – Arbitrary Tasks
  – NUMA Optimizations
  – Memory Model

• Next steps?
  – OpenMP is in active evolution to target the latest machine architectures.
  – Start writing parallel code … you can only learn this stuff by writing lots of code.