RWTH AACHEN UNIVERSITY

OpenMP Performance Tuning - Part 1 -

Christian Terboven <terboven@rz.rwth-aachen.de>

29.08.2012 / Aachen, Germany

Stand: 27.08.2012

Version 2.3

Review of OpenMP Introduction

- OpenMP is a parallel programming model for Shared-Memory machines. That is, all threads have access to a *shared* main memory. In addition to that, each thread may have *private* data.

- The parallelism has to be expressed explicitly by the programmer. The base construct is a *Parallel Region*:
  A *Team* of threads is provided by the runtime system.

- Using the *Worksharing* constructs, the work can be distributed among the threads of a team. The *Task* construct defines an explicit task along with it‘s data environment. Execution may be deferred.

- To control the parallelization, mutual exclusion as well as thread and task synchronization constructs are available.

Agenda

- The *Schedule* Clause
- Avoiding Overhead: The *If* Clause and the *Final* Clause
- Implicit Barriers and the *Nowait* Clause
- The *Memory Mod*el and the *Flush* Construct
- OpenMP Locks
- The *Taskyield* Directive
- More Worksharing: The *Section* and *Ordered* Constructs
- More *Environment Variables* and *API* Functions

Schedule Clause

Load Imbalance

Influencing the For Loop Scheduling

- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- **Default on most implementations is `schedule(static)`.**

If and Final Clause

If Clause: Parallel Region and Worksharing

- **If the expression of an `if` clause on a *Parallel Region* evaluates to false**
  - The Parallel Region is executed with a Team of one Thread only

  → Used for optimization, e.g. avoid going parallel
- **OpenMP data scoping rules still apply!**

C/C++

```
#pragma omp parallel if(expr)
    ...
```

Fortran

```
!$omp parallel if(expr)
    ...
```

If and Final Clause: Tasking

- **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
- **If the expression of a `final` clause on a *Task* evaluates to true**
  - All child tasks will be final and included, that means they will be executed sequentially in the task region, immediately by the encountering thread
- **OpenMP data scoping rules still apply!**

C/C++

```
#pragma omp task if(expr)
    ...
#pragma omp task final(expr)
    ...
```

Fortran

```
!$omp task if(expr)
    ...
!$omp task final(expr)
    ...
```

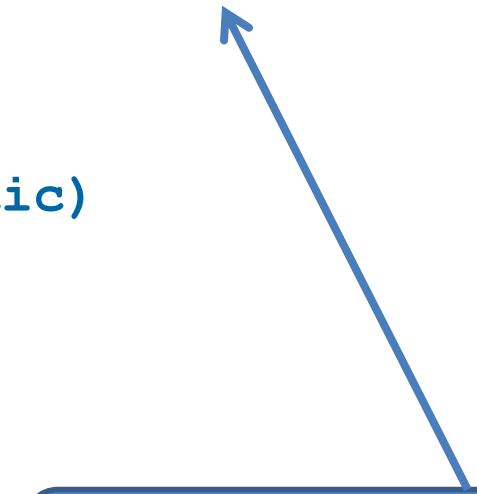
Nowait Clause

The nowait Clause

- A worksharing construct (do/for, sections, single) has no barrier on entry – however, an implied barrier exists at the end of the worksharing region, unless the *nowait* clause is specified.

- Static schedule guarantees since OpenMP 3.0:

```
#pragma omp for schedule(static) nowait
    for(i = 1; i < N; i++)
        a[i] = …
#pragma omp for schedule(static)
    for (i = 1; i < N; i++)
        c[i] = a[i] + …
```



Allowed in OpenMP 3.0 if and only if:
- Number of iterations is the same
- Chunk is the same (or not specified)

The collapse Clause

- **Loop collapsing: Ask the compiler to fuse perfectly nested loops to exploit a larger iteration space for the parallelization:**

```
#pragma omp for collapse(2)
    for(i = 1; i < N; i++)
        for(j = 1; j < M; j++)
            for(k = 1; k < K; k++)
                foo(i, j, k);
```

Iteration space from i-loop and j-loop is collapsed into a single one, if loops are perfectly nested and form a rectangular iteration space.

Memory Model and Flush Construct

Memory Model of OpenMP

- **OpenMP: Shared-Memory model**
  - All threads share a common address space (shared memory)
  - Threads can have private data (explicit user control)
  - Fork-Join execution model
- **Weak memory model**
  - Temporary View: Memory consistency is guaranteed only after certain points, namely implicit and explicit `flush`es
- **Any OpenMP `barrier` includes a `flush`**
- **Entry to and exit from `critical` regions include a `flush`**
- **Entry to and exit from lock routines (OpenMP API) include a `flush`**

The flush directive

C/C++

```
#pragma omp flush [(list)]
```

- **Enforces shared data to be consistent (but be cautious!)**
  - If a thread has updated some variables, their values will be flushed to memory, thus accessible to other threads
  - If a thread has not updated a value, the construct will ensure that any local copy will get latest value from memory
- **BUT: Do not use this for thread synchronization**
  - Compiler optimization might come in your way
  - Rather use OpenMP lock functions for thread synchronization

OpenMP Locks

- **OpenMP provides a set of low-level locking routines, similar to semaphores:**
  - `void omp_`*`func`*`_lock (omp_lock_t *lck)`, with func:
    - init / init_nest: Initialize the lock variable
    - destroy / destroy_nest: Remove the lock variable association
    - set / set_nest: Set the lock, wait until lock acquired
    - test / test_nest: Set the lock, but test and return if lock could not be acquired
    - unset / unset_nest: Unset the lock
  - Argument is address to an instance of `omp_lock_t` type
  - Simple lock: May not be locked if already in a locked state
  - Nested lock: May be locked multiple times by the same thread

Taskyield Directive

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
```

taskyield

taskyield Example (1/2)

```
#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)

```
#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.

Section and Ordered Construct

How to parallelize a Tree Traversal?

- How would you parallelize this code?

```
void traverse (Tree *tree)
{
    if (tree->left)    traverse(tree->left);

    if (tree->right)    traverse(tree->right);

    process(tree);
}
```

- One option: Use OpenMP‘s parallel sections.

How to parallelize a Tree Traversal?!

- **How would you parallelize this code?**

```
   void traverse (Tree *tree)
   {
#pragma omp parallel sections
{
#pragma omp section
       if (tree->left)    traverse(tree->left);
#pragma omp section
       if (tree->right)    traverse(tree->right);
} // end omp parallel
       process(tree);
   }
```

Nested Parallel Regions

Barrier here!

- **Downsides of this option:**
  - Unneccessary overhead and synchronization points
  - Not always well supported (how many threads to be used?)

How to parallelize a Tree Traversal!

- **Better option: Use Tasking in OpenMP 3.0!**

```
   void traverse (Tree *tree)
   {
#pragma omp task
       if (tree->left)    traverse(tree->left);
#pragma omp task
       if (tree->right)    traverse(tree->right);

       process(tree);
   }
```

Assume a Parallel Region to exist outside the scope of this routine.

No Barrier here!

The Section Construct

The ordered Construct

- **Allows to execute a structured block within a parallel loop in sequential order**
  - In addition, an `ordered` clause has to be added to the *for* construct which any *ordered* construct may occur

C/C++

```
#pragma omp ordered
    ...
```

Fortran

```
!$ omp ordered
    ...
!$ omp end ordered
```

- **Use Cases:**
  - Can be used e.g. to enforce ordering on printing of data
  - May help to determine whether there is a data race

More Env. Var. and APIs

OpenMP Environment Variables (1/2)

- `OMP_NUM_THREADS`: Controls how many threads will be used to execute the program.
- `OMP_SCHEDULE`: If the schedule-type *runtime* is specified in a schedule clause, the value specified in this environment variable will be used.
- `OMP_DYNAMIC`: The OpenMP runtime is allowed to smartly guess how many threads might deliver the best performance. If you want full control, set this variable to *false*.
- `OMP_NESTED`: Most OpenMP implementations require this to be set to *true* in order to enabled nested Parallel Regions. Remember: Nesting Worksharing constructs is not possible.

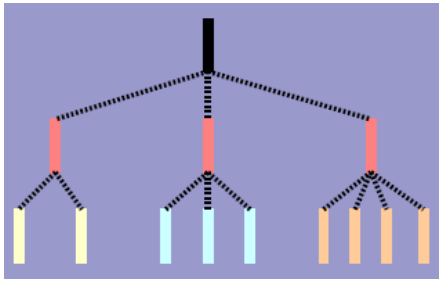
OpenMP Environment Variables (2/2)

- **Define interaction with system environment:**
  - Env. Var. OMP_MAX_NESTED_LEVEL + API functions
    - Controls the maximum number of active parallel regions
  - Env. Var. OMP_THREAD_LIMIT + API functions
    - Controls the maximum number of OpenMP threads
  - Env. Var. OMP_STACKSIZE
    - Controls the stack size of child threads
  - Env. Var. OMP_WAIT_POLICY
    - Control the thread idle policy:
      - active: Good for dedicated systems (e.g. in batch mode)
      - passive: Good for shared systems

Other API routines

- **API support for Nested Parallelism:**
  - How many (active) nested Parallel Regions?
    - `int omp_get_level(), int omp_get_active_level()`
  - Which thread-id was my ancestor, in given level?
    - `int omp_get_ancestor_thread_num(int level)`
  - How many Threads were in my ancestor‘s team, at given level?
    - `int omp_get_team_size(int level)`

```
omp_set_num_threads(3);
#pragma omp parallel {
  omp_set_num_threads(omp_get_thread_num() + 2);
  #pragma omp parallel {
     foo();
} }
```



The End

Thank you for your attention.