RWTH AACHEN UNIVERSITY

OpenMP Performance Tuning

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

13.03.2013 / Aachen, Germany

Stand: 07.03.2013

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

- OpenMP Thread Binding Howto
- The *Schedule* Clause
- Avoiding Overhead: The *If* Clause, Implicit Barriers and the *Nowait* Clause
- The *Memory Mod*el and the *Flush* Construct
- The *Taskyield* Directive

OpenMP Thread Binding Howto

Step 1: Get Info on the System Topology

- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**
  - Intel MPI's `cpuinfo` tool
    - `module switch openmpi intelmpi`
    - `cpuinfo`
    - Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.
  - hwlocs' `hwloc-ls` tool
    - `hwloc-ls`
    - Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.

Step 2: Decide for Binding Strategy

- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - Putting threads far apart, i.e. on different sockets
    - May improve the aggregated memory bandwidth available to your application
    - May improve the combined cache size available to your application
    - May decrease performance of synchronization constructs
  - Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - May improve performance of synchronization constructs
    - May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

Step 3: Implement Binding Strategy

Intel C/C++/Fortran Compiler

- Use environment variable `KMP_AFFINTIY`
  - `KMP_AFFINITY=scatter`: Put threads far apart
  - `KMP_AFFINITY=compact`: Put threads close together
  - `KMP_AFFINITY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
  - Add “`,verbose`“ to print out binding information to stdout.

GNU C/C++/Fortran Compiler

- use environment variable GOMP_CPU_AFFINITY
  - `GOMP_CPU_AFFINTIY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.

OpenMP 4.0: Places + Binding Policies (1/2)

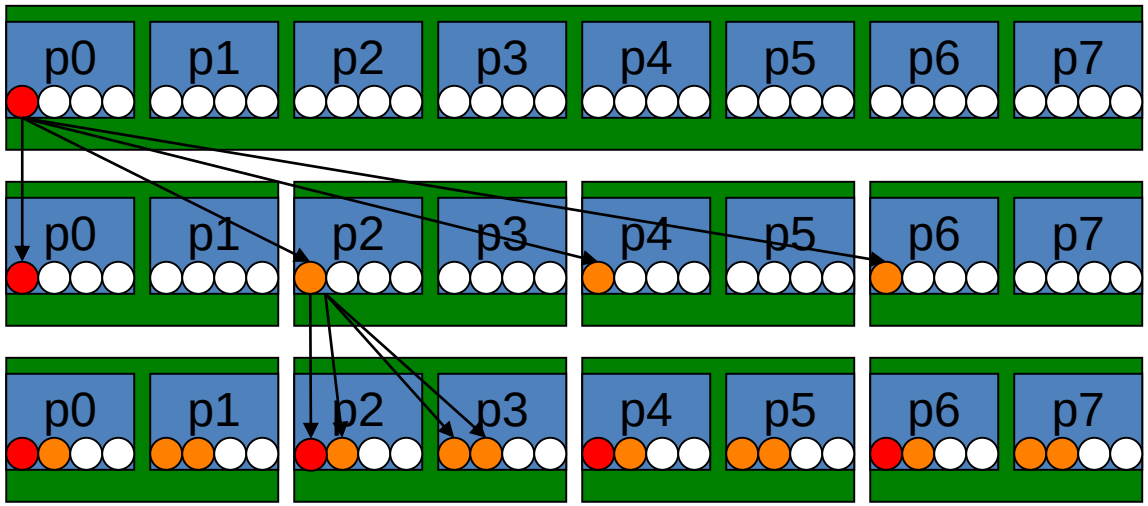
- **Define OpenMP Places**
  - set of OpenMP threads running on one or more processors
  - can be defined by the user
- **Define a set of OpenMP Thread Affinity Policies**
  - SPREAD: spread OpenMP threads evenly among the places
  - CLOSE: pack OpenMP threads near master thread
  - MASTER: collocate OpenMP thread with master thread
- **Goals**
  - user has a way to specify where to execute OpenMP threads for
  - locality between OpenMP threads / less false sharing / memory bandwidth

OpenMP 4.0: Places + Binding Policies (2/2)

- **Example's Objective:**
  - separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)**
  - spread creates partition, compact binds threads within respective partition

```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)
```

- **Example**
  - initial
  - spread 4
  - compact 4



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)`.**

Avoiding Overhead

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 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

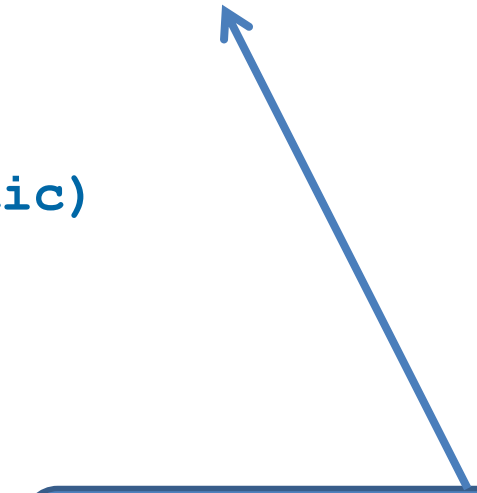
  → Used for optimization, e.g. avoid creation of small tasks

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.

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.

The End

Thank you for your attention.