OpenedMP Overview

Getting Up To Speed On OpenMP 4.0

Ruud van der Pas

ORACLE

Distinguished Engineer
Architecture and Performance
SPARC Microelectronics, Santa Clara, CA, USA

HPCS 2015
Amsterdam, The Netherlands
July 20, 2015

Agenda

• The One Hour Elevator Ride
• Using OpenMP
  - Directives
  - Environment Variables
  - Run time Functions
• Additional Functionality
• Performance Tuning Case Studies

The One Hour Elevator Ride

1. It's a Fan!
2. It's a Wall!
3. It's a Snake!
4. It's a Tree!

What Is OpenMP?

Just One Recent Example

"... Building using OpenMP in general, this is a bad idea, since OpenMP tends to much slower than pthreads for ATLAS use ...

... In ATLAS, thread affinity is the main reason pthreads wins against OpenMP."

Building using OpenMP is in general, this is a bad idea, since OpenMP tends to much slower than pthreads for ATLAS use. However, if your own application uses OpenMP, sometimes pthread usage can slow down your own thread, making it worthwhile to damage ATLAS performance in order to improve your OpenMP performance. In ATLAS, thread affinity is the main reason pthreads wins against OpenMP, and on OS-X (and probably FreeBSD), which don't support real affinity, are the platforms where it makes sense to use OpenMP regardless. To force ATLAS to use OpenMP rather than pthreads, you must add the following flags to your configure line:

--enable-shared --enable-openmp

http://math-atlas.sourceforge.net/errata.html#ompIn

Help and architectural defaults for 64-bit Windows

1997 - 2013

Shared Memory

0 1 P

L L L
OpenMP Overview

V1
RvdP
Ge#ng
Up
To
Speed
On
OpenMP
4.0

> July 2013

OpenMP

http://www.openmp.org
http://iwomp.org
http://openmpconf.org
http://www.compunity.org

OpenMP 4.0
(July 2013)

“Using OpenMP”
Portable Shared Memory Parallel Programming
Chapman, Jost, van der Pas
MIT Press, 2008
List price: ~35 $US

All 41 examples are available online!
Check out the forum on http://www.openmp.org

Download the examples and participate in the forum:

What Is OpenMP?

• The de-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
• Consists of Compiler Directives, Runtime routines and Environment variables
• Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
• Current version: 4.0 (released July 2013)
  ◦ The draft 4.1 specs are expected later this year
OpenMP is widely supported by the industry, as well as the academic community.

When To Consider OpenMP?
- Using an automatically parallelizing compiler
  - The compiler can not find the parallelism
- The data dependence analysis is not able to determine whether it is safe to parallelize
  - The granularity is not high enough, but you know better
- Compiler lacks the information to parallelize at a higher level
- Not using an automatically parallelizing compiler
  - No other choice than to parallelize yourself
  - Compilers can still help (e.g. auto-scoping, warnings, etc.)

The Advantages Of OpenMP
- De-facto and mature standard
  - Strong focus on backward compatibility
  - Continues to evolve and stays up to date
- Good performance and scalability
  - If you do it right ....
  - Portability
    - Supported by a large number of compilers
  - Requires little programming effort
    - But, ..... 
  - Allows the program to be parallelized incrementally
    - But, ..... 

OpenMP And Multicore
OpenMP is ideally suited for multicore architectures
Memory and threading model map naturally
Lightweight
Mature
 Widely available and used

Directive Format
- C: directives are case sensitive
  - Syntax: #pragma omp directive [clause [clause] ...]
  - Continuation: use \ in pragma
  - Conditional compilation: _OPENMP macro is set
- Fortran: directives are case insensitive
  - Syntax: sentinel directive [clause [clause] [clause] ...]
  - The sentinel is one of the following:
    - !$OMP or !$OMP or *$OMP (fixed format)
    - !$OMP (free format)
  - Continuation: follows the language syntax
  - Conditional compilation: F or CS \rightarrow 2 spaces

A parallel region is a block of code executed by all threads in the team

```c
#pragma omp parallel [clause[{},] clause] ...
{
  "this code is executed in parallel"
} // End of parallel section (note: implied barrier)
```

```fortran
 !$omp parallel [clause[{},] clause] ...
  "this code is executed in parallel"
 !$omp end parallel (note: implied barrier)
```
OpenMP Overview

Parallel Region - An Example/1

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Hello World\n");
    return(0);
}
```

Parallel Region - An Example/2

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        printf("Hello World\n");
    } // End of parallel region
    return(0);
}
```

Parallel Region - An Example/3

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$ 
```

The If Clause

```c
#pragma omp parallel if (n > some_threshold)
    shared(n,x,y) private(i)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            x[i] += y[i];
    } /*-- End of parallel region --*/
```

A More Elaborate Example

```c
#pragma omp parallel if (n>limit) default(none) 
    shared(n,m,a,b,c,x,y,z) private(f,i,scale)
    {
        f = 1.0;
        #pragma omp for
        for (i=0; i<n; i++)
            x[i] = a[i] + y[i];
        #pragma omp for
        for (i=0; i<n; i++)
            x[i] = b[i] + c[i];
        #pragma omp barrier
        synchronization
        scale = sum(a,0,m) + sum(z,0,n) + f;
    } /*-- End of parallel region --*/
```

Nested Parallelism

![Nested Parallelism Diagram]

- Master Thread
- Outer parallel region
- Nested parallel region
- Outer parallel region

Note: Nesting level can be arbitrarily deep.
Components Of OpenMP

- Worksharing
- Tasking
- Affinity
- Accelerators
- Cancellation
- Synchronization
- Thread Settings
- Work Scheduling
- Affinity
- Accelerators
- Cancellation
- Operational
- Thread Management
- Work Scheduling
- Tasking
- Affinity
- Accelerators
- Cancellation
- Operational
- Synchronization
- Environment variables
- Thread Settings
- Thread Controls
- Work Scheduling
- Affinity
- Accelerators
- Cancellation
- Operational
- Work Scheduling
- Affinity
- Accelerators
- Cancellation
- Operational
- Synchronization

The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning 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

Data-Sharing Attributes

- In an OpenMP program, data needs to be “labeled”
- Essentially there are two basic types:
  - Shared - There is only one instance of the data
    - Threads can read and write the data simultaneously unless protected through a specific construct
    - All changes made are visible to all threads
      - But not necessarily immediately, unless enforced
  - Private - Each thread has a copy of the data
    - No other thread can access this data
    - Changes only visible to the thread owning the data

The Private And Shared Clauses

private (list)
- No storage association with original object
- All references are to the local object
- Values are undefined on entry and exit

shared (list)
- Data is accessible by all threads in the team
- All threads access the same address space

About Storage Association

- Private variables are undefined on entry and exit of the parallel region
- A private variable within a parallel region has no storage association with the same variable outside of the region
- Use the firstprivate and lastprivate clauses to override this behavior
- We illustrate these concepts with an example

The Firstprivate And Lastprivate Clauses

firstprivate (list)
- All variables in the list are initialized with the value the original object had before entering the parallel construct

lastprivate (list)
- The thread that executes the sequentially last iteration or section updates the value of the objects in the list
Example Firstprivate

```c
n = 2; indx = 4;
#pragma omp parallel default(none) private(i,TID)
  firstprivate(indx) shared(n,a)
  { TID = omp_get_thread_num();
    indx = indx + n*TID;
    a[i] = TID + 1;
  } /*-- End of parallel region --*/
```

<table>
<thead>
<tr>
<th>TID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Example Lastprivate

```c
#pragma omp parallel for lastprivate(a)
for (int i=0; i<n; i++)
{ .......
    a = i + 1;
    .......
} // End of parallel region
b = 2 * a; // value of b is 2*n
```

The Default Clause

- default (none | shared | private | threadprivate)
  - none: No implicit defaults; have to scope all variables explicitly
  - shared: All variables are shared
  - The default in absence of an explicit "default" clause
  - private: All variables are private to the thread
    - Includes common block data, unless THREADPRIVATE
  - firstprivate: All variables are private to the thread; pre-initialized

Data Scoping – Gotcha’s

- 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 explain

The Flush Directive

```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 --*/
```

This fragment is from the source code for the pipeline example from "Using OpenMP" (overlaps I/O with processing)
### OpenMP Overview

#### About The Flush Construct

- **Do not use the flush 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 in court
- **The flush is included in many constructs**
  - A Good Thing - This is your safety net
  - Check the specifications for the specific constructs

```
#pragma omp flush ([list])
!$omp flush ([list])
Don't use the list
```

#### Implementing The Fork-Join Model

Master Thread

Worker Threads

Parallel region

Barrier

Parallel region

Worker Threads

Synchronization

**Use the OMP_WAIT_POLICY environment variable to control the behaviour of idle threads**

#### The Barrier/1

Suppose we run each of these two loops in parallel over i:

```
for (i=0; i < n; i++)
a[i] = b[i] + c[i];
```

```
for (i=0; i < n; i++)
d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

**Why?**

#### The Barrier/2

We need to have updated all of a[i] first, before using a[i]

```
for (i=0; i < n; i++)
c[i] = b[i] + c[i];
wait !
```

```
for (i=0; i < n; i++)
d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

*If there is the guarantee that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case*

#### The Barrier/3

Barrier Region

```
#pragma omp barrier
!$omp barrier
```

Barrier syntax in OpenMP:
When To Use Barriers?

- In case data is updated asynchronously and data integrity is at risk
- Examples:
  - Between parts in the code that read and write the same section of memory
  - After one timestep/iteration in a solver
- Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors
- Therefore, use them with care

The OpenMP Barrier

- Several constructs have an implied barrier
  - This is another safety net (has implied flush by the way)
  - In some cases, the implied barrier can be left out through the nowait clause
  - This can help fine tuning the application
    - But you’d better know what you’re doing
  - The explicit barrier comes in quite handy then

```c
#pragma omp barrier
```

The Nowait Clause

- To minimize synchronization, some directives support the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct
  - In C, it is one of the clauses on the pragma
  - In Fortran, it is appended at the closing part of the construct

```c
#pragma omp for nowait
{ ... }
```

OpenMP Directives (An Important Subset)

- A work-sharing construct does not launch any new threads

```c
#pragma omp for
{ ... }
```

The Worksharing Constructs

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless the nowait clause has been specified)
- A work-sharing construct does not launch any new threads

```c
#pragma omp for [nowait]
{ ... }
```

The Fortran Workshare Construct

- Fortran has a fourth worksharing construct:

```fortran
 !$OMP WORKSHARE <array syntax>
```

Example:

```fortran
 !$OMP WORKSHARE A(1:M) = A(1:M) + B(1:M)
```

```fortran
 !$OMP WORKSHARE NOWAIT
```
The Omp-For/Do Directive

```c
#pragma omp for [clauses]
for (.....) {
<code-block>
#pragma omp shared(n,a,b,c,d) private(i)

// Example
#pragma omp do [clauses]
do ...
<code-block>
#pragma omp end do[nowait]
```

The iterations of the loop are distributed over the threads.

The Omp-For/Do Directive – An Example

```c
#pragma omp parallel default(none)\ shared(n,a,b,c,d) private(i)

#pragma omp for nowait
#pragma omp for nowait

for (.....)
{
<code-block>
#pragma omp parallel default(none) &
#pragma omp shared(n,x) private(i)
#pragma omp do reduction (+:sum)
do i = 1, n
sum = sum + x(i)
end do
#pragma omp end do

!
!

/*-- End of parallel region --*/
(implied barrier)
```

The Reduction Clause - Example

```c
sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,a,b,c,d) private(i)
!$omp do reduction (+:sum)
do i = 1, n
sum = sum + x(i)
end do
!$omp end do
!$omp end parallel

// Example
!$omp declare reduction ....
!$omp declare reduction ....
```

Variable SUM is a shared variable.

Care needs to be taken when updating shared variable SUM.
With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided.

The Reduction Clause

```c
reduction ( operator: list )
reduction ( [operator | intrinsic] ) : list )
```

- A reduction is defined as:
  - `C/C++`
  - `x = x operator expr`
  - `x = expr operator x`
  - `x = intrinsic (x, expr_list)`
  - `x = intrinsic (expr_list, x)`
  - `x = min(x, y)` or `x = max(x, y)`
  - `x = intrinsic (x, expr_list)`

- Reduction variable(s) must be shared variables

Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed.

The reduction can be hidden in a function call.

User Defined Reductions (UDR)

- In addition to the pre-defined reduction operators supported, one can now write a user specific reduction operator.
- Combines ease of use of the reduction clause with application specific reductions.
- Restrictions apply (please read the fine print on this).

```c
#pragma omp declare reduction ....
!$omp declare reduction ....
```

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism.
- The collapse clause on for/do loop indicates how many loops should be collapsed.
- Compiler forms a single loop and then parallelizes it.

```c
!$omp parallel do collapse(2) ...
do i = il, iu, is
do j = jl, ju, js
do k = kl, ku, ks
......
end do
end do
!$omp end parallel do
```
SIMD Support/1

- SIMD = Single Instruction Multiple Data
  - Also called 'micro-vectorization'
- A single instruction performs the same operation on multiple data elements in parallel

\[
\begin{align*}
  c[0] & \quad c[1] & \quad c[2] & \quad c[3]
\end{align*}
\]

64-bit

Parallel Sections

```
#pragma omp sections [clauses]
  #pragma omp section
   (....)
  #pragma omp section
   (....)
} //End of sections --*/
```

The Sections Directive – An Example

```
#pragma omp parallel default(none) shared(n,a,b,c,d) private(i)
{
  #pragma omp sections nowait
  { #pragma omp section
      // Code
    } //End of section
  #pragma omp section
      // Code
  } //End of parallel region --*/
```

Overlap I/O And Processing/1

```
Input Thread | Processing Thread(s) | Output Thread
--- | --- | ---
0 | 0 | 0
1 | 1 | 1
2 | 2 | 2
3 | 4 | 3
4 | 5 | 4
5 | 5 | 5
```

Overlap I/O And Processing/2

```
#pragma omp parallel sections
  #pragma omp section
    for (int i=0; i<N; i++)
      (void) signal_read(i);
  #pragma omp section
    for (int i=0; i<N; i++)
      (void) wait_read[i];
    (void) process_data[i];
    (void) signal_processed[i];
} //End of parallel sections --*/
```
OpenMP Overview

Single Processor Region/1

Original Code

```
"read A[0..N-1];
.....
```

Only one thread executes the single region

```
#pragma omp parallel \
shared (A) 
{ 
    ....
    #pragma omp single nowait 
    {"read A[0..N-1]";}
    ....
    #pragma omp barrier 
    "use A" 
}
```

Parallel Version

Ideally suited for I/O or initializations

---

Single Processor Region/2

```

```

Other threads wait in the barrier here

---

The Single Directive

Only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
{ <code-block> }
```

```
#pragma omp single [private][firstprivate] \
{ <code-block> }
```

```
#pragma omp single [private][firstprivate][copyprivate][nowait]
```

```
#pragma omp single [private][firstprivate][copyprivate][nowait]
```

---

Worksharing Shortcuts

```
#pragma omp parallel 
#pragma omp for
for (...) 
#pragma omp parallel do...
```

```
#pragma omp parallel
#pragma omp do...
```

```
#pragma omp parallel sections
```

```
#pragma omp parallel sections
```

```
#pragma omp parallel sections
```

```
#pragma omp parallel sections
```

---

The Critical Region/1

If `sum` is a shared variable, this loop cannot be run in parallel by simply using a "#pragma omp for"

```
for (i=0; i < n; i++){
    ....
    sum += a[i];
} ....
```

```
#pragma omp parallel for for (i=0; i < n; i++){
    ....
    #pragma omp critical 
    {sum += a[i];}
} ....
```

All threads execute the update, but now only one at a time will do so

---

The Critical Region/2

- Very useful to avoid a race condition, or to perform I/O
  - but the order of execution is still undefined
- Be aware there is a performance cost associated with a critical region

Critical region
## OpenMP Overview

### The Critical Construct

```c
#pragma omp critical [name]
{<code-block>}
#pragma omp end critical [name]
```

Very useful to avoid data races.

### The OpenMP Environment Variables

#### OpenMP Environment Variable | Category
--- | ---
OMP_DISPLAY_ENV | Diagnostics
OMP_NUM_THREADS | Thread Management
OMP_THREAD_LIMIT | Thread Management
OMP_DYNAMIC (true | false) | Thread Management
OMP_NESTED (true | false) | Parallelism
OMP_MAX_ACTIVE_LEVELS | Parallelism
OMP_STACKSIZE "size [b|k|m|g]" | Operational

- The names have to be in uppercase; the values are case insensitive.
- Be careful when relying on defaults (they are compiler dependent).

### Environment Variables/1

### About The Stack

```c
void myfunc(float *Aglobal)
{
  int Alocal;
  ..........  
}
```

Variable Alocal is in private memory, managed by the thread owning it, and stored on the so-called stack.

### Environment Variables/2

### The OpenMP Runtime Functions
OpenMP provides a set of runtime functions
They all start with "omp_
These functions can be used to:
- Query for a specific feature or setting
  - For example, the current iteration scheduling policy for loops
- Change the setting
  - For example, to change the iteration scheduling policy for loops
- A special category consists of the locking functions

C/C++: Need to include file <omp.h>
Fortran: Add "use omp_lib" or include file "omp_lib.h"

**RunAme FuncAons/1**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_get_num_threads/omp_set_num_threads</td>
<td>Thread Management</td>
</tr>
<tr>
<td>omp_get_max_threads</td>
<td>Thread Management</td>
</tr>
<tr>
<td>omp_get_thread_num</td>
<td>Thread Management</td>
</tr>
<tr>
<td>omp_get_dynamic/omp_set_dynamic</td>
<td>Thread Management</td>
</tr>
<tr>
<td>omp_in_parallel</td>
<td>Parallelism</td>
</tr>
<tr>
<td>omp_get_nested/omp_set_nested</td>
<td>Parallelism</td>
</tr>
</tbody>
</table>

**RunAme FuncAons/2**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_get_num_procs</td>
<td>Operational</td>
</tr>
<tr>
<td>omp_get_wtime</td>
<td>Operational</td>
</tr>
<tr>
<td>omp_get_wtick</td>
<td>Operational</td>
</tr>
<tr>
<td>omp_init_lock/omp_init_nested_lock</td>
<td>Locking</td>
</tr>
<tr>
<td>omp_set_lock/omp_nested_lock</td>
<td>Locking</td>
</tr>
<tr>
<td>omp_unset_lock/omp_nested_nested_lock</td>
<td>Locking</td>
</tr>
<tr>
<td>omp_test_lock/omp_nested_nested_lock</td>
<td>Locking</td>
</tr>
<tr>
<td>omp_destroy_lock/omp_destroy_nested_lock</td>
<td>Locking</td>
</tr>
</tbody>
</table>

**RunAme FuncAons/3**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_in_final</td>
<td>Tasking</td>
</tr>
<tr>
<td>omp_get_proc_bind</td>
<td>Thread Affinity</td>
</tr>
<tr>
<td>omp_get_cancellation</td>
<td>Cancellation</td>
</tr>
<tr>
<td>omp_get_num_teams</td>
<td>Accelerators</td>
</tr>
<tr>
<td>omp_get_team_num</td>
<td>Accelerators</td>
</tr>
<tr>
<td>omp_get_default_device</td>
<td>Accelerators</td>
</tr>
<tr>
<td>omp_set_default_device</td>
<td>Accelerators</td>
</tr>
<tr>
<td>omp_get_num_devices</td>
<td>Accelerators</td>
</tr>
<tr>
<td>omp_is_initial_device</td>
<td>Accelerators</td>
</tr>
</tbody>
</table>

Using OpenMP
Using OpenMP

- The overview given so far can already be used to parallelize applications
- It is now time to look at some features in more detail
- And introduce some additional features

The Schedule Clause

```c
schedule (static | dynamic | guided | auto [, chunk])
schedule (runtime)
```

- static [, chunk]
  - Distribute iterations in blocks of size “chunk” over the threads in a round-robin fashion
  - In absence of “chunk”, each thread executes approx. \( N/P \) chunks for a loop of length \( N \) and \( P \) threads
    - Details are implementation defined
  - Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region

- dynamic [, chunk]
  - Fixed portions of work; size is controlled by the value of chunk
  - When a thread finishes, it starts on the next portion of work

- guided [, chunk]
  - Same dynamic behavior as “dynamic”, but size of the portion of work decreases exponentially

- auto
  - The compiler (or runtime system) decides what is best to use; choice could be implementation dependent

- runtime
  - Iteration scheduling scheme is set at runtime through environment variable `OMP_SCHEDULE`

Example Of A Static Schedule

A loop of length 16 using 4 threads

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>no chunk</td>
<td>1-4</td>
<td>5-8</td>
<td>9-12</td>
<td>13-16</td>
</tr>
<tr>
<td>chunk = 2</td>
<td>1-2</td>
<td>3-4</td>
<td>5-6</td>
<td>7-8</td>
</tr>
<tr>
<td></td>
<td>9-10</td>
<td>11-12</td>
<td>13-14</td>
<td>15-16</td>
</tr>
</tbody>
</table>

*) The precise distribution is implementation defined

Loop Workload Scheduling Choices

- dynamic [, chunk]
  - Fixed portions of work; size is controlled by the value of chunk

- guided [, chunk]
  - Same dynamic behavior as “dynamic”, but size of the portion of work decreases exponentially

- auto
  - The compiler (or runtime system) decides what is best to use; choice could be implementation dependent

- runtime
  - Iteration scheduling scheme is set at runtime through environment variable `OMP_SCHEDULE`

Experiment – 500 Iterations, 4 Threads

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Iteration Number
The Master Directive

Only the master thread executes this code block

```bash
#pragma omp master
{<code-block>}
```

There is no implied barrier on entry or exit!

```bash
#pragma omp master
{<code-block>}
#pragma omp end master
```

The Ordered Directive

The enclosed block of code within a parallel loop is executed in the order in which iterations would be executed sequentially:

```bash
#pragma omp ordered
{<code-block>}
#pragma omp end ordered
```

May introduce serialization (could be expensive)

Note: Need to use the ordered clause on the parallel for/do loop

The Atomic Directive

“An atomic operation can be executed without any other thread being able to read or change state that is read or changed during the operation”

```bash
#pragma omp atomic update
{
    x++;
}
```

Source: http://wiki.osdev.org/Atomic_operation

The Atomic Construct/1

This captures the original (or final) value, stored in a designated variable

```bash
#pragma omp atomic capture
{capture-statement}
```

```bash
old_value = *p;
(*p)++;
```

// Can use old_value here

Example – The Capture Clause

Clauses on the atomic:
- read, write, update, capture, seq_cst

Lightweight

```bash
#pragma omp atomic [clause]
```
OpenMP Overview

Sequentially Consistent Atomics

"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." – Leslie Lamport

Specify through the seq_cst clause on the atomic construct

This implies a flush without a list

Orphaning

The OpenMP specification does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned.

That is, they can appear outside the lexical extent of a parallel region

Example - Parallelizing Bulky Loops

for (i=0; i<n; i++) /* Parallel loop */
{
    a = ... b = ... c[i] = ...
    for (j=0; j<m; j++)
        <a lot more work in this loop>
} .......

Step 1: "Outlining"

Still a sequential program
- Should behave identically
- Easy to test for correctness
- But, parallel by design

More On Orphaning

When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored.
Step 2: Parallelize

```
#pragma omp parallel for private(..) shared(..)
for (int i=0; i<n; i++) /* Parallel loop */
    { (void) FuncPar(i,m,c,...) }
/*-- End of parallel for --*/
```

- Minimal scoping required
- Less error prone

```
for (int i=0; i<n; i++) /* Parallel loop */
    {
        (void) FuncPar(i,m,c,...)
    }
/*-- End of parallel for --*/
```

### Global Data

**Global Data – An Example/1**

```fortran
program global_data
  use mod_global_data
  !$omp parallel do private(j)
  do j = 1, n
      call suba(j)
  end do
!$omp end parallel do
module mod_global_data
    implicit none
    integer, parameter:: m= ..., n= ...
    integer           :: a(m,n), b(m)
end module mod_global_data
```

Arrays “a” and “b” are shared

**Global Data – A Solution/1**

```fortran
program global_data
  use mod_global_data
  !$omp parallel do private(j)
  do j = 1, n
      call suba(j)
  end do
!$omp end parallel do
module mod_global_data
    implicit none
    integer, parameter:: m= ..., n= ...
    integer           :: a(m,n), b(m,nthreads)
end module mod_global_data
```

**Global Data – A Data Race!**

```
subroutine suba(j)
  !$omp parallel do private(i)
  do i = 1, m
      a(i,1)=func_call(b(i))
  end do
end subroutine

subroutine suba(j)
  !$omp parallel do private(i)
  do i = 1, m
      a(i,2)=func_call(b(i))
  end do
end subroutine
```

Data Race!

```
do i = 1, m
  b(i) = 1
end do
```

```
do i = 1, m
  b(i) = 2
end do
```

```
do i = 1, m
  a(i,1)=func_call(b(i))
end do
```

```
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```

**Global Data – A Solution/2**

```
subroutine suba(j)
  do i = 1, m
      b(i) = 1
  end do
end subroutine
```

```
subroutine suba(j)
  do i = 1, m
      b(i) = 2
  end do
end subroutine
```

**Global Data – A Solution/2**

```
program global_data
  use mod_global_data
  !$omp parallel do private(j)
  do j = 1, n
      call suba(j)
  end do
!$omp end parallel do
module mod_global_data
    implicit none
    integer, parameter:: m= ..., n= ...
    integer           :: a(m,n), b(m,nthreads)
end module mod_global_data
```
**Overview**

**OpenMP**

103

Global Data – A Solution/2

```plaintext
subroutine suba(j)
    ....
    use omp_lib
    use mod_global_data
    ....
    TID = omp_get_thread_num() + 1
    do i = 1, n
        b(i, TID) = j
    end do
    do i = 1, n
        a(i, j) = func_call(b(i), TID)
    end do
    return
end
```

**About Global Data**

- Global data is shared and requires special care.
- A problem may arise in case multiple threads access the same memory section simultaneously:
  - Read-only data is no problem.
  - Updates have to be checked for race conditions.
- It is your responsibility to deal with this situation:
  - In general, one can do the following:
    - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel.
    - Manually create thread private copies of the latter.
    - Use the thread ID to access these private copies.
- Alternative: Use OpenMP’s threadprivate directive!

**The Threadprivate Directive**

- Thread private copies of the designated global variables and common blocks are created.
- Several restrictions and rules apply when doing this:
  - The number of threads has to remain the same for all parallel regions (i.e., no dynamic threads).
  - Some implementations support changing the number of threads.
  - Initial data is undefined, unless copyin is used.
  - Check the specifications when using threadprivate!

**The Copyin Clause**

- Applies to THREADPRIVATE common blocks only.
- At the start of the parallel region, data of the master thread is copied to the thread private copies.

Example:

```plaintext
common /block/velocity
common /fields/xfield, yfield, zfield
!
create thread private common blocks

!$omp threadprivate (/block/, /fields/)!
!$omp parallel!
!$omp default (private) &
!$omp copyin (/block/, /fields/)!
```

**Global Data – The Preferred Solution**

Only add the “threadprivate” directive to the module file; no other changes needed!

**C++ And Threadprivate**

- As of OpenMP 3.0, it has been clarified where/how threadprivate objects are constructed and destructed.
- Allow C++ static class members to be threadprivate.

```plaintext
class T {
public:
    static int i;
    #pragma omp threadprivate(i)
    ...
};
```
A Locking Example

Nested Locking

Simple locks: may not be locked if already in a locked state
Nestable locks: may be locked multiple times by the same thread before being unloaded
In the remainder, we discuss simple locks only
The interface for functions dealing with nested locks is

Simple locks: Nestable locks
omp_init_lock omp_init_nest_lock
omp_destroy_lock omp_destroy_nest_lock
omp_set_lock omp_unset_lock
omp_test_lock omp_test_nest_lock
omp_set_shared_lock omp_unset_shared_lock
omp_set_lock_free omp_set_lock_nest_lock
omp_unset_lock omp_unset_nest_lock
omp_destroy_lock omp_destroy_nest_lock
omp_destroy_lock

Locking Example - The Code

Example Output Using 2 Threads

Example Locking Routines

- Locks provide greater flexibility over critical sections and atomic updates:
  - Possible to implement asynchronous behavior
  - Not block structured
- The so-called lock variable, is a special variable:
  - C/C++: typeomp_lock_t and omp_nest_lock_t for nested locks
  - Fortran: type INTEGER and of a KIND large enough to hold an address
- Lock variables should be manipulated through the API only
- It is illegal, and behavior is undefined, in case a lock variable is used without the appropriate initialization

Protected Region - begin
OMP Parallel - begin
parallel region - begin

parallel region - end
OMP Parallel - end
Protected Region - end

Note: program has been instrumented to get this information
Additional Functionality

OpenMP continues to evolve
- Increasingly flexible to support more algorithms
- Support for modern parallel architectures
  - cc-NUMA and Accelerators
- In recent years, several key features have been added to significantly enhance the functionality
- The following features are discussed next:
  - Thread Cancellation
  - Accelerator Support
  - Tasking
  - Thread Affinity

Focus for today

Thread Cancellation

About Thread Cancellation

- Cancellation – Abort an OpenMP region
  - Implicit or explicit tasks proceed to the end of the canceled region
- Cancellation point – Point where tasks check if cancellation has been requested
  - Take action accordingly (abort or continue)
- Cancellation points are:
  - Implicit barriers, barrier regions, cancel regions, cancellation point regions
Cancellation Directives

- Activate cancellation of the innermost enclosing region (if cancellation has been enabled)
  
  ```
  #pragma omp cancel <construct> [if-clause]
  ```

- Define an explicit cancellation point
  
  ```
  #pragma omp cancellation point <construct>
  ```

Supported constructs:

- parallel
- sections
- for taskgroup

Accelerator Support

Host System

- Memory

Accelerator/GPU

- Memory

The "target" construct is used to execute code on a specific device

```
#pragma omp target device(2)
{
  "execute this code on target device #2"
}
```

- User needs to explicitly manage input and output data using the `map` clause
- This includes "to", "from" and "tofrom" syntax
Tasking In OpenMP

- Tasking was introduced in OpenMP 3.0
- Until then, it required a heroic effort to efficiently implement certain types of parallelism
  - Divide and Conquer algorithms
  - Recursive algorithms
  - Linked lists
  - ...”
- The initial tasking functionality in 3.0 was very simple
  - The idea was to augment tasking as we collectively gain more insight and experience
  - Successive specifications have added functionality

All threads execute this task

Tasking - Who Does What?

- When a thread encounters a task construct, a new explicit task is generated
  - But not necessarily executed yet
- Execution of tasks is handled by the run time system
  - Tasks are assigned to the threads in the current team
  - This is subject to the thread's availability and thus could be immediate, or deferred until later
- Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task

Note: Tasks can be nested (not for the faint of heart)

Tasking - And When?

- Task completion occurs when the end of the structured block associated with the construct that generated the task is reached
- Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs
  - Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits
- A task synchronization construct is a taskwait, taskgroup or a barrier construct
How To Enforce Task Completion?

```
#pragma omp taskwait
!$omp flush taskwait
```

*) This includes direct children only, not descendant tasks. Use the taskgroup construct for the latter.

A Common Tasking Structure

What will this program print?

A Simple Plan

Your Task for Today:

Write a program that prints either "A race car" or "A car race" and maximize the parallelism

Tasking Example/2

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
        printf("\n");
    }
    // End of parallel region
    printf("\n");
    return(0);
}
```

What will this program print using 2 threads?

Tasking Example/3

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note that this program could (for example) also print
"A A race race car car"
or
"A race A car race car",
or
"A race A race car car",
or
.....

But I have not observed this (yet)
OpenMP Overview

Tasking Example/4

```c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  #pragma omp parallel
  #pragma omp single
  {
    printf("A ");
    printf("race ");
    printf("car ");
  } // End of parallel region
  printf("\n");
  return(0);
}
```

What will this program print using 2 threads?

Tasking Example/5

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

But of course now only 1 thread executes .......

Tasking Example/6

```c
int main(int argc, char *argv[]) {
  #pragma omp parallel
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
    printf("race ");
    #pragma omp task
    printf("car ");
  } // End of parallel region
  printf("\n");
  return(0);
}
```

What will this program print using 2 threads?

Tasking Example/7

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A car race
$ ./a.out
A race car
```

Tasks can be executed in arbitrary order

Another Simple Plan

You did well and quickly, so here is a final task to do

Have the sentence end with "is fun to watch" (hint: use a print statement)

Tasking Example/8

```c
int main(int argc, char *argv[]) {
  #pragma omp parallel
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
    printf("race ");
    #pragma omp task
    printf("car ");
  } // End of parallel region
  printf("\n");
  return(0);
}
```

What will this program print using 2 threads?
### OpenMP Overview

#### Getting Up To Speed On OpenMP 4.0

#### Tasking Example/9

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A is fun to watch race car
A is fun to watch race car
A is fun to watch car race
```

Tasks are executed at a task execution point

#### Tasking Example/10

```c
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {
                printf("car ");
                #pragma omp task
                {
                    printf("race ");
                    #pragma omp taskwait
                    printf("is fun to watch ");
                } // End of parallel region
                printf("\n");
            }
        } // End of parallel region
    }
    return(0);
}
```

What will this program print using 2 threads?

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A is fun to watch race car
A is fun to watch race car
A is fun to watch car race
```

### Tasking Example/11

```bash
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A is fun to watch race car
A is fun to watch race car
A is fun to watch car race
```

Tasks are executed first now

### More About Tasking

- Several features have not been covered yet
- These are beyond the scope of this tutorial
  - but it doesn’t mean they’re not useful
- Please check the specifications for much more information on tasking
  - Or attend our tutorial at SC’15!

### Thread Affinity

### Why Worry?
Main Issue: How To Distribute The Data?

Where Does Your Data Go?
The OS decides on the placement of data
A common default is to use the “First Touch” placement policy

Example First Touch Placement/1

Example First Touch Placement/2

Thread Affinity – Machine Model

Why Thread Affinity Support?
For good performance and scalability it is key that Threads are close to the data they need most often

Benefits
Reduces remote memory references
Maximizes bandwidth and reduces latency
Basic Affinity Philosophy

Data is wherever it may be
Threads are moved to the data they need most

Two Key Concepts

The Place List

The Thread Affinity Policy

The Place List

A place consists of a (set of) numbers
Each number represents a scheduling unit
That is, something a thread can run on
For example, a hardware thread

Places – Definition And Notation

Example Multicore System

<table>
<thead>
<tr>
<th>Component</th>
<th>Count</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Cores/socket</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Threads/core</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

Example System Architecture

Cache Coherent System Interconnect
OpenMP Overview

Places - Example

\( \{0,1,2,3\} \) identifies the threads in socket 0, core 0

Easy interval notation: \( \{0:4:1\} \)

- start : count : stride

Places - The Order Does Not Matter

The order of the numbers within a place does not matter

\( \{0,1,2,3\} \) is the same as \( \{3,2,1,0\} \)

Assumption
No preference regarding memory access time

The Place List - Definition

The Place List consists of a comma separated list of places

For example: \( \{0,1,2,3\} , \{8,9,10,11\} \)

The Order Of The Places In The List DOES Matter

\( \{0,1,2,3\} , \{8,9,10,11\} \neq \{8,9,10,11\} , \{0,1,2,3\} \)

The Place List - How To Set It

Environment variable OMP_PLACES is used to define the place list

Example: OMP_PLACES= “\( \{0,1,2,3\} , \{8,9,10,11\}\)”

The interval notation is very convenient

Example: OMP_PLACES= “\( \{0:4:1\} , \{8:4:1\}\)”

The Place List - Abstract Names

Three abstract names are always available:

- sockets , cores , threads

Example: OMP_PLACES=cores

Example: OMP_PLACES=“cores(4)"

Note: Implementation can add names
Example – OMP_PLACES

OMP_PLACES=cores

Equivalent To

OMP_PLACES="{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}"

Equivalent To

OMP_PLACES="{0:4:1},{4:4:1},{8:4:1},{12:4:1}"

Equivalent To

OMP_PLACES = "{0:4:1}:4:4"

The Affinity Policy

The Affinity Policy defines which places to use

This is defined in a symbolic way:

- master, close, or spread
- Each parallel region has such an affinity policy
- Thread binding to a place is implied

Setting The Thread Affinity Policy

Defined through OMP_PROC_BIND

Example: OMP_PROC_BIND="spread,close"

Can also use the "proc_bind" clause

Applies to the current parallel region only

Recap Places And Affinity

The Place List defines what is available
(fixed for the duration of the program)

The Affinity Policy defines thread placement
(can be set for each parallel region)

Example – The Spread Policy

OMP_PLACES=cores
(this means 4 places in the place list)

OMP_PROC_BIND=spread

OMP_NUM_THREADS=4

Result: One OpenMP thread per place
OpenMP Overview

Example Spread Affinity Policy

Cache Coherent System Interconnect

Summary

Summary OpenMP

- Powerful, Easy to Use and Flexible
- Supports Structured and Unstructured Parallelism
- Can be used on any shared memory system
  - Including cc-NUMA Architectures and Accelerators
- Compilers with OpenMP support widely available

OpenMP Continues To Evolve!

Thank You And ..... Stay Tuned!

ruud.vanderpas@oracle.com

OpenMP Overview

OpenMP Overview

OpenMP Overview

OpenMP Overview