An Overview of OpenMP

Ruud van der Pas
Senior Staff Engineer
Technical Developer Tools
Sun Microsystems, Menlo Park, CA, USA

Nanyang Technological University
Singapore
Wednesday January 14, 2009
Outline

- A Guided Tour of OpenMP
- Case Study
- Wrap-Up
An Overview of OpenMP

http://www.openmp.org

http://www.compunity.org
The OpenMP API specification for parallel programming

OpenMP News

» Christian's First Experiments with Tasking in OpenMP 3.0

From Christian Terboven's blog:

OpenMP 3.0 is out, maybe a bit later than we hoped for, but I think that we got a solid standard document. At WCMP 2008 a couple of weeks ago, there was an OpenMP tutorial which included a talk by Alex Duran (from UPC in Barcelona, Spain) on what is new in OpenMP 3.0 - which is really worth a look! My talk was on some OpenMP application experiences, including a case study on Windows, and I really think that many of our codes can profit from Tasks. Motivated by Alex' talk I tried the updated Nanos compiler and prepared a couple of examples for my lectures on Parallel Programming in Maastricht and Aachen. In this post I am walking through the simplest one. Computing the Fibonacci number in parallel.

Read more...
Posted on June 6, 2008

» New Forum Created

The OpenMP 3.0 API Specifications forum is now open for discussing the specs document itself.
Posted on May 31, 2008

» New Links

New links and information have been added to the OpenMP Compilers and the OpenMP Resources pages.
Posted on May 23, 2008

» Recent Forum Posts

» strange behavior of C function strcmp() With OPENMP
» virtual destructor not called with first private clause
» recursive function (parallel pragma)
Shameless Plug - “Using OpenMP”

“Using OpenMP”
Portable Shared Memory Parallel Programming

Chapman, Jost, van der Pas

MIT Press, October 2007


List price: 35 $US

All examples available soon!
(also plan to start a forum on www.openmp.org)
What is OpenMP?

- **De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran**

- **Consists of:**
  - Compiler directives
  - Run time routines
  - Environment variables

- Specification maintained by the OpenMP Architecture Review Board ([http://www.openmp.org](http://www.openmp.org))

- Version 3.0 has been released May 2008
When to consider OpenMP?

- The compiler may not be able to do the parallelization in the way you like to see it:
  - It can not find the parallelism
    - The data dependence analysis is not able to determine whether it is safe to parallelize or not
  - The granularity is not high enough
    - The compiler lacks information to parallelize at the highest possible level
- This is when explicit parallelization through OpenMP directives comes into the picture
Advantages of OpenMP

- Good performance and scalability
  - If you do it right ....
- De-facto and mature standard
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
OpenMP is ideally suited for multicore architectures

Memory and threading model map naturally

Lightweight

Mature

Widely available and used
The OpenMP Memory Model

- All threads have access to the same, *globally shared*, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit
Data-Sharing Attributes

- **In an OpenMP program, data needs to be “labelled”**

- **Essentially there are two basic types:**
  
  - **Shared**
    
    ✓ There is only instance of the data
    
    ✓ All threads can read and write the data simultaneously, unless protected through a specific OpenMP 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 OpenMP Execution Model

Fork and Join Model

Master Thread

Parallel region

Worker Threads

Synchronization

Parallel region

Worker Threads

Synchronization
A first OpenMP example

For-loop with independent iterations

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

For-loop parallelized using an OpenMP pragma

```c
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 5
% a.out
Example parallel execution

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0-199</td>
<td>i=200-399</td>
<td>i=400-599</td>
<td>i=600-799</td>
<td>i=800-999</td>
</tr>
<tr>
<td>a[i]</td>
<td>a[i]</td>
<td>a[i]</td>
<td>a[i]</td>
<td>a[i]</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>b[i]</td>
<td>b[i]</td>
<td>b[i]</td>
<td>b[i]</td>
<td>b[i]</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>c[i]</td>
<td>c[i]</td>
<td>c[i]</td>
<td>c[i]</td>
<td>c[i]</td>
</tr>
</tbody>
</table>
Components of OpenMP 2.5

Directives

- Parallel region
- Worksharing
- Synchronization
- Data-sharing attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Wallclock timer
- Locking

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
Example - Matrix times vector

```c
#pragma omp parallel for default(none) \
private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```

**TID = 0**

```
for (i=0,1,2,3,4)

i = 0
sum = \sum_{j} b[i=0][j]*c[j]
    a[0] = sum

i = 1
sum = \sum_{j} b[i=1][j]*c[j]
    a[1] = sum

... etc ...
```

**TID = 1**

```
for (i=5,6,7,8,9)

i = 5
sum = \sum_{j} b[i=5][j]*c[j]
    a[5] = sum

i = 6
sum = \sum_{j} b[i=6][j]*c[j]
    a[6] = sum
```

... etc ...
OpenMP performance

*) With the IF-clause in OpenMP this performance degradation can be avoided
A more elaborate example

```c
#pragma omp parallel if (n>limit) default(none) \ 
    shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];

    #pragma omp for nowait
    for (i=0; i<n; i++)
        a[i] = b[i] + c[i];

    #pragma omp barrier

    ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
    ....
} /*-- End of parallel region --*/
```
OpenMP In Some More Detail
Terminology and behavior

- **OpenMP Team** := Master + Workers

- A **Parallel Region** is a block of code executed by all threads simultaneously
  
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially

- A **work-sharing construct** divides the execution of the enclosed code region among the members of the team; in other words: they split the work
The if/private/shared clauses

if (scalar expression)

✔ Only execute in parallel if expression evaluates to true
✔ Otherwise, execute serially

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

```c
#pragma omp parallel if (n > threshold) \ 
    shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
    x[i] += y[i];
}
```
Suppose we run each of these two loops in parallel over i:

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

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

This may give us a wrong answer (one day)

Why?
We need to have updated all of a[] first, before using a[] *

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

wait!

```c
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
Barrier Region

idle
idle
idle

Barrier syntax in OpenMP:

#pragma omp barrier

!$omp barrier
The nowait clause

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

```
#pragma omp for nowait
{
    :
}

!$omp do : :
!$omp end do nowait
```
A parallel region is a block of code executed by multiple threads simultaneously

```c
!$omp parallel [clause[[,[,] clause] ...]
   "this is executed in parallel"
!$omp end parallel (implied barrier)
```

```c
#pragma omp parallel [clause[[,[,] clause] ...]
{
   "this is executed in parallel"
}
(implied barrier)
```
Work-sharing constructs

The OpenMP work-sharing constructs

```c
#pragma omp for
{
    ....
}
!$OMP DO
    ....
!$OMP END DO
```

```c
#pragma omp sections
{
    ....
}
!$OMP SECTIONS
    ....
!$OMP END SECTIONS
```

```c
#pragma omp single
{
    ....
}
!$OMP SINGLE
    ....
!$OMP END SINGLE
```

- 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 nowait is specified)
- A work-sharing construct does not launch any new threads
Fortran has a fourth worksharing construct:

```fortran
!$OMP WORKSHARE

<array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

Example:

```fortran
!$OMP WORKSHARE

A(1:M) = A(1:M) + B(1:M)

!$OMP END WORKSHARE NOWAIT
```
The omp for/do directive

The iterations of the loop are distributed over the threads

```c
#pragma omp for [clause[[], clause] ...]
<original for-loop>
```

```c
 !$omp do [clause[[], clause] ...]
<original do-loop>
 !$omp end do [nowait]
```

Clauses supported:

- private
- firstprivate
- lastprivate
- reduction
- ordered*
- schedule
- nowait

*) Required if ordered sections are in the dynamic extent of this construct
The omp for directive - Example

```c
#pragma omp parallel default(none)\ 
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
}
/*-- End of parallel region --*/

(implied barrier)
```
The sections directive

The individual code blocks are distributed over the threads

```c
#pragma omp sections [clause(s)]
{
#pragma omp section
  <code block1>
#pragma omp section
  <code block2>
#pragma omp section
  :
}
```

Clauses supported:
- private
- firstprivate
- lastprivate
- reduction
- nowait

Note: The SECTION directive must be within the lexical extent of the SECTIONS/END SECTIONS pair
The sections directive - Example

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

        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

        #pragma omp section

        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

} /*-- End of parallel region --*/
```
#pragma omp parallel
#pragma omp for
for (....)

Single PARALLEL loop

 !$omp parallel
 !$omp do
 ...
 !$omp end do
 !$omp end parallel

Single WORKSHARE loop

 !$omp parallel workshare
...
 !$omp end workshare
 !$omp end parallel

#pragma omp parallel
#pragma omp sections
{
 ...
}

Single PARALLEL sections

 !$omp parallel
 !$omp sections
 ...
 !$omp end sections
 !$omp end parallel

#pragma omp parallel sections
{
 ...
}
This construct is ideally suited for I/O or initializations

Original Code

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

"declare A to be be shared"

```
#pragma omp parallel
{
.....
    one volunteer requested
    "read a[0..N-1];"

    thanks, we're done
}
.....
```

Parallel Version

May have to insert a barrier here
Usually, there is a barrier at the end of the region

Might therefore be a scalability bottleneck (Amdahl's law)
**SINGLE and MASTER construct**

Only one thread in the team executes the code enclosed

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

```c
#pragma omp end single [copyprivate][nowait]
```

Only the **master thread** executes the code block:

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

```c
#pragma omp end master
```

There is no implied barrier on entry or exit!
Critical Region/1

If sum is a shared variable, this loop can not run in parallel

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

We can use a critical region for this:

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

---

one at a time can proceed

next in line, please
Critical Region/2

- Useful to avoid a race condition, or to perform I/O (but that still has random order)
- Be aware that there is a cost associated with a critical region
Critical and Atomic constructs

Critical: All threads execute the code, but only one at a time:

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

There is no implied barrier on entry or exit!

```c
#pragma omp critical [(name)]
<code-block>
#pragma omp end critical [(name)]
```

Atomic: only the loads and store are atomic ....

```c
#pragma omp atomic
<statement>
```

This is a lightweight, special form of a critical section

```c
#pragma omp atomic
a[indx[i]] += b[i];
```
Why The Excitement About OpenMP 3.0?

Support for TASKS!

With this new feature, a wider range of applications can now be parallelized.
Example - A Linked List

```
while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
```

Hard to do before OpenMP 3.0:
First count number of iterations, then convert while loop to for loop
Example - A Linked List With Tasking

```c
my_pointer = listhead;

#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

OpenMP Task is specified here (executed in parallel)
Case Study
A Neural Network
Neural Network application*

Performance Analyzer Output

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>120.710 100.0</td>
<td>120.710 100.0</td>
<td>128.310</td>
<td>&lt;Total&gt;</td>
</tr>
<tr>
<td>116.960 96.9</td>
<td>116.960 96.9</td>
<td>122.610</td>
<td>calc_r_loop_on_neighbours</td>
</tr>
<tr>
<td>0.900 0.7</td>
<td>118.630</td>
<td>0.920</td>
<td>calc_r</td>
</tr>
<tr>
<td>0.590 0.5</td>
<td>1.380</td>
<td>0.590</td>
<td>_doprnt</td>
</tr>
<tr>
<td>0.410 0.3</td>
<td>1.030</td>
<td>0.430</td>
<td>init_visual_input_on_V1</td>
</tr>
<tr>
<td>0.280 0.2</td>
<td>0.280</td>
<td>1.900</td>
<td>_write</td>
</tr>
<tr>
<td>0.200 0.2</td>
<td>0.200</td>
<td>0.200</td>
<td>round_coord_cyclic</td>
</tr>
<tr>
<td>0.130 0.1</td>
<td>0.130</td>
<td>0.140</td>
<td>__arint_set_n</td>
</tr>
<tr>
<td>0.130 0.1</td>
<td>0.550</td>
<td>0.140</td>
<td>__k_double_to_decimal</td>
</tr>
<tr>
<td>0.090 0.1</td>
<td>1.180</td>
<td>0.090</td>
<td>fprintf</td>
</tr>
</tbody>
</table>

Callers-callees fragment:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>116.960</td>
<td>0.900</td>
<td>118.630</td>
<td>calc_r</td>
</tr>
<tr>
<td>116.960</td>
<td>116.960</td>
<td>116.960</td>
<td>*calc_r_loop_on_neighbours</td>
</tr>
</tbody>
</table>

*) Program was said not to scale on a Sun SMP system....
struct cell{
    double x; double y; double r; double I;
};

......

struct cell V1[NPOSITIONS_Y][NPOSITIONS_X];
double       h[NPOSITIONS][NPOSITIONS];
......

Excl. User CPU  Excl. Wall
sec.      %      sec.
1040. void
1041. calc_r_loop_on_neighbours
      (int y1, int x1)
      
      0.080   0.1     0.080
1042. {
1043. struct interaction_structure *next_p;
1044. 
1045. for (next_p = JJ[y1][x1].next;
1046.      next_p != NULL;
1047.      next_p = next_p->next) {
1048.    h[y1][x1] += next_p->strength * 
1049.          V1[next_p->y][next_p->x].r;
1052.   }
Data structure problem

- We only use 1/4 of a cache line!

- For sufficiently large problems this will:
  - Generate additional memory traffic
    - Higher interconnect pressure
  - Waste data cache capacity
    - Reduces temporal locality

- The above negatively affects both serial and parallel performance

- Fix: split the structure into two parts
  - One contains the "r" values only
  - The other one contains the \{x,y,I\} sets
double V1_R[NPOSITIONS_Y][NPOSITIONS_X];

void calc_r_loop_on_neighbours(int y1, int x1)
{
    struct interaction_structure *next_p;
    double sum = h[y1][x1];

    for (next_p = JJ[y1][x1].next;
        next_p != NULL;
        next_p = next_p->next) {
        sum += next_p->strength * V1_R[next_p->y][next_p->x];
    }
    h[y1][x1] = sum;
}
Parallelization with OpenMP

```c
void calc_r(int t)
{
#include <omp.h>

#pragma omp parallel for default(none) \ 
    private(y1,x1) shared(h,V1,g,T,beta_inv,beta)
for (y1 = 0; y1 < NPOSITIONS_Y; y1++) {
    for (x1 = 0; x1 < NPOSITIONS_X; x1++) {
        calc_r_loop_on_neighbours(y1,x1);
        h[y1][x1] += V1[y1][x1].I;
        <statements deleted>
    }
}
/*-- End of OpenMP parallel for --*/
```

Can be executed in parallel.
Scalability results

\[ T(\text{One proc})/T(\text{P procs}) \]

\[ T(\text{baseline})/T(\text{modified}) \]

Note:
Single processor run time is 5001 seconds for the baseline version (4847 for the modified version)
That's It

Thank You and ..... Stay Tuned!

Ruud van der Pas
ruud.vanderpas@sun.com