OpenMP Tutorial
CScADs Workshop 2009

Lei Huang
Barbara Chapman
University of Houston

Acknowledgements: Tim Mattson (Intel), Rud van der Paas (Sun), OpenMP ARB
Talk Outline

- Parallel Computing, threads, and OpenMP
- The important constructs of OpenMP
- OpenMP Practices for optimizations
- Hybrid MPI/OpenMP Applications
- Case Studies and Examples
OpenMP News

Christian’s First Experiments with Tasking in OpenMP 3.0

From Christian Terbrock’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 IWOMP 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 Nomes 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
• enabling serialization for std::map

OpenMP Overview:

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

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 moderate programming effort
- Allows the program to be parallelized incrementally
How Does OpenMP Enable Us to Exploit Threads?

- OpenMP provides thread programming model at a “high level”.
  - The user does not need to specify all the details
    - Assignment of work to threads
    - Creation of threads
- User makes strategic decisions
- Compiler figures out details
  - Compiler flags enable OpenMP (e.g. -openmp, -xopenmp, -fopenmp, -mp)
provides the means to:

- create and destroy threads
- assign / distribute work to threads
- specify which data is shared and which is private to a thread
- coordinate actions of threads on shared data
OpenMP Overview: How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.

- Synchronization protects data conflicts.
  - Synchronization is expensive.
  - Change how data is accessed to minimize the need for synchronization.
Fork-Join Parallelism:

Master thread spawns a team of threads as needed.

Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.
OpenMP Parallel Computing Solution Stack

User layer
- End User
- Application

Prog. Layer (OpenMP API)
- Directives, Compiler
- OpenMP library
- Environment variables

System layer
- Runtime library
- OS/system support for shared memory.
Most of the constructs in OpenMP are compiler directives or pragmas.

- For C and C++, the pragmas take the form:
  ```
  #pragma omp construct [clause [clause]...] 
  ```

- For Fortran, the directives take one of the forms:
  - Fixed form
    ```
    !$OMP construct [clause [clause]...] 
    C$OMP construct [clause [clause]...] 
    ```
  - Free form (but works for fixed form too)
    ```
    !$OMP construct [clause [clause]...] 
    ```

Include file and the OpenMP lib module

```
#include <omp.h>
use omp_lib
```
Introduction to OpenMP

Amitava Datta

University of Western Australia
Compiling OpenMP programs

- OpenMP programs written in C are compiled by (for example): gcc -fopenmp -o prog1 prog1.c
- We have assumed the name of the C file is prog1.c and the name of the executable is prog1
- The compiler will look for OpenMP directives in your program for generating code.
- No action is taken if there are no OpenMP directives in your program.
pragma directive

If you want the compiler to generate code using OpenMP, you have to use the pragma directive

```c
#include<stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        printf("The parallel region is executed by thread %d\n",omp_get_thread_num());
    }
}
```
When the compiler encounters the `parallel` directive, it generates multi-threaded code.

How many threads will execute the code will depend on how many threads are specified (more later).

The default is number of threads equal to number of cores.

The parallel region is executed by thread 4
The parallel region is executed by thread 3
The parallel region is executed by thread 7
The parallel region is executed by thread 2
The parallel region is executed by thread 5
The parallel region is executed by thread 1
The parallel region is executed by thread 6
The parallel region is executed by thread 0

But I have only 4 cores in my machine.
Hyperthreading is an Intel technology that treats each physical core as two logical cores.

Two threads are executed at the same time (logically) on the same core.

Processors (or cores) do not execute instructions in every clock cycle.

There is an opportunity to execute another instruction from another thread when the core is idle.

Hyperthreading schedules two threads to every core.

So, my processor has 4 physical cores and 8 logical cores.
Hyperthreading

- The purpose of hyperthreading is to improve the throughput (processing more per unit time).
- This may or may not happen. In fact hyperthreading may actually have slower performance.
- Your process will run slower when hyperthreading is turned on.
- It all depends on how well the L1 cache is shared.
- It is possible to turn hyperthreading off through the BIOS (more on lab sheet).
Threads run independently

- There is only one thread until the parallel directive is encountered.
- 7 other threads are launched at that point.
- Thread 0 is usually the master thread (that spawns the other threads.
- The parallel region is enclosed in curly brackets.
- There is an implied barrier at the end of the parallel region.
What is a barrier?

- A barrier is a place in the process where all threads must reach before further processing occurs.
- Threads may run away without barriers and it is necessary many times to have barriers at different places in a process.
- Barriers are sometime implicit (like here), barriers sometime can be removed (more later).
- Barriers are expensive in terms of run time performance. A typical barrier may take hundreds of clock cycles to ensure that all threads have reached the barrier.
- It is better to remove barriers, but this is fraught with danger.
A variation of our code

```c
#include<stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        if (omp_get_thread_num()==3) sleep(1);
        printf("The parallel region is executed by thread \%d\n",omp_get_thread_num());
    }
}
```
The parallel region is executed by thread 4
The parallel region is executed by thread 7
The parallel region is executed by thread 1
The parallel region is executed by thread 2
The parallel region is executed by thread 5
The parallel region is executed by thread 6
The parallel region is executed by thread 0
The parallel region is executed by thread 3

- Thread 3 is now suspended for 1 second, so all other threads complete before thread 3.
Basic Concepts in Parallelization

Ruud van der Pas

Senior Staff Engineer
Oracle Solaris Studio
Oracle
Menlo Park, CA, USA

IWOMP 2010
CCS, University of Tsukuba
Tsukuba, Japan
June 14-16, 2010
Outline

- Introduction
- Parallel Architectures
- Parallel Programming Models
- Data Races
- Summary
Introduction
Why Parallelization?

Parallelization is another optimization technique. The goal is to reduce the execution time.

To this end, multiple processors, or cores, are used.

Using 4 cores, the execution time is 1/4 of the single core time.
What Is Parallelization?

"Something" is parallel if there is a certain level of independence in the order of operations.

In other words, it doesn't matter in what order those operations are performed.

- A sequence of machine instructions
- A collection of program statements
- An algorithm
- The problem you're trying to solve
What is a Thread?

- Loosely said, a thread consists of a series of instructions with its own program counter ("PC") and state.
- A parallel program executes threads in parallel.
- These threads are then scheduled onto processors.

Diagram:
- Thread 0
- Thread 1
- Thread 2
- Thread 3
- PC arrows point to the program counter for each thread.
Parallel overhead

- *The total CPU time often exceeds the serial CPU time:* 
  - The newly introduced parallel portions in your program need to be executed
  - Threads need time sending data to each other and synchronizing (“communication”)
    - Often the key contributor, spoiling all the fun
- Typically, things also get worse when increasing the number of threads
- Efficient parallelization is about minimizing the communication overhead
Communication

- Embarrassingly parallel: 4x faster
- Wallclock time is $\frac{1}{4}$ of serial wallclock time

- Additional communication
- Less than 4x faster
- Consumes additional resources
- Wallclock time is more than $\frac{1}{4}$ of serial wallclock time
- Total CPU time increases
Load balancing

**Perfect Load Balancing**
- All threads finish in the same amount of time
- No threads is idle

**Load Imbalance**
- Different threads need a different amount of time to finish their task
- Total wall clock time increases
- Program does not scale well
About scalability

- Define the speed-up $S(P)$ as $S(P) := T(1)/T(P)$
- The efficiency $E(P)$ is defined as $E(P) := S(P)/P$
- In the ideal case, $S(P) = P$ and $E(P) = P/P = 1 = 100\%$
- Unless the application is embarrassingly parallel, $S(P)$ eventually starts to deviate from the ideal curve
- Past this point $P_{opt}$, the application sees less and less benefit from adding processors
- Note that both metrics give no information on the actual run-time
- As such, they can be dangerous to use

In some cases, $S(P)$ exceeds $P$

This is called "superlinear" behaviour

Don't count on this to happen though
Amdahl's Law

Assume our program has a parallel fraction “f”

This implies the execution time \( T(1) := f \times T(1) + (1-f) \times T(1) \)

On P processors: \( T(P) = \frac{f}{P} \times T(1) + (1-f) \times T(1) \)

Amdahl's law:

\[
S(P) = \frac{T(1)}{T(P)} = \frac{1}{\frac{f}{P} + 1-f}
\]

Comments:

- This "law' describes the effect the non-parallelizable part of a program has on scalability
- Note that the additional overhead caused by parallelization and speed-up because of cache effects are not taken into account
**Amdahl's Law**

- **It is easy to scale on a small number of processors**
- **Scalable performance however requires a high degree of parallelization i.e. f is very close to 1**
- **This implies that you need to parallelize that part of the code where the majority of the time is spent**

![Graph showing Amdahl's Law](image-url)
Amdahl's Law in practice

We can estimate the parallel fraction “f”

Recall: \( T(P) = \frac{f}{P}T(1) + (1-f)T(1) \)

It is trivial to solve this equation for “f”:

\[
f = \frac{1 - \frac{T(P)}{T(1)}}{1 - \frac{1}{P}}
\]

Example:

\( T(1) = 100 \) and \( T(4) = 37 \) \( \Rightarrow \) \( S(4) = \frac{T(1)}{T(4)} = 2.70 \)

\( f = \frac{1-37/100}{1-(1/4)} = 0.63/0.75 = 0.84 = 84\% \)

Estimated performance on 8 processors is then:

\( T(8) = \frac{0.84}{8} \times 100 + (1-0.84) \times 100 = 26.5 \)

\( S(8) = \frac{T}{T(8)} = 3.78 \)
Threads Are Getting Cheap .....  

Elapsed time and speed up for $f = 84\%$

Using 8 cores:  
$100/26.5 = 3.8x$ faster

- Elapsed time
- Speed up
Numerical results

Consider:
\[ A = B + C + D + E \]

Serial Processing

\[
\begin{align*}
A &= B + C \\
A &= A + D \\
A &= A + E
\end{align*}
\]

Parallel Processing

\[
\begin{align*}
T_1 &= B + C \\
T_1 &= T_1 + T_2 \\
T_2 &= D + E
\end{align*}
\]

☞ The roundoff behaviour is different and so the numerical results may be different too.
☞ This is natural for parallel programs, but it may be hard to differentiate it from an ordinary bug ....
OpenMP Parallel Regions: Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

```c
#pragma omp parallel
{
    id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```fortran
C$OMP PARALLEL
10    wrk(id) = garbage(id)
     res(id) = wrk(id)**2
     if(.not.conv(res(id))) goto 10
C$OMP END PARALLEL
```

```c
#pragma omp parallel for
for(l=0;l<N;l++){
    res[l] = big_calc(l);
}
```

```c
#pragma omp parallel for
for(I=0;I<N;I++){
    res[I] = big_calc(I);
    A[I] = B[I] + res[I];
}
```
OpenMP constructs can span multiple source files.

**Scope of OpenMP constructs:**

Orphan directives can appear outside a parallel region.

Dynamic extent of parallel region includes lexical extent.

**foo.f**

```fortran
C$OMP PARALLEL
  call whoami
C$OMP END PARALLEL
```

**bar.f**

```fortran
subroutine whoami
  external omp_get_thread_num
  integer iam, omp_get_thread_num
  iam = omp_get_thread_num()
  C$OMP CRITICAL
  print*, 'Hello from ', iam
  C$OMP END CRITICAL
  return
end
```
The “for” Work-Sharing construct splits up loop iterations among the threads in a team.

```c
#pragma omp parallel
#pragma omp for
    for (i=0; i<N; i++) {
        work(i);
    }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

```c
#pragma omp for nowait
```

“nowait” is useful between two consecutive, independent omp for loops.
for(i=0; i<N; i++)   { a[i] = a[i] + b[i];}

#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart; i<iend; i++)   { a[i] = a[i] + b[i];}
}

#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++)   { a[i] = a[i] + b[i];}

#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++)   { a[i] = a[i] + b[i];}
### Schedule Clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>

**Least work at runtime:** scheduling done at compile-time

**Most work at runtime:** complex scheduling logic used at run-time
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.

- `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.
Additional Schedule in OpenMP 3.0

- **Auto**
  - The compiler *(or runtime system)* decides what is best to use
  - Choice could be implementation dependent
Loop Collapsing (in OpenMP 3.0)

!$omp parallel do collapse(2)
do  i=1,n
   do  j=1,n
       ..... 
   end do
end do
The **Sections** work-sharing 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 a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
OpenMP Master

Work-Sharing Constructs

- The `master` construct denotes a structured block executed by the master thread. The other threads just skip it (no synchronization is implied).

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
#pragma omp master
    { exchange_boundaries(); }
#pragma barrier
    do_many_other_things();
}
The `single` construct denotes a block of code that is executed by only one thread. A barrier is implied at the end of the single block.

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```