OpenMP examples

Sergeev Efim
Senior software engineer
Singularis Lab, Ltd.
OpenMP Is:

• An Application Program Interface (API) that may be used to explicitly direct \textit{multi-threaded, shared memory} parallelism.

• Comprised of three primary API components:
  – Compiler Directives
  – Runtime Library Routines
  – Environment Variables

• An abbreviation for: \textit{Open Multi-Processing}
OpenMP Is Not:

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- Designed to handle parallel I/O. The programmer is responsible for synchronizing input and output.
Shared Memory Model:

- OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.

Uniform Memory Access

Non-Uniform Memory Access
Thread Based Parallelism

• OpenMP programs accomplish parallelism exclusively through the use of threads.
• A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
• Threads exist within the resources of a single process. Without the process, they cease to exist.
• Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
Explicit Parallelism

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

- Parallelization can be as simple as taking a serial program and inserting compiler directives...

- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.
Fork - Join Model

• All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.

• **FORK**: the master thread then creates a team of parallel **threads**.

• The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.

• **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
Three Components

• OpenMP compiler directives are used for various purposes:
  – `#pragma omp parallel shared(x) private(beta, pi)`

• Run-time Library Routines
  – `#include <omp.h>`
    `int omp_get_num_threads(void)`

• Environment Variables
  – `export OMP_NUM_THREADS=8`
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code . . .
  Beginning of parallel region.
  Fork a team of threads. Specify variable scoping
  #pragma omp parallel private(var1, var2) shared(var3)
  { Parallel region executed by all threads .
    Other OpenMP directives .
    Run-time Library calls .
    All threads join master thread and disband
  }
  Resume serial code . . .
}
Private vs. Shared

- In parallel region, default behavior is that all variables are **Shared**
  - All threads read and write the same memory location for each variable
  - This is ok if threads accessing different elements of an array
  - Problem if threads write same scalar or array element

- **Private** clause creates separate memory location for specified variable for each thread

```c
ifirst = 10;
#pragma omp parallel for private(i2)
for(i = 1; i <= imax; i++){
    i2 = 2*i;
    j[i] = ifirst + i2;
}
```
```c
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
{
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
```
How is OpenMP Typically Used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops
  - Split them up between threads

```c
void main()
{
    double Res[1000];

    for(int i =0; i<1000;i++) {
        Do_huge_comp(Res[i])
    }
}
```

```c
void main()
{
    double Res[1000];

    #pragma omp parallel for
    for(int i =0; i<1000;i++) {
        Do_huge_comp(Res[i])
    }
}
```

Sequential program

Parallel program
Work-sharing Constructs: for

- The “for” Work-Sharing constructs splits up loop iterations among the threads in a team

```cpp
#include <cmath>

int main()
{
    const int size = 256;
    double sinTable[size];

    #pragma omp parallel for
    for(int n=0; n<size; ++n)
        sinTable[n] = std::sin(2 * M_PI * n / size);

    // the table is now initialized
}
```
Exercise: Parallelizing Matrix Multiplication using OpenMP

- Allocate Xeon Phi
  - `salloc -N 1 -partition=mic`
- Change directory
  - `cd ./School2016/openmp`
- Check source file `matrix.cpp`
  - `icpc -mmic matrix.cpp`
- **Add OpenMP to source file**
  - `icpc -mmic -openmp matrix.cpp`
The schedule Clause

- The `schedule` clause affects how loop iterations are mapped onto threads

`schedule(static [,chunk])`
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

`schedule(dynamic [,chunk])`
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

`schedule(guided [,chunk])`
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”

`schedule (auto)`
`schedule (runtime)`
"schedule" Clause Example

```c
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i) ) gPrimesFound++;
    }
```

- Iterations are divided into chunks of 8

- If start = 3, then first chunk is

  \[ i=\{3,5,7,9,11,13,15,17\}\]
OpenMP Reduction Clause

```
reduction (op : list)
```

- The variables in “list” must be shared in the enclosing parallel region

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
OpenMP Reduction Example: Numerical Integration

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

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1. + x*x);
        }
    }

    double my_pi = sum*step;
    printf("Value of integral is: %f\n", my_pi);

    return 0;
}
```
A range of associative operands can be used with reduction
Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Credit: IOMPP
Exercise: Schedule & Reduction

- Check source file `schedule.cpp`
- Add required OpenMP for loops
- Add required schedule clauses
- Adjust chunk
- Build & Run on Xeon Phi

```c
schedule(kind [, chunk size])
```

```c
reduction (op : list)
```
**Work-Sharing constructs: Sections**

- **SECTIONS** - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

```c
a = computeA();
b = computeB();
c = computeC(a, b);
d = computeD();
printf("%6.2f\n", computeE(d, c));
```

A, B, D can be computed in parallel.
Work-Sharing constructs: Sections

- **#pragma omp sections**
- Must be inside a parallel region
- Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
- Encompasses each omp section

- **#pragma omp section**
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads
Work-Sharing constructs: Sections

```c
#pragma omp parallel sections
{
#pragma omp section
    double a = computeA();
#pragma omp section
    double b = computeB();
#pragma omp section
    double d = computeD();
}

double c = computeC(a, b);
printf("%6.2f\n", computeE(s, c));
```
Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```
Exercise: Sections

- Check source file `sections.cpp`
- Add required OpenMP section and sections
- Add required shared clauses
- Build & Run on Xeon Phi

```c
#pragma omp [parallel] sections [clauses] { #pragma omp section { code_block } }
```
Nested Parallelism

Parallel region

Nested Parallel region
Sample Nested OpenMP code

```c
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team: %d\n", level, omp_get_num_threads());
    }
}
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

Check source file: nested_sample.c

```
# icc nested_sample.c -openmp
```

```
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
% export OMP_NESTED=TRUE
% a.out
```
Sample Nested OpenMP code

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

void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team: \n", level, omp_get_num_threads());
    }
}

int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 3: number of threads in the team: 2

% export OMP_NESTED=TRUE
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2

Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P7
Nested API & Environment

- Runtime library API
  - `omp_set_nested(int val)`

- Environment variable
  - `omp_nested=true/false`
The single and master constructs

```c
#pragma omp parallel
{
    /*This code execute by all threads */
#pragma omp master
{
    /*This code execute by thread 0*/
}
/*This code execute by all threads */
}
```
The single and master constructs

```c
#pragma omp parallel
{
    /*This code executed by all threads */
    #pragma omp single
    {
        /*This code executed by only one thread
        It is unspecified which thread.*/
    }
    /*This code executed by all threads */
}
```
Exercise: Sections & Nested Parallelism

- Check source file *sections_nested.cpp*
- Add required OpenMP directives to make nested sections
- Add required shared and private clauses
- Build & Run on Xeon Phi

```c
#pragma omp [parallel] sections [clauses] { #pragma omp section { code_block } }
```
Synchronization Constructs

• `#pragma omp critical`

• `#pragma omp atomc`

• `#pragma omp ordered`

• `#pragma omp barrier`
Thank you!

Efim Sergeev

- efim.sergeev@singularis-lab.com

https://www.singularis-lab.com/
https://www.linkedin.com/company/singularis-lab-llc
http://habrahabr.ru/company/singularis
http://vk.com/singularis_lab