OpenMP 4.0/4.5: New Features and Protocols

Jemmy Hu

SHARCNET HPC Consultant
University of Waterloo

May 10, 2017

General Interest Seminar
Outline

- OpenMP overview
- Task constructs in OpenMP
- SIMP constructs in OpenMP
- Device model in OpenMP
- References
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++

§ Ease of Use: Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach

§ Standardizes established SMP practice + vectorization and heterogeneous device programming
OpenMP: Fork-Join Model

- OpenMP uses the fork-join model of parallel execution:

  **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.
OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is basically the same between Fortran and C/C++
Types of Work-Sharing Constructs:

**DO / for** - shares iterations of a loop across the team. Represents a type of "data parallelism".

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

**SINGLE** - serializes a section of code.
## A motivating example

### Sequential code

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

### OpenMP parallel region

```c
#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]; }
}
```

### OpenMP parallel region and a work-sharing for construct

```c
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++) { a[i] = a[i] + b[i]; }
```
Example: Calculating $\pi$

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2} \, dx = \pi$$

- **Discretization:**

$\Delta = 1/N$: \textbf{step} = 1/NBIN  
$x_i = (i+0.5)\Delta$ (i = 0,...,N-1)

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \approx \pi$$

```c
#include <stdio.h>
define NBIN 100000
void main() {
    int i; double step, x, sum=0.0, pi;
    step = 1.0/NBIN;
    for (i=0; i<NBIN; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum*step;
    printf("PI = %f\n", pi);
}
```
OpenMP: Pi with a loop and a reduction

#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
    }
Create a team of threads …
without a parallel construct, you’ll never have more than one thread

Break up loop iterations and assign them to threads … setting up a reduction into sum.
program compute
……
write(*,*) "start"
!$omp parallel
select case (omp_get_thread_num())
case (0)
do i = 1, NX
   ri = real(i)
x(i) = atan(ri)/ri
end do
end select
endif
end parallel
case (1)
do j = 1, NY
   rj = real(j)
y(j) = cos(rj)/rj
end do
endif
end parallel
case (2)
do k = 1, NZ
   rk = real(k)
z(k) = log10(rk)/rk
end do
endif
end parallel
write(*,*) "end"
end program
Not all programs have simple loops OpenMP can parallelize

• Consider a program to traverse a linked list:

```c
p=head;
while (p) {
    processwork(p);
    p = p->next;
}
```

• OpenMP can only parallelize loops in a basic standard form with loop counts known at runtime
Example: Fibonacci numbers

```c
int fib (int n)
{
    int x,y;
    if (n < 2) return n;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
}

int main()
{
    int NW = 1000;
    fib(NW);
}
```

- \( F_n = F_{n-1} + F_{n-2} \)
- Inefficient O(n^2) recursive implementation!
What are tasks?

• Tasks are independent units of work

• Tasks are composed of:
  – code to execute
  – data to compute with

• Threads are assigned to perform the work of each task.
  – The thread that encounters the task construct may execute the task immediately.
  – The threads may defer execution until later
Task constructs in OpenMP

• The task construct was added to support irregular programs:
  – While loops or loops whose iteration limits are not known at
    compiler time.
  – Recursive algorithms
  – divide and conquer problems.

• The task construct has expanded over the years with new features to
  support irregular problems with tasks in each new release of OpenMP

#pragma omp task
  - Creates a new task, Task added to task queue
  - Available thread picks next task from queue to execute

#pragma omp taskwait
  - Acts like barrier
  - Waits until all child tasks have finished
The task construct (OpenMP 4.5)

The task construct is defined as:

```
#pragma omp task [clause[,...clause]] structured-block
```

where `clause` is one of the following:

- `if(task : scalar-expression)`
- `untied`
- `default(shared | none)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `final(scalar-expression)`
- `mergeable`
- `depend(dependence-type : list)`
- `priority(priority-value)`

Generates an explicit task

Task consists of:
- Code to execute
- Data environment

#pragma omp taskgroup
#pragma omp taskloop
#pragma omp taskyield
Parallel Fibonacci

int fib (int n)
{
    int x,y;
    if (n < 2) return n;

    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return (x+y);
}

int main()
{ int NW = 1000;
  #pragma omp parallel
  {
    #pragma omp master
    fib(NW);
  }
}
Linked lists with tasks

```c
#pragma omp parallel
{
    #pragma omp single
    {
        p=head;
        while (p) {
            #pragma omp task firstprivate(p)
            processwork(p);
            p = p->next;
        }
    }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined
Vectorization?

Vectorization is an on-node, in-core way of exploiting data level parallelism in programs by applying the same operation to multiple data items in parallel.

DO I= 1, N
    Z(I) = X(I) + Y(I)
ENDDO

• Requires transforming a program so that a single instruction can launch many operations on different data
• Applies most commonly to array operations in loops
What is Required for Vectorization?

- Vector Hardware: vector registers and vector functional units
- Code transformation

```
DO I = 1, N
  Z(I) = X(I) + Y(I)
ENDDO

DO I = 1, N, 4
  Z(I) = X(I) + Y(I)
  Z(I+1) = X(I+1) + Y(I+1)
  Z(I+2) = X(I+2) + Y(I+2)
  Z(I+3) = X(I+3) + Y(I+3)
ENDDO
```

Compiler

VLOAD X(I), X(I+1), X(I+2), X(I+3)
VLOAD Y(I), Y(I+1), Y(I+2), Y(I+3)
VADD Z(I, ..., I+3) X+Y(I, ..., I+3)
VSTORE Z(I), Z(I+1), Z(I+2), Z(I+3)
SIMD loop construct in OpenMP

- SIMD = single instruction applies the same operation to multiple data concurrently

- vectorization = processing multiple elements of an array at the same time.

- OpenMP can enable vectorization of both serial as well as parallelized loops

- OpenMP uses SIMD constructs.

```c
#pragma omp simd [clause [ [,] clause], …]
for-loops
```
Example

```c
void sprod(float *a, float *b, int n)
{
    float sum=0.0;

    #pragma omp simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
```

- Vectorize a loop nest is to cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body
void sprod(float *a, float *b, int n){
    float sum=0.0;
    #pragma omp for simd reduction(+:sum)
    for (int k=0; k<n; k++)
        sum += a[k] * b[k];
    return sum;
}
Example: loops

```c
#include <stdio.h>
#define N 10000
int main()
{
    long long d1=0;
    double a[N], b[N], c[N], d2=0.0;

    for (int i=0; i<N; i++)
        d1+=i*(N+1-i);

    for (int i=0; i<N; i++)
    {
        a[i]=i;
        b[i]=N+1-i;
    }

    for (int i=0; i<N; i++)
        d2+=a[i]*b[i];

    printf("result1 = %ld\nresult2 = %.2lf\n", d1, d2);
}
```
OpenMP SIMD Loop Example

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

#define N 10000
int main()
{
    long long d1=0;
    double a[N], b[N], c[N], d2=0.0;
#pragma omp simd reduction(+:d1)
    for (int i=0; i<N; i++)
        d1+=i*(N+1-i);
#pragma omp simd
    for (int i=0; i<N; i++) {
        a[i]=i;
        b[i]=N+1-i;
    }
#pragma omp parallel for simd reduction(+:d2)
    for (int i=0; i<N; i++)
        d2+=a[i]*b[i];
    printf("result1 = %ld\nresult2 = %.2lf\n", d1, d2);
}
```
Effort to support a wide variety of compute devices/accelerators: GPU, Xeon Phi

**target constructs**

- Creates a device data environment for the extent of the region
  - when a target data construct is encountered, a new device data environment is created, and the encountering task executes the target data region
  - when an if clause is present and the if-expression evaluates to false, the device is the host

The syntax of the **target** construct is as follows:

```
#pragma omp target [clause[[,] clause],...] new-line structured-block
```

where *clause* is one of the following:

- **device**(*integer-expression*)
- **map**(*map-type:* list*)
- **if**(*scalar-expression*)
More Directives and Functions for Devices

**omp target data:** Creates a device data environment and execute the construct on the same device. The target construct specifies that the region is executed by a device and the encountering task waits for the device to complete the target region

**omp target update:** Makes the corresponding list items in the device data environment consistent with their original list items

**omp distribute:** distributes a loop over the teams in the league

**omp declare target:** marks function(s) that can be called on the device

**omp teams:** Creates a league of thread teams where the master thread of each team executes the region, associated with num_teams and num_threads clause

omp get team num()
omp get team size()
omp get num devices()
Execution and Data Model

- **Host-centric**: the execution of an OpenMP program starts on the *host device* and it may offload *target regions* to *target devices*
  - In principle, a target region also begins as a single thread of execution: when a target construct is encountered, the target region is executed by the implicit device thread and the encountering thread/task [on the host] waits at the construct until the execution of the region completes.
  - If a target device is not present, or not supported, or not available, the target region is executed by the host device.
  - If a construct creates a *data environment*, the data environment is created at the time the construct is encountered.

- **When an OpenMP program begins, each device has an initial *device data environment***
  - Directives accepting data-mapping attribute clauses determine how an *original* variable is mapped to a *corresponding* variable in a device data environment.
    - original: the variable on the host
    - corresponding: the variable on the device
    - the corresponding variable in the device data environment may share storage with the original variable (danger of data races)
Example: Execution and Data Model

- **Environment Variable OMP_DEFAULT_DEVICE=<int>:** sets the device number to use in target constructs

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

- map variable B to device, then execute parallel region on device, works probably pretty well on Intel Xeon Phi

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i+= num_blocks)
    #pragma omp parallel for
    for (b = i; b < i+num_blocks; b++)
        B[b] += sin(B[b]);
```

- same as above, but code probably better optimized for NVIDIA GPGPUs
References


2. OpenMP tutorials:


**SIMD Vectorization with OpenMP**, Michael Klemm

**OpenMP for Accelerators - RWTH Aachen**