New Developments in OpenMP

Jemmy Hu

SHARCNET HPC Consultant
University of Waterloo

May 22, 2019

General Interest Seminar
**OpenMP overview**

*OpenMP: An API for Writing Multithreaded Applications*

- A set of compiler directives, library routines, and environment variables for parallel application programming

- 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 Brief History

Past

1997

OpenMP Fortran 1.0

1998

OpenMP C/C++ 1.0

1999

OpenMP Fortran 1.1

2000

OpenMP Fortran 2.0

2002

OpenMP C/C++ 2.0

2005

OpenMP 2.5

2008

OpenMP 3.0

2011

OpenMP 3.1

Present

2013

OpenMP 4.0

2015

OpenMP 4.5

Future

2018

OpenMP 5.0
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.
Types of Work-Sharing Constructs (Past):

**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.
Loop Parallelism

Sequential code

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

OpenMP parallel region

#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

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

<table>
<thead>
<tr>
<th>OpenMP pragma, function, or clause</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>Parallel region, teams of threads</td>
</tr>
<tr>
<td>int omp_get_thread_num()</td>
<td>Number of threads, thread ID</td>
</tr>
<tr>
<td>int omp_get_num_threads()</td>
<td></td>
</tr>
<tr>
<td>double omp_get_wtime()</td>
<td>Walltime, speedup measure</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>Synchronization</td>
</tr>
<tr>
<td>#pragma omp critica</td>
<td></td>
</tr>
<tr>
<td>#pragma omp for</td>
<td>Worksharing, parallel loops</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td></td>
</tr>
<tr>
<td>reduction(op:list)</td>
<td>Reduction</td>
</tr>
<tr>
<td>schedule(dynamic [,chunk])</td>
<td>Loop schedules</td>
</tr>
<tr>
<td>schedule (static [,chunk])</td>
<td></td>
</tr>
<tr>
<td>private(list), firstprivate(list), shared(list)</td>
<td>Data environment</td>
</tr>
<tr>
<td>nowait</td>
<td>Disable implied barrier</td>
</tr>
</tbody>
</table>
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 the 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

Serial     Parallel
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
OpenMP 4.x

- OpenMP has been significantly modernized since the OpenMP 4.0 (July 2013) and OpenMP 4.5 (Nov 2015) specification releases.

- Major additions include: SIMD, task dependencies, task groups, thread affinity, user defined reductions, taskloop, doacross.

- Target device support was first introduced in OpenMP 4.0 and was the focus for enhancement for OpenMP 4.5.

<table>
<thead>
<tr>
<th>Task Groups</th>
<th>Task Dependencies</th>
<th>Taskloop</th>
<th>Task Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Affinity</td>
<td>SIMD</td>
<td>Target Device Support</td>
<td></td>
</tr>
<tr>
<td>Doacross</td>
<td>Fortran 2003 Support</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenMP 4.0 Major Additions

- Task dependences and task groups
- SIMD constructs
- Device constructs
- Thread affinity control
- User-defined reductions
- Initial support for Fortran 2003
- Support for array sections (including in C and C++)
- Sequentially consistent atomics
- Display of initial OpenMP internal control variables
OpenMP 4.5 Focused on Device Support

- Unstructured data mapping
- Asynchronous execution
- Scalar variables are firstprivate by default
- Improvements for C/C++ array sections
- Device runtime routines: allocation, copy, etc.
- Clauses to support device pointers
- Ability to map structure elements
- New combined constructs
- New way to map global variables (link)
OpenMP 4.5 Other New Features

• Many clarifications and minor enhancements
  - SIMD extensions
  - Addition of schedule modifiers: simd, monotonic, nonmonotonic
  - Clarifications of thread affinity policies
  - Support for if clause on combined/composite constructs
  - Reductions for C/C++ arrays
  - Runtime routines to support affinity

• Support for doacross loops

• Divide loop into tasks with taskloop construct

• Task priorities
OpenMP Programming Model

- Directive-based programming model:
  - Multi-level parallelism supported (cpus, threads, SIMD)
  - Task-based is the modern approach to parallelism
  - High-level access to parallelism

- Hybrid MPI/OpenMP
  - Running on devices such as CPUs and GPUs.
The task construct (OpenMP 4.5)

```c
#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

```c
#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.
for (i = 0; i<SIZE; i+=1)
{
}

for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE : i+TS;
    for (ii=i; ii<UB; ii++) {
    }
}

#pragma omp task private(ii) \ 
    firstprivate(i,UB) shared(S,A,B)
for (ii=i; ii<UB; ii++) {
}

- Manual transformation is cumbersome and error prone
- Applying blocking techniques for large loops can be tricky
- taskloop: improved programmability
Parallelizing doacross Loop

- Help with cross-iteration dependencies
- Use “ordered” clause to ensure structured blocks are executed on lexical order
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.

```plaintext
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
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
SIMD Worksharing Construct

```c
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;
}
```

- Distribute a loop’s iteration across a thread team
- Subdivide loop chunks to fit a SIMD vector register
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

#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);
}
Existing Parallel Loop Constructs

Existing parallel loop constructs are tightly bound to execution model:

- `#pragma omp for` for (i=0; i<N;++i) {...}
- `#pragma omp simd` for (i=0; i<N;++i) {...}
- `#pragma omp taskloop` for (i=0; i<N;++i) {...}

Diagram:

- Fork
- Generate tasks
OpenMP for Accelerators: host/device 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
Device 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:

```plaintext
#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()
omp_get_default_device()
```
```c
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
    #pragma omp target device(0)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

    #pragma omp target update device(0) to(input[:N])

    #pragma omp target device(0)
    #pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i)
}
```
**OpenMP 5.0 updates:** [https://www.openmp.org](https://www.openmp.org)

**OPENMP 5.0 IS A MAJOR LEAP FORWARD**
*Full Support for Accelerators and New Tool APIs*

- **Full support for accelerator devices.** OpenMP now has full support for accelerator devices, including mechanisms to require unified shared memory between the host system and coprocessor devices, the ability to use device-specific function implementations, better control of implicit data mappings, and the ability to override device offload at runtime. In addition, it supports reverse offload, implicit function generation, and the ability to copy object-oriented data structures easily.

- **Improved debugging and performance analysis.** Two new tool interfaces enable the development of third party tools to support intuitive debugging and deep performance analysis.
OpenMP 5.0 updates

• **Support for the latest versions of C, C++, and Fortran.** OpenMP now supports important features of Fortran 2008, C11, and C++17.

• **Support for a fully descriptive loop construct.** The loop construct lets the compiler optimize a loop while not forcing any specific implementation. This construct allows the compiler more freedom to choose a good implementation for a specific target than do other OpenMP directives.

• **Multilevel memory systems.** Memory allocation mechanisms are available that place data in different kinds of memories, such as high-bandwidth memory. New OpenMP features also make it easier to deal with the NUMA-ness of modern HPC systems.

• **Enhanced portability.** The declare variant directive and a new meta-directive allow programmers to improve performance portability by adapting OpenMP pragmas and user code at compile time.
## OpenMP 5.0: Some Main Features

<table>
<thead>
<tr>
<th>Task Reductions</th>
<th>Memory Allocators</th>
<th>Detachable Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++14 and C++17 support</td>
<td>Dependence Objects</td>
<td>Tools APIs: OMPD, OMPT</td>
</tr>
<tr>
<td>loop Construct</td>
<td>Fortran 2008 support</td>
<td>Unified Shared Memory</td>
</tr>
<tr>
<td>Multi-level Parallelism</td>
<td>Task-to-data Affinity</td>
<td>Collapse non-rect. Loops</td>
</tr>
<tr>
<td>Parallel Scan</td>
<td>Meta-directives</td>
<td>Data Serialization for Offload</td>
</tr>
<tr>
<td>User Defined Function Variants</td>
<td>Display Affinity</td>
<td>“Reverse Offloading”</td>
</tr>
<tr>
<td></td>
<td>Improved Task Dependences</td>
<td></td>
</tr>
</tbody>
</table>
Task Reductions

- Task reductions extend traditional reductions to arbitrary task graphs.
- Extend the existing task and taskgroup constructs.
- Also work with the taskloop construct.

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        {
            while (node) {
                #pragma omp task in_reduction(+: res) \
                    firstprivate(node)
                {
                    res += node->value;
                }
                node = node->next;
            }
        }
    }
}
```
## Memory Allocators

<table>
<thead>
<tr>
<th>Allocator name</th>
<th>Storage selection intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_default_mem_alloc</td>
<td>use default storage</td>
</tr>
<tr>
<td>omp_large_cap_mem_alloc</td>
<td>use storage with large capacity</td>
</tr>
<tr>
<td>omp_const_mem_alloc</td>
<td>use storage optimized for read-only variables</td>
</tr>
<tr>
<td>omp_high_bw_mem_alloc</td>
<td>use storage with high bandwidth</td>
</tr>
<tr>
<td>omp_low_lat_mem_alloc</td>
<td>use storage with low latency</td>
</tr>
<tr>
<td>omp_cgroup_mem_alloc</td>
<td>use storage close to all threads in the contention group of the thread requesting the allocation</td>
</tr>
<tr>
<td>omp_pteam_mem_alloc</td>
<td>use storage that is close to all threads in the same parallel region of the thread requesting the allocation</td>
</tr>
<tr>
<td>omp_thread_local_mem_alloc</td>
<td>use storage that is close to the thread requesting the allocation</td>
</tr>
</tbody>
</table>
Example: Using Memory Allocators

```c
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCATOR and/or omp_set_default_allocator
    double *p = (double *) ompmalloc(N*M*sizeof(*p)); alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a)
    allocate(my_allocator:a)
    {  
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c) // on target; must be compile-time expr
    {
        #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
        {  
            some_other_parallel_code();
        }
    }

    omp_free(p);
}
Requires Unified Shared Memory

• Single address space over CPU and GPU memories
• Data migrated between CPU and GPU memories transparently to the application - no need to explicitly copy data

```c
// No data directive needed.
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++)
{
    //pragma omp target teams distribute parallel for
    for (j=0; j<ARRAY_SIZE; j++) {
        a[j] = b[j] + scalar * c[j];
    }
}
```
OpenMP Compilers and Tools

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Compiler/Language</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>GCC C/C++/Frotran</td>
<td>Free and open source From GCC 6.1, OpenMP 4.5 is fully supported for C and C++. Compile with -fopenmp to enable OpenMP.</td>
</tr>
<tr>
<td>Intel</td>
<td>C/C++/Frotran</td>
<td>OpenMP 4.5 C/C++/Fortran supported in version 17.0, 18.0, and 19.0 compilers Compile with -qopenmp on Linux</td>
</tr>
</tbody>
</table>

https://www.openmp.org/resources/openmp-compilers-tools/

OpenMP 5.0:

GNU/GCC 9, 2019, only partial support will be ready.
Intel: upcoming versions
OpenMP API 5.0: A Major Leap Forward

- Task Reductions
- Meta-directives
- Detachable Tasks
- Memory Allocators
- Dependence Objects
- Function Variants
- Improved Task Dependencies
- Collapse Non-Rectangular Loops
- Fortran 2008 Support
- Unified Shared Memory
- Multi-level Parallelism
- Tools APIs
- C++14 and C++17 Support
- Improved Affinity Support
- Data Serialization for Offload
- Parallel Scan
- Task-to-data Affinity
- Reverse Offloading
Resources


https://www2.cisl.ucar.edu/sites/default/files/MultiCore8_HelenHe.pdf

