OpenMP – Moving Beyond Shared-Memory Parallelism

Henry Jin
Computational Technologies Branch
NAS Division, NASA Ames Research Center

Applied Modeling & Simulation (AMS) Seminar Series
NASA Ames Research Center, September 2, 2014
Outline

• Brief history of OpenMP
• Requirements from modern computing architectures
• Development of OpenMP 4
  – Supporting new types of parallelism
  – Towards awareness of non-uniform memory access
• Challenges in achieving desired performance
• Future effort
What’s OpenMP

- The de-facto standard for shared-memory multiprocessing programming
  - The API consists of compiler directives and library routines for C/C++ and Fortran
  - Specification defined and maintained by the OpenMP Architecture Review Board
  - Implemented and supported by many compiler vendors

- Brief history
  - First version released in 1997
  - Current version 4.0 released in 2013
OpenMP Programming Model

• The *Fork-and-Join* model
  – Global view of application memory space
  – Thread oriented with task support

• Application and strength
  – Often used for exploiting parallelism on a node
  – When mixed with MPI
    • Maps reasonably well with multicore architectures
  – Path for incremental parallelization
Modern Architecture

Multiple processing cores in a socket

Multiple sockets in a host node

Accelerator devices attached to the host

Socket Interconnect

Network Interface

Memory Controller

Local Memory

Connection across the sockets

Distributed memory cluster of multi-socket nodes

Within a node: mix of different processor and memory types

National Aeronautics and Space Administration
Modern Architecture and OpenMP

• Heterogeneous computing node
  – Multi-core and/or many-core processors
  – Host processors attached with accelerator devices (such GPUs, Intel MIC)
  – Parallelism at multiple hardware levels
  – Non-uniform memory access, disjoint memory access

• Mismatch of OpenMP 3 with modern architecture
  – No concept of non-uniform memory access
  – No handle for disjoint memory
  – Unaware of parallelism at different hardware levels
    • SIMD/MIMD parallelism
  – No support for accelerators

• OpenMP 4 was developed to overcome many of the deficiencies
New Features in OpenMP 4

- Support for accelerator devices (**target** construct)
- Initial support for error model (**cancel** construct)
- Task dependences (**depend** clause)
- Deep task synchronization (**taskgroup** construct)
- Fortran 2003 initial support
- User-defined reduction (**declare reduction** construct)
- SIMD extensions for vector parallelism (**simd** construct)
- Thread affinity (**proc_bind** clause and **OMP_PLACES**)
- Further enhancement to atomic operations
- Display OpenMP environment variables
New Features in OpenMP 4

- Support for accelerator devices (target construct)*
- Initial support for error model (cancel construct)
- Task dependences (depend clause)
- Deep task synchronization (taskgroup construct)
- Fortran 2003 initial support
- User-defined reduction (declare reduction construct)
- SIMD extensions for vector parallelism (simd construct)*
- Thread affinity (proc_bind clause and OMP_PLACES)*
- Further enhancement to atomic operations
- Display OpenMP environment variables

*Focus of this talk*
Programming Consideration for Accelerator Devices

• **Host driven model**
  – Host initiates execution
  – Host offloads work to devices
  – Host manages data transfer between host and devices

• **Two types parallelism on accelerators:**
  – SIMD – single instruction multiple data, such as
    • Threads in a single warp or single thread block in NVIDIA GPUs
    • Vector processing units in Intel Xeon Phis
  – MIMD – multiple instructions multiple data, such as
    • Multiple thread blocks executed in parallel on different NVIDIA multiprocessors
    • Threads spread across multiple cores

• **Concurrency between host and devices**
• **Disjoint memory between host and devices**
The Offloading Model

- **Host offloads task to accelerator**
  - The task is defined by the `target` construct
- **Host manages data transfer (mapping)**
  - Via the `map` clause
  - *Allocating* device data and *transferring to* device at entry to the `target` region
  - *Transferring from* device and *deallocating* device data at exit from the `target` region
- **Host waits for completion of the target region on device**

**In C:**

```c
int n, i;
float a, *x, *y, *z;
#pragma omp target map(to:x(:n), y(:n)) map(from:z(:n))
for (i = 0; i < n; i++)
    z[i] = a*x[i] + y[i];
```

**In Fortran:**

```fortran
integer :: n, i
real :: a, x(n), y(n), z(n)
!$omp target &
!$omp& map(to:x, y) map(from:z)
do i = 1, n
    z(i) = a*x(i) + y(i)
dend do
```
Target Execution on Device

- Target region is executed by a single team of threads by default
- Inside the target region follows the regular OpenMP execution model
  - May contain other OpenMP constructs except for the target related constructs
- Multiple teams may be defined by the teams construct
  - Mapped to MIMD parallelism on GPUs
  - Each team has its own contention group for thread synchronization
  - The distribute construct distributes loop iterations over teams

In C:
```c
int n,i;
float a,*x,*y,*z;
#pragma omp target teams map(to:x(:n),y(:n)) map(from:z(:n))
{
    #pragma omp distribute parallel for
    for (i = 0; i < n; i++)
        z[i] = a*x[i] + y[i];
}
```

In Fortran:
```fortran
integer :: n,i
real :: a,x(n),y(n),z(n)
!$omp target teams &
!$omp& map(to:x,y) map(from:z)
!$omp distribute parallel do
    do i = 1, n
        z(i) = a*x(i) + y(i)
    end do
!$omp end target teams
```
Cost of Data Transfer

• The MG offload codes on Intel Xeon Phi

• Three different versions with different granularity for offloading
  – One OpenMP loop, many data transfers
  – One subroutine, rest on host
  – Whole computation with single data transfer at the beginning and at the end

• Overhead from data transfer can be substantial

Saini et al. at the SC13 conference.
- Performance of acc directive is close to that of the cuda versions when not considering data transfer between the host and the device
- When considering data transfer, GPU performance is about the same as the host
Reducing Data Transfer

- **Target data region**
  - Specified by the `target data` construct
  - Allocating and transferring device data via the `map` clause
  - Codes executed by host, not by device

- **Declare target directive**
  - Allocates global data on device
  - Declares device functions/procedures

- **The present test**
  - No implicit data transfer at target and target data constructs if a variable is already mapped either by target data construct or by declare target directive

- **Target update construct**
  - Performs explicit data transfer

*C Example:*
```c
int n,i;
float a,*x,*y,*z;
init(x, y, n);
#pragma omp target data
  map(to:x[:n],y[:n]) map(from:z[:n])
{
  #pragma omp target
  #pragma omp parallel for
  for (i = 0; i < n; i++)
    z[i] = a*x[i] + y[i];
init_again(x,y,n);
  #pragma omp target update
    to(x[:n],y[:n])
  #pragma omp target
  #pragma omp parallel for
  for (i = 0; i < n; i++)
    z[i] += a*x[i] + y[i];
}
output(z,n);
```
Asynchronous Execution

- **Purpose**
  - Overlap host and device computation
  - Hide data transfer overhead

- **Synchronous execution of target construct**
  - Host task has to wait for the completion of the offloaded code

- **Indirect solution**
  - Wrap `target` construct inside a task
  - Use `taskwait` to synchronize execution
  - Carry all the baggage of the tasking model
    - Task data environment

*C Example:*
```c
int n,i;
float a,*x,*y,*z,*p;

init(x, y, n);
#pragma omp task shared(x,y)
{
    #pragma omp target \map(to:x[:n],y[:n]) map(from:z[:n])
    #pragma omp parallel for
    for (i = 0; i < n; i++)
        z[i] = a*x[i] + y[i];
}
#pragma omp task shared(p)
compute_p(p, n);
#pragma omp taskwait
output(z, p, n);
```
**SIMD Parallelism**

**Single Instruction Multiple Data (SIMD) or vector instruction**
- Process multiple data in one instruction
- Supported on many types of hardware

**Compiler auto-vectorization in general**
- Inner-most loop
- No function calls
- Independent loop iterations
- Compiler directives as hints

**SIMD in OpenMP**
- For cases where compiler cannot perform auto-vectorization
- Prescriptive in nature, user responsible for correctness
- Support for function calls

```c
for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];
```

```
vadd
```
SIMD Loops

- **Simd construct**
  - Applies to a loop to indicate that multiple iterations of the loop can be executed concurrently using SIMD instructions

- **Loop SIMD or distribute simd construct**
  - Specifies that a loop is first distributed among team of threads (loop SIMD) or teams (distribute simd) in chunks and then each chunk is applied with the simd construct

---

**A simple example:**
```
#pragma omp declare simd uniform(fact)
double add(double a, double b, double fact)
{ return (a + b + fact); }
void work(double *a, double *b, int n)
{ int i;
#pragma omp simd
  for (i = 0; i < n; i++)
    a[i] = add(a[i], b[i], 1.0);
}
```

**A more convoluted example:**
```
int n,i;
float *x,*y,*z;
init(x, y, n);
#pragma omp target teams 
  map(to:x(:n),y(:n)) map(from:z(:n))
{
  #pragma omp distribute simd
  for (i = 0; i < n; i++)
    z[i] = x[i] * y[i];
}
output(z, n);
```
Thread Affinity

- **Thread-processor binding**
  - Map OpenMP threads to hardware resources (such as cores)
  - Logical processor units via the `OMP_PLACES` environment variable
  - Affinity policy *(close, spread, master)* for threads in parallel regions
  - Handling thread affinity in nested parallel regions

- **Benefit**
  - May improve performance by reducing OS scheduling overhead and improving resource utilization
  - Reduce run-to-run timing variation

Example of using thread binding from two types of affinity settings to improve resource utilization
Thread Affinity Types

Examples of `OMP_NUM_THREADS=8` on a node with two quad-core sockets with HyperThreading:
`OMP_PLACES=“{0,8},{1,9},{2,10},{3,11},{4,12},{5,13},{6,14},{7,15}”`

**proc_bind(close)** – better cache sharing between threads

**proc_bind(spread)** – maximizing memory bandwidth utilization

**proc_bind(master)** – assigning threads to the same place as the master

“spread” usually gives better results for most cases
Challenges in Achieving Desired Performance

- **Software development**
  - For accelerator devices
    - Identify and offload hotspots
    - Minimize data transfer overhead
  - Exploiting sufficient parallelism at different levels to match with hardware
    - Cores, threads, vectors
  - Data structure consideration
    - Stride-one memory access
    - Cache blocking
  - Code modification is often required, but may not be portable

- **Application development**
  - Need enough parallelism to match with hardware
  - Potentially require different numerical algorithms
  - Concern about performance, load balance
### Timing Profile of SP on Pleiades-GPU

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>base-line</th>
<th>simple</th>
<th>mlocal</th>
<th>mirror</th>
<th>dim-prom</th>
<th>data-trans</th>
<th>cuda</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_rhs</td>
<td>16.21</td>
<td>15.82</td>
<td>126.84</td>
<td>38.37</td>
<td>3.47</td>
<td>3.47</td>
<td>3.41</td>
<td>1.72</td>
</tr>
<tr>
<td>x_solve</td>
<td>6.08</td>
<td>6.90</td>
<td>35.91</td>
<td>25.77</td>
<td>22.48</td>
<td>7.56</td>
<td>3.09</td>
<td>2.55</td>
</tr>
<tr>
<td>y_solve</td>
<td>6.20</td>
<td>6.26</td>
<td>34.39</td>
<td>20.51</td>
<td>17.27</td>
<td>2.07</td>
<td>2.11</td>
<td>1.74</td>
</tr>
<tr>
<td>z_solve</td>
<td>6.35</td>
<td>6.99</td>
<td>32.33</td>
<td>20.78</td>
<td>17.46</td>
<td>2.45</td>
<td>2.49</td>
<td>1.71</td>
</tr>
<tr>
<td>add</td>
<td>1.09</td>
<td>0.76</td>
<td>16.56</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>015</td>
</tr>
<tr>
<td>rest</td>
<td>3.88</td>
<td>2.49</td>
<td>57.89</td>
<td>5.30</td>
<td>0.94</td>
<td>0.94</td>
<td>0.95</td>
<td>0.18</td>
</tr>
<tr>
<td>total</td>
<td>39.81</td>
<td>39.22</td>
<td>302.58</td>
<td>110.56</td>
<td>61.65</td>
<td>16.63</td>
<td>12.22</td>
<td>8.03</td>
</tr>
<tr>
<td>GPUkrnl</td>
<td>0.00</td>
<td>0.00</td>
<td>59.37</td>
<td>59.60</td>
<td>59.78</td>
<td>15.14</td>
<td>10.32</td>
<td>7.34</td>
</tr>
<tr>
<td>GPUcomm</td>
<td>0.00</td>
<td>0.00</td>
<td>243.21</td>
<td>50.96</td>
<td>1.87</td>
<td>1.49</td>
<td>1.90</td>
<td>0.69</td>
</tr>
</tbody>
</table>

- dominated by data transfer
- inefficient memory access

*Jin et al., at IWOMP2012.*
- The *simple* and *mlocal* versions show much worse performance than baseline, dominated by data transfer between the host and the device.
- The *mirror* version is limited by the kernel performance of the three solvers.
- Code restructure in *dim-prom* and *data-trans* for better memory coalescing is the key for further improvement.
Remote Data Access and NUMA Effect

- Remote data access is more expensive
  - May cause memory access bottleneck
- Data layout and memory affinity are important

Performance of BT from the NAS Parallel Benchmarks on the SGI Altix
- Four types of data layout based on how data are initially distributed
Future OpenMP Extensions

- **Work in progress within the OpenMP language committee**
  - Technical report on extensions for accelerator support by SC14
  - The 4.1 release targeted for SC15
  - Features considered for 5.0

- **New features under consideration**
  - Refinement to accelerator device support
    - Unstructured data movement
    - Asynchronous execution
    - Multiple device types
  - Full error model
  - Full Fortran 2003 support
  - Interoperability with other models (MPI, pthreads)
  - Support for NUMA
    - Memory affinity
  - Tools interface
Refinement to Accelerator Support

• Unstructured data movement
  – target enter data construct
  – target exit data construct

• Asynchronous execution
  – Better integration with the tasking model
    • target task
  – Flexible control via task dependency
    • depend clause on target construct
Summary

- **OpenMP has been moving beyond shared memory parallelism**
  - Support for accelerator devices
    - Many features were adopted from OpenACC
  - Ability to exploit hierarchical multi-level parallelism
    - MIMD via `teams` construct
    - Thread level via parallel loop construct
    - SIMD via `simd` construct

- **OpenMP’s new mission statement**
  “Standardize directive-based multi-language high-level parallelism that is performant, productive and portable.”

- **Compilers with OpenMP support are widely available**
  - Although support for 4.0 is still in work
Summary (cont.)

• Achieving desired performance in applications is still challenging
  – Exploiting multi-level parallelism
  – Reducing cost of data transfer between the host and the device
  – Optimizing memory accesses
    • Changes to code structure may be needed, but not always portable

• More experience is needed to experiment with the new features
OpenMP ARB members

- Permanent members: AMD, Convey, Cray, Fujitsu, HP, IBM, Intel, NEC, NVIDIA, Oracle, Red Hat, STMicro, Texas Instruments
- Auxiliary members: ANL, Barcelona Supercomputing Center, cOMPunity, EPCC, LANL, LLNL, NASA Ames, ORNL, RWTH Aachen University, SNL, University of Houston, TACC
References

- **OpenMP specifications**
  - www.openmp.org/wp/openmp-specifications/

- **Resources**
  - www.openmp.org/wp/resources/
  - www.compunity.org/

- **Benchmarks**
  - OpenMP Microbenchmarks from EPCC (www.epcc.ed.ac.uk/research/openmpbench)
  - NAS Parallel Benchmarks (www.nas.nasa.gov/publications/npb.html)