OpenMP Language Committee Report
June 14, 2011

Bronis R. de Supinski
OpenMP Language Committee Chair
Center for Applied Scientific Computing
Science & Technology Principal Directorate - Computation Directorate
OpenMP is a vibrant growing organization

- ARB membership at an all-time high
  - 13 permanent members (implementers)
    - Most recent addition is Nvidia
  - 8 auxiliary members (user institutions)
    - Most recent addition is TACC
- Actively pursuing new specifications
  - OpenMP 3.1 released for comment in March 2011
  - Significant work already begun on OpenMP 4.0
  - Planning always extends beyond the next specification
  - Feedback from non-members always welcome
- International Workshop on OpenMP (IWOMP) going strong
The next OpenMP specification is all but finished and work on the following one is already begun

- **OpenMP 3.1**
  - Refine and extend existing specification
  - Do not break existing code
  - Minimal implementation burden beyond 3.0
  - Expect 85 tickets total (81 already passed)
  - Approval very soon (next month)
    - Wrap up final review tickets and examples Thursday
    - ARB vote on final version next month (July 9)

- **OpenMP 4.0**
  - Address several major open issues for OpenMP
  - Do not break existing code unnecessarily
  - Draft planned/hoped for SC12
Despite incremental nature, we are targeting several important items for OpenMP 3.1

- Extend atomics to support capture and write functionality
- Add $\min$ and $\max$ reduction operators in C/C++
- Extensions to OpenMP tasking model
  - Explicit task scheduling points (taskyield construct)
  - Ability to save data environment overhead
    - $\text{final}$ and mergeable clauses
    - $\text{omp_in_final}$ runtime library routine
- Initial support for thread binding
- Now allow intent(in) and const-qualified types in firstprivate clause
- Many clarifications, including improvements to examples
The final clause combines with new tasking concepts to reduce tasking overhead

- Recognizing an existing concept and creating three new ones
  - An **undeferred task** is a task for which execution is not deferred with respect to its generating task region
    
    ```
    #pragma omp task if(0)
    ```
  
  - An **included task** is an undeferred task that is sequentially included in generating task region (executed immediately)
  
  - A **merged task** has the same data environment, including ICVs, as its generating task region
  
  - A **final task** forces its descendant tasks to be included

- New extensions to the task construct
  - The **mergeable** clause suggests the task may be merged
  
  - The **final(expr)** clause if true results in a final task
Final tasks allow compiler to generate merged tasks, thus saving data environment overhead

- A simple example illustrates

```c
void fib(int n, int d) {
    int x, y;
    if (n < 2) return 1;
    #pragma omp task final(d > LIMIT) mergeable
        x = fib(n - 1, d + 1);
    #pragma omp task final(d > LIMIT) mergeable
        x = fib(n - 2, d + 1);
    #pragma omp taskwait
    return x + y;
}
```

- The combination enables fine-grain tasking control
- Support specialization with `omp_in_final` call
- Mergeable tasks that use `firstprivate` data can be unsafe
  - Result can vary since no data environment is created (i.e., no private copy) if task is merged
Using final and mergeable tasks performs comparably to hand optimizations

- Nqueens (14x14 board)
Additional kind of atomic operations addresses an obvious deficiency

- Currently cannot capture a value atomically

```c
int schedule (int upper) {
    static int iter = 0; int ret;
    ret = iter;
    #pragma omp atomic
    iter++;
    if (ret <= upper) { return ret; }
    else { return -1; }  // no more iters
}
```

- Atomic capture provides the needed functionality

```c
int schedule (int upper) {
    static int iter = 0; int ret;
    #pragma omp atomic capture
    ret = iter++;  // atomic capture
    if (ret <= upper) { return ret; }
    else { return -1; }  // no more iters
}
```
Adding initial high-level affinity support to the OpenMP 3.1 specification, more planned for 4.0

- Control of nested thread team sizes (in OpenMP 3.1)
  ```
  export OMP_NUM_THREADS=4,3,2
  ```

- Request binding of threads to resources (in OpenMP 3.1)
  ```
  export OMP_PROC_BIND=TRUE
  ```

- Restrict the processor set for program execution
  ```
  export OMP_PROCSET 0,1,2,3,8,10,12,14
  ```

- Control thread placement within a processor set
  ```
  export OMP_AFFINITY=scatter,scatter,compact
  ```

- Control initial placement of shared data
  ```
  export OMP_MEMORY_PLACEMENT=spread | local
  ```

- Adapt data placement at runtime
  ```
  #pragma omp migrate (list) strategy (local | spread)
  ```
User Defined Reductions (UDRs) are a major addition planned for OpenMP 4.0

- **Use** `declare reduction` directive to define new operators
- **New operators** used in reduction clause like predefined ops

```c
#pragma omp declare reduction (reduction-identifier : typename-list : combiner) [identity(identity-expr)]
```

- **reduction-identifier** gives a name to the operator
  - Can be overloaded for different types
  - Can be redefined in inner scopes
- **typename-list** is a list of types to which it applies
- **combiner** expression specifies how to combine values
- **identity** can specify the identity value of the operator
  - Can be an expression or a brace initializer
A simple UDR example

- Declare the reduction operator

```cpp
#pragma omp declare reduction (merge : std::vector<int> : 
omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

- Use the reduction operator in a reduction clause

```cpp
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
    #pragma omp parallel for reduction (merge : filtered) 
    for (std::vector<int>::iterator it = v.begin(); it < v.end(); it++)
        if (filter(*it) ) filtered.push_back(*it);
}
```

- Private copies created for a reduction are initialized to the identity that was specified for the operator and type
  - Default identity defined if no identity clause present
- Compiler uses combiner to combine private copies
  - `omp_out` refers to private copy that holds combined value
  - `omp_in` refers to the other private copy
We are actively discussing several major topics for OpenMP 4.0 and beyond

- Development of an error model
  - The `done` directive
  - Functionality like `errno`
  - Callbacks for integrated error handling

- Interoperability and composability
  - Interactions between thread models
  - Interfaces to support interactions with distributed models

- Refinements to the OpenMP tasking model
  - Specifying task dependencies (think data flow)
  - Task reductions, task-only threads, `omp while`

- Affinity (previous slide)
- Sequentially consistent atomic operations
- How to specify subarrays in C
Major new direction of OpenMP 4.0 will target a wide range of accelerator configurations

- Dedicated hardware for specific function(s)
  - Attached to a master processor
  - Multiple types or levels of parallelism
    - Process level, thread level, ILP/SIMD
  - May not support a full C/C++ or Fortran compiler
    - May lack stack or interrupts, may limit control flow, types
Planning an accelerator tasking model based on prototypes from several vendors

- Include a clause to indicate **accelerate region**
- Generate an explicit **accelerate-task**
  - Default is tied; may limit untied to same accelerator type
  - Run to completion on an accelerator
  - Scheduled by runtime as directed by user clauses
  - Complete at barriers, explicit accelerator task sync points
- A **data region** will define an accelerator data environment
  - Implicit for an accelerate region
  - Explicitly create persistent one with a **data region** construct
- Discussing a wide range of issues
  - Implications for loop construct (a new one?)
  - Possible restrictions of other constructs
Use data regions to limit data movement

```c
void foo(double A[], double B[], double C[], int nrows, int ncols) {
    #pragma omp data_region acc_copyout(C), host_shared(A,B)
    {
        #pragma omp acc_region
        for (int i=0; i < nrows; ++i)
            for (int j=0; j < ncols; j += NLANES)
                for (int k=0; k < NLANES; ++k) {
                    int index = (i * ncols) + j + k;
                } // end accelerator region
        print2d(A,nrows,ncols);
        print2d(B,nrows,ncols);
        Transpose(C); // calls function w/another accelerator construct
    } // end data_region
    print2d(C, nrows, ncols);
}
void Transpose(double X[], int nrows, int ncols) {
    #pragma omp acc_region acc_copy(X), acc_present(X)
    {
    ...
    }
```
Data placement clauses will support efficient and flexible data interaction with master

<table>
<thead>
<tr>
<th>Clause</th>
<th>Allocate on Accelerator</th>
<th>Instances</th>
<th>Construct</th>
<th>Implicit Update</th>
<th>Explicit Update</th>
<th>Present Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>host_shared</td>
<td>No</td>
<td>Host</td>
<td>Data/Accel</td>
<td>In/Out ?</td>
<td>No</td>
<td>False</td>
</tr>
<tr>
<td>acc_shared</td>
<td>Yes</td>
<td>Host&amp;Acc</td>
<td>Data/Accel</td>
<td>No</td>
<td>Yes</td>
<td>True</td>
</tr>
<tr>
<td>acc_copy</td>
<td>Yes</td>
<td>Host&amp;Acc</td>
<td>Data/Accel</td>
<td>In/Out</td>
<td>Yes</td>
<td>True</td>
</tr>
<tr>
<td>acc_copyin</td>
<td>Yes</td>
<td>Host&amp;Acc</td>
<td>Data/Accel</td>
<td>In</td>
<td>Yes</td>
<td>True</td>
</tr>
<tr>
<td>acc_copyout</td>
<td>Yes</td>
<td>Host&amp;Acc</td>
<td>Data/Accel</td>
<td>Out</td>
<td>Yes</td>
<td>True</td>
</tr>
<tr>
<td>acc_private</td>
<td>Yes</td>
<td>Acc per PE</td>
<td>Accel</td>
<td>No</td>
<td>No</td>
<td>Imp Def</td>
</tr>
<tr>
<td>acc_firstprivate</td>
<td>Yes</td>
<td>Acc per PE</td>
<td>Accel</td>
<td>In</td>
<td>No</td>
<td>Imp Def</td>
</tr>
<tr>
<td>acc_present</td>
<td>No</td>
<td>N/A</td>
<td>Data/Accel</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Future manycore systems will require tasking; OpenMP will evolve to meet this challenge

- Task reductions
- Task groups support grouping of tasks for:
  - Synchronization
  - Reductions
  - Termination
- User-defined privatization
- Task-only threads
- Task affinity
- Task dependencies
  - Evolve OpenMP task model into a dataflow model
  - Reduces synchronization to improve load balance
LU and QR factorization naturally suit OpenMP’s traditional fork-join parallelization approach

- Parallelize the update (the 2/3n^3 term in the FLOPs count)

Slide courtesy of Jack Dongarra
LU/LLᵀ/QR algorithms can also be viewed in terms of task parallelism

- Break into smaller tasks and remove dependencies
“Task-ified” algorithms require appropriate data layout to achieve good performance

- Tile data layout where each data tile is contiguous in memory
- Decomposed into several fine-grained tasks, which better fit the memory of the small core caches

Slide courtesy of Jack Dongarra
PLASMA: Parallel Linear Algebra Software for Multicore Architectures

- **Objectives**
  - High utilization of each core
  - Scaling to large number of cores
  - Shared or distributed memory

- **Methodology**
  - Dynamic DAG scheduling (QUARK)
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout

- **Arbitrary DAG with dynamic scheduling**

Slide courtesy of Jack Dongarra
Synchronization reducing algorithms

- Regular trace
- Factorization steps pipelined
- Stalling only due to natural load imbalance
- Dynamic
- Out of order execution
- Fine grain tasks
- Independent block operations

The colored area over the rectangle is the efficiency

Tile LU factorization; Matrix size 4000x4000, Tile size 200 8-socket, 6-core (48 cores total) AMD Istanbul 2.8 GHz

Slide courtesy of Jack Dongarra
Pipelining: Cholesky Inversion
3 Steps: Factor, Invert L, Multiply L’s

48 cores
POTRF, TRTRI and LAUUM.
The matrix is 4000 x 4000, tile size is 200 x 200,

POTRF+TRTRI+LAUUM: 25 (7t-3)
Cholesky Factorization alone: 3t-2

Pipelined: 18 (3t+6)

Slide courtesy of Jack Dongarra
We can garner many of PLASMA’s advantages in OpenMP through support for task dependencies

- Syntax to express task dependencies is still to be decided
  - Barcelona proposal specifies data dependencies explicitly
  - A simple example illustrates proposed syntax

```
#pragma omp task input (a)
{
    A
    #pragma omp task output(a)
    B
    #pragma omp task inout(a)
    C
}
#pragma omp task input (a)
D
```

- Tasks A, B & D can proceed in parallel; C must wait for B
- needs subarray or array slicing & other data region concepts
The OpenMP Language Committee must address the following long term question:

- With parallelism entering mainstream languages (Fortran 2008 do concurrent, threads in C++0X and next C standard) and the advent of shared memory libraries (e.g., Microsoft’s PPL and Intel's TBB), does OpenMP directive-based parallelism become obsolete?
Deterministic Parallel Java (DPJ) provides safety guarantees that OpenMP could standardize

- OpenMP *intends* strong safety properties
  - Sequential equivalence *(without critical, atomic)*
  - Data race freedom (always)
  - Deadlock freedom (always)
- Today’s compilers do not verify these properties
  - Unintentional data races can occur
  - Critical sections may be omitted or placed incorrectly
  - Task parallelism compounds these problems
- Can we extend OpenMP so a compiler or tool can verify the intended properties?

Slide courtesy of Vikram Adve
Example from SpecOMP

```c
// find_match(), match() access shared and private arrays
#pragma omp parallel private(mcp, busp)
for (int o=0; o < numWorkers; o++) {
    find_match(o);
    match(o, xcoor, ycoor, mcp, busp);
}
```

```
int match(const int o, int xcoor, int ycoor,
    double* const mcp, double** const busp) {...}
```

All checking is intraprocedural or intramodular

- Intraprocedural if effect summaries on *all* methods
- Intramodular but interprocedural if effect summaries only on module interface functions

Compiler can verify using method effect summaries

Slide courtesy of Vikram Adve
Effect summaries and regions naturally extend existing and proposed OpenMP concepts

- Effect summaries are written in terms of regions
  - reads(region-list)
  - writes(region-list)

- Regions are locations similar to task dependency proposal
  - Use variable names for scalars, array sections, “restricted” pointers: \( x, A[L:U], B[i][*], *p, p->m \)
  - Use region parameters on classes to distinguish pointers for unrestricted pointers, trees and graphs
  - Use logical preconditions on loops for array references with complex indexing such as indirection vectors:
    
    ```
    #pragma omp parallel for assume
    i!=J => Idx[i] != Idx[j]
    ```

    Only case that needs optional run-time checks to verify the preconditions; all other cases use purely static checking
These extensions could greatly simplify the writing of correct shared memory programs

- **Strong safety guarantees**
  - Sequential equivalence in many cases
  - Data race freedom and deadlock freedom in all cases
- **Machine-checkable documentation of sharing behavior**
  - Effect summaries are a concise, enforceable API contract
- **Modular development of shared memory programs**
  - Add effect summaries to internal and external APIs
  - Programmer can choose whether to check annotated APIs
- **Modular program analysis and reasoning**
  - More precise alias analysis, optimization

For more information, see Bocchino et al. [HotPar09]:
also Bocchino et al. [OOPSLA09, POPL11, ECOOP11], Karmani et al. [PPoPP’10]
We are considering these and several other topics for OpenMP 4.0 and beyond

- Other topics being considered for OpenMP 4.0
  - Transactional memory and thread level speculation
  - Additional task/thread synchronization mechanisms
  - Extending OpenMP to Fortran 2003
  - Extending OpenMP to additional languages
  - Incorporating tools support
  - Other miscellaneous extensions

- How can you help shape the future of OpenMP?
  - **Attend IWOMP, become a cOMPunity member**
  - Lobby your institution to join the OpenMP ARB
  - Contact me and beg ;-)