Progress on OpenMP Specifications

Wednesday, November 13, 2012

Bronis R. de Supinski
Chair, OpenMP Language Committee

LLNL-PRES-599952

This work has been authored by Lawrence Livermore National Security, LLC under contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting this work for dissemination, acknowledges that the United States Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the disseminated form of this work or allow others to do so, for United States Government purposes.

The OpenMP Language Committee schedule will meet community needs

- OpenMP 3.1 released in July 2011
- OpenMP 4.0 is nearing completion
  - Welcome comments on first draft (“RC1”)
    - http://openmp.org/wp/openmp-specifications/
    - OpenMP Forum topic for comments through January 18, 2013
  - Planning on second draft (“RC2”)
    - Several topics almost but not quite done
    - Will be released middle of February 2013
- Plan to work immediately after on OpenMP 5.0
- Feedback from non-members always welcome

OpenMP 3.1 specification completed and OpenMP 4.0 progressing

- OpenMP 3.1
  - Refine and extend existing specification
  - Do not break existing code
  - Minimal implementation burden beyond 3.0
  - Enacted 87 tickets total
- OpenMP 4.0
  - Draft planned for SC12 (adopting time-based releases)
  - Address several major open issues for OpenMP
  - Do not break existing code unnecessarily
  - RC1 includes 31 tickets (several major ones)
    - Added support for SIMD directives
    - Significantly extended support for thread affinity
    - Added UDRs, sequentially consistent atomics, atomic swap
    - Added initial support for Fortran 2003

Despite incremental nature, we added several important items for OpenMP 3.1

- New atomics 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
    - `final` and `mergeable` clauses
    - `omp_in_final` runtime library routine
- Initial support for thread binding
- Now allow `intent(in)` and const-qualified types in `firstprivate` clause
- Many clarifications, improvements to examples

Reminiscent of our roots, OpenMP 4.0 will provide portable SIMD constructs

- Use `simd` directive to indicate a loop should be SIMDized

```
#pragma omp simd [clause [[,] clause] …]
```

- Execute iterations of following loop in SIMD chunks
  - Region binds to the current task, so loop is not divided across threads
  - SIMD chunk is set of iterations executed concurrently by a SIMD lanes
- Creates a new data environment
- Clauses control data environment, how loop is partitioned
  - `safelen(`*`length`*`)` limits the number of iterations in a SIMD chunk
  - `linear` lists variables with a linear relationship to the iteration space
  - `aligned` specifies byte alignments of a list of variables
  - `private`, `lastprivate`, `reduction` and `collapse` have usual meanings
  - Would `firstprivate` be useful?

What happens if a SIMDized loop includes function calls?

- Could rely on compiler to handle
  - Compiler could in-line function to SIMDize its operations
  - Compiler could try to generate SIMDize version of function
  - Inefficient default would call function from each SIMD lane
- Provide `declare simd` directive to generate SIMD function

```
#pragma omp declare simd [clause [[,] clause] …]
function definition or declaration
```

  - Invocation of generated function processes across SIMD lanes
- Clauses control data environment, how function is used
  - `simdlen(length)` specifies the number of concurrent arguments
  - `uniform` lists invariant arguments across concurrent SIMD invocations
  - `inbranch` and `notinbranch` imply always/never invoked in a conditional statement
  - `linear`, `aligned`, and `reduction` are similar to `simd` clauses

The loop SIMD and parallel loop SIMD combine two types of parallelism

- The loop SIMD construct workshares and SIMDizes loop

```
#pragma omp for simd [clause [[,] clause] …]
```

  - Cannot be specified separately
  - Loop is first divided into SIMD chunks
  - SIMD chunks are divided across implicit tasks
  - Not guaranteed same schedule even with `static` schedule
- Use parallel loop SIMD construct for a parallel region that only contains a loop SIMD construct

```
#pragma omp parallel for simd [clause [[,] clause] …]
```

  - Purely a convenience that combines separate directives
  - Analogous to the combined parallel worksharing constructs
  - Would a parallel SIMD construct (i.e., no worksharing) be useful?

The declare simd construct supports SIMD execution of library routines

- Tell compiler to generate SIMD versions of functions

```
#pragma omp simd notinbranch
float min (float a, float b) {
   return a < b ? a : b;   }

#pragma omp simd notinbranch
float distsq (float x, float y) {
   return (x - y) * (x - y);   }
```

- Compile library and use functions in a SIMD loop

```
void minex (float *a, float *b, float *c, float *d) {
  #pragma omp parallel for simd
  for (i = 0; i < N; i++)
    d[i] = min (distsq(a[i], b[i]), c[i]);
}
```

- Creates implicit tasks of parallel region
- Divides loop into SIMD chunks
- Schedules SIMD chunks across implicit tasks
- Loop is fully SIMDized by using SIMD versions of functions

RC1 significantly extends initial high-level affinity support of OpenMP 3.1

- Control of nested thread team sizes (in OpenMP 3.1)

```
export OMP_NUM_THREADS=4,3,2
```

- Request binding of threads to places (in OpenMP 3.1)

```
export OMP_PROC_BIND=TRUE
```

- New extensions specify thread locations
  - Increased choices for `OMP_PROC_BIND`
    - Can still specify `true` or `false`
    - Can now provide a list (possible item values: `master`, `close` or `spread`) to specify how to bind implicit tasks of parallel regions
  - Added `OMP_PLACES` environment variable
    - Can specify abstract names including `threads`, `cores` and `sockets`
    - Can specify an explicit ordered list of places
    - Place numbering is implementation defined

Affinity support now supports targeting thread binding to specific parallel regions

- Added a new clause to the parallel construct

```
proc_bind(master | close | spread)
```

  - Overrides `OMP_PROC_BIND` environment variable
  - Ignored if `OMP_PROC_BIND` is `false`

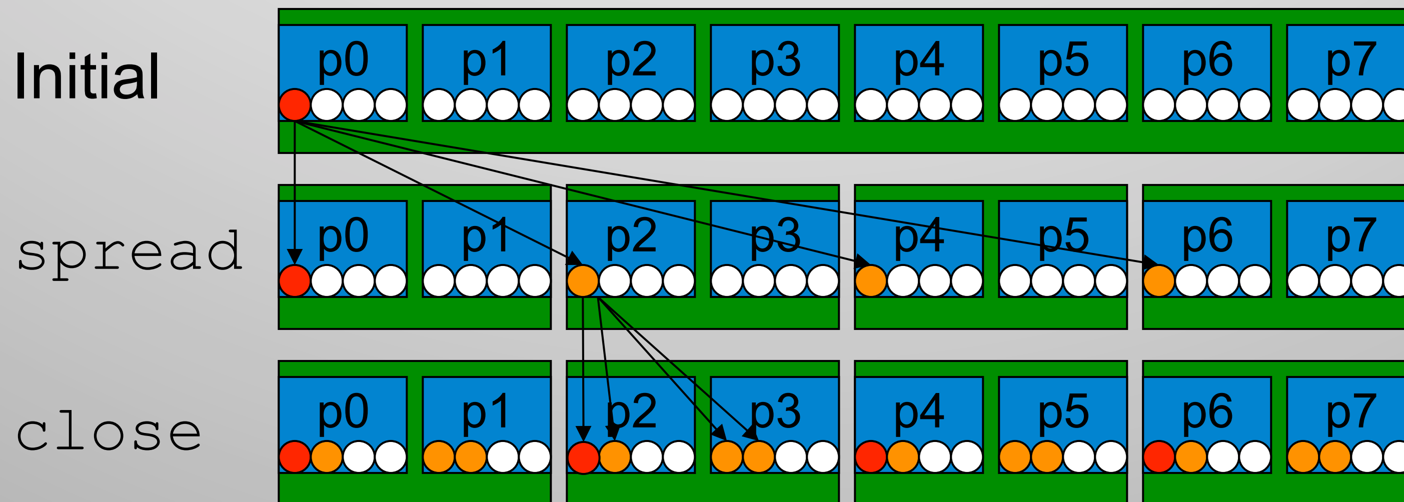
- New run time function to query current policy

```
omp_proc_bind_t omp_get_proc_bind(void);
```

- New policies determine relative bindings
  - Assign threads to same place as `master`
  - Assign threads `close` in place list to parent thread
  - Assign threads to maximize `spread` across places

An example show how to use for nested parallelism of depth two

- Objective: Maximize memory bandwidth of outer parallel region and exploit shared data of inner parallel region
- Solution: Use `spread` on outer region, `close` on inner
  - Can use list (`spread, close`) for `OMP_PROC_BIND`
  - Can use `proc_bind` clause on each region



User Defined Reductions (UDRs) are a major addition in OpenMP 4.0

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

```
#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

```
#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

```
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 `identity` clause not 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

OpenMP 4.0 will include initial support for Fortran 2003

- Added to list of base language versions
- Have a list of unsupported Fortran 2003 features
  - List initially included 24 items (some big, some small)
  - List has already been reduced to 18 items
  - List in specification reflects approximate priority
  - Priorities determined by importance and difficulty
- Strategy: Gradually reduce list
  - Already removed procedure pointers, renaming operators on the `USE` statement, `ASSOCIATE` construct, `VOLATILE` attribute, pointer `INTENT` and structure constructors
  - Hope to remove others in RC2
  - Expect some items will remain unsupported in OpenMP 4.0

4.0 adds the taskgroup construct to support simpler task synchronization

- Adds one easily shown construct

```
#pragma omp taskgroup
{
    create_a_group_of_tasks (could_create_nested_tasks);
}
```

  - Implicit task scheduling point at end of region; current task is suspended until all child tasks generated in the region and their descendants complete execution
  - Similar in effect to a deep `taskwait`
    - 3.1 would require more synchronization, more directives
- More significant tasking extensions planned for RC2
  - Will add concept of task dependence
  - Two forms being considered

OpenMP 3.1 atomic operation additions address an obvious deficiency

- Previously could not capture a value atomically

```
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

```
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
}
```

- Atomic swap in 4.0 performs capture followed by write
- Added seq_cst clause for atomics in 4.0; removes need for flush…

We anticipate that RC2 will address several major topics not in RC1

- Support for accelerators based on TR1 (next talk)
- The `cancel` construct provides initial error model support
  - Very close for parallel and worksharing regions
  - Provides algorithmic advances when applied to tasks
  - Anticipate callbacks for integrated error handling in OpenMP 5.0
- Ongoing work to support Fortran 2003 fully
- Task dependencies extend the OpenMP tasking model
- How to specify subarrays in C
  - Basically done but lack use case in RC1
  - Will be useful for accelerators and task dependencies
- Probably some refinements/extensions to affinity support

We are considering several other topics for OpenMP 5.0 and beyond

- Topics on the table for OpenMP 5.0
  - Support for memory affinity
  - Refinements to accelerator support
  - Transactional memory and thread level speculation
  - Additional task/thread synchronization mechanisms
  - Completing extension of OpenMP to Fortran 2003
  - Interoperability and composability
  - Incorporating tool support
- Help us shape the future of OpenMP
  - Attend IWOMP, become a cOMPunity member
  - Lobby your institution to join the OpenMP ARB

Lawrence Livermore
National Laboratory