OpenMP 6.0 Outlook: TR12 and Beyond

Bronis R. de Supinski
Chair
OpenMP Language Committee

November, 2023
OpenMP 6.0 will be released in November 2024

- TR12 demonstrates appropriate progress for second TR of a major version
- Major new feature targets have been clearly identified and are on track for 2024
  - Free-agent threads significantly change execution model, implementations
  - User-defined induction and *induction* clause expand parallelism support
  - Many significant device support improvements (e.g., *memscope*(*all*)) added or planned
  - Several other additions and improvements planned, including:
    - Rationalization of definition of combined constructs
    - Task dependences between concurrently generated tasks
  - Significant improvements to usability and correctness of specification
  - TR12 includes 153 completed issues, considering over 300 others (2 more already passed)
  - TR13 (final comment draft) will be released in summer 2024
Major new features will characterize OpenMP 6.0

- **Free-agent threads**
  - Support for top-level task parallelism (i.e., explicit `parallel` directive not needed)
  - “Any” thread can execute explicit tasks for which `threadset` clause evaluates is `omp_pool`
  - Adds associated runtime routines, environment variables and ICVs

- **Major improvements for use of a single device**
  - Explicit progress guarantee adopted in TR11
  - Default device and visible devices to simplify control of device use and availability
  - Mechanisms to simplify use of device memory (by providing greater certainty or clarity)
    - New `groupprivate` directive in TR11 is an initial mechanism in this direction
    - Added `selfmap` modifier to ensure no copy is created when possible
    - Unified host and device allocators and added significant cross-device improvements
  - TR12 added `coexecute` directive (i.e., descriptive array language offload support)
Recall the tasking execution model

- Supports unstructured parallelism
  - unbounded loops
  ```c
  while ( <expr> ) {
    ...
  }
  ```
  - recursive functions
  ```c
  void myfunc( <args> ) {
    ...; myfunc( <newargs> ); ...;
  }
  ```

- Example (unstructured parallelism)
  ```c
  #pragma omp parallel
  #pragma omp single
  while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
  }
  ```

- Why are the `parallel` and `single` directives needed?
Recall the tasking execution model

- Supports unstructured parallelism
  - unbounded loops
    ```
    while ( <expr> ) {
      ...
    }
    ```
  - recursive functions
    ```
    void myfunc( <args> ) {
      ...
      myfunc( <newargs> );
    }
    ```

- Why are the **parallel** and **single** directives needed?
  - Otherwise all threads in the team generate (duplicate) tasks

- Example (unstructured parallelism)
  ```
  #pragma omp parallel
  #pragma omp single
  while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
  }
  ```
Recall the tasking execution model

- Supports unstructured parallelism
  - unbounded loops
    ```c
    while ( <expr> ) {
      ...
    }
    ```
  - recursive functions
    ```c
    void myfunc( <args> ) {
      ...;
      myfunc( <newargs> );
    }
    ```

- Example (unstructured parallelism)
  ```c
  #pragma omp parallel
  #pragma omp single
  while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
  }
  ```

- Why are the **parallel** and **single** directives needed?
  - Otherwise all threads in the team generate (duplicate) tasks
  - Only threads in the team may execute tasks
Is restricting tasks to a team good?

- Positive aspects
  - Simplifies resource management
  - Clear semantics with respect to other teams

- Negative aspects
  - Ignores unutilized resources
  - Complicates code structure for task-only programs
Is restricting tasks to a team good?

- Positive aspects
  - Simplifies resource management
  - Clear semantics with respect to other teams

- Negative aspects
  - Ignores unutilized resources
  - Complicates code structure for task-only programs

- Alternative starting in OpenMP 6.0: free-agent threads
  - Unassigned threads in contention group may execute tasks
  - Can provide parallelism in the implicit parallel region
  - Exploits unused resources, common practice of parked threads
Is restricting tasks to a team good?

- Positive aspects
  - Simplifies resource management
  - Clear semantics with respect to other teams

- Negative aspects
  - Ignores unutilized resources
  - Complicates code structure for task-only programs

- Alternative starting in OpenMP 6.0: free-agent threads
  - Unassigned threads in contention group may execute tasks
  - Can provide parallelism in the implicit parallel region
  - Exploits unused resources, common practice of parked threads

Example (no parallel directive needed)

```c
while (elem != NULL) {
    #pragma omp task threadset(omp_pool)
    compute(elem);
    elem = elem->next;
}
```
Some details for free-agent threads

- Existing behavior is preserved by default
  - As if `threadset` clause is specified with value of `omp_team`

```plaintext
#pragma omp task threadset(omp_team)
  structured-block
```

- Tasks are still tied by default so free-agent thread executes the task completely if at all
- Task synchronization (e.g., dependences, `taskwait` and `taskgroup`) unchanged

- Can use environment variables to control ICVs to reserve threads

```plaintext
Setenv OMP_THREADS_RESERVE "structured(2),free_agent(2)"
```

- At least two threads available for structured parallelism, at least two available to act as free-agents
- Minimum for structured parallelism is one (the initial thread)
- Sum of reservations should not exceed `thread-limit-var` ICV
OpenMP 6.0 will include other significant new features

- A more complete set of loop transforming directives
  - TR12 includes **fuse**, **reverse** and **interchange** directives
  - Considering other transformations that include **fission** and **nestify**
  - Can now transform generated loops using the **apply** clause

- Clauses and directives to support generalized induction
  - Capture computation that follows a well-defined sequence across loop iterations
  - Generalizes behavior of **linear clause** and of loop iteration variables
  - Related to reductions, including addition of **declare induction directive**
Extending parallel directive to support complete user control of number of threads

- The parallel directive will accept a new modifier and two “new” clauses

```c
#pragma omp parallel [num_threads(prescriptiveness: nthreads)] \ 
[severity(fatal|warning)][message(msg-string)]
```

- Using strict prescriptiveness requires nthreads to be provided
- Clauses, previously available on error directive, effective with strict if cannot provide nthreads
  - Display msg-string as part of implementation-defined message
  - If severity is fatal execution is terminated
  - If severity is warning message is displayed but execution continues
- Also now allowed to provide a list for nthreads to support nested parallelism
Some other improvements expected in OpenMP 6.0

- May further extend descriptive and prescriptive control
- Removal of features that were deprecated in 5.0, 5.1 or 5.2
- Dependences and affinity for the `taskloop` construct
- Wider use of C++ attribute syntax: Make C++ support “more C++-like”
  - Likely to include improvements for `threadprivate` and `declare target`
  - Also clarified conditions for implicitly declared reduction operators for class types
  - Will also be supported in C
- Adding latest versions of base languages (C23, C++23, Fortran 2023)
- Continuing to extend support for tool interfaces
Some other improvements expected in OpenMP 6.0

- Expect to define combined constructs based on properties (a priority)
- Strengthening task-oriented execution changes begun in OpenMP 3.0
- Extending specification improvements begun in OpenMP 5.2
  - Includes making all clauses accept arguments
  - All clauses will take a directive name modifier for better control of combined constructs
- Immediate focus is several high priority, nearly completed issues
  - More single device and tasking improvements
  - Better control of number of threads
  - Simpler, broader user control of defaults
Things likely to be deferred to beyond 6.0

- True support for using multiple devices
  - Device-to-device scoping support for atomic and other memory operations
  - Support for bulk launch
  - Support to update data on multiple devices (broadcast/multicast, other collectives)
  - Support for work distribution across devices
  - Considering relaxing restrictions on nested target regions

- Efficient use of multiple compilation units (i.e., support for efficient IPO)

- Characterizing loop-based work distribution constructs as transformations

- Support for pipelining, data-flow, other parallelization models

- Support for event-based parallelism