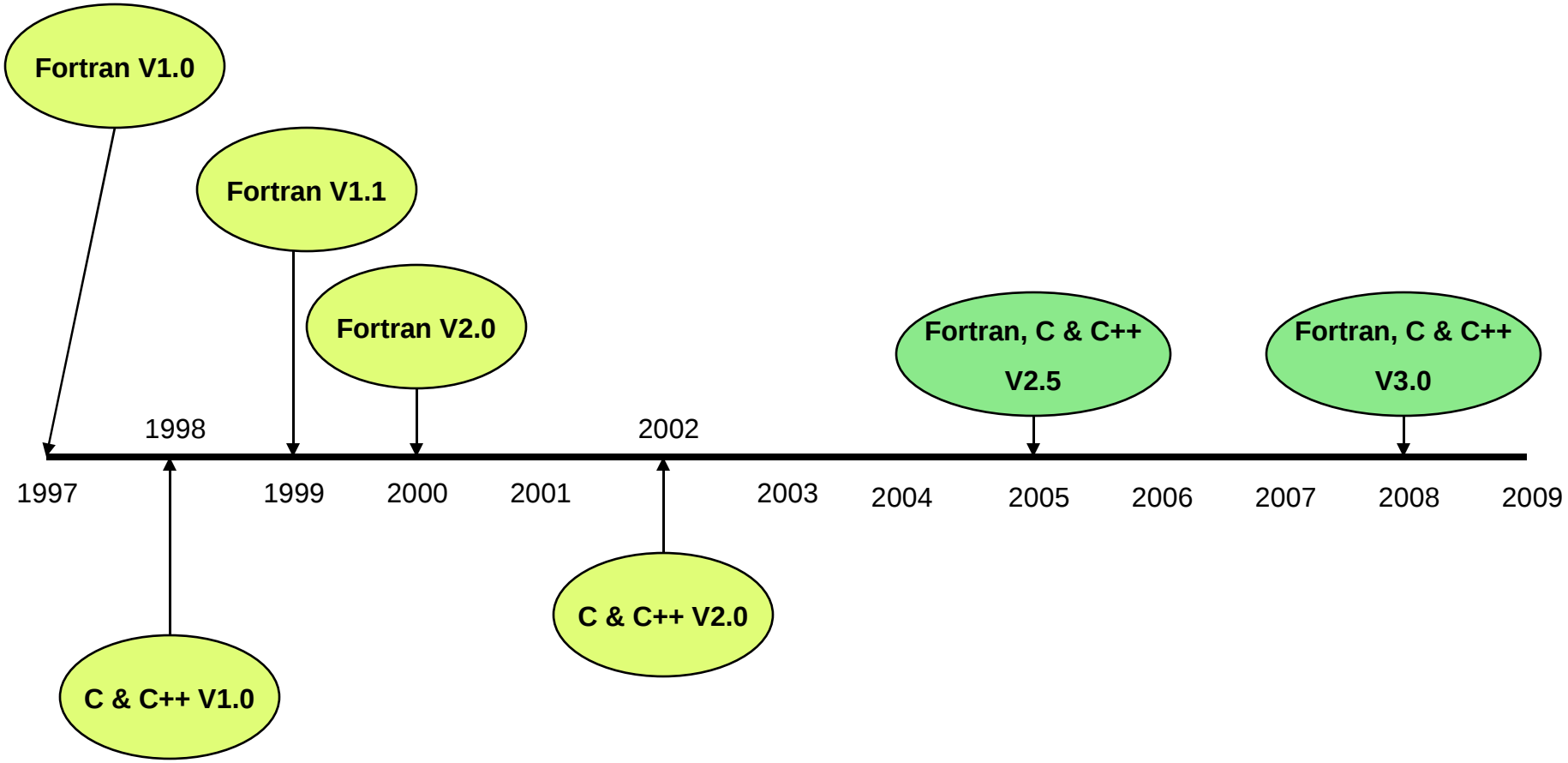
OpenMP API 3.0

Kelvin Li
(kli@ca.ibm.com)

OpenMP API

- **"… a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C, C++ and Fortran programs"**
- **maintained by the OpenMP Architecture Review Board (ARB)**
- **ARB members:**
  - permanent members: AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, The Portland Group (STMicroelectronics), SGI, Sun Microsystems, Microsoft
  - auxiliary members: ASC/LLNL, cOMPunity, EPCC, NASA, RWTH Aachen University

A brief history of OpenMP API



OpenMP API support

- **XL Fortran for AIX V12.1 (7/2008 GA)**
  XL Fortran for Linux V12.1 (10/2008 GA)
  - full support for V2.5 and partial support for V3.0
- **XL C/C++ for AIX V10.1 (7/2008 GA)**
  XL C/C++ for Linux V10.1 (10/2008 GA)
  - full support for V3.0

Task directive

- **irregular parallelism**
- **a task has**
  - code to execute
  - a data environment (it owns its data)
  - an assigned thread executes the code and uses the data
- **two activities: packaging and execution**
  - each encountering thread packages a new instance of a task (code and data)
  - some thread in the team executes the task
- **task construct**
  - defines an explicit task
  - directive: task / end task
  - clause: if, untied, private, firstprivate, default, and shared

Task directive (cont'd)

- **generate independent works with task construct**

```
!$OMP parallel
!$OMP single
      do while (...)
!$OMP task
        call process(p)
!$OMP end task
      enddo
!$OMP end single
!$OMP end parallel
```

Task directive (cont'd)

- an example from the spec

```
      recursive integer funciton fib(n)
      integer :: n, i, j
      if (n .lt. 2) then
        fib = n
      else
!$OMP task shared(i)
        i = fib(n-1)
!$OMP end task
!$OMP task shared(j)
        j = fib(n-2)
!$OMP end task
!$OMP taskwait
        fib = i + j
      endif
      end function
```

Task directive (cont'd)

- **task switching**
  - the act of a thread to switch from executing one task to another task
- **task scheduling point**
  - a point during the execution of the current task region at which it can be suspended to be resumed later; or the point of task completion, after which the executing thread may switch to a different task region
  - e.g. encountered task constructs, encountered taskwait constructs

Task directive (cont'd)

- **why?**
  - ```
    !$omp parallel
    !$omp single
          do i=1, 1000000
    !$omp task
            call process(items(i))
    !$omp end task
          enddo
    !$omp end single
    !$omp end parallel
    ```
  - too many tasks generated and unassigned; the "task pool" becomes very large that may exceed resource limit
  - the thread that generates tasks (t1) is allowed to suspend the task generation and
    - execute the unassigned task (draining the "task pool"); or
    - execute the encountered task (could be very cache-friendly)
  - when the number of unassigned tasks is reduced, t1 will resume the task generation
    ```

Task directive (cont'd)

- **what is the untied clause?**
  - by default, without untied clause, a task is tied to the thread that starts the execution (i.e. suspend and resume by the same thread)
  - untied task is a task that can be suspended by one thread and resumed by any thread in the team (not tied to any thread)
  - using the same example (adding the untied clause to the task directive)
    - when t1 suspended the task generation but execute a long task
    - when all the unassigned tasks are finished by other threads
    - another thread can resume the task generation (that task is not tied to any thread)

Task directive (cont'd)

- **taskwait construct**
  - specifies a way to wait on the completion of child tasks generated since the beginning of the current task
  - encountering task suspends at the point of the directive until all children tasks created within the encountering task up to this point are complete

Loop collapse

- **collapse perfect nested loops**
- **clause: collapse(n), where *n* specifies how many loops are associated to the loop construct (by default the loop that follows the construct)**
-

```
!$OMP do collapse(2) private(i,j,k) ! associated
                                    ! two outer loops
      do k=kl, ku, ks
        do j=jl, ju, js
          do i=il, iu, is
            call sub(a,i,j,k)
          enddo
        enddo
      enddo
```

Stack size control

- **controls the size of the stack for threads**
- **but not control the stack size for the initial thread (i.e. the master thread in a team)**
- **envirnoment variable: OMP_STACKSIZE=*size*|*size* B|*size* K|*size* M|*size* G**

Thread wait policy

- **a hint about the desired behavior of waiting threads during the execution of an OpenMP program**
- **environment variable: OMP_WAIT_POLICY=ACTIVE|PASSIVE**
  - ACTIVE – waiting threads be active (i.e. consume processor cycles, while waiting)
  - PASSIVE – waiting threads mostly be passive (i.e. not consume processor cycels, while waiting)

SCHEDULE kind - AUTO

- **specifies that the compiler/runtime can choose any possible mapping of iterations to threads and may be different in different loops**
- **environment variable:**
  OMP_SCHEDULE=static | dynamic | guided | auto
- **runtime routines:**
  - **[C/C++]**

```
omp_set_schedule(omp_sched_t sched, int arg)
omp_get_schedule(omp_sched_t *sched, int *arg)
```

[Fortran]

```
subroutine omp_set_schedule(kind, modifier)
  integer(kind=omp_sched_kind) :: kind
  integer ::  modifier

subroutine omp_get_schedule(kind, modifier)
  integer(kind=omp_sched_kind) :: kind
  integer :: modifier
```

Preserving private variable

- **remove possibility of reusing the storage of the original variable for private variables**
-

```
      x = 10
!$OMP parallel private(x)
      ...
      ! unspecified if reference original x
      ... = x
!$OMP end parallel
      ! x is defined after the region
```

Allocatable arrays

- **allow allocatable arrays on firstprivate, lastprivate, reduction, copyin and copyprivate clauses**
- **relax the requirement of having the allocatable arrays to be “not currently allocated” on entry to and on exit from the construct**

Allocatable arrays (cont'd)

- **if it is allocated, the private copy will have an initial state of allocated with the same array bounds**
-
```
      integer, allocatable :: arr(:,:)
      allocate(arr(500,4))
!$OMP parallel private(arr)
      ! private arr is allocated with the same
      ! bound and shape but not initialized
      ...
!$OMP end parallel
```
- **not yet supported in XLF**

STATIC schedule

- **modify STATIC schedule to allow safe use of NOWAIT**
- **ensure the same assignment of iteration numbers to threads will be used in two consecutive worksharing loops**
- ```
  !$OMP do schedule(STATIC)
        do i=1, N
          a(i) = ...
        enddo
  !$OMP end do nowait
  !$OMP do schedule(STATIC)
        do i=1, N
          ... = a(i)
        enddo
  ```
  ```

Nesting support

- **define maximum number of OpenMP threads a program can have**
  - runtime routine
    - omp_get_thread_limit – returns the maximum number of OpenMP threads available to the program
  - environment variable
    - OMP_THREAD_LIMIT – sets the number of OpenMP threads to use for the whole OpenMP program

Nesting support (cont'd)

- **define the max depth of nested active parallel regions**
  - runtime routine
    - omp_set_max_active_levels – limits the number of nested active parallel regions
    - omp_get_max_active_levels – returns the maximum number of nested active parallel regions
  - environment variable
    - OMP_MAX_ACTIVE_LEVELS – controls the maximum number of nested active parallel regions

Nesting support (cont'd)

- **nesting information**
  - runtime routine
    - omp_get_level – returns the number of nested parallel reigions enclosing the task that contains the call
    - omp_get_ancestor_thread_num – returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread
    - omp_get_team_size – returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs
    - omp_get_active_level – returns the number of nested, active parallel regions enclosing the task that contains the call

Miscellanous features

- **[C/C++] allow unsiged int as the for-loop iteration variable (only signed int is listed in 2.5)**
- **[C++] random access iterators can be used as loop iterators in loops associated with a loop construct**
- **[C++] static class members variables can appear in a threadprivate variable**
- **[C++] where constructors/destructors are called, how threadprivate objects should be initialized**

Miscellanous features (cont'd)

- **multiple internal control variables**
  - only one global copy of internal variable in a program in V2.5
  - define some internal control variables in per thread base
    - *dyn-var* (ref: OMP_DYNAMIC)
    - *nest-var* (ref: OMP_NESTED)
    - *nthreads-var* (ref: OMP_NUM_THREADS)
- **[Fortran] default clause allows firstprivate**

More information …

- **OpenMP API**
  - http://www.openmp.org
- **about IBM XL compilers:**
  - http://www-01.ibm.com/software/awdtools/fortran/
  - http://www-01.ibm.com/software/awdtools/xlcpp/