Pyjama Syntax Quick Reference Card

March 6, 2017

OpenMP Application Program Interface (API) is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

Pyjama is OpenMP Java language implementation. supports JVM shared-memory parallel programming on all architectures, from Unix platforms to Windows NT platforms to Android mobile platforms.

[x.x] after directives or runtime routines refers to this feature is available from version x.x.

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A structured-block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

Parallel [1.2]

The parallel construct forms a team of threads and starts parallel execution.

//#omp parallel [clause[[,]clause] ...]
structured-block
clause:
if(scalar-expression)
num_threads(integer-expression)
default(none | shared)
~~private(list)~~[1]
firstprivate(list)
shared(list)
reduction(operator|operation function: list)
~~copyin(list)~~[2]

Customize reduction operation:

Pyjama support user defined reduction operation. Programmers can define their own reduction operation by implementing reduction methods.

a function name comfort with <T> reductionFunctionName(<T> var1, <T> var2)

```
public class Point{
    private int x;
    private int y;
    public static void main (String[] argc) {
        xPoint p = new xPoint(0,0);
        //#omp parallel reduction(xPointReduction:p)
        {
            //parallel region code
        }
    }
    public Point PointReduction(Point p1, Point p2) {
        // user defined reduction operation here
```

[1] In Pyjama, in light of good object-oriented programming style, we discourage use of private data clause. Users may define new variables inside parallel region when thread-private variables are required.

[2] We eliminate threadprivate directive in Pyjama, since it break design rules of object-oriented design using global variables. So copyin data clause is banned as well.

```
        ...
    }
    ...
}
```

Loop [1.2]

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

//#omp for [clause[[,]clause] ...]
for-loops
clause:
~~private(list)~~
firstprivate(list)
lastprivate(list)
reduction(operator: list)
~~copyin(list)~~
schedule(kind[, chunk size])
ordered
nowait
kind:

• static: Iterations are divided into chunks of size chunk size. Chunks are assigned to threads in the team in round-robin fashion in order of thread number.

• dynamic: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be distributed.

• guided: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated chunk size as chunks are scheduled.

• auto: The decision regarding scheduling is delegated to the compiler and/or runtime system.

• runtime: The schedule and chunk size are taken from the run-sched-var ICV.

Sections [1.2]

The sections construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

//#omp sections [clause[[,] clause] ...]
{
[//#omp section]
structured-block
[//#omp section]
structured-block
...
}
clause:
~~private(list)~~
firstprivate(list)
lastprivate(list)
reduction(operator: list)
~~copyin(list)~~
nowait

Single [1.2]

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

//#omp single [clause[[,]clause] ...]

structured-block
clause:
~~private(list)~~
~~copyprivate(list)~~
firstprivate(list)
nowait

Parallel Loop [1.2]

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

//#omp parallel for [clause[[,]clause] ...]
for-loop
clause:

Any accepted by the parallel or for directives, except the nowait clause, with identical meanings and restrictions.

Simple Parallel Loop Example The following example demonstrates how to parallelize a simple loop using the parallel loop construct.

```
void simple(int n, oat *a, oat *b)
{
    int i;
    //#omp parallel for shared(i, n, a, b)
    for (i=1; i<n; i++){
        /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
    }
}
```

Parallel Sections [1.2]

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other statements.

//#omp parallel sections [clause[[,]clause] ...]
{
[//#omp section]
structured-block
[//#omp section]
structured-block
... }

clause: Any of the clauses accepted by the parallel or sections directives, except the nowait clause, with identical meanings and restrictions.

Master [1.2]

The master construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

//#omp master
structured-block

Critical [1.2]

The critical construct restricts execution of the associated structured block to a single thread at a time.

//#omp critical [(name)]
structured-block

Barrier [1.2]

The barrier construct specifies an explicit barrier at the point at which the construct appears.

//#omp barrier

Flush [1.2]

The flush construct executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

//#omp flush [(list)]

Ordered [1.2]

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

//#omp ordered
structured-block

Freeguithread [1.3]

The freeguithread directive is designed for GUI application, which make current GUI EDT thread free from executing current code and make it enable to process next event message.

//#omp freeguithread [parallel [,[for|sections [clause[[,]clause] ...]]]
structured-block

Gui [1.3]

The gui is designed directive is used for GUI application. Code block after gui directive will be executed in GUI event dispatch thread. Clause nowait make current thread directly execute next part of code instead of waiting EDT thread finish executing this code block. This directive usually is used in freeguithread to notify EDT thread execute following block.

//#omp gui
structured-block
clause:
nowait

Virtual Target [1.5.4]

The virtual target concept is designed for supporting asynchronization model for event-driven programming pattern. For detailed usage, please refer to the publication of Pyjama.

//#omp target virtual [clause[[,]clause] ...]
structured-block
clause:
asynchronous-property-clause
data-handling-clause
if-clause
where asynchronous-property-clause is one of the following:
nowait name_as(name-tag) await
where data-handling-clause is one of the following:
firstprivate(list) shared(list)
and if-clause is:
if(scalar-expression)

Async Call [1.5.4]

The async-call directive is designed for invoking a function in an asynchronous way.
//#omp async-call asynchronous-property-clause(function-declaration [[,]function-declaration]...)
structured-block
where asynchronous-property-clause is one of the following:
nowait name_as(name-tag) await

Runtime Library Routines

Parallel Execution Environment Routines [1.2]

Execution environment routines affect and monitor threads, processors, and the parallel environment.

- int Pyjama.omp_get_num_threads()

  Returns the number of threads in the current team.

- void Pyjama.omp_set_num_threads(int num_threads)

  Affects the number of threads used for subsequent parallel regions that do not specify a num threads clause.

- int Pyjama.omp_get_thread_num()

  Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

- boolean Pyjama.omp_in_parallel()

  Returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false.

- int Pyjama.omp_get_max_threads()

  Returns maximum number of threads that could be used to form a new team using a parallel construct without a num threads clause.

- int Pyjama.omp_get_num_procs()

  Returns the number of processors available to the program.

- int Pyjama.omp_get_dynamic()

  Returns the value of the dyn-var ICV, determining whether dynamic adjustment of the number of threads is enabled or disabled.

- void Pyjama.omp_set_dynamic()

  Enables or disables dynamic adjustment of the number of threads available by setting the value of the dyn-var ICV.

- void Pyjama.omp_set_nested(boolean nested)

  Enables or disables nested parallelism, by setting the nest-var ICV.

- boolean Pyjama.omp_get_nested()

  Returns the value of the nest-var ICV, which determines if nested parallelism is enabled or disabled.

Asynchronous Execution Environment Routines [1.5.4]

Execution environment routines affect and support the functionality for event-driven offloading and asynchronization.

- void omp_register_as_virtual_target(String targetName)

  Register current thread as a virtual target, with name of targetName.

- void omp_create_virtual_target(String targetName)

  Create a single-thread virtual target, with name of targetName.

- void omp_create_virtual_target(String targetName, int n)

  Create an n thread virtual target, with name of targetName.

- String omp_get_target_name()

  If current thread belongs to a virtual target, return the target name. If not, return null.

- void omp_set_platform(Platform platform)

  Set current GUI programming framework, choosing from one of the supporting platforms: Android, JavaFX and Swing.

- String omp_get_platform()

  Return the platform identifier current program is using.

Lock Routines [1.2]

These routines initialize an OpenMP lock.

- void Pyjama.omp_init_lock(omp lock t *lock)
- void Pyjama.omp_init_nest_lock(omp nest lock t *lock)

These routines ensure that the OpenMP lock is uninitialized.

- void Pyjama.omp_destroy_lock(omp lock t *lock)
- void Pyjama.omp_destroy_nest_lock(omp nest lock t *lock)

These routines provide a means of setting an OpenMP lock. This calling task region is suspended until the lock is set.

- void Pyjama.omp_set_lock(omp lock t *lock)
- void Pyjama.omp_set_nest_lock(omp nest lock t *lock)

These routines provide a means of unsetting an OpenMP lock.

- void Pyjama.omp_unset_lock(omp lock t *lock)
- void Pyjama.omp_unset_nest_lock(omp nest lock t *lock)

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

- int Pyjama.omp_test_lock(omp lock t *lock)
- int Pyjama.omp_test_nest_lock(omp nest lock t *lock)

Timing Routines [1.2]

Timing routines support a portable wall clock timer.

- double Pyjama.omp_get_wtime()

  Returns elapsed wall clock time in seconds.

- double Pyjama.omp_get_wtick()

  Returns the precision of the timer used by omp get Pyjama.omp_get_wtime().

Environment Variables [1.2]

- OMP_SCHEDULE
- OMP_NUM_THREADS
- OMP_DYNAMIC
- OMP_NESTED