OpenMP Directives and Runtime

Piotr Luszczek
Restrictions on OpenMP Loops

```
#pragma omp for
for (index = <START> ; index < END>;                        )

index++
++index
index--
--index
index += inc
index -= inc
index = index + inc
index = inc + index
index = index - inc
```

Changes iteration count

Allowed (because it does not change iteration count)

```
<
<=
>
```

Not allowed

```
break
exit()
go to
return
```

Changes iteration count
Working Around Restrictions

- The restrictions are in place so that the OpenMP runtime can compute the right schedule for all threads
  - Complicated loops require complicated math or cannot be computed at all
  - STL containers often cannot easily know their size to compute a balanced schedule for threads
- Example: go over powers of two
  - Bit shifting loop:
    ```c
    for (i = 1 << 31 ; i != 0 ; i >>= 1) {
    }
    ```
  - replace with:
    ```c
    // compiler “sees” that there are 31 iterations
    for (j = 31 ; j > 0 ; --j) {
        i = 1 << j;
    }
    ```
Scoping of OpenMP Directives

```c
int main(void) {
    /* ... */
    #pragma omp parallel
    { /* ... */
        #pragma omp for
        for (int i = 0; i < N; ++i) {
            /* ... */
            fnct1();
        }
        /* ... */
        fnct2();
        /* ... */
    }
    /* ... */
}
```

```c
void fnct1() {
    /* ... */
    #pragma omp critical
    /* ... */
}
void fnct2() {
    /* ... */
    #pragma omp sections
    { /*...*/
    }
    /*...*/
}
```

- **static extent:** parallel and for directives in the same scope
- **dynamic extent:** parallel region encloses orphaned directives at runtime
- **orphaned directive:** parallel region is not visible but it must be present outside the function or the result is unspecified
Nested Parallel Regions

- Use the `omp_get_nested()` library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
  - The `omp_set_nested()` library routine
  - Setting of the `OMP_NESTED` environment variable to TRUE
- If not supported by your OpenMP implementation:
  - A parallel region nested within another parallel region results in the creation of a new team, consisting of one thread.
#pragma omp parallel for shared(k) private(i)
for (int i = 0; i < 20; ++i)
    printf("%d\n", i+k);

for (int i0 = 0; i0 < 10; ++i0)
    printf("%d\n", i0+k);

for (int i1 = 10; i1 < 20; ++i1)
    printf("%d\n", i1+k);

#pragma omp barrier

Barrier is implicitly inserted for correctness and convenience unless nowait is used when opening a region
**Variable Scope: firstprivate and lastprivate**

```c
j = 13;
#pragma omp parallel for private(i) firstprivate(j)
for (int i = 0; i < 20; ++i)
    printf("%d\n", i+j);
```

Thread 0 (master)

```c
// i0 is uninitialized
j0 = j; // copy from master thread
for (int i0 = 0; i0 < 10; ++i0)
    printf("%d\n", i0+j0);
```

Thread 1

```c
// i1 is uninitialized
j1 = j; // copy from master thread
for (int i1 = 10; i1 < 20; ++i1)
    printf("%d\n", i1+j1);
```

`lastprivate` can be used for copying private variable out of the “last” thread into the master thread.
#ifdef _OPENMP
#include <omp.h>
#endif

omp_set_num_threads(13); // 13 threads for parallel regions

printf("%d\n", omp_in_parallel()); // in parallel region? returns 0

#pragma omp parallel
{
    printf("%f\n", omp_get_wtime()); // wall clock time with fractions
    // for sub-second intervals: 1.25
    printf("%d\n", omp_in_parallel()); // in parallel region? returns 1
    printf("%d\n", omp_get_num_threads()); // number of active threads: 13
    printf("%d\n", omp_get_thread_num()); // thread number (master is 0)
    printf("%d\n", omp_get_num_procs()); // number of processors (cores)
    printf("%f\n", omp_get_wtime()); // wall clock time
}
#pragma omp parallel for
for (int i=0; i<N; ++i) {
    if (omp_get_thread_num() == 7) // print 7 if omp_get_num_threads()>7
        printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

    #omp critical // each thread prints its number in unspecified order
    printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

    #omp single // only one thread prints its number
    printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

    #omp master
    printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() ); // 0
}
omp_lock_t L; // OpenMP lock data type
omp_init_lock( &L ); // initialize the lock

#pragma omp parallel for
{
    omp_set_lock( &L ); // acquire lock
    // . . . // critical section with mutual exclusion
    omp_unset_lock( &L ); // relinquish lock
    omp_test_lock( &L ); // test if lock is available
}
omp_destroy_lock( &L ); // dealloc lock’s resources
Scheduling for Loops

- `#pragma omp parallel for schedule(<T> [, <CHUNK>])`
- Not all loops benefit from the same type parallelism and/or are load balanced: `for (N=2; N<100; ++N) matmatmul(N, a, b, c)`
- `schedule(static)` – each thread gets `#iters / THREADS`
- `schedule(static, C)` – first thread gets `C` iterations, second thread gets the next `C` iterations, ...
- `schedule(dynamic)` – first thread gets an iteration and then gets another available iteration when its finished
- `schedule(dynamic, C)` – first thread gets `C` iterations, ...
- `schedule(guided)` – chunks exponentially decrease to 1
- `schedule(guided,C)` – chunks exponentially decrease to `C`
- `schedule(runtime)`
  - export/setenv OMP_SCHEDULE “static,1”
OpenMP Schedules’ Details

- **static**: Complete execution of the chunk
- **static,chunk**: Complete execution of the chunk and then a new chunk
- **dynamic**: Execution of 75% of the chunk and then another 75% of the chunk
- **dynamic,chunk**: Execution of 75% of the chunk and then a new chunk
- **guided**: Execution of 75% of the chunk, then 75% of another chunk
Collapsing Nested Loops

- Multiple loop nests are supported in OpenMP
  - Compiler automatically changes the code into single loop
  - The standard schedule types work on the reorganized loop
    - More opportunities for parallelism

- Consider matrix multiplication:
  - for (int \(i = 0; i < N; ++i\))
    for (int \(j = 0; j < N; ++j\))
      for (int \(k = 0; k < N; ++k\))
        \(C[i][j] += A[i][k] \times B[k][j]\);

- With OpenMP:
  - #pragma omp parallel for collapse(2)
    for (int \(i = 0; i < N; ++i\))
      for (int \(j = 0; j < N; ++j\))
        for (int \(k = 0; k < N; ++k\))
          \(C[i][j] += A[i][k] \times B[k][j]\);
OpenMP Memory Model

- OpenMP provides a relaxed-consistency and temporary view of thread memory
- Threads may cache their data and are not required to maintain exact consistency with the main memory all of the time
  - For efficiency, they rarely synchronize memory state
  - x86 hardware provides strong memory consistency
  - ARM and IBM processors have weaker consistency in hardware
- When all threads view a shared variable identically:
  - The programmer must ensure that the variable is FLUSHed by all threads as needed
  - FLUSH clause may be added to some directives
  - Using FLUSH creates a memory fence
    - It may be expensive because the compiler has to optimize less aggressively
    - It may be expensive because the hardware must synchronize cache memories