CME213/ME339
Lecture 15

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Shared variables: relaxed-consistency shared-memory model

- The OpenMP API provides a relaxed-consistency, shared-memory model.
- Each thread is allowed to have its own *temporary* view of the memory. This corresponds to the fact that the hardware relies on many optimizations and can store a variable in many different memories: machine registers, cache, or other local storage, between the thread and the memory.
- This temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable.

Synchronization: flush

- A tricky concept.
- As explained previously, threads keep a “local” copy or view of variables. This is important for optimization.
- There are cases however where threads need to write to and read from shared variables.
- In that case a flush is required to make sure the code executes correctly.
- `flush` makes a threads temporary view of memory consistent with memory:
    1. if a thread modifies a shared variable, a flush (after) will result in writing to memory
    2. if a thread reads a variable, a flush (before) will make sure that the value is up to date

Execution of a flush region affects the memory and the temporary view of memory of only the thread that executes the region.

It does not affect the temporary view of other threads.

Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the encountering threads flush operation.

```
#pragma omp flush [(list)]
```

Critical section

- Only one thread may enter that section.
- This can be very useful before a `printf` or `cout` otherwise the printouts from different threads can become interleaved.
- An optional name can be given: in that case, threads are restricted from entering any critical section with the same name.

```
#pragma omp critical [(name)]
```

Example

```
int main() {
    int i;
    double answer, res;
    answer = 0.0;
    #pragma omp parallel for private (res)
    for (i=0;i<N;i++) {
        res = big_comp(i);
        # pragma omp critical
        combine(answer,res);
    }
    printf("The answer to my complicated problem is %f\n",
           answer);
}
```

Atomics

An atomic directive provides a similar functionality but is more restrictive and can lead to better performance than `critical`.

```
#pragma omp atomic
  expression-stmt
```

where `expression-stmt` is something like

```
x binary_operation = expr
x++
++x
x--
--x
```

`binary_operation` is `+ * - / & ^ | << >>`

Example

```
float work1(int i) { return 1.0 * i; }
float work2(int i) { return 2.0 * i; }
void atomic_example(float *x, float *y, int *index, int n) {
  int i;
  #pragma omp parallel for shared(x, y, index, n)
  for (i=0; i<n; i++) {
    #pragma omp atomic update
    x[index[i]] += work1(i);
    y[i] += work2(i);
}}
```

No need to wait

- Parallel constructs typically have a synchronization point at the end.
- In some cases, it is possible for threads to continue executing past the end of the construct if the calculations that follow do not require results from the previous calculation.
- The `nowait` clause can be used for this purpose. This can be used with `for`, `sections`, and `single`.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < nmax; i++)
    if (isEqual(name, current_list[i])
      processCurrentName(name);
  #pragma omp for
  for (i = 0; i < mmax; i++)
    if (isEqual(name, past_list[i])
      processPastName(name);
}
```

Barriers

Explicit synchronization is possible using a barrier.

```
#pragma omp barrier
```

A barrier implies a flush as well.

Ordered for loops

- There are cases where certain statements in a for loop need to executed in the same order as the serial code. However there might be still opportunities for parallel execution in other parts of the loop block.
- The `ordered` directive serves that purpose.

```
cumul_sum[0] = some_massive_calculation(0);
#pragma omp parallel for ordered
for (i = 1; i < n; i++) {
  /* processing for list[i] */
  list[i] = some_massive_calculation(i);
  #pragma omp ordered
  cumul_sum[i] = cumul_sum[i-1] + list[i];
}
```

Locks

For most programmers, these “high-level” constructs are sufficient.

In some cases a finer level of control over synchronization is required or the overhead of using some of the previous constructs may be too large.

Low-level synchronization functions based on locks are also provided by the language.

Schedule clause

This gives some measure of control over the scheduling of loop iterations onto threads.

```
#pragma omp for schedule(static [,chunk])
```

If `chunk` is specified, the loop is broken into blocks of size `chunk`. Blocks are assigned to threads in a round-robin fashion.

If `chunk` is not specified, the loop is broken into blocks of approximately equal sizes and each thread is assigned a single block.

Schedule: dynamic

```
#pragma omp for schedule(dynamic [,chunk])
```

If chunk is specified, the loop is broken into blocks of size chunk. Blocks are assigned to threads dynamically, that is “on demand.”

This is useful when blocks take a varying amount of computational time. The overhead of this construct is greater.

If chunk is not specified, the block size is set to 1.

Variants

```
#pragma omp for schedule(guided [,chunk])
```

This is a variation on dynamic where large block sizes are chosen first and then progressively reduced. This allows reducing the cost of scheduling.

```
#pragma omp for schedule(runtime)
```

The schedule is specified at run time by `OMP_SCHEDULE`.

Matrix-matrix product

```
int chunk = 10;
#pragma omp parallel num_threads(8) private(tid,i,j,k)
{

  tid = omp_get_thread_num();

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations ***/
  #pragma omp critical
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++) {
    #pragma omp critical
    printf("Thread=%d did row=%d\n",tid,i);
    for (j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

Runtime library

- `omp_set_num_threads()`: requests that the OS provide that number of threads in subsequent parallel regions.
- `omp_get_num_threads()`: number of threads in the current team of threads.
- `omp_get_thread_num()`: ID of thread
- `omp_get_num_procs()`: number of processors available to execute the threaded program.