ECE459: Programming for Performance Winter 2015

Lecture 18 — February 13, 2015

Patrick Lam *version 1*

Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++) total += array[i];
```

What is the appropriate scope for `total`? We want each thread to be able to write to it, but we don't want race conditions. Fortunately, OpenMP can deal with reductions as a special case:

```
#pragma omp parallel for reduction (+:total)
```

specifies that the `total` variable is the accumulator for a reduction over `+`. OpenMP will create local copies of `total` and combine them at the end of the parallel region.

Accessing Private Data outside a Parallel Region

A related problem with private variables is that sometimes you need access to them outside their parallel region. Here's some contrived code.

```
int data=1;
#pragma omp parallel for private(data)
for (int i = 0; i < 100; i++)
  printf ("data=%d\n", data);
```

Since `data` is private, OpenMP is not going to copy in the initial value `1` for it, so you'll get an undefined value. To make OpenMP initialize private variables with the master thread's values for those variables, use `firstprivate(data)`. Conversely, to get a value out from the (sequentially) last iteration of the loop, use `lastprivate(data)`.

Thread-private Data. A variant on `private` is `threadprivate`. Thread-private data is also local to each thread. The main difference is that `private` is for transient data, typically local variables, declared at the start of a region, while `threadprivate` is for persistent data, e.g. declared at file scope, which lives beyond a single parallel region. Instead of `firstprivate(local)` for `private(local)`, use `copyin(global)` for `threadprivate(global)`.

```
#include <omp.h>
#include <stdio.h>

int tid, a, b;

#pragma omp threadprivate(a)
```

```
int main(int argc, char *argv[])
{
    printf("Parallel #1 Start\n");
    #pragma omp parallel private(b, tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    printf("Sequential code\n");
    printf("Parallel #2 Start\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    return 0;
}
```

This yields something like the following output:

```
% ./a.out
Parallel #1 Start
T6: a=6, b=6
T1: a=1, b=1
T0: a=0, b=0
T4: a=4, b=4
T2: a=2, b=2
T3: a=3, b=3
T5: a=5, b=5
T7: a=7, b=7
Sequential code
Parallel #2 Start
T0: a=0, b=0
T6: a=6, b=0
T1: a=1, b=0
T2: a=2, b=0
T5: a=5, b=0
T7: a=7, b=0
T3: a=3, b=0
T4: a=4, b=0
```

Collapsing Loops. Usually, when you have nested loops, it's best to parallelize the outermost loop. Why?

Sometimes, however, that just doesn't work out. The main issue is that the outermost loop may have too low a trip count to be worth parallelizing. You could instead parallelize the inner loop by putting the pragma just before the inner loop, but then you pay more overhead than you need to. OpenMP therefore supports *collapsing* loops, which creates a single loop performing all the iterations of the collapsed loops. Consider:

```
#include <math.h>
int main() {
  double array[2][10000];
  #pragma omp parallel for collapse(2)
  for (int i = 0; i < 2; i++)
    for (int j = 0; j < 10000; j++)
      array[i][j] = sin(i+j);
  return 0;
}
```

Parallelizing the outer loop only enables the use of 2 threads. Parallelizing both loops together enables the use of up to 20,000 threads, although the loop body is too small for that to be worthwhile.

Where have you seen something like a manually collapsed loop?

Better Performance Through Scheduling

OpenMP tries to guess how many iterations to distribute to each thread in a team. The default mode is called *static scheduling*; in this mode, OpenMP looks at the number of iterations it needs to run, assumes they all take the same amount of time, and distributes them evenly. So for 100 iterations and 2 threads, the first thread gets 50 iterations and the second thread gets 50 iterations.

This assumption doesn't always hold; consider, for instance, the following (contrived) code:

```
double calc(int count) {
  double d = 1.0;
  for (int i = 0; i < count*count; i++) d += d;
  return d;
}

int main() {
  double data[200][100];
  int i, j;
  #pragma omp parallel for private(i, j) shared(data)
  for (int i = 0; i < 200; i++) {
    for (int j = 0; j < 200; j++) {
      data[i][j] = calc(i+j);
    }
  }
  return 0;
}
```

This code gives sublinear scaling, because the earlier iterations finish faster than the later iterations, and the program needs to wait for all iterations to complete.

Telling OpenMP to use a *dynamic schedule* can enable better parallelization: the runtime distributes work to each thread in chunks, which results in less waiting. Just add `schedule(dynamic)` to the pragma. Of course, this has more overhead, since the threads need to solicit the work, and there is a potential serialization bottleneck in soliciting work from the single work queue.

The default chunk size is 1, but you can specify it yourself, either using a constant or a value computed at runtime, e.g. `schedule(dynamic, n/50)`. Static scheduling also accepts a chunk size.

OpenMP has an even smarter work distribution mode, `guided`, where it changes the chunk size according to the amount of work remaining. You can specify a minimum chunk size, which defaults to 1. There are also two meta-modes, `auto`, which leaves it up to OpenMP, and `runtime`, which leaves it up to the `OMP_SCHEDULE` environment variable.

Beyond `for` Loops: OpenMP parallel sections and tasks

The part of OpenMP we've seen so far has been strictly less powerful than pthreads (but harder to misuse): we have only parallelized specific forms of `for` loops. This reflects OpenMP's scientific-computation heritage, where you have huge FORTRAN matrix calculations to parallelize. However, these days we also care about parallelism in more general settings, so OpenMP now provides more ways to parallelize.

Parallel Sections. The first mechanism, *parallel sections*, is a purely-static mechanism for specifying independent units of work which ought to be run in parallel. For instance, you can set up two (and exactly two, in this example) linked lists simultaneously as follows:

```
#include <stdlib.h>

typedef struct s { struct s* next; } S;

void setuplist (S* current) {
  for (int i = 0; i < 10000; i++) {
    current->next = (S*) malloc (sizeof(S));
    current = current->next;
  }
  current->next = NULL;
}

int main() {
  S var1, var2;
  #pragma omp parallel sections
  {
    #pragma omp section
    { setuplist (&var1); }
    #pragma omp section
    { setuplist (&var2); }
  }
```

```
    return 0;
  }
```

Note that the structure of the parallelism is explicitly visible in the structure of the code, and that you don't get to start an unbounded number of threads with the parallel sections mechanism.

What's another potential barrier to parallelism in the above code?

Nested Parallelism. Instead of collapsing loops, we can specify nested parallelism; we might have, for instance, two parallel sections, each of which contain parallel for loops. To enable such nested parallelism, you have to call `omp_set_nested` with a non-zero value. The runtime might refuse; call `omp_get_nested` to find out if the runtime complied or not. You can also set the `OMP_NESTED` environment variable to enable nesting.

Here's an example of nested parallelism.

```
#include <stdlib.h>
#include <omp.h>

int main() {
  double *array1, *array2;
  omp_set_nested(1);
  #pragma omp parallel sections shared(array1, array2)
  {
    #pragma omp section
    {
      array1 = (double*) malloc(sizeof (double)*1024*1024);
      #pragma omp parallel for shared(array1)
      for (int i = 0; i < 1024*1024; i++)
        array1[i] = i;
    }
    #pragma omp section
    {
      array2 = (double*) malloc(sizeof (double)*1024*512);
      #pragma omp parallel for shared(array2)
      for (int i = 0; i < 1024*512; i++)
        array2[i] = i;
    }
  }
}
```

Tasks: OpenMP's thread-like mechanism. The main new feature in OpenMP 3.0 is the notion of *tasks*. When the program executes a `#pragma omp task` statement, the code inside the task is split off as a task and scheduled to run sometime in the future. Tasks are more flexible than parallel sections, because parallel sections constrain exactly how many threads are supposed to run, and there is also always a join at the end of the parallel section. On the other hand, the OpenMP runtime can assign any task to any thread that's running. Tasks therefore have lower overhead.

Two examples which show off tasks, from [ACD+09], include a web server (with unstructured requests) and a user interface which allows users to start tasks that are to run in parallel.

Here's pseudocode for the Boa webserver main loop from [ACD+09].

```
#pragma omp parallel
  /* a single thread manages the connections */
  #pragma omp single nowait
  while (!end) {
    process any signals
    foreach request from the blocked queue {
      if (request dependencies are met) {
        extract from the blocked queue
        /* create a task for the request */
        #pragma omp task untied
          serve_request(request);
      }
    }
    if (new connection) {
      accept_connection();
      /* create a task for the request */
      #pragma omp task untied
        serve_request(new connection);
    }
    select();
  }
```

The `untied` qualifier lifts restrictions on the task-to-thread mapping; we won't talk about that. The `single` directive indicates that the runtime is only to use one thread to execute the next statement; otherwise, it could execute N copies of the statement, which does belong to a OpenMP `parallel` construct.

More OpenMP features. Random fact: you can use the `flush` directive to make sure that all values in registers or cache are written to memory. Usually, this isn't a problem for OpenMP programs, because of the way they're written. On a related note, the `barrier` directive explicitly instructs the runtime to wait for all threads to complete; OpenMP also has implicit barriers at the end of parallel sections.

OpenMP also supports critical sections (one thread at a time), atomic sections, and typical mutex locks (`omp_set_lock`, `omp_unset_lock`).

References

[ACD+09] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.