Lecture 17 — February 11, 2015

Patrick Lam *version 0*

There's still a dependence in the inner loop. This dependence is because all inner loop iterations write to the same location, `out[i]`. We'll discuss that problem below.

In any case, the outer loop is the one that can actually improve performance, since parallelizing it imposes much less barrier synchronization cost waiting for all threads to finish. So, even if we tell the compiler to ignore the reduction issue, it will generally refuse to parallelize inner loops:

```
$ cc -g -O3 -xloopinfo -xautopar -xreduction fploop2.c
"fploop2.c", line 5: PARALLELIZED, and serial version generated
"fploop2.c", line 8: not parallelized, not profitable
```

Summary of conditions for automatic parallelization. Here's what I can figure out; you may also refer to Chapter 3 of the Solaris Studio *C User's Guide*, but it doesn't spell out the exact conditions either. To parallelize a loop, it must:

- have a recognized loop style, e.g. `for` loops with bounds that don't vary per iteration;
- have no dependencies between data accessed in loop bodies for each iteration;
- not conditionally change scalar variables read after the loop terminates, or change any scalar variable across iterations;
- have enough work in the loop body to make parallelization profitable.

Reductions. The concept behind a reduction (as made "famous" in MapReduce, which we'll talk about later) is reducing a set of data to a smaller set which somehow summarizes the data. For us, reductions are going to reduce arrays to a single value. Consider, for instance, this function, which calculates the sum of an array of numbers:

```
1  double sum (double *array, int length)
2  {
3    double total = 0;
4
5    for (int i = 0; i < length; i++)
6      total += array[i];
7    return total;
8  }
```

There are two barriers: 1) the value of `total` depends on what gets computed in previous iterations; and 2) addition is actually non-associative for floating-point values. (Why? When is it appropriate to parallelize non-associative operations?)

Nevertheless, the Solaris C compiler will explicitly recognize some reductions and can parallelize them for you:

```
$ cc -O3 -xautopar -xreduction -xloopinfo sum.c
"sum.c", line 5: PARALLELIZED, reduction, and serial version generated
```

Note: If we try to do the reduction on `fploop.c` with `restrict`s added, we'll get the following:

```
$ cc -O3 -xautopar -xloopinfo  -xreduction -c fploop.c
"fploop.c", line 5: PARALLELIZED, and serial version generated
"fploop.c", line 8: not parallelized, not profitable
```

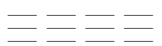
Dealing with function calls. Generally, function calls can have arbitrary side effects. Production compilers will usually avoid parallelizing loops with function calls; research compilers try to ensure that functions are pure and then parallelize them. (This is why functional languages are nice for parallel programming: impurity is visible in type signatures.)

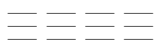
For builtin functions, like `sin()`, you can promise to the compiler that you didn't replace them with your own implementations (`-xbuiltin`), and then the compiler will parallelize the loop.

Another option is to crank up the optimization level (`-xO4`), or to explicitly tell the compiler to inline certain functions (`-xinline=`), thereby enabling parallelization. This doesn't work as well as one might hope; using macros will always work, but is less maintainable.

Helping the compiler parallelize. Let's summarize what we've seen. To help the compiler, we can use the `restrict` qualifier on pointers (possibly copying a pointer to a `restrict`-qualified pointer: `int * restrict p = s->p;`); and, we can make sure that loop bounds don't change in the loop (e.g. by using temporary variables). Some compilers can automatically create different versions for the alias-free case and the (parallelized) aliased case; at runtime, the program runs the aliased case if the inputs permit.

What happened last time? There was some confusion about manual parallelization. Recall that we manually parallelized three ways:


horizontal good:
create 4 threads to do 1000 iterations on sub-arrays.


horizontal bad:
1000 times, create 4 threads to iterate on sub-array.


vertical:
create 4 threads, handle 1 element at a time.

Timings were inconclusive. I tried harder and got these timings (in seconds) with `perf -r 5`:

	H good	H bad	V	auto
gcc, no opt	2.794	2.953	2.799	
gcc, -O3	0.588	1.490	0.980	
solaris, no opt	3.175	3.291	2.966	
solaris, -xO4	0.494	1.453	2.739	0.688

`perf` also told me other fun facts about the executions:

- fast executions had 3 to 7 cpu-migrations, slow ones had 4000 cpu-migrations.

- branch misses varied from 8k (gcc -O3, H good) to 208k (gcc -O3, bad).
- # cycles varied from 2B (gcc -O3, H good) to 9.7B (unopt).

Turns out that these stats don't perfectly predict the runtime. Frontend cycles was really high for solaris autoparallelization (which was quite fast).

Moving on. Now that we've seen automatic parallelization (and how that works in Solaris Studio and gcc), let's talk about manual—but compiler-aided—parallelization using OpenMP.

About OpenMP. OpenMP (Open Multiprocessing) is an API specification which allows you to tell the compiler how you'd like your program to be parallelized. Implementations of OpenMP include compiler support (present in Intel's compiler, Solaris's compiler, `gcc` as of 4.2, and Microsoft Visual C++) as well as a runtime library.

You use OpenMP[1] by specifying directives in the source code. In C and C++, these directives are pragmas of the form `#pragma omp ...`. There is also OpenMP syntax for Fortran.

Here are some benefits of the OpenMP approach:

- Because OpenMP uses compiler directives, you can easily tell the compiler to build a parallel version or a serial version (which it can do by ignoring the directives). This can simplify debugging—you have some chance of observing differences in behaviour between versions.
- OpenMP's approach also separates the parallelization implementation (inserted by the compiler) from the algorithm implementation (which you provide), making the algorithm easier to read. Plus, you're not responsible for dealing with thread libraries.
- The directives apply to limited parts of the code, thus supporting incremental parallelization of the program, starting with the hotspots.

Let's look at a simple example:

```
void calc (double *array1, double *array2, int length) {
  #pragma omp parallel for
  for (int i = 0; i < length; i++) {
    array1[i] += array2[i];
  }
}
```

This `#pragma` instructs the C compiler to parallelize the loop. It is the responsibility of the developer to make sure that the parallelization is safe; for instance, `array1` and `array2` had better not overlap. You no longer need to supply `restrict` qualifiers, although it's still not a bad idea. (If you wanted this to be autoparallelized without OpenMP, you would need to provide `restrict`.)

OpenMP will always start parallel threads if you tell it to, dividing the iterations contiguously among the threads.

Let's look at the parts of this `#pragma`.

[1] More information: `https://computing.llnl.gov/tutorials/openMP/`

- `#pragma omp` indicates an OpenMP directive;
- `parallel` indicates the start of a parallel region; and
- `for` tells OpenMP to run the following `for` loop in parallel.

When you run the parallelized program, the runtime library starts up a number of threads and assigns a subrange of the loop range to each of the threads.

Restrictions. OpenMP places some restrictions on loops that it's going to parallelize:

- the loop must be of the form

```
for (init expression; test expression; increment expression);
```

- the loop variable must be integer (signed or unsigned), pointer, or a C++ random access iterator;
- the loop variable must be initialized to one end of the range;
- the loop increment amount must be loop-invariant (constant with respect to the loop body);
- the test expression must be one of `>`, `>=`, `<`, or `<=`, and the comparison value (bound) must be loop-invariant.

(These restrictions therefore also apply to automatically parallelized loops.) If you want to parallelize a loop that doesn't meet the restriction, restructure it so that it does, as we saw last time.

Runtime effect. When you compile a program with OpenMP directives, the compiler generates code to spawn a *team* of threads and automatically splits off the worker-thread code into a separate procedure. The code uses fork-join parallelism, so when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back. Then the master thread continues running, while the worker threads wait for more work.

You can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable (adjustable by calling `omp_set_num_threads()`), and you can get the Solaris compiler to tell you what it did by giving it the options `-xopenmp -xloopinfo`.

Variable scoping

When using multiple threads, some variables, like loop counters, should be thread-local, or *private*, while other variables should be *shared* between threads. Changes to shared variables are visible to all threads, while changes to private variables are visible only to the changing thread. Let's look at the defaults that OpenMP uses to parallelize the above code.

```
$ er_src parallel-for.o
     1.   void calc (double *array1, double *array2, int length) {
        <Function: calc>

    Source OpenMP region below has tag R1
    Private variables in R1: i
    Shared variables in R1: array2, length, array1
     2.     #pragma omp parallel for

    Source loop below has tag L1
    L1 autoparallelized
    L1 parallelized by explicit user directive
    L1 parallel loop-body code placed in function _$d1A2.calc along with 0 inner loops
    L1 multi-versioned for loop-improvement:dynamic-alias-disambiguation.
        Specialized version is L2
     3.     for (int i = 0; i < length; i++) {
     4.       array1[i] += array2[i];
     5.     }
     6.   }
```

We can see that the loop variable `i` is private, while the `array1`, `array2` and `length` variables are shared. Actually, it would be fine for the `length` variable to be either shared or private, but if it was private, then you would have to copy in the appropriate initial value. The `array` variables, though, need to be shared.

Summary of default rules. Loop variables are private; variables defined in parallel code are private; and variables defined outside the parallel region are shared.

You can disable the default rules by specifying `default(none)` on the `parallel` pragma, or you can give explicit scoping:

```
#pragma omp parallel for private(i) shared(length, array1, array2)
```

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.