ECE459: Programming for Performance Winter 2016

Lecture 12 — Autoparallelization and OpenMP

Patrick Lam

Automatic Parallelization and OpenMP

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:

```
double sum (double *array, int length)
{
  double total = 0;

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

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
```

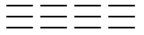
Dealing with function calls. Generally, function calls can have arbitrary side effects. Production compilers will usually avoid parallelizing loops with function calls; research compilers try to ensure that functions are pure and then parallelize them. (This is why functional languages are nice for parallel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

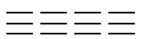
For builtin functions, like `sin()`, you can promise to the compiler that you didn't replace them with your own implementations (`-xbuiltin`), and then the compiler will parallelize the loop.

Another option is to crank up the optimization level (`-x04`), 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`.

- `#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;

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

- 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)
```