Lecture 17—Automatic Parallelization, OpenMP

ECE 459: Programming for Performance

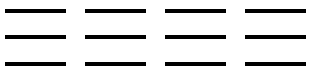
February 11, 2015

Road Map

- Now: compilers & automatic parallelization (when)
- Later today: under the hood
  (OpenMP: how compilers parallelize)

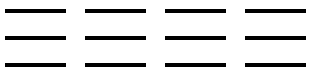
Lingering Questions about Runtimes

What happened here?



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.

Last year, `perf -r 5` gave following task-clocks (in seconds):

	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

Runtimes—Why?

Observations:

- Good runs had 5 to 7 cpu-migrations; bad had 4000.
- # cycles varied from 2B to 9.7B (no opt).
- Branch misses varied from 8k to 208k.

Reductions

- Reductions combine input data into a smaller (summary) set.
- We'll see a more complete definition when we touch on functional programming.
- Simplest instance: computing the sum of an array.

Consider the following code:

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

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

Can we parallelize this?

Reduction Problems

Barriers to parallelization:

1. value of `total` depends on previous iterations;
2. addition is actually non-associative for floating-point values (is this a problem?)

Recall that "associative" means:

$$a + (b + c) = (a + b) + c.$$

n this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.

Reduction Problems

Barriers to parallelization:

1. value of `total` depends on previous iterations;
2. addition is actually non-associative for floating-point values (is this a problem?)

Recall that "associative" means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.

Automatic Parallelization via Reduction

If we compile the program with `solarisstudio` and add the flag `-xreduction`, it will parallelize the code:

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

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

```
% solarisstudio -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

- A general function could have arbitary side effects.
- Production compilers tend to avoid parallelizing any loops with function calls.

Some built-in functions, like `sin()`, are "pure", have no side effects, and are safe to parallelize.

Note: this is why functional languages are nice for parallel programming: impurity is visible in type signatures.

Dealing with Function Calls in `solarisstudio`

- For `solarisstudio` you can use the `-xbuiltin` flag to make the compiler use its whitelist of "pure" functions.
- The compiler can then parallelize a loop which uses `sin()` (you shouldn't replace built-in functions with your own if you use this option).

Other options which may work:

1. Crank up the optimization level (`-xO4`).
2. Explicitly tell the compiler to inline certain functions (`-xinline=`, or use the `inline` keyword).

Summary of Automatic Parallelization

To help the compiler, we can:

- use `restrict` (make a `restricted` copy); and,
- make sure that loop bounds are constant (temporary variables).

Some compilers automatically create different versions for the alias-free case and the (parallelized) aliased case.

At runtime, the program runs the aliased case if correct.

Part I

OpenMP

Context for OpenMP

So far: Pthreads and automatic parallelization.

Next: “Manual” parallelization using OpenMP.

What is OpenMP?

OpenMP = Open Multiprocessing.

You specify parallelization; compiler implements.

All major compilers have OpenMP
(GNU, Solaris, Intel, Microsoft).

Use OpenMP[1] by specifying directives in the source.

C/C++: pragmas of the form `#pragma omp ...`

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

Benefits of OpenMP

- uses compiler directives—
  - easily compile same codebase for serial or parallel.
- separates the parallelization implementation from the algorithm implementation.

Directives apply to limited parts of the code, enabling incremental parallelization of the program. (Start with the hotspots.)

Simple OpenMP Example

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

Without OpenMP:

Could compiler parallelize this automatically?

How The Example Works

`#pragma` will make the compiler parallelize the loop.

- It does not look at loop contents, only loop bounds.
- It is your responsibility to make sure the code is safe.

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

You don't need to declare `restrict`, but it's a good idea. Need `restrict` for auto-parallelization (non-OpenMP).

Basic `pragma` syntax

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

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

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

What OpenMP can Parallelize

```
for (int i = 0; i < length; i++) { ... }
```

Can only parallelize loops which satisfy these conditions:

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

Note: these restrictions therefore also apply to automatically parallelized loops.

What OpenMP Does

- Compiler generates code to spawn a team of threads; automatically splits off worker-thread code into a separate procedure.
- Generated code uses `fork-join` parallelism; when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back.
- Afterwards, the master thread continues running, while the worker threads wait for more work .

As we saw, you can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable. (You can also adjust by calling `omp_set_num_threads()`).

- Solaris compiler tells you what it did if you use the flags `-xopenmp -xloopinfo`, or `er_src`.

Variable Scoping

Concept: thread-local variables (private) vs shared variables.

- Writes to private variables:
  visible only to writing thread.
- Writes to shared variables:
  visible to all threads.

Variable Scoping for Example

```
for (int i = 0; i < length; i++) { ... }
```

- `length` could be either shared or private.
  - if it was private, then you would have to copy in the appropriate initial value.
- `array` variables must be shared.

Default Variable Scoping

Let's look at the defaults that OpenMP uses to parallelize the `parallel-for` code:

```
% er_src parallel-for.o
      1.   <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
```

Parallelization information (via OpenMP)

```
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.    }
```

Default Variable Scoping Rules: A Summary

- Loop variables are private.
- Variables defined in parallel code are private.
- 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)
```