Lecture 16—Automatic Parallelization

ECE 459: Programming for Performance

February 9, 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)
```

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`?

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`?
Well, it should be shared.

- We want each thread to be able to write to it.

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`?
Well, it should be shared.

- We want each thread to be able to write to it.
- But, is there a race condition? (of course)

Aha! 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 the + operator.

Accessing Private Data outside a Parallel Region

Sometimes you want private variables, but want them initialized before the loop.

Consider this (silly) code:

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

- `data` is private, so OpenMP will not copy in initial 1.
- To make OpenMP copy the data before the threads start, use `firstprivate(data)`.
- To publish a variable after the (sequentially) last iteration of the loop, use `lastprivate(data)`.

Thread-Private Data

You might have a global variable, for which each thread should have a persistent local copy—lives across parallel regions.

- Use the `threadprivate` directive.
- Add `copyin` if you want something like `firstprivate`.
- There is no `lastprivate` since the data is accessible after the loop.

Thread-Private Data Example (1)

```
#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");
```

Thread-Private Data Example (2)

```
    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;
}
```

```
% ./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

- Normally, it's best to parallelize the outermost loop.

Consider this code:

```
#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;
}
```

- Would parallelizing this outer loop benefit us?
  What about the inner loop?

OpenMP supports *collapsing* loops:

- Creates a single loop for all the iterations of the two loops.
- Outer loop only enables the use of 2 threads.
- Collapsed loop lets us use up to 20,000 threads.

Better Performance Through Scheduling: An Example

Default mode: *Static scheduling*.

- Assumes each iteration takes the same running time.

Does that assumption hold for this 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;
}
```

Better Performance Through Scheduling

- In example, earlier iterations are faster than later iterations. Result: sublinear scaling—wait for all iterations to finish.

- Turn on *dynamic schedule* mode by adding `schedule(dynamic)` to the pragma:
    - Breaks the work into chunks;
    - Distributes the work to each thread in chunks;
    - Higher overhead;
    - Default chunk size of 1 (can modify, e.g. `schedule(dynamic, n/50)`).

More Scheduling

Other schedule modes exist, e.g. `guided`, `auto` and `runtime`.

- `guided` changes the chunk size based on work remaining.
    - Default minimum chunk size = 1 (can modify)
- `auto` lets OpenMP decide what's best.
- `runtime` doesn't pick a mode until runtime.
    - Tune with `OMP_SCHEDULE` environment variable

Part II

Beyond `for` loops: OpenMP Parallel Sections and Tasks

Why more than for?

So far, we can parallelize (some) `for` loops with OpenMP.

Less powerful than Pthreads. (Also harder to get wrong.)

Reflects OpenMP's scientific-computation heritage.

Today, we need more general parallelism, not just matrices.

Parallel Sections Example: Linked Lists (1)

Purely-static mechanism for specifying independent work units which should run in parallel.

Linked list example:

```
#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;
}
```

Parallel Sections Example: Linked Lists (2)

(Exactly) 2 linked lists:

```
int main() {
  S var1, var2;
  #pragma omp parallel sections
  {
    #pragma omp section
    { setuplist (&var1); }
    #pragma omp section
    { setuplist (&var2); }
  }
  return 0;
}
```

Parallelism structure explicitly visible.

Finite number of threads.

(What’s another barrier to parallelism here?)

Nested Parallelism

Sometimes you don't want to collapse loops.

Example: (better example in PDF notes!)

```
    #pragma omp parallel sections
    {
      #pragma omp section
      {
          #pragma omp parallel for
          for (int i = 0; i < 1000; i++) { ... }
      }
      #pragma omp section
      {
          #pragma omp parallel for
          for (int i = 0; i < 1000; i++) { ... }
      }
    }
```

To enable nested parallelism, call `omp_set_nested(1)` or set `OMP_NESTED`. (Runtime might refuse.)

OpenMP Tasks

Main new feature in OpenMP 3.0.

`#pragma omp task`:

code splits off and scheduled to run later.

More flexible than parallel sections:

- can run as many threads as needed;
- tasks do not need to join (like detached threads).

OpenMP does the task-to-thread mapping—lower overhead.

Examples of tasks

Two examples:

- web server

  unstructured requests
- user interface

  allows users to start concurrent tasks

Boa webserver main loop example

```
#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();
  }
```

Other OpenMP qualifiers

`untied`: lifts restrictions on task-to-thread mapping.

`single`: only one thread runs the next statement (not N copies).

`flush` directive: write all values in registers or cache to memory.

`barrier`: wait for all threads to complete. (OpenMP also has implicit barriers at ends 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`).