Lecture 10 - OpenMP Basics

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 25, 2012

Course Comments

- Overwhelmingly you think the mic should be fixed
- Pace seems fine
- Some interest in good practices / other techniques
- Assignment 1 is probably too easy
- Probably keep things more or less the same

Assignment 1

- Don't worry about the quality of the result
- Worry about parallelizing the code
- Mainly for pthread experience and benchmarking results
- Don't be afraid to remove stuff from the `tex` file if it's not needed

Assignment 1 Improvement

```
@@ -8,11 +8,10 @@
 static unsigned long int iterations = 0;
 static unsigned long int count = 0;

-unsigned long int montecarlo(unsigned long int iterations)
+unsigned long int montecarlo(unsigned long int iterations,
                              unsigned int seed)
 {
     unsigned long int i, c = 0;
     double x, y, z;
-    unsigned int seed = 1;

     for (i = 0; i < iterations; ++i) {
         x = (double)rand_r(&seed)/RAND_MAX;
@@ -69,7 +68,7 @@ int main(int argc, char *argv[])
     // Insert your code below
     #else
     // Serial calculation of pi
-    count += montecarlo(iterations);
+    count += montecarlo(iterations, 1);
     #endif
```

Introduction

Now that we've seen automatic parallelization, let's talk about manual parallelization using OpenMP.

- OpenMP (Open Multi-Processing) is an API specification which allows you to tell the compiler how you'd like your program to be parallelized
- Implementations are present in all major compilers (GNU, Solaris, Intel, Microsoft)

You use OpenMP[1] by specifying directives in the source code. In C/C++, these directives are pragmas of the form
`#pragma omp ...`

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

Benefits

Here are some benefits of the OpenMP approach:

- OpenMP uses compiler directives
    - Easily compile a serial or parallel version
- OpenMP's approach also separates the parallelization implementation from the algorithm implementation
- The directives apply to limited parts of the code, thus supporting incremental parallelization of the program

Example

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];
    }
}
```

Could we parallelize this automatically?

Operation

- `#pragma` will make the compiler parallelize the loop
- It does not look at anything inside the loop, only the loop bounds
- It is your responsibility to make sure the code is safe
- The pointers should be declared as `restrict` in the example for automatic parallelization

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

Basic pragma

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 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 expr; test expr; increment expr);`
- 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

Note: these restrictions therefore also apply to automatically parallelized loops

Runtime Effect

- the compiler generates code to spawn a team of threads and automatically splits off the worker-thread code into a separate procedure
- 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
- After 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 can tell you what it did by using the flags `-xopenmp -xloopinfo`

Default Variable Scoping

- We are familiar with the concept of thread-local variables (private) and shared variables
- Changes to private variables are visible only to the changing thread
- Changes to shared variables are visible to all threads

Let's look at the defaults that OpenMP uses to parallelize the `calc` 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
     3.      for (int i = 0; i < length; i++) {
     4.        array1[i] += array2[i];
     5.      }
     6.    }
```

Variable Scoping

- 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
- `array` variables have to be shared

Summary of default rules:

- 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`? Well, it should be shared

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

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 +

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 it
- To make OpenMP copy the data before the threads start use `firstprivate(data)`
- To set a variable equal to the last iteration of the loop, use `lastprivate(data)`

Thread-Private Data

- You could have a global variable which you want to make local to each thread
- You can do this with the `threadprivate` directive
- Use `copyin` directive 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? 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 look lets us use up to 20,000 threads

Better Performance Through Scheduling Example

Default mode: *Static scheduling*

- Assumes each iteration takes the same amount of time to run

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

- Earlier iterations are faster than later iterations
- You can use the *dynamic schedule* mode by adding `schedule(dynamic)` to the pragma
    - Breaks the work into chunks
    - Distributes the work to each thread in chunks
    - Requires more overhead
    - Default chunk size of 1, can modify

More Scheduling

Other modes, such as `guided`, `auto` and `runtime`

- `guided` changes the chunk size based on the amount of work remaining
    - Minimum chunk size defaults to 1, can modify
- `auto` lets OpenMP decide what's best
- `runtime` doesn't pick a mode until the program actually runs
    - Changed with the `OMP_SCHEDULE` environment variable