ECE459: Programming for Performance Winter 2014

Lecture 10 — February 6, 2014

Patrick Lam *version 2*

Following Gove, we'll parallelize the following code:

```
1  #include <stdlib.h>
2
3  void setup(double *vector, int length) {
4      int i;
5      for (i = 0; i < length; i++)
6      {
7          vector[i] += 1.0;
8      }
9  }
10
11 int main()
12 {
13     double *vector;
14     vector = (double*) malloc (sizeof (double) * 1024 * 1024);
15     for (int i = 0; i < 1000; i++)
16     {
17         setup (vector, 1024*1024);
18     }
19 }
```

Automatic Parallelization. Let's first see what compilers can do automatically. The Solaris Studio compiler yields the following output:

```
$ cc -O3 -xloopinfo -xautopar omp_vector.c
"omp_vector.c", line 5: PARALLELIZED, and serial version generated
"omp_vector.c", line 15: not parallelized, call may be unsafe
```

Note: The Solaris compiler generates two versions of the code, and decides, at runtime, if the parallel code would be faster, depending on whether the loop bounds, at runtime, are large enough to justify spawning threads.

Under the hood, most parallelization frameworks use `OpenMP`, which we'll see next time. For now, you can control the number of threads with the `OMP_NUM_THREADS` environment variable.

Autoparallelization in `gcc`. `gcc` 4.3+ can also parallelize loops, but there are a couple of problems: 1) the loop parallelization doesn't seem very stable yet; 2) I can't figure out how to make `gcc` tell you what it did; and, perhaps most importantly for performance, 3) `gcc` doesn't have any heuristics yet for guessing which loops are profitable.

One way to inspect the resulting code is by giving `gcc` the `-S` option and looking at the resulting assembly code yourself. This is obviously not practical for production software.

```
$ gcc -std=c99 omp_vector.c -O2 -ftree-parallelize-loops=2 -S
```

The resulting `.s` file contains the following code:

```
call    GOMP_parallel_start
movl    %edi, (%esp)
call    setup._loopfn.0
call    GOMP_parallel_end
```

`gcc` code appears to ignore `OMP_NUM_THREADS`. Here's some potential output from a parallelized program:

```
$ export OMP_NUM_THREADS=2
$ time ./a.out
real 0m5.167s
user 0m7.872s
sys 0m0.016s
```

(When you use multiple (virtual) CPUs, CPU usage can increase beyond 100% in `top`, and real time can be less than user time in the `time` output, since user time counts the time used by all CPUs.)

Let's look at some gcc examples from: `http://gcc.gnu.org/wiki/AutoparRelated`.

Loops That gcc's Automatic Parallelization Can Handle.

Single loop:

```
for (i = 0; i < 1000; i++)
    x[i] = i + 3;
```

Nested loops with simple dependency:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        X[i][j] = X[i][j] + Y[i-1][j];
```

Single loop with not-very-simple dependency:

```
for (i = 0; i < 10; i++)
    X[2*i+1] = X[2*i];
```

Loops That gcc's Automatic Parallelization Can't Handle.

Single loop with if statement:

```
for (j = 0; j <= 10; j++)
    if (j > 5) X[i] = i + 3;
```

Triangle loop:

```
for (i = 0; i < 100; i++)
    for (j = i; j < 100; j++)
        X[i][j] = 5;
```

Manual Parallelization. Let's first think about how we could manually parallelize this code.

- **Option 1:** horizontal, ☰☰☰☰
  Create 4 threads; each thread does 1000 iterations on its own sub-array.
- **Option 2:** bad horizontal, ☰☰☰☰
  1000 times, create 4 threads which each operate once on the sub-array.
- **Option 3:** vertical ||||
  Create 4 threads; for each element, the owning thread does 1000 iterations on that element.

We can try these and empirically see which works better. As you might expect, bad horizontal does the worst. Horizontal does best. See L11 notes for benchmark results and further discussion.

Case study: Multiplying a Matrix by a Vector.

Next, we'll see how automatic parallelization does on a more complicated program. We will progressively remove barriers to parallelization for this program:

```
1  void matVec (double **mat, double *vec, double *out,
2               int *row, int *col)
3  {
4    int i, j;
5    for (i = 0; i < *row; i++)
6    {
7      out[i] = 0;
8      for (j = 0; j < *col; j++)
9      {
10       out[i] += mat[i][j] * vec[j];
11     }
12   }
13 }
```

The Solaris C compiler refuses to parallelize this code:

```
$ cc -O3 -xloopinfo -xautopar fploop.c
"fploop.c", line 5: not parallelized, not a recognized for loop
"fploop.c", line 8: not parallelized, not a recognized for loop
```

For definitive documentation about Sun's automatic parallelization, see Chapter 10 of their *Fortran Programming Guide* and do the analogy to C:

`http://download.oracle.com/docs/cd/E19205-01/819-5262/index.html`

In this case, the loop bounds are not constant, and the write to `out` might overwrite either `row` or `col`. So, let's modify the code and make the loop bounds `int`s rather than `int *`s.

```
1  void matVec (double **mat, double *vec, double *out,
2               int row, int col)
3  {
4    int i, j;
5    for (i = 0; i < row; i++)
6    {
7      out[i] = 0;
8      for (j = 0; j < col; j++)
9      {
10       out[i] += mat[i][j] * vec[j];
11     }
12   }
13 }
```

This changes the error message:

```
$ cc -O3 -xloopinfo -xautopar fploop1.c
"fploop1.c", line 5: not parallelized, unsafe dependence
"fploop1.c", line 8: not parallelized, unsafe dependence
```

Now the problem is that `out` might alias `mat` or `vec`; as I've mentioned previously, parallelizing in the presence of aliases could change the run-time behaviour.

`restrict` qualifier. Recall that the `restrict` qualifier on pointer `p` tells the compiler[1] that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p`.

```
1  void matVec (double **mat, double *vec, double * restrict out,
2               int row, int col)
3  {
4    int i, j;
5    for (i = 0; i < row; i++)
6    {
7      out[i] = 0;
8      for (j = 0; j < col; j++)
9      {
10       out[i] += mat[i][j] * vec[j];
11     }
12   }
13 }
```

Now Solaris `cc` is happy to parallelize the outer loop:

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

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.

[1] `http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html`

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 the `restrict`ed version of the case study, 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.

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.