**ECE459: Programming for Performance** Winter 2013

# Lecture 8 — January 31, 2013

*Patrick Lam* *version 1*

### How to Parallelize Code

Here's a four-step outline of what you need to do.

1. Profile the code.
2. Find dependencies in hotspots. For each dependency chain in a hotspot, figure out if you can execute the chain as multiple parallel tasks or a loop over multiple parallel iterations. Think about changing the algorithm, if that would help.
3. Estimate benefits.
4. If they're not good enough (e.g. far from linear speedup), step back and see if you can parallelize something else up the call chain, or at a higher level of absraction. (Think Mandelbrot sets and computing different points in parallel).

Try to reduce the amuont of synchronization that you have to do (waiting for parallel tasks to finish), because that always slows you down.

## Thread Pools

We talked about "single task, multiple threads" last week. The idea behind a *thread pool* is that it's relatively expensive to start a thread; it costs resources to keep the threads running at the operating system level; and the threads won't run optimally anyway, because they'll spend too much time swapping state in and out of the cache. Instead, you start an appropriate number of threads, which each grab work from a work queue, do the work, and report the results back. Web servers are a good application of thread pools[1].

A key question is: how many threads should you create? This depends on which resources your threads use; if you are writing computationally-intensive threads, then you probably want to have fewer threads than the number of virtual CPUs. You can also use Amdahl's Law to estimate the maximum useful number of threads, as discussed previously.

Here's a longer discussion of thread pools:

`http://www.ibm.com/developerworks/library/j-jtp0730.html`

[1]Apache does this: `http://httpd.apache.org/docs/2.0/mod/worker.html`. Also see an assignment where the students write thread-pooled web servers: `http://www.cse.nd.edu/~dthain/courses/cse30341/spring2009/project4/project4.html`.

Modern languages provide thread pools; Java's `java.util.concurrent.ThreadPoolExecutor`[2], C#'s `System.Threading.ThreadPool`[3], and GLib's `GThreadPool`[4] all implement thread pools.

**GLib.** GLib is a C library developed by the GTK team. It provides many useful features that you might otherwise have to implement yourself in C. Consider using GLib's thread pool in your assignment 2, unless you want to implement the work queue yourself. (I see no reason to do that!)

## Automatic Parallelization

We'll now talk about automatic parallelization. The vision is to take your standard sequential C program and convert it into a parallel C program which leverages multiple cores, CPUs, machines, etc. This was an active area of research in the 1990s, then tapered off in the 2000s (because it's a hard problem!); it is enjoying renewed interest now (but it's still hard!)

**What can we parallelize?** The easiest kind of program to parallelize is the classic Fortran program which performs a computation over a huge array. C code—if it's the right kind—is a bit worse, but still tractable, given enough hints to the compiler. For us, the right kind of code is going to be array codes. Some production compilers, like the non-free Intel C compiler `icc`, the free-as-in-beer Solaris Studio compiler[5], and the free GNU C compiler `gcc`, include support for parallelization, with different maturity levels.

Following Gove, we'll parallelize the following code:

```c
#include <stdlib.h>

void setup(double *vector, int length) {
    int i;
    for (i = 0; i < length; i++)
    {
        vector[i] += 1.0;
    }
}

int main()
{
    double *vector;
    vector = (double*) malloc (sizeof (double) * 1024 * 1024);
    for (int i = 0; i < 1000; i++)
    {
        setup (vector, 1024*1024);
    }
}
```

**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.

[2]`http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html`
[3]`http://msdn.microsoft.com/en-us/library/3dasc8as\%28v=vs.80\%29.aspx`
[4]`http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html#GThreadPool`
[5]`http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html`

- **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.

**Automatic Parallelization.** 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` 4.4 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;
```

## 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:

```
void matVec (double **mat, double *vec, double *out,
             int *row, int *col)
{
  int i, j;
  for (i = 0; i < *row; i++)
  {
    out[i] = 0;
    for (j = 0; j < *col; j++)
    {
      out[i] += mat[i][j] * vec[j];
    }
  }
}
```

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[6] 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
```

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

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.

**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.