

## Lecture 12 — Autoparallelization

Patrick Lam

### 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 [?] and the free GNU C compiler `gcc`, include support for parallelization, with different maturity levels.

Following Gove, we'll parallelize the following code:

```
#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);
    }
}
```

**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 in a comprehensible way (you can try `-fdump-tree-parloops-details`); and, perhaps most importantly for performance, 3) gcc doesn't have many heuristics yet for guessing which loops are profitable (since 4.8, it can use profiling data and tries to infer the number of loop iterations happen)<sup>1</sup>.

clang also has the polly parallelization framework, but I couldn't figure out how to try it.

One way to inspect gcc's output is by giving it 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 -floop-parallelize-all -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:

---

<sup>1</sup><https://gcc.gnu.org/wiki/AutoParInGCC>

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

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

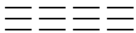
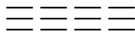

Single loop with if statement:

```
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:

```
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 ints rather than int \*s.

```
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];
        }
    }
}
```

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<sup>2</sup> 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.

```
void matVec (double **mat, double *vec, double * restrict 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];
        }
    }
}
```

---

<sup>2</sup><http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html>

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
```