Lecture 16—Automatic Parallelization

ECE 459: Programming for Performance

February 9, 2015

Road Map

- Past: manual parallelization with threads
- Now: compilers & automatic parallelization (when)
- Future: under the hood (how compilers parallelize)

Example Code to Parallelize

```
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 of Example Code

Let's try automatic parallelization.

Compiling with solarisstudio and automatic parallelization yields the following:

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

How will this code compare to our manual efforts?
(If you weren't in class, you'll have to try it yourself.)

Note: solarisstudio generates two versions of the code, and decides, at runtime, if the parallel code would be faster.

Autoparallelization implementation: OpenMP

Under the hood, most parallelization frameworks use `OpenMP`, which we'll see next lecture.

For now: you can control the number of threads with the `OMP_NUM_THREADS` environment variable.

Automatic Parallelization in gcc

gcc (since 4.3) can also auto-parallelize loops.
However, there are a few problems:

1. It will not tell you which loops it parallelizes (nicely).
2. It only operates with a fixed number of threads.
3. The profitability metrics are quite simple.
4. Only operates in simple cases.

Use the flags -floop-parallelize-all
-ftree-parallelize-loops=N where N is the # of threads.

Note: gcc also uses OpenMP but ignores OMP_NUM_THREADS.

Understanding Automatic Parallelization in gcc

Flag -fdump-tree-parloops-details shows what the automatic parallelizations were, but it's quite unreadable.

Instead, you can look at the assembly code to see the parallelizations (obviously, impractical for a large project).

```
% gcc −std=c99 −O3 −ftree−parallelize−loops=4
  omp_vector_gcc.c −S −o omp_vector_gcc_auto.s
```

The resulting .s file contains the following code:

```
call       GOMP_parallel_start
leaq       80(%rsp), %rdi
call       setup._loopfn.0
call       GOMP_parallel_end
```

Note: gcc also parallelizes main._loopfn.2 and main._loopfn.3, although it looks like it serves little purpose.

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;
```

Examples from:
http://gcc.gnu.org/wiki/AutoparRelated

Summary of Conditions for Automatic Parallelization

From Chapter 10 of Oracle's *Fortran Programming Guide* [1] translated to C, a loop 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; and
- have enough work in the loop body to make parallelization profitable.

---

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

Manually Parallelizing the Example Code

What can we do to parallelize this code?
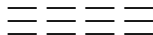
Option 1:

Option 2:

Option 3:

Manually Parallelizing the Example Code

What can we do to parallelize this code?

Option 1: horizontal   ☰ ☰ ☰ ☰

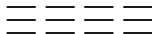- Create 4 threads; each thread does 1000 iterations on its own sub-array.

Option 2:
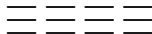

Option 3:

Manually Parallelizing the Example Code

What can we do to parallelize this code?

Option 1: horizontal $\equiv\equiv\equiv\equiv$

- Create 4 threads; each thread does 1000 iterations on its own sub-array.

Option 2: bad horizontal $\equiv\equiv\equiv\equiv$

- 1000 times, create 4 threads which each operate once on the sub-array.

Option 3:

Manually Parallelizing the Example Code

What can we do to 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.

Manual Parallelization Demo

I'll show a demo of three example PThread parallelizations.

Methodology: compiling with `solarisstudio`,
flags `-O3 -lpthread`.

Which manual option performs better?

Comparing Parallelization Results

How does autoparallelization compare to manual parallelization?

Case Study 2: Multiplying a Matrix by a Vector

Let's see how automatic parallelization does on a more complicated program (could we parallelize this?):

```
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  }
```

Reminder: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 32 \end{bmatrix}$

Case Study: Automatic Parallelization, Attempt 1

Well, based on our knowledge, we could parallelize the outer loop.

Let's see what `solarisstudio` will do for us...

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

...it refuses to do anything, guesses?

Case Study: Automatic Parallelization, Attempt 2

- The loop bounds are not constant, since one of the variables may alias `row` or `col`, even though `int` $\neq$ `double`.

So, let's add `restrict` to `row` and `col` and see what happens. . .

```
% solarisstudio −cc −O3 −xautopar −xloopinfo −c fploop.c
"fploop.c", line 5: not parallelized, unsafe dependence
"fploop.c", line 8: not parallelized, unsafe dependence
```

Now it recognizes the loop, but still won't parallelize it. Why?

Case Study: Automatic Parallelization, Attempt 3

- out might alias `mat` or `vec`, which would make this unsafe

Let's add another `restrict` to `out`:

```
% solarisstudio −cc −O3 −xautopar −xloopinfo −c fploop.c
"fploop.c", line 5: PARALLELIZED, and serial version
  generated
"fploop.c", line 8: not parallelized, unsafe dependence
```

Now, we can get the outer loop to parallelize.

- Parallelizing the outer loop is almost always better than inner loops, and usually it's a waste to do both, so we're done.

Note: We can parallelize the inner loop as well (it's similar to Assignment 1). We'll see that `solarisstudio` can do it automatically.