

# Lecture 10—Automatic Parallelization

ECE 459: Programming for Performance

February 6, 2014

# Last Time

Data vs Task Parallelism  
(including SIMD example)

7 Parallelization Patterns

Parallelization Strategy and Tactics (e.g. thread pools)

## Assignment 2 Preview

Three parts:

- Restructuring code for Automatic Parallelization.
- Manual parallelization of a loop with OpenMP.
- Using OpenMP tasks effectively.

Didn't take me very long to make solution.

# Part I

## Parallelizing Code

## Example Code from the Textbook

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 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:

- ❶ It will not tell you which loops it parallelizes (nicely).
- ❷ It only operates with a fixed number of threads.
- ❸ The profitability metrics are quite simple.
- ❹ Only operates in simple cases.

Use the flag `-ftree-parallelize-loops=N` where N is the number 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.

# Manually Parallelizing the Example Code

What can we do to parallelize this code?

**Option 1:**

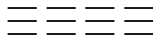
**Option 2:**

**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 

- 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:**

# 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 autparallelization 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 }
```

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



## 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.

# 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*<sup>1</sup> 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.

---

<sup>1</sup><http://download.oracle.com/docs/cd/E19205-01/819-5262/index.html>

# Reductions

- Reductions combine input data into a smaller (summary) set.
- We'll see a more complete definition when we touch on functional programming.
- Simplest instance: computing the sum of an array.

Consider the following code:

```
double sum (double *array, int length)
{
    double total = 0;

    for (int i = 0; i < length; i++)
        total += array[i];
    return total;
}
```

Can we parallelize this?

# Reduction Problems

Barriers to parallelization:

- ① value of `total` depends on previous iterations;
- ② addition is actually non-associative for floating-point values (is this a problem?)

Recall that “associative” means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.



# Reduction Problems

Barriers to parallelization:

- ❶ value of `total` depends on previous iterations;
- ❷ addition is actually non-associative for floating-point values (is this a problem?)

Recall that “associative” means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative.

# Automatic Parallelization via Reduction

If we compile the program with `solarisstudio` and add the flag `-xreduction`, it will parallelize the code:

```
% solarisstudio -cc -xautopar -xloopinfo -xreduction -O3 -c sum.c  
"sum.c", line 5: PARALLELIZED, reduction, and serial version  
generated
```

**Note:** If we try to do the reduction on the restricted version of the case study, we'll get the following:

```
% solarisstudio -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

- A general function could have arbitrary side effects.
- Production compilers tend to avoid parallelizing any loops with function calls.

Some built-in functions, like `sin()`, are “pure”, have no side effects, and are safe to parallelize.

**Note:** this is why functional languages are nice for parallel programming: impurity is visible in type signatures.

## Dealing with Function Calls in solarisstudio

- For solarisstudio you can use the `-xbuiltin` flag to make the compiler use its whitelist of “pure” functions.
- The compiler can then parallelize a loop which uses `sin()` (you shouldn't replace built-in functions with your own if you use this option).

Other options which may work:

- ① Crank up the optimization level (`-xO4`).
- ② Explicitly tell the compiler to inline certain functions (`-xinline=`, or use the `inline` keyword).

# Summary of Automatic Parallelization

To help the compiler, we can:

- use `restrict` (make a restricted copy); and,
- make sure that loop bounds are constant (temporary variables).

Some compilers automatically create different versions for the alias-free case and the (parallelized) aliased case.

At runtime, the program runs the aliased case if correct.