Lecture 08 - Automatic Parallelization

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

January 20, 2012

Introduction

- Today's topic is automatic parallelization
- Vision: take a sequential C program and convert it into a parallel version
- Lots of research in the early 1990s, then tapered off
- Renewed interest now since multicores are so common

Arrays vs Dynamically-allocated Data Structures

- Easiest to parallelize programs which performs a computation over a huge array
- Some languages are easier than others to reason about (and therefore automatically parallelize)
- C can be easy to parallelize, given the right code and compiler hints
- For this course, we'll just worry about automatic parallelization on arrays (as was the case last year)
- Some production compilers support parallelization: `icc` (Intel's non-free compiler), `solarisstudio` (Oracle's free-as-in-beer compiler [1]) and `gcc` (GNU's free-as-in-speech compiler)

[1] `http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html`

Example Code from the Textbook

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);
    }
}
```

Example Code Parallelization

What can we do to parallelize this code?

Option 1:

Option 2:

Option 3:

Example Code Parallelization

What can we do to parallelize this code?

Option 1:

- Divide up the array on **line 5** so each thread operates on a sub-array

Option 2:

Option 3:

Example Code Parallelization

What can we do to parallelize this code?

Option 1:

- Divide up the array on **line 5** so each thread operates on a sub-array

Option 2:

- Divide up the number of iterations on **line 15** so each thread has an even amount of calls to `setup`

Option 3:

Example Code Parallelization

What can we do to parallelize this code?

Option 1:

- Divide up the array on **line 5** so each thread operates on a sub-array

Option 2:

- Divide up the number of iterations on **line 15** so each thread has an even amount of calls to `setup` (unsafe)

Option 3:

- Divide up the array before the loop on **line 15** and each thread does it's iterations on a sub-array

Example Code with Manual Parallelization

I'll show a demo of two example parallelizations

Compiling with `solarisstudio`, flags `-O3 -lpthread`

Which manual option indeed performs better?

Example Code with Automatic Parallelization

Let's try with automatic parallelization

Compiling with `solarisstudio` and automatic parallelization flags yields the following:

```
% solarisstudio -cc -O3 -xautopar -xloopinfo omp_vector.c
  -o omp_vector_auto
"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?

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

Example Code Comparison Between Methods

- Under the hood, most parallelization frameworks utilize `OpenMP`, which we'll see next lecture
- For now, just know you can control the number of threads with the `OMP_NUM_THREADS` environment variable

How does it compare?

Example Code Comparison Between Methods

- Under the hood, most parallelization frameworks utilize `OpenMP`, which we'll see next lecture
- For now, just know you can control the number of threads with the `OMP_NUM_THREADS` environment variable

How does it compare?

- Relative ordering: **Option 3** > Automatic > **Option 1**
- Its automatic parallelization of **Option 1** was better than ours, why?

Example Code Comparison Between Methods

- Under the hood, most parallelization frameworks utilize `OpenMP`, which we'll see next lecture
- For now, just know you can control the number of threads with the `OMP_NUM_THREADS` environment variable

How does it compare?

- Relative ordering: **Option 3** > Automatic > **Option 1**
- Its automatic parallelization of **Option 1** was better than ours, why?
- Our **Option 3** performed better, even though both used the same number of threads, why?

Example Code with Automatic Parallelization in gcc

- gcc (since 4.3) can also 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 flag, `-ftree-parallelize-loops=N` where `N` is the number of threads

Note: `gcc` also uses `OpenMP` but just ignores the `OMP_NUM_THREADS` environment variable

Example Code Automatic Parallelization Inspection in gcc

There's a flag `-fdump-tree-parloops-details` to see 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

Case Study: Multiplying a Matrix by a Vector

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

```
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];
    }
  }
}
```

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 to `row` or `col`, despite being different types

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 its a waste to do both, so we're done

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

Examples of Loops Automatic Parallelization Can Handle

One nested and simple 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];
```

One nested loop with Not-very-simple dependency

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

Examples of Loops Automatic Parallelization Can't Handle

Simple 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

Here's some conditions for automatic parallelization from Chapter 10 of Oracle's *Fortran Programming Guide* [2] with analogies 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
- have enough work in the loop body to make parallelization profitable

---

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

Reductions

- Reductions combine the data to a smaller set
- We'll see a more complete definition when we touch on functional programming
- Simplest instance is 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? (it should look somewhat similar)

Reduction Problems

The problems:

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

Recall associate means: $a + (b + c) = (a + b) + c$

Reduction Problems

The problems:

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

Recall associate 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 Automatic Parallelization

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 arbitary side effects
- Production compilers tend to avoid parallelizing any loops with function calls

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

Note: this is why functional languages are nice for parallel programming, since you explicitly state pure and impure functions

Dealing with Function Calls in `solarisstudio`

- For `solarisstudio` you can use the `-xbuiltin` flag to make the compiler use its whitelist of "pure" functions
- This means the compiler can 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)