Lecture 13—Parallelization: Patterns and Strategy

ECE 459: Programming for Performance

January 30, 2015

Assignment 2 Preview

Three parts:

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

Part I

More Parallelization Patterns

Pattern 5: Pipeline of Tasks

Seen briefly in computer architecture.

- Use multiple stages; each thread handles a stage.

  Example: a program that handles network packets: (1) accepts packets, (2) processes them, and (3) re-transmits them. Could set up the threads such that each packet goes through the threads.

- Improves throughput; may increase latency as there's communication between threads.

- In the best case, you'll have a linear speedup.

  Rare, since the runtime of the stages will vary, and the slow one will be the bottleneck (but you could have 2 instances of the slow stage).

Pattern 6: Client-Server

To execute a large computation, the server supplies work to many clients—as many as request it.

Client computes results and returns them to the server.

Examples: botnets, `SETI@Home`, GUI application (backend acts as the server).

Server can arbitrate access to shared resources (such as network access) by storing the requests and sending them out.

- Parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks

Pattern 7: Producer-Consumer

Variant on the pipeline and client-server models.
Producer generates work, and consumer performs work.

Example: producer which generates rendered frames; consumer which orders these frames and writes them to disk.

Any number of producers and consumers.

- This approach can improve throughput and also reduces design complexity

Combining Strategies

Most problems don't fit into one category, so it's often best to combine strategies.

For instance, you might often start with a pipeline, and then use multiple threads in a particular pipeline stage to handle one piece of data.

Tip: estimate to see what divisions of strategies would work best (might have to do more iterations of Amdahl's law depending on the amount of strategies you can use).

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate**.

- build system, e.g. parallel make
- optical character recognition system

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate**.

- build system, e.g. parallel make
  - Multiple independent tasks, at a per-file granularity
- optical character recognition system
  - Pipeline of tasks
  - 2 tasks - finding characters and analyzing them

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:

- producer-consumer (no rendering frames, please):

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
  - Computation of a mathematical function with independent sub-formulas.
- producer-consumer (no rendering frames, please):
  - Processing of stock-market data: a server might generate raw financial data (quotes) for a particular security. The server would be the producer. Several clients (or consumers) may take the raw data and use them in different ways, e.g. by computing means, generating charts, etc.

Part II

Parallelizing Code

How To Parallelize Code: Strategy

Four-step outline:

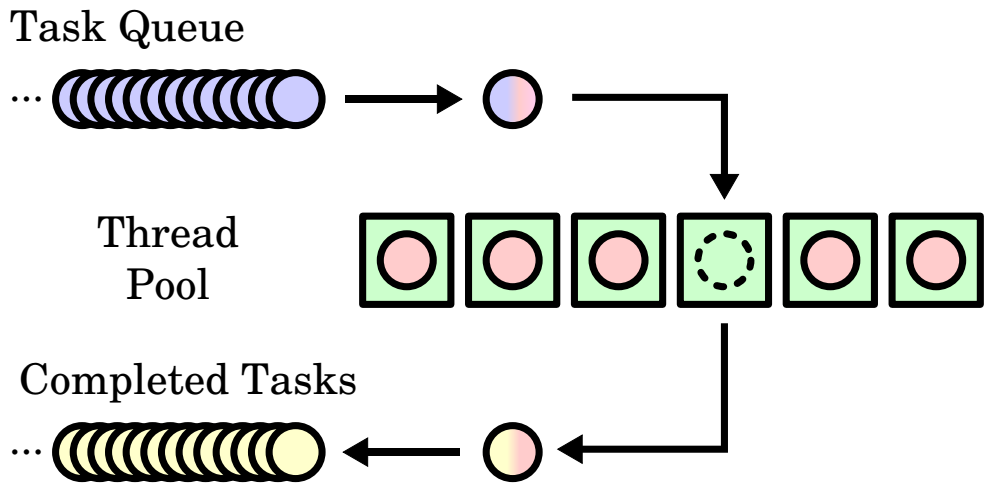
1. Profile the code.
2. Look at hotspots; find and optimize dependencies; parallelize dependency chains; change the algorithm if you can.
3. Estimate benefits.
4. If not good enough, step back and try higher level of abstraction.

Always try to minimize synchronization.

Low-level Implementation Tactic: Thread Pools

Instead of creating threads, destroying them and recreating them, you can use a thread pool.

- It creates *n* threads; you just `push` work onto them.



- Only question is: How many threads should you create? (Experiment to find out).
- Implementation from GLib: `GThreadPool`.

Introduction to Automatic Parallelization

Vision: take a sequential C program and automatically convert it into a parallel version.

Lots of research in the early 1990s, then tapered off.
(it's hard!)

Renewed interest now since multicores are so common.
(it's still hard!)

What Can We Parallelize?

- Some languages are easier than others to reason about (and therefore to automatically parallelize).
- C can be easy to parallelize, given the right code, plus compiler hints.
- "The right code" = arrays with no loop-carried dependencies.

Automatic Parallelization in Practice

Some production compilers support automatic parallelization:

- `icc` (Intel's non-free compiler);
- `solarisstudio` (Oracle's free-as-in-beer compiler [1]);
- `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

We saw automatic parallelization of some code.
Let's revisit the whole issue in Lecture 15.

Part III

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:

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 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?

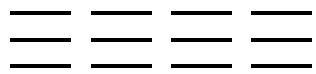
Option 1:

Option 2:

Option 3:

Manually Parallelizing the Example Code

What can we do to parallelize this code?

Option 1: horizontal 

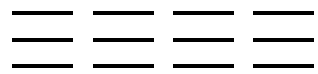
- Create 4 threads; each thread does 1000 iterations on its own sub-array.

Option 2:

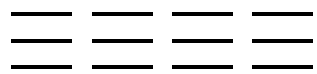
Option 3:

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.

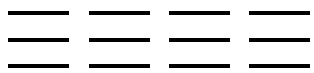
Option 2: bad horizontal 

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

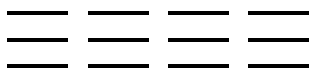
Option 3:

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 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.

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* [2] 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.

[2] `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:

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

n 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:

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.

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 `fploop.c` with `restricts` added, 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", 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.