Fibonnaci OpenMP Tasking

Dr. Christopher Marcotte — *Durham University*

We are likely all familar with the recursive definition of the Fibonacci series due to Pingala in 200 BCE,

$$F_n = \begin{cases} n \text{ if } n < 2 \\ F_{n-1} + F_{n-2} \text{ otherwise} \end{cases}$$

which gives the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946..., beginning from $n = 0$. In lecture I said the structure of this recursion nicely maps onto OpenMP tasks. In this exercise, you will implement this program and analyze the performance characteristics.

A serial implementation of the recursive calculation of the F_n, with n taken from the command line arguments, is given simply by ***fib_serial.c***:

fib_serial.c

```c
#include <stdio.h>
#include <stdlib.h>

// Serial version:
long fib(long n) {
    if (n < 2) return n;
    long x = fib(n - 1);
    long y = fib(n - 2);
    return x+y;
}
int main(int argc,char* argv[]){
    long input = atoi(argv[1]);
    long output = fib(input);
    printf("fib(%ld) = %ld\n", input, output);
    return 0;
}
```

Exercise

Try running ***fib_serial.c*** with different input n. Time the calculation. How does the time vary with n?

Solution

It is hopefully obvious that each call to `long fib(long n)` yields two additional calls: `long fib(long n-1)` and `long fib(long n-2)` – thus the simple recursive approach has exponential time complexity with a base of 2, i.e. $O(2^n)$.

In Figure 1, I report the timing for the serial Fibonnaci code on my laptop, for a few different n. You may find something different on your machine, but the scaling should be similar.

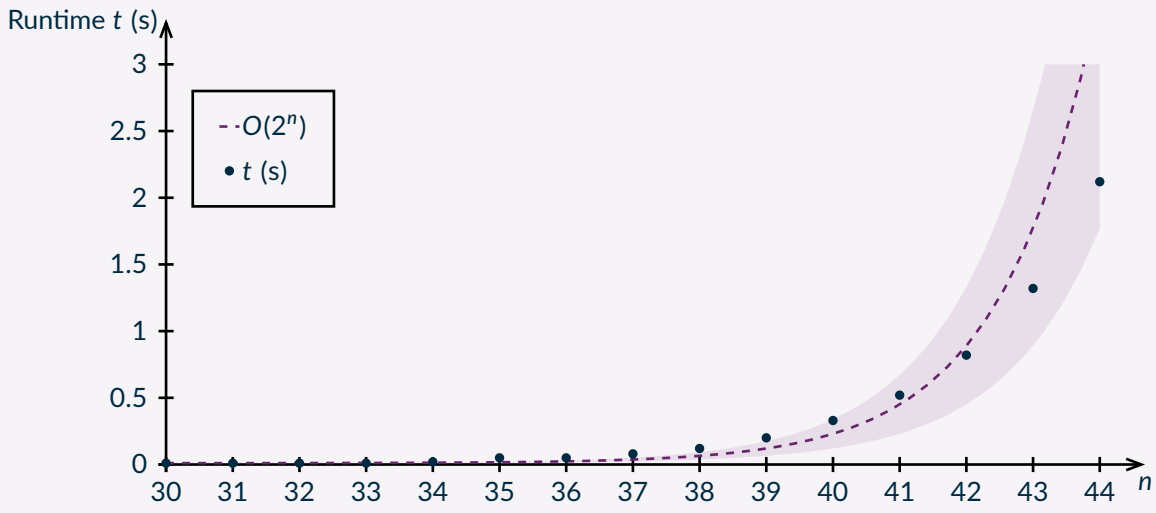

Figure 1: Serial timing for the recursive Fibonnaci calculation.

Obviously, our approach here is not well-considered – this is going to be very slow for not very large n – but it is compact and simple. We'll return to the serial case later, but for now we'll use this as an excuse to try some asynchronous tasking.

Warning

In the following, we will mostly ignore the *correctness* of the Fibonacci calculation in favor of *consistency* – i.e., that different implementations produce the same result rather than precisely F_n.* This is so we can provide larger inputs than we can actually compute F_n for, just to test the actual timing of the computation. We use a `long` type for the inputs and outputs here, because for not especially large inputs n, you may suffer an overflow in the output, and for even larger n you will overflow multiple times. This is only fighting the inevitable, as $F_n \sim a^n$ with $1 < a < 2$, so we'll overflow for an m-bit datatype certainly by the time $n = 2m$, and likely well before.

Indeed, $F_n \equiv \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rceil$ with $\varphi = \left(1+\sqrt{5}\right)/2$ the Golden Ratio – this is the simplest and typically fastest way to calculate F_n.†

OpenMP Tasks

In the lecture I gave a shortened code for how to implement the recursive Fibonnaci sequence using tasks. The `long fib(long n)` and `int main()` functions from that code are given in ***fib_tasks.c***:

fib_tasks.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

long fib(long n) {
    if (n < 2) return n;
```

*Alternatively, think of this exercise as computing F_n over the ring $\mathbb{Z}/m\mathbb{Z}$.

†You may be wondering: 'if F_n has a closed form, then why should we bother with this whole topic anyway – don't we already have a fast, efficient method to compute the sequence?' Keep in mind that this exercise is not about computing the Fibonacci sequence, *per se*; this exercise is about computing an iterative value determined recursively. We could do this same thing in floating point values with the Hénon map, $x_{n+1} = 1 - ax_n^2 + bx_{n-1}$, for example.

```
    long x, y;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait
    return x+y;
}
int main(int argc, char* argv[]){
    long input = atol(argv[1]);
    long output = 0;
    #pragma omp parallel
    {
      #pragma omp single
     {
     output = fib(input);
     }
    }
    printf("fib(%ld) = %ld\n", input, output);
    return 0;
}
```

Exercise

Benchmark ***fib_ser.c*** and ***fib_tasks.c*** for several different inputs *n* and varying the number of `OMP_NUM_THREADS` for the task-based version. What do you find? Is this an efficient calculation? How might you speed up the tasking-based approach?

Solution

An optimized implementation of the Fibonacci tasking program can be seen in ***fib_tasks_opt.c***:

fib_tasks_opt.c

```c
long serfib(long n){
    if (n < 2) return n;
    return serfib(n-1) + serfib(n-2);
}
long fib(long n) {
    if (n < 2){
        return n;
    }
    if (n <= 30){
        return serfib(n);
    }
    long x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
```

```
26     return x+y;
27 }
```

Why is this better performing? Because it *has a larger base case* where anything under $n \leq 30$ runs serially, so for fast enough runs there's no tasking overhead. Where this base case threshold should be set may vary by machine – it is the size below which the recursive serial code is as fast as launching an asynchronous task.

Hint

See these slides for a discussion of one route to speeding up the task-based calculation of Fibonacci numbers.

Dynamic Programming

One composable way of speeding up the recursive calculation directly is to save the intermediate calculations to a buffer, so that we can simply check whether they've already been computed and return them if they have. This means we replace a potentially $O(2^m), m < n$ call sequence with a memory load; this has the cost of n writes, 2^n reads, and n additions. If the buffer is held in-cache, then this is quite fast. This approach is sometimes called *memoization*. This is a very powerful technique when the input space is discrete (e.g. integers) and low-dimensional (e.g. the input `long n` in `long fib(long n)` is one-dimensional).

Exercise

Memoize the original recursive function call to avoid re-computing all $n-1$ Fibonnaci terms when calling `long fib(long n)`. Use the memoized function to determine for which n does F_n overflow the `long` type.

Solution

A truncated serial program demonstrating the memoization of the calculation is shown below.

fib_memo.c

```c
#define N 1024    // note n > 93 will overflow long
static long fib_mem[N];

// Serial version:
long fib(long n) {
   long out, x, y;
   if (n < 2) return n;
   if (fib_mem[n] != 0){
      out = fib_mem[n];
   }else{
    x = fib(n - 2);
    if (n-2 < N){
        fib_mem[n-2] = x;
     }
     y = fib(n - 1);
    if (n-1 < N){
```

```
22          fib_mem[n-1] = y;
23        }
24        out = x+y;
25     }
26     return out;
27 }
```

If you repeat the serial timing exercise with this code, you'll find that it executes pretty quickly, around 20 µs for $n = 900$. This is quite a nice improvement, at the cost of allocating a bit of extra memory and accessing that array out-of-order. If you check carefully, the Fibonacci sequence overflows around $n = 93$, so in practice we should never compute larger ones without some further effort (i.e. `long long` 128 bit integers, or multi-precision integers). Further gains can be made by using a better storage structure, like a hash table, or ensuring the buffer fits in L1 cache. If the bandwidth of the L1 cache is, we expect, the limiting factor, then perhaps we can consider an approach which stays entirely in registers...

Exercise

Add OpenMP tasking to your memoized implementation. What additional concerns around data access arise? Is your implementation race-condition free? Does it suffer from thread contention? Is it faster than the serial memoized approach?

Solution

In addition to each task having it's *thread-local* computation of F_n, it now needs to *read from* and *write to* a *global* array to check for precomputed results. Suddenly, we have a substantially more complex data management issue – the simplest resolution is to preface all accesses to the global array by `#pragma omp atomic [read|write]`, so that if more than one task wishes to access index n in the global array, those tasks are (temporarily) serialized.

fib_memo_tasks.c

```c
11 long fib(long n) {
12    long x, y, tmp;
13    if (n < 2) return n;
14    #pragma omp private(tmp,n) atomic read
15    tmp = fib_mem[n];
16    //printf("n = %ld, tmp = %ld\n", n, tmp);
17    if (tmp != 0) {
18     return tmp;
19    }else if (tmp == 0){
20       #pragma omp task shared(x)
21       x = fib(n - 2);
22       #pragma omp task shared(y)
23      y = fib(n - 1);
24       #pragma omp taskwait
25       //printf("(x,y) = (%ld,%ld)\n", x, y);
26     #pragma omp atomic write
27       fib_mem[n] = x+y;
```

```
28      return x+y;
29    }
30 }
```

Asymptotic Optimization

Obviously, $O(\log_2 n) \ll O(n) \ll O(2^n)$, so one might suspect that to truly optimize this Fibonnaci calculation, we should stop using tail-recursion altogether[‡]. However, the asymptotically fastest way to calculate the Fibonacci sequence is the *fast doubling* method, $O(\log_2 n)$; given F_k and F_{k+1}:

$$F_{2k} = F_k(2F_{k+1} - F_k), \tag{1.1}$$

$$F_{2k+1} = F_k^2 + F_{k+1}^2. \tag{1.2}$$

This method generates the $2k$ and $2k+1$ Fibonacci values from F_k and F_{k+1}, skipping the intermediate values. By applying this recursively, we save a lot of intermediate calculation.

Exercise

Implement Equation 1 in a serial C code, and benchmark it against the original implementation for several values of n. Is it faster?

Solution

You can find an implementation of the fast doubling method in ***fib_fast.c***:

fib_fast.c

```c
long fib_skip(long n) {
  if (n < (long)(2)) return (long)(n);
  long a = fib_skip(n / 2);
  long b = a + fib_skip((n / 2) - 1);
  long c = a * (2*b - a);
  long d = a * a + b * b;
  if (n%2 == 0){
    return c;
  }else{
    return d;
  }
}
```

along with an approach without recursion, which skips no intermediate calculations:

fib_fast.c

```c
long fib_noskip(long n){
  if (n < (long)(2)) return (long)(n);
```

[‡]See the warning, above.

```
9     long a = 1, b = 1, c = 0;
10    for (long m = 3; m <= n; m++){
11      c = a;
12      a = a + b;
13      b = c;
14    }
15    return a;
16  }
```

For $n \lesssim 10^3$, these take about the same time to complete, ~ 5 µs.

Summary

So what was the point of all this? In part, to give you a chance to practice with tasking in OpenMP in a way that is very legible and classically computer science. On the other hand, an important lesson to take from this exercise is that speeding up *parallel* code is often a matter of speeding up *serial* code – we added completely arbitrary, asynchronous tasking to our problem, but because we couldn't utilize the compute power of a modern CPU, it only made things worse. Read on for a slightly different take.

A Python Divergence

Python is often a very slow language due to the interpreted nature of the execution, and the global interpreter lock. I would not recommend you write any performance critical code in Python, though many people do.

One thing Python is exceptionally useful for is *arbitrary precision integer arithmetic* – i.e., Python integers "[...] have unlimited precision". One typically pays a performance cost for such indulgences, but in Python every integer operation takes the slow path, so not much is lost.

Challenge

Implement the serial *C* functions in Python, and identify how large you can make n in Python and still get a correct answer. Consider whether these are more or less performant than the *C* functions.

Solution

I will spare you the rewrite and comparison of the Python and *C* functions; suffice it to say Python is more slow than *C* but capable of producing F_n correctly for much larger n than we can with a standard `long`. You can check the upper limit very easily in Python:

fib.py

```py
import math
def fib(n):
  phi = (1.0 + math.sqrt(5.0)) * 0.5
  return round(pow(phi, n) / math.sqrt(5.0))
n = 0
while (True):
```

```
    try:
      fn = fib(n)
      n+=1
    except:
      print(f"(n,F_n) = ({n-1}, {fib(n-1)})")
      break
```

for which I get $n = 1474$, and $F_{1474} \approx 5 \times 10^{307}$ or something.

A Julian Divergence

Julia, in particular, is unusually bad at recognizing and optimizing tail-recursion.[§] So bad, in fact, that the tail-recursive serial code will be *slower* than a task-based approach; this is a reversal of the situation in *C*, where the compiler will recognize tail-recursion and optimize it into a tight, efficient, loop.

I whipped up an analogous (serial, non-tasking) function in Julia:

fib_examples.jl

```jl
function fib(n::T) where {T<:Integer}
   if n < 2
      return n
   else
      return fib(n - 1) + fib(n - 2)
   end
end
```

With `using BenchmarkTools`, we can benchmark the function with `@benchmark fib($n)`.

Challenge

With the serial Fibonnaci Julia code above:

- Add tasking to the Fibonnaci Julia function by using `Threads.@spawn` and `fetch()`, in analogy to the OpenMP code.
- Implement the dynamic programming approach (memoization) to the serial Fibonnaci sequence calculation, and benchmark it against your optimized tasking solution.
- Implement the fast loop and fast doubling methods in Julia, and compare them to the other approaches.

With all the comparable implementations, benchmark your Julia and *C* implementations for some values of *n*, and compare them. What are some advantages of the *C* implementations? What advantages might the Julia implementations have?

Solution

You can add tasking to the julia function pretty simply, in direct analogy to the OpenMP tasking approach.

[§]Note: tail-recursion is a loop in disguise, and the transformation to the loop from the tail-recursion can be trivially done with a macro in Julia.

fib_examples.jl

```jl
function fib_tasks(n::T) where {T<:Integer}
   if n < 2
      return n
   else
      a = Threads.@spawn fib(n - 1)
      b = Threads.@spawn fib(n - 2)
      return fetch(a) + fetch(b)
   end
end
```

It's also very straight-forward to implement the tight-loop implementation,

fib_examples.jl

```jl
function fib_loop(n::T) where {T<:Integer}
   a = 0
   b = 1
   c = 0
   for m in 1:n
      c = a
      a = a + b
      b = c
   end
   return a
end
```

and the doubling-recursion.

fib_examples.jl

```jl
function fib_doubler(n::T) where {T<:Integer}
  if n < 2
    return n
  else
    a = fib_doubler(n ÷ 2)
    b = a + fib_doubler((n ÷ 2) - 1)
    c = a * (2b - a)
    d = a * a + b * b
    if iseven(n)
      return c
    else
      return d
    end
  end
end
```

By loading a memoization library `using Memoization` and adding the macro `@memoize` before the function `function fib_memo(n::T) where {T<:Integer}`, ***we can calculate the*** $n = 900$ ***case in roughly*** 125 ***ns.***¶

¶Obviously, memoization is cheating, in some sense, because we amortize the cost of the current calculation by front-loading it in an earlier calculation.

fib_examples.jl

```jl
@memoize function fib_memo(n::T) where {T<:Integer}
    if n < 2
        return n
    else
        return fib_memo(n - 1) + fib_memo(n - 2)
    end
end
```

I time the different functions for many n using the `@belapsed` macro, and plot these timings below.

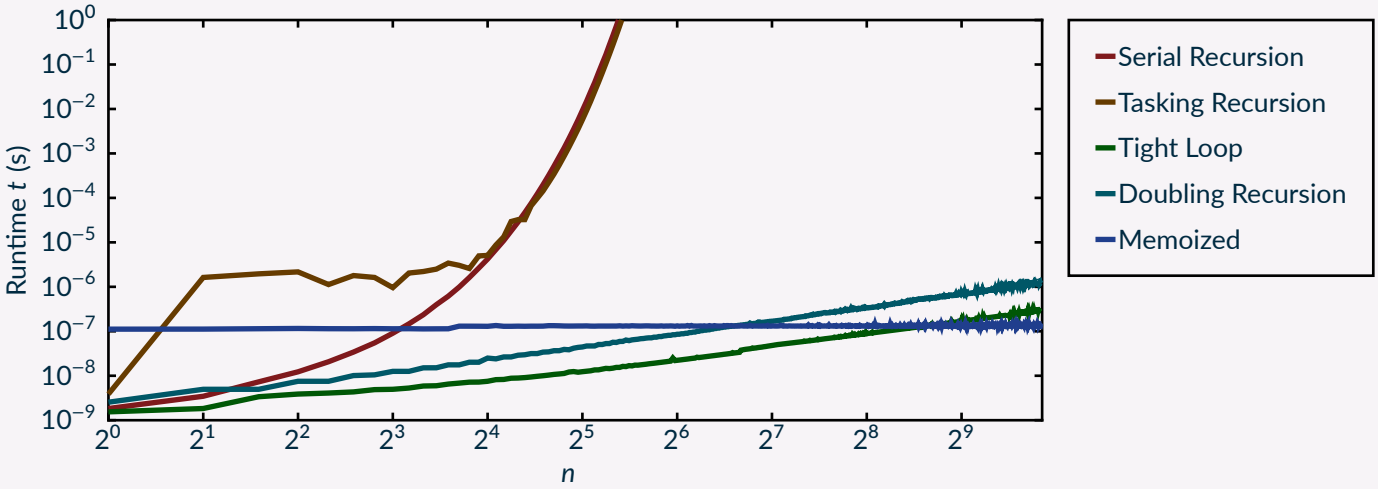

Figure 2: Timing of the various julia methods for computing the Fibonacci sequence.

One notable takeaway from Figure 2 is that the fastest method asymptotically (for large n) is not the fastest method for all n – the fastest method changes around $n \lesssim 2^9$, and the ranking of the methods changes at $n \approx 2^{0.5}, 2^1, 2^3, 2^{6.5}, 2^9$.

Since some of these methods are stateful – i.e., the memoization calls on additional memory, so calculating F_{n+1} is only two lookups and an add if F_{n-1} and F_n were computed beforehand – it is more fair to shuffle (i.e. randomly permute) the n we select for each timing in the sequence. These results are shown in Figure 3, where we note that the results are... basically the same as for the sequential timing results.

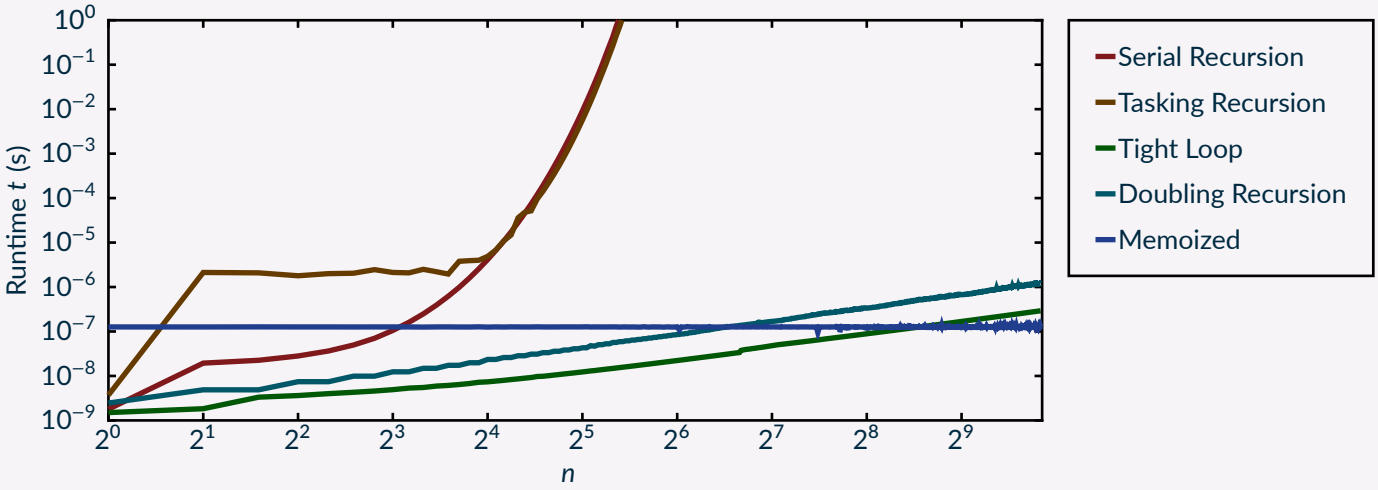

Figure 3: Timing of the various julia methods for computing the Fibonacci sequence with shuffled n sequence.

With the solutions, it should be trivial to try testing the performance of the *C* equivalents to the Julia functions. This is left as an exercise for the reader. One advantage of the *C* implementations is a svelte and tiny memory usage – some of these function calls only work with three `long`s, and will take less than 512 bytes to compute F_n for *any* n. In contrast, Julia has a hefty runtime which affords some conveniences but also raises the floor for the smallest non-trivial executables. One advantage of the Julia implementations is that they will work with 128 bit integers transparently (`Int128`s, or equivalently `long long`s), allowing us to compute a larger range of F_n with the same function; indeed, we can easily compute arbitrarily large F_n by giving a `BigFloat` as input, similarly to Python, but faster in the cases where it can be.

Aims

- Foundational understanding of task use in OpenMP
- Critical understanding of the performance tradeoffs of tasking
- Application experience of tasking-based concurrency