Outline of this lecture:

1. Finish Delay Seq. Argument
2. Implementation of Cilk 5
   (a) Work-first principle
   (b) Shadow Frames
   (c) Two-clone compilation strategy

1 Finish Delay Seq. Argument

To finish up on last class we are looking to bound the number of steals by workers, which leaves us with the inequality

\[ Pr[\text{At least } 2P(2T_{\infty} + R) \text{ steal attempts}] \leq Pr[\text{Exists some } (U, R, T_1)] \leq (\# \text{ of delay seq.})(\text{Max probability that my } (U, R, T) \text{ occurs}) \]

\[ = 2^{2T_{\infty}} \left( \frac{2T_{\infty} + R}{2T_{\infty}} \right)^{e^{-R}} \leq \left( \frac{2e(2T_{\infty} + R)}{2T_{\infty}} \right)^{2T_{\infty}} e^{-R} \leq \left( \frac{2e(1+\epsilon)^{\frac{1}{\epsilon}}}{e} \right)^{R} \]

So if we let \( \epsilon = \left( \frac{2e(1+\epsilon)^{\frac{1}{\epsilon}}} {e} \right)^{R} \), then \( Pr[\text{Greater than } 2P(2T_{\infty} + R) \text{ steal attempts occur}] \leq \epsilon \), when \( R = \Omega(lg(\frac{1}{\epsilon})) \). And with probability \( 1 - \epsilon \),

\[ T_p \leq \frac{T_1}{p} + O \left( T_{\infty} + \log \left( \frac{1}{\epsilon} \right) \right) \]

When an application has ample parallelism, the time bound shows we will get linear speedup.

Parallelism: \( \frac{T_1}{T_{\infty}} \)    Ample Parallelism: \( p << \frac{T_1}{T_{\infty}} \)
2 Implementation of Cilk 5

Work-first Principle

Cilk-5’s scheduling strategy conforms to the work-first principle [1], which states that, one should borne the scheduling overhead on the critical path term instead of on the work term. Cilk-5’s work-stealing scheduler has the following time bound:

\[
T_P \leq \frac{C_1 T_S}{p} + C_\infty T_\infty + O(lg[...])
\]

In other words, we want to put the scheduling overhead on the \(C_\infty T_\infty\) term, not \(\frac{C_1 T_S}{p}\). (\(T_S\) is the serial execution time.)

For a system that employs work-stealing, the system needs to enable the following operations:

- At spawn, allow the continuation of the parent functions to be stolen.
- Allow a thief to resume a stolen function at the right program point with the most up-to-date values for local variables.
- At sync, check if there are any outstanding children.
- A function returning can find its parent to return to if the parent is stolen.

Shadow Frames

Cilk-5 uses heap-based shadow frames as a book-keeping mechanism for workers to keep track of information when stealing and returning. A single shadow frame keeps

- Up-to-date value of local variables
- Join counter (the number of outstanding children of a frame)
- Entry number (similar to a program counter to allow a thief to resume at the right program point)

Compilation of Cilk 5

Cilk-5 uses Figure 1 for compilation steps. Starting with the Cilk source code, there is a variety of compilation and linking steps until the full binary is made.

During compilation, each function is compiled into two versions of the function, a slow clone and a fast clone. A slow clone contains all the necessary bookkeeping information to enable parallel execution, so a function’s slow clone is only invoked once the function is stolen. A fast clone is always invoked before a function is stolen; while it maintains some bookkeeping data, such as the flushing the most up-to-date values for local variables onto the shadow frame, it does allow parallel execution.

Based on how work-stealing works (always steal from top of a deque), a slow clone will be on top of some worker’s deque. Each function call after the slow clone will be a function call to a fast clone. A slow clone also contains extra bookkeeping to keep track of parents. Below is an example
from "Cilk 5.4.6 Reference Manual" of a fast clone and slow clone using our fib() example [2].
struct _fib_frame{
    StackFrame header;
    struct {int n;} scope0;
    struct {int x;int y;} scope1;
};

int fib(int n)
{
    struct _fib_frame *frame;
    _INIT_FRAME(_frame,sizeof(struct _fib_frame),_fib_sig);
    {
        if (n < 2)
        {
            _BEFORE_RETURN_FAST();return (n);}

    }
    else
    {
        int x;int y;
        { _frame->header.entry=1;
            _frame->scope0.n=n;
            x=fib(n-1);
            _XPOP_FRAME_RESULT(_frame,0,x);
        }
        { _frame->header.entry=2;
            _frame->scope1.x=x;
            y=fib(n-2);
            _XPOP_FRAME_RESULT(_frame,0,y);
        }
        /* sync */;
        {
            _BEFORE_RETURN_FAST();return (x+y);
        }
    }
}


static void _fib_slow(struct _fib_frame *frame)
{
    int n;
    switch (_frame->header.entry) {
    case 1: goto _sync1;
    case 2: goto _sync2;
    case 3: goto _sync3;
    }
    n=_frame->scope0.n
    if (n < 2)
    {
        _SET_RESULT((n)); _BEFORE_RETURN_SLOW(); return;
    }
    else {
        int _temp0;
        _frame->header.entry=1;
        _frame->scope0.n=n;
        _temp0=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,/* return nothing */, _temp0);
        _frame->scope1.x=_temp0;
        if (0) { _sync1; n=_frame->scope.n }
    }
    int _temp1;
    _frame->header.entry=2;
    _temp1=fib(n-2);
    _XPOP_FRAME_RESULT(_frame,/* return nothing */, _temp1);
    _frame->scope1.y=_temp1;
    if (0) { _sync2; }
    }
    _frame->header.entry=3;
    if (!_SYNC) {
        return;
        _sync3;
    }
    }
    _SET_RESULT(_frame->scope1.x+_frame->scope1.y);
    _BEFORE_RETURN_SLOW(); return;
}
}
References
