Principles of Concurrency and Parallelism

Lecture 10: Cilk and

Workstealing Schedulers

4/3/12

So far ...

- Erlang:
  - functional
  - message-passing
  - language-primitives for communication, synchronization,...
- Posix
  - library
  - C-based
- In this lecture:
  - Cilk
    - C-based
    - language primitives for communication, synchronization,...

Cilk

```
cilk int fib (int n) {

  int n1, n2;

  if (n < 2) return n;
  else {
    n1 = spawn fib(n-1);
    n2 = spawn fib(n-2);
    sync;
    return (n1 + n2);
  }
}
```

spawn: procedure call can execute asychronously with the caller

sync: join point: current thread waits for all locally-spawned tasks to complete
procedures never terminate while they have outstanding (spawned) children

Logical parallelism:
Cilk does not mandate creation of threads or mapping tasks to processes

Cilk

- Faithful extension to C
  - eliding Cilk keywords leads to a serial C program
- Features
  - spawn keyword can only be applied to a Cilk function
    - cannot be used within a C function
  - Cilk functions cannot be called with normal C conventions
    - must be called with a spawn and waited for by a sync

Terminology

- Thread: maximal sequence of instructions not containing spawn, sync, return, etc.

```
cilk int fib (int n) {

  int n1, n2;

  if (n < 2) return n;
  else {
    n1 = spawn fib(n-1);
    n2 = spawn fib(n-2);
    sync;
    return (n1 + n2);
  }
}
```

Thread A: if statement upto first spawn

Thread B: computation of n-2 before second spawn

Thread C: n1+n2 before return

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>
cilk double sum(int L, int U)
{
  if (L == U) return L;
  else {
    double lower, upper;
    int mid = (U+L)/2;
    lower = spawn sum(L, mid);
    upper = spawn sum(mid+ 1, U);
    sync;
    return (lower + upper);
  }
}
```

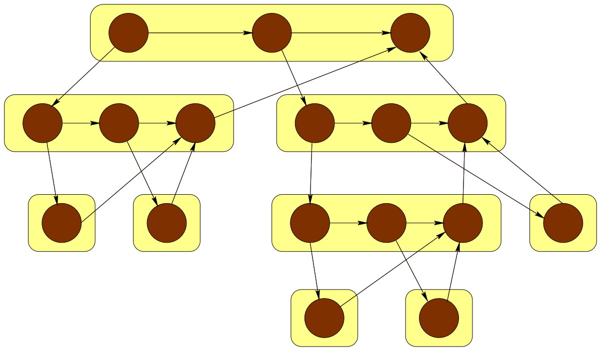
```
cilk int main(int argc, char *argv[])
{
  int n;
  double result;
  n = atoi(argv[1]);
  if (n <= 0) {
    printf("'n␣=␣%d:'"
      "n␣must␣be␣positive\n",n);
  } else {
    result = spawn sum(1, n);
    sync;
    printf("Result:␣%lf\n", result);
  }
  return 0;
}
```

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>
int * v = 0;
cilk double sum(int L, int U)
{
  if (L == U) return v[L];
  else {
    double lower, upper;
    int mid = (U + L)/2;
    lower = spawn sum(L, mid);
    upper = spawn sum(mid+ 1, U);
    sync;
    return (lower + upper);
  }
}
```

```
cilk void
init(int L, int U)
{
  if (L == U) v[L] = L + 1;
  else {
    int mid = (U + L)/2;
    spawn init(L, mid);
    spawn init(mid + 1, U);
    sync;
  }
}
cilk int main(int argc, char *argv[])
{
  int n; double result; n = atoi(argv[1]);
  v = malloc(sizeof(int) * n);
  spawn init(0, n-1); sync;
  result = spawn sum(0, n-1); sync;
  free(v);
  printf("Result:␣%lf\n", result);
  return 0;
}
```

Model



A Cilk procedure is broken into a sequence of threads (circles)

Downward edges indicate spawning of a new subcomputation

Horizontal edges indicate control transfer (continuation) to successor thread

Upward edge indicates returning a value to a parent procedure

Cilk and C

- Source-to-source compiler
- C functions cannot directly spawn or call Cilk procedures
  - Use automatically-generated stub functions for this purpose
  - A Cilk context entails allocating OS resources (e.g., threads)

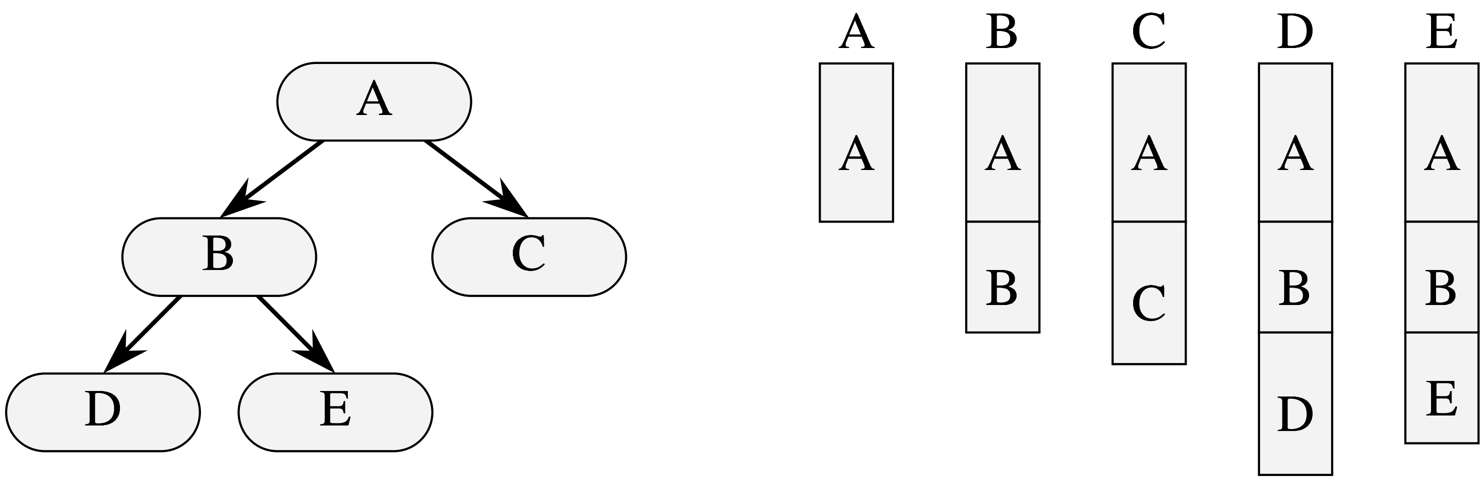
Example

```c
#include <cilk.h>

cilk float g (double x)
{
    /* do something */
}

cilk void h (int i)
{
    float y;

    y = spawn g (2.7);
    sync;
}

int main (int argc, char *argv[])
{
    float        y;
    CilkContext* context;

    context = Cilk_init (&argc, argv);

    y = EXPORT(g) (context, 3.14);

    Cilk_terminate (context);

    return 0;
}
```

```c
#include <cilk.h>

extern float EXPORT(g) (CilkContext* context,
                        double x);

void f ()
{
    char* argv[] = { "f", "--nproc", "4", 0 };
    int   argc   = 3;

    float        y;
    double       x = 0.0;
    CilkContext* context;

    context = Cilk_init (&argc, argv);

    y = EXPORT(g) (&x);

    Cilk_terminate (context);
}
```

(b)

Storage



Cactus stack

Sharing and Races

```
cilk int foo (void)
{
  int x = 0, y;

  spawn bar(&x);
  y = x + 1;
  sync;
  return (y);
}

cilk void bar (int *px)
{
  printf("%d", *px + 1);
  return;
}
```

```
cilk int foo (void)
{
  int x = 0;

  spawn bar(&x);
  x = x + 1;
  sync;
  return (x);
}

cilk void bar (int *px)
{
  *px = *px + 1;
  return;
}
```

Inlets

```
cilk int fib (int n)
{
    int x = 0;
    inlet void summer (int result)
    {
        x += result;
        return;
    }

    if (n<2) return n;
    else {
        summer(spawn fib (n-1));
        summer(spawn fib (n-2));
        sync;
        return (x);
    }
}
```

Inlets guaranteed to execute atomically

Programming Model

- View computation as a DAG
  - a thread cannot be executed until all threads on which it depends have completed.
  - Dependency between threads assigned to different processors requires communication
- Key challenge:
  - Efficient scheduling of threads
  - Work-stealing: when a processor runs out of work, ask another processor for work.

Work Stealing

- Locally, a processor executes procedures in ordinary serial order
  - explore the spawn tree in a depth-first manner
  - when a child procedure is spawned, save the parent's continuation (context) at the bottom of the stack
    - stacks grow downwards
    - start commencing work on the child
  - when another processor "steals" work, it steals from the top of the stack
    - least recent

Performance Model

- What are the fundamental limits that guide how fast a Cilk computation can run?
  - T_P : Execution time of a computation on P processors
  - T_1: Total time needed to execute all threads comprising the task tree (DAG). Refer to this as *work*.
    - Lower bound: $T_P \geq T_1/P$
  - Program's *span*: T_∞
    - Execution time of computation on an infinite number of processors
    - Time needed to execute threads along longest dependency path
    - $T_P \geq T_\infty$

Performance Model

$$T_P \approx T_1/P + T_\infty$$

Critical path overhead:

$$T_P \leq T_1/P + c_\infty T_\infty \ .$$

Parallelism

$$\overline{P} \ = \ T_1/T_\infty$$

Average amount of work for every step taken along the span
When $P \ll \overline{P}$ then $T_P \approx T_1/P$

Compilation

- Generate two copies of a procedure
  - fast clone: behaves like the Cilk-elided version with no support for parallelism
  - slow clone: full support for parallelism
- Each processor (*worker*) maintains a dequeue (doubly-ended queue) of ready (runnable) procedures
  - The worker operates locally on the tail treating it much like a call stack
  - When a worker runs out of work, it steals work from the the head of the victim's dequeue.

Clones

- When a procedure is spawned, the fast clone runs.
- When a thief steals a procedure, the procedure is converted to a slow clone.
  - Fast clones never stolen
  - No descendents of a fast clone ever stolen
    - stealing from the head guarantees that parents are stolen before their children
    - sync statements in the fast clone are no-ops
- Slow clone -
  - use a goto to restore the program counter and local variables from the frame

Fast Clone

```
int fib (int n)
{
    fib_frame *f;                   frame pointer
    f = alloc(sizeof(*f));          allocate frame
    f->sig = fib_sig;               initialize frame
    if (n<2) {
        free(f, sizeof(*f));        free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;               save PC
        f->n = n;                   save live vars
        *T = f;                     store frame pointer
        push();                     push frame
        x = fib (n-1);              do C call
        if (pop(x) == FAILURE)      pop frame
            return 0;               frame stolen
        ...                         second spawn
        ;                           sync is free!
        free(f, sizeof(*f));        free frame
        return (x+y);
    }
}
```

Microscheduler

- Schedules procedures across a fixed set of processors
- Executes slow clone
  - Receives pointer to frame as argument
    - args and local state inside frame
    - restores program counter
    - sync waits for children

Protocol

- Shared memory deque
  - T: first unused
  - H: head
  - E: exception
- Work-first
  - move costs from worker to thief
- One worker per deque
- One thief at a time
  - enforced by lock

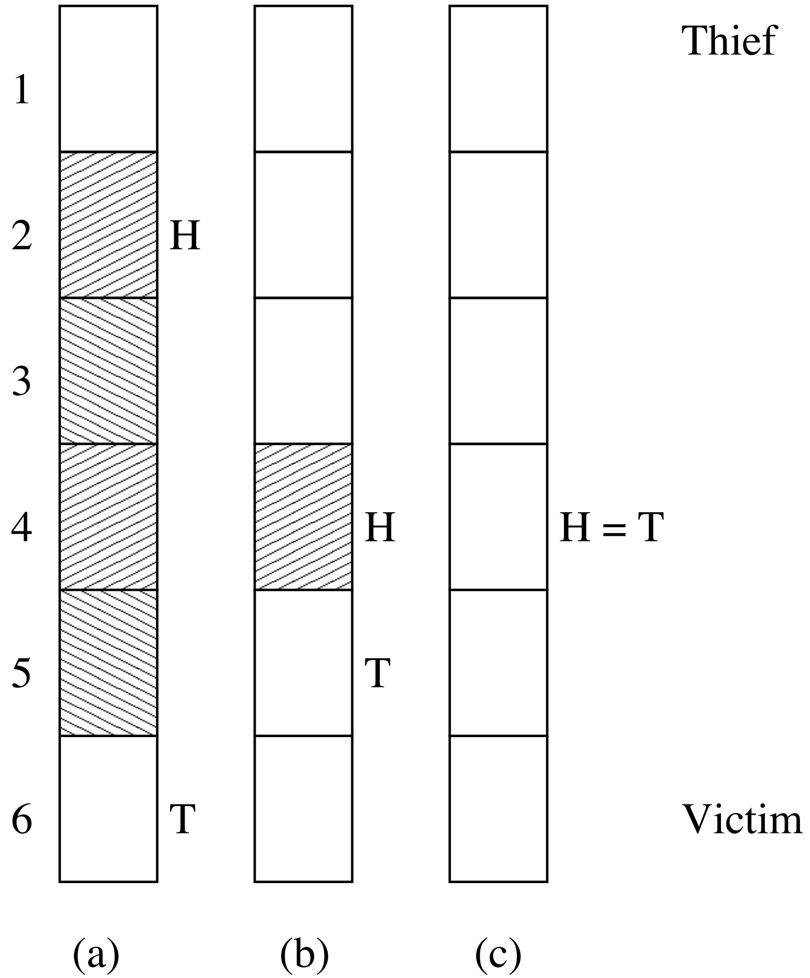
Worker/Victim

```
push() {
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

Stealing



Threaded Building Blocks (TBB)

- Set of library templates
- Aim to reduce some of the low-level reasoning needed to effectively program Posix threads
- Tasks vs threads
  - Inspired by Cilk work-stealing scheduler

Example: parallel-for

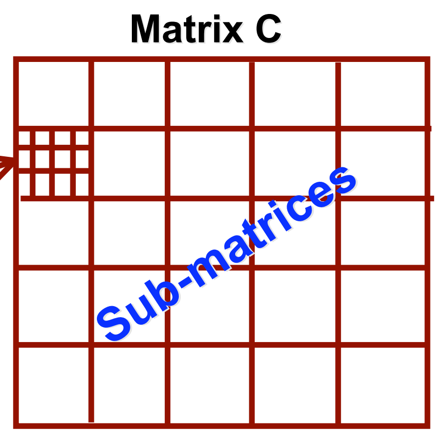
```
void SerialMatrixMultiply( float c[M][N], float a[M][L], float b[L][N] )
{
    for( size_t i=0; i<M; ++i ) {
        for( size_t j=0; j<N; ++j ) {
            float sum = 0;
            for( size_t k=0; k<L; ++k )
                sum += a[i][k]*b[k][j];
            c[i][j] = sum;
        }
    }
}
```

Example: parallel-for

```
#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"

// Initialize task scheduler
tbb::task_scheduler_init tbb_init;

// Do the multiplication on submatrices of size ≈ 32x32
tbb::parallel_for ( blocked_range2d<size_t>(0, N, 32, 0, N, 32),
                    MatrixMultiplyBody2D(c,a,b) );
```

Example: parallel-for

```
class MatrixMultiplyBody2D {
    float (*my_a)[L], (*my_b)[N], (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a;   // a,b,c used in example to emphasize
        float (*b)[N] = my_b;   // commonality with serial code
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i )
            for( size_t j=r.cols().begin(); j!=r.cols().end(); ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
    }

MatrixMultiplyBody2D( float c[M][N], float a[M][L], float b[L][N] ) :
        my_a(a), my_b(b), my_c(c) {}
};
```

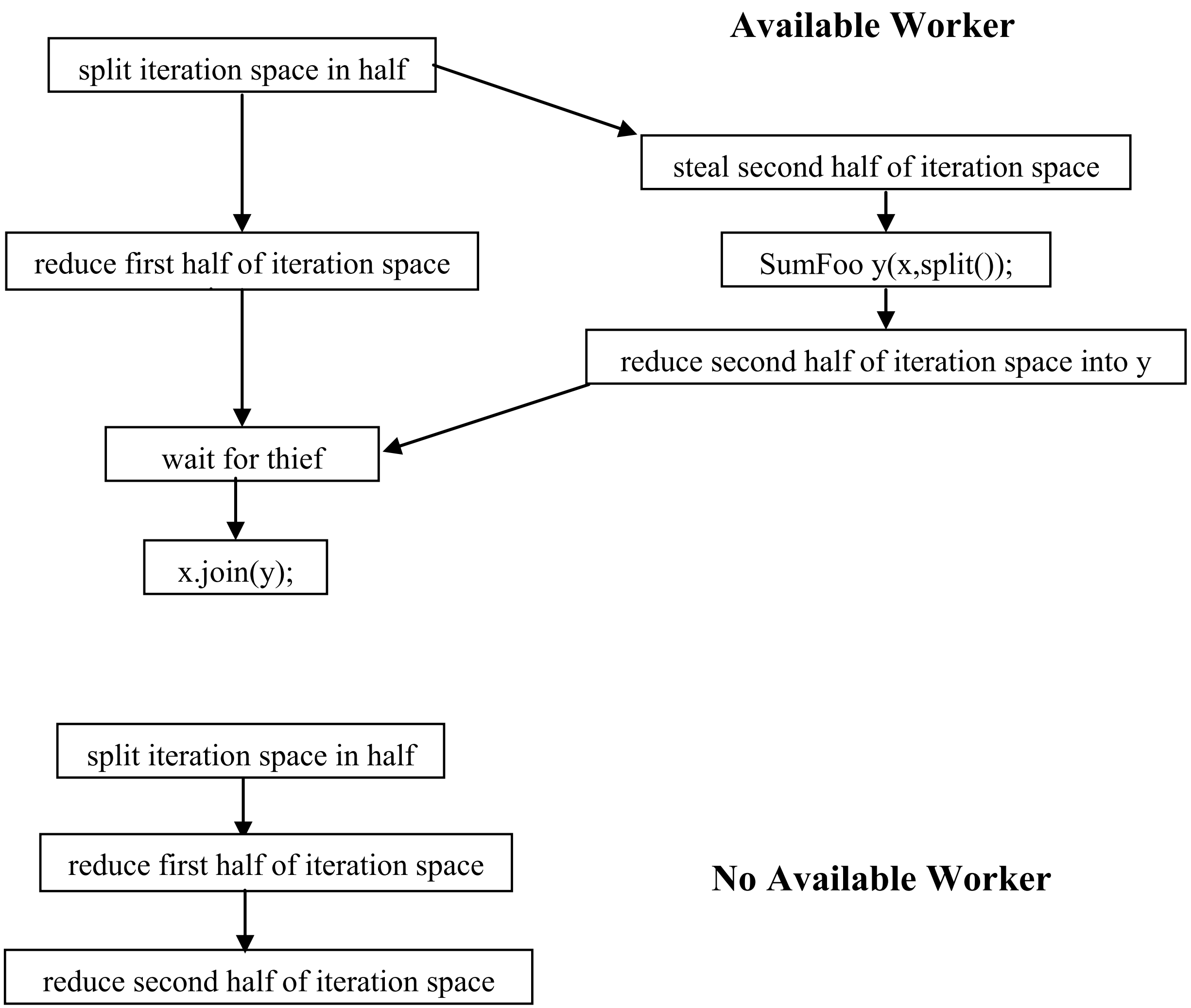


Example: parallel-reduce

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce( blocked_range<size_t>(0,n), sf );
    return sf.my_sum;
}
```

Splitting and Joining



Example: parallel-reduce

```
class SumFoo {
    float* my_a;
public:
    float my_sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        float sum = my_sum;
        size_t end = r.end();
        for( size_t i=r.begin(); i!=end; ++i )
            sum += Foo(a[i]);
        my_sum = sum;
    }

    SumFoo( SumFoo& x, split ) : my_a(x.my_a), my_sum(0) {}

    void join( const SumFoo& y ) {my_sum+=y.my_sum;}

    SumFoo(float a[] ) :
        my_a(a), my_sum(0)
```