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 asynchronously 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 up to first spawn
- Thread B: computation of n-2 before second spawn
- Thread C: n1+n2 before return
#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\n must be positive\n", n);
    } else {
        result = spawn sum(1, n);
        sync;
        printf("Result: %lf\n", result);
    }
    return 0;
}
Example

```c
#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;
}
```
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);
}
```

(a)

(b)
2.5. SHARED MEMORY

Figure 2.5: A cactus stack. Procedure A spawns B and C, and B spawns D and E. The left part of the figure shows the spawn tree, and the right part of the figure shows the view of the stack by the five procedures. (The stack grows downward.)

Since the object will be deallocated automatically when the child returns. Similarly, sibling procedures cannot reference each other’s local variables. Just as with the C stack, pointers to objects allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree.

You can allocate size bytes of storage on the stack by calling the C library function Cilk_alloca:

```
ptr = Cilk_alloca(size);
```

Memory allocated by Cilk_alloca is freed when the procedure in which it was called returns.

In the current release, Cilk’s version of Cilk_alloca does not work properly when it is called from within a C function. Similarly, the C function alloca does not work properly when called within a Cilk procedure.

2.4.2 Heap memory

To allocate heap storage, you call

```
ptr = malloc(size);
```

which allocates size bytes out of heap storage and returns a pointer to the allocated memory. The memory is not cleared. Heap storage persists until it is explicitly freed:

```
free(ptr);
```

where ptr is a pointer previously returned by a call to malloc. Unlike storage allocated by Cilk_alloca, a pointer to storage allocated by malloc can be safely passed from a child procedure to its parent.

2.5 Shared memory

Cilk supports shared memory. Sharing occurs when a global variable is accessed by procedures operating in parallel, but sharing can also arise indirectly from the passing of pointers to spawned procedures, allowing more than one procedure to access the object addressed by the pointer. (Cilk supports the same semantics for pointer passing as C. See Section 2.4.) Updating shared objects...
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: $T_\infty$
  - 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

\[ T_P \approx \frac{T_1}{P} + T_\infty \]

Critical path overhead:

\[ T_P \leq \frac{T_1}{P} + c_\infty T_\infty . \]

Parallelism

\[ \overline{P} = \frac{T_1}{T_\infty} \]

Average amount of work for every step taken along the span

When \( P \ll \overline{P} \) then \( T_P \approx \frac{T_1}{P} \)
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 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
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)  // frame stolen
            return 0;
        ...  // 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

```c
Worker/Victim
1  push() {        Thief
2    T++;        1  steal() {
3  }
4  pop() {        2    lock(L);
5    T--;        3    H++;
6    if (H > T) {
7        T++;        4    if (H > T) {
8        lock(L);        5        H--;
9        T--;        6        unlock(L);
10        if (H > T) {
11            T++;        7        return FAILURE;
12            unlock(L);        8        }
13            return FAILURE;
14        }
15        unlock(L);
16    }
17    return SUCCESS;
18 }
```

CS390C: Principles of Concurrency and Parallelism
Stealing

(a)  
1  
2  
3  
4  
5  
6  

(b)  

1  
2  
3  
4  
5  
6  

(c)  

1  
2  
3  
4  
5  
6  

Thief

H = T

Victim

CS390C: Principles of Concurrency and Parallelism
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

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

// Initialize task scheduler
using namespace tbb;
tbb::task_scheduler_init tbb_init;

// Do the multiplication on submatrices of size \approx 32x32
using blocked_range2d = blocked_range2d<size_t>;
tbb::parallel_for ( blocked_range2d(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

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

```cpp
def 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

Available Worker

- split iteration space in half
  - reduce first half of iteration space
    - wait for thief
      - x.join(y);

- steal second half of iteration space
  - SumFoo y(x,split());
  - reduce second half of iteration space into y

No Available Worker

- split iteration space in half
- reduce first half of iteration space
- reduce second half of iteration space

CAUTION: Because split/join are not used if workers are unavailable, parallel_reduce does not necessarily do recursive splitting.

CAUTION: Because the same body might be used to accumulate multiple subranges, it is critical that operator() not discard earlier accumulations. The code below shows an incorrect definition of SumFoo::operator().

```cpp
class SumFoo {
    ...
    operator() { /* Incorrect definition */ }
};
```
Example: parallel-reduce

```cpp
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)
};
```

Parallelizing Simple Loops

Tutorial 2

A large subrange might use cache inefficiently. For example, suppose the processing of a subrange involves repeated sweeps over the same memory locations. Keeping the subrange below a limit might enable the repeated referenced memory locations to fit in cache. See the use of `parallel_reduce` in `examples/parallel_reduce/primes/primes.cpp` for an example of this scenario.

A loop can do reduction, as in this summation:

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

If the iterations are independent, you can parallelize this loop using the template class `parallel_reduce` as follows:

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

The class `SumFoo` specifies details of the reduction, such as how to accumulate subsums and combine them. Here is the definition of class `SumFoo`:

```cpp
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)
};
```