CSE502: Foundations of Parallel Programming

Lecture 15: Cilk Language & Runtime

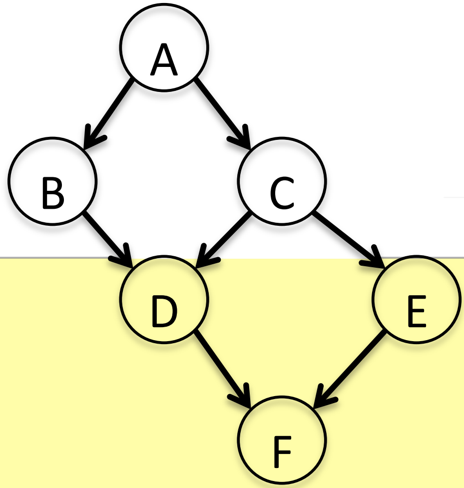
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

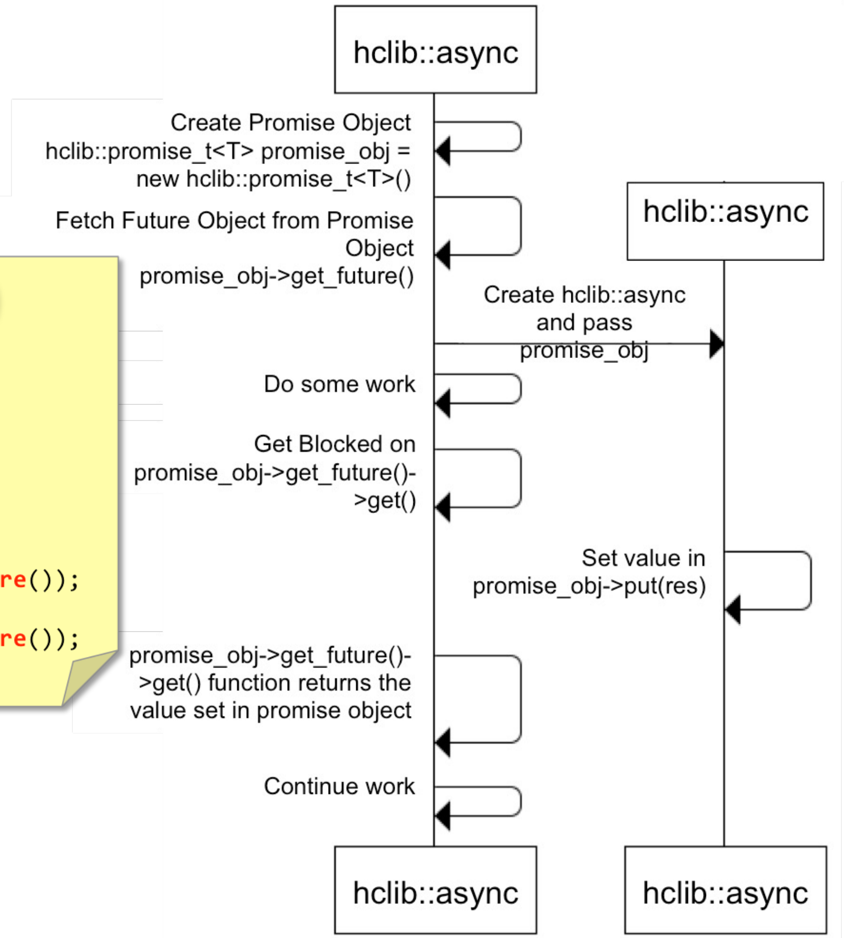
vivekk@iiitd.ac.in

Last Class




```
promise_t<void>* prom_A = new promise_t<void>();
promise_t<void>* prom_B = new promise_t<void>();
promise_t<void>* prom_C = new promise_t<void>();
promise_t<void>* prom_D = new promise_t<void>();
promise_t<void>* prom_E = new promise_t<void>();
promise_t<void>* prom_F = new promise_t<void>();
finish([=]() {
  async([=]() { A(); prom_A->put(); });
  async_await([=]() { B(); prom_B->put(); }, prom_A->get_future());
  async_await([=]() { C(); prom_C->put(); }, prom_A->get_future());
  async_await([=]() { D(); prom_D->put(); }, prom_B->get_future(), prom_C->get_future());
  async_await([=]() { E(); prom_E->put(); }, prom_C->get_future());
  async_await([=]() { F(); prom_F->put(); }, prom_D->get_future(), prom_E->get_future());
});
```

hclib::promise_t and hclib::future_t workflow



HClib Futures: Tasks with Return Values

future_t<T> *f = async_future { S }

- Creates a new child task that executes S, which must terminate with a return statement and return value
- Async expression returns a pointer to a container of type future_t

T result = f.get();

- get() evaluates f and blocks if f's value is unavailable
- Unlike finish which waits for all tasks in the finish scope, a get operation only waits for the specified async_future

hclib::promise v/s hclib::future

- *"A promise is an object that can store a value of type T to be retrieved by a future object (possibly in another thread), offering a synchronization point"*
  - Writable end of an object
- *"A future is an object that can retrieve a value from some provider object or function, properly synchronizing this access if in different threads"*
  - Readable end of an object

Today's Lecture

- Parallel programming using Cilk
  - spawn & sync
  - inlet & abort
    - These interesting features are only available in MIT Cilk-5.4.6, and not in Intel Cilk Plus. Hence, we would use MIT Cilk-5.4.6 for this lecture
  - Mutual exclusion

Lecture-14 completed Part-1: Parallel programming in shared memory using Habanero-C library (HClib)

Acknowledgements: Habanero Team Members, Rice University

Cilk

From Wikipedia, the free encyclopedia

Not to be confused with SYCL.

Cilk, **Cilk++** and **Cilk Plus** are general-purpose programming languages designed for multithreaded parallel computing. They are based on the C and C++ programming languages, which they extend with constructs to express parallel loops and the fork–join idiom.

Originally developed in the 1990s at the Massachusetts Institute of Technology (MIT) in the group of Charles E. Leiserson, Cilk was later commercialized as Cilk++ by a spinoff company, Cilk Arts. That company was subsequently acquired by Intel, which increased compatibility with existing C and C++ code, calling the result Cilk Plus.

- We will use MIT Cilk-5.4.6 for this lecture, as it supports inlet & abort
  - Download: http://supertech.lcs.mit.edu/cilk/cilk-5.4.6.tar.gz
  - Installation
    - cd cilk-5.4.6
    - ./configure --prefix=/absolute path/install-directory
    - make install (tested with gcc-4.9)
  - Run tests
    - export PATH=/absolute path/install-directory/bin:$PATH
    - cd example
    - cilkc -D_XOPEN_SOURCE=600 -D_POSIX_C_SOURCE=200809L fib.cilk -o fib
    - ./fib --nproc <number of workers>

Introducing Cilk

Identifies a function as a Cilk procedure, capable of being spawned in parallel

Cilk is a faithful extension of C, i.e., it supports serial elision

The named Child procedure can execute in parallel with the parent caller

```
cilk uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x + y);
  }
}
cilk int main(int argc, char** argv) {
  int result = spawn fib(40);
  sync;
}
```

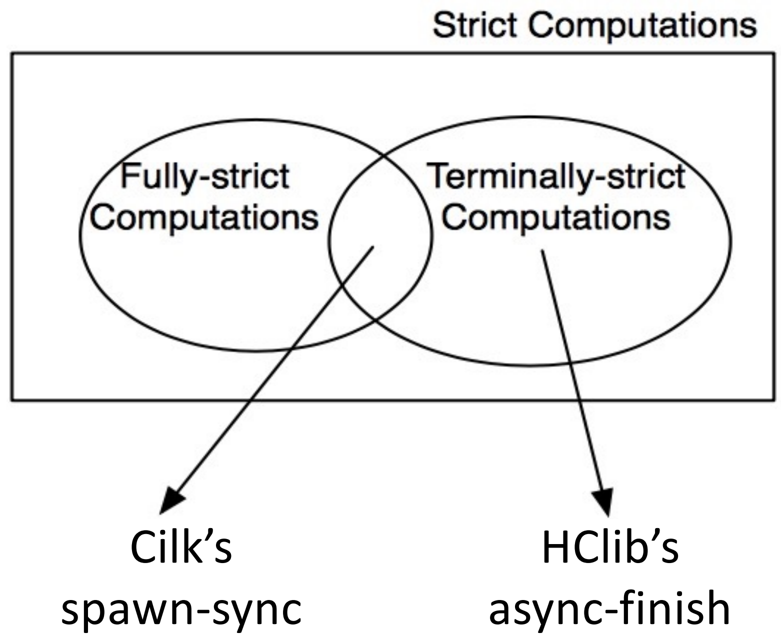
Control cannot pass this point until all spawned children have returned

spawn keyword can only be applied to a Cilk function, and cannot be used in a C function

Cilk function cannot be called as normal C function, and must be called with spawn & waited for by a sync

Cilk's *spawn-sync* v/s HClib's *async-finish*

- What is a "strict" computation?
  - A strict computation is one in which all join edges from a task go to one of its ancestor tasks in the computation graph



Fully-strict v/s Terminally-strict

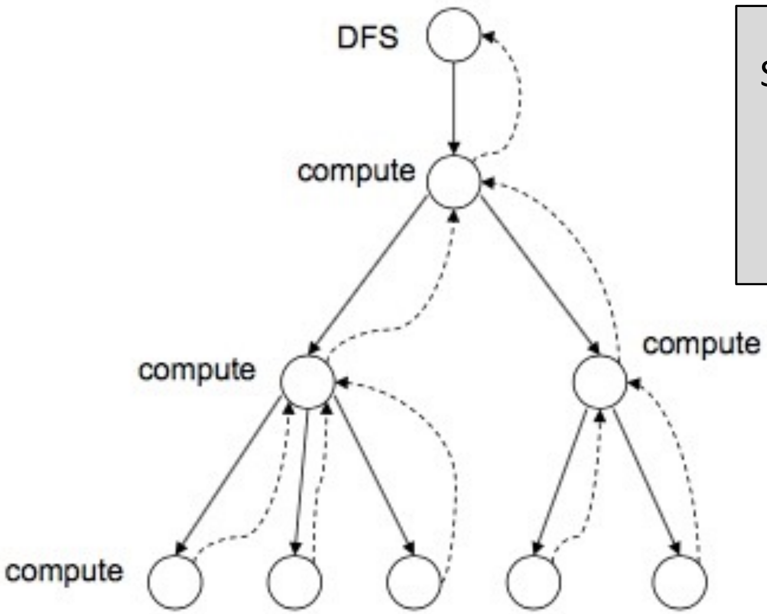
```
cilk void compute(Node* node) {
  int i;
  process(node); //sequential
  for(i=0; i<node->numChild; i++) {
    spawn compute(node->child[i]);
  }
  sync;
}
cilk void DFS(Node* root) {
  spawn compute(root);
  sync;
}
```

Cilk

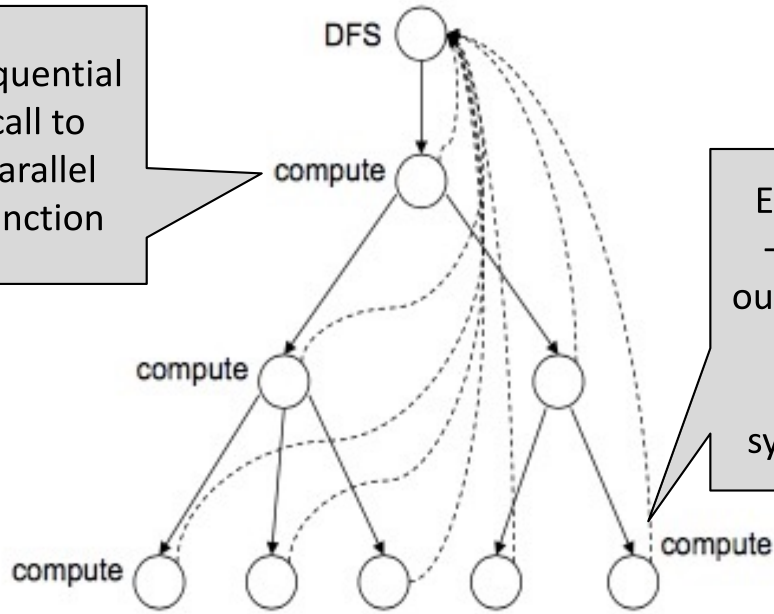
```
void compute(Node* node) {
  int i;
  process(node); //sequential
  for(i=0; i<node->numChild; i++) {
    async compute(node->child[i]);
  }

}
void DFS(Node* root) {
  finish compute(root);
}
```

HClib



Sequential call to parallel function



Escaping async –> a child can outlive its parent, i.e. avoiding unnecessary synchronization

Cilk Scheduler

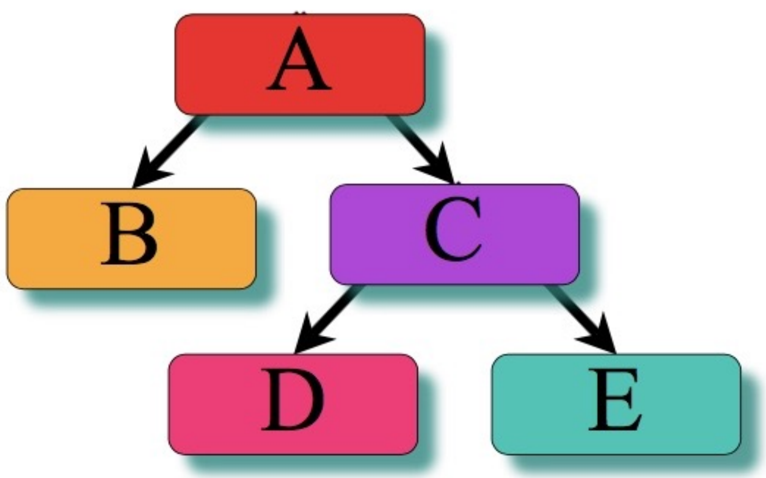
- Uses work-first work-stealing runtime

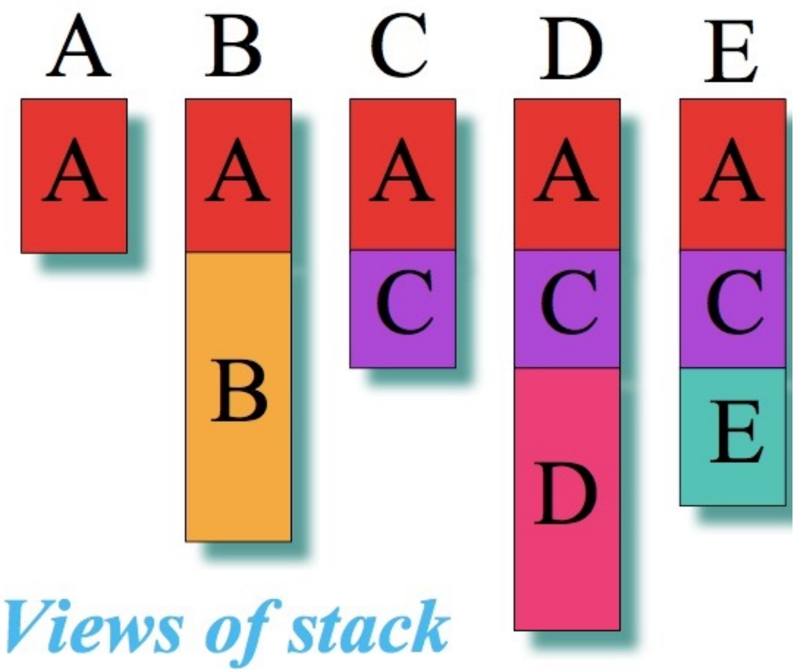
```
cilk void foo() {
  for (uint64_t i=0; i<SIZE; i++) {
    spawn S(i); // can execute in parallel for all i
  }
  sync;
}
```

Loop runs perfectly fine, without any risk of blowing out memory, Why?

The Cactus Stack Abstraction

- Cilk runtime maintains cactus stack abstraction so that each worker has the complete stack similar to sequential execution
  - Supports C's rules for pointers
    - A pointer to stack variable can be passed from parent to child, but not from child to parent





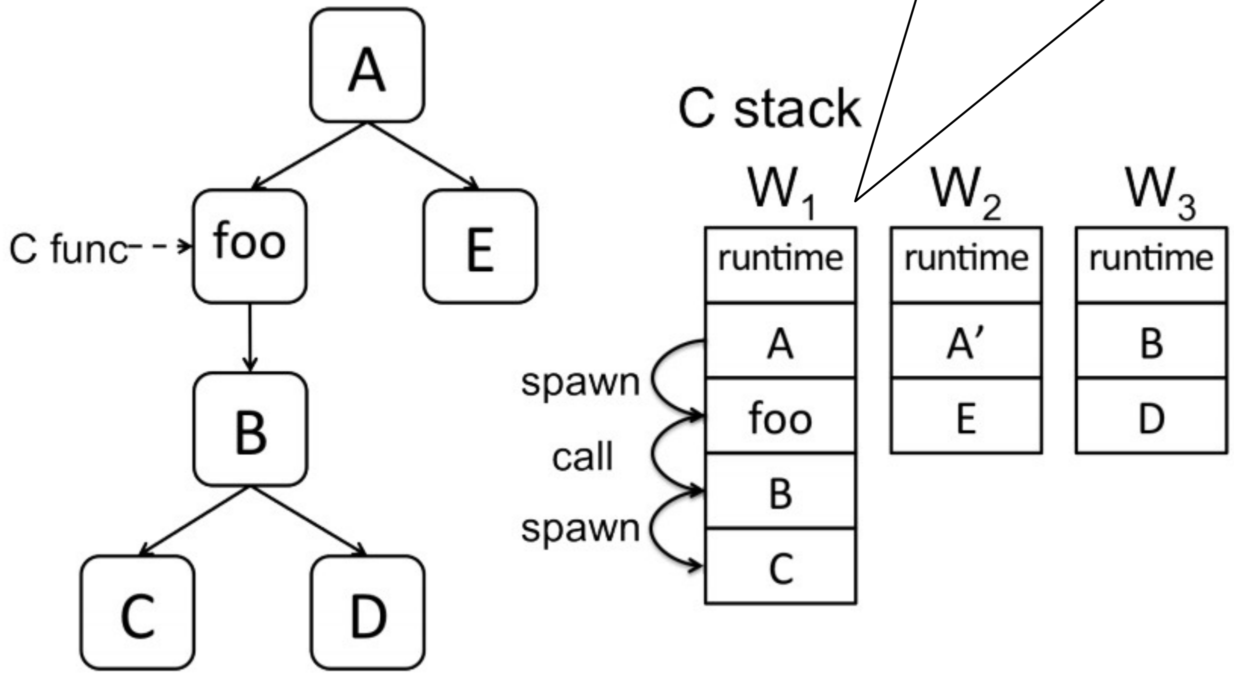
Views of stack

Lack of Serial-Parallel Reciprocity

- Recall
  - Cilk functions must be spawned, not called
  - C functions must be called, not spawned
  - Compilation error if above two rules are not followed

W1 after returning from C realizes that B has been stolen. As it is a work-first work-stealing, W1 should discard all the frames on its stack before attempting a steal. However, it cannot discard frame foo as it isn't a cilk function

```
cilk void A() {
    spawn foo();
    spawn E();
    sync;
}
void foo() {
    B();
}
cilk void B() {
    spawn C();
    spawn D();
    sync;
}
```



Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

C

```
void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
  }
}
```

- Parallelization strategy:
  1. Convert loops to recursion

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

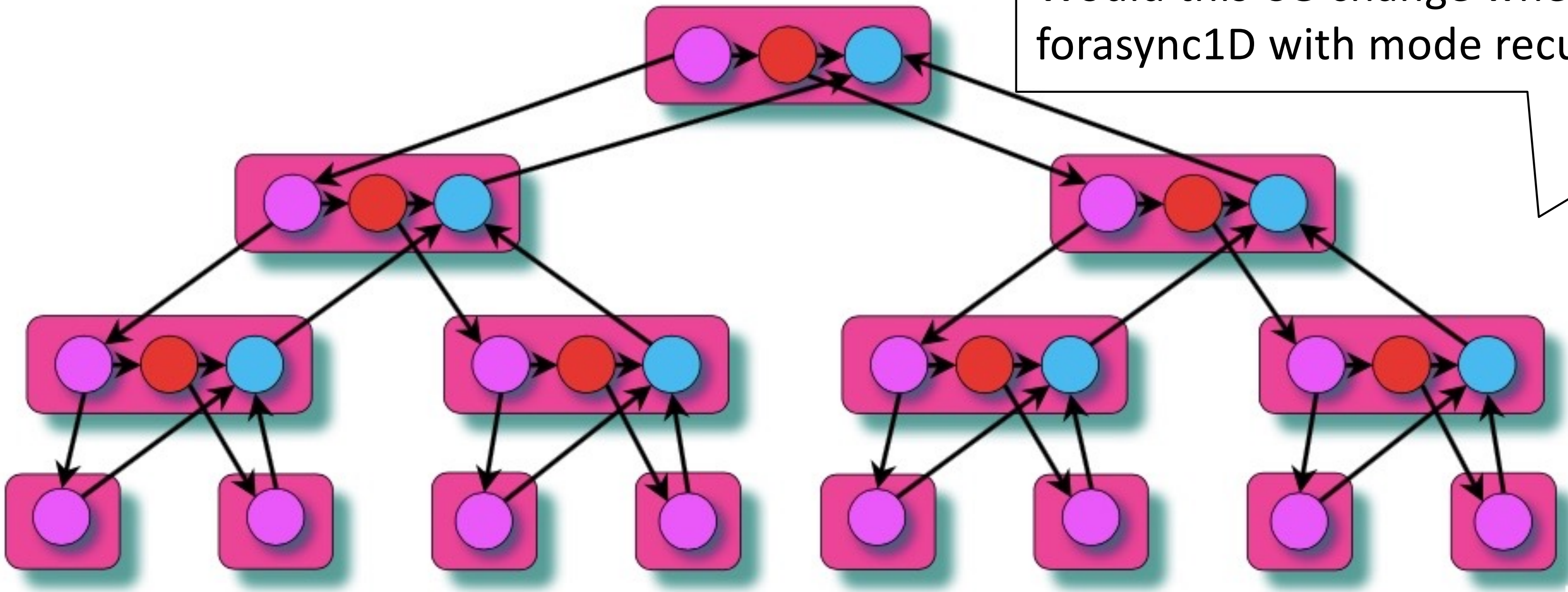
```
cilk void vadd (real *A, real *B, int n){
   if (n<=BASE) {
      int i; for (i=0; i<n; i++) A[i]+=B[i];
   } else {
  spawn vadd (A, B, n/2);
  spawn vadd (A+n/2, B+n/2, n-n/2);  sync;
   }
}
```

- Parallelization strategy:
  1. Convert loops to recursion
  2. Insert Cilk keywords

Parallelizing Vector Addition

```
cilk void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
    sync;
  }
}
```

Would this CG change when using HClib's forasync1D with mode recursive?



Today's Lecture

- Parallel programming using Cilk
  - spawn & sync
  - inlet
  - abort
  - Mutual exclusion

Operating on Returned Values

- Programmers may wish to operate on a return value without waiting on a sync

Example:

```
for (i=0; i<1000000; i++) {
  update(spawn foo(i), i );
}
sync;
/* All spawns and updates are now
completed */
```

- Cilk achieves this functionality by using an internal function, called an inlet, which can be viewed as an “event handler” task executed by the parent when the child returns

Semantics of **inlet**

```
int max, ix = -1;
inlet void update ( int val, int index ) {
   if ( val > max ) {
      ix = index; max = val;
   }
}
:
for (i=0; i<1000000; i++) {
   update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

- The inlet keyword defines a void internal function to be an inlet
- inlet function cannot contain a spawn
- Only the first argument of the inlet may be spawned at the call site
- Only one inlet per cilk function

Semantics of **inlet**

```
int max, ix = -1;
inlet void update ( int val, int index ) {
   if ( val > max ) {
      ix = index; max = val;
   }
}
:
:
for (i=0; i<1000000; i++) {
   update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

1. The non-spawn args to update() are evaluated
2. The Cilk procedure foo(i) is spawned
3. Control passes to the next statement
4. When foo(i) returns, update() is invoked

Semantics of **inlet** (Fib with inlet)

```
cilk uint64_t fib(uint64_t n) {
  uint64_t x = 0;
  inlet void summer(uint64_t result) {
    x += result;
    return;
  }
  if(n<2) {
    return n;
  } else {
    summer(spawn fib(n-1));
    summer(spawn fib(n-2));
    sync;
    return x;
  }
}
```

No data race on **local** variable x!

Notice there is no data-race on addition inside inlet. Cilk **guarantees** that tasks from a function instance, including inlets, operate atomically with respect to one another

Question

```
cilk uint64_t fib(uint64_t n) {
  if(n<2) {
    return n;
  } else {
    uint64_t x = 0;
    x += spawn fib(n-1);
    x += spawn fib(n-2);
    sync;
    return x;
  }
}
```

Is there a data-race now?

Implicit inlets

- For assignment operators, the Cilk compiler automatically generates an implicit inlet to perform the update
  - Hence, no data race above!

Today's Lecture

- Parallel programming using Cilk
  - spawn & sync
  - inlet
  - abort
  - Mutual exclusion

Computing a Product

```
int product(int *A, int n) {
  int i, p=1;
  for (i=0; i<n; i++) {
    p *= A[i];
  }
  return p;
}
```

Optimization: Quit early if the partial product ever becomes 0

Computing a Product

```
int product(int *A, int n) {
  int i, p=1;
  for (i=0; i<n; i++) {
    p *= A[i];
    if (p == 0) break;
  }
  return p;
}
```

Optimization: Quit early if the partial product ever becomes 0

Computing a Product in Parallel

```
cilk int prod(int *A, int n) {
  int p = 1;
  if (n == 1) {
    return A[0];
  } else {
    /* Note use of implicit inlets */
    p *= spawn product(A, n/2);
    p *= spawn product(A+n/2, n-n/2);
    sync;
    return p;
  }
}
```

How do we quit early now once we discover a zero?

Computing a Product in Parallel

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

1. Recode the implicit inlet to make it explicit

Computing a Product in Parallel using **inlet** & **abort**

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

1. Recode the implicit inlet to make it explicit

Computing a Product in Parallel using **inlet** & **abort**

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;



    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

1. Recode the implicit inlet to make it explicit
2. Check for 0 within the inlet

Computing a Product in Parallel using **inlet** & **abort**

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.       */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

1. Recode the implicit inlet to make it explicit
2. Check for 0 within the inlet

Computing a Product in Parallel using **inlet** & **abort**

```
cilk int product(int *A, int n) {
  int p = 1;
  inlet void mult(int x) {
    p *= x;
    if (p == 0) {
      abort; /* Aborts existing children, */
    }        /* but not future ones.      */
    return;
  }

  if (n == 1) {
    return A[0];
  } else {
    mult( spawn product(A, n/2) );
    if (p == 0) { /* Add check for future */
      return 0;   /* children             */
    }
    mult( spawn product(A+n/2, n-n/2) );
    sync;
    return p;
  }
}
```

Today's Lecture

- Parallel programming using Cilk
  - spawn & sync
  - inlet
  - abort
  - Mutual exclusion

Mutual Exclusion

- Cilk's solution to mutual exclusion is very primitive
- It provides a library of spin locks declared with **Cilk_lockvar**
  - spawn/sync should not be called inside the critical section

Next Lecture

- Introduction to OpenMP programming model
- **Lab-3 & Lab-4 next week during lecture days**
- **No lectures next week**

Reading Material

- Cilk-5.4.6 reference manual
  - http://supertech.lcs.mit.edu/cilk/manual-5.4.6.pdf

Acknowledgements

- Prof. Vivek Sarkar
  - COMP422, Rice University
- Prof. I-Ting Angelina Lee
  - CSE539, Washington University in St. Louis