CS529 Lecture 04: Cilk

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

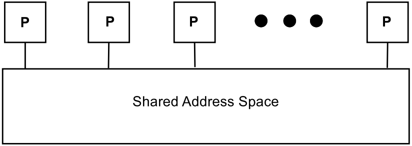
March 7, 2011

Outline

Sources of material

- Cilk 5.4.6 reference manual and the Cilk project documentation, http://supertech.csail.mit.edu/cilk/
- Charles Leiserson, Bradley Kuzmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems. http://theory.lcs.mit.edu/classes/6.895/fall03/scribe/master.ps

Shared-memory architectures



- Hardware model
  - Shared global memory
  - processors virtually equidistant from memory
- Software model
  - threads
  - shared variables
  - communication
    - read shared data (loads)
    - write shared data (stores)

Introducing 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);
  }
}
```

- Cilk constructs
  - cilk: Cilk function. Without it, functions are standard C
  - spawn: call can execute asynchronously in a concurrent thread
  - sync: current thread waits for all locally-spawned functions

Introducing 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);
  }
}
```

- Cilk constructs specify logical parallelism in the program
  - what computations can be performed in parallel
  - not mapping of tasks to processes

The Cilk Language

- Cilk is a faithful extension of C
  - if Cilk keywords are elided the program maintains C program semantics
- Idiosyncrasies
  - spawn keyword can only be applied to a cilk function
  - spawn keyword cannot be used in a C function
  - cilk function cannot be called with normal C call conventions
    - must be called with a spawn & waited for by a sync

Cilk Terminology

- Parallel control = spawn, sync, return from spawned function
- Thread = maximal sequence of instructions not containing parallel control (task in earlier terminology)

```
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

Outline

Sum of first N integers

```
#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;
}
```

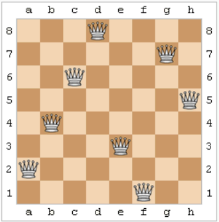
Initialize and sum a vector

```
#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;
}
```

Example: N Queens

- Problem
  - Place N queens on a $N \times N$ chess board
  - no 2 queens in same row, column, or diagonal

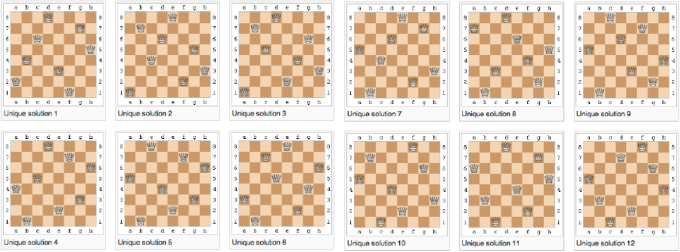


One possible solution

image credit: http://en.wikipedia.org/wiki/Eight_queens_puzzle

N Queens has many possible solutions

- Example: 8 queens
  - 92 distinct solutions
  - 12 unique solutions, if solutions derived from rotation and reflection count as equivalent



N Queens solutions sketch

Sequential recursive enumeration of all solutions

```
int nqueens(n, j, placement) {
  // precondition: placed j queens so far
  if (j == n) { print placement; return; }
  for (k = 0; k < n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      add queen j+1 to placement
      nqueens(n, j+1, placement)
      remove queen j+1 from placement
}
```

- Potential for parallelism?
- Other issues to consider?

N Queens solutions sketch

Sequential recursive enumeration of all solutions

```
int nqueens(n, j, placement) {
  // precondition: placed j queens so far
  if (j == n) { print placement; return; }
  for (k = 0; k < n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      add queen j+1 to placement
      nqueens(n, j+1, placement)
      remove queen j+1 from placement
}
```

- Parallelism exists across correct placements
- Adding queens to placements needs synchronization

N Queens solutions sketch

```
cilk void nqueens(n, j, placement) {
  // precondition: placed j queens so far
  if (j == n) { /* found a placement */ process placement; return; }
  for (k = 1; k <= n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      copy placement into newplacement and add extra queen
      spawn nqueens(n,j+1,newplacement)
      discard newplacement
  sync
}
```

- Issues regarding placements
  - how can we report placements without conflicts?
  - what if we only need one valid placement?
    - no need to compute all legal placements
    - need a way to terminate children that explore alternate placements

Approaches to reporting valid placements

- Count valid placements
  - Need a protected counter
- Print valid placements
  - Need thread-safe library for output
- Collect then print
  - Need protected data structure for collection (e.g. array)

Race Conditions (Data Races)

- Two or more concurrent accesses to the same address
- At least one is a write

```
cilk int f() {
  int x = 0;
  spawn g(&x);
  spawn g(&x);
  sync;
  return x;
}

cilk void g (int *p)
{
  *p += 1;
}
```

serial semantics:
f returns 2

parallel semantics:
may return 1 or 2

parallel execution of two instances of g: g,g,
many interleavings possible

one interleaving:

read x
read x
add 1
add 1
write x; x =1
write x; x=1!

N Queens solution with races

```
cilk void nqueens(n, j, placement) {
  // precondition: placed j queens so far
  if (j == n) { /* found a placement */ process placement; return; }
  for (k = 1; k <= n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      place j+1 queen in kth position in row j+1 in placement
      spawn nqueens(n, j+1,placement)
      remove queen in kth position in row j+1 in placement
  sync
}
```

Problems with races

- Different interleavings produce different results
- Hard to debug programs with races
  - Non-deterministic execution, different outputs
- Bugs often appear during production runs
- Races can be benign or malicious!
  - Busy-wait on a flag versus updating a shared counter

Programming with race conditions

- First approach: avoid races completely
  - No read-write sharing between tasks
  - only share between parent and child tasks in Cilk
- Second approach: use caution and protection
  - guard against data corruption
    - word read-write operations are atomic in all modern microprocessors
    - definition of word is processor-specific, usually 32-bit or 64-bit
    - locks can enforce atomic access to shared addresses

inlets

- Normal spawn: `x = spawn f(...);`
  - Result of `f` is copied to caller's frame
- Problem:
  - May need to handle receipt of result immediately after spawned child returns
    - Do not wait until sync point to collect result
  - Nqueens: update legal placement upon return of child
- Solution: `inlet`
  - block of code within a function used to process result of function upon completion
  - executes atomically with respect to enclosing function
- Syntax: inlets must appear in declarations

inlets example

```
cilk int f(...) {
  inlet void my_inlet (ResultType* result, iarg2, ..., iargn) {
    // atomically incorporate result into f's variables
    return;
  }
  my_inlet(spawn g(...), iarg2, ..., iargn);
}
```

inlet example

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

```
cilk int fib(int n) {
  int result = 0;
  inlet void add(int r) {
    result += r;
    return;
  }
  if (n < 2) return n;
  else {
    int n1, n2;
    add(spawn fib(n-1));
    add(spawn fib(n-2));
    sync;
    return result;
  }
}
```

- Cilk guarantees that inlet instances are atomic with respect to each other
- inlet has access to variables of enclosing context

abort

- Syntax: `abort;`
- Where: within a `cilk` procedure p
- Purpose: terminate execution of all of p's spawned children
- Does this help with an nqueens example for a single solution?

```
cilk void nqueens(n,j, placement) {
  // precondition: placed j queens so far
  if (j == n) return placement
  for (k = 0; k < n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      copy placement into newplacement and add extra queen
      spawn nqueens(n,j+1,newplacement)
      discard newplacement
  sync;
  if some child found a legal result return one, else return null
}
```

- Need a way to invoke abort when a child yields a solution

Nqueens revisited

Solution that finishes after first legal result is found

```
cilk void nqueens(n,j,placement) {
  int *result = null
  // precondition: placed j queens so far
  inlet void doresult(childplacement) {
    if (childplacement == null) return; else { result = copy(childplacement); abort; }
  }
  if (j == n) return placement
  for (k = 0; k < n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
      copy placement into newplacement and add extra queen
      doresult(spawn nqueens(n,j+1,...))
      discard newplacement
  sync
  return result
}
```

Implicit inlets

- General `spawn` syntax
  - statement: [lhs op] `spawn` proc(arg1,...,argn);
  - lhs op may be omitted
    - `spawn update(&data)`
  - if lhs is present
    - it must be a variable matching the return type of the function
    - op may be:
      $=, * =, / =, \% =, + =, - =, <<=, >>=, \& =,^{=}, | =$
  - Implicit inlets execute atomically with respect to caller

Using an implicit inlet

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

```
cilk int fib(int n) {
  int result = 0;
  if (n < 2) return n;
  else {
    int n1, n2;
    result += spawn fib(n-1));
    result += spawn fib(n-2));
    sync;
    return result;
  }
}
```

SYNCHED

- Determine if a procedure has any currently outstanding children without executing `sync`
  - if children have not completed
    - `SYNCHED=0`
  - if children have completed
    - `SYNCHED=1`
- Why `SYNCHED`? Save storage and enhance locality

```
state *state1, state2;
state1 = (state *) Cilk_alloca(state_size);
spawn foo(state1); /* fill in state1 with data */
if (SYNCHED) state2 = state1;
else state2 = (state *) Cilk_alloca(state_size);
spawn bar(state2);
sync;
```

Locks

- Why locks? Guarantee mutual exclusion while accessing shared state
  - Locks are the only way to guarantee atomicity when concurrent procedure instances operate on shared data
- Library primitives for locking

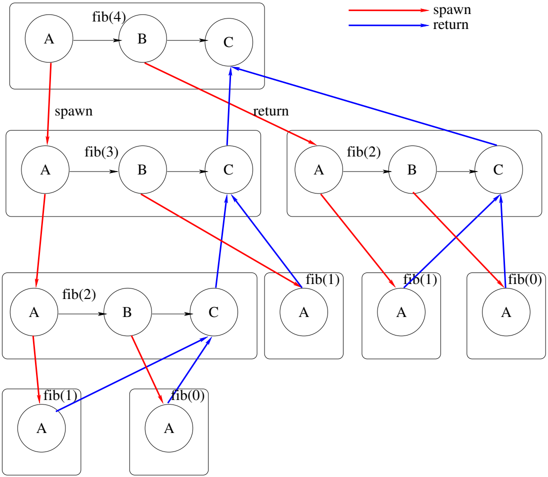
```
Cilk_lock_init(Cilk_lockvar k)
Cilk_lock(Cilk_lockvar k)
Cilk_unlock(Cilk_lockvar k)
```

- Usage examples
  - can use a lock to protect I/O from parallel writes in nqueens
  - parallel solution could enumerate all solutions in the order that they are found

Cilk concurrency implications

- Cilk atomicity guarantees
  - all threads of a single procedure operate atomically
  - threads of a procedure include
    - all code in the procedure body proper, including inlet code
- Guarantee implications
  - can coordinate caller and callees using inlets without locks
- Only limited guarantees between descendants or ancestors
  - DAG precedence order maintained and nothing more
  - atomicity can not be assumed across different procedures

Cilk program execution as a DAG



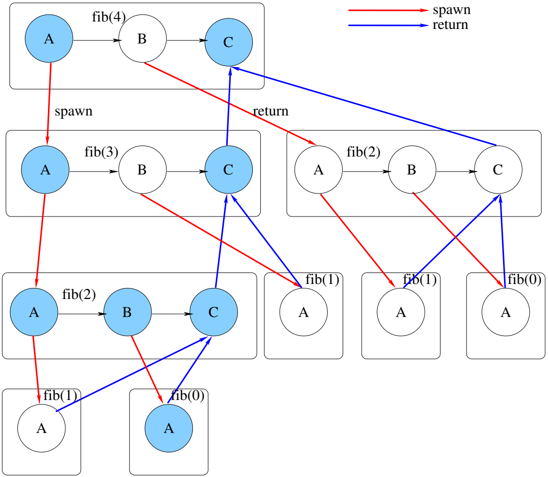
Thread scheduling in Cilk

- work-sharing: thread scheduled to run in parallel at every spawn
  - benefit: maximizes parallelism
  - drawback: cost of setting up new threads to run remotely (on another processor) is high
- work-stealing: processor looks for work when it becomes idle
  - lazy parallelism: put off extra work for parallel execution until necessary
  - benefits
    - executes with precisely as much parallelism as needed
    - minimizes the number of (Cilk) threads that must be set up
    - runs with the same efficiency as serial program on uniprocessor
  - drawback: work stealing is an expensive operation requiring synchronization and transfer of state

Cilk performance metrics

- T_1: sequential work; minimum running time on 1 processor
- T_p: minimum running time on p processors
- T_∞: minimum running time on infinite number of processors
  - longest path in DAG
    - length reflects the cost of computation at nodes along the path
  - known as critical path length

Work and critical path example



Lower bounds on execution time

- $T_p \geq T_1/P$
  - P processors can do at most P work in one step
  - suppose $T_p < T_1/P_1$ then $PT_P < T_1$ (a contradiction)
- $T_p \geq T_\infty$
  - suppose not: $T_p < T_\infty$
  - could use P of unlimited processors to reduce T_∞

Greedy scheduling

- Types of schedule steps
  - complete step
    - at least P threads ready to run
    - select any P and run them
  - incomplete step
    - strictly $<$ P threads ready to run
    - greedy scheduler runs them all
- Theorem: On P processors, a greedy scheduler executes any computation G with work T_1 and critical path of length T_∞ in time $T_p \leq T_1/P + T_\infty$
- Proof sketch
  - only two types of scheduler steps: complete, incomplete
  - cannot be more than T_1/P complete steps, else work $> T_1$
  - every incomplete step reduces remaining critical path length by 1
    - No more than T_∞ incomplete steps

Speedup

- T_s / T_p = speedup
  - with P processors, maximum speedup is P (for simplified model)
  - Possibilities
    - linear speedup: $T_s / T_p = \Theta(P)$
    - sublinear speedup: $T_s / T_p = o(P)$
    - superlinear speedup: $T_s / T_p = \Omega(P)$
- $\bar{P} = T_1 / T_\infty$, maximum speedup on ∞ processors

Parallel slackness

- critical path overhead = smallest constant c_∞ such that:

$$T_p \leq \frac{T_1}{P} + c_\infty T_\infty \quad (1)$$

$$T_p \leq \left(\frac{T_1}{T_\infty P} + c_\infty\right) T_\infty = \left(\frac{\bar{P}}{P} + c_\infty\right) T_\infty \quad (2)$$

- Parallel slackness assumption

$$\frac{\bar{P}}{P} >> c_\infty \text{ thus } \frac{T_1}{P} >> c_\infty T_\infty \quad (3)$$

$$T_p \approx \frac{T_1}{P} \quad (4)$$

- critical path overhead has little effect on performance when sufficient parallel slackness exists

Work overheads

$$c_1 = \frac{T_1}{T_s} \text{ work overhead} \quad (5)$$

$$T_p \le c_1 \frac{T_s}{P} + c_\infty T_\infty \quad (6)$$

$$T_p \approx c_1 \frac{T_s}{P} \quad (7)$$

- Minimizing work overhead c_1 at the expense of a larger critical path overhead c_∞ because work overhead has a more direct impact on performance

Compilation

Cilk compiler generates two copies of each procedure

- Fast clone: optimized execution on a single processor
  - spawning threads is fast
- Slow clone: triggered by work stealing, support for parallel execution
  - handles execution of stolen procedure frames
  - supports Cilk's work stealing scheduler
  - steals will be few if there is enough parallel slackness
    - speed of slow copy is considered not critical for performance
- Work-first principle: minimize cost in fast clones

- Nanoscheduler: compiled into cilk program
  - executes cilk function and spawns in exactly the same order as C
  - on one core: when no microscheduling needed, same order as C
  - efficient coordination with microscheduler
- Microscheduler:
  - schedules procedures across a fixed set of processors
  - randomized work-stealing scheduler
    - when a processor runs out of work it becomes a thief
    - steals from a victim chosen uniformly at random

Fast clone and nanoscheduler

- Fast clone is never stolen
  - converted to slow when steal occurs
  - enables optimizations
- No sync is needed in fast clone
  - No children spawned
- Frame saves state
  - PC (entry number)
  - live, dirty variables
- push, pop must be fast

```
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 variables
    *T=f;
//store frame pointer
    push();                  //push frame
    x = fib(n-1);            //do C call
    if (pop(x) == FAILURE)
      return 0;              //pop frame
    free(f,sizeof(*f));
    return(x+y);
  }
}
```

Nanoscheduler overheads

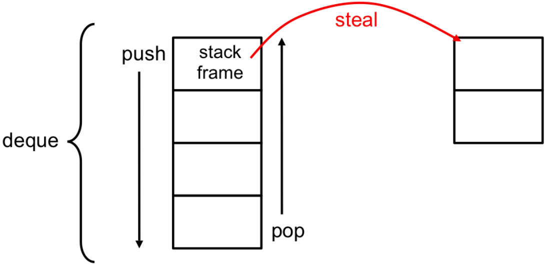
Basis for comparison: serial C

- Allocation and initialization of frame, push onto 'stack'
  - a few assembly instructions
- Procedure's state needs to be saved before each spawn
  - entry number, live variables
- Check whether frame is stolen after each spawn
  - two reads, compare, branch
- On return, free frame - a few instructions
- One extra variable to hold frame pointer

Runtime support for scheduling

- Each processor has a ready deque (double ended queue)
  - Tail: worker adds or removes procedures (like C call stack)
  - Head: thief steals from head of a victim's deque

Scheduling using deques

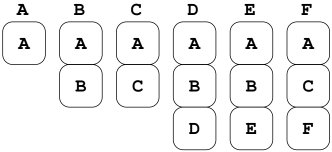


- Deque grows forward
- Stack frame contains local variables for a procedure invocation
- Procedure call → new frame is pushed onto the bottom of the deque
- Procedure return → bottom frame is popped from the deque
- Deque maintains order (synchronizes) between caller and callee

Cilk cactus stacks

A cactus stack enables sharing of a C function's local variables

```
void A() { B(); C;}
void B() { D(); E;}
void C() { F(); }
void D() {}
void E() {}
void F() {}
```



- Pointers can be passed down call chain
- Can pass pointers up only if they point to the heap
- Functions can not return pointers to local variables

Microscheduler

schedules procedures across a fixed set of processors

- When a processor runs out of work, it becomes a thief
  - steals from victim processor chosen uniformly at random
- When it finds victim with frames in its deque
  - takes the topmost frame (least recently pushed)
  - places frame into its own deque
  - gives the corresponding procedure to its own nanoscheduler
- Microscheduler executes slow clone
  - Receives only pointer to frame as argument
    - Real args and local state are inside frame
  - Restores program counter to proper place using switch statement
  - At a sync must wait for children
  - before procedure returns, places return value into frame

Coordinating thief and worker

- Runtime system uses a lock to manipulate each worker's deque
- Can use a lock-free deque data structure instead (Hakan Sundell, Ph.D. Thesis, Chalmers University)
- Use a software mutex protocol
  - Dijkstra's algorithm

Simplified scheduling protocol (without exceptions)

- 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

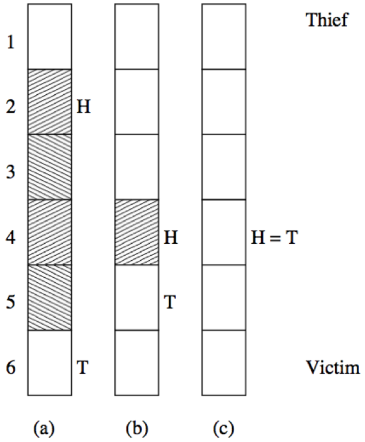
```
//Worker/Victim
push() {
  T++
}

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;
}
```

Deque pop

- (a) no conflict
- (b) At least one thief or victim finds (H $>$ T) and backs up; other succeeds
- (c) Deque is empty, both threads return



Outline

Cilk++: Differences from Cilk

- `cilk_main` instead of main
- `cilk_spawn` instead of `spawn`
- No need to mark procedures with the `cilk` keyword primitive
  - can call procedures directly or use `cilk_spawn`
- `clik_sync` instead of `sync`
- `cilk::mutex` instead of Cilk lock variables
  - methods: `void lock(), void unlock(), bool_try_lock()`
- No support for abort
- `cilk_for`
- `cilk::hyperobject` and reducers rather than inlets
- Race detection with cilkscreen

Cilk++ parallel for: cilk_for

```
cilk_for (T v = begin; v < end; v++) {
  statement_1;
  statement_2;
  ...
}
```

- Loop index v
  - type T can be an integer, pointer, or a C++ random access iterator
- Some restrictions
  - must compare v with end value using $<, <=, !=, >=, >$
  - loop increment must use $++, --, +=, v = v + incr, v = v - incr$
    - if v not a signed integer, loop must count up
  - runtime must be able to compute total number of iterations

Cilk++ parallel for: more restrictions

- No early exit
  - no break or return statement inside loop
  - no goto in loop unless target is in loop body
- Illegal examples
  - `cilk_for (unsigned int i,j=42; j<1; i++,j++) {...}`
    - only one loop variable allowed
  - `cilk_for (unsigned int i=1; i<16; ++i) i=f();`
    - can't modify variable inside loop
  - `cilk_for (unsigned int i=1;i<x;++i) x=f();`
    - can't modify loop bounds inside loop
  - `int i; cilk_for(i=0;i<100;i++) {...}`
    - loop variable must be declared in loop header

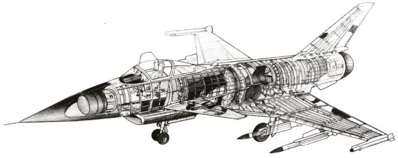
Cilk++ `cilk_for` implementation

- Iterations divided into chunks to be executed serially
  - chunk is sequential collection of one or more iterations
- Invisible `cilk_spawn` for each chunk
- Maximum size of chunk is called “grain size”
  - grain size too small: spawn overhead reduces performance
  - grain size too large: reduces parallelism and hurts load balancing
- Can override default grain size
  - `#pragma cilk_grainsize = expr`
    - expression is any C++ expression that yields an integral type (e.g. int, long) e.g. n/(4*workers)
  - pragma should immediately precede `cilk_for` to which it applies

Clik++ hyperobjects

- Nonlocal variables are a common programming construct
  - nonlocal = declared in a scope outside that where it is used
  - global variables = nonlocal variables in outermost scope
- Rewriting parallel applications to avoid them is painful
- Cilk++ hyperobjects support deterministic sharing of non-local variables
  - e.g. output stream, global sum, list, ...
  - can be used without significant code restructuring
- Retain serial semantics
  - result using reducers is same as serial version
  - independent of # processors or scheduling
- Implemented efficiently
  - Cilk Arts claim: runtime performance using reducers can be better than passing variables as arguments

Motivating example for hyperobjects



Computing cutaway view

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        output_list.push_back(x);
    break;
  case Node::INTERNAL:
    for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)
        walk(child);
    break;
  }
}
```

Cilk++ parallelization of cutaway view

Computing cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        output_list.push_back(x);
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)
        walk(child);
    break;
  }
```

Global list access creates races

First solution: locking

Computing cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
cilk::cilk_mutex m;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
    { m.lock(); output_list.push_back(x); m.unlock(); }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)
        walk(child);
    break;
  }
}
```

- Add a mutex to coordinate accesses to `output_list`
- Drawback: lock contention can hurt parallelism

Second solution: refactoring the code

Computing cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x, std::list<Node *> o_list) {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        o_list.push_back(x);
    break;
  case Node::INTERNAL:
    std::vector<std::list<Node *> >
          children_list(x.num_children);
    cilk_for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)
        walk(child, children_list[child]);
    for (int i=0; i < x.num_children; ++i)
        o_list.splice( o_list.end(), children_list[i]);
    break;
  }
```

- Have each child accumulate results in a separate list
- Splice them all together
- Drawback: development time, debugging

Third solution: using Cilk++ hyperobjects

Computing cutaway view in parallel

```
Node *target;
cilk::hyperobject< cilk::reducer_list_append<Node *>
> output_list;
...
void walk(Node *x) {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        output_list().push_back(x);
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)q
          walk(child);
    break;
  }
}
```

- Resolve data races without locking or refactoring
- Parallel strands may see different views of hyperobject, but these views are combined into a single consistent view

Memory management

- Memory management issues
  - C/C++ memory management routines are thread safe, but
    - optimized for use in single-threaded environment
    - uses global lock to provide exclusive access to allocator state
  - false sharing: different workers have different data in the same cache line
  - fragmentation
- Miser memory management
  - separate pool per strand
  - avoids fragmentation by rounding up to powers of 2 for $<$ 256 bytes
  - allocations for $>$ 256 bytes use system allocator

False sharing

Computing cutaway view in parallel

```
int* a = new int[n];
cilk_for(int i = 0; i < n; i++) {
   // Populate A
   a[i] = func(i);
}
```

- Elements in `a` are 4 bytes wide
- Cache lines in x86 architectures are typically 64 bytes
- Example contains on races
  - result will be correct when loop terminates
- If two processors store in different element locations in the same cache line, each store on one processor will invalidate the cache line on the other processor

Race conditions

- Data race
  - two parallel strands access the same data
  - at least one access is a write
  - no locks held in common
- General determinacy race
  - two parallel strands access the same data
  - at least one access is a write
  - a common lock protects both accesses

Cilkscreen

```
// code with a data race
int sum = 0;
cilk_for (int i = 0; i < n; i++) {
  sum += a[i];
}
```

- Detects and reports data races when program terminates
  - finds all data races even those by third-party or system libraries
- Does not report determinacy races
  - e.g. two concurrent strands use a lock to access a queue
    - enqueue & dequeue operations could occur in different order
    - potentially leads to different results

Race Detection Strategies in Cilkscreen

- Lock covers
  - two conflicting accesses to a variable don't race if some lock L is held while each of the accesses is performed by a strand
- Happens-before
  - two conflicting accesses do not race if one must happen before the other
    - access A is by a strand X, which precedes the spawn of strand Y which performs access B
    - access is performed by strand X, which precedes a sync that is an ancestor of strand Y