Shared-memory Parallel Programming with Cilk Plus

Milind Chabbi

Nikhil Hegde

Department of Computer Science
IIT Dharwad

Outline for Today

- **Cilk Plus (Cont…)**
  - —**parallel loops**
  - —**reducers**

Cilk Parallel Loop: cilk_for

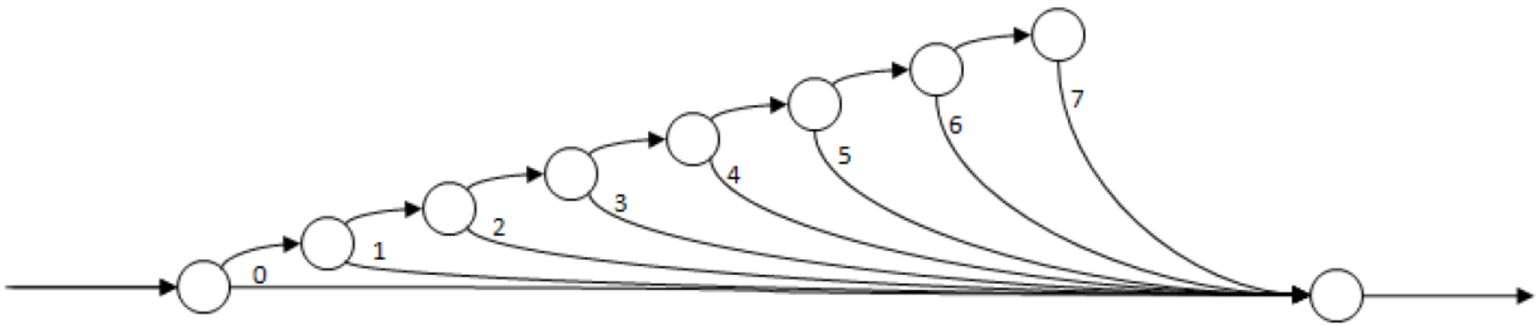
```
cilk_for (T v = begin; v < end; v++) {
      statement_1;
      statement_2;
      ...
}
```

- Loop index v
  - type T can be an integer, ptr, or a *C++ random access iterator*
- Main restrictions
  - runtime must be able to compute total # of iterations on entry to cilk_for
    - must compare v with end value using <, <=, !=, >=, or >
    - loop increment must use ++, --, +=, v = v + incr, or v = v - incr

      if v is not a signed integer, the loop must count up
- Implicit cilk_sync at the end of a cilk_for

Loop with a cilk_spawn vs. cilk_for

- **for (int i = 0; i < 8; i++) { cilk_spawn work(i); } cilk_sync;**

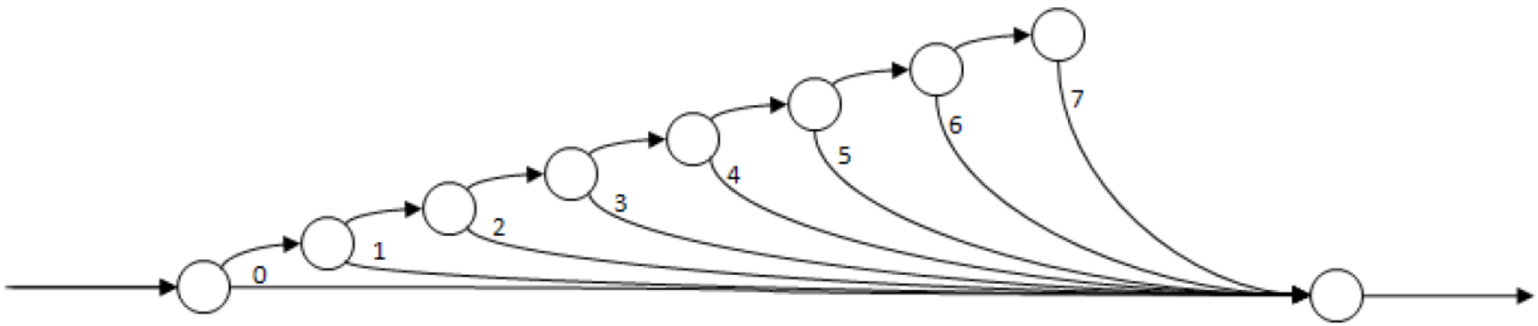


Note: computation on edges

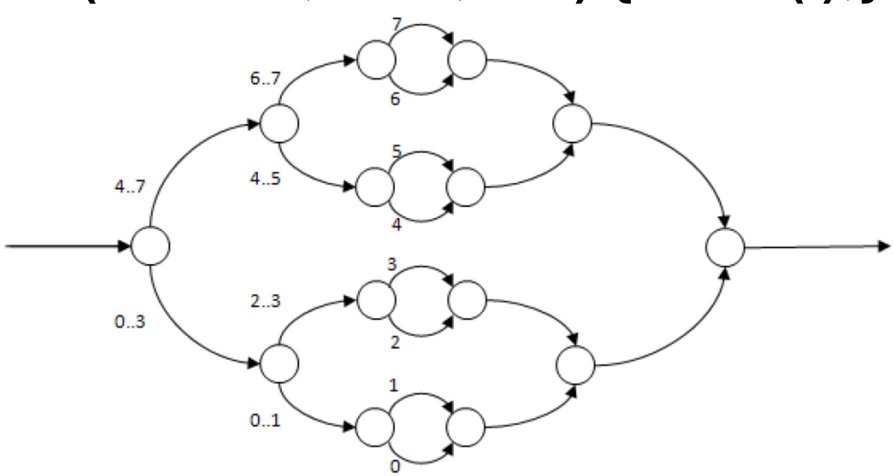
Figure credits: Intel Cilk++ Programmer's Guide. Document # 322581-001US.

Loop with a cilk_spawn vs. cilk_for

- for (int i = 0; i < 8; i++) { cilk_spawn work(i); } cilk_sync;



- cilk_for (int i = 0; i < 8; i++) { work(i);}



Note: computation on edges

cilk_for uses divide-and-conquer

Figure credits: Intel Cilk++ Programmer's Guide. Document # 322581-001US.

Restrictions for cilk_for

- No early exit
  - —no break or return statement within loop
  - —no goto in loop unless target is within loop body
- Loop induction variable restrictions
  - —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 loop variable within the loop
  - —cilk_for (unsigned int i = 1; i < x; ++i) x = f();
    - can’t modify end within the loop
  - —int i; cilk_for (i = 0; i<100; i++) { ... }
    - loop variable must be declared in the loop header

cilk_for Implementation Sketch

- **Recursive bisection used to subdivide iteration space down to chunk size**

```
void run_loop(first, last)
{

    if (last - first) < grainsize)
    {
        for (int i=first; i<last ++i) LOOP_BODY;
    }
    else
    {
        int mid = (last-first)/2;
        cilk_spawn run_loop(first, mid);
        run_loop(mid, last);
    }
}
```

cilk_for Grain Size

- Iterations divided into *chunks* to be executed serially
  - chunk is sequential collection of one or more iterations
- Maximum size of chunk is called *grain size*
  - grain size too small: spawn overhead reduces performance
  - grain size too large: reduces parallelism and load balance
- Default grain size
  - #pragma cilk grainsize = min(2048, N / (8*p))
- Can override default grain size
  - #pragma cilk grainsize = expr
    - expr is any C++ expression that yields an integral type (e.g. int)

      e.g. #pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
  - pragma must immediately precede cilk_for to which it applies

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

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

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

The Problem with Non-local Variables

- Nonlocal variables are a common programming construct
  - global variables = nonlocal variables in outermost scope
  - nonlocal = declared in a scope outside that where it is used
- Example

```
int sum = 0;
for(int i=1; i<n; i++) {
  sum += i;
}
```

Understanding a Data Race

- Example

```
int sum = 0;
cilk_for(int i=1; i<n; i++) {
  sum += i;
}
```

- What can go wrong?
  - concurrent reads and writes can interleave in unpredictable ways

time ↓

read sum
read sum
write sum + i_j
write sum + i_k

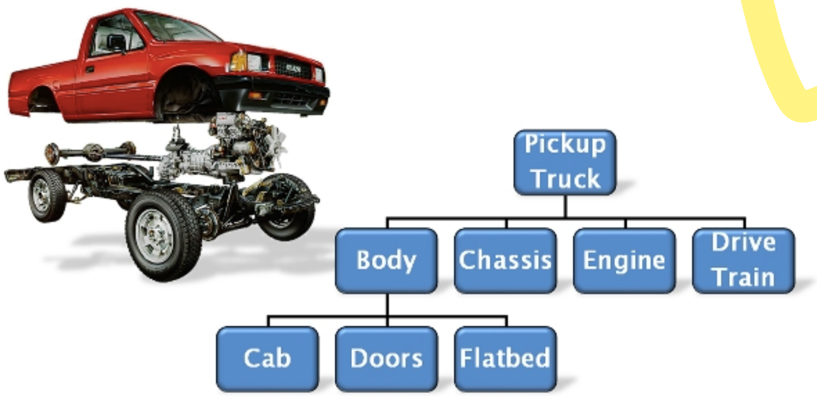
legend
thread n
thread m

  - the update by thread m is lost!

- Rewriting parallel applications to avoid non-local variables can be painful

Collision Detection

Automaker: hierarchical 3D CAD representation of assemblies



Computing a 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;
  }
}
```

Adding cilk_forParallelism

Computing a 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 variable causes data races!

Solution 1: Locking

Computing a cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
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**

Solution 2: Refactor the Code

```
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 *>>
           child_list(x.num_children);
       cilk_for (Node::const_iterator
                child = x->begin();
                child != x->end();
                ++child)
         walk(child, child_list[child]);
       for (int i=0; i < x.num_children; ++i)
         o_list.splice(o_list.end(), child_list[i]);
       break;
     }
```

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

Solution 3: Cilk Reducers

```
Node *target;
std::list<Node *> cilk_reducer(l, R) 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;
      }
    }
```

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

Cilk Reducers

- Reducers support update of nonlocal variables without races
  - —deterministic update using associative operations
    - e.g., global sum, list and output stream append, ...
    - result is same as serial version
      - independent of # processors or scheduling
- Can be used without significant code restructuring
- Can be used independently of the program's control structure
  - — unlike constructs defined only over loops
- Implemented efficiently with low overhead
  - —they don't use global mutual exclusion in their implementation
    - avoids loss of parallelism from enforcing mutual exclusion when updating a global variable

Cilk Reducers Operate on Monoids

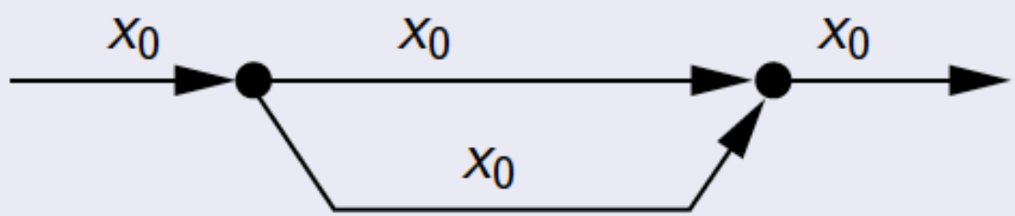
- Suppose that S is a set and • is some binary operation
  - $S \times S \rightarrow S$
- A monoid is a set that is closed under an associative binary operation and has an identity element
- S with • is a monoid if it satisfies the following two axioms:
  - identity element
    - there exists an element I in S such that for every element a in S, the equation $\mathrm{I} \bullet a = a \bullet \mathrm{I} = a$ holds
  - associativity
    - for all a, b and c in S, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds

Reducers Under the Hood

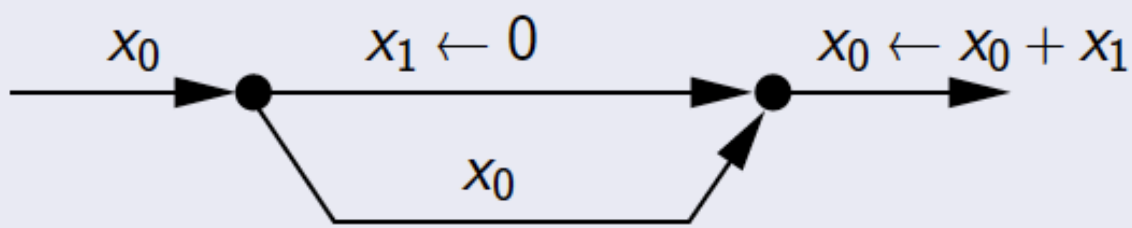
- If no steal occurs, a reducer behaves like a normal variable
- If a steal occurs
  - the continuation receives a view with an identity value
  - the child receives the reducer as it was prior to the spawn
  - at the corresponding cilk_sync
    - the value in the continuation is merged into the reducer held by the child using the reducer's `reduce` operation
    - the continuation's view is destroyed
    - the original (updated) object survives

Reducers

Serial execution (depth first):



Parallel execution:



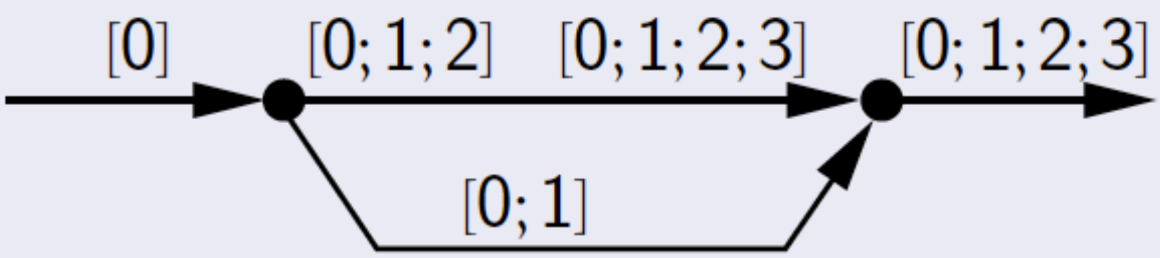
Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Reducing Over List Concatenation

Program:

```
x.append(0);
cilk_spawn x.append(1);
x.append(2);
x.append(3);
cilk_sync;
```

Serial execution:



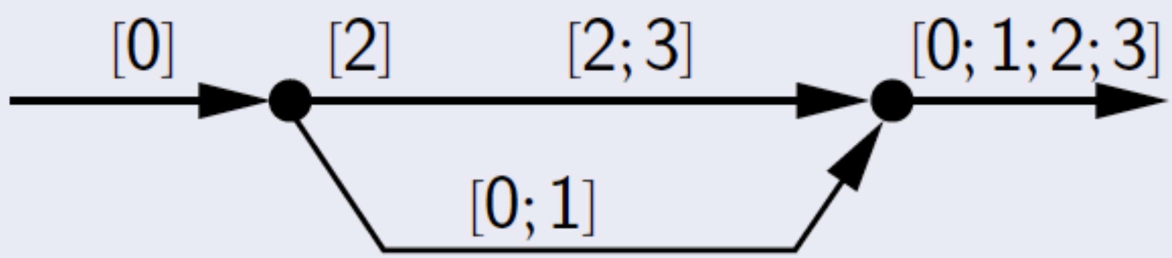
Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Reducing Over List Concatenation

Program:

```
x.append(0);
cilk_spawn x.append(1);
x.append(2);
x.append(3);
cilk_sync;
```

Parallel execution:



Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Reducer Demo - I

- **Compare a program with a racing reduction, a mutex protecting the race, and a reducer**
- **Versions:**
  - **sum_race.cpp: code with a racing sum reduction**
  - **sum_race.cpp: (-DSYNC) code with a mutex to avoid the race**
  - **sum_reducer.cpp: code with a reducer to avoid the race**
- **Compare performance of the various versions**
    - **cilk_run.sh**
  - **how does the performance of the parallel summations using reducers compare to**
    - **the parallel summations with races?**
    - **the parallel summations with locks?**
    - **the serial summation?**