Reducers and other Cilk++ hyperobjects

Matteo Frigo (Intel)
Pablo Halpern (Intel)
Charles E. Leiserson (MIT)
Stephen Lewin-Berlin (Intel)

August 11, 2009
Collision detection

Assembly:
Represented as a tree of subassemblies down to individual parts.

C++ code:
Finds all "collisions" between two assemblies. \( \approx 3000 \) lines of code.
**Simplified collision detection**

**Goal:**
Create a list of all the parts in a mechanical assembly that collide with a given target object.

**Pseudo code:**

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

void walk(Node *x) {
    if (x->kind == Node::LEAF) {
        if (target->collides_with(x)) {
            output_list.push_back(x);
        }
    } else {
        for (Node::iterator child = x.begin(); child != x.end(); ++child) {
            walk(child);
        }
    }
}
```
Naive parallelization

**Problem:**
Race condition on the global variable `output_list`.

**Pseudo code:**
```cpp
Node *target;
std::list<Node *> output_list;

void walk(Node *x) {
    if (x->kind == Node::LEAF) {
        if (target->collides_with(x)) {
            output_list.push_back(x);
        }
    } else {
        cilk_for (Node::iterator child = x.begin(); child != x.end(); ++child) {
            walk(child);
        }
    }
}
```
Naive parallelization

Problem:
Race condition on the global variable output_list.

Pseudo code:
Node *target;
std::list<Node *> output_list;

void walk(Node *x) {
    if (x->kind == Node::LEAF) {
        if (target->collides_with(x)) {
            output_list.push_back(x);
        }
    } else {
        cilk_for (
            Node::iterator child = x.begin();
            child != x.end();
            ++child) {
            output_list.push_back(x);
        }
    }
}
Problems:
Lock contention inhibits speedup.
Output order is nondeterministic.

Pseudo code:

```cpp
mutex lock;

void walk(Node *x) {
    if (x->kind == Node::LEAF) {
        if (target->collides_with(x)) {
            lock.acquire();
            output_list.push_back(x);
            lock.release();
        }
    } else {
        cilk_for (Node::iterator child = x.begin(); child != x.end(); ++child) {
            walk(child);
        }
    }
}
```
Hyperobject solution

This talk:
How to define output_list so that the program works
- correctly;
- efficiently;
- without code re-structuring.

Pseudo code:
Node *target;
cilk::reducer_list_append<Node *>
    output_list;

void walk(Node *x) {
    if (x->kind == Node::LEAF) {
        if (target->collides_with(x)) {
            output_list.push_back(x);
        }
    } else {
        cilk_for (  
            Node::iterator child = x.begin();
            child != x.end();
            ++child) {
            walk(child);
        }
    }
}
Outline

1. Reducers
2. Implementation of reducers
3. Other hyperobjects
4. Conclusion
The Cilk++ language and runtime system:

- Parallel extension of C++.
- Commercial spinoff of the MIT Cilk project [Leiserson et al.]

Cilk++ code

```cpp
int fib(int n)
{
    if (n < 2) return n;
    else {
        int x, y;
        x = cilk_spawn fib(n - 1);
        y = fib(n - 2);
        cilk_sync;
        return x + y;
    }
}
```

Serial elision

```cpp
int fib(int n)
{
    if (n < 2) return n;
    else {
        int x, y;
        x = fib(n - 1);
        y = fib(n - 2);
        return x + y;
    }
}
```
Computation DAGs

**Cilk++ code:**

```c++
A();
cilk_spawn B();
C();
cilk_sync;
D();
```

**Serial elision:**

```c++
A();
B();
C();
D();
```

**DAG:**

```
A --> C --> D

B
```


Hyperobjects

Object: All observers share the same view.

Hyperobject: The view depends upon the observer.

No determinacy races:

\[ A \parallel B \Rightarrow x_A \neq x_B \]
Reducers

**Serial execution (depth first):**

![Diagram of serial execution]

**Parallel execution:**

![Diagram of parallel execution]
Reducing over list concatenation

**Program:**

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

**Serial execution:**

```
[0] [0;1;2] [0;1;2;3] [0;1;2;3]
```

[Diagram showing the execution flow]
Reducing over list concatenation

**Program:**

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

**Parallel execution:**

```
[0] [2] [2; 3] [0; 1; 2; 3]
```

![Diagram showing parallel execution of the program]
Reducing over list concatenation

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

Parallel execution:
```
[0]
```
```
[0; 1; 2; 3]
```
```
[0; 1]
```
```
[2; 3]
```
```
[2]
```

Reduction over monoids is deterministic.
Monoid: associative operation with an identity.
Outline

1 Reducers

2 Implementation of reducers

3 Other hyperobjects

4 Conclusion
Implementation of reducers

- Implemented as part of the Cilk++ product.
- First-class objects:
  - Can be stored in arrays, classes, etc.
- Not tied to loop, map, fold, or other control structures.
- Allow user-defined reductions.
- Source code available on Sourceforge.
- Efficient implementation with work-stealing scheduler.
  - Overhead: mapping from reducer to view requires an associative lookup.
  - Paper in SPAA 2009.
Performance of collision detection

- **C++**
- **locks**
- **manual**
- **reducer**

The graph shows the time (in seconds) required for collision detection as a function of the number of processors. The time decreases significantly with an increase in the number of processors, indicating improved performance.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.
Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

Randomized work stealing:
When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

Randomized work stealing:
When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

Randomized work stealing:
When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

Randomized work stealing:
When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
The Cilk work-stealing scheduler

Each worker maintains a work deque, and it manipulates the bottom of the deque like a stack.

Randomized work stealing:
When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a **hypermap**, which maps reducers into views.

**Sequential execution:**
Local spawn/return operations create no new hypermaps and no new new reducer views.
Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a **hypermap**, which maps reducers into views.

**Sequential execution:**
Local spawn/return operations create no new hypermaps and no new reducer views.
Each worker maintains a **hypermap**, which maps reducers into views.

**Sequential execution:**
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Sequential execution:
Local spawn/return operations create no new hypermaps and no new reducer views.
Work stealing with reducers

Each worker maintains a hypermap, which maps reducers into views.

Parallel execution:
When stealing, the thief creates a new hypermap containing identity views.
Each worker maintains a hypermap, which maps reducers into views.

Parallel execution:
When stealing, the thief creates a new hypermap containing identity views.
Each worker maintains a **hypermap**, which maps reducers into views.

**Parallel execution:**

When stealing, the thief creates a new hypermap containing identity views.
Uses and abuses of reducers

- Reductions over arbitrary algorithms and data structures.
  - E.g., collision detection.
- “Composable” parallel dynamic scoping.
  - E.g., multiple reductions in parallel.
- Implementation of C++ exceptions.
- Reorder buffer in file I/O.
  - Parallelized bzip2 using a reducer.
  - Write the leftmost view to disk. No need to wait until completion.
- Generalization of thread-local storage.
  - E.g., region-based memory allocator as a reducer.
  - Backward compatible with serial semantics.
- Debugging/profiling.
  - Reduction of int over + with “identity” 1: counts the number of views.
Other hyperobjects

Splitters (hard to implement?):

\[ x_0 \rightarrow x_1 \leftarrow x_0 \rightarrow x_0 \leftarrow ? \]

Holders:

\[ x_0 \rightarrow x_1 \leftarrow 0 \rightarrow x_0 \leftarrow \bot \]
Related work

- OpenMP’s reduction construct.
  - Tied to parallel for loop.
  - “Shallow” lifting of reduction variable — simple name substitution within the loop.
  - Reduction variable is not first-class.
  - Fixed set of reduction operations.
- Data-parallel reduction (APL, NESL, ZPL, Matlab, etc.).
  - Reduction tied to the vector operation.
- MapReduce.
  - Reduction tied to the map function.
Stateful code is a fact of life.

Mutable state tends to be incompatible with parallelism.

Restructuring existing code is impractical.

Hyperobjects provide a way to parallelize a stateful program without changing the program’s logic.