Cilk Multithreaded Language

Introduction and implementation

Robert D. Blumfoe, Christopher F. Joerg, Bradley C. Kuszmaul, Yuli Zhou (V 1.)
Matteo Frigo (V 5., Cilk++)
Pablo Halpern, Stephen Lewin-Berlin (Cilk++)
Keith H. Randall (Cilk, Cilk5), Charles E Leiserson,
MIT Laboratory for Computer Science

Rice University
COMP 522
Advait Balaji
Outline

- Introduction
- The Cilk Language
- Work-first principle
- Cilk compiler
- Work Stealing
- Evaluations
- Cilk++ - Hyperobjects
- Summary
Introduction

• General purpose programming language for multi-threaded computing.
• Designed at MIT in 1990’s (Cilk-1 launched in 1994)
• Generalizes semantics of C language.
• Cilk Scheduler gives guarantee of application performance—Work Stealing!
• Performance measures – Work and Critical Path

https://www.cilkplus.org/cilk-history
Introduction

- Represented as a **DAG**. Collection of Cilk procedures and sequence of threads
- Each thread is **non-blocking**.
- Threads are required to spawn successor that can accept values from children.
- Thread receiving value can’t begin until another thread sends value – **Dependency**
- Execution is constrained to follow precedence relation determined by **DAG**.

The Cilk Language

• Philosophy- make Cilk a true parallel extension of C
  • On a parallel computer, Cilk control constructs allow the program to execute parallely
  • If Cilk keywords are elided- C elision
• On a uniprocessor – Cilk nearly as fast as C.
• Performance characterization measures
  • Work - Time used by one processor execution ($T_1$)
  • Critical Path – Time required for execution by an infinite processor ($T_\infty$)
  • For $P$ processors – $T_P \geq T_1/P$
• Follows the Work First principle
  • “Minimize scheduling overhead borne by the work of a computation. Move overheads out of work and onto the critical path”
The Cilk Language

• Work-first principle – strategy for compilation
  • Cilk2c compiler – transforms a Cilk source to a C postsource
  • C post source run through a gcc compiler
  • Two clones – “fast clone” and “slow clone”

• Communication due to scheduling occurs in the slow clone and contributes to the critical-path overhead.

• Work-first principle – Mutual exclusion and load-balancing scheduler

• ”Thieves” and “victims” – Idle-processor steal threads from busy processors. Guarantees overhead contributes only to critical-path

• Minimize work overhead – Dijkstra-like mutual exclusion (THE)
The Cilk Language

- Cilk-5

```cilk
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

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

cilk int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf("Result: %d\n", result);
    return 0;
}

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);
    }
}
```
Cilk Plus Terminology

- **Parallel control**
  - `cilk_spawn`, `cilk_sync`
  - return from spawned function

- **Strand**
  - maximal sequence of instructions not containing parallel control

```c
unsigned int fib(n) {
if (n < 2) return n;
else {
  unsigned int n1, n2;
  n1 = cilk_spawn fib(n - 1);
  n2 = cilk_spawn fib(n - 2);
  cilk_sync;
  return (n1 + n2);
}
}
```

- **Strand A**: code before first spawn
- **Strand B**: compute n-2 before 2nd spawn
- **Strand C**: n1+ n2 before the return
Cilk Program Execution as a DAG

Legend:
- **continuation**
- **spawn**
- **return**

Each circle represents a strand.

- **fib(4)**: A → B → C

- **fib(3)**: A → B → C

- **fib(2)**: A → B → C

- **fib(1)**: A → A

- **fib(0)**: A → A

The diagram illustrates the execution flow of a Cilk program, showing how functions are spawned and executed in parallel or sequentially.
The Cilk Language

• Cilk-5

```c
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** - C function internal to Cilk
  • In normal Cilk syntax – spawning cannot be linked to a statement
  • Inlets can call spawn as an argument.
  • Control of the parent procedure shifts to the statement after the inlet call.
  • Returned result added to x within inlet.
  • Cilk provides atomicity implicitly among threads so updates aren’t lost.
  • Don’t spawn from an inlet!
  • x += spawn fib(n-1)
The Cilk Language

• Cilk-5

```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);
    }
}
```

• Abort - ”Speculative work” can be aborted inside an inlet.
• Think parallel searches!
• When executed inside the inlet all the spawned children of the procedure automatically terminate.
• Authors considered using other synchronizing techniques but critical path cost too high.
• Sync - useful for systems that support relaxed memory-consistency model.
• Cilk programmers can also use additional locking for mutual exclusion – Future work
**Work-first principle**

- Three assumptions for work-first principle:
  - Cilk scheduler operates in practice according to the theoretical analysis
  - Ample ”Parallel slackness” (enough parallel work to keep all threads busy)
  - Every Cilk program has a C elision against which its one-processor performance is measured

- Two fundamental lower bounds must hold:
  - \( T_P \geq T_1/P \)
  - \( T_P \geq T_\infty \)

- Cilk’s randomized work-stealing scheduler executes a Cilk computation on \( P \) processors in expected time
  - \( T_P = T_1/P + O(T_\infty) \) ----[Eqn 1.]
  - This equation is optimal within a constant factor since RHS are both lower bounds.
Work-first principle

• The first term in equation 1 – work term and the second term is the critical path term.

• Modifying Eqn 1 to make overheads explicit:
  • \( T_P \leq \frac{T_1}{P} + c_\infty T_\infty \) ----[Eqn 2.]
  • Define smallest constant \( c_\infty \) as the critical-path overhead.

• Terms relevant to second assumption
  • Average parallelism \( \overline{P} = \frac{T_1}{T_\infty} \)
  • Parallel slackness \( \overline{P}/P \) (assumption. \( \gg c_\infty \))
  • From Equation 2 we have, \( \frac{T_1}{P} \gg c_\infty T_\infty ; T_P \cong \frac{T_1}{P} \)

• Third assumption
  • \( C_1 = \frac{T_1}{T_S} \)
  • \( T_P \leq c_1 T_S / P + c_\infty T_\infty ; c_1 T_S / P \) [Minimize \( C_1 \) even at the expense of larger \( C_\infty \)!]
Cilk’s compilation strategy

- Cilk scheduling
  - Worker maintains ready deque of ready procedures.
  - Worker operates on its tail- C call stack
  - Thief attempts to steal procedure; worker becomes a victim.
  - Thief grabs procedures from the head of the deque
  - When spawned fast clone runs and as soon as thief steals procedure converted to slow clone
Cilk’s compilation strategy

1. **Cilk2c**
   - Lines 4 and 5 represent the activation frame for `fib`. Frame initialized in 5 by storing static structure.
   - First spawn [Lines 12 – 18]
     - Lines 12-13 state of fib is saved onto the activation frame.
     - Lines 14-15 the frame is pushed on to the runtime deque.
     - Line 16 C call to function
     - Lines 17-18 check to whether parent procedure was stolen.

```
int fib (int n)
{
    fib_frame *f;
    f = alloc(sizeof(*f));
    f->sig = fib_sig;
    if (n<2) {
        free(f, sizeof(*f));
        return n;
    }
    else {
        int x, y;
        f->entry = 1;
        f->n = n;
        *T = f;
        push();
        x = fib (n-1);
        if (pop(x) == FAILURE)
            return 0;
        ...
        free(f, sizeof(*f));
        return (x+y);
    }
}
```
Cilk’s compilation strategy

- **Cilk2c**
  - In a **fast clone** all sync statements compile to no-ops.
  - Line 20, sync is empty! Line 21-22 fib deallocates the frame and returns to parent procedure.
  - **Slow clone** – when a procedure is stolen control has been suspended between spawn or sync points
  - **Goto** statement used to restore program counter after slow cone resumes.

```c
1 int fib (int n)
2 {
3   fib_frame *f;
4   f = alloc(sizeof(*f));
5   f->sig = fib_sig;
6   if (n<2) {
7     free(f, sizeof(*f));
8     return n;
9   }
10  else {
11     int x, y;
12     f->entry = 1;
13     f->n = n;
14     *T = f;
15     push();
16     x = fib (n-1);
17     if (pop(x) == FAILURE)
18        return 0;
19     ...  
20     ;
21     free(f, sizeof(*f));
22     return (x+y);
23   }
24 }
```
Cilk’s compilation strategy

- **Cilk2c runtime linkage**
  - Sync in slow clone – cilk2c inserts a call to runtime system which checks for spawned children
  - Parallel book-keeping is minimum as:
    - No contribution to *work*
    - Stealing guaranteed to be minimum
  - Separation between fast clones and slow clones allows efficient compilation of *inlets* and *abort*
  - Implicit inlet calls compile directly to C elision. An abort statement, similar to sync, is a no-op.
Cilk’s compilation strategy

• Runtime system tethers fast and slow clones
  • Includes protocols for stealing procedures, returning values between processors, executing inlets and aborting computation subtrees.
  • All costs amortized against critical path

• What’s the work overhead?
  • Stealing protocol executed by the worker
  • Allocating and freeing of activation frame, saving state before a spawn and checking if a procedure is stolen or not.
  • A small portion of this overhead is due to Cilk compiler duplicating the work done by the C compiler – overhead is small!
Cilk’s compilation strategy

• Allocating activation frames is an important step during Cilk2c operation
  • Cilk-4: Stack-based allocation
  • Cilk-5: Heap-based allocation

• So, Stack or Heap?
  • ‘Cactus Stack’ – Cilk-4 had to manage the virtual-memory map on each processor explicitly.
  • Overhead due to page fault in critical sections lead to complicated protocols- an expensive user-level interrupt during which memory map is modified
Cilk’s compilation strategy

Critical path is a concern!

• These overheads could be moved on to the critical path.
• But in practice, it overburdens the critical path and violates the assumption of parallel slackness.
• One-processor execution – fast but insufficient slackness sometimes resulted in poor parallel performance.

Cilk-5 has a Heap

• Frame allocated off a free list and deallocation requires frame to be pushed into free list. Heap allocation only slightly more expensive.
• Heap has a disadvantage—potentially waste a lot more memory because of fragmentation of memory.
Cilk’s compilation strategy

• Carefully evaluate critical-path overheads.
  • Can tip the scales where underlying parallel slackness assumption will not hold.

• Cilk-5 overhead believed to be optimal.
  • Portability vs performance tradeoff

• Lazy threads obtains better efficiency
  • Implementing its own calling conventions, stack layouts etc.
Work Stealing

• Work-stealing mechanism called “THE” protocol.

• Implementations:
  • Thief interrupts a worker and demand attention from this victim.
  • Post steal requests and workers could periodically poll them.

• Possible data-race between Thief and Victim- steal the same frame victim is trying to pop!

• One possible solution: add lock to deque

• Adopt a solution similar to mutual exclusion where only reads and writes are atomic!
### Work Stealing

- **“THE”**:  
  - 3 atomic variables, T, H, E  
  - Aim to move costs from the worker to the thief  
  - Many thieves and one victim - need a hardware lock  
  - Worker and a sole thief - can use mutual exclusion with little work overhead.

- **Pseudocode:**  
  - T, H stored in shared memory and visible to all processors.  
  - Worker treats deque as a stack  
    - Before spawn: push frame to tail  
    - After spawn: pop frame

```
<table>
<thead>
<tr>
<th>Worker/Victim</th>
<th>Thief</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 push() {</td>
<td>1</td>
</tr>
<tr>
<td>2 T++;</td>
<td>2</td>
</tr>
<tr>
<td>3 }</td>
<td>3</td>
</tr>
<tr>
<td>4 pop() {</td>
<td>4</td>
</tr>
<tr>
<td>5 T--;</td>
<td>5</td>
</tr>
<tr>
<td>6 if (H &gt; T) {</td>
<td>6</td>
</tr>
<tr>
<td>7 T++;</td>
<td>7</td>
</tr>
<tr>
<td>8 lock(L);</td>
<td>8</td>
</tr>
<tr>
<td>9 T--;</td>
<td>9</td>
</tr>
<tr>
<td>10 if (H &gt; T) {</td>
<td>10</td>
</tr>
<tr>
<td>11 T++;</td>
<td>11</td>
</tr>
<tr>
<td>12 unlock(L);</td>
<td>12</td>
</tr>
<tr>
<td>13 return FAILURE;</td>
<td>13</td>
</tr>
<tr>
<td>14 }</td>
<td>14</td>
</tr>
<tr>
<td>15 unlock(L);</td>
<td>15</td>
</tr>
<tr>
<td>16 }</td>
<td>16</td>
</tr>
<tr>
<td>17 return SUCCESS;</td>
<td>17</td>
</tr>
</tbody>
</table>
```

Work Stealing

- Always safe to push onto deque!
- Case (a) - enough frames available for thief and worker
- Case (b) - only one frame – data race condition!
- Case (c) – deque empty. Pop fails and steal fails!
  Will there be a deadlock?
- No significant overhead – Push just updates T and pop takes 6 operations. Expensive lock on theft- depends on $T_\infty$, can be considered critical path.
Work Stealing

• Performance:
  • Compared to pop with lock - THE performs 25% faster in UltraSPARC –I. Requires membar between lines 5 and 6.
  • On Pentium Pro- THE is only 5% faster, spends about half of its time in this memory fence

• “Non-blocking” THE has advantages:
  • Less prone to problems arising out of spin lock
  • The infrequency of locking means that a thief can usually complete a steal operation on the workers deque.
Work Stealing

- Introducing E:
  - Simplified model can be extended to incorporate communication.
  - In the simplified version, H marks head of deque and marks points that victim can’t cross.
  - Now, E marks this point and E > T asserted in line 6.
  - Lines 7 to 15 are replaced by a call to an exception handler.
  - Before stealing, thief increments E. If stolen, increment H, else, restore E.
  - Exception mechanism executes abort.
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>$T_1$</th>
<th>$T_\infty$</th>
<th>$\bar{P}$</th>
<th>$c_1$</th>
<th>$T_8$</th>
<th>$T_1/T_8$</th>
<th>$T_S/T_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>35</td>
<td>12.77</td>
<td>0.0005</td>
<td>25540</td>
<td>3.63</td>
<td>1.60</td>
<td>8.0</td>
<td>2.2</td>
</tr>
<tr>
<td>blockedmul</td>
<td>1024</td>
<td>29.9</td>
<td>0.0044</td>
<td>6730</td>
<td>1.05</td>
<td>4.3</td>
<td>7.0</td>
<td>6.6</td>
</tr>
<tr>
<td>notempmul</td>
<td>1024</td>
<td>29.7</td>
<td>0.015</td>
<td>1970</td>
<td>1.05</td>
<td>3.9</td>
<td>7.6</td>
<td>7.2</td>
</tr>
<tr>
<td>strassen</td>
<td>1024</td>
<td>20.2</td>
<td>0.58</td>
<td>35</td>
<td>1.01</td>
<td>3.54</td>
<td>5.7</td>
<td>5.6</td>
</tr>
<tr>
<td>*cilksort</td>
<td>4,100,000</td>
<td>5.4</td>
<td>0.0049</td>
<td>1108</td>
<td>1.21</td>
<td>0.90</td>
<td>6.0</td>
<td>5.0</td>
</tr>
<tr>
<td>†queens</td>
<td>22</td>
<td>150.</td>
<td>0.0015</td>
<td>96898</td>
<td>0.99</td>
<td>18.8</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>†knapsack</td>
<td>30</td>
<td>75.8</td>
<td>0.0014</td>
<td>54143</td>
<td>1.03</td>
<td>9.5</td>
<td>8.0</td>
<td>7.7</td>
</tr>
<tr>
<td>lu</td>
<td>2048</td>
<td>155.8</td>
<td>0.42</td>
<td>370</td>
<td>1.02</td>
<td>20.3</td>
<td>7.7</td>
<td>7.5</td>
</tr>
<tr>
<td>*cholesky</td>
<td>BCSSTK32</td>
<td>1427.</td>
<td>3.4</td>
<td>420</td>
<td>1.25</td>
<td>208.</td>
<td>6.9</td>
<td>5.5</td>
</tr>
<tr>
<td>heat</td>
<td>4096 × 512</td>
<td>62.3</td>
<td>0.16</td>
<td>384</td>
<td>1.08</td>
<td>9.4</td>
<td>6.6</td>
<td>6.1</td>
</tr>
<tr>
<td>fft</td>
<td>$2^{20}$</td>
<td>4.3</td>
<td>0.0020</td>
<td>2145</td>
<td>0.93</td>
<td>0.77</td>
<td>5.6</td>
<td>6.0</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>$2^{16}$</td>
<td>124.</td>
<td>0.15</td>
<td>853</td>
<td>1.02</td>
<td>16.5</td>
<td>7.5</td>
<td>7.4</td>
</tr>
</tbody>
</table>
Benchmarks

- 466 MHz Alpha 21164
  - 27 ns

- 200 MHz Pentium Pro
  - 78 ns

- 167 MHz Ultra SPARC I
  - 113 ns

- 195 MHz MIPS R10000
  - 115 ns

---

Normalized Speedup

- Experimental data
  - Model \( T_1/P + T_{inf} \)
  - Work bound
  - Critical path bound

Normalized Machine Size

02/19/2019

COMP 522 Multicore computing
Cilk++ Hyperobjects

- Non-local variables introduce “race conditions” in otherwise independent threads of multi-threaded program.
- A determinacy race exists if strands access the same shared location and at least one of the strands modifies values in the location.
- Code shows an example of walking down a binary tree to check for node property.
- There might be trouble in `output_list`! 

```cpp
bool has_property(Node *x);  
std::list<Node *> output_list;  
// ...
void walk(Node *x)  
{  
  if (x)  
    if (has_property(x))  
      output_list.push_back(x);  
    walk(x->left);  
    walk(x->right);  
}
```
**Cilk++ Hyperobjects**

**Solution:** Associate a mutual-exclusion lock (mutex) \( L \) with output list.

- Mutex is acquired in line 9 and released in line 11.
- But, Mutex creates a bottleneck.
- Alternative may be to restructure the code to accumulate the output lists in each sub-computation. Ordering might be a challenge but may be possible.

**Hyperobjects**  – Linguistic construct that allows strands to coordinate in updating a shared variable.
More on Hyperobjects

• Hyperobject as seen by a given strand of execution is called a “view”

• Strands view is private, but when two or more strands combine their views are combined.

• Any query or update to the hyperobject may update the strand’s view.

• Why are hyperobjects important?
  • Simplify the parallelization of programs with non-local variable, without forcing the programmer to restructure logic of the program.
Cilk++ Reducers

- Similar Reduce?
  - Open MP – reduction clause
  - Intel Thread Building Blocks
  - Microsoft parallel Pattern Library – combinalbe object

- Result is declared as the reduction variable.
- Iterations of the loop spread across processors and local copies of the variable result are made.
- In order for the result to be same as serial code reduction operation- associative and commutative

```cpp
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...
    int result(0);
    #pragma omp parallel for 
    reduction (+:result) 
    for (std::size_t i = 0; i < n; ++i) 
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```

Reducers in OpenMP
Cilk++ Reducers

- Reducers in Cilk++ similar to other languages with some augmentations
  - Can parallelize global or non-local variables
  - Associativity is necessary and sufficient.
  - Operate independently of control constructs
- `Sum_reducer<int>` declares result to be a reducer hyperobject over integers.
- Cilk for – all iterations of the loop can operate in parallel. This is similar to OpenMP but Cilk++ doesn’t wait to combine local views

```cpp
int compute(const X& v);
int cilk_main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...
    sum_reducer<int> result(0);
    cilk_for (std::size_t i = 0; i < n; ++i)
        result += compute(myArray[i]);
    std::cout << "The result is: " << result.get_value() << std::endl;
    return 0;
}
```
Cilk++ Reducers

- Tree walking code with reducers:
  - Declare a reducer output_list.
  - Output list has a list_append reducer operation
- Cilk++ runtime load balances computation.
- When the branches synchronize, the private views are reduced by concatenating the lists.
- No additional logic needs to be restructured!
- OpenMP, TBB and PPL have limitations w.r.t race-free parallelization.
Defining Reducers

- Define Reducers as Monoids:
  - Set $T$, operator $op$ and identity $e$
  - Closure, identity and associative defined
- In Cilk++, class M inherits from cilk::monoid_base<T>.
- Class M supplies a reduce() and identity().
- View() -> runtime returns the local view as a reference to underlying type T upon which M is defined.
- Two disadvantages:
  - clumsy syntax: for incrementing x will be x.view()++
  - Access to reducer is unconstrained x.view() *=2
- Wrap reducers into abstract data types

```cpp
struct sum_monoid : cilk::monoid_base<int> {
  void reduce(int* left, int* right) const {
    *left += *right;
  }
  void identity(int* p) const {
    new (p) int(0);
  }
};
cilk::reducer<sum_monoid> x;
```
Semantics of Reducers

- View of the reducer is an object that is uniquely “owned” by one strand.
- `Cilk_spawn` and `Cilk_sync` execution transfers or creates additional views.
- `Cilk_spawn` creates two new cilk++ strands (child and continuation)
- $X_C \leftarrow X_C$ $\text{op}$ $X_P$; Delete $X_P$; Parent strand P becomes the new owner of $X_C$.
- Why not swap the view of cont and child?
  - Helps in serial execution and allows the entire program to executed with a single view with no overhead for reduce.
  - Parent having no view does not result in error because parent doesn’t resume till child has returned
- Cilk++ doesn’t wait for sync to reduce -> Need unbounded amount of memory to store all unreduced views.
Implementation of Reducers

- Frames stalled at a `cilk_sync` lie outside any extended deque. The youngest frame of an extended deque has no children. All other frames in the extended deque have exactly one child.
- Cilk++ partitions frames into two classes: stack frames, which only store a continuation and a parent pointer (but not a lock, join counter, or list of children), and full frames, which store the full parallel state.
Implementation of Reducers

• Invariants:
  1. The oldest frame is a full frame
  2. A frame not belonging to deque is full frame
  3. All descendants of stack frames are stack frames
  4. Youngest frame on level-$i$ stack is the parent of frame on level-$i+1$ stack
  5. A stack frame belongs to (only) one deque
  6. Oldest frame is a stack frame created by spawn or a full frame
  7. Every frame except the oldest frame was created by a function call.
  8. When a frame is stolen it is converted to a full frame
  9. A frame being executed is the youngest frame in deque
  10. Execution of stack frame (frame has no children) cilk_sync is a no op.
Implementation of Reducers

**Function call.** To call a procedure instance B from a procedure instance A, a worker sets the continuation in A’s frame so that the execution of A resumes immediately after the call when B returns. The worker then allocates a stack frame for B and pushes B onto the current call stack as a child of A’s frame. The worker then executes B.

**Spawn.** To spawn a procedure instance B from a procedure instance A, a worker sets the continuation in A’s frame so that the execution of A resumes immediately after the cilk_spawn statement. The worker then allocates a stack frame for B, pushes the current call stack onto the tail of its deque, and starts a fresh current call stack containing only B. The worker then executes B.
Implementation of Reducers

Return from a call. If the frame A executing the return is a stack frame, the worker pops A from the current call stack. The current call stack is now nonempty (Invariant 6), and its youngest frame is A’a parent.

Return from a spawn. If the frame A executing the return is a stack frame, the worker pops A from the current call stack, which empties it (Invariant 7). The worker tries to pop a call stack S from the tail of its deque. If the pop operation succeeds (the deque was nonempty), the execution continues from the continuation of A’s parent (the youngest element of S), using S as the new current call stack. Otherwise, the worker begins random work stealing.

Sync. If the frame A executing a cilk_sync is a stack frame, do nothing. (Invariant 10). Otherwise, A is a full frame with a join counter. Pop A from the current call stack (which empties the extended deque by Invariant 1), increment A’s join counter, and steal A.
Modification for reducers

• Cilk++ uses address of reducer object as a key into hypermap hash table.
• Hypermap is lazy: elements are not stored until accessed for the first time
• Hypermaps maintained only in full frames
• USER hypermap, CHILDREN hypermap and RIGHT hypermap.

For left hypermap $L$ and right hypermap $R$, we define the operation $\text{REDUCE}(L, R)$ as follows. For all reducers $x$, set

$$L(x) \leftarrow L(x) \otimes R(x),$$

where $L(x)$ denotes the view resulting from the look-up of the address of $x$ in hypermap $L$, and similarly for $R(x)$. The left/right distinction is important, because the operation $\otimes$ might not be commutative. If the operation $\otimes$ is associative, the result of the computation is the same as if the program executed serially. $\text{REDUCE}$ is destructive: it updates $L$ and destroys $R$, freeing all memory associated with $R$. 
Modification for reducers

• **Return from a call.** Let C be a child frame of the parent frame P that originally called C, and suppose that C returns. We distinguish two cases: the “fast path” when C is a stack frame, and the “slow path” when C is a full frame.
  
  • If C is a stack frame, do nothing,
  
  • Otherwise, C is a full frame. We update $\text{USER}_P \leftarrow \text{USER}_C$, which transfers ownership of child views to the parent

• **Return from a spawn.** Let C be a child frame of the parent frame P that originally spawned C, and suppose that C returns.
  
  • If C is a stack frame, do nothing. Because C is a stack frame, P has not been stolen since C was spawned.
  
  • Otherwise, C is a full frame. We update $\text{USER}_C \leftarrow \text{REDUCE}(\text{USER}_C, \text{RIGHT}_C)$, which is to say that we reduce the views of all completed right-sibling frames of C into the views of C
Cilk general purpose multithreading language based on C/C++.

Adopts Work-first principle based on the assumption of sufficient parallelism.

Work-stealing protocol implemented on shared-memory between victim and thief processors.

“The” protocol results in significance performance speedup.

Efficient implementation of reducers and other hyperobjects help resolves determinacy race conditions.