Reducers and Other Cilk++ Hyperobjects

Matteo Frigo Pablo Halpern Charles E. Leiserson Stephen Lewin-Berlin

Cilk Arts, Inc.
55 Cambridge Street, Suite 200
Burlington, MA 01803

ABSTRACT

This paper introduces ***hyperobjects***, a linguistic mechanism that allows different branches of a multithreaded program to maintain coordinated local views of the same nonlocal variable. We have identified three kinds of hyperobjects that seem to be useful — ***reducers***, ***holders***, and ***splitters*** — and we have implemented reducers and holders in Cilk++, a set of extensions to the C++ programming language that enables multicore programming in the style of MIT Cilk. We analyze a randomized locking methodology for reducers and show that a work-stealing scheduler can support reducers without incurring significant overhead.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; D.3.3 [**Software**]: Language Constructs and Features—*Concurrent programming structures*.

General Terms

Algorithms, Languages, Theory.

1 INTRODUCTION

Many serial programs use ***nonlocal variables*** — variables that are bound outside of the scope of the function, method, or class in which they are used. If a variable is bound outside of all local scopes, it is a ***global variable***. Nonlocal variables have long been considered a problematic programming practice [22], but programmers often find them convenient to use, because they can be accessed at the leaves of a computation without the overhead and complexity of passing them as parameters through all the internal nodes. Thus, nonlocal variables have persisted in serial programming.

In the world of parallel computing, nonlocal variables may inhibit otherwise independent "strands" of a multithreaded program from operating in parallel, because they introduce "race conditions." We define a ***strand*** to be a sequence of executed instructions containing no parallel control. A ***determinacy race*** [7] (also called a ***general race*** [18]) exists if logically parallel strands access the same shared location, and at least one of the strands modifies the value in the location. A determinacy race is often a bug, because the program may exhibit unexpected, nondeterministic behavior depending on how the strands are scheduled. Serial code containing nonlocal variables is particularly prone to the introduction of determinacy races when the code is parallelized.

```
1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  // ...
4  void walk(Node *x)
5  {
6    if (x) {
7      if (has_property(x))
8        output_list.push_back(x);
9      walk(x->left);
10     walk(x->right);
11   }
12 }
```

Figure 1: C++ code to create a list of all the nodes in a binary tree that satisfy a given property.

As an example of how a nonlocal variable can introduce a determinacy race, consider the problem of walking a binary tree to make a list of which nodes satisfy a given property. A C++ code to solve the problem is abstracted in Figure 1. If the node `x` being visited is nonnull, the code checks whether `x` has the desired property in line 7, and if so, it appends `x` to the list stored in the global variable `output_list` in line 8. Then, it recursively visits the left and right children of `x` in lines 9 and 10.

Figure 2 illustrates a straightforward parallelization of this code in Cilk++, a set of simple extensions to the C++ programming language that enables multicore programming in the style of the MIT Cilk multithreaded programming language [8]. The keyword `cilk_spawn` preceding a function invocation causes the currently executing "parent" function to call the specified function just like a normal function call. Unlike a normal function call, however, the parent may continue executing in parallel with its spawned child, instead of waiting for the child to complete as with a normal function call. A `cilk_spawn` keyword does not say that the parent *must* continue executing in parallel with its child, only that it ***may***. (The Cilk++ runtime system makes these scheduling decisions in a provably efficient fashion, leaving the programmer to specify the potential for parallelism.) In line 9 of the figure, the `walk` function is spawned recursively on the left child, while the parent may continue on to execute an ordinary recursive call of `walk` in line 10. The `cilk_sync` statement in line 11 indicates that control should not pass this point until the spawned child returns. As the recursion unfolds, the running program generates a tree of parallel execution that follows the structure of the binary tree. Unfortunately, this

Charles E. Leiserson is also Professor of Computer Science and Engineering at MIT. This work was supported in part by the National Science Foundation under SBIR Grants 0712243 and 0822896.

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

```
1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  // ...
4  void walk(Node *x)
5  {
6    if (x) {
7      if (has_property(x))
8        output_list.push_back(x);
9      cilk_spawn walk(x->left);
10     walk(x->right);
11     cilk_sync;
12   }
13 }
```

Figure 2: A naive Cilk++ parallelization of the code in Figure 1. This code has a determinacy race in line 8.

```
1  bool has_property(Node *);
2  std::list<Node *> output_list;
3  mutex L;
4  // ...
5  void walk(Node *x)
6  {
7    if (x) {
8      if (has_property(x)) {
9        L.lock();
10       output_list.push_back(x);
11       L.unlock();
12     }
13     cilk_spawn walk(x->left);
14     walk(x->right);
15     cilk_sync;
16   }
17 }
```

Figure 3: Cilk++ code that solves the determinacy race using a mutex.

naive parallelization contains a determinacy race. Specifically, two parallel strands may attempt to update the shared global variable `output_list` in parallel at line 8.

The traditional solution to fixing this kind of determinacy race is to associate a mutual-exclusion lock (mutex) `L` with `output_list`, as is shown in Figure 3. Before updating `output_list`, the mutex `L` is acquired in line 9, and after the update, it is released in line 11. Although this code now operates correctly, the mutex may create a bottleneck in the computation. If there are many nodes that have the desired property, the contention on the mutex can destroy all the parallelism. For example, on one set of test inputs for a real-world tree-walking code that performed collision-detection of mechanical assemblies, lock contention actually degraded performance on 4 processors so that it was worse than running on a single processor.

In addition, the locking solution has the problem that it jumbles up the order of list elements. For this application, that might be okay, but some applications may depend on the order produced by the serial execution.

An alternative to locking is to restructure the code to accumulate the output lists in each subcomputation and concatenate them when the computations return. If one is careful, it is also possible to keep the order of elements in the list the same as in the serial execution. For the simple tree-walking code, code restructuring may suffice, but for many larger codes, disrupting the original logic can be time-consuming and tedious undertaking, and it may require expert skill, making it impractical for parallelizing large legacy codes.

This paper provides a novel approach to avoiding determinacy races in code with nonlocal variables. We introduce "hyperobjects," a linguistic construct that allows many strands to coordinate in updating a shared variable or data structure independently by providing different but coordinated views of the object to different threads at the same time. Hyperobjects avoid problems endemic to locking, such as lock contention, deadlock, priority inversion, convoying, etc. We describe three kinds of hyperobjects: reducers, holders, and splitters.

The hyperobject as seen by a given strand of an execution is called the strand's "view" of the hyperobject. A strand's view is not a value, but a stateful object with a memory address (a C++ "lvalue"). A strand can access and change its view's state independently, without synchronizing with other strands. Throughout the execution of a strand, the strand's view of the hyperobject is private, thereby providing isolation from other strands. When two or more strands join, their different views are combined according to a system- or user-defined method, one or more of the views may be destroyed, and one or more of the views may be transferred to another strand. The identity of the hyperobject remains the same from strand to strand, even though the strands' respective views of the hyperobject may differ. Thus, any query or update to the hyperobject — whether free or bound in a linguistic construct, whether accessed as a named variable, as a global variable, as a field in an object, as an element of an array, as a reference, as a parameter, through a pointer, etc. — may update the strand's view. This transparency of reference, whereby a strand's query or update to a hyperobject always refers to the strand's view, is not tied to any specific linguistic construct, but happens automatically wherever and whenever the hyperobject is accessed. Hyperobjects simplify the parallelization of programs with nonlocal variables, such as the global variable `output_list` in Figure 1. Moreover, they preserve the advantages of parallelism without forcing the programmer to restructure the logic of his or her program.

The remainder of this paper is organized as follows. Section 2 describes prior work on "reduction" mechanisms. Section 3 describes reducer hyperobjects, which allow associative updates on nonlocal variables to be performed in parallel, and Section 4 describes how we have implemented them in Cilk++. Section 5 describes and analyzes a randomized protocol for ensuring atomicity in the reducer implementation which incurs minimal overhead for mutual-exclusion locking. Section 6 describes holder hyperobjects, which can be viewed as a structured means of providing thread-local storage. Section 7 describes splitter hyperobjects, which provide a means of parallelizing codes that perform an operation on a nonlocal variable; call a subroutine, perhaps recursively; and then undo the operation on the nonlocal variable. Section 8 concludes with a discussion of more general classes of hyperobjects.

2 BACKGROUND

The first type of hyperobject we shall examine is a "reducer," which is presented in detail in Section 3. In this section, we'll review the notion of a reduction and how concurrency platforms have supported reductions prior to hyperobjects.

The idea of "reducing" a set of values dates back at least to the programming language APL [12], invented by the late Kenneth Iverson. In APL, one can "sum-reduce" the elements of a vector `A` by simply writing `+/A`, which adds up all the numbers in the vector. APL provided a variety of reduction operators besides addition, but it did not let users write their own operators. As parallel computing technology developed, reductions naturally found their way into parallel programming languages — including *Lisp [14], NESL [2], ZPL [5], and High Performance Fortran [13], to name only a few — because reduction can be easily implemented as a logarithmic-height parallel tree of execution.

The growing set of modern multicore concurrency platforms all feature some form of reduction mechanism:

- OpenMP [19] provides a reduction clause.
- Intel's Threading Building Blocks (TBB) [20] provides a `parallel_reduce` template function.
- Microsoft's upcoming Parallel Pattern Library (PPL) [15] provides a "combinable object" construct.

For example, the code snippet in Figure 4 illustrates the OpenMP syntax for a sum reduction within a parallel `for` loop. In this code,

```
1  int compute(const X& v);
2  int main()
3  {
4    const std::size_t n = 1000000;
5    extern X myArray[n];
6    // ...
7    int result(0);
8    #pragma omp parallel for \
9            reduction (+:result)
10   for (std::size_t i = 0; i < n; ++i) {
11     result += compute(myArray[i]);
12   }
13   std::cout << "The result is: " << result
14             << std::endl;
15   return 0;
16 }
```

Figure 4: An example of a sum reduction in OpenMP.

```
1  int compute(const X& v);
2  int cilk_main()
3  {
4    const std::size_t n = 1000000;
5    extern X myArray[n];
6    // ...
7    sum_reducer<int> result(0);
8    cilk_for (std::size_t i = 0; i < n; ++i)
9      result += compute(myArray[i]);
10
11   std::cout << "The result is: "
12             << result.get_value()
13             << std::endl;
14   return 0;
15 }
```

Figure 5: A translation of the code in Figure 4 into Cilk++ with reducers.

```
1  bool has_property(Node *);
2  list_append_reducer<Node *> output_list;
3  // ...
4  void walk(Node *x)
5  {
6    if (x) {
7      if (has_property(x))
8        output_list.push_back(x);
9      cilk_spawn walk(x->left);
10     walk(x->right);
11     cilk_sync;
12   }
13 }
```

Figure 6: A Cilk++ parallelization of the code in Figure 1 which uses a reducer hyperobject to avoid determinacy races.

the variable `result` is designated as a reduction variable of a parallel loop in the pragma preceding the `for` loop. Without this designation, the various iterations of the parallel loop would race on the update of `result`. The iterations of the loop are spread across the available processors, and local copies of the variable `result` are created for each processor. At the end of the loop, the processors' local values of `result` are summed to produce the final value. In order for the result to be the same as the serial code produces, however, the reduction operation must be associative and commutative, because the implementation may jumble up the order of the operations as it load-balances the loop iterations across the processors.

TBB and PPL provide similar functionality in their own ways. All three concurrency platforms support other reduction operations besides addition, and TBB and PPL allow programmers to supply their own. Moreover, TBB does not require the reduction operation to be commutative in order to produce the same result as serial code would produce — associativity suffices.

3 REDUCERS

The hyperobject approach to reductions differs markedly from earlier approaches, as well as those of OpenMP, TBB, and PPL. Although the general concept of reduction is similar, Cilk++ reducer hyperobjects provide a flexible and powerful mechanism that offers the following advantages:

- Reducers can be used to parallelize many programs containing global (or nonlocal) variables without locking, atomic updating, or the need to logically restructure the code.
- The programmer can count on a deterministic result as long as the reducer operator is associative. Commutativity is not required.
- Reducers operate independently of any control constructs, such as parallel `for`, and of any data structures that contribute their values to the final result.

This section introduces reducer hyperobjects, showing how they can be used to alleviate races on nonlocal variables without substantial code restructuring. We explain how a programmer can define custom reducers in terms of algebraic monoids, and we give an operational semantics for reducers.

Using reducers

Figure 5 illustrates how the code in Figure 4 might be written in Cilk++ with reducers. The `sum_reducer<int>` template, which we will define later in this section (Figure 8), declares `result` to be a reducer hyperobject over integers with addition as the reduction operator. The `cilk_for` keyword indicates that all iterations of the loop can operate in parallel, similar to the parallel `for` pragma in OpenMP. As with OpenMP, the iterations of the loop are spread across the available processors, and local views of the variable ***result*** are created. There, however, the similarity ends, because Cilk++ does not wait until the end of the loop to combine the local views, as OpenMP does. Instead, it combines them in such a way that the operator (addition in this case) need not be commutative to produce the same result as would a serial execution. When the loop is over, the underlying integer value can be extracted from the reducer using the `get_value()` member function.

As another example, Figure 6 shows how the tree-walking code from Figure 1 might be parallelized using a reducer. Line 2 declares `output_list` to be a reducer hyperobject for list appending. (We will define the `list_append_reducer` later in this section (Figure 9).) This parallelization takes advantage of the fact that list appending is associative. As the Cilk++ runtime system load-balances this computation over the available processors, it ensures that each branch of the recursive computation has access to a private view of the variable `output_list`, eliminating races on this global variable without requiring locks. When the branches synchronize, the private views are reduced by concatenating the lists, and Cilk++ carefully maintains the proper ordering so that the resulting list contains the identical elements in the same order as in a serial execution.

By using reducers, all the programmer does is identify the global variables as reducers when they are declared. No logic needs to be restructured, and if the programmer fails to catch all the use instances, the compiler reports a type error. By contrast, most concurrency platforms have a hard time expressing race-free parallelization of this kind of code. The reason is that reductions in most languages are tied to a control construct. For example, reduction in OpenMP is tied to the parallel `for` loop pragma. Moreover, the set of reductions in OpenMP is hardwired into the language, and list appending is not supported. Consequently, OpenMP cannot solve the problem of races on global variables using its mechanism. TBB and PPL have similar limitations, although they do allow programmer-defined reduction operators.

```
1  struct sum_monoid : cilk::monoid_base<int> {
2    void reduce(int* left, int* right) const {
3      *left += *right;
4    }
5    void identity(int* p) const {
6      new (p) int(0);
7    }
8  };
9
10 cilk::reducer<sum_monoid> x;
```

Figure 7: A C++ representation of the monoid $(\mathbf{Z}, +, 0)$, of integers (more precisely, `int`'s) with addition. Line 10 defines `x` to be a reducer over `sum_monoid`.

Defining reducers

Hyperobject functionality is not built into the Cilk++ language and compiler. Rather, hyperobjects are specified as ordinary C++ classes that interface directly to the Cilk++ runtime system. A Cilk++ reducer can be defined with respect to any C++ class that implements an algebraic "monoid." Recall that an algebraic ***monoid*** is a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an associative binary operation over T with identity e. In Cilk++, a monoid $(T, \otimes, e)$ is defined in terms of a C++ class `M` that inherits from the base class `cilk::monoid_base<T>`, where `T` is a type that represents the set T. The class `M` must supply a member function `reduce()` that implements the binary operator $\otimes$ and a member function `identity()` that constructs a fresh identity e. (If the `identity()` function is not defined, it defaults to the value produced by the default constructor for `T`.) Figure 7 shows a simple definition for the monoid of integers with addition.

The template `cilk::reducer<M>` is used to define a reducer over a monoid `M`, as is shown in line 10 of Figure 7, and it connects the monoid to the Cilk++ runtime system. When the program accesses the member function `x.view()`, the runtime system looks up and returns the local view as a reference to the underlying type `T` upon which the monoid `M` is defined. The template also defines `operator()` as a synonym for `view()`, so that one can write the shorter `x()`, instead of `x.view()`.

As a practical matter, the `reduce()` function need not actually be associative — as in the case of floating-point addition — but a reducer based on such a "monoid" may operate nondeterministically. Similarly, `identity()` need not be a true identity. If `reduce()` is associative and `identity()` is a true identity, however, the behavior of such a "properly defined" reducer is guaranteed to be the same no matter how the computation is scheduled. Properly defined reducers greatly simplify debugging, because they behave deterministically.

Although a definition such as that in Figure 7 suffices for obtaining reducer functionality, it suffers from two problems. First, the syntax for accessing reducers provided by Figure 7 is rather clumsy. For example, in order to increment the reducer `x` from Figure 7, a programmer needs to write `x.view()++` (or `x()++`), rather than the simpler `x++`, as is probably written in the programmer's legacy C++ code. Second, access to the reducer is unconstrained. For example, even though the reducer in Figure 7 is supposed to reduce over addition, nothing prevents a programmer from accidentally writing `x.view() *= 2`, because `x.view()` is an ordinary reference to `int`, and the programmer is free to do anything with the value he or she pleases.

To remedy these deficiencies, it is good programming practice to "wrap" reducers into abstract data types. For example, one can write a library wrapper, such as is shown in Figure 8, which allows the code in Figure 5 to use the simple syntax `result += X`. Moreover, it forbids users of the library from writing `result *= X`, which would be inconsistent with a summing reducer. Similarly, Figure 9 shows how the monoid of lists with operation append might be similarly wrapped.

```
1  template<class T>
2  class sum_reducer
3  {
4    struct Monoid : cilk::monoid_base<T> {
5      void reduce(T* left, T* right) const {
6        *left += *right;
7      }
8      void identity(T* p) const {
9        new (p) T(0);
10     }
11   };
12
13   cilk::reducer<Monoid> reducerImp;
14
15 public:
16   sum_reducer() : reducerImp() { }
17
18   explicit sum_reducer(const T &init)
19     : reducerImp(init) { }
20
21   sum_reducer& operator+=(T x) {
22     reducerImp.view() += x;
23     return *this;
24   }
25
26   sum_reducer& operator-=(T x) {
27     reducerImp.view() -= x;
28     return *this;
29   }
30
31   sum_reducer& operator++() {
32     ++reducerImp.view();
33     return *this;
34   }
35
36   void operator++(int) {
37     ++reducerImp.view();
38   }
39
40   sum_reducer& operator--() {
41     --reducerImp.view();
42     return *this;
43   }
44
45   void operator--(int) {
46     --reducerImp.view();
47   }
48
49   T get_value() const {
50     return reducerImp.view();
51   }
52 };
```

Figure 8: The definition of `sum_reducer` used in Figure 5.

Cilk++ provides a library of frequently used reducers, which includes a summing reducer (called `reducer_opadd`), list append reducers, and so on. Programmers can also write their own reducers in the style shown in Figure 8.

Semantics of reducers

The semantics of reducers can be understood operationally as follows. At any time during the execution of a Cilk++ program, a view of the reducer is an object that is uniquely "owned" by one strand in the Cilk++ program. If h is a reducer and S is a strand, we denote by h_S the view of h owned by S. When first created, the reducer consists of a single view owned by the strand that creates the hyperobject. When a Cilk directive such as `cilk_spawn` and `cilk_sync` is executed, however, ownership of views may be transferred and additional views may be created or destroyed.

In particular, a `cilk_spawn` statement creates two new Cilk++ strands: the child strand that is spawned, and the parent strand that continues after the `cilk_spawn` statement. Upon a `cilk_spawn` statement:

- The child strand owns the view owned by the parent function before the `cilk_spawn`.
- The parent strand owns a new view, initialized to e.

After a spawned child returns, the view owned by the child is "reduced" with the view owned by the parent. To ***reduce*** the view x_C

```
1  template<class T>
2  class list_append_reducer
3  {
4    struct Monoid
5      : cilk::monoid_base<std::list<T> >
6    {
7      void identity(std::list<T>* p) const {
8        new (p) std::list<T>;
9      }
10     void reduce(std::list<T>* a,
11                 std::list<T>* b) const {
12       a->splice(a->end(), *b);
13     }
14   };
15
16   cilk::reducer<Monoid> reducerImp;
17
18 public:
19   list_append_reducer() : reducerImp() { }
20
21   void push_back(const T& element) {
22     reducerImp().push_back(element);
23   }
24
25   const std::list<T>& get_value() const {
26     return reducerImp();
27   }
28 };
```

Figure 9: The definition of `list_append_reducer` used in Figure 6.

of a completed child strand C with the view x_P of a parent strand P means the following:

- $x_C \leftarrow x_C \otimes x_P$, where the symbol "$\leftarrow$" denotes the assignment operator and $\otimes$ is the binary operator implemented by the appropriate `reduce()` function. As a "special" optimization, if a view x is combined with the identity view e, Cilk++ assumes that the resulting view can be produced as x without applying a `reduce()` function.
- Destroy the view x_P.
- The parent strand P becomes the new owner of x_C.

Why do we choose a spawned child to own the view owned by the parent function before the `cilk_spawn`, rather than passing the view to the continuation and creating a new view for the child? The reason is that in a serial execution, the "special" optimization above allows the entire program to be executed with a single view with no overhead for reducing.

The Cilk++ runtime system guarantees that all children views are reduced with the parent by the time the parent passes the `cilk_sync` construct that waits for those children, and that all reductions are performed in some order consistent with the serial execution of the program. The Cilk++ runtime system does not delay all reductions until a `cilk_sync`, however, because such a delay may require an unbounded amount of memory to store all unreduced views. Instead, we allow the views of completed children to be reduced with each other at any time before passing the `cilk_sync`, provided that the serial left-to-right order is preserved.

At an ordinary function call, the child inherits the view owned by the parent, the parent owns nothing while the child is running, and the parent reacquires ownership of the view when the child returns. The fact that the parent owns no view while the child is running does not cause an error, because the parent performing a function call does not resume execution until the child returns.

No special handling of reducers is necessary for `cilk_for` loops, because the Cilk++ compiler translates the loop into divide-and-conquer recursion using `cilk_spawn` and `cilk_sync` so that each iteration of the loop body conceptually becomes a leaf of a logarithmic-depth tree of execution. Thus, the runtime system only needs to manage reducers at `cilk_spawn`'s and `cilk_sync`'s.

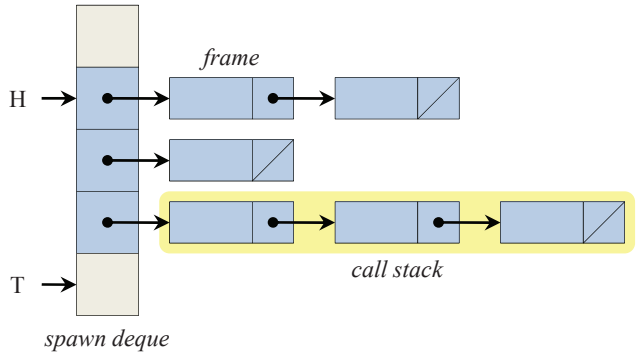

Figure 10: Runtime system data structures as seen by a single worker. Each worker owns a spawn deque, each element of which is a call stack implemented as a linked list of frames, ordered left to right in Figure 10 as youngest to oldest. The deque itself is implemented as an array of pointers, where array position X contains a valid pointer for $H \leq X < T$.

4 IMPLEMENTATION OF REDUCERS

This section describes how we have implemented reducers in Cilk++. We begin with a brief overview of the Cilk++ runtime system, which mimics aspects of the MIT Cilk runtime system [8]. Then, we describe the changes necessary to implement reducers. Finally, we discuss some optimizations.

The Cilk++ runtime system implements a work-stealing scheduler in the style of [3]. A set of ***worker*** threads (such as a Pthread [11] or Windows API thread [10]) cooperate in the execution of a Cilk++ program. As long as a worker has work to do, it operates independently of other workers. When idle, a worker obtains work by stealing it from another worker. Recall from Section 1 that the `cilk_spawn` keyword indicates the potential for concurrent execution rather than mandating it. This potential parallelism is realized only if stealing actually occurs.

Runtime data structures

Frames. Calling or spawning a Cilk++ procedure creates a new procedure instance, which results in the runtime creation of an ***activation record***, or ***frame***. As in C++ and many other languages, the frame provides storage for the local variables of the procedure instance, storage for temporary values, linkage information for returning values to the caller, etc. In addition, Cilk++ frames maintain the following state needed for a parallel execution:

- a lock;
- a ***continuation***, which contains enough information to resume the frame after a suspension point;
- a ***join counter***, which counts how many child frames are outstanding;
- a pointer to the parent frame;
- a doubly-linked list of outstanding children — specifically, each frame keeps pointers to its first child, its left sibling, and its right sibling.

Although frames are created and destroyed dynamically during the execution of the program, they always form a rooted tree (or what is sometimes called a "cactus stack" reminiscent of [17]). We say that a node in the cactus stack is ***older*** than its descendants and ***younger*** than its ancestors.

Data structures of a worker. Figure 10 illustrates the runtime system data structures from the point of view of a worker. The primary scheduling mechanism is a ***spawn deque***[1] of ***call stacks***, where each call stack is implemented as a (singly) linked list of

[1] A deque [6, p. 236] is a double-ended queue.

frames, each frame pointing to its parent. Each worker owns a spawn deque. The spawn deque is an "output-restricted" deque, in that a worker can insert and remove call stacks on the ***tail*** end of its deque (indexed by T), but other workers ("thieves") can only remove from the ***head*** end (indexed by H). In addition to the call stacks stored within the spawn deque, each worker maintains a ***current call stack*** — a call stack under construction that has not been pushed onto the deque — as well as other ancillary structures such as locks and free lists for memory allocation. Although we store the current call stack separately, it is sometimes convenient to view it as part of an ***extended deque***, where we treat the current call stack abstractly as an extra element of the deque at index T.

Stack frames and full frames. At any point in time during the execution, frames stalled at a `cilk_sync` lie outside any extended deque, but those that belong to an extended deque admit a simplified storage scheme. The youngest frame of an extended deque has no children, unless it is also the oldest frame in the extended deque. All other frames in the extended deque have exactly one child. Thus, there is no need to store the join counter and the list of children for frames in an extended deque, except for the oldest frame. Thus, 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.

This partitioning improves the overall efficiency of the system [8]. Roughly speaking, stack-frame manipulation is cheap and is inlined by the Cilk++ compiler, whereas full-frame manipulation is more expensive, usually involving the acquisition of a lock. Figure 11 shows a typical instance of the runtime-system data structures, illustrating deques, stack frames, and full frames.

Invariants

In order to understand the operation of the Cilk++ runtime system, it is helpful to bear in mind the following invariants, which we state without proof.

1. The oldest frame in an extended deque, if any, is a full frame. All other frames are stack frames.
2. A frame not belonging to any extended deque is a full frame.
3. All descendants of a stack frame are stack frames. Equivalently, all ancestors of a full frame are full frames.
4. In each extended deque, the youngest frame on a level-i call stack is the parent of the frame on the level-$i+1$ call stack.
5. A stack frame belongs to one (and only one) extended deque.
6. The oldest frame in a call stack is either a stack frame created by a spawn, or a full frame. That is, the oldest frame was not created by a function call.)
7. Every frame in a call stack, except for the oldest (Invariant 6), was created by a function call, that is, not by a spawn.
8. When a stack frame is stolen, it is promoted to a full frame. Thus, a stack frame has never been stolen.
9. A frame being executed by a worker is the youngest frame in the worker's extended deque.
10. While a worker executes a stack frame, the frame has no children, and thus the execution of a `cilk_sync` statement is a no-op. (This invariant is false for full frames.)

Actions of the runtime system

A Cilk++ program executes most of the time as a C++ program. Its execution differs from C++ at distinguished points of the program: when calling and spawning functions, when synching, and when returning from a function. We now describe the action of the runtime system at these points. The actions we describe are intended to execute as if they were atomic, which is enforced using locks stored in full frames, as described in Section 5.

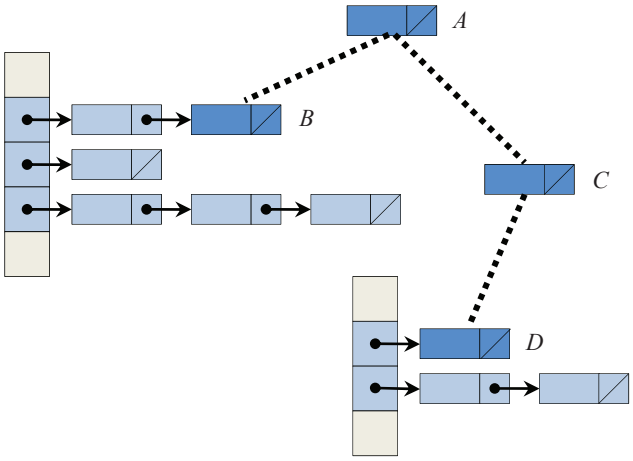

Figure 11: A global view of the Cilk++ runtime system data structures. Rectangles represent frames, with the dark rectangles denoting full frames. Frames A and C belong to no deque, and consequently they are full. In particular, C stores an explicit pointer to its full-frame parent A. Frames B and D are full because they are the oldest frames in their respective deques. Their children are stored implicitly in the deque, but these frames maintain an explicit pointer to their respective full-frame parents A and C.

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.

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. The worker resumes the execution from the continuation of A's parent.

Otherwise, the worker pops A (a full frame) from the current call stack. The worker's extended deque is now empty (Invariant 1). The worker executes an unconditional-steal of the parent frame (which is full by Invariant 3).

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.

If A is a full frame, the worker pops A from the current call stack, which empties the worker's extended deque (Invariant 1). The worker executes a provably-good-steal of the parent frame (which is full by Invariant 3).

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 provably-good-steal A.[2]

[2]The counter-intuitive increment of the join counter arises because we consider a sync equivalent to a spawn of a fake child in which the parent is immediately stolen and the child immediately returns. The increment

Randomly steal work. When a worker w becomes idle, it becomes a ***thief*** and steals work from a ***victim*** worker chosen at random, as follows:

- Pick a random victim v, where $v \neq w$. Repeat this step while the deque of v is empty.
- Remove the oldest call stack from the deque of v, and promote all stack frames to full frames. For every promoted frame, increment the join counter of the parent frame (full by Invariant 3). Make every newly created child the rightmost child of its parent.
- Let *loot* be the youngest frame that was stolen. Promote the oldest frame now in v's extended deque to a full frame and make it the rightmost child of *loot*. Increment *loot*'s join counter.
- Execute a resume-full-frame action on *loot*.

Provably good steal. Assert that the frame A begin stolen is a full frame and the extended deque is empty. Decrement the join counter of A. If the join counter is 0 and no worker is working on A, execute a resume-full-frame action on A. Otherwise, begin random work stealing.[3]

Unconditionally steal. Assert that the frame A being stolen is a full frame, the extended deque is empty, and A's join counter is positive. Decrement the join counter of A. Execute a resume-full-frame action on A.

Resume full frame. Assert that the frame A being resumed is a full frame and the extended deque is empty. Set the current call stack to a fresh stack consisting of A only. Execute the continuation of A.

Modifications for reducers

The Cilk++ implementation of reducers uses the address of the reducer object as a key into a ***hypermap*** hash table, which maps reducers into local views for the worker performing the look-up. Hypermaps are lazy: elements are not stored in a hypermap until accessed for the first time, in which case the Cilk++ runtime system inserts an identity value of the appropriate type into the hypermap. Laziness allows us to create an ***empty hypermap*** $\emptyset$, defined as a hypermap that maps all reducers into views containing identities, in $\Theta(1)$ time.

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

$$L(\mathtt{x}) \leftarrow L(\mathtt{x}) \otimes R(\mathtt{x}) \ ,$$

where $L(\mathtt{x})$ denotes the view resulting from the look-up of the address of $\mathtt{x}$ in hypermap L, and similarly for $R(\mathtt{x})$. 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. REDUCE is destructive: it updates L and destroys R, freeing all memory associated with R.

The Cilk++ implementation maintains hypermaps in full frames only. To access a reducer $\mathtt{x}$ while executing in a stack frame, the worker looks up the address of $\mathtt{x}$ in the hypermap of the least ancestor full frame, that is, the full frame at the head of the deque to which the stack frame belongs.

To allow for lock-free access to the hypermap of a full frame while siblings and children of the frame are terminating, each full frame stores three hypermaps, denoted by USER, RIGHT, and CHILDREN. The USER hypermap is the only one used for look-up of reducers in the user's program. The other two hypermaps are used for bookkeeping purposes. The three hypermaps per node are reminiscent of the three copies of values used in the Euler tour technique [21]. Informally, the CHILDREN hypermap contains the accumulated results of completed children frames, but to avoid races with user code that might be running concurrently, these views have not yet been reduced into the parent's USER hypermap. The RIGHT hypermap contains the accumulated values of the current frame's right siblings that have already terminated. (A "right" sibling of a frame is one that comes after the frame in the serial order of execution, and its values are therefore on the right-hand side of the $\otimes$ operator.)

When the top-level full frame is initially created, all three hypermaps are initially empty. The hypermaps are updated in the following situations:

- upon a look-up failure,
- upon a steal,
- upon a return from a call,
- upon a return from a spawn,
- at a `cilk_sync`.

We discuss each of these cases in turn.

Look-up failure. A look-up failure inserts a view containing an identity element for the reducer into the hypermap. The look-up operation returns the newly inserted identity.

Random work stealing. A random steal operation steals a full frame P and replaces it with a new full frame C in the victim. At the end of the stealing protocol, update the hypermaps as follows:

- $\text{USER}_C \leftarrow \text{USER}_P$;
- $\text{USER}_P \leftarrow \emptyset$;
- $\text{CHILDREN}_P \leftarrow \emptyset$;
- $\text{RIGHT}_P \leftarrow \emptyset$.

In addition, if a random steal operation creates new full frames, set all their hypermaps to $\emptyset$. These updates are consistent with the intended semantics of reducers, in which the child owns the view and the parent owns a new identity view.

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, because both P and C share the view stored in the map at the head of the deque to which both P and C belong.
- Otherwise, C is a full frame. We update $\text{USER}_P \leftarrow \text{USER}_C$, which transfers ownership of child views to the parent. The other two hypermaps of C are guaranteed to be empty and do not participate in the update.

Return from a spawn. Let C be a child frame of the parent frame P that originally spawned C, and suppose that C returns. Again we distinguish the "fast path" when C is a stack frame from the "slow path" when C is a full frame:

- If C is a stack frame, do nothing, which correctly implements the intended semantics of reducers, as can be seen as follows. Because C is a stack frame, P has not been stolen since C was spawned. Thus, P's view of every reducer still contains the identity e created by the spawn. The reducer semantics allows for C's views to be reduced into P's at this point, and since we are reducing all views with an identity e, the reduction operation is trivial: all we have to do is to transfer the ownership of the views from C to P. Since both P's and C's views are stored in the map at the head of the deque to which both P and C belong, such a transfer requires no action.
- 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

accounts for the fake child. This equivalence holds because in the Cilk++ scheduler, the last spawn returning to a parent continues the execution of the parent.

[3] This steal is "provably good," because it guarantees the space and time properties of the scheduler [3].

views of C. Then, depending on whether C has a left sibling or not[4], we have two subcases:

1. If C has a left sibling L, we update $\text{RIGHT}_L \leftarrow \text{REDUCE}(\text{RIGHT}_L, \text{USER}_C)$, accumulating into the RIGHT hypermap of L.
2. Otherwise, C is the leftmost child of P, and we update $\text{CHILDREN}_P \leftarrow \text{REDUCE}(\text{CHILDREN}_P, \text{USER}_C)$, thereby storing the accumulated values of C's views into the parent, since there is no left sibling into which to reduce.

Observe that a race condition exists between C reading RIGHT_C, and the right sibling of C (if any), who might be trying to write RIGHT_C at the same time. Resolving this race condition efficiently is a matter of some delicacy, which we discuss in detail in Section 5.

Sync. A `cilk_sync` statement waits until all children have completed. When frame P executes a `cilk_sync`, one of following two cases applies:

- If P is a stack frame, do nothing. Doing nothing is correct because all children of P, if any exist, were stack frames, and thus they transferred ownership of their views to P when they completed. Thus, no outstanding child views exist that must be reduced into P's.
- If P is a full frame, then after P passes the `cilk_sync` statement but before executing any client code, we perform the update $\text{USER}_P \leftarrow \text{REDUCE}(\text{CHILDREN}_P, \text{USER}_P)$. This update reduces all reducers of completed children into the parent.

Optimizations

To access a reducer `x`, the worker performs the associative look-up described above. The overhead of this operation is comparable to that of a few function calls, and thus it is desirable to optimize it. The following paragraphs describe some optimizations.

Common subexpression elimination. The semantics of reducers ensures that two references to a reducer `x` return the same view as long as the intervening program code does not contain a `cilk_spawn` or `cilk_sync` statement or cross iterations of a `cilk_for` loop. The intervening program code may call a function that spawns, however, because the operational semantics of reducers guarantee that the view before the function call is the same as the view after the function call. In these situations, the Cilk++ compiler emits code to perform the associative look-up only once per fragment, reducing the overhead of accessing a reducer to a single indirect reference. This optimization is a type of common-subexpression elimination [1, p. 633] routinely employed by compilers. It is especially effective at minimizing the reducer overhead in `cilk_for` loops.

Dynamic caching of look-ups. The result of an associative look-up can be cached in the reducer object itself. In this optimization, each reducer object provides an array A of P pointers to views, where P is the maximum number of workers in the system. All such pointers are initially null. When accessing a reducer `x`, worker w first reads the pointer $\texttt{x}.A[w]$. If the pointer is not null, then the worker can use the pointer to access the view. Otherwise, the worker looks up the address of `x` in the appropriate hypermap and caches the result of the look-up into $\texttt{x}.A[w]$. When the hypermap of a worker changes, for example, because the worker steals a different frame, the pointers cached by that worker are invalidated. This optimization, which we have not yet implemented, can reduce the cost of a look-up to the cost of checking whether a pointer is null.

[4]Recall that a full frame stores pointers to its left and right siblings, and so this information is available in $O(1)$ time.

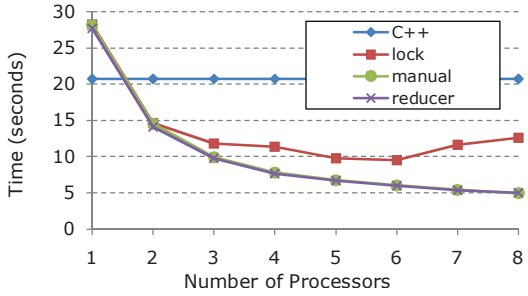

Figure 12: Benchmark results for a code that detects collisions in mechanical assemblies.

Static storage class. When the reducer has the static C++ storage class, the associative look-up can be avoided entirely. Since the address of the reducer is known at link-time, we can allocate a static global array of size P, where P is the maximum number of workers, to store views of the reducer. The views are as in the dynamic-caching optimization, but when a worker looks up a view, it simply indexes the array with its unique worker ID. An alternative is to allocate the views of a reducer at a common fixed location in worker-local storage. We have not yet implemented either of these optimizations.

Loop variables. When a loop contains several reducers allocated at the same level of nesting outside the loop, the compiler can aggregate the reducers into a single data structure, and only one associative look-up need be done for the entire data structure, rather than one for each reducer. This scheme works, because the knowledge of how the compiler packs the reducers into the fields of the data structure outside the loop is visible to the compiler when processing reducer accesses inside the loop. The Cilk++ compiler does not currently implement this optimization.

Performance

Figure 12 compares the reducer strategy with locking and with manually rewriting the code to pass the nonlocal variable as a parameter. The benchmark is a collision-detection calculation for mechanical assemblies, such as motivated the example in Figure 1, although nodes in the tree may have arbitrary degree. As can be seen from the Figure 12, all three methods incur some overhead on 1 processor. The locking solution bottoms out due to contention, which gets worse as the number of processors increases. The reducer solution achieves almost exactly the same performance as the manual method, but without drastic code rewriting.

5 ANALYSIS OF WORK-STEALING WITH REDUCERS

When a spawned child completes and is a full frame, we provably-good-steal its parent and reduce the view of the child into the view of either its parent or its left sibling. To do so atomically, the runtime system must acquire locks on two frames. In this section, we describe the locking methodology in detail and show that the Cilk++ work-stealing scheduler incurs no unusual overhead for waiting on locks.

Recall from Section 4 that when full frame F returns from a spawn, the Cilk++ runtime system accumulates the reducer map of F into another node $F.p$, which is either the left sibling of F in the spawn tree, or the parent of F in the spawn tree if no such sibling exists[5]. The relation $F.p$ can be viewed as defining a binary

[5]$F.p$ is undefined if F is the root of the spawn tree, which is never reduced into any other node.

tree in which $F.p$ is the parent of F, and in fact such a ***steal tree*** is just the left-child, right-sibling representation [6, p. 246] of the spawn tree. After accumulating the reducers of F into $F.p$, the runtime system ***eliminates*** F by splicing it out of the steal tree, in a process similar to tree contraction [16].

When F completes, it has no children in the spawn tree, and thus it has at most one child in the steal tree (its right sibling in the spawn tree). Thus, the situation never occurs that a node is eliminated that has two children in the steal tree.

Every successful steal causes one elimination, which occurs when node F completes and its reducers are combined with those of $F.p$. To perform the elimination atomically, the runtime system engages in a locking protocol. The steal tree is doubly linked, with each node F containing $F.p$, $F.lchild$, and $F.rsib$. In addition, these fields each have associated locks *pL*, *lchildL*, and *rsibL*, respectively. The protocol maintains the invariant that to change either of the cross-linked pointers between two adjacent nodes in the tree, both locks must be held. Thus, to eliminate a node F, which is a right sibling of its parent and having one child — without loss of generality, $F.lchild$ — the locks $F.pL$, $F.p.rsibL$, $F.lchildL$, and $F.rsib.pL$ must all be held, after which the F can be spliced out by setting $F.p.rsib = F.lchild$ and $F.lchild.p = F.p$. Before the elimination, F's hypermap is reduced into $F.p$'s.

The four locks correspond to two pairs of acquisitions, each of which ***abstractly locks*** a single edge between two nodes in the steal tree. That is, to abstractly lock an edge, the two locks at either end of the edge must be acquired. To avoid deadlock, the locking protocol operates as follows.

To abstractly lock the edge $(F, F.p)$, do the following:

1. ACQUIRE$(F.pL)$.
2. If F is a right sibling of $F.p$, then ACQUIRE$(F.p.rsibL)$, else ACQUIRE$(F.p.lchildL)$.

To abstractly lock the edge $(F, F.rsib)$, do the following:

1. ACQUIRE$(F.rsibL)$.
2. If $\neg$TRY-ACQUIRE$(F.rsib.pL)$, then RELEASE$(F.rsibL)$ and go to step 1.

The TRY-ACQUIRE function attempts to grab a lock and reports whether it is successful without blocking if the lock is already held by another worker. To abstractly lock the edge $(F, F.lchild)$, the code follows that of $(F, F.rsib)$.

This protocol avoids deadlock, because a worker never holds a lock in the steal tree while waiting for a lock residing lower in the tree. Moreover, if two workers contend for an abstract lock on the same edge, one of the two is guaranteed to obtain the lock in constant time. Finally, the protocol is correct, circumventing the problem that might occur if abstractly locking $(F, F.rsib)$ were implemented by ACQUIRE$(F.rsib.pL)$ followed by ACQUIRE$(F.rsibL)$, where the node $F.rsib$ could be spliced out and deallocated by another worker after F's worker follows the pointer but before it can acquire the *pL* lock in the node.

The remainder of the protocol focuses on abstractly locking the two edges incident on F so that F can be spliced out. Perhaps surprisingly, the two abstract locks can be acquired in an arbitrary order without causing deadlock. To see why, imagine each node F as containing an arrow oriented from the edge in the steal tree that F's worker abstractly locks first to the edge it abstractly locks second. Because these arrows lie within the steal tree, they cannot form a cycle and therefore deadlock cannot occur. Nevertheless, while an arbitrary locking policy does not deadlock, some policies may lead to a long chain of nodes waiting on each other's abstract locks. For example, if we always abstractly lock the edge to the parent first, it could happen that all the nodes in a long chain up the tree all need to be eliminated at the same time, and they all grab the abstract locks on the edges to their parents, thereby creating a chain of nodes, each waiting for the abstract lock on the edge to its child. In this case, the time to complete all eliminations could be proportional to the height of the steal tree.

Our strategy to avoid these long delay chains is to acquire the two abstract locks in random order: with probability $1/2$, a node F abstractly locks the edge to its parent followed by the edge to its child, and with probability $1/2$ the other way around. This online randomization strategy is reminiscent of the offline strategy analyzed in [9] for locking in static graphs. We now prove that a system that implements this policy does not spend too much time waiting for locks.

Lemma 1. *If abstract locks are acquired in random order by the P processors and the* `reduce()` *function takes $\Theta(1)$ time to compute, then the expected time spent in elimination operations is $O(M)$ and with probability at least $1-\varepsilon$, at most $O(M + \lg P + \lg(1/\varepsilon))$ time is spent in elimination operations, where M is the number of successful steals during the computation.*

Proof. Assuming that the `reduce()` function takes $\Theta(1)$ time to compute, the two abstract locks are held for $\Theta(1)$ time. Since only two nodes can compete for any given lock simultaneously, and assuming linear waiting on locks [4], the total amount of time nodes spend waiting for nodes holding two abstract locks is at most proportional to the number M of successful steals. Thus, we only need to analyze the time waiting for nodes that are holding only one abstract lock and that are waiting for their second abstract lock.

Consider the eliminations performed by a given worker, and assume that the worker performed m steals, and hence m eliminations and $2m$ abstract lock acquisitions. Let us examine the steal tree at the time of the ith abstract lock acquisition by the worker on node F. Every other node G in the tree that has not yet been eliminated creates an arrow within the node , oriented in the direction from the first edge it abstractly locks to the second. These edges create directed paths in the tree. The delay for the worker's ith lock acquisition can be at most the length of such a directed path starting at the edge the worker is abstractly locking. Since the orientation of lock acquisition along this path is fixed, and each pair of acquisitions is correctly oriented with probability $1/2$, the waiting time for F acquiring one of its locks can be bounded by a geometric distribution:

$$\Pr\{\text{the worker waits for} \geq k \text{ eliminations}\} \leq 2^{-k-1} .$$

We shall compute a bound on the total time Δ for all $2m$ abstract lock acquisitions by the given worker. Notice that the time for the ith abstract lock acquisition by the worker is independent of the time for the jth abstract lock acquisition by the same worker for $i < j$, because the worker cannot wait twice for the same elimination. Thus, the probability that the $2m$ acquisitions take time longer than Δ eliminations is at most

$$\begin{aligned} \binom{\Delta}{2m} 2^{-\Delta} &\leq \left(\frac{e\Delta}{2m}\right)^{2m} 2^{-\Delta} \\ &\leq \varepsilon'/P \end{aligned}$$

by choosing $\Delta = c(m + \lg(1/\varepsilon'))$ for a sufficiently large constant $c > 1$. The expectation bound follows directly.

Since there are at most P workers, the time for all the abstract lock acquisitions is $O(M + \lg P + \lg(1/\varepsilon))$ with probability at least $1-\varepsilon$ (letting $\varepsilon' = \varepsilon/P$), where M is the total number of successful steals during the computation. □

This analysis allows us to prove the following theorem.

Theorem 2. *Consider the execution of any Cilk++ computation with work T_1 and span T_∞ on a parallel computer with P processors, and assume that the computation uses a reducer whose*

```
1  holder<T> global_variable;
2  // originally: T global_variable
3
4  void proc1() {
5    cilk_for (i = 0; i < N; ++i) { //was: for
6      global_variable = f(i);
7      proc2();
8    }
9  }
10
11 void proc2() { proc3(); }
12 void proc3() { proc4(); }
13
14 void proc4() {
15   use(global_variable);
16 }
```

Figure 13: An illustration of the use of a holder. Although Cilk++ does not yet support this syntax for holders, programmers can access the functionality, since holders are a special case of reducers.

`reduce()` *function takes* $\Theta(1)$ *time to compute. Then, the expected running time, including time for locking to perform reductions, is* $T_1/P + O(T_\infty)$. *Moreover, for any* $\varepsilon > 0$, *with probability at least* $1-\varepsilon$, *the execution time on* P *processors is at most* $T_P \leq T_1/P + O(T_\infty + \lg P + \lg(1/\varepsilon))$.

Proof. The proof closely follows the accounting argument in [3], except with an additional bucket to handle the situation where a worker (processor) is waiting to acquire an abstract lock. Each bucket corresponds to a type of task that a worker can be doing during a step of the algorithm. For each time step, each worker places one dollar in exactly one bucket. If the execution takes time T_P, then at the end the total number of dollars in all of the buckets is PT_P. Thus, if we sum up all the dollars in all the buckets and divide by P, we obtain the running time. In this case, by Lemma 1, the waiting-for-lock bucket has size proportional to the number of successful steals, which is PT_∞ and thus contributes at most a constant factor additional to the "Big Oh" in the expected running time bound $T_1/P + O(T_\infty)$ proved in [3]. Moreover, with probability at least $1-\varepsilon$, the waiting-for-lock bucket has at most $O(M + \lg P + \lg(1/\varepsilon))$ dollars, again contributing at most a constant factor to the "Big Oh" in the bound $T_1/P + O(T_\infty + \lg P + \lg(1/\varepsilon))$ proved in [3]. □

6 HOLDERS

A ***holder*** is a hyperobject that generalizes the notion of thread-local storage. In the code fragment shown in Figure 13, the global variable is used as a mechanism to pass values from `proc1` to `proc4` without passing spurious parameters to `proc2` and `proc3`. The original `for` loop in line 5 has been replaced by a `cilk_for` loop, which appears to create races on `global_variable`. Races are avoided, however, because `global_variable` is declared to be a holder in line 1. This technique avoids the need to restructure `proc2` and `proc3` to be aware of the values passed from `proc1` to `proc4`.

The implementation of holders turns out to be straightforward, because a holder is a special case of a reducer whose (associative) binary operator $\otimes$ always returns the left input element. Figure 14 shows how a holder can be defined in terms of a reducer.

7 SPLITTERS

Another type of hyperobject that appears to be useful for parallelizing legacy applications is a "splitter."[6] Consider the example code in Figure 15 which walks a binary tree and computes the maximum

[6] We have not yet implemented splitters in Cilk++.

```
1  template<class T>
2  struct void_monoid : cilk::monoid_base<T>
3  {
4    typedef T value_type;
5    void reduce(T* left, T* right) const { }
6    void identity(T* p) const { new (p) T(); }
7  };
8
9  template <class T>
10 class holder
11   : public cilk::reducer<void_monoid<T> >
12 {
13 public:
14   operator T&() { return this->view(); }
15 };
```

Figure 14: The definition of `holder` in terms of reducers.

```
1  int depth(0);
2  int max_depth(0);
3  /* ... */
4  void walk(Node *x)
5  {
6    switch (x->kind) {
7      case Node::LEAF:
8        max_depth = max(max_depth, depth);
9        break;
10     case Node::INTERNAL:
11       ++depth;
12       walk(x->left);
13       walk(x->right);
14       --depth;
15       break;
16   }
17 }
```

Figure 15: A C++ program that determines the maximum depth of a node in a binary tree using global variables.

depth `max_depth` of any leaf in the tree. The code maintains a global variable `depth` indicating the depth of the current node. It increments `depth` in line 11 before recursively visiting the children of a node and decrements `depth` in line 14 after visiting the children. Whenever the depth of a leaf exceeds the maximum depth seen so far, stored in another global variable `max_depth`, line 8 updates the maximum depth. Although this code makes use of a global variable to store the depth, the code could be rewritten to pass the incremented depth as an argument.

Unfortunately, rewriting a large application that uses global variables in this way can be tedious and error prone, and the operations can be more complex than simple increments/decrements of a global variable. In general, the kind of usage pattern involves an operation paired with its inverse operation. The paired operations might be a push/pop on a stack or a modification/restoration of a complex data structure. To implement a backtracking search, for example, one can keep a global data structure for the state of the search. Each step of the search involves modifying the data structure, and when the search backtracks, the modification is undone.

Parallelizing the code in Figure 15 at first may seem straightforward. We can spawn each of the recursive `walk()` routines in lines 12–13. The `max_depth` variable can be made into a reducer with the maximum operator. The `depth` variable is problematic, however. If nothing is done, then a determinacy race occurs, because the two spawned subcomputations both increment `depth` in parallel. Moreover, as these subcomputations themselves recursively spawn, many more races occur. What we would like is for each of the two spawned computations to treat the global variable `depth` as if it were a local variable, so that it has the same value in a parallel execution as it does in a serial execution.

A splitter hyperobject provides this functionality, allowing each subcomputation to modify its own view of `depth` without interference. Figure 16 shows how the code from Figure 15 can be parallelized by declaring the global variable `depth` to be a splitter.

Let us be precise about the semantics of splitters. Recall that a

```
1  splitter<int> depth;
2  reducer_max<int>  max_depth;
3  /* ... */
4  void walk_s(Node *x)
5  {
6    switch (x->kind) {
7      case Node::LEAF:
8        max_depth = max(max_depth, depth);
9        break;
10     case Node::INTERNAL:
11       ++depth;
12       cilk_spawn walk_s(x->left);
13       walk_s(x->right);
14       sync;
15       --depth;
16       break;
17   }
18 }
```

Figure 16: A Cilk++ program that determines the maximum depth of a node in a binary tree using a reducer and a splitter.

`cilk_spawn` statement creates two new Cilk++ strands: the child strand that is spawned, and the continuation strand that continues in the parent after the `cilk_spawn`. Upon a `cilk_spawn`:

- The child strand owns the view C owned by the parent procedure before the `cilk_spawn`.
- The continuation strand owns a new view C', initialized nondeterministically to either the value of C before the `cilk_spawn` or the value of C after the child returns from the `cilk_spawn`.

Notice that in Figure 15, the value of the `depth` is the same before and after each call to `walk()` in lines 12–13. Thus, for the corresponding parallel code in Figure 15(b), the nondeterministic second condition above is actually deterministic, because the values of `depth` before and after a `cilk_spawn` are identical. Commonly, a splitter obeys the ***splitter consistency condition***: when executed serially, the splitter value exhibits no net change from immediately before a `cilk_spawn` to immediately after the `cilk_spawn`. That is, if the spawned subcomputation changes the value of the splitter during its execution, it must restore the value one way or another before it returns.

Implementation of splitters

We now describe how to implement splitter hyperobjects. The main idea is to keep a ***hypertree*** of hypermaps. Accessing a splitter x involves a search from the hypermap associated with the executing frame up the hypertree until the value is found. Splitter hypermaps support the following two basic operations:

- HYPERMAP-INSERT(h,x,v) — insert the key-value pair (x,v) into the hypermap h.
- HYPERMAP-FIND(h,x) — look up the splitter x in the hypermap h, and return the value stored in h that is associated with x, or return NIL if the value is not found. If $h =$ NIL (the hypermap does not exist), signal an error.

The runtime data structures described in Section 4 can be be extended to support splitters. Recall that each worker owns a spawn deque *deque* implemented as an array, where each index i stores a call stack. The top and bottom of the deque are indexed by worker-local variables H and T, where array position i contains a valid pointer for $H \le i < T$. We augment each deque location to store a pointer $deque[i].h$ to a hypermap. Each worker *worker* also maintains an ***active hypermap*** *worker*.*h*. In addition, we store a ***parent*** *h*.*parent* pointer with each hypermap h, which points to the parent hypermap in the hypertree (or NIL for the root of the hypertree). Each hypermap h has two children, identified as *h*.*spawn* and *h*.*cont*.

The runtime system executes certain operations at distinguished points in the client program:

- when the user program accesses a splitter hyperobject,
- upon a `cilk_spawn`,
- upon return from a `cilk_spawn`, and
- upon a random steal.

We now describe the actions of the runtime system in these cases. Each action is executed as if it is atomic, which can be enforced through the use of a lock stored with the worker data structure.

Accessing a splitter. Accessing a splitter hyperobject x in a worker w can be accomplished by executing SPLITTER-LOOKUP $(w.h,x)$, where the SPLITTER-LOOKUP(h,x) function is implemented by the following pseudocode:

- Set $hiter = h$.
- While $(v =$ HYPERMAP-FIND$(hiter,x)) ==$ NIL:
  - Set $hiter = hiter.parent$.
- If $h \neq hiter$, then HYPERMAP-INSERT(h,x,v).

This implementation can be optimized:

- For hypermaps in the deque, rather than following *parent* pointers in the search up the hypertree, the auxiliary pointers in hypermaps can be omitted and the search can walk up the deque itself.
- After looking up a value in an ancestor hypermap, all intermediate hypermaps between the active hypermap and the hypermap where the value was found can be populated with the key-value pair.

Spawn. Let w be the worker that executes `cilk_spawn`.

- Set $parent = w.h$, and create *child* as a fresh empty hypermap.
- Set $parent.spawn = child$.
- Set $parent.cont =$ NIL.
- Set $child.parent = parent$.
- Push *parent* onto the bottom of w's deque.
- Set $w.h = child$.

Return from a spawn. Let w be the worker that executes the return statement. Let $child = w.h$, and let $parent = child.parent$. We have two cases to consider. If the deque is nonempty:

- For all keys x that are both in *child* and *parent*, update the value in *parent* to be the value in *child*.
- Destroy *child*.
- Set $w.h = parent$.

If the deque is empty:

- Destroy *child*.
- For all keys x that are in *parent* but not in *parent*.*cont*, insert the parent value into *parent*.*cont*.
- Set $w.h = parent.cont$.
- Splice *parent* out the hypertree.
- Destroy *parent*.

In either case, control resumes according to the "Return from a spawn" description in Section 4.

Random steal. Recall that on a random steal, the thief worker *thief* removes the topmost call stack from the victim *victim*'s deque *victim*.*deque* of the victim. Let *looth* be the youngest hypermap on *victim*'s deque.

- Create a fresh empty hypermap h.
- Set $h.parent = looth$.
- Set $looth.cont = h$.
- Set $thief.h = h$.

This implementation copies a view when the splitter is accessed (read or write), but it is also possible to implement a scheme which copies a view only when it is written.

8 CONCLUSION

We conclude by exploring other useful types of hyperobjects besides reducers, holders, and splitters. For all three types, the child always receives the original view at a `cilk_spawn`, but for parents, there are two cases:

COPY: The parent receives a copy of the view.

IDENTITY: The parent receives a view initialized with an identity value.

The joining of views, which can happen at any strand boundary before the `cilk_sync`, also provides two cases:

REDUCE: The child view is updated with the value of the parent view according to a reducing function, the parent view is discarded, and the parent view gets the view of the child.

IGNORE: The parent view is discarded, and the parent receives the view of the child.

Of the four combinations, three are the hyperobjects we have discussed:

(IDENTITY, IGNORE): Holders.

(IDENTITY, REDUCE): Reducers.

(COPY, IGNORE): Splitters.

The last combination, **(COPY, REDUCE)**, may also have some utility, although we have not encountered a specific need for this case in the real-world applications we have examined. We can imagine a use in calculating the span of a computation, for example, since the state variables for computing a longest path in a computation involve the behavior of both a splitter and a max-reducer.

There may also be other useful types of hyperobjects than the four produced by this taxonomy.

ACKNOWLEDGMENTS

A big thanks to the great engineering team at Cilk Arts. Thanks especially to John Carr, who contributed mightily to the implementation and optimization of reducers. Thanks also to Will Leiserson, who Cilkified and ran the benchmark collision-detection code. We thank the referees for their excellent comments. We are especially grateful to the particular referee who provided several pages of admirably prodigious and insightful remarks which significantly improved the quality of our presentation.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Guy E. Blelloch. Programming parallel algorithms. In *Proceedings of the 1992 Dartmouth Institute for Advanced Graduate Studies (DAGS) Symposium on Parallel Computation*, pages 11–18, Hanover, New Hampshire, June 1992.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[4] James E. Burns. Mutual exclusion with linear waiting using binary shared variables. *SIGACT News*, 10(2):42–47, 1978.

[5] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[7] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[9] Phuong Hoai Ha, Philippas Tsigas, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient multi-word locking using randomization. In *PODC '05: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 249–257, New York, NY, USA, 2005. ACM.

[10] Johnson M. Hart. *Windows System Programming*. Addison-Wesley, third edition, 2004.

[11] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[12] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, 1962.

[13] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Jr. Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[14] Clifford Lasser and Stephen M. Omohundro. *The Essential *Lisp Manual, Release 1, Revision 3*. Thinking Machines Technical Report 86.15, Cambridge, MA, 1986. Available from: `http://omohundro.files.wordpress.com/2009/03/omohundro86_the_essential_starlisp_manual.pdf`.

[15] Don McCrady. Avoiding contention using combinable objects. Microsoft Developer Network blog post, September 2008. Available from: `http://blogs.msdn.com/nativeconcurrency/archive/2008/09/25/avoiding-contention-using-combinable-objects.aspx`.

[16] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Symposium on the Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[17] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. Technical Report memo AI-199, Massachusetts Institute of Technology Artificial Intelligence Laboratory, June 1970.

[18] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[19] OpenMP: A proposed industry standard API for shared memory programming. OpenMP white paper, October 1997. Available from: `http://www.openmp.org/specs/mp-documents/paper/paper.ps`.

[20] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc., 2007.

[21] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.

[22] William Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973.