Mutable state in parallel programs

Matteo Frigo

June 10, 2009

The need for parallel state. Mutable state is generally regarded as problematic in the presence of parallelism.

Consider for example the imperative program in Figure 1, which is intended to be an oversimplified proxy for more complicated situations. Like many real programs, `fib()` executes an irregular control flow, occasionally updating the mutable state variable `x`. This mutable state makes the program hard to parallelize. For example, one cannot execute the two recursive calls to `fib(n-1)` and `fib(n-2)` concurrently, because otherwise a determinacy race would arise between the concurrent updates to `x`. One can resolve the determinacy race by using some form of atomic addition, but this solution serializes all updates to `x`, and therefore its completely destroys the parallelism of the program.

Although in this simple case the obvious solution is to rewrite `fib()` in a functional style, such a code restructuring is not practical in existing large programs that were never meant to execute in parallel in the first place. I claim that, in order to parallelize these programs, *the meaning of mutable state must be redefined so as to be compatible with parallelism*. I now show how one way to accomplish this goal.

Hyperobjects. One way to avoid the determinacy race on `x` is to split `x` into multiple copies that are never accessed concurrently. Well-known examples of this idea can be found in the `__thread` storage class used by `gcc` to denote thread-local storage, and in the `private` directive in OpenMP. Expanding on this concept, we define a ***hyperobject*** as an object that shows distinct ***views*** to different observers, where the views of observers running concurrently are guaranteed to be distinct (in the `eq?` sense). Hyperobject views are therefore race-free by construction. Unlike the `__thread` annotation, which is a storage class and only applies to variables, hyperobjects are first-class objects that can be allocated dynamically, stored in classes or arrays, etc. Unlike the OpenMP `private` directive, hyperobjects are not tied to the lexical contour delimited, e.g., by a `for` loop.

Figure 2 shows how `fib` can be written using hyperobjects. The program is written in the Cilk++ language, which augments C++ with fork/join parallelism expressed by means of the `cilk_spawn` and `cilk_sync` keywords. The syntactic form `x()` (the ***view operator***) returns a reference to a view of hyperobject `x`.

Reducers. More specifically, the hyperobject `x` from Figure 2 is a ***reducer***, which in addition to supporting multiple views, obeys peculiar semantic rules that guarantee that `fib()` works as intended. We first define some terminology, and then explain the rules.

Cilk Arts, Inc. matteo@cilk.com. This work was supported in part by the National Science Foundation under SBIR Grants 0712243 and 0822896.

Submitted to the Fun Ideas and Thoughts session of PLDI'09.

```
int x;

void fib(int n) {
    if (n < 2) {
        x += n;
    } else {
        fib(n-1);
        fib(n-2);
    }
}

int main() {
    x = 0;
    fib(35);
    printf("%d\n", x);
}
```

Figure 1: A sequential C/C++ imperative program that computes Fibonacci numbers.

A ***parent*** procedure executing the statement `cilk_spawn f()` forks the control flow of the program into a ***child*** `f()` and a ***continuation***, which is the portion of the parent procedure that follows the spawn in the dynamic execution. (In the example, the continuation starts with the execution of `fib(n-2)`.) A `cilk_sync` statement waits until both the child and the continuation have completed, and then joins the control flow.[1] A ***strand*** is a maximal segment of the dynamic control flow of a program that does not contain forks or joins. To ***reduce*** a right view r into a left view l means invoking procedure `reduce(&`l`, &`r`)`. An ***identity view*** is one constructed by the `identity()` procedure.

We are now ready to state the rules of reducers.

1. All invocations of the view operator in the same strand return the same (`eq?`) view. We say that the strand ***owns*** the view.
2. No two concurrent strands own the same view.
3. After a fork in the control flow, the child owns the view of the parent, and the continuation owns a fresh identity view.
4. An implementation of reducers can axiomatically assume that reducing an identity into a view does not alter the value of the view. The implementation is free not to perform such reductions at all.
5. Before completing a `cilk_sync`, the continuation's view is reduced into the child's view. After this reduction, the parent procedure owns the child's view.
6. The reduction in Rule 5 can be executed at any time before the join, provided that Rules 1 and 2 are not violated. Specifically, if the child has completed before the continuation has started, then the implementation can reduce the continuation's view into the child's view, and give the child's view to the

[1] In the actual Cilk++ language, the `cilk_sync` statement waits for multiple children, but here for simplicity we consider only binary join operations.

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

extern "C++" {
    template <typename T>
    struct add_monoid : cilk::monoid_base<T> {
        void reduce(T *left, T *right) const {
            *left += *right;
        }
        void identity(T *p) const {
            new (p) T(0);
        }
    };
}

cilk::reducer<add_monoid<int> > x;

void fib(int n) {
    if (n < 2) {
        x() += n;
    } else {
        cilk_spawn fib(n-1);
        fib(n-2);
        cilk_sync;
    }
}

int cilk_main() {
    x() = 0;
    fib(35);
    printf("%d\n", x());
}
```

Figure 2: A parallel imperative program that computes Fibonacci numbers, written in Cilk++ with hyperobjects. The `add_monoid` class is meant as an illustration of the internals of a reducer. In production programs, one would use the more complete `reducer_opadd` provided by the Cilk++ library.

continuation. Since the continuation's view is an identity before the reduction, the reduction is a no-op by Rule 4.

The reader can verify that, under these rules, `fib()` computes the correct answer no matter how often the implementation chooses to apply Rule 6. Correctness depends upon integer addition being associative and having identity 0, but commutativity is not required. (One could, for example, reduce over list append and still obtain a result in the correct order.)

Reducers can be implemented efficiently within a work-stealing scheduler [1]. An open source implementation is available at `www.cilk.com`.

Uses and abuses of reducers.

We initially thought of reducers as a way to compute parallel reductions over arbitrary data structures while minimizing code restructuring, but experience has shown that reducers are more versatile than we expected. We now discuss some of the surprising uses of reducers.

Multiple reductions in parallel. While one may think that the result of the reduction is only valid "at the end" of the computation (whatever that means), the value of a view is always constrained by the reducer rules, and the value can be deterministic in certain cases. For example, one could spawn multiple parallel instances of `cilk_main()` in Figure 2, in which case the reducer rules imply that each instance would accumulate the correct result, even though they all share the same hyperobject! Thus, it is not correct to think of reducers as some strange kind of atomic object that is magically implemented without mutual exclusion.

Exceptions as reducers. When throwing an exception, Cilk++ stores the exception into a reducer. The reduce operation propagates the leftmost exception (the one that would be propagated in a serial execution).

Backward compatibility. Global variables can be retroactively reinterpreted as special cases of reducers for programs executing on one processor. In this case, no actual concurrency exists, Rule 6 always applies, and therefore all strands in the program share the same (`eq?`) view, which is the "global variable." Since only one view is ever created, the reduce operation and the identity view are irrelevant and need not even exist.

Generalization of thread-local storage. The standard thread-local storage machinery is not too useful in environments where a scheduler moves strands across threads. Reducers (with no-op reduction) subsume the TLS concept in a useful way. For example, Cilk Arts has successfully used reducers to implement a mostly lock-free region-based memory allocator, storing regions in reducers.

Reducers for file I/O. Cilk Arts has parallelized the bzip2 file compressor by spawning the compression of independent blocks, and using a reducer to write to the output file in the correct order. The implementation is considerably simpler than equivalent producer/consumer organizations with a reorder buffer. The "leftmost" view is flushed to disk as early as possible. (The leftmost view is the one owned by the earliest strand in serial order. To know if a view is the leftmost, reduce over OR with identity FALSE. Set the view to TRUE at the beginning of the program.)

Counting the number of views. One can write intentionally nondeterministic reducers where the nondeterministic value of a view reveals properties of the execution. The simplest example is to count the number of reducer views created, which can be interpreted as a reduction over `+` with "identity" `1`.

Open problems.

Although hyperobjects have proven really useful in practice (they are used in all non-toy Cilk++ programs that we know of), our current understanding of hyperobjects is limited. I conclude by identifying some of the remaining problems.

Other kinds of hyperobjects. When forking a reducer view, the child gets the view and the continuation gets an identity view. Another possibility would be to give the continuation a *copy* of the child's view instead of an identity. This kind of hyperobject (a ***splitter***) appears to be useful for backtrack search, but we don't know how to implement it efficiently. Another possibility is to split the view into two (roughly equal?) parts whose "sum" is the original view. This pattern appears to be useful for parallel iteration over certain data structures.

Reification of `reduce()`. You may wonder why the `add_monoid` class in Figure 2 is declared as `extern "C++"`. The reason is that the `reduce()` function is in effect part of the implementation of Cilk++, and it currently cannot be written in Cilk++ itself. In particular, it cannot spawn and it cannot use hyperobjects. Giving a proper meaning to hyperobjects in `reduce()` appears to be a tricky problem that is currently unsolved. For example, it is unclear in which order views should be reduced that are created by `reduce()`, and how to guarantee termination of the reduction process.

Syntax of reducers. Calling the view operator `x()` every time is annoying for users. One could imagine compiler support that allows the user to write `x` instead of `x()`, in which case we would need to define some new syntax to denote the hyperobject itself rather than a view. This would be a deep change that alters the normal meaning of self-evaluating symbols.

References

[1] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '09)*, 2009. To appear.