

# Dynamically Deterministic Data Parallelism

David F. Bacon  
IBM T.J. Watson Research Center

Hardware-supported transactional programming constructs are being hailed as the solution to the nightmare of parallelism which is being visited upon us in the form of ever more processor cores.

However, transactional memory will not solve the fundamental problem created by parallelism: non-determinism. Transactions simply reduce the number of potential interleavings, just as sequentially consistent memory reduces potential interleavings relative to weak memory models.

My experience is that even sophisticated programmers have trouble programming in non-deterministic model; how much more so the mass of programmers which will now have to confront large-scale parallelism?

In my experience, the only parallel programming models that are sufficiently simple to be widely usable are coarse-grained “embarrassing” parallelism and SIMD-style parallelism a la Connection Machine Fortran. However, most data-parallel constructs have suffered either from excessive static restrictions (disallowing potentially racy executions), excessive slow-downs (sequential execution when the absence of races can not be proved by the compiler), or unchecked and unreported data races.

The discontinuous jump in complexity comes with the introduction of non-determinism. I advocate a parallel programming construct which I call *dynamic deterministic data parallelism* (DDDP). DDDP shares with transactions the use of transparent rollback, but is otherwise quite different, and I argue, much simpler to program.

DDDP has the following fundamental features and properties:

**Fork/Join Parallelism.** All parallelism is introduced by fork/join constructs (either **do**-like constructs that specify explicit groups of parallel statements, or **do all**-like constructs. A group of simultaneously spawned threads is called a *cohort*.

**Latching.** Data is *latched* at a fork-point: each thread sees the data values from the fork point; modifications by concurrent threads are not visible. Thus there are no read/write conflicts.

**Mutual Exclusion.** The threads of a cohort must update mutually exclusive data.

**Reconciliation.** At a join point, the modifications from the threads are *reconciled*, that is, applied atomically to the state as it existed at the fork point.

**Race Freedom.** A race condition is an error. If two threads in a cohort try to update the same datum (a write/write conflict), all computation up to their least common parent cohort is aborted and an exception is thrown.

**Predictable Parallelism.** Because race conditions are checked at run-time (although a good compiler may of course rule them out via static analysis), the programmer’s parallel constructs will always be executed in parallel by the run-time system, leading to predictable performance. The use of a fork/join by the programmer is thus simultaneously a demand for parallelism and an *assertion* of race-freedom.

The compiler and run-time system undertake the obligation to check that assertion.

**Determinism.** Since a potential data race causes a rollback, there is no non-deterministic side-effect of the race. The information in the exception must also be deterministic (that is, either complete information about all conflicts regardless of order, or no information about the source of the conflicts), subsequent execution can not observe the non-determinism of the interleaving. Furthermore, an attempt to re-execute the same construct will deterministically lead to the same error.

**Granularity, not Synchronization.** The programmer specifies only the granularity at which invariants are maintained. All synchronization is implicit and simply prevents concurrent modifications to these granules. There are no programmer-specified locks, transactions, or barriers. The default granularity is an object reference or primitive variable.

**Abstraction and Compositionality.** The parallelism of a sub-computation is not exposed to external threads. The use of parallelism within a computation is purely a local implementation decision and is not visible externally except in terms of performance. Thus the use of parallelism is compositional.

**Symmetry and Nesting.** Fork/join constructs may be nested, and latching and reconciliation are symmetric.

**Location Transparency.** A program expresses computation over data. Location of the data on processors or in memories affects only performance. Data layout, when needed, is specified abstractly as partitions of the computation, not assignments to processors.

**Deterministic Reduction.** Reduction operators are provided for parallel constructs, but the order of the reduction is deterministic, by default using a left-biased tree reduction.

**Simplicity.** Side-effects are limited to the outermost (singleton) cohort. There are no compensating actions, top-actions, or other complicating features to the model.

So much for the advantages of the approach. The liabilities and unresolved issues can be summarized as follows:

**Granule Specification.** When customization of the level of granularity for reconciliation is required, how can that be done in a simple way? At the object level? With static object groupings? With dynamic constructs?

**Parallelism over Collections.** Providing parallel operations over encapsulated data structures would seem to require either some sort of generator capability or else an idiom for functions that take closure which are internally used in parallel constructs.

**Application Domain.** How many applications can be written this way? I believe many can, but obviously many can’t either. What is the right way to incorporate this into a more flexible model without giving up its desirable properties?

I welcome any suggestions on these points.