Disciplined Concurrent Programming Using Tasks with Effects

Stephen Heumann and Vikram Adve
University of Illinois at Urbana-Champaign
{heumann1,vadve}@illinois.edu

Abstract

Concurrent programming has become ubiquitous, but today's widely-used concurrent programming models provide few safety guarantees, making it easy to write code with subtle errors. Models that do give strong guarantees often can only express a relatively limited class of programs. We argue that a concurrent programming model should offer strong safety guarantees, while still providing the flexibility and performance needed to support the many ways that concurrency is used in complex, interactive applications.

To achieve this, we propose a new programming model based on *tasks with effects*. In this model, the core unit of work is a dynamically-created *task*. The key feature of our model is that each task has programmer-specified, statically-checked *effects*, and a runtime scheduler is used to ensure that two tasks are run concurrently only if their effects are non-interfering. Our model guarantees strong safety properties, including data race freedom and a form of atomicity. We describe this programming model and its properties, and propose several research questions related to it.

1 Motivation

Concurrent programming is ubiquitous. To exploit the full capabilities of the multicore processors in today's desktops and even handheld devices, parallel programming must be used. This will only become more true in the future, as processor performance gains continue to come largely from increased parallelism. Concurrency is also used for reasons other than providing parallel performance gains. In an interactive application, long-running operations should be run concurrently with user interface event processing in order to preserve responsiveness, whether or not the time-consuming operation is itself parallel. Some operations are also simply most naturally expressed in terms of concurrent constructs such as actors [4] communicating asynchronously.

Today's complex applications often combine these forms of concurrency. In nearly any interactive application that does time-consuming computations, it will be desirable both to parallelize those computations if possible and to run them concurrently with user interface operations. A game, for example, may have high-performance graphics and AI algorithms that are internally parallel, and may also run these operations and others such as network communications concurrently with each other and with user interface operations.

Unfortunately, concurrent programming introduces a great deal of complexity and many opportunities for errors. Subtle correctness challenges such as data races, deadlocks, atomicity violations, and the unexpected behavior of memory models [2] are a drag on productivity. One major issue is that concurrent programs can have nondeterministic behavior that varies from run to run depending on the interleaving of operations, often leading to bugs that only occasionally manifest themselves.

Today's mainstream programming models do little to address these problems. The most common approach today is to use explicit threads and locks, which are low-level, error prone, and difficult to reason about. They provide no well-defined structure for the parallel control flow and no guarantees about correctness properties.

Some systems support the more abstract notion of *tasks*, including Intel's Cilk [8] and Threading Building Blocks (TBB) [14], Apple's Grand Central Dispatch and operation queues [6], Microsoft's Task Parallel Library in .NET [18], the `ForkJoinTask` framework in Java 7 [16, 22], and the tasking operations in OpenMP 3.x [21]. These systems provide more structured parallel control and synchronization constructs, but they still do not provide checked guarantees of strong safety properties, not even data-race freedom.

A few production languages such as Erlang are based on the Actor model of concurrency [4], which does not support shared memory, and so eliminates errors such as data races and atomicity violations. These languages

do not eliminate other concurrency errors, such as deadlock and unintentional nondeterminism. Moreover, a model without shared mutable memory is not well suited for many high performance algorithms that require fine-grained updates to shared global state.

Some research systems have stronger safety properties, but the properties they provide are sometimes still limited, and often come at a cost in expressivity and performance. For example, SharC [5] uses a system of type annotations along with static and dynamic checks to provide a guarantee of race freedom, but it does not provide structured concurrency constructs, and it cannot provide guarantees of stronger properties such as determinism.

Systems like Kendo [20] and CodeDet [7] use execution-level techniques to provide a deterministic execution order for a particular program and input, but the deterministic execution order does not relate in an obvious way to the structure of the program code. Small changes to the code or to the input may change the deterministic execution order in ways that the programmer cannot easily predict, which limits the value of the determinism property as a means of simplifying reasoning about the program. Moreover, these models have significant performance overheads, and they cannot exploit any performance gains possible by allowing portions of a program to execute non-deterministically.

A number of proposed models for safe parallel programming include mechanisms to execute code speculatively and roll back execution if two pieces of code executing concurrently perform conflicting accesses. A wide variety of systems use such mechanisms, including software transaction memory systems [12], Galois [15], and Aida [17]. These systems can offer appealing programming models, but there is generally a cost in performance, both from the need to log and check information about memory accesses, and from the possibility of discarded work due to rollbacks. Moreover, transactional memory systems do not provide any particular structure for the concurrency in a program, while Galois and Aida impose parallelism structures that are not suitable for general, event-driven concurrent programming.

When designing a concurrent programming model, a fundamental question is what safety guarantees it should provide. The Deterministic Parallel Java (DPJ) language [10] provided a strong set of guarantees that greatly simplify reasoning about parallel programs written in it, which can combine deterministic and nondeterministic algorithms. DPJ guaranteed four properties: (a) data race freedom; (b) strong atomicity [1]; (c) deadlock freedom; (d) deterministic semantics with full sequential equivalence for parallel computations that do not explicitly use nondeterministic parallel constructs. These guarantees are stronger than any other parallel programming model we know of that supports both deterministic and non-deterministic parallelism. *We believe these properties are appropriate for future applications that combine interactive and compute-intensive components.*

Although DPJ provides strong guarantees, it is too restrictive to express the various forms of concurrency used in complex interactive applications. Most critically, DPJ is restricted to a fork-join parallelism structure, which is not suited to the general, event-driven form of concurrency required by the interactive aspects of these applications. DPJ may be able to express some individual parallel computations within these programs, but it faces two restrictions here as well. First, it can only express fork-join parallelism, and excludes cases like pipelined computations or algorithms with more general task graphs [3]. Second, DPJ relies on a purely static type system to enforce its correctness requirements, and many algorithms cannot be checked with such a system, e.g. graph-based algorithms. Also, DPJ's support for nondeterministic computations relies on a software transactional memory system, which performs poorly due to the cost of logging and rollbacks.

We seek to define a new programming model that can provide most or all of the guarantees provided by DPJ, but with the flexibility and expressiveness to support a wide variety of concurrent programs including interactive applications, and without the performance problems of speculation and rollback. To satisfy these goals, we propose a new programming model based on *tasks with effects*. This model uses tasks similar to those of other task-based systems, but requires the programmer to declare the *effects* of each task. The run-time system then schedules tasks so as to ensure that only tasks with non-interfering effects can run concurrently. In combination with a static phase that ensures the declared effects of each task are sound, this can provide a guarantee that the program is data-race free. It also guarantees that certain portions of tasks behave atomically and that our effect system does not give rise to deadlocks. Finally, our model supports (but is not limited to) deterministic algorithms, and can provide a static guarantee of determinism for many algorithms when requested by the programmer.

2 Example

Figure 1 gives an example of how our task system might be used in an image editing program. It shows a class `Image` representing an image, with the pixel values held in two arrays, `topHalf` and `bottomHalf`. We would like to support operations in parallel on these two halves of the image. (We adopt this arrangement for simplicity. In a more realistic code, it would be possible to use finer-grained parallelism.) We also want to support a variety of operations to read and manipulate the image, which may be invoked as asynchronous tasks. This is

```
1  class Image {
2      region Top, Bottom;
3      final int[]<Top> topHalf;        // pixel values
4      final int[]<Bottom> bottomHalf;
5      ...
6      @Task void increaseContrast() writes Top, Bottom {
7          SpawnedTaskFuture<Void, writes Top> f =
8              increasePixelContrast.spawn(topHalf);
9          increasePixelContrast(bottomHalf);
10         f.join();
11     }
12
13     @Task private <region R> void
14     increasePixelContrast(int[]<R> pixels) writes R {
15         // modify values in pixels array
16     }
17 }
```

Figure 1: Example computation.

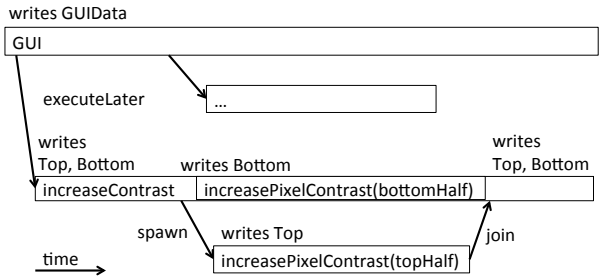

Figure 2: Tasks in example computation.

useful, for example, when the user directs the program to perform a lengthy operation that should not block the user interface thread until it completes.

We show the task `increaseContrast` (lines 6–11), which can be executed to increase the contrast of the image. It relies on the separate method `increasePixelContrast` (lines 13–16) to actually update the pixel values in each array. This enables the `increaseContrast` operation to work on the top and bottom halves of the image in parallel, by spawning a child task to work on the top half while the parent task works on the bottom half.

Figure 2 shows the tasks created in this computation. The GUI task executes the `increaseContrast` task in response to user input. That task in turn spawns off another child task so that the two halves of the image can be processed in parallel, and then joins that child task after it completes. Meanwhile, the GUI task continues running and might execute additional tasks in response to user input events. If those tasks have effects that do not interfere with the `increaseContrast` computation, they may run concurrently with it.

We will use this computation as a running example, showing how the mechanisms in our system allow the concurrency in the computation to be exploited while guaranteeing strong safety properties including task isolation and data race freedom. Note that the event-driven form of concurrency used here cannot be expressed with DPJ, which supports only fork-join parallelism structures.

3 Basic Programming Model

The core idea behind the tasks-with-effects approach is to use compile-time checking to enforce the *intra-thread* property that an effect summary is a superset of the run-time reads and writes of a task, and run-time checking to enforce the *inter-thread* property that no two tasks with conflicting effects can execute concurrently. This allows for flexible concurrency such as non-fork-join constructs and arbitrary sharing patterns, because the system can check for conflicts at run time instead of compile time.

In our programming model, a program execution consists entirely of a set of tasks. A program is launched by creating an initial task, and further tasks may be created as it executes. Each task can optionally take arguments and return a value at its completion, like a future. In our implementation, we use the `executeLater` operation to create a task, and the `getValue` operation to await the completion of a task and get the value it returned, if any.

Effects are used to control the scheduling of tasks. Each task has an effect specification, which is checked at compile time to ensure that it accurately (conservatively) reflects the task's memory accesses. These effect specifications of tasks are in turn used at run time by the task scheduler, which will ensure that no two tasks with interfering effects can run concurrently.

3.1 Effects and Regions

To perform effect-based scheduling of tasks, it must be possible to characterize the effects of each task, and to check whether the effects of two different tasks interfere with each other. We focus on the effects of memory accesses, although effects could also be used to control access to other types of resources. Each effect permits either read or read/write access to a certain set of memory locations. Intuitively, two effects interfere if they could cover accesses to the same memory location and at least one of those accesses could be a write. Two tasks can only be run concurrently if their effects do not interfere, which is the core property enforced by our scheduler.

In our prototype implementation, we use the effect system originally developed for Deterministic Parallel Java (DPJ) [9], an extended version of Java that uses effect annotations. Based on those annotations, the DPJ compiler can guarantee strong safety properties for programs using fork-join parallelism constructs, as discussed in section 1. In this work, however, we use this effect system in combination with our effect-based task scheduling model, which can support a much broader range of concurrent programs.

The DPJ effect system is based on a partitioning of memory into *regions*. The programmer can declare the names of regions, and each object field and array cell is

specified to be in a particular region. Nested hierarchies of regions can be defined using *region path lists* (*RPLs*), and a wildcard * can be used in RPLs to specify effects covering a set of regions. The region system also includes region-parameterized types and a mechanism to place each element of an array in its own region.

In Figure 1, the cells of the arrays `topHalf` and `bottomHalf` are defined as being in the regions `Top` and `Bottom`, respectively. This potentially allows the two halves of the image to be modified concurrently, since their data is in different regions. (In a real program, we could put each array cell in its own region. This would allow for more fine-grained parallelism, e.g. at the level of rows in the image. It would also be possible to place the data for different `Image` objects in different regions, potentially allowing them to be updated concurrently.)

With memory partitioned into regions, the effects of any operation in the program can be specified in terms of read and write effects on regions. Our system adopts DPJ's region-based type and effect system and requires the programmer to declare the effects of each task and method, which are statically checked as in DPJ. Unlike in DPJ, however, the compiler generates code to keep track of the effects of each task at run time. This information is then used by the run-time scheduler to guarantee noninterference of effect between concurrent tasks.

In our example code, the `increaseContrast` task is declared with the effects `writes Top, Bottom`, meaning it can read and write the pixel values in both halves of the image. The `increasePixelContrast` task has a region parameter `R` corresponding to the region containing the cells of the array passed to it. Since its declared effect is `writes R`, `increasePixelContrast(topHalf)` has the effect `writes Top`.

3.2 Effect-Based Task Scheduling

The key property that our run-time task scheduler must enforce is that two tasks with interfering effects will not be run concurrently. To do this, the scheduler will have to delay the execution of tasks that are created while another task with interfering effects is already executing.

In Figure 2, the `increaseContrast` task with effects `writes Top, Bottom` is run while the GUI task with effect `writes GUIData` continues to execute. To determine whether the new task may be run concurrently with the already-executing task, the scheduler will check if these two sets of effects interfere with each other. In this case, the region `GUIData` is disjoint from `Top` and `Bottom`, so the two tasks have non-interfering effects and may be run concurrently.

If a third task is run with `executeLater` while these two tasks are executing, its effects will be checked against those of both existing tasks. Thus, another task trying to access the image data in the regions `Top` and `Bottom` would have to wait until the `increaseContrast` task is done, but a task accessing different regions might be able to run concurrently. (The `increasePixelContrast(topHalf)` task is run with the `spawn` operation, which uses effect transfer to avoid the need for these run-time checks; see section 4.1.)

Considerable variation is possible in the design of an effect-aware task scheduler. Our initial prototype implementation uses a fairly simple approach based on a single queue of tasks. In a higher-performance implementation, the effect checking could be structured around regions, so that tasks accessing entirely disjoint regions do not need to be explicitly checked against each other. It may also be helpful to have the scheduler enforce additional properties related to fairness or task ordering, in addition to the basic property of noninterference. We believe that the design of an efficient effect-based scheduler will be an important area of future work.

4 Effect Transfer

The model we have described so far envisions the effects of each task remaining unchanged while it runs. However, it can be valuable to change the effects of tasks during their lifetimes, and in particular to transfer effects from one task to another. An implementation of effect transfer should preserve the property that no two tasks have interfering effects while they are executing concurrently. Effect transfer can be used, among other things, to support a guarantee that certain computations are deterministic and to eliminate a class of deadlocks.

4.1 Effect Transfer on Task Creation and Completion

Our implementation supports two types of effect transfer. The first is useful for fork-join styles of parallelism. It is a mechanism to transfer some of the effects of a parent task to a newly-created child task, and later transfer those effects back to the parent task when the child task completes. We call these operations `spawn` and `join`. A child task created with `spawn` may run immediately, since "ownership" of its effects is transferred directly from the parent to the child task, and thus no other tasks with conflicting effects may be running concurrently.

In Figure 1, these mechanisms are used to operate in parallel on the two halves of the image. We use the `spawn` operation to create an instance of the `increasePixelContrast` task with the argument `topHalf` (line 8). This transfers the effect `writes Top` directly from the parent `increaseContrast` task to the new child task, which means the new task can be enabled for execution immediately. The

parent task also continues executing concurrently, with its remaining effect `writes Bottom`. It runs `increasePixelContrast(bottomHalf)` as a method within the same task, which is possible since it still retains the effect `writes Bottom`.

After that computation finishes, the parent task joins the future returned when the child task was spawned. This `join` operation also transfers the child task's effect `writes Top` back to the parent task. After this, both halves of the image will have been updated, so any other task that waits for the `increaseContrast` task to finish will know that the full operation is complete.

4.2 Effect Transfer when Waiting

The second kind of effect transfer we use in our system is primarily intended to prevent deadlocks. It occurs when one task uses a `getValue` or `join` operation to wait until another task completes. We allow the waiting task's effects to be transferred to the task it is waiting on, if necessary in order for that task to execute. This effect transfer is also applied recursively through a chain of waiting tasks. The effects are automatically transferred back to the original task before it resumes execution.

More specifically, there are three cases where this type of transfer occurs. One is where a running task A calls `getValue` or `join` to wait for another task B that has not yet started. In this case, A transfers its effects to B, which can allow B to execute even if B's effects would otherwise conflict with A's effects.

Another case is when a running task A waits for a task B that has already started. In this case, A's effects are transferred to B at the time that A waits on B. This transfer is not necessary for B to start, but the effects may be further transferred to other tasks that B waits on, which might require that transfer in order to be able to execute.

Finally, if a task B that is not yet started has effects transferred to it, it may in turn transfer those effects to any running tasks whose effects conflict with B's effects, since B cannot execute until those tasks complete. This is also not directly necessary to allow those tasks to execute, but the effects may be transferred on further.

Effects may be transferred through a chain of the three types of links described above, although only the first type of link can directly enable a task to begin executing when it would otherwise be unable to. If a task that has already had effects transferred to it does a waiting operation, the transferred effects as well as the waiting task's own effects will be transferred to the task it waits on.

When the effects of a task A that has already begun to execute are transferred to some other task C that has not, this reflects the fact that A has used a waiting operation (`getValue` or `join`) and will not be able to resume execution until after task C has completed, because of some chain of waiting or blocking relationships between tasks. Therefore, in trying to schedule C, the task scheduler will disregard any effect conflict between A and C, which might otherwise prevent C from running.

Note that this only has any direct effect (i.e. allowing C to run when otherwise it could not) in cases that would otherwise give rise to deadlock, since A cannot continue until C finishes, and without effect transfer C would be unable to start until A finishes. There are many possible such cases, from a task simply waiting for another task whose effects conflict with its own through other more complex patterns involving several tasks.

In using our prototype implementation to develop an application, we found several cases where this mechanism would prevent cases that would otherwise deadlock. These generally involved cases where there were multiple types of tasks that could act on a single region, corresponding to a data structure or logical component of the program. One of these tasks might simply launch and wait for another task operating on the same region, or it might launch and wait for some task operating on a different region that "calls back" to another task operating on the same region as the first task. In either of these cases, effect transfer serves to prevent a deadlock.

In addition to preventing deadlocks, this mechanism also allows for a programming paradigm similar to a synchronized or atomic block in traditional programming models. One task can launch a second task with a superset of its effects, and then use a `getValue` operation to wait for the second task. This transfers the first task's effects to the second task (allowing it to access the same regions as the first task), and leaves the second task to wait until it can acquire access to the regions covered by its other effects, which would typically correspond to a shared resource. The second task is thus similar to a synchronized block holding the lock for the shared resource in a traditional programming model, and it is also atomic if there are no tasking operations inside it.

5 Safety Properties

Our model guarantees several strong safety properties:

Data race freedom: Our system guarantees data race freedom, through a combination of statically checking effect specifications and using a dynamic scheduler to ensure that tasks with conflicting effects do not run concurrently. Each task may only access memory regions as specified by its declared effects, and this combined with our effect-based task scheduler ensures that two accesses to the same memory location cannot race with each other.

Atomicity: A task or portion of a task that does not create or wait for any other tasks behaves atomically. It has fixed effects that cover all the memory locations it can

access, and the scheduler will ensure that no other tasks performing conflicting accesses run concurrently with it, which ensures it is atomic. This atomicity property can also be extended to portions of tasks that contain task creation operations (but not waiting operations), in the sense that the semantics are equivalent to those given by creating the new tasks only at the end of the parent task or the next waiting operation in it.

Atomicity cannot generally extend across waiting operations, as our mechanism for effect transfer when waiting may allow other tasks with conflicting effects to run before the waiting operation completes. However, a deterministic computation (discussed below) effectively executes atomically, as it is semantically equivalent to a sequential execution with no task-related operations.

Deadlock avoidance: We do not provide a guarantee of deadlock freedom, but we do avoid all deadlocks related to our effect system. That is, a program will not deadlock because one task directly or indirectly waits on another task whose effects conflict with its own. In such a case, the effects of the waiting task will be transferred to the task that it waits on, allowing it to run, as discussed in section 4.2. A set of tasks may circularly wait on each other and deadlock (e.g. if A calls `getValue` to wait until B completes, and B does the same for A), but since this waiting is mediated by our runtime implementation, it is possible to detect such a deadlock and report an error, if desired. Fully guaranteeing deadlock freedom could be a valuable contribution in future work.

Optional determinism: Programs or algorithms that create and wait for tasks only using the `spawn` and `join` operations described in section 4.1 behave deterministically. We allow tasks and methods to be annotated with a `@Deterministic` annotation, which will cause the compiler to statically check this property. This feature allows the system to give a guarantee of determinism for selected algorithms, even if the full program they appear in is not deterministic. Our support for guaranteed determinism is discussed further in [13].

6 Research Questions

There are several key research questions raised by our model of tasks with effects. These include the following:

How should effects be represented in source code? Considerations in designing an effect system include ease of use for the programmer, expressiveness to support various patterns of data access and sharing, and run-time performance overheads. We have adopted the region-based effect system from DPJ in our initial prototype, but other effect systems are possible. Ownership type systems, for example, express effects in terms of objects [11, 19], which have somewhat more limited expressivity (at least in systems so far) but could perhaps be easier to use. It might also be desirable to introduce new types of effects focused on resources such as files, database records, or other shared resources.

How should effects be represented and compared at run time? As discussed in section 3.2, various approaches to task scheduling are possible. We would like to develop an effect-checking scheme based on the actual regions on which each task has effects, so that effect checking would only require checks against other tasks accessing the same regions. To design a system based on these principles, we will also have to define suitable data structures to represent effects. If we continue to use our current hierarchical region system, perhaps a tree-like structure accessed using hand-over-hand locking could be appropriate. It may also be desirable to implement fast special cases for simple, common patterns.

How can the tasks-with-effects model be extended for heterogeneous and non-shared-memory systems? Our work so far has focused on shared-memory multicore systems, but we believe our model can also be applied to other architectures. Most client systems today include both a CPU and a programmable GPU, and future systems may have even more diverse hardware. It would be desirable to program these using a unified programming model, and we believe the tasks-with-effects model can be used for a wide range of different hardware designs, including heterogeneous systems. One advantage of our model is that explicit, region-based effects provide a natural means for specifying the data movement in non-shared-memory or non-coherent systems.

7 Summary

We have described a new programming model based on tasks with effects, using statically-checked effect declarations plus a run-time task scheduler that ensures tasks with interfering effects are not run concurrently. Our effect-based scheduling approach guarantees the absence of data races and provides an atomicity property. Effect transfer between tasks is used to eliminate a class of cases that would otherwise deadlock, and also to ensure certain algorithms are deterministic. We believe our programming model is suitable for a wide range of concurrent and parallel programs, and that it offers numerous opportunities for valuable ongoing research.

Acknowledgments

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] M. Abadi et al. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp., Special Issue on Shared-Mem. Multiproc.*, pages 66–76, December 1996.

[3] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. on Comp. Systs.*, 22(1):94–136, 2004.

[4] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] Z. Anderson et al. SharC: Checking data sharing strategies for multithreaded c. *PLDI*, 2008.

[6] Apple. Concurrency Programming Guide. `http://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/`, Jan. 2011.

[7] T. Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[8] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *PPOPP*, 1995.

[9] R. L. Bocchino et al. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[10] R. L. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[11] C. Boyapati et al. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[12] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition (Synthesis Lectures on Computer Architecture)*. 2010.

[13] S. Heumann and V. Adve. Tasks with effects: A model for disciplined concurrent programming. In *WoDet*, 2012.

[14] Intel. Intel Thread Building Blocks Reference Manual. `http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf`, Aug. 2011.

[15] M. Kulkarni et al. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[16] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[17] R. Lublinerman et al. Delegated isolation. In *OOPSLA*, 2011.

[18] Microsoft. Task Parallel Library. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[19] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, 2007.

[20] M. Olszewski et al. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[21] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[22] Oracle. Java Platform, Standard Edition 7 API specification. `http://download.oracle.com/javase/7/docs/api/`.